A Semantic Approach to Protocold using Process Algebra

Bas van Vlijmen
Jos van Wamel*
University of Amsterdam, Programming Research Group
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
E-mail: `vlijmen@fwi.uva.nl - vanw@fwi.uva.nl`

Abstract

We study of the operational behaviour of the specification language Protocold. We examine whether BPA($\dotplus$) (Basic Process Algebra with backtracking) forms a useful basis for a clear representation of the execution of certain language constructs from Protocold. Our primary focus is on those constructs that contain some form of backtracking. Backtracking in Protocold contains the undoing of bindings of object names (logical variables).

We first provide a short introduction to both formalisms, and then make a rough comparison of them both. Some elementary Protocold features, such as object names and the binding of object names, are then defined relative to a special data environment for BPA($\dotplus$). Finally, a small language PC_L is defined for the description of the operational behaviour of Protocold-like expressions, as well as a mapping from $\Sigma(\mathrm{PC}_L)$ expressions to $\Sigma(\mathrm{BPA}(\dotplus))$ terms. Some operational equivalences that hold in Protocold are defined as rules in PC_L, such that algebraic manipulation of $\Sigma(\mathrm{PC}_L)$ expressions is possible. Normal forms over $\Sigma(\mathrm{PC}_L)$ are defined, and it turns out that a mapping of these normal forms to $\Sigma(\mathrm{BPA}(\dotplus))$ terms preserves some typical properties of backtracking in PC_L. A canonical mapping does not exist, however, because backtracking in a Protocold (and $\Sigma(\mathrm{PC}_L)$) expression is 'global': it is not limited to the scope of certain operators.

We think that, although the language Protocold is too rich to be fully expressed in terms of BPA($\dotplus$) (particularly where the specification of abstract data types is at stake), the operational behaviour of Protocold specifications can be specified adequately in BPA($\dotplus$).

Key words & Phrases: backtracking, operational semantics, process algebra, specification languages.
1987 CR Categories: D.2, D.3, F.3, I.2.8.

1 Introduction

In the following, introductory text fragments on COLD (Common Object-oriented Language for Design, see e.g. [FJ92b]) and PROTOCOLD are cited from [Jon91a]. "PROTOCOLD is a member of the COLD family of languages. The COLD languages were developed by Philips Research Laboratories Eindhoven with the main aim of making *formal techniques* useful in the context of industrial software development."

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

The COLD languages are said to constitute "a *wide spectrum language*: an integrated linguistic framework that supports descriptive techniques relevant to the various stages of software development (ranging from specification to implementation)."

COLD-1 is a user-friendly member of the COLD family: "COLD-1 is intended as a practical vehicle for writing formal specifications at various levels of abstraction. The language supports e.g. algebraic specification techniques, pre- and postcondition style definitions as well as algorithmic forms of description ('formal pseudo code'). Specifications written in COLD-1 are generally not executable, i.e. they cannot be compiled automatically to executable code."

PROTOCOLD is regarded as an executable fragment of COLD-1: "specifications written in PROTOCOLD, besides being used as specifications, can also be used as *programs* that can be run on a computer, thus adding a *prototyping* capability to COLD." This is not completely true, because an executable COLD-1 specification of a given system, also referred to in [FJ92a] as a *mixed specification*, actually consists of two layers. A bottom layer, specified in COLD-1, describes in pre- and postcondition style the physical components of the system. A top layer, specified in PROTOCOLD, describes the control structure.

The following fragment clearly states that a *description* written in PROTOCOLD can be viewed as either a specification or a program: "Each description in PROTOCOLD has an *abstract meaning* as determined by its interpretation as a COLD-1 specification. In addition, it has an *operational meaning* (semantics, v.V.&v.W.) as determined by the behaviour it displays when it is executed as a program. The operational meaning should of course comply with the abstract meaning in the sense that the behaviour displayed during execution of the description satisfies the specification, which is a requirement to be satisfied by the language implementation."

It would be good if the user of PROTOCOLD did not need to focus on operational semantics, but "Unfortunately there are a number of restrictions in the language (e.g. well-formedness requirements, v.V.&v.W. [1]) that can only be explained and understood if at least a basic understanding of the operational meaning of the PROTOCOLD language constructs exists. This applies particularly to those restrictions that are aimed at achieving efficiency of execution."

It is in the semantics of PROTOCOLD that some difficulties arise. In [Klu91] and [Dro92] attempts were made to provide PROTOCOLD with a formal semantics, particularly w.r.t. backtracking, which is used to undo the binding of object names in case of failure. Both authors approached the semantics of PROTOCOLD in a relational style, which concerns the I/O behaviour of program constructs. This turned out to be a difficult undertaking.

For the study of the semantical intricacies of backtracking in PROTOCOLD, an algebraic, axiomatic approach to backtracking may be useful: if one confines oneself to an algebraic setting, the properties of backtracking have to be expressed in a relatively simple and clear way (the triggering of backtracking and the undoing of undesired effects are explicitly described on the syntactic level). Moreover, it may be easier to understand backtracking in PROTOCOLD through bisimulation semantics (see e.g. [BW90]), which is a finer equivalence than relational semantics.

[1] There are a number of requirements on the well-formedness of PROTOCOLD expressions, most of which can be detected statically (*at compile time*). Important in this category are the restrictions on the binding of *object names* (see Section 3.3). A requirement that has to be verified dynamically (*run-time*) is the *Sequential Composition Requirement* (see Section 4.2). For detailed information on the well-formedness of PROTOCOLD expressions, the reader is referred to [Jon91b].

The specific process algebra variant that may fit this purpose is BPA(↶), Basic Process Algebra with backtracking, which was recently developed for the specification and study of processes with backtracking (see [BPW93]).

A general, informal definition of backtracking could be the following: *backtracking is the return from some point B in a program or formal specification where no further progress is possible to the nearest point A where executable alternatives exist.* In accordance with this view, backtracking can be concerned not only with the restoring of old values and states of data structures, but also with a return to previous choice moments in the control structure of a program or specification.

However, we will see that there are some important differences between the ways BPA(↶) and PROTOCOLD look at backtracking: BPA(↶) regards backtracking as *the undoing of data-state transformations in case a process fails*, while bindings of object names are not explicitly available in the syntax. PROTOCOLD regards backtracking as *the undoing of the bindings of object names in case a process fails*, while backtracking over data-state transformations is not possible. In PROTOCOLD bindings of object names are explicitly available as important syntactic notions, and data-state transformations are called *side effects* in PROTOCOLD. The binding of object names and the undoing of bindings are not considered side effects.

We can now formulate the main topic of this paper: we will *not* try to provide the full language PROTOCOLD with a formal semantics, but will study the operational meaning of some elementary language constructs from PROTOCOLD in the style of process algebra. Finally, we will try to find out whether the execution of PROTOCOLD fragments can be adequately specified as BPA(↶) processes. Before we can start our study of PROTOCOLD constructs, a more elementary job has to be done: we will have to establish the most important similarities and differences between PROTOCOLD and BPA(↶).

Our study must be regarded preliminary because our main aims are to elucidate the backtracking mechanism of PROTOCOLD, and to formulate some ideas for an algebraic approach for its semantics. We are aware of the fact that as a result of our general, semi formal approach, the reader may find a lack of mathematical rigour in this paper. We think that it is a bit premature to get *really* formal on matters whereupon no consensus exists yet. We hope, however, that this paper offers some new and useful perspectives on the semantics of PROTOCOLD.

Conventions. We associate PROTOCOLD with the top layer of a mixed specification. Unless stated otherwise, we interpret PROTOCOLD specifications accordingly, and if we talk about the meaning of a PROTOCOLD specification, we talk about its *operational* semantics. The *abstract* semantics of a PROTOCOLD specification is associated with the semantics of COLD-1.

Acknowledgements. We would like to thank Jan Bergstra, Alban Ponse and Arjan van Waveren from the University of Amsterdam, and Willem Jan Fokkink and Steven Klusener from the CWI for the various stimulating discussions we shared, as well as for their useful comments. We are also grateful to Bob Rigg who helped us with our English. Last but not least we thank Loe Feijs from Philips Research Laboratories Eindhoven for his valuable contribution of PROTOCOLD knowledge.

Contents

2 A short introduction to BPA with backtracking

BPA($\dotplus$), Basic Process Algebra with backtracking, is semantically based on processes that transform *data-states* in a *deterministic* way (see [BPW93]). In the BPA($\dotplus$) formalism, backtracking is regarded as the undoing of the effects on some initial data-state in case this process fails.

The process language of BPA($\dotplus$) is basically that of BPA (Basic Process Algebra, a basic fragment from ACP [BK84, BW90]) with *guarded commands* (or conditionals, i.e., **if - then - fi** constructs, [Dij76, BB91, GP90]). We assume that atomic 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 formalism that combines BPA with guarded commands and enabledness is $\mathrm{BPA_{gce}}$.

Backtracking is modelled by a binary operator $\dotplus$. The $\dotplus$ operator is axiomatised in such a way that it can be eliminated from closed terms in favour of the $+$ (choice), the $\cdot$ (sequential composition) and the guarded command. The undoing of actions (representing backtracking on the syntactic level) is modelled by the insertion of *undo* actions in the alternative to be executed upon failure. Special *flag* actions are used to replace an action as soon as it has invoked backtracking. This is done in order to prevent a single action from invoking new *undo* actions (and thus backtracking) more than once. This could be the case if an action occurs in the scope of more than one $\dotplus$ operator. As soon as an action has triggered backtracking it is replaced by its copy, which is a *flag* action. By definition *flag* actions cannot trigger backtracking.

For example: $undo(a)$ denotes the action which, semantically speaking, can undo effects of the action a on data-states, and $flag(a)$ denotes a duplicate of a. The action $flag(a)$ is a duplicate of a in the sense that it has the same data-state transformations associated with it, but it does not behave in a manner identical to a in the context of $\dotplus$.

