Process Algebra with Backtracking

J.A. Bergstra [1,2]
A. Ponse [2]
J.J. van Wamel [2] ⋆

[1] Utrecht University, Department of Philosophy
Heidelberglaan 8, 3584 CS Utrecht, The Netherlands

[2] University of Amsterdam, Programming Research Group
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

E-mail: `janb@fwi.uva.nl` - `alban@fwi.uva.nl` - `vanw@fwi.uva.nl`

Abstract. An extension of process algebra for modelling processes with backtracking is introduced. This extension is semantically based on processes that transform data because, in our view, backtracking is the undoing of the effects caused by a process in some initial data-state if this process fails.
The data-states are given by a data environment, which is a structure that also defines in which data-states guards hold, and how (atomic) actions either transform these states or block and prevent subsequent processes from being executed. State operators are used to relate process terms to a given data environment.
Backtracking is axiomatised in a few phases. First guarded commands (conditionals) and a standard type of guards, expressing the enabledness of actions, are added to basic process algebra (process algebra without operators for parallelism) by involving a Boolean algebra. Then the set of actions is partitioned in order to distinguish between different types of behaviour of actions in the scope of a (binary) operator for backtracking. Also functions on actions are defined that change the 'type' of an action. Next an axiom system for modelling processes with backtracking is presented, and it is proved that backtracking is associative, provided that some semantic constraints are satisfied. Finally a method for recursively specifying processes is defined and an example of a recursively defined process with backtracking is provided.
An operational semantics is defined relative to the Boolean algebra, describing transitions between process terms labelled with 'guarded actions'. The operational semantics is studied modulo strong bisimulation equivalence.

Key words & Phrases: process algebra, guarded commands, backtracking.
1987 CR Categories: F.1.2, F.2.2, F.3.2, I.2.8.

Note: To appear in the proceedings of the REX 1993 School/Symposium, Noordwijkerhout, The Netherlands. This report is a minor revision of Report P9106: *inertness* and *traces* are defined slightly different, and furthermore, proofs of the Congruence Lemmas 2.3.4 and 4.3.1 are added.

⋆ Partly supported by the Foundation for Computer Science in the Netherlands (SION) and by the Netherlands Organisation for Scientific Research (NWO).

1 Introduction

In this paper we introduce an operator for modelling *backtracking* in process algebra. We regard backtracking as the undoing of *data-state* transformations caused by a process, if this process blocks. In our view backtracking is based on (atomic) actions that transform data-states in a deterministic way. The interaction of processes with data-states is independently defined in a *data environment*.

We work in the setting of BPA (Basic Process Algebra, a basic fragment from ACP [BK84, BW90]) with guarded commands or conditionals, i.e., **if - then - fi** constructs (see e.g. [Dij76, BB91, GP90b]). We assume that actions are subject to *enabledness* in an implicit way, i.e., for any action a:

$$a = enabled(a) :\rightarrow a$$

where $:\rightarrow$ denotes the guarded command operator. The alternative, i.e., assuming that actions are uniformly enabled, would shift our interest from simple actions as the most basic processes considered, to 'atomic processes' of the form

$$\phi :\rightarrow a.$$

This is because backtracking is only at stake if no subsequent action is enabled. We claim that the choice for implicit enabledness of actions simplifies notation and specifications considerably.

In our approach, backtracking is modelled by a distinct operator $\dotplus$, and the undoing of actions is modelled by syntactical insertion of 'undo actions' in the alternative to be executed upon failure. Roughly, the idea can be illustrated by the following equation:

$$(a \cdot \delta) \dotplus b = a \cdot undo(a) \cdot b$$

where the $\cdot$ represents sequential composition, a and b are actions, a is invertible and uniformly enabled, and δ is the standard process that blocks.

The operator $\dotplus$ is axiomatised in such a way that it can be eliminated from closed terms in favour of the $+$ (choice), the $\cdot$ (sequential composition) and guarded commands. The following consequences are typical for our set-up:

1. We need restrictions and additional structure on actions. For example, an action that is regarded as the inverse of some other action cannot be invertible itself.
2. The $\dotplus$ is only associative (and hence suitable for elegant reasoning) if inverse actions are uniformly enabled.

Because backtracking is a phenomenon that semantically speaking is quite involved, an *algebraic* characterisation of it may be worthwhile: the restriction to an algebraic setting enforces one to express the properties of backtracking in a relatively simple way: the triggering of backtracking and 'undo actions' are described on the syntactic level. The interaction of processes with data-states is described by state operators. These relate a separate semantic level, described by a 'data environment', to our process language. State operators are defined in [BB88] and extended to the setting with guarded commands in [BB91].

Backtracking as a useful concept in programming practice is probably most commonly known from PROLOG [Bra86, CM87]. This research has been initiated by a study of the current implementation of backtracking in the specification language PROTOCOLD (see [Jon91]), which is an executable fragment of the wide spectrum language COLD [FJ92]. Both languages are developed at Philips Research Laboratories. In PROTOCOLD, a choice operator is implemented in such a way that backtracking over failing alternatives occurs. PROTOCOLD has in common with PROLOG that backtracking is based on the undoing of bindings of logical variables, contrary to our 'transformation-based' point of view. For previous work concerning the semantics of PROTOCOLD, see e.g. [Klu91]. For more recent work see [VW93].

Another aim of this paper is that our approach leads to a useful operator that can be incorporated in specification languages that are based on process algebra, such as LOTOS [ISO87], PSF [MV90] and μCRL [GP90a, GP91]. In the context of algebraic specification and verification practice, our approach may be easily applicable to backtracking 'geared' problems, e.g., the well-known *Eight Queens Problem.* We give an example of such an application in Section 5.

The notion of 'undo actions' can be traced back to ELIËNS in [Eli92], and such actions also occur in the work of KLUSENER [Klu91]. The name *try*, which we will use to denote our (binary) backtrack operator, is due to KLINT. In [Kli82] he defines the programming construct *<try-expression>* to provide "a facility for eliminating the side-effects of the evaluation of a failing expression".

Acknowledgements. We thank Willem Jan Fokkink and Chris Verhoef for useful comments.

Table of Contents

2 $\mathrm{BPA}_{\mathrm{gce}}$, BPA **with guarded commands and enabledness**

In this section we introduce $\mathrm{BPA}_{\mathrm{gc}}$ (Basic Process Algebra with guarded commands) as the basic framework of our paper. In $\mathrm{BPA}_{\mathrm{gc}}$, the enabledness of a process can be restricted by the use of a *guarded command*: $\phi :\rightarrow p$ can only be executed if $\phi \neq \mathbf{false}$ holds in the Boolean algebra in which the guards are defined. Next we define the enabledness of atomic actions by considering Boolean algebras that contain special 'enabledness' guards, and by extending $\mathrm{BPA}_{\mathrm{gc}}$ with one axiom to $\mathrm{BPA}_{\mathrm{gce}}$.

We then define transition systems that represent the operational characteristics of process terms. Over such systems, we define bisimulation semantics for $\mathrm{BPA}_{\mathrm{gce}}$ processes.

Finally, we introduce the notion of a *data environment* and the evaluation of process terms in such a data environment: processes are considered as interacting with a set of data-states. To 'evaluate' the execution of a process in a certain initial data-state, we use the *state operator* defined by BAETEN and BERGSTRA in [BB88], and extended to the setting with guarded commands in [BB91].

2.1 Signature and axioms of $\mathrm{BPA}_{\mathrm{gc}}$

We start off with the core system BPA_δ (Basic Process Algebra with δ, see e.g. [BW90]). The signature of BPA_δ has a set of *(atomic) actions* as a parameter. Actions represent the basic activities that processes can perform, such as reading input, incrementing counters and so forth. Let A be a set of actions with typical elements $a, b, \ldots$ For each action a the signature of BPA_δ, denoted as $\Sigma(\mathrm{BPA}_\delta)$, contains an identically named constant a. The special constant δ (*inaction* or *deadlock*) represents the process that cannot perform any activity and prevents subsequent processes from being executed. We also have the binary infix operators $+$ (alternative composition) and $\cdot$ (sequential composition) available. We summarise the signature $\Sigma(\mathrm{BPA}_\delta)$ in Table 1.

constants:	a for any atomic action $a \in A$
	δ models *inaction* or *deadlock*
binary operators :	$+$ alternative composition (sum)
	$\cdot$ sequential composition (product)

Table 1. The signature $\Sigma(\mathrm{BPA}_\delta)$.

In term formation, brackets and variables from a set $V = \{x, y, z, \ldots\}$ are used. The function symbol $\cdot$ is generally left out, and brackets are omitted according to the convention that $\cdot$ binds stronger than $+$. The symbol $\equiv$ is used to denote syntactic equivalence (modulo associativity) between terms. Finally, letters $t, t', \ldots$ range over open terms and $p, q, r, \ldots$ over closed terms. Let $\mathcal{P}$ denote the set of terms over $\Sigma(\mathrm{BPA}_\delta)$.

The axioms presented in Table 2 constitute the axiom system BPA_δ. These axioms describe the basic identities between terms over $\Sigma(\mathrm{BPA}_\delta)$. The operator $+$ is commutative, associative and idempotent (A1 – A3). The operator $\cdot$ right distributes over $+$ and is associative (A4, A5). Note that left distributivity of $\cdot$ over $+$ is absent. Furthermore δ behaves as the neutral element for $+$ (A6), and absorbs subsequent terms (A7).

(A1)	$x + y = y + x$
(A2)	$x + (y + z) = (x + y) + z$
(A3)	$x + x = x$
(A4)	$(x + y)z = xz + yz$
(A5)	$(xy)z = x(yz)$
(A6)	$x + \delta = x$
(A7)	$\delta x = \delta$

Table 2. The axioms of BPA_δ.

Next we introduce $\mathrm{BPA}_{\mathrm{gc}}$, Basic Process Algebra with *guarded commands* ([Dij76, BB91]). We extend the signature $\Sigma(\mathrm{BPA}_\delta)$ to $\Sigma(\mathrm{BPA}_{\mathrm{gc}}, \mathbb{B})$ in the following way. Given a Boolean algebra $\mathbb{B}$ with Boolean expressions $\mathcal{B}$, let $\mathcal{P}^+ \supseteq \mathcal{P}$ be defined inductively by involving the guarded command construct:

$$. :\rightarrow . : \mathcal{B} \times \mathcal{P}^+ \rightarrow \mathcal{P}^+.$$

So $. :\rightarrow .$ relates the Boolean expressions $\mathcal{B}$ defined over $\mathbb{B}$ and the set $\mathcal{P}$ of process terms. An expression $\phi :\rightarrow p$ is to be read as **if** ϕ **then** p. The ϕ in this expression is often referred to as a *guard* [Dij76, GP90b]. To avoid confusion with the operators $+$ and $\cdot$ from $\Sigma(\mathrm{BPA}_\delta)$, we use $\vee, \wedge$ and $\neg$ as Boolean operator symbols. Moreover we use $\leq$ and the constants **true** and **false** in their usual meaning. For instance, $\phi \leq \psi \Leftrightarrow \phi \vee \psi = \psi$ and $\phi \leq \psi \Leftrightarrow \phi \wedge \psi = \phi$.

(GC1)	$\mathbf{true} :\rightarrow x = x$
(GC2)	$\mathbf{false} :\rightarrow x = \delta$
(GC3)	$\phi :\rightarrow (x + y) = \phi :\rightarrow x + \phi :\rightarrow y$
(GC4)	$(\phi \vee \psi) :\rightarrow x = \phi :\rightarrow x + \psi :\rightarrow x$
(GC5)	$(\phi \wedge \psi) :\rightarrow x = \phi :\rightarrow (\psi :\rightarrow x)$
(GC6)	$(\phi :\rightarrow x)y = \phi :\rightarrow (xy)$

Table 3. The axioms for *guarded commands*, where $\phi, \psi \in \mathcal{B}$.

The system $\mathrm{BPA}_{\mathrm{gc}}$ consists of the axioms A1 – A7 of BPA_δ, and the axioms GC1 – GC6, presented in Table 3, which define the guarded commands. The binding power of $:\rightarrow$ is defined less than $\cdot$ and stronger than $+$. The axioms GC1 and GC2 relate the guards **true** and **false** to process terms. The axiom GC3 states that $+$ does not change the evaluation of a guard ϕ. It does not matter whether the choice is exercised before or after the evaluation of ϕ. The axiom GC4 describes the relation between $\vee$ and $+$, and GC5 that between $\wedge$ and the guarded command construct $:\rightarrow$. The last axiom GC6 defines the relation between $\cdot$ and $:\rightarrow$. Furthermore, all Boolean identities in $\mathbb{B}$ transfer to guarded commands, e.g., $\phi :\rightarrow x = \psi :\rightarrow x$ if $\phi = \psi$ holds in $\mathbb{B}$.

The following definition of basic terms over $\Sigma(\mathrm{BPA}_{\mathrm{gc}}, \mathbb{B})$ and the Representation Lemma (2.1.2) imply that we can prove statements about closed process terms over $\mathrm{BPA}_{\mathrm{gc}}$ by structural induction, and that we have to distinguish 5 cases in such proofs.

Definition 2.1.1. We inductively define basic terms over $\Sigma(\mathrm{BPA}_{\mathrm{gc}}, \mathbb{B})$ by the following BNF grammar, where $a \in A$, $\phi \in \mathcal{B}$:

$$p ::= \delta \mid a \mid p + p \mid a \cdot p \mid \phi :\rightarrow\ p.$$

□

Note that basic terms may be provably equal, for instance $\mathrm{BPA}_{\mathrm{gc}} \vdash$ **false** $:\rightarrow a = \delta$.

Lemma 2.1.2 (*Representation*)**.** *Each closed term p over $\Sigma(\mathrm{BPA}_{\mathrm{gc}}, \mathbb{B})$ can be proved equal to a basic term.*

Proof. Follows easily by structural induction from the axioms of $\mathrm{BPA}_{\mathrm{gc}}$ and the definition of basic terms. □

2.2 $\mathrm{BPA}_{\mathrm{gce}}$, $\mathrm{BPA}_{\mathrm{gc}}$ with enabledness

Given a set A of actions, we will only consider a special type of Boolean algebras, namely those that contain expressions

$$\mathit{enabled}(a) \text{ for all } a \in A$$

because 'enabledness of actions' is a crucial notion in things to come (we come back to this point in Section 6). So *enabled* is regarded as a predicate over A. Write $\mathbb{B}(A)$ for a Boolean algebra $\mathbb{B}$ satisfying this condition.

For a suitable axiomatisation of backtracking in the setting of $\mathrm{BPA}_{\mathrm{gc}}$, it turns out that the domain of the *enabled* predicate must be extended to the set $\mathcal{P}$ of process terms.

Definition 2.2.1. The predicate $\mathit{enabled} : \mathcal{P} \rightarrow \mathcal{B}$ axiomatised in Table 4 defines whether or not a process can perform an initial action. □

(En1)	$enabled(\delta) = \mathbf{false}$
(En2)	$enabled(a \cdot x) = enabled(a)$
(En3)	$enabled(x + y) = enabled(x) \vee enabled(y)$
(En4)	$enabled(\phi :\rightarrow x) = \phi \wedge enabled(x)$

Table 4. The axioms for the predicate *enabled*, where $a \in A$, $\phi \in \mathcal{B}$.

(En5) $\neg enabled(x) :\rightarrow x = \delta$

Table 5. The axiom for not enabled processes.

For non-atomic, closed process terms p, we regard *enabled*(p) as an abbreviation for a Boolean expression over $\mathbb{B}(A)$ conform the axioms in Table 4. Note that these axioms are consistent with the axioms of $\mathrm{BPA_{gc}}$.

A process axiom En5 is needed to make it explicit that a process that is not enabled equals δ.

Definition 2.2.2. The axiom system $\mathrm{BPA_{gce}}$ is defined by extending $\mathrm{BPA_{gc}}$ with the axiom En5 defined in Table 5. □

From the axioms of $\mathrm{BPA_{gce}}$ the following identity can be derived.

Lemma 2.2.3. *Any process term x over the signature $\Sigma(\mathrm{BPA_{gce}}, \mathbb{B}(A))$ is implicitly preceded by a test on enabledness:*

$$\mathrm{BPA_{gce}} \vdash x = enabled(x) :\rightarrow x.$$

Proof.

$$\begin{aligned} x &= \mathbf{true} :\rightarrow x \\ &= enabled(x) \vee \neg enabled(x) :\rightarrow x \\ &= enabled(x) :\rightarrow x + \neg enabled(x) :\rightarrow x \\ &= enabled(x) :\rightarrow x + \delta \\ &= enabled(x) :\rightarrow x. \end{aligned}$$

□

2.3 Transition systems and bisimulation semantics

In process algebra closed process terms are often related to *labelled transition systems*, which provide an operational semantics in the style of PLOTKIN [Plo81].

Definition 2.3.1. A *labelled transition system* $\mathcal{A}$ is a tuple $\langle S_{\mathcal{A}}, L_{\mathcal{A}}, \longrightarrow_{\mathcal{A}}, s_{\mathcal{A}} \rangle$, where

- $S_{\mathcal{A}}$ is a set of *states*,
- $L_{\mathcal{A}}$ is a set of *labels*,
- $\longrightarrow_{\mathcal{A}}$ is a *transition relation*,
- $s_{\mathcal{A}} \in S_{\mathcal{A}}$ is the *initial state*.

□

We consider the closed terms over $\Sigma(\mathrm{BPA_{gce}}, \mathbb{B}(A))$ as the set of states $S_{\mathcal{A}}$; the transition system related to a process term p has initial state p. Contrary to the traditional approach in process algebra, we label transitions with expressions

$$\phi :\to a$$

with $\phi \neq \mathbf{false}$ in $\mathbb{B}(A)$ and $a \in A$. This idea is based on BAETEN and BERGSTRA [BB91]. We consider $\longrightarrow_{\mathcal{A}}$ as containing transitions

$$. \overset{\cdot}{\longrightarrow} . \subseteq S_{\mathcal{A}} \times L_{\mathcal{A}} \times S_{\mathcal{A}},$$

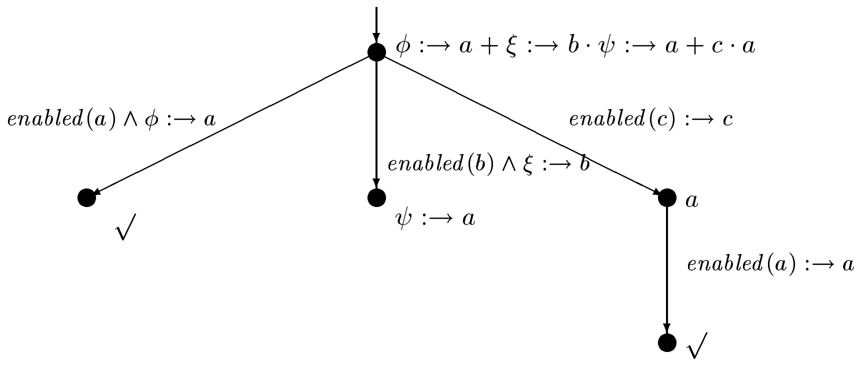
and for modelling *(successful) termination*, special transitions of the form

$$. \overset{\cdot}{\longrightarrow} \surd \subseteq S_{\mathcal{A}} \times L_{\mathcal{A}}$$

(pronounce $\surd$ as "tick"). The rules in Table 6, where in the labels $\phi :\to a$ the ϕ range over $\mathbb{B}(A)$ and the a over A, determine the transition relation $\longrightarrow_{\mathcal{A}}$ that contains exactly all derivable transitions from the closed terms over $\Sigma(\mathrm{BPA_{gce}}, \mathbb{B}(A))$.

