An Algebra for Process Creation

Jos C.M. Baeten

Frits W. Vaandrager

Department of Software Technology,
Centre for Mathematics and Computer Science,
P.O.Box 4079, 1009 AB Amsterdam, The Netherlands.

In this paper, we study the issue of process creation from an algebraic perspective. The key to our approach, which is inspired by the work of AMERICA & DE BAKKER [AB], consists of giving a new interpretation to the operator symbol · (sequential composition) in the axiom system BPA of BERGSTRA & KLOP [BK1,2,3]. We present a number of other models for BPA and show how the new interpretation of · naturally generalises the usual interpretation in BPA or ACP. We give an operational semantics based on Plotkin style inductive rules, and give a complete finite axiomatisation of the associated bisimulation model.

1. INTRODUCTION.

In process algebra theories like CCS, CSP, MEIJE and ACP, not much attention has been paid so far, to the concept of process creation. Instead, parallel composition is used as a primitive constructor of concurrent systems. In SMOLKA & STROM [SS] and VAANDRAGER [VA], process algebra semantics is given for languages with process creation (NIL resp. POOL), but there the process creation construct is translated to an architectural expression with parallel composition.

A first attempt to deal more directly with process creation in an algebraic setting is described in BERGSTRA [B], where the axiomatic system ACP is extended with a mechanism for process creation. The key axiom here is

$$E_{\phi}(cr(d)\cdot x) = \overline{cr}(d)\cdot E_{\phi}(\phi(d) \parallel x).$$

The operator E_{ϕ} denotes an environment in which process creation can take place. If an action $cr(d)$ is performed in this environment, a process $\phi(d)$ is created and placed in parallel with the remaining process.

Since process creation is an important concept, present for instance in ADA, NIL, POOL and UNIX, it seems worthwhile to look for a more direct and compositional treatment of process creation which does not need a global environment like an E_{ϕ}-operator. Here, we profit to a large extent of the work of AMERICA & DE BAKKER [AB]. The simple but crucial observation which they make, is that in order to give a compositional semantics to process creation, one has to interpret the sequential composition differently. As an example consider the expression

$$a\cdot new(b\cdot c)\cdot d.$$

Both authors are sponsored by ESPRIT project 432, METEOR (A formal integrated approach to industrial software development), and RACE project 1046, SPECS (Specification and Programming Environment for Communication Software).

The intuitive semantics of this expression is a process which first performs a, after which a new process is created doing b followed by c. The newly created process executes in parallel with the continuation d. Thus, the traces of this process are abcd, abdc and adbc. Consequently, we cannot interpret $x \cdot y$ as 'first do x and then y' (as is usual), because in a setting with process creation process x may continue after process y has started.

For this reason, in AMERICA & DE BAKKER [AB], a new semantical operator : is introduced, which serves as the interpretation of $\cdot$ in a setting with process creation. In the algebra for process creation that we present in this paper, we will interpret the $\cdot$ as a *continuation* operator in essentially the same way as in [AB]. But before we come to this operator, we first give an extensive overview of a number of other interpretations of $\cdot$. What all these interpretations have in common with the continuation operator, is that in a setting with alternative composition (+), they all satisfy the axioms of BPA (Basic Process Algebra) of BERGSTRA & KLOP [BK1,2, 3]:

$$x + y = y + x \qquad (x + y) \cdot z = x \cdot z + y \cdot z$$
$$(x + y) + z = x + (y + z) \qquad (x \cdot y) \cdot z = x \cdot (y \cdot z)$$
$$x + x = x.$$

Most of the discussion of this paper takes place in the setting of interleaving semantics. However, we show that a particular interpretation of $\cdot$ as *sequential composition* (like in ACP) and also our interpretation of $\cdot$ as continuation, can both be lifted in a natural way to the world of *event structures* of WINSKEL [W]. In both these interpretations, we have an instance of *action refinement* in the sense of [CDP] and [GG]. In fact, and this is surprising, sequential composition and continuation have the *same* definition on event structures, only sequential composition is defined on a more restricted domain of processes. Hence the rather substantial differences between the two operators on the level of interleaving semantics almost disappear on the level of True concurrency.

Whereas in AMERICA & DE BAKKER [AB] operational as well as denotational models are presented (and proven to be equivalent), we concentrate on operational models in this paper. As is done in [AB], we use Plotkin-style rules for the operational semantics. There are a number of differences, however.