In addition to the introduction of notions such as *flag* actions and *undo* actions, other design decisions were made to provide a backtracking operator with pleasant computational properties, the most characteristic of which are listed below.

- Restrictions on and additional structure for the set of actions are needed. This is achieved by partitioning the actions of A according to three unary (partitioning) predicates that represent different properties in the context of the backtracking operator. The partition of A basically consists of the types of action explained below.

  1. Actions that satisfy the predicate *Invertible*, which are actions that are invertible on the semantic level and, moreover, are considered to be invertible on the syntactic level. For any formally invertible action a, exactly one inverse is selected as *the* inverse and is further called $undo(a)$. Also a copy $flag(a)$ is made. Inverse actions (*undo* actions) and *flag* actions are considered to be not invertible in any sense: they satisfy the predicate *Pass*.
  2. Actions that are transparent to backtracking are characterised by the predicate *Pass*.
  3. Actions that refute any backtrack possibility are characterised by the predicate *Commit*.

  The classification of the actions from the set A is done with the help of an algorithm, described in [BPW93]. This classification leads to an extended set of actions:

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

  There are also restrictions (w.r.t. enabledness) on A_{flag} and the partitioning predicates. These restrictions are combined in a Boolean algebra $\mathbb{B}(A_{flag})$, which thus defines the interpretation of guards in $\Sigma(\mathrm{BPA}(\dotplus))$ expressions.

- The $\dotplus$ is associative (and hence more suitable for algebraic reasoning), provided that inverse actions are uniformly enabled, i.e. in every data-state.

Example 2.1. The different behaviour of the various types of action is best illustrated by the 'backtracking axiom' Ba3 from [BPW93]. Let $a \in A_{flag}$, and p, q be terms over $\Sigma(\mathrm{BPA}(\dotplus))$ [2].

$$\begin{aligned} a \cdot p \dotplus q \ \overset{Ba3}{=} \ & enabled(a) \wedge Invertible(a) :\rightarrow Flag(a) \cdot (p \dotplus Undo(a) \cdot q) \ + \\ & enabled(a) \wedge Pass(a) :\rightarrow a \cdot (p \dotplus q) \ + \\ & enabled(a) \wedge Commit(a) :\rightarrow a \cdot p \ + \\ & \neg enabled(a) :\rightarrow q. \end{aligned}$$

[2]The operators of BPA($\dotplus$) in order of decreasing binding strength: $\cdot, \dotplus, :\rightarrow, +$.

For $\mathit{Invertible}(a) = \mathbf{true}$ we have $\mathit{Flag}(a) = \mathit{flag}(a)$ and $\mathit{Undo}(a) = \mathit{undo}(a)$, otherwise $\mathit{Flag}(a) = \mathit{Undo}(a) = \delta$ (the special constant δ denotes *deadlock* or *inaction*). $\Box$

The notion of a *data environment* is fundamental for backtracking in BPA($\dotplus$), because backtracking in this formalism is based on transformations of data(-states). Such a data environment is defined relative to a set A_{flag} and a Boolean algebra $\mathbb{B}(A_{flag})$. Data-states form a basic ingredient of a data environment, as well as the functions *action*, *effect* and *eval*. The data-states are given by an abstract set S and are deterministically transformed by actions according to a function

$$\mathit{effect} : A_{flag} \times S \rightarrow S.$$

The interaction of processes with data-states is described by *state operators*

$$\lambda_s : \mathcal{P} \rightarrow \mathcal{P},$$

where $\mathcal{P}$ denotes the set of Σ(BPA($\dotplus$)) terms, and $s \in S$. State operators relate the separate semantic level (described by the data environment) to the process language. Conceptually, an application of the state operator to a process expression in a given initial data-state can be regarded as an evaluation or execution. For the use of the state operator in a given data environment two more functions are defined:

$$\mathit{action} : A_{flag} \times S \rightarrow A_{flag} \cup \{\delta\},$$

which defines for every action the resulting action under application of the state operator, and

$$\mathit{eval} : \mathbb{B}(A_{flag}) \times S \rightarrow \mathbb{B}(A_{flag}),$$

which relates guards to data-states. We finish this section with a small example of a recursively specified process over $\mathrm{BPA_{gce}}$ (thus without backtracking).

Example 2.2. Let $a, b, c \in A$, $X = (a + b + c) \cdot X$, and let the data environment be defined as follows:

- The set S of data-states is defined as set of states of a stack over a set $D = \{d_1, d_2\}$, with the following functions (a sort *Boolean* is assumed to be available):

$$\begin{array}{lll} \emptyset_S & : & \rightarrow \mathit{Stack} \\ \mathit{push} & : D \times \mathit{Stack} & \rightarrow \mathit{Stack} \\ \mathit{tail} & : \mathit{Stack} & \rightarrow \mathit{Stack} \\ \mathit{eq} & : \mathit{Stack} \times \mathit{Stack} & \rightarrow \mathit{Boolean}. \end{array}$$

  These functions satisfy the following equations, where $s \in \mathit{Stack}$, and $d \in D$:

$$\mathit{tail}(\emptyset_S) = \emptyset_S, \quad \mathit{tail}(\mathit{push}(d, s)) = s, \quad \mathit{eq}(\_, \_) = \ldots\ .$$

- The *action* function is defined as the identity.

- For all $s \in \mathit{Stack}$ the *effect* function is given by

$$\mathit{effect}(a, s) = \mathit{tail}(s), \quad \mathit{effect}(b, s) = \mathit{push}(d_1, s), \quad \mathit{effect}(c, s) = \mathit{push}(d_2, s).$$

- The evaluation of the *enabled* predicate is given by

$$eval(enabled(a), s) = not(eq(s, \emptyset_S)), \quad eval(enabled(b), s) = eval(enabled(c), s) = \mathbf{true}.$$

The data environment is related to the process X by means of the state operator; in any data-state X can be evaluated. If $s \in Stack$ and $d \in D$, we can derive the following expressions:

$$\begin{aligned} \lambda_{\emptyset_S}(X) &= b \cdot \lambda_{push(d_1,\emptyset_S)}(X) + c \cdot \lambda_{push(d_2,\emptyset_S)}(X), \\ \lambda_{push(d,s)}(X) &= a \cdot \lambda_s(X) + b \cdot \lambda_{push(d_1,push(d,s))}(X) + c \cdot \lambda_{push(d_2,push(d,s))}(X). \end{aligned}$$

□

3 A short introduction to COLD-1 and PROTOCOLD

Relevant notions from COLD-1 and PROTOCOLD are introduced and compared with notions from BPA($\dotplus$). We start by discussing COLD-1, by briefly explaining how states and state transformations are looked at, and how the basic language constructs (i.e. *assertions*, *terms* and *procedures*) interact with these notions. Much attention is paid to procedures, as these form the building blocks of PROTOCOLD expressions. In parallel, most operators from PROTOCOLD are introduced. In conclusion, we analyse some semantic notions that play a central role in backtracking in PROTOCOLD: *bindings of object names* and the *status* of these bindings.

3.1 States, assertions and terms

In COLD-1, the global state space associated with a specification is regarded as a collection of algebraic structures, also called *states*. A state consists of the following *state components*:

- *sorts*, which are sets containing *objects*,
- *predicates*, which are relations defined on sorts,
- *functions*, which are possibly partial mappings from a number of sorts to another sort. (In fact, predicates are special, Boolean, functions; mappings from a number of sorts to Boolean expressions.)

A state component can be *variable* or *constant*, which indicates whether or not it can be instantiated and modified during an execution. State transformations are directly associated with modifications of variable state components. This type of state components thus plays a central role in the interpretation of COLD-1 specifications, because the semantics of COLD-1 is relational; COLD-1 specifications are regarded state transformers.

Example 3.1. Some definitions of sorts, predicates and functions are given below. For instance, the sort *Mode* consists of 3 constants, also called *objects*. Also a variable function *mode* is defined on *Mode*. The last line of code gives the initialisation of this function [3].

SORT	*Natural*			
FUNC	*destination*	: −>	*Natural*	**VAR**
SORT	*Mode*			
FUNC	*pausing*	: −>	*Mode*	
FUNC	*moving*	: −>	*Mode*	
FUNC	*loading*	: −>	*Mode*	
FUNC	*mode*	: −>	*Mode*	**VAR**
PRED	*is_busy*	:		
PRED	*stopped_loading*	:		**VAR**
AXIOM	**INIT** =>	$mode = pausing$		

□

State information, i.e., information on the values of sorts, functions or predicates, can be referred to with the help of *assertions* and *terms*.

Assertions in a COLD-1 specification are statements about the state which, during an execution, are evaluated either as **TRUE** or as **FALSE**. Assertions cannot change the state.

As is the case with languages such as PASCAL or C, assertions can be composed using propositional operators such as **AND**, **OR** and **NOT**. Relational operators (built-in predicates) such as '=' and '/=' (inequality) are also available. Two special types of assertions are the *precondition* assertion (see Section 3.2) and the *binding* assertion (see Section 3.3).

Furthermore, there are the terms of COLD-1, which are also closely related to states. A term is a *typed* element in the language that, during an execution, denotes a specific element of a sort (e.g. a natural number, a string of ASCII characters). Like assertions, terms cannot influence the state.

The terms of COLD-1 are quite similar to terms from PASCAL or C, except that a term can be instantiated in a non-deterministic way. In this paper the structure of terms does not play an important role.

We see that the notion of states from COLD-1 is a more concrete notion than the data-states from BPA($\dotplus$). However, there is one important limitation to the observability of state information in COLD-1. The definition of state components in a particular *component* [4] of the specification is *hidden* if it is not part of the *export section* of that component. If information from a component C_1 of a specification is hidden, it cannot be referred to by another component C_2 that imports C_1.

[3] Most examples of COLD-1 and PROTOCOLD specifications are borrowed from [Jon91a]. Keywords are printed bold.

[4] Components are the building blocks of a specification, and the notion of components is central in the SPRINT method (see [Jon90] and [Jon92]), which is a formal method for the development of control software for audio- and video systems.