The idea is that for $a \in A$, a transition $p \xrightarrow{\phi :\to a} p'$ expresses that by executing a, the process p can evolve into p' if $\phi \neq \mathbf{false}$ holds in $\mathbb{B}(A)$. In this case p' represents the process that remains to be executed. The transition $p \xrightarrow{\phi :\to a} \surd$ expresses that the process p can terminate successfully after executing a if $\phi \neq \mathbf{false}$ holds in $\mathbb{B}(A)$. The state δ expresses that no further activity is possible. Note that a and $enabled(a) :\to a$ always have the same transitions.

Example 2.3.2. Consider the following partially depicted transition system related to the process term $\phi :\to a + \xi :\to b \cdot \psi :\to a + c \cdot a$, where the initial state is marked with a little arrow:



$$a \in A \qquad \frac{}{a \xrightarrow{enabled(a):\rightarrow a} \surd} \text{ if } enabled(a) \neq \mathbf{false}$$

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

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

$$\cdot \qquad \frac{x \xrightarrow{\phi:\rightarrow a} x'}{x \cdot y \xrightarrow{\phi:\rightarrow a} x' \cdot y} \qquad \frac{x \xrightarrow{\phi:\rightarrow a} \surd}{x \cdot y \xrightarrow{\phi:\rightarrow a} y}$$

$$:\rightarrow \qquad \frac{x \xrightarrow{\phi:\rightarrow a} x'}{\psi :\rightarrow x \xrightarrow{\phi\wedge\psi:\rightarrow a} x'} \text{ if } \phi \wedge \psi \neq \mathbf{false} \qquad \frac{x \xrightarrow{\phi:\rightarrow a} \surd}{\psi :\rightarrow x \xrightarrow{\phi\wedge\psi:\rightarrow a} \surd} \text{ if } \phi \wedge \psi \neq \mathbf{false}$$

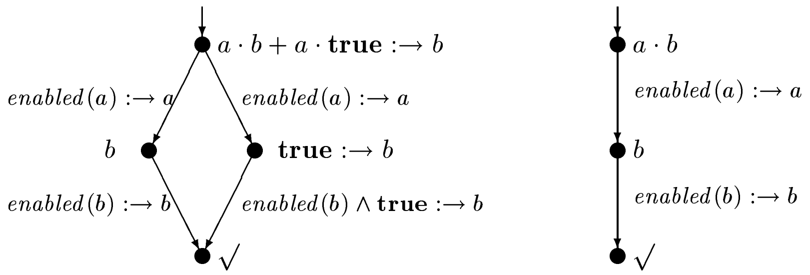
Table 6. Transition rules for $\Sigma(\mathrm{BPA_{gce}}, \mathbb{B}(A))$, where $a \in A$, $\phi \in \mathbb{B}(A)$.

The (implicit) information about the Boolean algebra $\mathbb{B}(A)$ present in this transition system tells us that apparently

$$enabled(a) \wedge \phi \neq \mathbf{false},\ enabled(b) \wedge \xi \neq \mathbf{false},$$
$$enabled(c) \neq \mathbf{false},\ enabled(a) \wedge \psi = \mathbf{false}.$$

End example.

Consider the following (partially depicted) transition systems of $a \cdot b + a \cdot \mathbf{true} :\rightarrow b$ and $a \cdot b$:



Observe that the transition system for $a \cdot b + a \cdot \mathbf{true} :\rightarrow b$ is shaped as two transition systems for $a \cdot b$. With respect to *operational* behaviour it does not matter whether the summand $a \cdot b$ or the summand $a \cdot \mathbf{true} :\rightarrow b$ is executed. Therefore we would like to consider both transition systems as equivalent. This can be achieved by identifying *bisimilar* process terms (see [Par81]). We adapt bisimilarity to the setting with "guarded labels" following the ideas of [BB91].

Definition 2.3.3. A binary relation $R \subseteq \Sigma(\mathrm{BPA_{gce}}, \mathbb{B}(A)) \times \Sigma(\mathrm{BPA_{gce}}, \mathbb{B}(A))$ is a *bisimulation* iff R satisfies for all $p, q \in \Sigma(\mathrm{BPA_{gce}}, \mathbb{B}(A))$ with pRq:

1. Whenever $p \xrightarrow{\phi:\rightarrow a} p'$ for some label $\phi :\rightarrow a$ and p', there are $\phi_1, ..., \phi_k$ and $q_1, ..., q_k$, satisfying for $i = 1, ..., k$
   - $q \xrightarrow{\phi_i:\rightarrow a} q_i$,
   - $p' R q_i$,
   - $\phi \leq \phi_1 \vee ... \vee \phi_k$ holds in $\mathbb{B}(A)$.
2. Conversely, whenever $q \xrightarrow{\phi:\rightarrow a} q'$ for some label $\phi :\rightarrow a$ and q', there are $\phi_1, ..., \phi_l$ and $p_1, ..., p_l$, satisfying for $i = 1, ..., l$
   - $p \xrightarrow{\phi_i:\rightarrow a} p_i$,
   - $p_i R q'$,
   - $\phi \leq \phi_1 \vee ... \vee \phi_l$ holds in $\mathbb{B}(A)$.
3. If $p \xrightarrow{\phi:\rightarrow a} \surd$ for some label $\phi :\rightarrow a$, there are $\phi_1, ..., \phi_m$, satisfying for $i = 1, ..., m$
   - $q \xrightarrow{\phi_i:\rightarrow a} \surd$,
   - $\phi \leq \phi_1 \vee ... \vee \phi_m$ holds in $\mathbb{B}(A)$.
4. Finally, if $q \xrightarrow{\phi:\rightarrow a} \surd$ for some label $\phi :\rightarrow a$, there are $\phi_1, ..., \phi_n$, satisfying for $i = 1, ..., n$
   - $p \xrightarrow{\phi_i:\rightarrow a} \surd$,
   - $\phi \leq \phi_1 \vee ... \vee \phi_n$ holds in $\mathbb{B}(A)$.

We call p and q *bisimilar*, notation

$$p \leftrightarroweq q,$$

iff there is a bisimulation containing the pair (p, q). □

As a consequence of the way bisimulation relates the guards in the labels, the typical guarded command axiom

$$ax + ay = ax + ay + a(\phi :\rightarrow x + \neg\phi :\rightarrow y) \quad (a \in A)$$

defined in [GP90b] does not respect our notion of bisimilarity.

Lemma 2.3.4 (*Congruence*)**.** *The relation* $\leftrightarroweq$ *between closed terms over* $\Sigma(\mathrm{BPA_{gce}}, \mathbb{B}(A))$ *is a congruence with respect to the operators of* $\Sigma(\mathrm{BPA_{gce}}, \mathbb{B}(A))$.

Proof. See Appendix. □

Moreover, it is not hard to prove that $\mathrm{BPA_{gce}}$ is a *sound* axiom system with respect to bisimulation equivalence.

Theorem 2.3.5 (*Soundness*). *Let p, q be closed terms over $\Sigma(\text{BPA}_{\text{gce}}, \mathbb{B}(A))$. If $\text{BPA}_{\text{gce}} \vdash p = q$ then $p \leftrightarrows q$.*

Proof. The relation $\leftrightarrows$ between the closed terms over the signature $\Sigma(\text{BPA}_{\text{gce}}, \mathbb{B}(A))$ is a congruence and hence respects the inference rules for equality. We have to show that all axioms are valid. As an example we prove this for GC6.

Assume that $\phi \in \mathcal{B}$ and p, q are closed process terms over $\Sigma(\text{BPA}_{\text{gce}}, \mathbb{B}(A))$. We have to show $(\phi :\rightarrow p)q \leftrightarrows \phi :\rightarrow (pq)$. We define the relation R as follows:

$$R \stackrel{\text{def}}{=} \mathit{Id} \cup \{((\phi :\rightarrow p)q, \phi :\rightarrow (pq))\}$$

where Id is the identity relation on $\Sigma(\text{BPA}_{\text{gce}}, \mathbb{B}(A))$. It follows easily that R is a bisimulation, because any outgoing transition from $(\phi :\rightarrow p)q$ has a corresponding transition from $\phi :\rightarrow (pq)$ to a state that is syntactically the same and vice versa. Hence $(\phi :\rightarrow p)qR\phi :\rightarrow (pq)$. □

2.4 State operators and data environments

We can consider processes as interacting with a set of data-states. This view can be formalised with help of the *state operator* defined in [BB88], and extended to the setting with guarded commands in [BB91]. The idea is as follows: assume a set S of *data-states* with typical elements $s, s', \ldots$. Then $\lambda_s(p)$ represents the process p in initial data-state s. The execution of actions may affect a specific data-state, so we have equations of the form

$$\lambda_s(ax) = a'\lambda_{s'}(x).$$

Here a' is the action that occurs as the result of executing a in data-state s, and s' is the data-state that results when executing a in s. The a' and s' generally depend on a and s, and are defined by the functions

$$\begin{aligned} \mathit{action} &: A \times S \rightarrow A \ \cup \ \{\delta\}, \\ \mathit{effect} &: A \times S \rightarrow S. \end{aligned}$$

In order to relate guards with the data-states in S, a third ingredient is needed for the definition of state operators: a function

$$\mathit{eval} : \mathbb{B}(A) \times S \rightarrow \mathbb{B}(A)$$

that satisfies the axioms given in Table 7. The *eval* function must respect the *action* function in the following sense: we require that the enabledness of an action $\mathit{action}(a, s)$ is equal to the enabledness of a, evaluated in s. This is captured by the axiom E3. Observe that for some uniformly enabled action a (i.e., $\mathit{enabled}(a) = \textbf{true}$), it follows from E3 and E1 that $\mathit{action}(a, s)$ is also uniformly enabled for all $s \in S$. Similarly, the function *action* must rename uniformly *dis*abled actions into uniformly *dis*abled actions or δ (recall that $\mathit{enabled}(\delta) = \textbf{false}$).

State operators are defined by the axioms in Table 8. Note that with SOG4 it follows that $\lambda_s(\delta) = \lambda_s(\textbf{false} :\rightarrow x) = \textbf{false} :\rightarrow \lambda_s(x) = \delta$. Moreover, it follows immediately that state operators can be eliminated from closed terms (cf. Lemma 2.1.2).

(E1)	$eval(\mathbf{true}, s) = \mathbf{true}$
(E2)	$eval(\mathbf{false}, s) = \mathbf{false}$
(E3)	$eval(enabled(a), s) = enabled(action(a, s))$
(E4)	$eval(\phi \vee \psi, s) = eval(\phi, s) \vee eval(\psi, s)$
(E5)	$eval(\phi \wedge \psi, s) = eval(\phi, s) \wedge eval(\psi, s)$
(E6)	$eval(\neg\phi, s) = \neg eval(\phi, s)$

Table 7. The axioms for the evaluation function *eval*, where $s \in S$, $\phi, \psi \in \mathcal{B}$.

(SOG1)	$\lambda_s(a) = action(a, s)$
(SOG2)	$\lambda_s(ax) = action(a, s) \cdot \lambda_{effect(a,s)}(x)$
(SOG3)	$\lambda_s(x + y) = \lambda_s(x) + \lambda_s(y)$
(SOG4)	$\lambda_s(\phi :\rightarrow x) = eval(\phi, s) :\rightarrow \lambda_s(x)$

Table 8. The axioms for *state operators*, where $a \in A$, $s \in S$, $\phi \in \mathcal{B}$.

Given A and $\mathbb{B}(A)$, we summarise the setting with state operators in the following definition.

Definition 2.4.1. A *data environment* $\mathcal{S}$ over a set A of actions and a Boolean algebra $\mathbb{B}(A)$ is a tuple $\langle S, action, effect, eval\rangle$, where

- S is a non-empty set of data-states,
- $action : A \times S \to A \cup \{\delta\}$,
- $effect : A \times S \to S$,
- $eval : \mathbb{B}(A) \times S \to \mathbb{B}(A)$, satisfying the axioms in Table 7.

Given any signature Σ occurring in this paper and some data environment $\mathcal{S}$, we write

$$\Sigma^{\lambda,\mathcal{S}}$$

for the signature obtained by adding all state operators λ_s to Σ. □

We give the transition rules over $\Sigma(\mathrm{BPA_{gce}}, \mathbb{B}(A))^{\lambda,\mathcal{S}}$ for state operators in Table 9. Without proof we state that the Soundness Theorem 2.3.5 can be extended to the setting with state operators. In Lemma 4.3.1 we prove a congruence result for a more general setting with state operators.

Theorem 2.4.2 (*Soundness*). *Let $\mathcal{S}$ be given and p, q be closed terms over the signature $\Sigma(\mathrm{BPA_{gce}}, \mathbb{B}(A))^{\lambda,\mathcal{S}}$. It holds that*

$$\mathrm{BPA_{gce}} + \mathrm{E1–6} + \mathrm{SOG1–4} \vdash p = q \quad \Longrightarrow \quad p \leftrightarrows q.$$

In order to reason about the possible data-state transformations that a process may induce in a certain data environment, we define a predicate that expresses *local enabledness* (we regard predicates as functions with codomain $\{\mathbf{true}, \mathbf{false}\}$).

$$\frac{x \xrightarrow{\phi :\rightarrow a} x'}{\lambda_s(x) \xrightarrow{eval(\phi,s):\rightarrow action(a,s)} \lambda_{effect(a,s)}(x')} \quad \text{if } eval(\phi \wedge enabled(a), s) \neq \mathbf{false}$$

$$\frac{x \xrightarrow{\phi :\rightarrow a} \surd}{\lambda_s(x) \xrightarrow{eval(\phi,s):\rightarrow action(a,s)} \surd} \quad \text{if } eval(\phi \wedge enabled(a), s) \neq \mathbf{false}$$

Table 9. Transition rules for *state operators*, where $a \in A$, $s \in S$, $\phi \in \mathcal{B}$.

Definition 2.4.3. Let $\mathcal{S} = \langle S, action, effect, eval \rangle$ be given. The predicate

$$enabled(a,s) \subseteq A \times S$$

is defined by

$$enabled(a,s) \stackrel{\text{def}}{=} \begin{cases} \mathbf{true} \text{ if } eval(enabled(a), s) \neq \mathbf{false} \\ \mathbf{false} \text{ otherwise.} \end{cases}$$

Likewise, $enabled(p,s)$ abbreviates $eval(enabled(p),s)$ (cf. Table 4). □

Furthermore, we define the *traces* of a process in a specific data environment.

Definition 2.4.4. Let $\sigma \in A^*$ denote a string over A, and λ the empty string. Given a data environment $\mathcal{S} = \langle S, action, effect, eval \rangle$, we extend the function *effect* to A^*:

$effect$: $A^* \times S \rightarrow S$, where $effect(\lambda, s) \stackrel{\text{def}}{=} s$, and $effect(a\sigma, s) \stackrel{\text{def}}{=} effect(\sigma, effect(a,s))$.

We define the relation $\xrightarrow{(\sigma,s)}\!\!\!\!\rightarrow$ as follows:

- $x \xrightarrow{(\lambda,s)}\!\!\!\!\rightarrow x$
- $\dfrac{x \xrightarrow{\phi:\rightarrow a} x'}{x \xrightarrow{(a,s)}\!\!\!\!\rightarrow x'}$ if $\begin{cases} eval(\phi,s) \neq \mathbf{false}, \\ enabled(a,s) = \mathbf{true} \end{cases}$ $\quad \dfrac{x \xrightarrow{\phi:\rightarrow a} \surd}{x \xrightarrow{(a,s)}\!\!\!\!\rightarrow \surd}$ if $\begin{cases} eval(\phi,s) \neq \mathbf{false}, \\ enabled(a,s) = \mathbf{true} \end{cases}$
- $\dfrac{x \xrightarrow{(a,s)}\!\!\!\!\rightarrow x' \quad x' \xrightarrow{(\sigma, effect(a,s))}\!\!\!\!\rightarrow x''}{x \xrightarrow{(a\sigma,s)}\!\!\!\!\rightarrow x''}$ $\quad \dfrac{x \xrightarrow{(a,s)}\!\!\!\!\rightarrow x' \quad x' \xrightarrow{(\sigma, effect(a,s))}\!\!\!\!\rightarrow \surd}{x \xrightarrow{(a\sigma,s)}\!\!\!\!\rightarrow \surd}$

An element $(\sigma, s) \in A^* \times S$ is called a *trace* of a process p iff either $p \xrightarrow{(\sigma,s)}\!\!\!\!\rightarrow q$ or $p \xrightarrow{(\sigma,s)}\!\!\!\!\rightarrow \surd$. The set $str(p,s)$ is defined as the set of all strings σ, such that (σ, s) is a trace of p. □

3 Requirements on backtracking

We regard backtracking as a technique for undoing data-state transformations. To support this intuition, we first study it in the setting of a fixed data environment over $\Sigma(\mathrm{BPA}_{\mathrm{gce}}, \mathbb{B}(A))$. Later, in Section 4, we will follow a reverse approach and define backtracking on an abstract, algebraic level. In order to reason about the data-state transformations a process can perform, we define a relational semantics. Then we partition the set of atomic actions and define some requirements on an operator that models backtracking.

3.1 Relational semantics for $\mathrm{BPA}_{\mathrm{gce}}$

A very first intuition of backtracking is that it can undo data-state transformations; if a process p cannot terminate successfully, then backtracking must offer the possibility of undoing the effect of p in its initial data-state. After this the option of executing p again must be discarded. Note that backtracking rather contains the undoing of data-state transformations than of actions. For this reason, it may be useful to give a semantic account of backtracking in terms of I/O relations on data-states. Such a semantic approach is relational: it relates initial data-states to final ones resulting from successful termination. Relational semantics is a central issue in, for instance, Floyd-Hoare logic ([Bak80]). First we give the relational semantics for $\mathrm{BPA}_{\mathrm{gce}}$ in some data environment.