First, we want all rules to be as simple as possible, and each rule should embody a clear intuition about a certain operator. Therefore, we reject a rule like

$$\langle ..., (s_1;s_2);r, ...,w\rangle \rightarrow \langle ..., s_1;(s_2;r), ..., w\rangle,$$

which occurs in [AB]: we think it is not part of a natural operational intuition about the ;-operator that brackets can move to the right.

A second design criterion that we used in the construction of our operational semantics is that all rules should be in the *tyft/tyxt* format of GROOTE & VAANDRAGER [GV]. This format poses certain restrictions on the inductive rules which guarantee that bisimulation equivalence is a congruence. Thus, any set of rules in *tyft/tyxt* format immediately induces an abstract compositional semantics. In [GV] it is shown that this format cannot be generalised in any obvious way, unless one is willing to work in a setting of terms over a many sorted signature, or use rules with negative hypotheses.

Our third design criterion was that the transition systems generated by the inductive rules should contain no silent or internal steps. If such transitions are present, one is more or less forced to say something about the nature of τ and to choose whether one adopts all of Milner's τ-laws or only a few of them. We prefer to separate the issue of abstraction from other concerns.

A final design criterion is upward compatibility with a non-interleaved event structure semantics. By now, many algebraic concurrency languages have been provided with a non-interleaved semantics (see e.g. [DDM], [O], [BC], [W]). We think that a general requirement for an interleaved semantics of a concurrent language is that there exists a natural non-interleaved semantics which is compatible with it. More specifically, we require that there is an event structure semantics with this property. The idea is that event structures (see WINSKEL [W]) constitute one of the most important domains for 'True' concurrency, and one must have a good reason to present a semantics which is incompatible with them. We show how some proposals for an operational semantics can be discarded because it is unclear how they could meet this last requirement.

In section 3, we present operational rules for a simple language APC for concurrent communicating processes with process creation. We claim that these rules meet all the requirements above. An interesting feature of these rules is that one of them has a look-ahead of more than one action: in order to compute the initial transitions of process $x \cdot y$, one needs information about the first *two* transitions of x. This implies in particular, that our $\cdot$ operator is not definable in terms of CCS, CSP, MEIJE or ACP.

In section 4, we present a sound and complete axiomatisation of the bisimulation semantics induced by the rules for APC. This axiomatisation uses a number of auxiliary operators.

With a number of examples, we illustrate in section 5 how APC can be used to specify concurrent systems, and how identities between processes can be proved algebraically.

2. BASIC PROCESS ALGEBRA.

2.1 The aim of this paper is to give an algebraic treatment of the feature of process creation. It will turn out that the key to our solution consists of giving a new interpretation to the operator symbol $\cdot$ in the axiom system BPA (Basic Process Algebra) of BERGSTRA & KLOP [BK2,3]. Therefore, we start with a review of BPA. We will see that there exist at least five very different interpretations of the operator symbol $\cdot$. One thing that all these interpretations have in common is that the laws of BPA are satisfied, and this similarity may be considered as a surprising fact.

2.2 BPA starts from a given set A of **actions**. These actions, denoted by a,b,c,... are constants in the language. Further, BPA has two binary operators: **sum**, denoted +, and **product**, denoted $\cdot$. Processes x,y,... constructed with these operators will always satisfy the axioms in the following equational specification BPA.

$x + y = y + x$	A1
$(x + y) + z = x + (y + z)$	A2
$x + x = x$	A3
$(x + y) \cdot z = x \cdot z + y \cdot z$	A4
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	A5

TABLE 1. BPA.

Of all operators, $\cdot$ will always bind the strongest, and + the weakest. Thus, $x \cdot y + z$ means $(x \cdot y) + z$. We often write xy instead of $x \cdot y$. We denote the set of closed terms over BPA by $\mathbb{T}(BPA)$.

2.3 In the work of BERGSTRA & KLOP [BK2,3], the elements of A are often called **atomic actions**, the + operator is called **alternative composition**, and the $\cdot$ operator is called **sequential composition**. The intuition is that actions a,b,... are events without positive duration in time; they are atomic and instantaneous. The interpretation of $(a + b) \cdot c$ is a process that first chooses between executing a or b and, second, performs the action c after which it is finished. Since time has a direction, product is not commutative; but sum is, and in fact it is stipulated that the options possible in each state always form a *set* (axioms A1, A2, A3). The other distributive law $x(y + z) = xy + xz$ is not included, because the moment of choice between y and z in the two processes is different.