3.2 Procedures and state transformations

The basic syntactic objects of COLD-1 on which operations are defined are functions, predicates and procedures. In the above the functions and predicates are called state components because they constitute, together with the sorts, the state space. A procedure p of COLD-1 is viewed as a *relation* between states, i.e., a set of pairs of states, where each pair of states (s, s') semantically represents a *state transition* (referred to as an atomic action in [FJ92b] and [Jon91a]). If p has also input parameters $t_1, ..., t_n$, then p is regarded as a relation with elements of the form $(s, t_1, ..., t_n, s')$. Only state components that are specified as variable (using **VAR**) can be transformed by a procedure. Transformations of state components are called *side effects*. In operational contexts (SPRINT jargon) the state components are also referred to as *state observers* and the procedures as *state transformers*.

Closely related to state transitions are the notions *success* and *failure*. Abstractly speaking, a call of a procedure p is said to succeed in a state s if there exists a state s' such that p can transform s to s'. Otherwise p is said to fail in s. For those who are familiar with this, we note that there are many similarities between COLD-1 and PROTOCOLD on the one hand and dynamic logic on the other (see [Jon92]).

Roughly speaking, there are two ways in which a procedure can be specified in COLD-1 [5]. The first is the *pre- and postcondition* style, which is as follows: a procedure (a *name* with possibly some arguments) consists of two assertions, where one assertion (the precondition) must hold before execution and the other (the postcondition) holds after execution.

This type of procedure is conceptually very similar to the actions of BPA($\dot{+}$). The precondition tests the enabledness, and the postcondition gives the effect. An important difference is that postconditions can introduce non-determinism in a specification. An example of this is given below.

Example 3.2. In the specification below, non-determinism is introduced by the **OR** function in the postcondition. It is not known beforehand whether or not *stopped_loading* will hold after the execution of *load_step*.

$$
\begin{array}{ll}
\textbf{PROC} & load\_step :\ ->\\
\textbf{PRE} & mode = loading\\
\textbf{SAT} & \textbf{MOD}\ mode, stopped\_loading\\
\textbf{POST} & (mode = loading\ \textbf{AND NOT}\ stopped\_loading)\ \textbf{OR}\\
 & (mode = pausing\ \textbf{AND}\ stopped\_loading)
\end{array}
$$

□

The second type of procedure is given by the definition of some composite *expression*. We will refer to these procedures as being specified in the *algorithmic* style. The binary operators used for the construction of these procedures are the sequential composition ';' and choice '|'. From an abstract point of view, the choice is made non-deterministically.

In composite expressions *guard* expressions often appear, which have the form A**?**, where A is an assertion. Semantically, such guard expressions are interpreted as follows: if A is

[5] Next to the two types of procedure specifications we treat in this paper, several intermediate forms are possible.

evaluated as **TRUE** in a given state, the guard expression A**?** succeeds in this state, otherwise it fails. The assertion **TRUE?** is operationally equivalent to the special PROTOCOLD expression **SKIP**.

Precondition calls often appear in guard expressions. If p is a procedure, then p**?** is a precondition call (in the context _**?**, p is called a *precondition assertion*). An evaluation of p**?** is interpreted as a test of whether p can be executed successfully. If p is a procedure in pre- and postcondition style, a test of the precondition will suffice: it can be regarded as a test on the enabledness of p. If p is a procedure in algorithmic style things are different, because such a precondition call will often lead to partial execution of p. Some complications may occur due to the fact that, during a precondition call, *object names* (logical variables, see Section 3.3) may become bound. Precondition calls are not allowed to cause side-effects.

Another type of expression that can occur in procedures is the *binding* expression (see Section 3.3). One important operator, which we do not investigate in this paper, is the unary iteration operator '$*$'. Study of this operator requires an algebraic setting with recursion or iteration (for iteration in process algebra see [BBP93]).

Example 3.3. A typical PROTOCOLD specification.

```
PROC  control_step : - >
DEF   ( power_on_event? ;          initialise
      | NOT power_on_event? ;      ( play_control
                                   | selection_control
                                   | credit_control )
      ) ;
      adjust_display
```

□

In COLD-1 jargon, a distinction is made between procedures and expressions (expressions being defined in the body of 'algorithmic style' procedures). We will not always make this distinction. If no confusion can arise, expressions that can transform states are identified with procedures, in order to preserve the analogy between the process terms over $\Sigma(\mathrm{BPA}(\dot{+}))$ and procedures.

To sum up this section: at the lowest level, procedures in PROTOCOLD can be regarded as the counterparts of actions in $\mathrm{BPA}(\dot{+})$, except that the latter formalism does not allow for non-deterministic effects. Such procedures can be modelled otherwise, however, because this non-determinism can be modelled as process behaviour (using the choice operator, see Section 5.3). At a higher level, procedures are operationally comparable with processes.

3.3 Object names and bindings of object names

An *object name* denotes an element or an *object* from a sort and is considered a single assignment variable (and therefore differs from the variables of e.g. PASCAL). After an object name has been bound to a certain object, it always refers to this object in a given scope. In the literature this property is sometimes called *referential transparency*.

In PROTOCOLD we can distinguish between two types of object names: *local* object names, which are created with the keyword **LET**, and *parameter* object names, which are created with **IN**. The scope of an object name is the body of the function, predicate or procedure in which it is created.

Example 3.4.

$$\begin{array}{ll} \textbf{PRED} & p : \mathit{Natural} \\ \textbf{IN} & x \\ \textbf{DEF} & \textbf{LET}\ \ y : \mathit{Natural}; \\ & \ldots \end{array}$$

□

The binding of an object name x to an object denoted by a term t is done by $x := t$. A binding can occur in an assertion or an expression, as a *binding assertion* or a *binding expression* respectively. A binding can also be part of a term, in which case the binding is the *prefix* of another term (cf. the guards in expressions). Bindings do not occur as terms as such.

Object names are considered single assignment variables. Therefore, the ':=' must be regarded an equality test rather than an assignment if the left argument (an object name) is already bound.

Example 3.5.

$$\begin{array}{ll} \textbf{PRED} & \mathit{false} : \\ \textbf{DEF} & \textbf{LET}\ \ x : \mathit{Natural}; \\ & x := 5\ \textbf{AND}\ x := 5\ \textbf{AND}\ x := 10 \end{array}$$

Initially x is not bound to any object, so the first $x := 5$ succeeds [6]. The next $x := 5$ also succeeds because x and 5 denote the same object. After this binding, x and 10 denote different objects, so $x := 10$ fails and the predicate as a whole fails. □

A *binding status* which can have three values - *free*, *bound*, or *dummy* - is associated with each object name in PROTOCOLD. If an object name is free, a binding changes it to bound. If an object name is bound, a binding of this name can be interpreted as an equality test. Any attempt to bind an object name with the dummy status fails.

Function calls and predicate calls must be done with bound object names. Procedures can be called with free object names, which will be bound or dummy after execution. This also holds for precondition calls, so precondition calls can have *binding effects* (every change of the binding status of object names will be called a binding effect).

The binding status is important, because it is forbidden in PROTOCOLD to refer to the value of an object name before it has been bound. In order to detect at possible violations of this demand at *compile time*, the dummy status is used: the dummy status signals 'suspect' situations. If an object name with free or dummy status is referred to in a specification, the specification is incorrect. On the various demands on the use of object names that must be satisfied by a correct specification, the reader is referred to [Jon91a] and [Jon91b].

[6] In Example 3.5 ,we already assume some specific order in which the bindings are evaluated. A more abstract point of view is also possible: the **DEF** clause in this example can be interpreted as the predicate $\exists x \in \mathit{Natural} \,.\, x = 5 \wedge x = 5 \wedge x = 10$.

In general, if the binding status of an object is bound or dummy this remains true, unless backtracking is applied (for a counterexample, see the binding effects of the *negation assertion* in [Jon91a]). In the following sections we will see how backtracking is used to change the binding status back to free. We note once again that changes in binding status are not considered side effects.

Nevertheless, if we want to model the backtracking in PROTOCOLD with BPA($\dotplus$), a binding effect has to be regarded a data-state transformation, because backtracking in BPA($\dotplus$) is based on transformations of data-states.

4 The execution of mixed specifications

We now discuss what an executable specification in COLD-1 is, and what is meant by an execution of such a specification. Next we show how two fundamentally different interpretations of the choice operator of PROTOCOLD are conceivable. In one interpretation the non-determinism is removed from the choice expressions during an execution, and in the other the non-determinism is preserved at the cost of a more complicated operational semantics. We also argue that the execution of a mixed specification can be viewed as the evaluation of a Σ(BPA($\dotplus$)) term. Finally, some typical examples of PROTOCOLD expressions with possible backtracking are discussed.

4.1 Specifications in COLD-1 and PROTOCOLD

A specification in executable COLD-1 is also referred to as a *mixed specification*. A mixed specification consists of two layers. The lower layer is formed by a COLD-1 specification of which the procedures are written in pre- and postcondition style. The higher level of a mixed specification is called PROTOCOLD. At this level procedures are specified in the algorithmic style. As a consequence pure, executable PROTOCOLD specifications do not exist.

From an abstract, relational (I/O) point of view, the set of possible final states that can be reached by *successfully* executing a mixed specification is a subset of the set of final states that exists according to the abstract COLD-1 semantics. We will see that this 'subsetting of final states' can be accounted for in a quite comprehensible way when we approach the execution of mixed specifications operationally.

We now provide an informal definition of the execution of a mixed specification: *the execution of a mixed specification is the process that tries to find a successfully terminating 'trace'.* As soon as such a trace is found, the execution is successfully completed. If no such trace is found, the execution has failed. We observe that the above notion of execution is very similar to that of the language PROLOG (see e.g. [CM87]), except that PROLOG can find more than one solution (successfully terminating trace) in succession. Note that from now on we approach PROTOCOLD operationally, and that we implicitly use a semantics that identifies fewer expressions than a relational semantics.