Definition 3.1.1. Let A, $\mathbb{B}(A)$ and $\mathcal{S} = \langle S, \mathit{action}, \mathit{effect}, \mathit{eval}\rangle$ be given. We define the *relational semantics* for $\mathrm{BPA}_{\mathrm{gce}}$

$$[\![\cdot]\!] : \Sigma(\mathrm{BPA}_{\mathrm{gce}}, \mathbb{B}(A)) \to (S \to 2^S)$$

as follows ($a \in A$):

- $[\![\delta]\!](s) \stackrel{\mathrm{def}}{=} \emptyset$,
- $[\![a]\!](s) \stackrel{\mathrm{def}}{=} \begin{cases} \{\mathit{effect}(a,s)\} & \text{if } \mathit{enabled}(a,s) = \mathbf{true} \\ \emptyset & \text{otherwise,} \end{cases}$
- $[\![p+q]\!](s) \stackrel{\mathrm{def}}{=} [\![p]\!](s) \cup [\![q]\!](s)$,
- $[\![p \cdot q]\!](s) \stackrel{\mathrm{def}}{=} \{s' \mid \exists s''.s'' \in [\![p]\!](s) \wedge s' \in [\![q]\!](s'')\}$,
- $[\![\phi :\rightarrow p]\!](s) \stackrel{\mathrm{def}}{=} \begin{cases} [\![p]\!](s) & \text{if } \mathit{eval}(\phi,s) \neq \mathbf{false} \\ \emptyset & \text{otherwise.} \end{cases}$

□

So $[\![p]\!](s)$ contains the data-states that can result after successful termination of p in an initial data-state s.

Lemma 3.1.2 (*Soundness*)**.** *Provable equality in* $\mathrm{BPA}_{\mathrm{gce}}$ *preserves the relational semantics, i.e., if closed terms p, q over* $\Sigma(\mathrm{BPA}_{\mathrm{gce}}, \mathbb{B}(A))$ *are provably equal, then the relational semantics of p is equal to the relational semantics of q in any data environment $\langle S, \mathit{action}, \mathit{effect}, \mathit{eval}\rangle$:*

$$\mathrm{BPA}_{\mathrm{gce}} \vdash p = q \quad\Longrightarrow\quad \forall s \in S\,.\, [\![p]\!](s) = [\![q]\!](s).$$

Proof. All axioms of $\mathrm{BPA_{gce}}$ are valid in the relational semantics, and the congruence properties of "=" preserve the relational semantics. □

3.2 Classification of actions

In order to reason about the operational behaviour of atomic actions in the scope of an operator for backtracking, we classify atomic actions. We start off by distinguishing between actions that cause data-state transformations and actions that do not.

Definition 3.2.1. Let $a \in A$ and $\mathcal{S} = \langle S, \mathit{action}, \mathit{effect}, \mathit{eval}\rangle$ be given. An action a is called *(operationally) inert* iff

$$\forall s \in S\,.\,(\mathit{enabled}(a,s) = \mathbf{true} \implies \mathit{effect}(a,s) = s).$$

□

What is usually referred to in the literature as inert actions, forms a subset of the operationally inert actions defined above (see e.g. [BB91, BW90]). We will not make this distinction and just call any operationally inert action inert.

Because inert actions cause no data-state transformations, backtracking over inert actions must be avoided. If a process preceded by an inert action a cannot terminate, there is no effect of a that has to be undone. In order to distinguish between inert and non-inert actions, we define a predicate *Inert* over A that is exactly satisfied by the inert actions of A.

We further classify the actions for which *Inert* does not hold. Invertibility of such actions is at stake if a process cannot execute its subsequent part, so when it is deadlocked. Let $a \in A$. If there is an action $b \in A$ such that any possible data-state transformation caused by a can be undone by b, we call a *semantically invertible.* More formally:

$$a \text{ is } \mathit{semantically\ invertible} \iff \exists b \in A \forall s \in S.([\![a]\!](s) \neq \emptyset \implies [\![a{\cdot}b]\!](s) = \{s\}).$$

The action b is an *inverse* action of a in this case.

We give an algorithm to partition $\{a \in A \mid \neg \mathit{Inert}(a)\}$ into three subsets that satisfy mutually exclusive predicates:

- A predicate *Invertible*, which expresses that a semantically invertible action is 'formally' invertible,
- A predicate *Pass*, which expresses that an action is transparent w.r.t. backtracking,
- A predicate *Commit*, which expresses that an action refutes any backtrack possibility.

We assume that initially none of $\{a \in A \mid \neg \mathit{Inert}(a)\}$ satisfies one of these predicates. The algorithm runs on a set *Source* that initially equals $\{a \in A \mid \neg \mathit{Inert}(a)\}$. In Section 4 some design decisions, implicit in the algorithm, are motivated.

while *Source* is not empty
do choose an action a in *Source* and remove a from *Source*,
  if a is semantically invertible,
    then
      either define $Commit(a)$,
      or define $Invertible(a)$ and select a b in $Source \cup \{c \in A \mid Pass(c)\}$ that is an inverse action of a.
        If $b \in Source$, define $Pass(b)$ and remove b from *Source*,
    else define $Commit(a)$.
  fi
od

The *Invertible* predicate holds for those actions in $\{a \in A \mid \neg Inert(a)\}$ that are considered invertible on the syntactic level. Note that the way we select formally inverse actions suggests a deterministic notion of invertibility: we require that a single action, say b, is a uniform inverse of an invertible action a (i.e. for any initial data-state). So we require that if $[\![a]\!](s) = \{s'\}$, then b can always be performed in the data-state s' to undo the effect of a in the initial data-state s. We also have that different invertible actions can have the same inverse, and that an inverse action cannot be invertible itself.

The *Pass* predicate holds for actions that, if backtracking happens, actually undo data-state transformations caused by invertible actions. We now extend the class of actions for which *Pass* holds because once an action is executed in a certain data-state in the scope of a backtrack operator, it must not be executed again. There is no reason to repeat its data-state transformation plus its 'undoing' more than once (if nested backtracking occurs). To avoid repeated backtracking over a single action a, we make a duplicate $flag(a)$ and extend the set of actions with a copy:

$$\{flag(a) \mid a \in A \wedge Invertible(a)\}.$$

We further reason about the 'extended' set of atomic actions A_{flag}, where

$$A_{flag} \stackrel{\text{def}}{=} A \cup \{flag(a) \mid a \in A \wedge Invertible(a)\},$$

and define $Pass(flag(a))$. Consequently, the domains of the predicates are extended from A to A_{flag}.

If an action a is not formally invertible and must not be passed in the scope of an operator for backtracking, $Commit(a)$ holds, and backtracking over any process ap is impossible once a is executed.

Having classified the non-inert actions we return to the inert actions. Inert actions cause no data-state transformations, and therefore we do not regard them as invertible. We have the freedom, however, to classify inert actions as actions that, in the context of backtracking, either behave as *Pass* actions or as *Commit* actions.

We define variants of the above predicates by involving the evaluation of the *enabled* predicate.

$$\begin{array}{lcl} invertible(a,s) & \stackrel{\text{def}}{=} & enabled(a,s) \wedge Invertible(a), \\ pass(a,s) & \stackrel{\text{def}}{=} & enabled(a,s) \wedge Pass(a), \\ commit(a,s) & \stackrel{\text{def}}{=} & enabled(a,s) \wedge Commit(a). \end{array}$$

Note that adding the predicate $enabled(a,s) = \mathbf{false}$ defines a partition on $A_{flag} \times S$.

We finally extend the *pass* and *commit* predicates to strings over A_{flag}. Let σ denote a finite string over A_{flag}, and λ the empty string.

$$
\begin{aligned}
pass(\lambda, s) &\stackrel{\text{def}}{=} \mathbf{false},\\
pass(a\sigma, s) &\stackrel{\text{def}}{=} pass(a,s) \vee pass(\sigma, effect(a,s)),\\
commit(\lambda, s) &\stackrel{\text{def}}{=} \mathbf{false},\\
commit(a\sigma, s) &\stackrel{\text{def}}{=} commit(a,s) \vee commit(\sigma, effect(a,s)).
\end{aligned}
$$

3.3 Four requirements

Now we can characterise the crucial property of whether a process can give rise to backtracking by a predicate *fail*:

$$
\begin{aligned}
fail(x,s) \stackrel{\text{def}}{=} {} & \exists \sigma \in str(x,s) \exists x' \,\mathbf{.}\\
& (x \xrightarrow{(\sigma,s)}\!\!\!\!\rightarrow x' \wedge \neg enabled(x', effect(\sigma,s)) \wedge \neg commit(\sigma,s)).
\end{aligned}
$$

So $fail(x,s)$ holds if the process x can transform a data-state s according to a string σ that is not committed in s, and gets stuck. If $enabled(x,s) = \mathbf{false}$ then clearly $fail(x,s)$ holds, because *commit* does not hold for the empty string λ. As a consequence, $fail(\delta,s)$ holds by definition.

Having defined the predicate *fail*, we can formulate four requirements on an operator that models backtracking. We use the symbol

$$\dotplus$$

(pronounce *try*) for this operator. These requirements are formulated in terms of the relational semantics $[\![.]\!]$ for $\Sigma(\mathrm{BPA_{gce}}, \mathbb{B}(A_{flag}))$ and the above predicates.

At this stage we define "$[\![p \dotplus q]\!]$" only informally. The idea is that if p contains no *pass* actions, then $[\![p \dotplus q]\!]$ can be interpreted as follows: if backtracking is not triggered the relational semantics of $[\![p \dotplus q]\!]$ in s is equivalent to the the relational semantics of p in s, otherwise it is equivalent to the union of the relational semantics of p and q in s (see the Requirements I and II). If p does contain *pass* actions, we only partially define $[\![p \dotplus q]\!]$ (see the Requirements III and IV). Let $a \in A_{flag}$.

Req I. $\forall \sigma \in str(x,s) \,\mathbf{.}\, \neg pass(\sigma,s) \wedge \neg fail(x,s) \implies [\![x \dotplus y]\!](s) = [\![x]\!](s)$,
Req II. $\forall \sigma \in str(x,s) \,\mathbf{.}\, \neg pass(\sigma,s) \wedge fail(x,s) \implies [\![x \dotplus y]\!](s) = [\![x]\!](s) \cup [\![y]\!](s)$,
Req III. $pass(a,s) \implies [\![a \dotplus y]\!](s) = [\![a]\!](s)$,
Req IV. $pass(a,s) \implies [\![(a \cdot x) \dotplus y]\!](s) = [\![x \dotplus y]\!](effect(a,s))$.

Observe that Requirement II implies that

$$\neg enabled(x,s) \implies [\![x \dotplus y]\!](s) = [\![y]\!](s).$$

For reasons of simplicity, the Requirements I and II are a bit more restrictive than necessary. The premisse $\forall \sigma \in str(x,s) \,\mathbf{.}\, \neg pass(\sigma,s)$ could be replaced by a form in which only strings with non-inert *pass* actions are considered: the formal inverses and the *flag* actions.

The Requirements III and IV express the transparency of actions for which *pass* holds w.r.t. backtracking. These requirements also express the simple behaviour of inert *pass* actions in the scope of the backtracking operator $\not+$.

The requirements above only partially express the semantic properties of our backtracking operator as a result of the clause $\forall \sigma \in str(x,s) \,.\, \neg pass(\sigma, s)$ in the Requirements I and II. We give an example to illustrate the complications that occur when a process contains a mixture of actions for which *pass* holds and actions for which *pass* does not hold. Let $x \equiv (a+b)\cdot\delta$ with $pass(a,s)$ and $invertible(b,s)$. This process satisfies none of the requirements, while the desired relational semantics is obvious:

$$[\![(a+b)\cdot\delta \not+ y]\!](s) = [\![y]\!](effect(a,s)) \cup [\![y]\!](s).$$

The difficulty of formulating requirements for this general type of processes is that every non-inert *pass* action in the left argument of $\not+$ that is part of a not successfully terminating string, influences the initial state of the right argument of $\not+$.

In the sequel $\not+$ will be defined in an algebraic way. Indeed it will turn out that this operator satisfies all the requirements (see Theorem 4.3.4).

4 BPA($\not+$), Basic Process Algebra with backtracking

In this section we formalise the notions introduced in the previous section. However, we reverse our approach and start off from a partitioned set of actions, instead of a data environment. We define criteria for 'admissible' Boolean algebras and data environments: these must respect the definitions of Section 3.2.

Next the binary operator $\not+$ for backtracking is axiomatised. For this operator some fundamental properties are proved, the most important of which is associativity. This important property only holds if the Boolean algebra that defines the guards satisfies some special constraints.

4.1 Signature and axioms of BPA($\not+$)

The starting point for the axiomatisation of backtracking is formed by the signature $\Sigma(\mathrm{BPA_{gce}}, \mathbb{B}(A))$ (see Section 2). We continue by extending $\Sigma(\mathrm{BPA_{gce}}, \mathbb{B}(A))$ to a setting with A_{flag} and the predicates defined in Section 3.2. The approach in that section was based on a specific data environment in order to provide some intuition for the partitioning of A and the extension to A_{flag}. We now take a reverse approach, and assume that we have *given* a set of actions A_{flag}, partitioned by mutually exclusive predicates *Invertible*, *Pass* and *Commit*. We have also a predicate *Inert* that is a subset of *Pass* $\cup$ *Commit*. Instead of starting from the data environment based notions 'inertness', 'semantic invertibility' and 'inverse actions', we take a more abstract point of departure and define criteria on Boolean algebras and data environments that preserve the meaning of these notions.

The class of non-inert actions for which *Pass* holds is divided in formal inverses and *flag* actions (see Section 3.2). These two special types of atomic actions are studied more closely now, in order to define a setting in which backtracking can be axiomatised.

Recall that $\mathit{flag}(a)$ was introduced to indicate that the atomic action a, where $\mathit{Invertible}(a)$ holds, has been executed in the scope of a backtracking operator and has induced an inverse action. We defined $\mathit{Pass}(\mathit{flag}(a))$. We regard *flag* actions as the result of an application of a function *Flag* to elements from the set of actions $\{a \in A \mid \mathit{Invertible}(a)\}$. In order to make *Flag* a total function, its domain is extended to A_{flag}. Below, the formal definition of the *Flag* function is given.

Definition 4.1.1. The function $\mathit{Flag} : A_{flag} \to A_{flag} \cup \{\delta\}$ is defined by

$$\mathit{Flag}(a) \stackrel{\text{def}}{=} \begin{cases} \mathit{flag}(a) & \text{if } \mathit{Invertible}(a) \text{ holds} \\ \delta & \text{otherwise.} \end{cases}$$

□

For an action $a \in \{b \in A \mid \mathit{Invertible}(b)\}$ its formal inverse is written as $\mathit{undo}(a)$. From the partitioning algorithm it can be seen that an action $b \neq a$ can exist with $\mathit{undo}(b) = \mathit{undo}(a)$. We defined $\mathit{Pass}(\mathit{undo}(a))$. The *undo* actions are now regarded as the result of the application of a function *Undo* to elements from the set of actions $\{a \in A \mid \mathit{Invertible}(a)\}$. The domain of the *Undo* function is extended to A_{flag} as follows:

Definition 4.1.2. The function $\mathit{Undo} : A_{flag} \to A_{flag} \cup \{\delta\}$ is defined by

$$\mathit{Undo}(a) \stackrel{\text{def}}{=} \begin{cases} \mathit{undo}(a) & \text{if } \mathit{Invertible}(a) \text{ holds} \\ \delta & \text{otherwise.} \end{cases}$$

□

We have the following identities, which state that double application of the functions *Flag* and *Undo* yields δ. Iterated application is not allowed because both functions are not defined on δ.

Corollary 4.1.3. *Let* $a \in A_{flag}$.

$$\begin{aligned} \mathit{Invertible}(a) \quad \Longrightarrow \quad & (\mathit{Flag}(\mathit{Flag}(a)) = \delta) \wedge (\mathit{Undo}(\mathit{Undo}(a)) = \delta) \wedge \\ & (\mathit{Undo}(\mathit{Flag}(a)) = \delta) \wedge (\mathit{Flag}(\mathit{Undo}(a)) = \delta). \end{aligned}$$

Proof. Follows easily from the Definitions 4.1.1 and 4.1.2. □

Having defined the functions *Flag* and *Undo*, we extend $\Sigma(\mathrm{BPA}_{\mathrm{gce}}, \mathbb{B}(A))$ to a setting with A_{flag}. First the Boolean algebra needed for backtracking is defined.

Definition 4.1.4. Given a partitioned set of actions A_{flag}, a Boolean algebra $\mathbb{B}(A_{flag})$ is defined as containing expressions

- $\{\mathit{enabled}(a) \mid \text{ for all } a \in A_{flag}\}$,
- $\{\mathit{Inert}(a), \mathit{Invertible}(a), \mathit{Pass}(a), \mathit{Commit}(a) \mid a \in A_{flag}\}$,

and satisfying

- $\mathit{Inert}(a) = \begin{cases} \textbf{true} \text{ if } \mathit{Inert}(a) \text{ holds in the partition} \\ \textbf{false} \text{ otherwise,} \end{cases}$

  and similarly for the predicates $\mathit{Invertible}, \mathit{Pass}$ and Commit,
- $\mathit{enabled}(\mathit{flag}(a)) = \mathit{enabled}(a)$ whenever $\mathit{Invertible}(a)$ holds.

$\square$

Closed terms over $\Sigma(\mathrm{BPA_{gce}}, \mathbb{B}(A_{flag}))$ are now evaluated in a data environment $\mathcal{S}_{\mathit{flag}}$.

Definition 4.1.5. A *data environment* $\mathcal{S}_{\mathit{flag}}$ over a partitioned set of actions A_{flag} and a Boolean algebra $\mathbb{B}(A_{flag})$ is a tuple $\langle S, \mathit{action}, \mathit{effect}, \mathit{eval}\rangle$, where

- S is a non-empty set of data-states,
- $\mathit{action} : A_{flag} \times S \to A_{flag} \cup \{\delta\}$,
- $\mathit{effect} : A_{flag} \times S \to S$, satisfying

$$\mathit{Invertible}(a) = \textbf{true} \quad \Longrightarrow \quad \begin{cases} \forall s \in S \,.\, \mathit{effect}(\mathit{flag}(a), s) = \mathit{effect}(a, s), \\ \forall s \in S \,.\, \mathit{effect}(\mathit{undo}(a), \mathit{effect}(a, s)) = s, \end{cases}$$

$$\mathit{Inert}(a) = \textbf{true} \quad \Longrightarrow \quad \forall s \in S \,.\, (\mathit{enabled}(a, s) = \textbf{true} \Rightarrow \mathit{effect}(a, s) = s),$$

- $\mathit{eval} : \mathbb{B}(A_{flag}) \times S \to \mathbb{B}(A_{flag})$, satisfying the axioms in Table 7 and whenever $\mathit{Invertible}(a) = \textbf{true}$, also satisfying

$$\mathit{eval}(\mathit{enabled}(a), s) \leq \mathit{eval}(\mathit{enabled}(\mathit{undo}(a)), \mathit{effect}(a, s)).$$

$\square$

It seems straightforward to require $\mathit{action}(\mathit{flag}(a), s) = \mathit{action}(a, s)$ whenever $\mathit{Invertible}(a) = \textbf{true}$. We do not, however, because it may be desirable to keep the distinction between $\mathit{flag}(a)$ and a after evaluation of the process in which they occur with the state operator.

In the previous section a data-state dependent predicate was defined in some data environment $\mathcal{S}$ that described the actual status of an action in a certain data-state: $\mathit{invertible}(a, s)$, which holds if a is enabled in s (i.e. $\mathit{eval}(\mathit{enabled}(a), s) \neq \textbf{false}$), and a is an invertible action. In a similar way $\mathit{pass}(a, s)$ and $\mathit{commit}(a, s)$ were defined. These semantic predicates have their counterparts in a Boolean algebra $\mathbb{B}(A_{flag})$.

Definition 4.1.6. Given $\mathbb{B}(A_{flag})$, we define the following abbreviations for all $a \in A_{flag}$:

$$\begin{aligned} \mathit{invertible}(a) &= \mathit{enabled}(a) \wedge \mathit{Invertible}(a), \\ \mathit{pass}(a) &= \mathit{enabled}(a) \wedge \mathit{Pass}(a), \\ \mathit{commit}(a) &= \mathit{enabled}(a) \wedge \mathit{Commit}(a). \end{aligned}$$

$\square$

From the definition of the partition of A_{flag} it follows that also these predicates are mutually exclusive. From the above information we can derive a simple result:

$$enabled(a) = invertible(a) \vee pass(a) \vee commit(a).$$

Now we add the binary backtracking operator $\dotplus$ (introduced in Section 3), and the functions *Flag* and *Undo* to the signature $\Sigma(\mathrm{BPA}_{\mathrm{gce}}, \mathbb{B}(A_{flag}))$. The signature thus obtained, $\Sigma(\mathrm{BPA}_{\mathrm{gce}}, Flag, Undo, \dotplus, \mathbb{B}(A_{flag}))$, will further be abbreviated as

$$\Sigma(\mathrm{BPA}(\dotplus)).$$

Let $\mathcal{P}$ denote the set of process terms over $\Sigma(\mathrm{BPA}(\dotplus))$, and $\mathcal{B}$ the set of Boolean expressions over $\mathbb{B}(A_{flag})$. Consequently, we extend the domain of the *enabled* predicate to terms over $\Sigma(\mathrm{BPA}(\dotplus))$.

The axioms for the $\dotplus$ operator are listed in Table 10. The binding power of $\dotplus$ is taken to be less than $\cdot$ and stronger than $:\rightarrow$. The axiom system $\mathrm{BPA}_{\mathrm{gce}}$, extended with the 5 axioms for the $\dotplus$ operator will be referred to as $\mathrm{BPA}(\dotplus)$. Observe that this way of axiomatising the $\dotplus$ operator is in accordance with the basic term scheme of Definition 2.1.1.

(Ba1)	$\delta \dotplus x = x$
(Ba2)	$a \dotplus x = enabled(a) :\rightarrow a + \neg enabled(a) :\rightarrow x$
(Ba3)	$a \cdot x \dotplus y = invertible(a) :\rightarrow Flag(a) \cdot (x \dotplus Undo(a) \cdot y) +$ $pass(a) :\rightarrow a \cdot (x \dotplus y) +$ $commit(a) :\rightarrow a \cdot x +$ $\neg enabled(a) :\rightarrow y$
(Ba4)	$(x + y) \dotplus z = enabled(x) :\rightarrow x \dotplus z + enabled(y) :\rightarrow y \dotplus z +$ $\neg enabled(x) \wedge \neg enabled(y) :\rightarrow z$
(Ba5)	$(\phi :\rightarrow x) \dotplus y = \phi :\rightarrow x \dotplus y + \neg\phi :\rightarrow y$

Table 10. The axioms of $\mathrm{BPA}(\dotplus)$ for backtracking, where $a \in A_{flag}$, $\phi \in \mathcal{B}$.

Axiom Ba1 expresses that δ in the left argument of the $\dotplus$ operator leads to the choice of the right argument. The axiom Ba2 states that backtracking over a single action as the left argument of the $\dotplus$ operator does not occur: the occurrence of a single action a leads either to successful termination or to the choice of the right argument, depending on the enabledness of a. The summand $enabled(a) :\rightarrow a$ equals a according to Lemma 2.2.3. The guards in Ba3, which are mutually exclusive, represent the actual test on the atomic actions under the $\dotplus$ operator. In this axiom the core of the backtracking mechanism is best visible: if an action a is invertible, then a is removed from the scope of $\dotplus$ after applying the function *Flag* to a, and the formal inverse of a is inserted with the function *Undo*, prefixing the right argument of the $\dotplus$ operator. Note that certain restrictions on the set of actions A_{flag} may lead to the cancellation of summands in the right hand side of Ba3. Axiom Ba4 defines how the choice operator $+$ distributes over $\dotplus$, and axiom Ba5 defines how the guarded command construct $:\rightarrow$ is removed from the scope of $\dotplus$.

The Booleans representing the partitioning predicates *Inert*, *Invertible*, *Pass* and *Commit* have the useful property of any Boolean $\phi \in \{\mathbf{true}, \mathbf{false}\}$:

$$\phi :\rightarrow x \cdot y = \phi :\rightarrow x \cdot (\phi :\rightarrow y).$$

As a consequence, we can replace $\mathit{Flag}(a)$ by $\mathit{flag}(a)$ in axiom Ba3 as soon as it is known that $\mathit{Invertible}(a) = \mathbf{true}$. Likewise we can replace $\mathit{Undo}(a)$ by $\mathit{undo}(a)$.

The following theorem states that the $\dotplus$ operator can be eliminated from closed $\Sigma(\mathrm{BPA}(\dotplus))$ terms. Consequently, properties of $\Sigma(\mathrm{BPA}(\dotplus))$ terms can be proved by induction on the structure of basic terms over $\Sigma(\mathrm{BPA_{gce}}, \mathbb{B}(A_{flag}))$ (see the Representation Lemma 2.1.2).

Theorem 4.1.7 (*Elimination*)**.**

1. *If p is a closed term over $\Sigma(\mathrm{BPA}(\dotplus))$, then there is a basic term $\tilde{p}$ over the signature $\Sigma(\mathrm{BPA_{gce}}, \mathbb{B}(A_{flag}))$ such that*

$$\mathrm{BPA}(\dotplus) \vdash p = \tilde{p}.$$

2. $\mathrm{BPA}(\dotplus)$ *is a conservative extension of* $\mathrm{BPA_{gce}}$*, i.e., for all closed terms p and q over $\Sigma(\mathrm{BPA_{gce}}, \mathbb{B}(A_{flag}))$ we have*

$$\mathrm{BPA}(\dotplus) \vdash p = q \iff \mathrm{BPA_{gce}} \vdash p = q.$$

Proof. See Appendix. □

4.2 Properties of the $\dotplus$ operator

In this section some properties of the $\dotplus$ operator are proved. The Enabledness Theorem 4.2.2 shows that $\mathit{enabled}(x \dotplus y)$ is equivalent to $\mathit{enabled}(x + y)$. After this, we will show that associativity does not hold in general for the $\dotplus$ operator: an extra constraint on $\mathbb{B}(A_{flag})$ must be satisfied to have associativity.

We start off by giving some identities which demonstrate the interaction of $\dotplus$ and the guarded command $:\rightarrow$ in various ways. These identities are used for further results.

Lemma 4.2.1.

1. $$\phi :\rightarrow (\phi :\rightarrow x) \dotplus y = \phi :\rightarrow x \dotplus y,$$

2. $$\begin{aligned}(\phi :\rightarrow x + \psi :\rightarrow y) \dotplus z = {} & \phi \wedge \mathit{enabled}(x) :\rightarrow x \dotplus z + {} \\ & \psi \wedge \mathit{enabled}(y) :\rightarrow y \dotplus z + {} \\ & (\neg\phi \vee \neg \mathit{enabled}(x)) \wedge (\neg\psi \vee \neg \mathit{enabled}(y)) :\rightarrow z,\end{aligned}$$

3. $$x \dotplus y = \mathit{enabled}(x) :\rightarrow x \dotplus y + \neg \mathit{enabled}(x) :\rightarrow y,$$

4. $$(\phi :\rightarrow x + \neg\phi :\rightarrow y) \dotplus z = \phi :\rightarrow x \dotplus z + \neg\phi :\rightarrow y \dotplus z.$$

Proof. See Appendix. □

The following theorem states that $enabled(x \dotplus y)$ is equivalent to $enabled(x+y)$ whenever x represents a closed term. A result which from a semantic point of view is not very surprising.

Theorem 4.2.2 (*Enabledness*). *If p is a closed term over $\Sigma(\mathrm{BPA}(\dotplus))$ then the following holds:*

$$enabled(p \dotplus y) = enabled(p) \vee enabled(y).$$

Proof. See Appendix. $\square$

In order to obtain an associative operator for backtracking, a restriction on the Boolean algebra $\mathbb{B}(A_{flag})$ is needed. We illustrate this with an example.

Example 4.2.3. Let $a, b, c \in A_{flag}$, and $Invertible(a) = enabled(undo(b)) = \mathbf{true}$, then

1. $(a \cdot undo(b) \cdot \delta \dotplus \delta) \dotplus c \ =$
   $enabled(a) :\rightarrow flag(a) \cdot undo(b) \cdot (undo(a) \cdot c + \neg enabled(undo(a)) :\rightarrow c) +$
   $\neg enabled(a) :\rightarrow c,$

2. $a \cdot undo(b) \cdot \delta \dotplus (\delta \dotplus c) \ =$
   $enabled(a) :\rightarrow flag(a) \cdot undo(b) \cdot undo(a) \cdot c +$
   $\neg enabled(a) :\rightarrow c$

(see the Appendix for a proof). It is easy to see that the two process terms above are not equal. Consequently, associativity cannot hold in general for the $\dotplus$ operator. **End example.**

We can obtain associativity for $\dotplus$ by only regarding Boolean algebras in which the actions $undo(a)$ are uniformly enabled (as suggested by the example above).

Definition 4.2.4. Let $a \in A_{flag}$. A *restricted Boolean algebra* $\mathbb{B}(A_{flag})^-$ is a Boolean algebra $\mathbb{B}(A_{flag})$ that satisfies the constraint

$$Invertible(a) = enabled(Undo(a)).$$

We write $\Sigma(\mathrm{BPA}(\dotplus))^-$ for a signature $\Sigma(\mathrm{BPA}(\dotplus))$ defined over a Boolean algebra $\mathbb{B}(A_{flag})^-$. $\square$

Lemma 4.2.5. *Let $a \in A_{flag}$ and $Invertible(a) = \mathbf{true}$. In a restricted Boolean algebra $\mathbb{B}(A_{flag})^-$ the following identity holds:*

$$pass(undo(a)) = \mathbf{true}.$$

Proof. By definition. $\square$

Theorem 4.2.6 (*Associativity*). *If p is a closed term over $\Sigma(\mathrm{BPA}(\dotplus))^-$, then backtracking is associative:*

$$(p \dotplus y) \dotplus z = p \dotplus (y \dotplus z).$$

Proof. See Appendix. □

We give an example that shows why we defined *flag* actions, and consequently had to extend the set of atomic actions from A to A_{flag}, in order to obtain associativity of $\dotplus$. For this purpose, we can assume that $Flag(a) = a$ for $a \in A$, and use axiom Ba3 in its current form. If $Invertible(a) = \mathbf{true}$, then the following identity can be derived.

$$(a \cdot \delta \dotplus \delta) \dotplus b = a \cdot undo(a) \cdot undo(a) \cdot b + \neg enabled(a) :\rightarrow b,$$

which is in general not equal to

$$a \cdot \delta \dotplus (\delta \dotplus b) = a \cdot undo(a) \cdot b + \neg enabled(a) :\rightarrow b.$$

Another design decision was to define $Pass(undo(a))$ for all invertible actions a. The reason for this is again the associativity of $\dotplus$; it can neither be allowed to define $Commit(undo(a))$ nor to define $Invertible(undo(a))$. We illustrate the inaptitude of the second alternative with an example. Suppose we have $Invertible(a) = Invertible(undo(a)) = \mathbf{true}$, and $flag(undo(a))$ denoting the flagged duplicate of $undo(a)$ and $undo(undo(a))$ denoting the inverse of $undo(a)$. Then we can derive

$$\begin{aligned}&(a \cdot \delta \dotplus \delta) \dotplus b \;=\\ &\quad enabled(a) :\rightarrow flag(a) \cdot flag(undo(a)) \cdot undo(undo(a)) \cdot b + \neg enabled(a) :\rightarrow b,\end{aligned}$$

which can in general not be equal to

$$a \cdot \delta \dotplus (\delta \dotplus b) = enabled(a) :\rightarrow flag(a) \cdot undo(a) \cdot b + \neg enabled(a) :\rightarrow b.$$

4.3 Bisimulation and relational semantics

First we give the transition rules for $\Sigma(\mathrm{BPA}(\dotplus))^{\lambda,\mathcal{S}}$ by combining those of Tables 6 and 9 with the ones given in Table 11.

Lemma 4.3.1 (*Congruence*). *Let $\mathcal{S}$ be given. The relation $\leftrightarrows$ between closed terms over the signature $\Sigma(\mathrm{BPA}(\dotplus))^{\lambda,\mathcal{S}}$ is a congruence with respect to the operator $\dotplus$.*

Proof. See Appendix for a sketch. □

We have the following result:

Theorem 4.3.2 (*Soundness*). *Let a data environment $\mathcal{S}$ be given, and let p, q be closed terms over $\Sigma(\mathrm{BPA}(\dotplus))^{\lambda,\mathcal{S}}$. It holds that*

$$\mathrm{BPA}(\dotplus) + \mathrm{E1–6} + \mathrm{SOG1–4} \vdash p = q \quad \Longrightarrow \quad p \leftrightarrows q.$$

$$a \in A \quad \frac{}{a \xrightarrow{enabled(a):\rightarrow a} \surd} \text{ if } enabled(a) \neq \mathbf{false}$$

$$a \in A \quad \frac{}{\mathit{flag}(a) \xrightarrow{enabled(a):\rightarrow a} \surd} \text{ if } enabled(a) \neq \mathbf{false} \wedge Invertible(a) = \mathbf{true}$$

$$\dotplus \qquad \frac{y \xrightarrow{\phi:\rightarrow b} y'}{x \dotplus y \xrightarrow{\phi:\rightarrow b} y'} \text{ if } \phi \wedge enabled(x) = \mathbf{false} \qquad \frac{y \xrightarrow{\phi:\rightarrow b} \surd}{x \dotplus y \xrightarrow{\phi:\rightarrow b} \surd} \text{ if } \phi \wedge enabled(x) = \mathbf{false}$$

$$\frac{x \xrightarrow{\phi:\rightarrow b} x'}{x \dotplus y \xrightarrow{\phi:\rightarrow b} x' \dotplus undo(a) \cdot y} \text{ if } Invertible(b) = \mathbf{true}$$

$$\frac{x \xrightarrow{\phi:\rightarrow b} x'}{x \dotplus y \xrightarrow{\phi:\rightarrow b} x' \dotplus y} \text{ if } Pass(b) = \mathbf{true} \qquad \frac{x \xrightarrow{\phi:\rightarrow b} x'}{x \dotplus y \xrightarrow{\phi:\rightarrow b} x'} \text{ if } Commit(b) = \mathbf{true}$$

$$\frac{x \xrightarrow{\phi:\rightarrow b} \surd}{x \dotplus y \xrightarrow{\phi:\rightarrow b} \surd}$$

Table 11. Additional transition rules for BPA($\dotplus$), where $b \in A_{flag}$, $\phi \in \mathcal{B}$.

Proof. It is easy to check that all new axioms of BPA($\dotplus$) are valid. By the Congruence Lemma 4.3.1 the soundness of BPA($\dotplus$) follows immediately. □

In order to prove that the Requirements I - IV, formulated in Section 3, are satisfied by the relational semantics of $\dotplus$, we formally define the relational semantics for closed terms over Σ(BPA($\dotplus$)).

Definition 4.3.3. Let p, q denote closed terms over Σ(BPA($\dotplus$)), and a data environment $\mathcal{S}$ be given. Then

$$\forall s \in S \,.\, [\![p \dotplus q]\!](s) \stackrel{\text{def}}{=} [\![r]\!](s),$$

where r is a closed term over $\Sigma(\text{BPA}_{\text{gce}}, \mathbb{B}(A_{flag}))$ such that BPA($\dotplus$) $\vdash p \dotplus q = r$. □

By the Elimination Theorem 4.1.7 the expression $[\![p \dotplus q]\!](s)$ is *well-defined*: according to this theorem the $\dotplus$ operator can be eliminated from closed $\Sigma(\mathrm{BPA}(\dotplus))$ expressions. Moreover, because $\mathrm{BPA}(\dotplus)$ is a conservative extension of $\mathrm{BPA_{gce}}$, we find that if $\mathrm{BPA}(\dotplus) \vdash p \dotplus q = r$ and $\mathrm{BPA}(\dotplus) \vdash p \dotplus q = r'$, then $\mathrm{BPA_{gce}} \vdash r = r'$. Using Lemma 3.1.2 we find that $\forall s \in S \,.\, [\![r]\!](s) = [\![r']\!](s)$.

Theorem 4.3.4 (*Requirements*)**.** *Requirements I – IV are satisfied by the relational semantics of* $\dotplus$.

Proof. See Appendix. □

5 Recursively defined processes and an example

In this section Basic Process Algebra with backtracking is extended with recursion. Furthermore, an example is given of a recursive specification over $\Sigma(\mathrm{BPA}(\dotplus))$, as well as its evaluation with state operators.

5.1 Specifying processes by recursive equations.

We introduce processes defined by possibly recursive equations. We do *not* consider state operators as a means to specify processes in this way.

Definition 5.1.1. A *recursive specification* $E = \{x = t_x \,|\, x \in V_E\}$ *over* the signature $\Sigma(\mathrm{BPA}(\dotplus))$ is a set of equations where V_E is a possibly infinite set of indexed variables and t_x a term over $\Sigma(\mathrm{BPA}(\dotplus))$ such that the variables in t_x are also in V_E. □

A *solution* of a recursive specification $E = \{x = t_x \,|\, x \in V_E\}$ is an interpretation of the variables in V_E as processes, such that the equations of E are satisfied. For instance, the recursive specification $\{x = x\}$ has any process as a solution for x, and $\{x = ax\}$ has the infinite process "a^ω" as a solution for x. The following syntactical restriction on recursive specifications turns out to enforce unique solutions (modulo bisimilarity).