We can now make the following interesting observation: the expression $p\mathbf{?};p$ must be operationally different from p, because in case of success some process-like activity (the evaluation of $p\mathbf{?}$) precedes the execution of p.

For a proper understanding of the operational semantics of PROTOCOLD we comment on the fact that, in the execution of a PROTOCOLD specification, the non-determinism can be removed from expressions that contain the choice operator '$|$' or the function **OR**. The execution of $p_1 \mid p_2$ can be interpreted as 'try to execute p_1 and if this fails try to execute p_2'. The evaluation of an assertion A_1 **OR** A_2 is then interpreted as 'see whether A_1 is true, otherwise see whether A_2 is true'. This way of looking at executions implies that expressions that contain choice operators are operationally fully sequential and deterministic.

There is, however, another way of looking at choice in PROTOCOLD. Choice can be kept non-deterministic by interpreting choice expressions $p \mid q$ as follows: 'try to execute p_1 and if this fails try to execute p_2 *or* try to execute p_2 and if this fails try to execute p_1'. In order to distinguish between these two fundamentally different semantic notions of choice, we write '$|_{\mathrm{L}}$' (*left choice*) when we explicitly want to reason about a deterministic choice operator, and '$|_{\mathrm{C}}$' (*commutative choice*) when we want to reason about the non-deterministic variant. If our remarks concern both approaches we simply write '$|$'.

At this stage we can already see a parallel between the execution of a mixed specification and the evaluation of a $\Sigma(\mathrm{BPA}(\dotplus))$ term with the help of the state operator. However, also the evaluation of assertions displays process-like behaviour. As a consequence, we cannot take full advantage of the relatively simple way Boolean expressions are handled in $\mathrm{BPA}(\dotplus)$ (in $\mathrm{BPA}(\dotplus)$ the logical connectives $\vee$ and $\wedge$ are commutative).

The possibility of regarding executions of mixed specifications as $\Sigma(\mathrm{BPA}(\dotplus))$ terms is the main topic of Section 5.

4.2 Examples of expressions with backtracking

As COLD-1 was designed to be a single assignment language (see Section 3.3), with its inherent complications with variable bindings, some form of backtracking is needed to undo the bindings of objects in case an execution fails. In fact, backtracking in PROTOCOLD can be regarded as the modification of the binding status of objects from bound or dummy to free.

We demonstrate the process of execution, particularly w.r.t. backtracking with the help of some simple expressions from PROTOCOLD. We start by discussing the choice operator. Let $p_1, p_2, \ldots$ range over PROTOCOLD expressions.

The expression $p_1 \mid_{\mathrm{L}} p_2$ must be interpreted as 'try to execute p_1 and if this fails undo the binding effects made by p_1 and try to execute p_2' [7]. (So if the binding of variables and the undoing of binding effects are operationally regarded as actions, the expression $p \mid p$ does generally not behave as p.) A fundamental question that now arises is whether or not the binding effects of p_2 are undone if p_2 fails.

We need a working hypothesis on this problem, so we extend our definition of the execution of a mixed specification as follows: *the execution of a mixed specification is the process that tries to find a successfully terminating 'trace'. If this process fails, all its binding effects are undone* (note that we keep the backtracking mechanism of PROLOG in mind). So the answer to the above question is 'yes'.

According to the above definition, the expression $p_1;p_2$ must be interpreted as: 'try to execute p_1 and if this fails undo the binding effects. Otherwise try to execute p_2 and, if this fails, undo the binding effects of both p_1 and p_2' [8].

[7] If p_1 fails after a side effect, the execution of $p_1 \mid_{\mathrm{L}} p_2$ leads to a so-called *run-time error*, because backtracking over side effects is not allowed.

[8] An important (semantic) restriction on PROTOCOLD specifications is that expressions $p_1;p_2$ must satisfy

The way procedures can be constructed from other procedures is not *compositional* in the sense $\Sigma(\mathrm{BPA}(+))$ terms are. Textual substitution of a procedure p_1 defined by $p_1' \mid p_1''$ in a procedure $p_1;p_2$ yields a procedure $(p_1' \mid p_1'');p_2$ that, in general, is operationally different to $p_1;p_2$. This difference can be accounted for as follows: if p_1 succeeds in $p_1;p_2$, it is not possible to backtrack *inside* p_1 if p_2 fails. In PROTOCOLD one is not allowed to backtrack into the body of a procedure, in a context of some larger, composite, higher level, procedure. In other words: once a procedure has succeeded, it cannot be re-entered by backtracking. If we refer to this property, we say that backtracking is *contextual*. Backtracking *over* a procedure is possible if no side effects have been caused by this procedure.

Example 4.1. This example demonstrates in a more formal way the contextual behaviour of backtracking in PROTOCOLD: an execution of the procedure *surprise* fails, and *ok* succeeds.

$$\begin{array}{ll}
\textbf{PROC} & p : \mathit{Natural}\ -> \\
\textbf{IN} & x \\
\textbf{DEF} & x := 2 \mid_{\mathrm{L}} x := 3 \\
 & \\
\textbf{PROC} & \mathit{surprise} :\ -> \\
\textbf{DEF} & \textbf{LET}\ \ y : \mathit{Natural}; \\
 & p(y); y := 3 \\
 & \\
\textbf{PROC} & \mathit{ok} :\ -> \\
\textbf{DEF} & \textbf{LET}\ \ y : \mathit{Natural}; \\
 & (y := 2 \mid_{\mathrm{L}} y := 3); y := 3
\end{array}$$

Note that *surprise* would succeed if p were defined by

$$\begin{array}{ll}
\textbf{PROC} & p : \mathit{Natural}\ -> \\
\textbf{IN} & x \\
\textbf{DEF} & x := 3 \mid_{\mathrm{L}} x := 2
\end{array}$$

As a consequence, the choice operator '$\mid_{\mathrm{L}}$' cannot be commutative, not even when interpreted in a relational semantics. □

We now consider the non-deterministic (and therefore commutative) choice operator '$\mid_{\mathrm{C}}$' as defined in the previous section. If we turn to the example above and replace '$\mid_{\mathrm{L}}$' by '$\mid_{\mathrm{C}}$' in the procedure p, we see that the execution of *surprise* may fail, but may also succeed. Generally speaking, an execution of a specification with '$\mid_{\mathrm{C}}$' operators may fail, but may also succeed, due to the combination of non-determinism and the contextual behaviour of backtracking.

What is also demonstrated by the examples above is that a successful execution of a PROTOCOLD specification in general yields a subset of the set of final states that exist according to the abstract semantics of COLD-1.

The most characteristic example of backtracking in PROTOCOLD is given by $(p_1 \mid p_2);p_3$. This expression behaves in the same way as $p_1;p_3 \mid p_2;p_3$, which demonstrates that backtracking is not bound to the scope of the '$\mid$' operator only, but pervades expressions as a whole.

the *Sequential Composition Requirement*, i.e., if p_1 succeeds with a side effect, then p_2 must succeed. This requirement is a consequence of the design decision not to allow backtracking over side effects in PROTOCOLD.

This property will be referred to as the *(right-)distributivity of* ';' *over* '|'. It leads us to the conclusion that a canonical mapping from PROTOCOLD to BPA($\dotplus$) operators does not exist: in BPA($\dotplus$) sequential composition does not distribute over the $\dotplus$ operator. We see, however, that the parallel between PROTOCOLD and PROLOG, where conjunction right-distributes over disjunction, becomes more convincing once backtracking is introduced.

5 An interpretation of PROTOCOLD using process algebra

A data environment for the specification of object names and object name bindings is defined, suitable for use in BPA($\dotplus$). Next a small program language PC_L is defined for the description of the operational behaviour of PROTOCOLD-like expressions, in which precondition calls and guard expressions are not considered. Operational equivalences that hold in PROTOCOLD are modelled in PC_L by rules, e.g., procedure calls are modelled with a unary (*call*) operator '[_]', and the specific contextual properties of backtracking in PROTOCOLD are formalised with rules for '[_]'. The distributivity of ';' over '$|_{\mathrm{L}}$' is modelled by a distribution rule, and normal forms over $\Sigma(\mathrm{PC}_L)$ are defined as expressions, without the *call* operator, to which the distribution rule can no longer be applied. We comment on a possible language PC_C, based on a non-deterministic, commutative choice operator '$|_{\mathrm{C}}$'. In conclusion, we define a mapping of normal forms over $\Sigma(\mathrm{PC}_L)$ to $\Sigma(\mathrm{BPA}(\dotplus))$ terms.

5.1 A data environment with object names and object name bindings

For the specification of processes with object names and object name bindings over $\Sigma(\mathrm{BPA}(\dotplus))$, a special data environment is defined. For simplicity we assume that all object names have the same type. We assume that the *Booleans* are given, and that the constants **true**, **false** and the standard functions *and*, *or* and *not* are given. Before we can give a data environment for BPA($\dotplus$) we turn to $\mathrm{BPA}_{\mathrm{gce}}$, as it forms the basis for BPA($\dotplus$).

Definition 5.1. The sorts and functions for a data environment for $\mathrm{BPA}_{\mathrm{gce}}$ with object names and object name bindings.

- X_{Type} (abbreviated as X_T): a set of object names associated with a sort called *Type*.
- $\mathit{Activity} \stackrel{\mathrm{def}}{=} \{\mathit{binding}, \mathit{test}\}$: a set that contains the possible manipulations of object names. An equality function $eq_A : \mathit{Activity} \times \mathit{Activity} \rightarrow \mathit{Boolean}$ is assumed.
- $OS(X_T) \stackrel{\mathrm{def}}{=} \{t(x) \mid x \in X_T\}$: a set with object name binding states that contains for each $x \in X_T$ a triple $t(x) \stackrel{\mathrm{def}}{=} \langle t, bs, h\rangle_x$, where
  - $t \in \mathit{Type}$, where *Type* is a sort with an element $\emptyset_T$ and an equality function $eq_T : \mathit{Type} \times \mathit{Type} \rightarrow \mathit{Boolean}$,
  - $bs \in \mathit{Status}$, where $\mathit{Status} = \{\mathit{free}, \mathit{bound}\}$ represents the status of an object name binding. On *Status* an equality function $eq_S : \mathit{Status} \times \mathit{Status} \rightarrow \mathit{Boolean}$ is assumed [9],

[9] We do not consider the *dummy* status because it relates to the well-formedness of PROTOCOLD expressions rather than to operational semantics.