Definition 5.1.2. Let t be a term over the signature $\Sigma(\mathrm{BPA}(\dotplus))$, and $E = \{x = t_x \,|\, x \in V_E\}$ a recursive specification over $\Sigma(\mathrm{BPA}(\dotplus))$.

- An occurrence of a variable x in t is *guarded* iff t has a subterm of the form $a \cdot M$ with $a \in A_{flag}$, and this x occurs in M.
- The specification E is *syntactically guarded* iff all occurrences of variables in the terms t_x are guarded.
- The specification E is *guarded* iff there is a syntactically guarded specification $E' = \{x = t'_x \,|\, x \in V_E\}$ over $\Sigma(\mathrm{BPA}(\dotplus))$ such that $\mathrm{BPA}(\dotplus) \vdash t_x = t'_x$ for all t_x.

□

Now the signature $\Sigma(\mathrm{BPA}(+))_{\mathrm{REC}}$, containing representations of recursively defined processes, is defined as follows.

Definition 5.1.3. The signature $\Sigma(\mathrm{BPA}(+))_{\mathrm{REC}}$ is obtained by extending the signature $\Sigma(\mathrm{BPA}(+))$ in the following way: for each guarded specification $E = \{x = t_x \,|\, x \in V_E\}$ over $\Sigma(\mathrm{BPA}(+))$ a set of constants $\{<x\,|\,E>|\; x \in V_E\}$ is added, where the construct $<x\,|\,E>$ denotes the x-component of a solution of E. □

Some more notations: let $E = \{x = t_x \,|\, x \in V_E\}$ be a guarded specification over $\Sigma(\mathrm{BPA}(+))$, and t some term over $\Sigma(\mathrm{BPA}(+))_{\mathrm{REC}}$. Then $<t\,|\,E>$ denotes the term in which each occurrence of a variable $x \in V_E$ in t is replaced by $<x\,|\,E>$, e.g., the expression $<aax\,|\,\{x = ax\}>$ denotes the term $aa<x\,|\,\{x = ax\}>$.

For the constants of the form $<x \mid E>$ there are two axioms in Table 12. In these axioms the letter E ranges over guarded specifications. The axiom REC states that the constant $<x \mid E>$ $(x \in V_E)$ is a solution for the x-component of E, so it expresses that each guarded recursive system has *at least* one solution for each of its (bound) variables. The conditional rule RSP (Recursive Specification Principle) expresses that E has *at most* one solution for each of its variables: whenever one can find processes p_x $(x \in V_E)$ satisfying the equations of E, notation $E(\vec{p_x})$, then $p_x = <x\,|\,E>$.

(REC)	$<x\,|\,E> = <t_x\,|\,E>$ if $x = t_x \in E$
(RSP)	$\dfrac{E(\vec{p_x})}{p_x = <x\,|\,E>}$ if $x \in V_E$

Table 12. Axioms for guarded recursive specifications.

Finally, a convenient notation is to abbreviate $<x\,|\,E>$ for $x \in V_E$ by X once E is fixed, and to represent E only by its REC instances. The following example shows all notations concerning recursively specified processes, and illustrates the use of REC and RSP.

Example 5.1.4. Consider the guarded recursive specifications $E = \{x = ax\}$ and $E' = \{y = ayb\}$ over $\Sigma(\mathrm{BPA}(+))$. So by the convention just introduced, we write $X = aX$ and $Y = aYb$. With REC and RSP one can prove

$$\mathrm{BPA}(+) + \mathrm{REC} + \mathrm{RSP} \vdash X = Y$$

in the following way. First note that $Xb = aXb$ by REC, so $E(Xb)$ is derivable. Application of RSP yields

$$Xb = X. \tag{1}$$

Moreover, $Xb \overset{\mathrm{REC}}{=} aXb \overset{(1)}{=} aXbb$, and hence $E'(Xb)$ is derivable. A second application of RSP yields $Xb = Y$. Combining this with (1) gives the desired result. **End example.**

The general transition rule by which processes defined by guarded recursive specifications are associated with transitions systems is given in Table 13. The specification $E = \{x = t_x \mid x \in V_E\}$ denotes a guarded recursive specification over the signature $\Sigma(\mathrm{BPA}(\dotplus))$.

$$\frac{\langle t_x \mid E\rangle \xrightarrow{\phi:\to a} x'}{\langle x \mid E\rangle \xrightarrow{\phi:\to a} x'} \quad \text{if } x = t_x \in E$$

Table 13. Transition rule for guarded recursive specifications, where $a \in A_{flag}$, $\phi \in \mathcal{B}$.

The algebraic manipulation of process terms over $\Sigma(\mathrm{BPA}(\dotplus))_{\mathrm{REC}}$ may require the axiom SB, *Standard Backtracking*, stating that backtracking is associative. The reader should keep in mind that this axiom is sound in a signature with a restricted Boolean algebra $\mathbb{B}(A_{flag})^-$, but that associativity does not hold in general. Without proof we state that the Enabledness Theorem 4.2.2 is derivable for process terms over $\Sigma(\mathrm{BPA}(\dotplus))_{\mathrm{REC}}$.

(SB)	$(x \dotplus y) \dotplus z = x \dotplus (y \dotplus z)$

Table 14. The axiom for *Standard Backtracking.*

5.2 An example of a process with backtracking

In this section we evaluate in $\Sigma(\mathrm{BPA_{gce}}, \mathbb{B}(A_{flag}))_{\mathrm{REC}}$ a recursive specification over $\Sigma(\mathrm{BPA}(\dotplus))_{\mathrm{REC}}$ in a given data environment $\mathcal{S}_{flag}$ in a specific initial data-state. By virtue of the Elimination Theorem 4.1.7 we can apply any state operator to closed terms over $\Sigma(\mathrm{BPA}(\dotplus))_{\mathrm{REC}}$. By the definition of guardedness, also recursive specifications over $\Sigma(\mathrm{BPA}(\dotplus))_{\mathrm{REC}}$ can be evaluated.

Our example shows in an easy way that many of the previously defined notions can be combined to analyse a small problem, borrowed from the chess game.

Example 5.2.1. A well-known problem from the chess game that can be solved with backtracking is the *8 Queens Problem* (see e.g. [Bra86]). The problem can be formulated as follows: “Put 8 queens on a chessboard such that none of the queens attacks another”. In order to illustrate the backtracking mechanism of $\mathrm{BPA}(\dotplus)_{\mathrm{REC}}$, we reduce this problem to a much simpler one that is in essence analogue. Our simplified version of this problem is "Put 3 rooks on a 3×3 'chessboard' such that none of the rooks attacks another". We start off by defining the set of atomic actions A and the Boolean algebra $\mathbb{B}(A)$. We use the sort *Nat* for representing the natural numbers. On *Nat* we have the functions $+$, $\dot{-}$ and an equality function *eq*.

As a set A of atomic actions for the *3 Rooks Problem* we choose

$$A \stackrel{\text{def}}{=} \{put_i, putback_i, write, write(k_1, k_2, k_3), ready \mid i \in \{1,2,3\}, k_i \in Nat\},$$

where put_i and $putback_i$ put Rook i on another position, *write* is evaluated as $write(k_1, k_2, k_3)$ writing the current data-state (the positions of the three rooks) to some external device, and *ready* indicates that the process has terminated.

The Boolean algebra $\mathbb{B}(A)$ we use for the *3 Rooks Problem* contains, next to expressions $enabled(a)$ with $a \in A$, expressions

$$eq(n_1, n_2),$$

where n_1, n_2 represent natural numbers.

Let $i \in \{1,2,3\}$ and $n_i, k_i, k_i' \in Nat$. For solving the *3 Rooks Problem* we take the following data environment $\mathcal{S} = \langle S, action, effect, eval\rangle$:

- $S \stackrel{\text{def}}{=} \{(k_1, k_2, k_3) \mid k_1, k_2, k_3 \in Nat\}$.
- The *action* function is defined as the identity function on A, except for

$$action(write, (k_1, k_2, k_3)) \stackrel{\text{def}}{=} write(k_1, k_2, k_3).$$

- The *effect* function is defined with the help of a predicate *attack*. The substitution of k_i' for k_i in (k_1, k_2, k_3) is denoted by $(k_1, k_2, k_3)[k_i'/k_i]$.

$$attack(n_i, (k_1, k_2, k_3)) \stackrel{\text{def}}{=} \bigvee_{1 \leq j < i} n_i = k_j \text{ in } Nat,$$

$$effect(put_i, (k_1, k_2, k_3)) \stackrel{\text{def}}{=}$$
$$\begin{cases} (k_1, k_2, k_3)[(k_i + 1)/k_i] & \text{if } \neg attack((k_i+1)_i, (k_1, k_2, k_3)) \text{ in } Nat \\ effect(put_i, (k_1, k_2, k_3)[(k_i+1)/k_i]) & \text{otherwise,} \end{cases}$$

$$effect(putback_i, (k_1, k_2, k_3)) \stackrel{\text{def}}{=}$$
$$\begin{cases} (k_1, k_2, k_3) & \text{if } k_i = 0 \text{ in } Nat \\ (k_1, k_2, k_3)[(k_i \dot{-} 1)/k_i] & \text{if } k_i \neq 0 \text{ in } Nat \text{ and} \\ & \neg attack((k_i \dot{-} 1)_i, (k_1, k_2, k_3)) \text{ in } Nat \\ effect(putback_i, (k_1, k_2, k_3)[(k_i \dot{-} 1)/k_i]) & \text{otherwise.} \end{cases}$$

  The actions *write*, $write(k_1, k_2, k_3)$ and *ready* are inert, and uniformly enabled.
- The *eval* function is given by

$$eval(enabled(put_i), (k_1, k_2, k_3)) \stackrel{\text{def}}{=} \begin{cases} \textbf{true} & \text{if } \exists n_i \,.\, k_i < n_i \leq 3 \text{ and} \\ & \quad \neg attack(k_i, (k_1, k_2, k_3)) \text{ and} \\ & \quad \neg attack(n_i, (k_1, k_2, k_3)) \text{ in } Nat \\ \textbf{false} & \text{otherwise,} \end{cases}$$

$$eval(enabled(putback_i), (k_1, k_2, k_3)) \stackrel{\text{def}}{=} \textbf{true},$$

$$eval(eq(n_1, n_2), (k_1, k_2, k_3)) \stackrel{\text{def}}{=} \begin{cases} \textbf{true} & \text{if } n_1 = n_2 \text{ in } Nat \\ \textbf{false} & \text{otherwise.} \end{cases}$$

Without proof we state that $putback_i$ is an inverse action of put_i so we partition A by defining $Invertible(put_i)$, $Pass(putback_i)$ and $Pass(write)$. We furthermore define $Pass(write(k_1, k_2, k_3))$ and $Pass(ready)$, though we also could have classified these actions as *Commit*. For the partitioned set of actions we then have

$$\begin{aligned} A_{flag} = \{ & put_i, undo(put_i), flag(put_i), write, write(k_1, k_2, k_3), ready \\ & \mid i \in \{1, 2, 3\}, k_i \in Nat\}, \end{aligned}$$

where

$$undo(put_i) \stackrel{\text{def}}{=} putback_i.$$

The definition of an extended data environment $\mathcal{S}_{flag}$ is simple: extend the function *action* to A_{flag} with the identity on *flag* actions (note that the requirements in Definition 4.1.5 are satisfied). Let $i \in Nat$. The process that generates solutions of the 3 Rooks Problem is specified by

$$\begin{aligned} E \stackrel{\text{def}}{=} \{\ B_i = \; & eq(i, 1) :\rightarrow (put_1 \cdot B_2 + \neg enabled(put_1) :\rightarrow ready) + \\ & eq(i, 2) :\rightarrow put_2 \cdot B_3 \dotplus B_1 + \\ & eq(i, 3) :\rightarrow put_3 \cdot write \cdot \delta \dotplus put_2 \cdot B_3\ \}. \end{aligned}$$

The process B_i always tries to put Rook i on a next position on column i, such that it does not attack any rook on a column $j < i$. If this is not possible then if $i > 1$ it tries to put the rook on column $i - 1$ on a next position. As soon as the rook on column 1 cannot be put on a next position, the process terminates. If Rook 3 is put on a new position a *write* action follows, after which a δ (triggering backtracking) is met.

Finally we prove that E specifies a process that can generate all possible solutions of the 3 Rooks Problem. We demonstrate this by evaluating $\lambda_{(0,0,0)}(B_1)$, which is the process that starts by putting Rook 1 on our chessboard, where all rooks are on a row $(0, 0, 0)$ (say, not on the chessboard).

The evaluation of B_1 in data-state $(0, 0, 0)$ yields a $\Sigma(\text{BPA}_\delta)$ process that performs $write(k_1, k_2, k_3)$ actions for all possible solutions of the 3 Rooks Problem as follows (see the Appendix for a proof):

$$\begin{aligned} \lambda_{(0,0,0)}(B_1) = \; & put_1 \cdot \\ & \quad flag(put_2) \cdot \\ & \quad\quad flag(put_3) \cdot write(1, 2, 3) \cdot undo(put_3) \cdot \\ & \quad\quad flag(put_2) \cdot \\ & \quad\quad\quad flag(put_3) \cdot write(1, 3, 2) \cdot undo(put_3) \cdot \\ & \quad\quad undo(put_2) \cdot \\ & \quad undo(put_2) \cdot \\ & \lambda_{(1,0,0)}(B_1) \end{aligned}$$

$$\begin{aligned} \lambda_{(1,0,0)}(B_1) = \; & put_1 \cdot \\ & \quad flag(put_2) \cdot \\ & \quad\quad flag(put_3) \cdot write(2, 1, 3) \cdot undo(put_3) \cdot \\ & \quad\quad flag(put_2) \cdot \\ & \quad\quad\quad flag(put_3) \cdot write(2, 3, 1) \cdot undo(put_3) \cdot \\ & \quad\quad undo(put_2) \cdot \\ & \quad undo(put_2) \cdot \\ & \lambda_{(2,0,0)}(B_1) \end{aligned}$$

$$
\begin{aligned}
\lambda_{(2,0,0)}(B_1) = put_1 \cdot & \\
& \mathit{flag}(put_2)\cdot \\
& \quad \mathit{flag}(put_3) \cdot write(3,1,2) \cdot undo(put_3)\cdot \\
& \quad \mathit{flag}(put_2)\cdot \\
& \qquad \mathit{flag}(put_3) \cdot write(3,2,1) \cdot undo(put_3)\cdot \\
& \quad undo(put_2)\cdot \\
& undo(put_2)\cdot \\
ready. &
\end{aligned}
$$

The inverse actions used here for the specification of the 3 Rooks Problem are uniformly enabled. Consequently, the Boolean algebra $\mathbb{B}(A_{flag})$ is a restricted algebra $\mathbb{B}(A_{flag})^-$. However, for a proof of the evaluation above this is not necessary, because we do not use associativity of the $\dotplus$ operator. The 3 Rooks Problem can also be solved using a finite domain $\{0,1,2,3\}$ instead of *Nat*. In this case, the definition of a data environment for specifying this problem needs some adaptations. **End example.**

The 8 Queens Problem can be specified and evaluated analogously by changing S into a tuple of 8 natural numbers instead of 3, and by changing the *attack* predicate, such that attacks on the diagonals of the chessboard are included. The specification E requires only small adaptations.

Moreover we mention here that many variants of the 3 Rooks Problem are conceivable. For instance, only small changes in the definition of the data environment $\mathcal{S}$ make it possible to specify this problem with only one *put* and one *putback* action. Also non-deterministic choices between the rooks to be put can be specified, such that evaluation leads to various correct solutions in the form of traces with the desired *write* actions.

6 Concluding remarks

The operator $\dotplus$ for modelling backtracking over a given data environment was axiomatised. For this purpose, we defined the axiom system $\mathrm{BPA_{gce}}$ as a core system, containing an explicit notion of enabledness.

Observe that the special case

$$\mathit{enabled}(a) = \mathbf{true} \text{ for all } a \in A_{flag}$$

simplifies some parts of the theory considerably. Notably, the $\dotplus$ becomes associative for any Boolean algebra $\mathbb{B}(A_{flag})$. We remark that this case is equivalent to a setting without explicit enabledness: backtracking can only be triggered by guards or δ (and not by atomic actions anymore).

A closer study of what backtracking implies led us to the conclusion that the introduction of only the predicate *enabled* on the atomic actions is not sufficient for an axiomatisation of a backtracking operator. Some additional information on the nature of atomic actions is needed in order to decide how to deal with an action that is subject to backtracking. When every action gets the same treatment in the scope of a backtracking operator, the notion of backtracking becomes diffuse. For instance, if repeated backtracking on a single action is allowed, the mechanism becomes inefficient, and moreover a binary and associative backtracking operator seems impossible to axiomatise.

In order to motivate some of the design decisions we had to make for obtaining associativity of $+$, some examples were given. A basic design decision, which we made in Section 3, was to choose for a deterministic notion of invertibility: if an action a is invertible, then $undo(a)$ can undo any possible effect s' of a in some initial data-state s. So according to this notion we already know in data-state s that $undo(a)$ exists in s' *and* that it is enabled in s'.

A totally different, more general backtracking mechanism can be obtained by a different, non-deterministic, notion of invertibility. We can for instance, given an action a, also select a unique inverse action, say b, but not require enabledness of b in every possible effect of a (this implies a notion "possible invertibility" instead of semantic invertibility). In the case of backtracking, a test after execution of a is then needed to verify whether b is enabled or not, next to an invertibility test on a. Some study after this option led us to the conjecture that backtracking, using a non-deterministic notion of invertibility, is essentially different from backtracking with a deterministic notion of invertibility, and that it would be much more complicated to axiomatise an associative backtracking operator.

However, in the approach we took, also strong measures had to be taken in order to obtain associativity of $+$ within the setting of BPA($+$). We had to require uniform enabledness of inverse actions. This led us to defining a restricted Boolean algebra $\mathbb{B}(A_{flag})^-$.

The signature $\Sigma(\mathrm{BPA_{gce}}, \mathbb{B}(A))$ as introduced in Section 3 can easily be extended with parallel operators, suitable for the description of concurrent, communicating processes, and the same holds for $\Sigma(\mathrm{BPA}(+))$. The axiom system ACP (Algebra of Communicating Processes, see e.g. [BW90]), forms a suitable basis for such extensions. Care has to be taken, however, with the communications defined between atomic actions: if two actions a and b communicate to some resulting action c, then it can be derived that c must exactly be enabled if both a and b are (provided that the interaction between the guarded command $:\rightarrow$ and the *communication merge* $|$ is axiomatised by $(\phi :\rightarrow x) \mid (\psi :\rightarrow y) = \phi \wedge \psi :\rightarrow (x \mid y))$. In other words, the Boolean algebra $\mathbb{B}(A)$ must be compatible with the communication function γ by satisfying $enabled(c) = enabled(a) \wedge enabled(b)$ whenever $\gamma(a, b) = c$ $(a, b, c \in A)$.