- $h \in History$, where $History$ is a stack that contains manipulations (activities) applied to the object name x. The empty history is given by $\emptyset_H$ and we assume functions

$$\begin{array}{lll} push & : Activity \times History & \rightarrow History, \\ pop & : History & \rightarrow Activity, \\ tail & : History & \rightarrow History, \\ eq_H & : History \times History & \rightarrow Boolean. \end{array}$$

π_1, π_2 and π_3 represent projection functions that refer to information from $t(x) = \langle t, bs, h\rangle_x$ as follows:

$$\pi_1(t(x)) = t, \quad \pi_2(t(x)) = bs, \quad \pi_3(t(x)) = h.$$

□

The expression $t(x)$ can be regarded as the state of the object name x. An object name is said to be in its *initial binding state* if

$$t(x) = \langle \emptyset_T, free, \emptyset_H\rangle_x.$$

By $s_0 \in OS(X_T)$ we denote the data-state in which each $x \in X_T$ is in its initial binding state.

Definition 5.2. A set with object name binding and *clear* actions for $\mathrm{BPA}_{\mathrm{gce}}$, parameterised with a set of objects X_T is defined by

$$OB(X_T) \stackrel{\mathrm{def}}{=} \{\langle x := t\rangle, clear(x) \mid x \in X_T, t \in Type\}.$$

□

According to [BPW93], a data environment for $\mathrm{BPA}_{\mathrm{gce}}$ can be constructed from a set A of atomic actions, a set S of data-states, and a Boolean algebra that contains expressions $enabled(a)$ for all $a \in A$. We enlarge this notion of data environments with object name bindings by incorporating a set X_T of object names, a set of object name binding states $OS(X_T)$, and a set of object name binding and *clear* actions $OB(X_T)$, in accordance with the above definitions.

Definition 5.3. A data environment $\mathcal{S}(X_T)$ for $\mathrm{BPA}_{\mathrm{gce}}$ over a set A of atomic actions, a set $OB(X_T)$ of object name binding and *clear* actions, and a Boolean algebra $\mathbb{B}(A(X_T))$ is a tuple $\langle S(X_T), action, effect, eval\rangle$, where

- $A(X_T) \stackrel{\mathrm{def}}{=} A \cup OB(X_T)$ is a non-empty set of actions,
- $S(X_T) \stackrel{\mathrm{def}}{=} S \times OS(X_T)$ is a non-empty set of data-states (S is assumed to be some set of data-states for a data environment for $\mathrm{BPA}_{\mathrm{gce}}$),
- $action : A(X_T) \times S(X_T) \rightarrow A(X_T) \cup \{\delta\}$, for all $a \in OB(X_T)$ and $s \in S(X_T)$ we define

$$action(a, s) \stackrel{\mathrm{def}}{=} a,$$

- *effect* : $A(X_T) \times S(X_T) \to S(X_T)$. For all $s \in S(X_T)$, $x \in X_T$ and $t \in \mathit{Type}$, the *effect* function satisfies the following constraints (we write $s[t'_1/t_1, ..., t'_n/t_n]$ for s with t'_1 substituted for t_1, etc.):
$\mathit{effect}(\langle x := t\rangle, s) \stackrel{\text{def}}{=}$

$$\begin{cases} s[\langle t, \mathit{bound}, \mathit{push}(\mathit{binding}, \emptyset_H)\rangle_x/t(x)] & \text{if } eq_S(\pi_2(t(x)), \mathit{free}) = \mathbf{true} \\ s[\mathit{push}(\mathit{test}, \pi_3(t(x)))/\pi_3(t(x))] & \text{if } eq_T(\pi_1(t(x)), t) \text{ and } eq_S(\pi_2(t(x)), \mathit{bound}) = \mathbf{true} \\ s & \text{otherwise,} \end{cases}$$

$\mathit{effect}(\mathit{clear}(x), s) \stackrel{\text{def}}{=}$

$$\begin{cases} s[\langle \emptyset_T, \mathit{free}, \emptyset_H\rangle_x/t(x)] & \text{if } eq_A(\mathit{pop}(\pi_3(t(x))), \mathit{binding}) = \mathbf{true} \\ s[\mathit{tail}(\pi_3(t(x)))/\pi_3(t(x))] & \text{if } eq_A(\mathit{pop}(\pi_3(t(x))), \mathit{test}) = \mathbf{true} \\ s & \text{otherwise,} \end{cases}$$

- *eval* : $\mathbb{B}(A(X_T)) \times S(X_T) \to \mathbb{B}(A(X_T))$. For all $s \in S(X_T)$, $x \in X_T$ and $t \in \mathit{Type}$, the *eval* function satisfies the following constraints:
$\mathit{eval}(\mathit{enabled}(\langle x := t\rangle), s) \stackrel{\text{def}}{=}$

$$\begin{cases} \mathbf{true} & \text{if } (eq_T(\pi_1(t(x)), \emptyset_T) \text{ and } eq_S(\pi_2(t(x)), \mathit{free}) \text{ and } eq_H(\pi_3(t(x)), \emptyset_H)) \\ & \quad \text{or } (eq_T(\pi_1(t(x)), t) \text{ and } eq_S(\pi_2(t(x)), \mathit{bound})) = \mathbf{true} \\ \mathbf{false} & \text{otherwise,} \end{cases}$$

$\mathit{eval}(\mathit{enabled}(\mathit{clear}(x)), s) \stackrel{\text{def}}{=} \mathbf{true}.$

We assume that for $a \in A$ the functions *action*, *effect* and *eval* are given relative to a set S, and that these functions are trivially extended and defined on the set $S(X_T)$. □

The following step towards a data environment for BPA($\dotplus$) is the partitioning and extension of the set of actions $A(X_T)$. It can be easily proved that for all $x \in X_T$ and $t \in \mathit{Type}$ the action $\mathit{clear}(x)$ is an inverse of $\langle x := t\rangle$ (note that $\mathit{clear}(x)$ is uniformly enabled), so the following choices lead to a useful partition of $A(X_T)$:

$$\mathit{Invertible}(\langle x := t\rangle) \stackrel{\text{def}}{=} \mathit{Pass}(\mathit{clear}(x)) \stackrel{\text{def}}{=} \mathbf{true}.$$

By definition we have $\mathit{Pass}(\mathit{flag}(\langle x := t\rangle)) = \mathbf{true}$, and we further assume a partitioned and extended set of actions A_{flag}. We define

$$A_{\mathit{flag}}(X_T) \stackrel{\text{def}}{=} A_{\mathit{flag}} \cup OB(X_T) \cup \{\mathit{flag}(\langle x := t\rangle) \mid x \in X_T\}.$$

Without further comment we note that a data environment $\mathcal{S}_{\mathit{flag}}(X_T)$ for BPA($\dotplus$) can be simply defined using the above definitions. We now provide an example that illustrates how an evaluation of a process term over Σ(BPA($\dotplus$)) by means of the state operator can be accomplished in a data environment $\mathcal{S}_{\mathit{flag}}(X_T)$.

Example 5.4. Let *Type* be the natural numbers, and q some term over $\Sigma(\mathrm{BPA}(\dot{+}))$. We assume a data environment $\mathcal{S}_{\mathit{flag}}(X_T)$, and write $\langle x := 2\rangle_f$ for $\mathit{flag}(\langle x := 2\rangle)$. If the object names are in their initial state $s_0 \in OS(X_T)$, $s \in S$ and the process term p is defined by $p \stackrel{\text{def}}{=} \langle x := 2\rangle \cdot \langle x := 2\rangle \cdot \langle x := 3\rangle \dot{+} q$, then p behaves as follows [10]:

$$\begin{aligned}
\lambda_{(s,s_0)}(p) &= \lambda_{(s,s_0)}(\langle x := 2\rangle \cdot \langle x := 2\rangle \cdot \langle x := 3\rangle \dot{+} q)\\
&= \langle x := 2\rangle_f \cdot\\
&\quad\ \lambda_{(s,s_0[\langle 2,bound,push(binding,\emptyset_H)\rangle_x/t(x)])}(\langle x := 2\rangle \cdot \langle x := 3\rangle \dot{+} clear(x)\cdot q)\\
&= \langle x := 2\rangle_f \cdot \langle x := 2\rangle_f \cdot \lambda_{(s,s_0[\langle 2,bound,push(test,push(binding,\emptyset_H))\rangle_x/t(x)])}\\
&\quad\ (\langle x := 3\rangle \dot{+} clear(x)\cdot clear(x)\cdot q)\\
&= \langle x := 2\rangle_f \cdot \langle x := 2\rangle_f \cdot\\
&\quad\ \lambda_{(s,s_0[\langle 2,bound,push(test,push(binding,\emptyset_H))\rangle_x/t(x)])}(clear(x)\cdot clear(x)\cdot q)\\
&= \langle x := 2\rangle_f \cdot \langle x := 2\rangle_f \cdot clear(x)\cdot\\
&\quad\ \lambda_{(s,s_0[\langle 2,bound,push(binding,\emptyset_H)\rangle_x/t(x)])}(clear(x)\cdot q)\\
&= \langle x := 2\rangle_f \cdot \langle x := 2\rangle_f \cdot clear(x)\cdot clear(x)\cdot \lambda_{(s,s_0)}(q).
\end{aligned}$$

In the evaluation of p above, the first binding of x to 2 succeeds because x is in its initial binding state. The second binding of x to 2 also succeeds because x is already bound to 2; the second binding simply behaves as an equality test on x and 2. The subsequent attempt to bind x to 3 fails, backtracking is triggered, and the binding of x to 2 is undone by the second $clear(x)$ action. After this, the process is back in the initial state (s, s_0), and the right argument q of the $\dot{+}$ operator can be evaluated. □

The above example illustrates the occurrence of two types of bindings of object names $\langle x := t\rangle$, as well as of two types of $clear(x)$ actions. For instance, there are $clear(x)$ actions that do actually change the binding status and $clear(x)$ actions that do not. If desired, this type of additional information related to the data-states can be encoded with the *action* function.

5.2 A small language for mixed specifications

In this section we give a small language PC_L for PROTOCOLD-like expressions with some rules that express important operational equivalences that hold in PROTOCOLD. One important operational identity was discussed in Section 4.2: the distributivity of ';' over '|'. We will also attempt to formalise the contextual behaviour of backtracking in PROTOCOLD. We abstract from the parameter passing mechanisms that are used in PROTOCOLD. For reasons of simplicity the **LET** construct is not modelled in PC_L.

[10] Note the analogy with Example 3.5. In PROTOCOLD the **AND** function may be replaced by ';' and the **OR** function by '|'.

First we define the basic ingredients of a $\Sigma(\mathrm{PC}_L)$ term. We assume a set A_{pc} of actions with typical elements α, β, ..., as well as a special constant **FAIL** [11]. The sort P_{pc} of procedures, with elements p, p', q, ..., is inductively defined. Every action $\alpha \in A_{pc}$ is a procedure, **FAIL** is a procedure, and three operators on the sort of procedures define new procedures as follows:

$$\begin{array}{llll} \_ \mid_{\mathrm{L}} \_ & : P_{pc} \times P_{pc} & \rightarrow P_{pc} & \text{alternative composition or } \textit{left choice}, \\ \_ \,;\, \_ & : P_{pc} \times P_{pc} & \rightarrow P_{pc} & \text{sequential composition,} \\ [\_] & : P_{pc} & \rightarrow P_{pc} & \text{procedure call.} \end{array}$$

We define the following rules for PC_L. Let x, y and z range over the $\Sigma(\mathrm{PC}_L)$ expressions, and let ';' bind stronger than '$|_{\mathrm{L}}$'. Most of the rules in Table 1 are straightforward interpretations

(Assoc)	$(x; y); z = x; (y; z)$
(Distr)	$(x \mid_{\mathrm{L}} y); z = x; z \mid_{\mathrm{L}} y; z$
(Call1)	$[\alpha] = \alpha$
(Call2)	$[\alpha; x] = \alpha; [x]$
(Call3)	$[x \mid_{\mathrm{L}} y] = [x]; \zeta \mid_{\mathrm{L}} \zeta; [y]$
(Fail1)	$\mathbf{FAIL}; x = \mathbf{FAIL}$
(Fail2)	$\mathbf{FAIL} \mid_{\mathrm{L}} x = x$
(Fail3)	$[\mathbf{FAIL}] = \mathbf{FAIL}$

Table 1: The rules of PC_L, where $\alpha, \zeta \in A_{pc}$.

of the operational behaviour of expressions (procedures) in mixed specifications. However, the rule (Call3) deserves some attention, because it describes the normalisation of choice expressions that are referred to by a procedure call: it can be used to remove the modular structure from PROTOCOLD expressions. We assume that the special action ζ is initially always enabled, and that it is disabled as soon as it is executed. So (Call3) describes how ζ 'causes' the commitment to x as soon as it has succeeded, simply by blocking y.

The introduction of actions ζ by (Call3) leads us to two rules that are not easily formulated in the language over $\Sigma(\mathrm{PC}_L)$, so we provide them informally. The first is that every application of (Call3) has to introduce a *fresh* action ζ. The second is that an application of *Distr*, with z containing some special actions ζ_1, ζ_2, ..., has to lead to a duplicate z' with fresh actions ζ_1', ζ_2', ... respectively:

$$(x \mid_{\mathrm{L}} y); z \stackrel{\mathit{Distr}}{=} x; z \mid_{\mathrm{L}} y; z'.$$

It can be easily shown that the left choice operator is neither associative nor commutative as a result of the rule (Call3). By the normal congruence properties for '=', it is easy to demonstrate that direct substitution of a procedure q in a procedure p must not be confused with a call of q by p.

[11] **FAIL** is not a special PROTOCOLD expression like **SKIP** (see Section 3.2), but it can, however, be interpreted as the assertion **FALSE?**. We think that for **SKIP** it is sufficient to assume that **SKIP**$\in A_{pc}$, and that no special rules are needed.

Example 5.5. Let $q \stackrel{\text{def}}{=} q' \mid_{\mathrm{L}} q''$, $p \stackrel{\text{def}}{=} q; \alpha$ and $p' \stackrel{\text{def}}{=} [q]; \alpha$. We derive

$$p = (q' \mid_{\mathrm{L}} q''); \alpha \stackrel{\mathit{Distr}}{=} q'; \alpha \mid_{\mathrm{L}} q''; \alpha,$$

$$p' = [q' \mid_{\mathrm{L}} q'']; \alpha \stackrel{\mathit{Call3}}{=} ([q']; \zeta \mid_{\mathrm{L}} \zeta; [q'']); \alpha \stackrel{\mathit{Distr}}{=} [q']; \zeta; \alpha \mid_{\mathrm{L}} \zeta; [q'']; \alpha.$$

□

The following step is the definition of some special types of $\Sigma(\mathrm{PC}_L)$ expressions. First we define *flat* expressions, being expressions without the *call* operator. Second we define *normal forms*, being a subset of the flat expressions.

Definition 5.6. A *flat* expression over $\Sigma(\mathrm{PC}_L)$ is inductively defined by the following BNF grammar, where $\alpha \in A_{pc}$:

$$p ::= \quad \mathbf{FAIL} \quad | \quad \alpha \quad | \quad p; p \quad | \quad p \mid_{\mathrm{L}} p.$$

□

In the following lemma we state that the *call* operator can be eliminated from all expressions over $\Sigma(\mathrm{PC}_L)$ [12].

Lemma 5.7. *For each expression p over $\Sigma(\mathrm{PC}_L)$ there is a flat expression $\bar{p}$ over $\Sigma(\mathrm{PC}_L)$ such that*

$$\mathrm{PC}_L \vdash p = \bar{p}.$$

Proof. Easy. □

Definition 5.8. A *normal form* over $\Sigma(\mathrm{PC}_L)$ is inductively defined by the following BNF grammar, where $\alpha \in A_{pc}$:

$$p ::= \quad \mathbf{FAIL} \quad | \quad \alpha \quad | \quad \alpha; p \quad | \quad p \mid_{\mathrm{L}} p.$$

□

Lemma 5.9. *For each expression p over $\Sigma(\mathrm{PC}_L)$ there is a normal form $\tilde{p}$ over $\Sigma(\mathrm{PC}_L)$ such that*

$$\mathrm{PC}_L \vdash p = \tilde{p}.$$

The convention is to say that p is in *normal form iff $p \equiv \tilde{p}$.*

Proof. Easy, use Lemma 5.7. □

[12] In a setting with recursion the *call* operator may become more problematic: as recursion cannot be removed in general, the *call* operator cannot be eliminated from recursively specified procedures.

Normal forms over $\Sigma(\mathrm{PC}_L)$ are not unique, due to the fact that different ways of normalising an expression p may lead to different configurations of the special 'ζ actions'. It is, however, important to note that if we rename some 'ζ actions' in two normal forms of p, say $\tilde{p}$ and $\tilde{p}'$, then we can obtain $\tilde{p} = \tilde{p}'$.

We think that it is also possible to define a language PC_C, based on PC_L, with a commutative choice operator '$|_{\mathrm{C}}$'. However, as we argued above, such a choice operator is not compatible with the rule (Call3), so we cannot simply replace $|_{\mathrm{L}}$ by $|_{\mathrm{C}}$. It seems necessary to add another commutative operator, say $+_{\mathrm{C}}$, to model the non-determinism of '$|_{\mathrm{C}}$'. For instance by a rule

$$p \,|_{\mathrm{C}}\, q = p \,|_{\mathrm{L}}\, q +_{\mathrm{C}} q \,|_{\mathrm{L}}\, p.$$

Although a rule like the above looks attractive, and may be interesting enough to be considered more closely, a solid language based on PC_L extended with rules for $+_{\mathrm{C}}$ and '$|_{\mathrm{C}}$' may not be straightforward. We leave this as an open problem.

5.3 A translation of mixed specifications to process algebra

A translation of parts of mixed specifications to process algebra can be achieved by means of the language PC_L. We assume that no **LET** constructs, guard expressions or precondition calls are present in the specification we want to translate, and we only consider mixed specifications with a single set of objects *Type*, and a single set of object names X_T, associated with *Type*. Of course, our ideas can easily be extended to different sets of object names.

Before defining a mapping from PC_L to BPA($\dotplus$), we will describe how mixed specifications can be interpreted in PC_L. The names of COLD-1 procedures (procedures in pre- and postcondition style that form the lower layer of a mixed specification, see Section 3.2) are interpreted as actions $\alpha \in A_{pc}$. The binding expressions $x := t$, with $t \in \mathit{Type}$ and $x \in X_T$, are modelled as a special kind of actions (cf. Section 5.1, where we modelled object name bindings as atomic actions). We will probably be more readily understood if we speak of the set $A_{pc}(X_T)$ as the union of the set A_{pc} and the set of binding expressions associated with X_T and *Type*. Likewise, we write $\mathrm{PC}_L(X_T)$ for the language PC_L, extended with the binding expressions. Procedures specified in the compositional style are interpreted straightforwardly as PC_L expressions, provided that the *call* operator is applied where necessary: every procedure name p occurring in the body of another procedure has to be replaced by $[p]$. The result of this translation is a non-recursive equational specification E_{pc} over $\Sigma(\mathrm{PC}_L(X_T))$. We assume that there is exactly one *root expression* p_0 in E_{pc}, which is an expression that is referred to by none of the other expressions from E_{pc}.