The empty process ϵ (**skip**, axiomatised by $\epsilon \cdot x = x \cdot \epsilon = x$), is compatible with $\mathrm{BPA_{gce}}$ but *not* with BPA($+$): from the axiom $\epsilon \cdot x = x$ it follows that $enabled(\epsilon) =$ **true** must hold, and it is also evident that $\epsilon + x = \epsilon$. Now assume that $invertible(a) =$ **true** for some a. Then we can derive $a = a + x = a \cdot \epsilon + x = flag(a) \cdot (\epsilon + undo(a) \cdot x) = flag(a)$, contradicting the use and meaning of *flag* actions.

It is beyond the scope of this paper to make a detailed comparison of the backtracking mechanism of the language PROLOG with that of BPA($+$). We only mention one interesting similarity: in both formalisms there is the possibility to specify programs that refute any possibility of backtracking after a given program part (trace) has been executed. In PROLOG the *cut* predicate can be used to block the way back, and in BPA($+$) any uniformly enabled *commit* action can be used for this. An important difference, however, is that backtracking in a PROLOG program has a 'global' character; it is not restricted to the scope of a specific operator, such as in BPA($+$), but it covers a whole program.

References

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

[BB91] J.C.M. Baeten and J.A. Bergstra. Process algebra with signals and conditions. In M. Broy, editor, *Programming and Mathematical Methods, Proceedings Summer School Marktoberdorf 1991, NATO ASI Series F88*, pages 273–323. Springer-Verlag, 1991.

[BK84] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60(1/3):109–137, 1984.

[BV93] J.C.M. Baeten and C. Verhoef. A congruence theorem for structured operational semantics with predicates. Report CSN 93/05, Eindhoven University of Technology, 1993.

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

[Bak80] J.W. de Bakker. *Mathematical Theory of Program Correctness*. Prentice-Hall International, 1980.

[Bra86] I. Bratko. *PROLOG programming for artificial intelligence*. International Computer Science Series, Addison-Wesley Publishing Company, 1986.

[CM87] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, 1987.

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall International, Englewood Cliff, 1976.

[Eli92] A. Eliëns. *DLP – A language for Distributed Logic Programming*. Wiley, 1992.

[FJ92] L.M.G. Feijs and H.B.M. Jonkers. *Formal Specification and Design*. Cambridge University Press, 1992.

[GP90a] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. Report CS-R9076, CWI, Amsterdam, 1990.

[GP90b] J.F. Groote and A. Ponse. Process algebra with guards. Report CS-R9069, CWI, Amsterdam, 1990. To appear in *Formal Aspects of Computing*.

[GP91] J.F. Groote and A. Ponse. μCRL: A base for analysing processes with data. In E. Best and G. Rozenberg, editors, *Proceedings* 3$^{\text{rd}}$ *Workshop on Concurrency and Compositionality, Goslar, GMD-Studien Nr. 191*, pages 125–130. Universität Hildesheim, 1991.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs, 1985.

[ISO87] ISO. *Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour* ISO/TC97/SC21/N DIS8807, 1987.

[Jon91] H.B.M. Jonkers. *Protocold 1.1 User Manual*. Technical report RWR-513-hj-91080-hj, Philips Research Laboratories, 1991.

[Kli82] P. Klint. *From SPRING to SUMMER. Design, definition and implementation of programming languages for string manipulation and pattern matching*. PhD thesis, Technische Hogeschool Eindhoven, 1982.

[Klu91] A.S. Klusener. An executable semantics for a subset of COLD. Report CS-R9145, CWI, Amsterdam, 1991.

[MV90] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, XIII:85–139, 1990.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall International, Englewood Cliffs, 1989.

[Mon89] J.D. Monk, editor. *Handbook of Boolean Algebras*, Volume I. North-Holland, 1989.

[Par81] D.M.R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, 5^{th} *GI Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1981.

[Plo81] G.D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[VW93] S.F.M. van Vlijmen and J.J. van Wamel. A semantic approach to Protocold using process algebra. Report P9317, University of Amsterdam, 1993.

Appendix

Proof of the Congruence Lemma 2.3.4. This can be proved either in a direct manner, or by using the main result of BAETEN and VERHOEF [BV93]. We give a proof using the second alternative: in [BV93] it is shown that if transition rules satisfy some syntactical restrictions, the so-called *path* format, then strong bisimulation is a congruence. We cannot use this result in a direct way, because our definition of bisimulation is not standard: we demand evaluation of the guards in labels and, moreover, the existence of a *finite number* of 'matching' transitions. However, with some standard facts about Boolean algebras, we can relate our operational semantics with one to which the setting of [BV93] applies. As a general reference to Boolean algebras we mention [Mon89]. We first sketch our proof:

1. Embed $\mathbb{B}(A)$ in a complete, atomic Boolean algebra, say $\mathbb{B}(A)^+$, by some embedding f. Such embeddings exist by STONE's representation theorem.
2. Any Boolean $f(\phi)$ can be represented in $\mathbb{B}(A)^+$ as $\bigvee_{i\in I_\phi} at_i$ with at_i atomic. (For 1 it is essential that ϕ is a finite sum or product.)
3. The rules in Table 6 can be adapted to corresponding "$\mathbb{B}(A)^+$-transition rules" in the spirit of 1 and 2 above. For example, an axiom $a \xrightarrow{enabled(a):\rightarrow a} \surd$ corresponds with axioms $a \xrightarrow{at_i:\rightarrow a} \surd$ if $f(enabled(a)) = \bigvee_{i\in I} at_i$.
4. The $\mathbb{B}(A)^+$-transition rules satisfy the path format (we regard guarded commands unary operators).
5. Two transition systems are bisimilar (in the sense of Definition 2.3.3) iff their corresponding $\mathbb{B}(A)^+$ systems are. Here 'corresponding' means having the *same* initial state.
6. For the $\mathbb{B}(A)^+$-transition systems, our definition of bisimilarity coincides with the one following from [BV93] in this case.

From 4 – 6 it follows that our definition of bisimularity is a congruence.

We further comment on 3 and 4: We regard transition systems that differ in two aspects from the ones defined in Section 4.3. First, guarded commands are considered unary operators: for each ϕ in $\mathbb{B}(A)$, there is an operator $\phi :\rightarrow$ **.** Second, as labels we use expressions of the form

$$at :\rightarrow a$$

where at ranges over the Boolean *atoms* of $\mathbb{B}(A)^+$. This affects the precise definition of the *labels* of the transition systems defined before.

As for the transition rules, the axiom for atomic actions in Table 6 has to be replaced by

$$a \in A \quad \frac{}{a \xrightarrow{at_i:\rightarrow a} \surd} \quad \text{if } f(enabled(a)) = \textstyle\bigvee_{i\in I} at_i$$

(note that this may give rise to an infinite number of transitions). The two transition rules for the guarded commands have to be exchanged by

$$\frac{x \xrightarrow{at:\rightarrow a} x'}{\phi :\rightarrow x \xrightarrow{at:\rightarrow a} x'} \text{ if } at \leq f(\phi), \qquad \text{and} \qquad \frac{x \xrightarrow{at:\rightarrow a} \surd}{\phi :\rightarrow x \xrightarrow{at:\rightarrow a} \surd} \text{ if } at \leq f(\phi).$$

The remaining transition rules are the same as those in Table 6, though ϕ now ranges over the atoms of $\mathbb{B}(A)^+$. These rules indeed satisfy the *path* format defined in [BV93].

Further comments on 5 and 6: Call two closed terms p, q over $\Sigma(\text{BPA}_{\text{gce}}, \mathbb{B}(A))$ bisimilar w.r.t. $\mathbb{B}(A)^+$, notation

$$p \leftrightarroweq^+ q,$$

if their $\mathbb{B}(A)^+$ transition systems are strongly bisimilar according to the standard definition (cf. [BW90]).

Lemma A.1. For all closed terms p, q over $\Sigma(\text{BPA}_{\text{gce}}, \mathbb{B}(A))$ we have that $p \leftrightarroweq^+ q$ iff $p \leftrightarroweq q$.

Proof. A binary relation over the closed process terms is a bisimulation according to Definition 2.3.3 iff it is a bisimulation in the standard sense (regarding $\mathbb{B}(A)^+$ transitions). The following two properties can be used to prove this fact:

1. If $p \xrightarrow{at:\rightarrow a} p'$, then for some ϕ in $\mathbb{B}(A)$, $p \xrightarrow{\phi:\rightarrow a} p'$ and $at \leq f(\phi)$, and likewise for transitions ending in $\surd$;
2. If $p \xrightarrow{\phi:\rightarrow a} p'$, then $p \xrightarrow{at_i:\rightarrow a} p'$ for $f(\phi) = \bigvee_{i\in I} at_i$, and likewise for $\surd$-transitions.

Both these properties follow easily by structural induction.

As an example we show "only if" (the converse statement that a bisimulation according to Definition 2.3.3 is one for the $\mathbb{B}(A)^+$ transitions in the standard sense can be proved similarly).

Suppose pRq for some bisimulation R in the $\mathbb{B}(A)^+$ sense. Observe that the number of states in a transition system (of either type) connected to the root is finite, so we may assume that R is finite.

Now assume $p \xrightarrow{\phi:\rightarrow a} p'$. By property 2 we find at_i such that $p \xrightarrow{at_i:\rightarrow a} p'$ and $f(\phi) = \bigvee_{i\in I} at_i$. By R being a finite bisimulation, there is a *finite* number of different q_{j_i}'s with $q \xrightarrow{at_i:\rightarrow a} q_{j_i}$ and $p'Rq_{j_i}$. By property 1, for each such q_{j_i} there is a Boolean expression ψ_{j_i} with $q \xrightarrow{\psi_{j_i}:\rightarrow a} q_{j_i}$ and $at_i \leq f(\psi_{j_i})$. Because f is an embedding, it follows from $f(\phi) = \bigvee_{i\in I} at_i \leq \bigvee_{i\in I} f(\psi_{j_i})$ that $\phi \leq \bigvee_{i\in I} \psi_{j_i}$. The remaining three clauses of Definition 2.3.3 follow in the same way. Hence R is also a bisimulation in the sense of Definition 2.3.3. □

Proof of the Elimination Theorem 4.1.7 for BPA($\dotplus$).

1. If $p \equiv \tilde{q} \dotplus \tilde{r}$, with $\tilde{q}$, $\tilde{r}$ basic terms over $\Sigma(\mathrm{BPA_{gce}}, \mathbb{B}(A_{flag}))$, then it can be proved by induction on the structure of $\tilde{q}$. According to Lemma 2.1.2, 5 cases have to be distinguished.
   Case 1. $\tilde{q} \equiv \delta$, trivial.
   Case 2. $\tilde{q} \equiv a$, where $a \in A_{flag}$.
   $$a \dotplus \tilde{r} = enabled(a) :\rightarrow a + \neg enabled(a) :\rightarrow \tilde{r}.$$
   The right hand side of this expression is a basic term.
   Case 3. $\tilde{q} \equiv a \cdot q'$, where $a \in A_{flag}$. By Lemma 2.1.2 q' is also a basic term, smaller than $\tilde{q}$. Induction hypothesis: Theorem 4.1.7.1 holds for q'.
   $$\begin{aligned} a \cdot q' \dotplus \tilde{r} = \ & invertible(a) :\rightarrow Flag(a) \cdot (q' \dotplus Undo(a) \cdot \tilde{r}) + \\ & pass(a) :\rightarrow a \cdot (q' \dotplus \tilde{r}) + \\ & commit(a) :\rightarrow a \cdot q' + \\ & \neg enabled(a) :\rightarrow \tilde{r}. \end{aligned}$$
   By induction $q' \dotplus Undo(a) \cdot \tilde{r}$ and $q' \dotplus \tilde{r}$ are provably equal to basic terms. Using Lemma 2.1.2 we see that the right hand side of $a \cdot q' \dotplus \tilde{r}$ is equal to a basic term $\tilde{p}$.
   Case 4. $\tilde{q} \equiv q_1 + q_2$. Both q_1 and q_2 are basic terms, smaller than $\tilde{q}$ by Lemma 2.1.2. Induction hypothesis: Theorem 4.1.7.1 holds for q_1 and q_2.
   $$\begin{aligned} (q_1 + q_2) \dotplus \tilde{r} = \ & enabled(q_1) :\rightarrow q_1 \dotplus \tilde{r} + enabled(q_2) :\rightarrow q_2 \dotplus \tilde{r} + \\ & \neg enabled(q_1) \wedge \neg enabled(q_2) :\rightarrow \tilde{r}. \end{aligned}$$
   By induction $q_1 \dotplus \tilde{r}$ and $q_2 \dotplus \tilde{r}$ are provably equal to basic terms. According to Lemma 2.1.2, the right hand side of $(q_1 + q_2) \dotplus \tilde{r}$ equals a basic term.
   Case 5. $\tilde{q} \equiv \phi :\rightarrow q'$. According to Lemma 2.1.2, q' is also a basic term, smaller than $\tilde{q}$. Induction hypothesis: Theorem 4.1.7.1 holds for q'.
   $$(\phi :\rightarrow q') \dotplus \tilde{r} = \phi :\rightarrow q' \dotplus \tilde{r} + \neg\phi :\rightarrow \tilde{r}.$$
   By induction $q' \dotplus \tilde{r}$ is provably equal to a basic term. According to Lemma 2.1.2, the right hand side of $(\phi :\rightarrow q') \dotplus \tilde{r}$ equals a basic term.
   Finally, if $p \equiv q \dotplus r$ with q and r closed terms over $\Sigma(\mathrm{BPA}(\dotplus))$, it has to be proved that there is a basic term $\tilde{p}$ such that $p = \tilde{p}$. Induction hypothesis:
   $$\mathrm{BPA}(\dotplus) \vdash q = \tilde{q} \ \text{ and } \ r = \tilde{r}.$$
   So $p = \tilde{q} \dotplus \tilde{r}$. Using the results above $\tilde{q} \dotplus \tilde{r}$ can be represented by a basic term $\tilde{p}$ such that $p = \tilde{p}$. This finishes the proof.
2. Standard.

□

Proof of Lemma 4.2.1.

1.

$$\begin{aligned} \phi :\rightarrow (\phi :\rightarrow x) \mathbin{\not+} y &= \phi :\rightarrow (\phi :\rightarrow x \mathbin{\not+} y + \neg\phi :\rightarrow y) \\ &= \phi \wedge \phi :\rightarrow x \mathbin{\not+} y + \phi \wedge \neg\phi :\rightarrow y \\ &= \phi :\rightarrow x \mathbin{\not+} y. \end{aligned}$$

2.

$$\begin{aligned} (\phi :\rightarrow x + \psi :\rightarrow y) \mathbin{\not+} z \quad &= \quad enabled(\phi :\rightarrow x) :\rightarrow (\phi :\rightarrow x) \mathbin{\not+} z + \\ &\qquad enabled(\psi :\rightarrow y) :\rightarrow (\psi :\rightarrow y) \mathbin{\not+} z + \\ &\qquad \neg enabled(\phi :\rightarrow x) \wedge \neg enabled(\psi :\rightarrow y) :\rightarrow z \\ \\ &= \quad \phi \wedge enabled(x) :\rightarrow (\phi :\rightarrow x) \mathbin{\not+} z + \\ &\qquad \psi \wedge enabled(y) :\rightarrow (\psi :\rightarrow y) \mathbin{\not+} z + \\ &\qquad \neg(\phi \wedge enabled(x)) \wedge \neg(\psi \wedge enabled(y)) :\rightarrow z \\ \\ &\overset{4.2.1.1}{=} \quad \phi \wedge enabled(x) :\rightarrow x \mathbin{\not+} z + \\ &\qquad \psi \wedge enabled(y) :\rightarrow y \mathbin{\not+} z + \\ &\qquad (\neg\phi \vee \neg enabled(x)) \wedge (\neg\psi \vee \neg enabled(y)) :\rightarrow z. \end{aligned}$$

3.

$$\begin{aligned} x \mathbin{\not+} y &= (x + x) \mathbin{\not+} y \\ \\ &= \quad enabled(x) :\rightarrow x \mathbin{\not+} y + enabled(x) :\rightarrow x \mathbin{\not+} y + \\ &\qquad \neg enabled(x) \wedge \neg enabled(x) :\rightarrow y \\ \\ &= \quad enabled(x) :\rightarrow x \mathbin{\not+} y + \neg enabled(x) :\rightarrow y. \end{aligned}$$

4.

$$\begin{aligned} (\phi :\rightarrow x + \neg\phi :\rightarrow y) \mathbin{\not+} z \quad &\overset{4.2.1.2}{=} \quad \phi \wedge enabled(x) :\rightarrow x \mathbin{\not+} z + \\ &\qquad \neg\phi \wedge enabled(y) :\rightarrow y \mathbin{\not+} z + \\ &\qquad \phi \wedge \neg enabled(x) :\rightarrow z + \neg\phi \wedge \neg enabled(y) :\rightarrow z + \\ &\qquad \neg enabled(x) \wedge \neg enabled(y) :\rightarrow z \\ \\ &= \quad \phi :\rightarrow (enabled(x) :\rightarrow x \mathbin{\not+} z + \\ &\qquad \neg enabled(x) \vee (\neg enabled(x) \wedge \neg enabled(y)) :\rightarrow z) + \\ &\qquad \neg\phi :\rightarrow (enabled(y) :\rightarrow y \mathbin{\not+} z + \\ &\qquad \neg enabled(y) \vee (\neg enabled(x) \wedge \neg enabled(y)) :\rightarrow z) \\ \\ &= \quad \phi :\rightarrow (enabled(x) :\rightarrow x \mathbin{\not+} z + \neg enabled(x) :\rightarrow z) + \\ &\qquad \neg\phi :\rightarrow (enabled(y) :\rightarrow y \mathbin{\not+} z + \neg enabled(y) :\rightarrow z) \\ \\ &\overset{4.2.1.3}{=} \quad \phi :\rightarrow x \mathbin{\not+} z + \neg\phi :\rightarrow y \mathbin{\not+} z. \end{aligned}$$

□

Proof of the Enabledness Theorem 4.2.2. By induction on the structure of p. According to the Elimination Theorem 4.1.7, we have to distinguish five cases. Let q, q_1 and q_2 be basic terms over $\Sigma(\mathrm{BPA_{gce}}, \mathbb{B}(A_{flag}))$, and $a \in A_{flag}$.

Case 1. $p \equiv \delta$, trivial.