Notation 5.10. Let q be some PROTOCOLD expression, and q' be defined as q with all procedure names replaced by their calls using $[\_]$. A procedure p with a parameter object name x, defined by

$$\begin{array}{lll} \textbf{PROC} & p & : \mathit{Type} \; -> \\ \textbf{IN} & x & \\ \textbf{DEF} & q(x) & \end{array}$$

is interpreted as the expression $p'(x) = q'(x)$ over $\Sigma(\mathrm{PC}_L)$.

We can define a data environment $\mathcal{S}_{pc}(X_T)$ for E_{pc}, which specifies the underlying data structures on which the actions of $A_{pc}(X_T)$ act. For instance, the definitions of pre- and postconditions of COLD-1 procedures are part of $\mathcal{S}_{pc}(X_T)$, as these define the effects on both the states and the enabledness of the actions. However, $\mathcal{S}_{pc}(X_T)$ cannot be interpreted simple as a data environment for $\mathrm{BPA}_{\mathrm{gce}}$, because an action $\alpha \in A_{pc}(X_T)$ may have non-deterministic effects. We will not go more deeply into the definition of $\mathcal{S}_{pc}(X_T)$. A pair $(E_{pc}, \mathcal{S}_{pc}(X_T))$ thus models the operational behaviour of some given mixed specification.

Definition 5.11. A translation

$$T : \Sigma(\mathrm{PC}_L(X_T)) \rightarrow \Sigma(\mathrm{BPA}(\dotplus))$$

of the root expression p_0 of a specification E_{pc} to $\mathrm{BPA}(\dotplus)$ basically consists of the following steps.

1. A normalisation procedure, which consists of the manipulation of the equations of E_{pc} according to the rules of $\mathrm{PC}_L(X_T)$. This results in normal forms, see Definition 5.8.

2. A simultaneous translation of the set of actions $A_{pc}(X_T)$ to a set $A(X_T)$, and of the data environment $\mathcal{S}_{pc}(X_T)$ to $\mathcal{S}(X_T)$. There are two important restrictions on this. First, if an action $\alpha \in A_{pc}(X_T)$ has a non-deterministic *effect* function associated with it, it has to be translated to a *sum* of actions from $A(X_T)$, such that each action in this sum has a deterministic effect. Second, for all object names $x \in X_T$ a special action *clear*(x) has to be added to $A(X_T)$ (see Section 5.1). More formally: for each action $\alpha \in A_{pc}(X_T)$ we either have

$$\begin{array}{l} T(\alpha) \in A(X_T) \quad \text{or} \\ T(\alpha) \quad \text{is a sum of actions from } A(X_T), \text{ and we have} \\ T(\mathbf{FAIL}) \stackrel{\text{def}}{=} \delta. \end{array}$$

   For all $x \in X_T$ and $t \in \mathit{Type}$ we define

$$T(x := t) \stackrel{\text{def}}{=} \langle x := t \rangle.$$

3. A transformation of $A(X_T)$ to $A_{flag}(X_T)$, and of $\mathcal{S}(X_T)$ to $\mathcal{S}_{flag}(X_T)$. Note the importance of defining the special actions ζ, used for modelling the contextual behaviour of backtracking in PROTOCOLD, as

$$\mathit{Pass}(\zeta) \stackrel{\text{def}}{=} \mathbf{true}.$$

4. A translation of expressions over $\Sigma(\mathrm{PC}_L(X_T))$ into terms over $\Sigma(\mathrm{BPA}(\dotplus))$. This only makes sense if normal forms are considered: in Section 4.2 it was argued that the distributivity of ';' over '$|_{\mathrm{L}}$' causes the global character of backtracking in PROTOCOLD expressions. By calculating the result of all possible distributions of ';' over '$|_{\mathrm{L}}$', the scope of the choice operator '$|_{\mathrm{L}}$' (and thus of backtracking) is maximised. Let $\alpha \in A_{pc}(X_T)$ and let p, q be normal forms over $\Sigma(\mathrm{PC}_L(X_T))$, then

$$\begin{array}{l} T(\alpha; p) \stackrel{\text{def}}{=} T(\alpha) \cdot T(p), \\ T(p \mid_{\mathrm{L}} q) \stackrel{\text{def}}{=} T(p) \dotplus T(q). \end{array}$$

5. A small technicality. We required in Section 4.2 that all possible binding effects had to be undone in case of failure. This is achieved by placing the root expression p_0 of the specification E_{pc} in a context $\_ \dotplus \delta$:

$$T(E_{pc}) \stackrel{\text{def}}{=} T(p_0) \dotplus \delta.$$

It is easy to see that the translation T respects the rules of $\mathrm{PC}_L(X_T)$, e.g., if p is a normal form then **FAIL** $|_{\mathrm{L}}\ p$ is a normal form, and if we use axiom Ba1 from [BPW93] we derive

$$T(\mathbf{FAIL} \mid_{\mathrm{L}} p) = T(\mathbf{FAIL}) \dotplus T(p) = \delta \dotplus T(p) \stackrel{Ba1}{=} T(p).$$

□

Some simple examples of PROTOCOLD specifications with backtracking can be found in [CDJ90], where a specification of a television set is given. Unfortunately, we cannot handle these examples yet with the means described in this paper, e.g., we did not treat *precondition calls* and **LET** constructs, which are needed for these specific examples. However, in the example below we translate some simple procedures with backtracking to BPA($\dotplus$).

Example 5.12. Let A, B and $C(y)$ be procedures in the pre- and postcondition style, and p and q be specified in the algorithmic style. We assume that A, B, $C(y)$ are translated to the A_{pc} actions α, β and $\gamma(y)$ respectively, and p and q to p' and q'. $T(\alpha) \stackrel{\text{def}}{=} a$, $T(\beta) \stackrel{\text{def}}{=} b$, $T(\gamma(x)) \stackrel{\text{def}}{=} c(x)$ and $T(\zeta) \stackrel{\text{def}}{=} z$.

$$\begin{array}{ll}
\mathbf{PROC} & p : \mathit{Type} \ -> \\
\mathbf{IN} & x \\
\mathbf{DEF} & x := 1; A \mid_{\mathrm{L}} x := 2; B
\end{array}$$

$$\begin{array}{ll}
\mathbf{PROC} & q : \mathit{Type} \ -> \\
\mathbf{IN} & y \\
\mathbf{DEF} & p(y); C(y)
\end{array}$$

The first step is the translation of the procedures above to PC_L using Notation 5.10:

$$\begin{array}{lcl}
p'(x) & = & x := 1; [\alpha] \mid_{\mathrm{L}} x := 2; [\beta], \\
q'(y) & = & [p'(y)]; [\gamma(y)].
\end{array}$$

The following step is the normalisation of these equations using the rules of PC_L:

$$p'(x) \stackrel{Call1}{=} x := 1; \alpha \mid_{\mathrm{L}} x := 2; \beta$$

The above expression for $p'(x)$ can be substituted in the expression for $q'(x)$:

$$
\begin{aligned}
q'(y) &\overset{Call1}{=} [p'(y)];\gamma(y) \\
&= [y := 1;\alpha \mid_{\mathrm{L}} y := 2;\beta];\gamma(y) \\
&\overset{Call3}{=} ([y := 1;\alpha];\zeta \mid_{\mathrm{L}} \zeta;[y := 2;\beta]);\gamma(y) \\
&\overset{Call2}{=} (y := 1;[\alpha];\zeta \mid_{\mathrm{L}} \zeta;y := 2;[\beta]);\gamma(y) \\
&\overset{Call1}{=} (y := 1;\alpha;\zeta \mid_{\mathrm{L}} \zeta;y := 2;\beta);\gamma(y) \\
&\overset{Distr}{=} y := 1;\alpha;\zeta;\gamma(y) \mid_{\mathrm{L}} \zeta;y := 2;\beta;\gamma(y)
\end{aligned}
$$

We finally translate the above normal form of $q'(y)$ to BPA($\dotplus$) by applying the translation function T given in Definition 5.11. Because we do not translate a complete equational specification, the context $\_ \dotplus \delta$ is omitted.

$$
\begin{aligned}
T(q'(y)) &= T(y := 1;\alpha;\zeta;\gamma(y) \mid_{\mathrm{L}} \zeta;y := 2;\beta;\gamma(y)) \\
&= T(y := 1;\alpha;\zeta;\gamma(y)) \dotplus T(\zeta;y := 2;\beta;\gamma(y)) \\
&= T(y := 1)\cdot T(\alpha)\cdot T(\zeta)\cdot T(\gamma(y)) \dotplus T(\zeta)\cdot T(y := 2)\cdot T(\beta)\cdot T(\gamma(y)) \\
&= \langle y := 1\rangle\cdot a\cdot z\cdot c(y) \dotplus z\cdot\langle y := 2\rangle\cdot b\cdot c(y)
\end{aligned}
$$

We show how evaluations of $T(q'(y))$ model the execution of the procedure q. For example, let the precondition of A be **FALSE** and those of B and $C(y)$ be **TRUE**. Then we have $enabled(a) = \mathbf{false}$ and $enabled(b) = enabled(c(y)) = \mathbf{true}$. Note that we may replace a by δ because for all $a \in A_{flag}$ we have $a = enabled(a) :\rightarrow a$. Let the set S of data-states be given by the set of Booleans $\{\mathbf{true}, \mathbf{false}\}$, s_o be the state in which y is in its initial binding state, and b_z a variable over S. The special action z is defined as follows:

$$
\begin{aligned}
action(z, \mathbf{true}) &\overset{\mathrm{def}}{=} z \\
action(z, \mathbf{false}) &\overset{\mathrm{def}}{=} \delta \\
effect(z, b_z) &\overset{\mathrm{def}}{=} \mathbf{false} \\
eval(enabled(z), b_z) &\overset{\mathrm{def}}{=} \mathbf{true}
\end{aligned}
$$

Let the *action* and *effect* functions on a, b and $c(y)$ (for all $y \in Type$) be defined as the identity. We derive the following evaluation of $T(q'(y))$:

$$\lambda_{(\mathbf{true},s_0)}(T(q'(y)))$$

$$= \lambda_{(\mathbf{true},s_0)}(\langle y := 1\rangle \cdot \delta + z \cdot \langle y := 2\rangle \cdot b \cdot c(y))$$

$$= \langle y := 1\rangle \cdot \lambda_{(\mathbf{true},s_0[\langle 1,bound,push(binding,\emptyset_H)\rangle_y/t(y)])}(\delta + clear(y) \cdot z \cdot \langle y := 2\rangle \cdot b \cdot c(y))$$

$$= \langle y := 1\rangle \cdot \lambda_{(\mathbf{true},s_0[\langle 1,bound,push(binding,\emptyset_H)\rangle_y/t(y)])}(clear(y) \cdot z \cdot \langle y := 2\rangle \cdot b \cdot c(y))$$

$$= \langle y := 1\rangle \cdot clear(y) \cdot \lambda_{(\mathbf{true},s_0)}(z \cdot \langle y := 2\rangle \cdot b \cdot c(y))$$

$$= \langle y := 1\rangle \cdot clear(y) \cdot z \cdot \lambda_{(\mathbf{false},s_0)}(\langle y := 2\rangle \cdot b \cdot c(y))$$

$$= \langle y := 1\rangle \cdot clear(y) \cdot z \cdot \langle y := 2\rangle \cdot \lambda_{(\mathbf{false},s_0[\langle 2,bound,push(binding,\emptyset_H)\rangle_y/t(y)])}(b \cdot c(y))$$

$$= \langle y := 1\rangle \cdot clear(y) \cdot z \cdot \langle y := 2\rangle \cdot b \cdot \lambda_{(\mathbf{false},s_0[\langle 2,bound,push(binding,\emptyset_H)\rangle_y/t(y)])}(c(y))$$

$$= \langle y := 1\rangle \cdot clear(y) \cdot z \cdot \langle y := 2\rangle \cdot b \cdot c(y)$$

We now evaluate a different specification by defining the precondition of A as **TRUE** and that of $C(y)$ as **FALSE**. So $enabled(a) = \mathbf{true}$ and $enabled(c(y)) = \mathbf{false}$. We derive the following evaluation of $T(q'(y))$:

$$\lambda_{(\mathbf{true},s_0)}(T(q'(y)))$$

$$= \lambda_{(\mathbf{true},s_0)}(\langle y := 1\rangle \cdot a \cdot z \cdot c(y) + z \cdot \langle y := 2\rangle \cdot b \cdot c(y))$$

$$= \langle y := 1\rangle \cdot \lambda_{(\mathbf{true},s_0[\langle 1,bound,push(binding,\emptyset_H)\rangle_y/t(y)])}(a \cdot z \cdot \delta + clear(y) \cdot z \cdot \langle y := 2\rangle \cdot b \cdot \delta)$$

$$= \langle y := 1\rangle \cdot a \cdot \lambda_{(\mathbf{true},s_0[\langle 1,bound,push(binding,\emptyset_H)\rangle_y/t(y)])}(z \cdot \delta + clear(y) \cdot z \cdot \langle y := 2\rangle \cdot b \cdot \delta)$$

$$= \langle y := 1\rangle \cdot a \cdot z \cdot \lambda_{(\mathbf{false},s_0[\langle 1,bound,push(binding,\emptyset_H)\rangle_y/t(y)])}(\delta + clear(y) \cdot z \cdot \langle y := 2\rangle \cdot b \cdot \delta)$$

$$= \langle y := 1\rangle \cdot a \cdot z \cdot \lambda_{(\mathbf{false},s_0[\langle 1,bound,push(binding,\emptyset_H)\rangle_y/t(y)])}(clear(y) \cdot z \cdot \langle y := 2\rangle \cdot b \cdot \delta)$$

$$= \langle y := 1\rangle \cdot a \cdot z \cdot clear(y) \cdot \lambda_{(\mathbf{false},s_0)}(z \cdot \langle y := 2\rangle \cdot b \cdot \delta)$$

$$= \langle y := 1\rangle \cdot a \cdot z \cdot clear(y) \cdot \delta$$

We can explain this evaluation in terms of the original mixed specification on page 23. Given the preconditions of A and $C(y)$, the procedure $q(y)$ is executed as follows: $q(y)$ calls $p(y)$ with a free object name y. The body of the procedure p is entered and y is bound to 1 ($\langle y := 1\rangle$). Then the procedure A is successfully executed (a) because its precondition is **TRUE**. The body of p is left (z) and a deadlock is met (first δ). Then the binding effect of p on y is undone ($clear(y)$), and, because the procedure p cannot be re-entered again, (z is not enabled anymore), the expression $q(y)$ fails (δ). □

6 Concluding remarks

A study of the operational behaviour of the specification language PROTOCOLD has been conducted by analysing simple language constructs in an algebraic way. In particular the backtracking mechanism of PROTOCOLD was studied extensively. We argued that, to a certain degree, the execution of a mixed specification can be regarded as the evaluation of a process term over BPA($\dotplus$).

For the purpose of translating parts of mixed specifications to BPA($\dotplus$), a small language PC_L was defined for the manipulation of PROTOCOLD-like expressions, which preserves some typical, operational equivalences. An informal interpretation of mixed specifications was given in PC_L, and in a more formal way a mapping was defined of $\Sigma(\mathrm{PC}_L)$ expressions (normal forms) to $\Sigma(\mathrm{BPA}(\dotplus))$ terms.

The following two properties of backtracking that are preserved by this translation are the (as we called it) contextual behaviour of backtracking in PROTOCOLD, as well as the fact that backtracking is not limited to the scope of the choice operator '$|$' (a fact that makes a canonical mapping from PC_L to BPA($\dotplus$) impossible).

A special data environment for BPA($\dotplus$) had to be defined in order to capture the very specific nature of the binding of object names, and the undoing of such bindings. We think that Example 5.4 nicely illustrates how backtracking over bindings works in BPA($\dotplus$), and how backtracking in PROTOCOLD can be interpreted.

Another interesting problem in PROTOCOLD is that two different operational interpretations of the choice operator are conceivable: a deterministic version '$|_{\mathrm{L}}$', and the non-deterministic '$|_{\mathrm{C}}$'. We restricted ourselves mainly to the deterministic left choice, because it looks as if '$|_{\mathrm{C}}$' is a much more complicated operator for which a richer language than PC_L has to be defined.

We also restricted our study to a small subset of PROTOCOLD. One argument is that we think that we have already traced some important problems of backtracking in PROTOCOLD this way. The second argument is that for a more thorough treatment of some parts of PROTOCOLD, particularly that with the assertions, the formalism BPA($\dotplus$) may not be expressive enough. The reader is reminded of the fact that assertions are highly dependent on abstract data structures, as they can be specified in PROTOCOLD. For instance, assertions can consist of object name bindings, and all kinds of Boolean expressions, such as equality between terms, expressions with the functions $<$, $>$, etc.

In addition to assertions, we also did not provide a thorough treatment of precondition calls and guard expressions (which are directly dependent on the assertions). For procedures in pre- and postcondition style, the precondition calls can be interpreted as straightforward tests on enabledness. For procedures in algorithmic style, the evaluation of precondition calls can possibly be modelled in BPA($\dotplus$) by processes that have the potential for transforming data-states (simply by causing binding effects, without side effects). We also argued that the evaluation of assertions (in guard expressions) displays process-like behaviour. This is best illustrated by the comparison of **AND** with ';' and **OR** with '$|$'. Moreover, as is the case with precondition calls, guard expressions cannot cause side effects, but can cause binding effects. Consequently, in both types of PROTOCOLD expressions backtracking over binding effects can occur (backtracking is also here being viewed as contextual). However, we are quite confident that the execution of precondition calls and guard expressions can be modelled in a similar way as the execution of procedures.

References

[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.

[BBP93] J.A. Bergstra, I. Bethke and A. Ponse. Process algebra with iteration. Report P9314, University of Amsterdam, 1993.

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

[BPW93] J.A. Bergstra, A. Ponse and J.J. van Wamel. Process algebra with backtracking. Report P9306, University of Amsterdam, 1993.

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

[CDJ90] W. Christis, J.A. Droppert and H.B.M. Jonkers. *SPRINT Specification of FL1.* Technical report RWR-113-hj-90135-hj, Philips Research Laboratories, 1990.

[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 Cliffs, 1976.

[Dro92] J.A. Droppert. Protocold 1.1 semantics. Technical report (draft version), Philips Research Laboratories, 1992.

[FJ92a] L.M.G. Feijs and H.B.M. Jonkers. *Specification and design using COLD-1.* Technical report RWB-508-re-92216, Philips Research Laboratories, 1992.

[FJ92b] L.M.G. Feijs and H.B.M. Jonkers. *Formal Specification and Design.* Cambridge University Press, 1992.

[GP90] J.F. Groote and A. Ponse. Process algebra with guards. Report CS-R9069, CWI, Amsterdam, 1990. To appear in *Formal Aspects of Computing.*

[Jon90] H.B.M. Jonkers. *The SPRINT Method.* Technical report RWR-113-hj-90071-hj, Philips Research Laboratories, 1990.

[Jon91a] H.B.M. Jonkers. *Protocold 1.1 User Manual.* Technical report RWR-513-hj-91080-hj, Philips Research Laboratories, 1991.

[Jon91b] H.B.M. Jonkers. *Protocold 1.1 Reference Manual.* Technical report RWR-508-re-91094, Philips Research Laboratories, 1991.

[Jon92] H.B.M. Jonkers. *An overview of the SPRINT method.* Technical report RWB-508-re-92358, Philips Research Laboratories, 1992.

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