Case 2. $p \equiv a$.

$$\begin{aligned} \mathit{enabled}(a \dotplus y) &= \mathit{enabled}(\mathit{enabled}(a) :\rightarrow a + \neg \mathit{enabled}(a) :\rightarrow y) \\ &= \mathit{enabled}(a) \vee (\neg \mathit{enabled}(a) \wedge \mathit{enabled}(y)) \\ &= \mathit{enabled}(a) \vee \mathit{enabled}(y). \end{aligned}$$

Case 3. $p \equiv a \cdot q$.

$$\begin{aligned} \mathit{enabled}(a \cdot q \dotplus y) = {} & \mathit{enabled}(\mathit{pass}(a) :\rightarrow a \cdot (q \dotplus y) + \\ & \mathit{invertible}(a) :\rightarrow \mathit{Flag}(a) \cdot (q \dotplus \mathit{Undo}(a) \cdot y) + \\ & \mathit{commit}(a) :\rightarrow a \cdot q + \\ & \neg \mathit{enabled}(a) :\rightarrow y) \\ \\ = {} & (\mathit{invertible}(a) \wedge \mathit{enabled}(\mathit{Flag}(a))) \vee \ (\mathit{pass}(a) \wedge \mathit{enabled}(a)) \vee \\ & (\mathit{commit}(a) \wedge \mathit{enabled}(a)) \vee (\neg \mathit{enabled}(a) \wedge \mathit{enabled}(y)) \\ \\ = {} & \mathit{invertible}(a) \vee \mathit{pass}(a) \vee \mathit{commit}(a) \vee \ (\neg \mathit{enabled}(a) \wedge \mathit{enabled}(y)) \\ = {} & \mathit{enabled}(a) \vee \mathit{enabled}(y) \\ = {} & \mathit{enabled}(a \cdot q) \vee \mathit{enabled}(y). \end{aligned}$$

Case 4. $p \equiv q_1 + q_2$, induction hypothesis: Theorem 4.2.2 holds for q_1 and q_2.

$$\begin{aligned} \mathit{enabled}((q_1 + q_2) \dotplus y) \quad &= \quad \mathit{enabled}(\mathit{enabled}(q_1) :\rightarrow q_1 \dotplus y + \\ & \qquad \mathit{enabled}(q_2) :\rightarrow q_1 \dotplus y + \\ & \qquad \neg \mathit{enabled}(q_1) \wedge \neg \mathit{enabled}(q_2) :\rightarrow y) \\ \\ &= \quad \mathit{enabled}(\mathit{enabled}(q_1) :\rightarrow q_1 \dotplus y) \vee \\ & \qquad \mathit{enabled}(\mathit{enabled}(q_2) :\rightarrow q_2 \dotplus y) \vee \\ & \qquad \mathit{enabled}(\neg \mathit{enabled}(q_1) \wedge \neg \mathit{enabled}(q_2) :\rightarrow y) \\ \\ &= \quad (\mathit{enabled}(q_1) \wedge \mathit{enabled}(q_1 \dotplus y)) \vee \\ & \qquad (\mathit{enabled}(q_2) \wedge \mathit{enabled}(q_2 \dotplus y)) \vee \\ & \qquad (\neg \mathit{enabled}(q_1) \wedge \neg \mathit{enabled}(q_2) \wedge \mathit{enabled}(y)) \\ \\ &\overset{Ind.Hyp.}{=} \quad (\mathit{enabled}(q_1) \wedge (\mathit{enabled}(q_1) \vee \mathit{enabled}(y))) \vee \\ & \qquad (\mathit{enabled}(q_2) \wedge (\mathit{enabled}(q_2) \vee \mathit{enabled}(y))) \vee \\ & \qquad (\neg \mathit{enabled}(q_1) \wedge \neg \mathit{enabled}(q_2) \wedge \mathit{enabled}(y)) \\ \\ &= \quad \mathit{enabled}(q_1) \vee \mathit{enabled}(q_2) \vee \\ & \qquad (\neg(\mathit{enabled}(q_1) \vee \mathit{enabled}(q_2)) \wedge \mathit{enabled}(y)) \\ \\ &= \quad \mathit{enabled}(q_1) \vee \mathit{enabled}(q_2) \vee \mathit{enabled}(y) \\ &= \quad \mathit{enabled}(q_1 + q_2) \vee \mathit{enabled}(y). \end{aligned}$$

Case 5. $p \equiv \phi :\rightarrow q$, induction hypothesis: Theorem 4.2.2 holds for q.

$$\begin{array}{rcl}
enabled((\phi :\rightarrow q) \not+ y) & = & enabled(\phi :\rightarrow q \not+ y + \neg\phi :\rightarrow y) \\
& = & enabled(\phi :\rightarrow q \not+ y) \vee enabled(\neg\phi :\rightarrow y) \\
& = & (\phi \wedge enabled(q \not+ y)) \vee (\neg\phi \wedge enabled(y)) \\
& \overset{Ind.Hyp.}{=} & (\phi \wedge (enabled(q) \vee enabled(y))) \vee (\neg\phi \wedge enabled(y)) \\
& = & (\phi \wedge enabled(q)) \vee (\phi \wedge enabled(y)) \vee (\neg\phi \wedge enabled(y)) \\
& = & enabled(\phi :\rightarrow q) \vee enabled(y).
\end{array}$$

□

Proof of Example 4.2.3. We prove the two identities of Example 4.2.3 in reversed order.

$$\begin{array}{l}
a \cdot undo(b) \cdot \delta \not+ (\delta \not+ c) = a \cdot undo(b) \cdot \delta \not+ c \\
\quad = \neg enabled(a) :\rightarrow c + enabled(a) :\rightarrow a \cdot undo(b) \cdot \delta \not+ c \\
\quad = \neg enabled(a) :\rightarrow c + enabled(a) :\rightarrow flag(a) \cdot (undo(b) \cdot \delta \not+ undo(a) \cdot c) \\
\quad = \neg enabled(a) :\rightarrow c + enabled(a) :\rightarrow flag(a) \cdot undo(b) \cdot (\delta \not+ undo(a) \cdot c) \\
\quad = \neg enabled(a) :\rightarrow c + enabled(a) :\rightarrow flag(a) \cdot undo(b) \cdot undo(a) \cdot c.
\end{array}$$

The term $a \cdot undo(b) \cdot \delta \not+ \delta$ occurring in the following identity can be treated analogously.

$$\begin{array}{rl}
\multicolumn{2}{l}{(a \cdot undo(b) \cdot \delta \not+ \delta) \not+ c} \\
= & (\neg enabled(a) :\rightarrow \delta + enabled(a) :\rightarrow flag(a) \cdot undo(b) \cdot undo(a) \cdot \delta) \not+ c \\
\overset{4.2.1.4}{=} & \neg enabled(a) :\rightarrow \delta \not+ c + enabled(a) :\rightarrow flag(a) \cdot undo(b) \cdot undo(a) \cdot \delta \not+ c \\
= & \neg enabled(a) :\rightarrow c + enabled(a) :\rightarrow flag(a) \cdot undo(b) \cdot (undo(a) \cdot \delta \not+ c) \\
= & \neg enabled(a) :\rightarrow c + enabled(a) :\rightarrow flag(a) \cdot undo(b) \cdot \\
& \qquad (enabled(undo(a)) :\rightarrow undo(a) \cdot c + \neg enabled(undo(a)) :\rightarrow c) \\
= & \neg enabled(a) :\rightarrow c + enabled(a) :\rightarrow flag(a) \cdot undo(b) \cdot \\
& \qquad (undo(a) \cdot c + \neg enabled(undo(a)) :\rightarrow c).
\end{array}$$

□

Two lemmas for a proof of the associativity of the $\not+$ operator. Below, two lemmas on backtracking in a restricted signature $\Sigma(\mathrm{BPA}(\not+))^-$ are proved. Both are needed for a proof of the associativity of the $\not+$ operator.

Lemma A.2. In a restricted signature $\Sigma(\mathrm{BPA}(\not+))^-$ the following identity holds, where $a \in A_{flag}$.

$$Invertible(a) :\rightarrow Undo(a) \cdot (y \not+ z) = Invertible(a) :\rightarrow (Undo(a) \cdot y \not+ z).$$

Proof.
By case distinction.
<u>Case 1</u>. $\mathit{Invertible}(a) = \mathbf{false}$, trivial.
<u>Case 2</u>. $\mathit{Invertible}(a) = \mathbf{true}$, so $\mathit{Undo}(a) = \mathit{undo}(a)$, and from Lemma 4.2.5 we have $\mathit{pass}(\mathit{undo}(a)) = \mathbf{true}$. It follows immediately that

$$\mathit{Invertible}(a) :\rightarrow \mathit{undo}(a) \cdot (y \dotplus z) = \mathit{pass}(\mathit{undo}(a)) :\rightarrow \mathit{undo}(a) \cdot (y \dotplus z) + \mathbf{false} :\rightarrow z.$$

From Definition 4.2.4 it follows that

$$\mathit{Invertible}(\mathit{undo}(a)) \wedge \neg \mathit{enabled}(\mathit{undo}(a)) = \mathbf{false},$$

so

$$\begin{aligned}\mathit{Invertible}(a) :\rightarrow \mathit{undo}(a) \cdot (y \dotplus z) = {} & \mathit{pass}(\mathit{undo}(a)) :\rightarrow \mathit{undo}(a) \cdot (y \dotplus z) + \\ & \mathit{Invertible}(\mathit{undo}(a)) \wedge \neg \mathit{enabled}(\mathit{undo}(a)) :\rightarrow z.\end{aligned}$$

By Lemma 4.2.5 we also have

$$\mathit{invertible}(\mathit{undo}(a)) = \mathbf{false} \quad \text{and} \quad \mathit{commit}(\mathit{undo}(a)) = \mathbf{false}.$$

From axiom Ba3 it follows then that

$$\begin{aligned}\mathit{Invertible}(a) :\rightarrow \mathit{undo}(a) \cdot (y \dotplus z) &= \mathit{pass}(\mathit{undo}(a)) :\rightarrow \\ & \quad (\mathit{pass}(\mathit{undo}(a)) :\rightarrow \mathit{undo}(a) \cdot (y \dotplus z) + \\ & \quad \neg \mathit{enabled}(\mathit{undo}(a)) :\rightarrow z) \\ &= \mathit{Invertible}(a) :\rightarrow (\mathit{undo}(a) \cdot y \dotplus z).\end{aligned}$$

□

Using the property of predicates $\phi \in \{\mathbf{true}, \mathbf{false}\}$ we can prove the following lemma.

Lemma A.3. In a restricted signature $\Sigma(\mathrm{BPA}(\dotplus))^-$ the following identity holds, where $a \in A_{flag}$.

$$\begin{aligned}a \cdot x \dotplus (y \dotplus z) = {} & \mathit{invertible}(a) :\rightarrow \mathit{Flag}(a) \cdot (x \dotplus (\mathit{Undo}(a) \cdot y \dotplus z)) + \\ & \mathit{pass}(a) :\rightarrow a \cdot (x \dotplus (y \dotplus z)) + \\ & \mathit{commit}(a) :\rightarrow a \cdot x + \\ & \neg \mathit{enabled}(a) :\rightarrow y \dotplus z.\end{aligned}$$

Proof.
By case distinction.
<u>Case 1</u>. $\mathit{Invertible}(a) = \mathbf{false}$, trivial.
<u>Case 2</u>. $\mathit{Invertible}(a) = \mathbf{true}$, so $\mathit{Flag}(a) = \mathit{flag}(a)$ and $\mathit{Undo}(a) = \mathit{undo}(a)$.
After application of Ba3 to $a \cdot x \dotplus (y \dotplus z)$ we find that only the first summand in not in the desired form yet:

$$\mathit{invertible}(a) :\rightarrow \mathit{flag}(a) \cdot (x \dotplus \mathit{undo}(a) \cdot (y \dotplus z)).$$

Using the definition of *invertible* and the property of predicates $\phi \in \{\mathbf{true}, \mathbf{false}\}$ we insert $\mathit{Invertible}(a)$ in this expression:

$$\mathit{invertible}(a) :\rightarrow \mathit{flag}(a) \cdot (x \dotplus (\mathit{Invertible}(a) :\rightarrow \mathit{undo}(a) \cdot (y \dotplus z))).$$

Application of Lemma A.2 and subsequently again the property of predicates $\phi \in \{\mathbf{true}, \mathbf{false}\}$ finishes the proof. □

Proof of the Associativity Theorem 4.2.6. By induction on the structure of p. According to the Elimination Theorem 4.1.7, we have to distinguish five cases. Let q, q_1 and q_2 be basic terms over $\Sigma(\mathrm{BPA}_{\mathrm{gce}}, \mathbb{B}(A_{flag}))$, and $a \in A_{flag}$.
<u>Case 1</u>. $p \equiv \delta$, trivial.
<u>Case 2</u>. $p \equiv a$.

$$
\begin{aligned}
(a \mathbin{\triangleleft\!\!\!+} y) \mathbin{\triangleleft\!\!\!+} z &= (\mathit{enabled}(a) :\rightarrow a + \neg \mathit{enabled}(a) :\rightarrow y) \mathbin{\triangleleft\!\!\!+} z \\
&\overset{4.2.1.4}{=} \mathit{enabled}(a) :\rightarrow a \mathbin{\triangleleft\!\!\!+} z + \neg \mathit{enabled}(a) :\rightarrow y \mathbin{\triangleleft\!\!\!+} z \\
&= \mathit{enabled}(a) :\rightarrow a + \neg \mathit{enabled}(a) :\rightarrow y \mathbin{\triangleleft\!\!\!+} z \\
&= a \mathbin{\triangleleft\!\!\!+} (y \mathbin{\triangleleft\!\!\!+} z).
\end{aligned}
$$

<u>Case 3</u>. $p \equiv a \cdot q$, induction hypothesis: associativity holds for q.

$$
\begin{aligned}
(a \cdot q \mathbin{\triangleleft\!\!\!+} y) \mathbin{\triangleleft\!\!\!+} z &= (\mathit{invertible}(a) :\rightarrow \mathit{Flag}(a) \cdot (q \mathbin{\triangleleft\!\!\!+} \mathit{Undo}(a) \cdot y) + \\
&\quad\ \mathit{pass}(a) :\rightarrow a \cdot (q \mathbin{\triangleleft\!\!\!+} y) + \\
&\quad\ \mathit{commit}(a) :\rightarrow a \cdot q + \neg \mathit{enabled}(a) :\rightarrow y) \mathbin{\triangleleft\!\!\!+} z \\[1ex]
&= \mathit{invertible}(a) \wedge \mathit{enabled}(\mathit{Flag}(a)) :\rightarrow \\
&\quad\ \ \mathit{Flag}(a) \cdot (q \mathbin{\triangleleft\!\!\!+} \mathit{Undo}(a) \cdot y) \mathbin{\triangleleft\!\!\!+} z + \\
&\quad \mathit{pass}(a) \wedge \mathit{enabled}(a) :\rightarrow (a \cdot (q \mathbin{\triangleleft\!\!\!+} y)) \mathbin{\triangleleft\!\!\!+} z + \\
&\quad \mathit{commit}(a) \wedge \mathit{enabled}(a) :\rightarrow a \cdot q \mathbin{\triangleleft\!\!\!+} z + \\
&\quad \neg \mathit{enabled}(a) \wedge \mathit{enabled}(y) :\rightarrow y \mathbin{\triangleleft\!\!\!+} z + \\
&\quad \neg(\mathit{pass}(a) \wedge \mathit{enabled}(a)) \wedge \\
&\quad \neg(\mathit{commit}(a) \wedge \mathit{enabled}(a)) \wedge \\
&\quad \neg(\mathit{invertible}(a) \wedge \mathit{enabled}(\mathit{Flag}(a))) \wedge \\
&\quad \neg(\neg \mathit{enabled}(a) \wedge \mathit{enabled}(y)) :\rightarrow z \\[1ex]
&= \mathit{invertible}(a) :\rightarrow \mathit{Flag}(a) \cdot ((q \mathbin{\triangleleft\!\!\!+} \mathit{Undo}(a) \cdot y) \mathbin{\triangleleft\!\!\!+} z) + \\
&\quad \mathit{pass}(a) :\rightarrow a \cdot ((q \mathbin{\triangleleft\!\!\!+} y) \mathbin{\triangleleft\!\!\!+} z) + \\
&\quad \mathit{commit}(a) :\rightarrow a \cdot q + \\
&\quad \neg \mathit{enabled}(a) \wedge \mathit{enabled}(y) :\rightarrow y \mathbin{\triangleleft\!\!\!+} z + \\
&\quad \neg(\mathit{pass}(a) \vee \mathit{commit}(a) \vee \mathit{invertible}(a) \vee \\
&\quad (\neg \mathit{enabled}(a) \wedge \mathit{enabled}(y))) :\rightarrow z \\[1ex]
&= \mathit{invertible}(a) :\rightarrow \mathit{Flag}(a) \cdot ((q \mathbin{\triangleleft\!\!\!+} \mathit{Undo}(a) \cdot y) \mathbin{\triangleleft\!\!\!+} z) + \\
&\quad \mathit{pass}(a) :\rightarrow a \cdot ((q \mathbin{\triangleleft\!\!\!+} y) \mathbin{\triangleleft\!\!\!+} z) + \\
&\quad \mathit{commit}(a) :\rightarrow a \cdot q + \\
&\quad \neg \mathit{enabled}(a) \wedge \mathit{enabled}(y) :\rightarrow y \mathbin{\triangleleft\!\!\!+} z + \\
&\quad \neg \mathit{enabled}(a) \wedge \neg \mathit{enabled}(y) :\rightarrow z \\[1ex]
&\overset{4.2.1.3}{=} \mathit{invertible}(a) :\rightarrow \mathit{Flag}(a) \cdot ((q \mathbin{\triangleleft\!\!\!+} \mathit{Undo}(a) \cdot y) \mathbin{\triangleleft\!\!\!+} z) + \\
&\quad \mathit{pass}(a) :\rightarrow a \cdot ((q \mathbin{\triangleleft\!\!\!+} y) \mathbin{\triangleleft\!\!\!+} z) + \\
&\quad \mathit{commit}(a) :\rightarrow a \cdot q + \\
&\quad \neg \mathit{enabled}(a) :\rightarrow y \mathbin{\triangleleft\!\!\!+} z
\end{aligned}
$$

$$
\begin{aligned}
&\overset{Ind.Hyp.}{=} && invertible(a) :\rightarrow Flag(a) \cdot (q \triangleleft\!\!\!+ (Undo(a) \cdot y \triangleleft\!\!\!+ z)) + \\
& && pass(a) :\rightarrow a \cdot (q \triangleleft\!\!\!+ (y \triangleleft\!\!\!+ z)) + \\
& && commit(a) :\rightarrow a \cdot q + \\
& && \neg enabled(a) :\rightarrow y \triangleleft\!\!\!+ z \\
&\overset{A.3}{=} && a \cdot q \triangleleft\!\!\!+ (y \triangleleft\!\!\!+ z).
\end{aligned}
$$

<u>Case 4</u>. $p \equiv q_1 + q_2$, induction hypothesis: associativity holds for q_1 and q_2.

$$
\begin{aligned}
((q_1 + q_2) \triangleleft\!\!\!+ y) \triangleleft\!\!\!+ z = {} & (enabled(q_1) :\rightarrow q_1 \triangleleft\!\!\!+ y + enabled(q_2) :\rightarrow q_2 \triangleleft\!\!\!+ y + \\
& \neg enabled(q_1) \wedge \neg enabled(q_2) :\rightarrow y) \triangleleft\!\!\!+ z
\end{aligned}
$$

$$
\begin{aligned}
&= && enabled(enabled(q_1) :\rightarrow q_1 \triangleleft\!\!\!+ y) :\rightarrow (enabled(q_1) :\rightarrow q_1 \triangleleft\!\!\!+ y) \triangleleft\!\!\!+ z + \\
& && enabled(enabled(q_2) :\rightarrow q_2 \triangleleft\!\!\!+ y) :\rightarrow (enabled(q_2) :\rightarrow q_2 \triangleleft\!\!\!+ y) \triangleleft\!\!\!+ z + \\
& && enabled(\neg enabled(q_1) \wedge \neg enabled(q_2) :\rightarrow y) :\rightarrow \\
& && (\neg enabled(q_1) \wedge \neg enabled(q_2) :\rightarrow y) \triangleleft\!\!\!+ z + \\
& && \neg enabled(enabled(q_1) :\rightarrow q_1 \triangleleft\!\!\!+ y) \wedge \\
& && \neg enabled(enabled(q_2) :\rightarrow q_2 \triangleleft\!\!\!+ y) \wedge \\
& && \neg enabled(\neg enabled(q_1) \wedge \neg enabled(q_2) :\rightarrow y) :\rightarrow z \\
&\overset{4.2.2\ \&\ 4.2.1.1}{=} && enabled(q_1) :\rightarrow (q_1 \triangleleft\!\!\!+ y) \triangleleft\!\!\!+ z + \\
& && enabled(q_2) :\rightarrow (q_2 \triangleleft\!\!\!+ y) \triangleleft\!\!\!+ z + \\
& && \neg enabled(q_1) \wedge \neg enabled(q_2) \wedge enabled(y) :\rightarrow y \triangleleft\!\!\!+ z + \\
& && \neg enabled(q_1) \wedge \neg enabled(q_2) \wedge \neg enabled(y) :\rightarrow z \\
&\overset{4.2.1.3}{=} && enabled(q_1) :\rightarrow (q_1 \triangleleft\!\!\!+ y) \triangleleft\!\!\!+ z + \\
& && enabled(q_2) :\rightarrow (q_2 \triangleleft\!\!\!+ y) \triangleleft\!\!\!+ z + \\
& && \neg enabled(q_1) \wedge \neg enabled(q_2) :\rightarrow y \triangleleft\!\!\!+ z \\
&\overset{Ind.Hyp.}{=} && enabled(q_1) :\rightarrow q_1 \triangleleft\!\!\!+ (y \triangleleft\!\!\!+ z) + \\
& && enabled(q_2) :\rightarrow q_2 \triangleleft\!\!\!+ (y \triangleleft\!\!\!+ z) + \\
& && \neg enabled(q_1) \wedge \neg enabled(q_2) :\rightarrow y \triangleleft\!\!\!+ z \\
&= && (q_1 + q_2) \triangleleft\!\!\!+ (y \triangleleft\!\!\!+ z).
\end{aligned}
$$

<u>Case 5</u>. $p \equiv \phi :\rightarrow q$, induction hypothesis: associativity holds for q .

$$
\begin{aligned}
((\phi :\rightarrow q) \triangleleft\!\!\!+ y) \triangleleft\!\!\!+ z &= (\phi :\rightarrow q \triangleleft\!\!\!+ y + \neg\phi :\rightarrow y) \triangleleft\!\!\!+ z \\
&\overset{4.2.1.4}{=} \phi :\rightarrow (q \triangleleft\!\!\!+ y) \triangleleft\!\!\!+ z + \neg\phi :\rightarrow y \triangleleft\!\!\!+ z \\
&\overset{Ind.Hyp.}{=} \phi :\rightarrow q \triangleleft\!\!\!+ (y \triangleleft\!\!\!+ z) + \neg\phi :\rightarrow y \triangleleft\!\!\!+ z \\
&= (\phi :\rightarrow q) \triangleleft\!\!\!+ (y \triangleleft\!\!\!+ z).
\end{aligned}
$$

□

Proof of the Congruence Lemma 4.3.1 (sketch). Because a direct proof seems to be rather complex, application of the congruence result from [BV93] as used in the proof of Lemma 2.3.4 is attractive here. Referring to the idea and terminology of that proof and of the paper mentioned above, the following ingredients are needed:

1. The transition rules for the state operators (see Table 9) must have associated versions in the $\mathbb{B}(A)^+$ setting, and the properties 1 and 2 defined in the proof of Lemma A.1 must be satisfied.
2. For any $\phi \in \mathbb{B}(A)$, the unary predicate

$$\phi \wedge \mathit{enabled}(.) = \mathbf{false} \;\subseteq\; \Sigma(\mathrm{BPA}(\dotplus))^{\lambda,\mathcal{S}}.$$

   must be definable in the *path* format.

For the first ingredient, a function $\mathit{eval}^+ : \mathbb{B}(A)^+ \times S \to \mathbb{B}(A)^+$ with the property

$$\mathit{eval}^+(f(\phi), s) = f(\mathit{eval}(\phi, s)) \quad \text{for all } \phi \in \mathbb{B}(A), s \in S$$

must be assumed (cf. the definition of the function *eval* in Table 7). The transition rules for the state operators then become

$$\frac{x \xrightarrow{at:\to a} x'}{\lambda_s(x) \xrightarrow{\mathit{eval}^+(at,s):\to \mathit{action}(a,s)} \lambda_{\mathit{effect}(a,s)}(x')}$$

if $\mathit{eval}^+(at \wedge f(\mathit{enabled}(a)), s) \neq \mathbf{false}$, and likewise for the case $x' \equiv \surd$ and $\lambda_{\mathit{effect}(a,s)}(x') \equiv \surd$. These rules indeed satisfy the *path* format defined in [BV93]. Moreover, it is not hard to prove that the transition rules for the $\mathbb{B}(A)^+$ format defined thus far satisfy the properties 1 and 2 referred to above.

As for the second ingredient, let P_{at} abbreviate the predicate $at \wedge f(\mathit{enabled}(.)) = \mathbf{false}$. In Table 15 we define for each atom at in $\mathbb{B}(A)^+$ the predicate P_{at} in *path* format.

$P_{at}(\delta)$	
$P_{at}(a)$	if $at \wedge f(\mathit{enabled}(a)) = \mathbf{false}$
$P_{at}(x+y)$	if $P_{at}(x)$ and $P_{at}(y)$
$P_{at}(x \cdot y)$	if $P_{at}(x)$
$P_{at}(\phi :\to x)$	if $P_{at}(x)$
$P_{at}(\phi :\to x)$	if $at \not\leq f(\phi)$

Table 15. The predicates for "not enabledness under *at*", where $a \in A$, $\phi \in \mathcal{B}$ and at in $\mathbb{B}(A)^+$.

It is evident that $\phi \wedge \mathit{enabled}(x) = \mathbf{false} \iff \bigvee_{i\in I} P_{at_i}(x)$ for $f(\phi) = \bigvee_{i \in I} at_i$. Now the first rule for the try operator becomes

$$\frac{y \xrightarrow{at:\to b} y' \quad P_{at}(x)}{x \dotplus y \xrightarrow{at:\to b} y'}$$

and the second rule is adapted likewise with $y' \equiv \surd$. If we change the remaining rules as in the proof of Lemma 2.3.4, and let ϕ in Table 11 range over the atoms of $\mathbb{B}(A)^+$, all transition rules for the $\mathbb{B}(A)^+$ setting are defined in *path* format. A subtlety is that according to [BV93], the bisimilarity of p and q thus obtained, say 'not-enabledness bisimilarity under Boolean atoms', covers the property that $P_{at}(p) \iff P_{at}(q)$. Regarding the meaning of P_{at}, this type of bisimilarity is not really different from our notion of bisimilarity, because this property means that p has no outgoing transitions (with some abuse of notation: $P_{\mathbf{true}}(p)$) iff q has none. Just as in the proof of Lemma 2.3.4, the congruence result of [BV93] can be used to prove the lemma. $\square$

Proof of Theorem 4.3.4. Requirements III and IV follow immediately from Ba2 and Ba3, respectively. Requirements I and II can be proved simultaneously by induction on the structure of the first argument of the $\dotplus$ operator, which we may assume to be a basic term over $\Sigma(\text{BPA}_{\text{gce}}, \mathbb{B}(A_{flag}))$. We will not give a complete proof because of its length. Instead we give a proof for one of the induction steps.

Let $a \in A_{flag}$, p be a basic term over $\Sigma(\text{BPA}_{\text{gce}}, \mathbb{B}(A_{flag}))$, and $s \in \mathcal{S}_{flag}$. We prove that $a \cdot p$ satisfies Requirements I and II if p does. We assume that $\forall \sigma \in str(a \cdot p, s) \,.\, \neg pass(\sigma, s)$ holds. This implies that if $enabled(a, s) = \mathbf{true}$ then $\forall \rho \in str(p, effect(a, s)) . \neg pass(\rho, effect(a, s))$ also holds. We distinguish the following cases:
<u>Case 1</u>. $Invertible(a) = \mathbf{true}$. With Ba3 we find

$$\begin{aligned}
[\![a \cdot p \dotplus y]\!](s) &= [\![enabled(a) :\rightarrow flag(a) \cdot (p \dotplus undo(a) \cdot y)]\!](s) \cup \\
&\quad\; [\![\neg enabled(a) :\rightarrow y]\!](s) \\
&= \begin{cases} [\![p \dotplus undo(a) \cdot y]\!](effect(a, s)) & \text{if } enabled(a, s) = \mathbf{true} \\ [\![y]\!](s) & \text{otherwise.} \end{cases}
\end{aligned}$$

<u>Case 1.1</u>. $enabled(a, s) = \mathbf{false}$. Then $fail(a \cdot p, s) = \mathbf{true}$, so Requirement I follows trivially. Furthermore $[\![a \cdot p \dotplus y]\!](s) = [\![y]\!](s)$, and because $[\![a \cdot p]\!](s) = \emptyset$ in this case, we find

$$[\![a \cdot p \dotplus y]\!](s) = [\![a \cdot p]\!](s) \;\cup\; [\![y]\!](s),$$

which proves Requirement II.
<u>Case 1.2</u>. $enabled(a, s) = \mathbf{true} \wedge fail(a \cdot p, s) = \mathbf{true}$. Now Requirement I follows trivially and $fail(p, effect(a, s)) = \mathbf{true}$ in this case. By induction we have

$$[\![p \dotplus undo(a) \cdot y]\!](effect(a, s)) = [\![p]\!](effect(a, s)) \cup [\![undo(a) \cdot y]\!](effect(a, s)).$$

We derive

$$\begin{aligned}
[\![a \cdot p \dotplus y]\!](s) &\stackrel{\text{Ba3}}{=} [\![p \dotplus undo(a) \cdot y]\!](effect(a, s)) \\
&\stackrel{Ind.Hyp.}{=} [\![p]\!](effect(a, s)) \;\cup\; [\![undo(a) \cdot y]\!](effect(a, s)) \\
&= [\![a \cdot p]\!](s) \;\cup\; [\![y]\!](effect(undo(a), effect(a, s))) \\
&= [\![a \cdot p]\!](s) \;\cup\; [\![y]\!](s),
\end{aligned}$$

which proves Requirement II.
<u>Case 1.3</u>. $enabled(a, s) = \mathbf{true} \wedge fail(a \cdot p, s) = \mathbf{false}$. Then Requirement II follows trivially, and $fail(p, effect(a, s)) = \mathbf{false}$. By induction we have $[\![p \dotplus undo(a) \cdot$

$y\rrbracket(\mathit{effect}(a,s)) = \llbracket p\rrbracket(\mathit{effect}(a,s))$, which equals $\llbracket a \cdot p\rrbracket(s)$ by definition in this case. This proves Requirement I.
Case 2. $\mathit{Commit}(a) = \mathbf{true}$. With Ba3 we find

$$\llbracket a \cdot p \dotplus y\rrbracket(s) = \llbracket \mathit{enabled}(a) :\rightarrow a \cdot p\rrbracket(s) \cup \llbracket \neg \mathit{enabled}(a) :\rightarrow y\rrbracket(s)$$

$$= \begin{cases} \llbracket p\rrbracket(\mathit{effect}(a,s)) & \text{if } \mathit{enabled}(a,s) = \mathbf{true} \\ \llbracket y\rrbracket(s) & \text{otherwise.} \end{cases}$$

Case 2.1. $\mathit{enabled}(a,s) = \mathbf{false}$. As case 1.1.
Case 2.2. $\mathit{enabled}(a,s) = \mathbf{true}$. Then $\mathit{fail}(a \cdot p, s) = \mathbf{false}$, so Requirement II follows trivially. By the expansion of $\llbracket a \cdot p \dotplus y\rrbracket(s)$ above, Requirement I also follows trivially in this case. □

Proof of Example 5.2.1.

$$\lambda_{(0,0,0)}(B_1) = put_1 \cdot \lambda_{(1,0,0)}(B_2)$$

$$= put_1 \cdot \mathit{flag}(put_2) \cdot \lambda_{(1,2,0)}(B_3 \dotplus \mathit{undo}(put_2) \cdot B_1)$$

$$= put_1 \cdot \mathit{flag}(put_2) \cdot \mathit{flag}(put_3) \cdot \mathit{write}(1,2,3) \cdot \lambda_{(1,2,3)}((\delta \dotplus \mathit{undo}(put_3) \cdot put_2 \cdot B_3) \dotplus \mathit{undo}(put_2) \cdot B_1)$$

$$= put_1 \cdot \mathit{flag}(put_2) \cdot \mathit{flag}(put_3) \cdot \mathit{write}(1,2,3) \cdot \lambda_{(1,2,3)}(\mathit{undo}(put_3) \cdot put_2 \cdot B_3 \dotplus \mathit{undo}(put_2) \cdot B_1)$$

$$= put_1 \cdot \mathit{flag}(put_2) \cdot \mathit{flag}(put_3) \cdot \mathit{write}(1,2,3) \cdot \mathit{undo}(put_3) \cdot \lambda_{(1,2,0)}(put_2 \cdot B_3 \dotplus \mathit{undo}(put_2) \cdot B_1)$$

$$= put_1 \cdot \mathit{flag}(put_2) \cdot \mathit{flag}(put_3) \cdot \mathit{write}(1,2,3) \cdot \mathit{undo}(put_3) \cdot \mathit{flag}(put_2) \cdot \lambda_{(1,3,0)}(B_3 \dotplus \mathit{undo}(put_2) \cdot \mathit{undo}(put_2) \cdot B_1)$$

$$= put_1 \cdot \mathit{flag}(put_2) \cdot \mathit{flag}(put_3) \cdot \mathit{write}(1,2,3) \cdot \mathit{undo}(put_3) \cdot \mathit{flag}(put_2) \cdot \mathit{flag}(put_3) \cdot \mathit{write}(1,3,2) \cdot \lambda_{(1,3,2)}((\delta \dotplus \mathit{undo}(put_3) \cdot put_2 \cdot B_3) \dotplus \mathit{undo}(put_2) \cdot \mathit{undo}(put_2) \cdot B_1)$$

$$= put_1 \cdot \mathit{flag}(put_2) \cdot \mathit{flag}(put_3) \cdot \mathit{write}(1,2,3) \cdot \mathit{undo}(put_3) \cdot \mathit{flag}(put_2) \cdot \mathit{flag}(put_3) \cdot \mathit{write}(1,3,2) \cdot \mathit{undo}(put_3) \cdot \lambda_{(1,3,0)}(put_2 \cdot B_3 \dotplus \mathit{undo}(put_2) \cdot \mathit{undo}(put_2) \cdot B_1)$$

$$= put_1 \cdot \mathit{flag}(put_2) \cdot \mathit{flag}(put_3) \cdot \mathit{write}(1,2,3) \cdot \mathit{undo}(put_3) \cdot \mathit{flag}(put_2) \cdot \mathit{flag}(put_3) \cdot \mathit{write}(1,3,2) \cdot \mathit{undo}(put_3) \cdot \lambda_{(1,3,0)}(\mathit{undo}(put_2) \cdot \mathit{undo}(put_2) \cdot B_1)$$

$$\begin{aligned}
&= put_1 \cdot \mathit{flag}(put_2) \cdot \mathit{flag}(put_3) \cdot write(1,2,3) \cdot undo(put_3) \cdot \mathit{flag}(put_2)\cdot \\
&\quad \mathit{flag}(put_3) \cdot write(1,3,2) \cdot undo(put_3) \cdot undo(put_2)\cdot \\
&\quad \lambda_{(1,2,0)}(undo(put_2) \cdot B_1)
\end{aligned}$$

$$\begin{aligned}
&= put_1 \cdot \mathit{flag}(put_2) \cdot \mathit{flag}(put_3) \cdot write(1,2,3) \cdot undo(put_3) \cdot \mathit{flag}(put_2)\cdot \\
&\quad \mathit{flag}(put_3) \cdot write(1,3,2) \cdot undo(put_3) \cdot undo(put_2) \cdot undo(put_2)\cdot \\
&\quad \lambda_{(1,0,0)}(B_1).
\end{aligned}$$

Analogously we find

$$\begin{aligned}
\lambda_{(1,0,0)}(B_1) &= put_1 \cdot \mathit{flag}(put_2) \cdot \mathit{flag}(put_3) \cdot write(2,1,3) \cdot undo(put_3)\cdot \\
&\quad \mathit{flag}(put_2) \cdot \mathit{flag}(put_3) \cdot write(2,3,1) \cdot undo(put_3) \cdot undo(put_2)\cdot \\
&\quad undo(put_2) \cdot \lambda_{(2,0,0)}(B_1)
\end{aligned}$$

$$\begin{aligned}
\lambda_{(2,0,0)}(B_1) &= put_1 \cdot \mathit{flag}(put_2) \cdot \mathit{flag}(put_3) \cdot write(3,1,2) \cdot undo(put_3)\cdot \\
&\quad \mathit{flag}(put_2) \cdot \mathit{flag}(put_3) \cdot write(3,2,1) \cdot undo(put_3) \cdot undo(put_2)\cdot \\
&\quad undo(put_2) \cdot ready
\end{aligned}$$

□

This article was processed using the LaTeX macro package with LLNCS style