We would like to stress again that this is just one possible interpretation of the elements of the signature of BPA; we will consider other interpretations in the sequel.

2.4 SEQUENCING.
We will now present our first model for BPA. Like all models in this paper, it is defined using **structural rules** in the style of PLOTKIN [PL]. We introduce, for each constant $a \in A$, a binary **action relation** $\xrightarrow{a}$ on terms in $\mathbb{T}(BPA)$. The intended meaning of $x \xrightarrow{a} y$ is that process x may perform an a-action, and thereby evolve into process y.

In order to define the model, we have to extend the signature of BPA with an auxiliary constant δ. We denote the set of closed terms over this extended signature by $\mathbb{T}(BPA_\delta)$. In the ACP framework of Bergstra & Klop, δ is called **deadlock**. This name suggests a particular intuition about the behaviour of this process which is not in accordance with the interpretation of δ in our first model. Rather, δ plays the same role as NIL of CCS (see MILNER [M]) or STOP of CSP (see HOARE [H]).

The model we consider here, interprets $\cdot$ as **sequencing**: $x \cdot y$ starts with the execution of x, and if x can do no more actions, then execution of y starts. In all models that we present in this paper, the constant δ is characterised by being unable to perform any actions, i.e. $\delta \xrightarrow{a} x$ for no a,x. We write $x \not\rightarrow$ to denote that x has no outgoing transition (so we have $\delta \not\rightarrow$).

We define the predicates $\xrightarrow{a}$ inductively by means of the rules in table 2.

$$a \xrightarrow{a} \delta$$

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

$$\frac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y} \qquad \frac{x \not\rightarrow \quad y \xrightarrow{a} y'}{x \cdot y \xrightarrow{a} y'}$$

TABLE 2. Action relations for BPA with sequencing.

2.5 A problem with this definition is the appearance of negative premisses. This makes that it is not immediately clear that there exists a distinguished transition relation agreeing with the rules. That such a relation exists in this case is due to the fact that the presence of an outgoing transition of a term only depends on the presence or absence of outgoing transitions from terms of a lower complexity. BLOOM, ISTRAIL & MEYER [BIM] (who also present the above rules for sequencing) observe that negative premisses are needed for the definition of this operator.

We turn the structure of action relations into a model for BPA by means of the notion of bisimulation (due to PARK [PA]).

2.6 DEFINITION. A binary relation R on process expressions is a (strong) **bisimulation** if it satisfies the so-called **transfer property**:

1. if $x \xrightarrow{a} y$ and $R(x,x')$ then there exists y' with $x' \xrightarrow{a} y'$ and $R(y,y')$ (for all labels a);
2. conversely, if $x' \xrightarrow{a} y'$ and $R(x,x')$ then there exists y with $x \xrightarrow{a} y$ and $R(y,y')$.

Two processes x,x' are called **bisimilar**, notation $x \leftrightarrow x'$, if there exists a bisimulation R with $R(x,x')$.

Then, the following theorems are standard:

2.7 THEOREM. Bisimulation is a congruence relation on $\mathbb{T}(BPA_\delta)$.

2.8 THEOREM. BPA is a complete axiomatisation of $\mathbb{T}(BPA)/\leftrightarrow$, i.e. for all terms s,t from $\mathbb{T}(BPA)$ we have

$$BPA \vdash s=t \quad \Leftrightarrow \quad \mathbb{T}(BPA)/\leftrightarrow \ \models s=t \quad \Leftrightarrow \quad s \leftrightarrow t.$$

2.9 Notice that theorem 2.8 only talks about terms from $\mathbb{T}(BPA)$, so terms not involving δ. As was already remarked by BERGSTRA & KLOP [BK1], axiom A4 is not valid any more on $\mathbb{T}(BPA_\delta)$ (using the valid axiom $\delta \cdot x = x$, we can derive $a \cdot b = (a + \delta) \cdot b = a \cdot b + \delta \cdot b = a \cdot b + b$). Therefore, if we want to extend theorem 2.8 to the case with δ, we have to restrict A4.

2.10 THEOREM. Let A4*, A6 and A7* be the following axioms:

$(ax + by + y')z = axz + (by + y')z$	A4*
$x + \delta = x$	A6
$\delta \cdot x = x$	A7*.

Then A1,2,3,4*,5,6,7* form a complete axiomatisation of $\mathbb{T}(BPA_\delta)/\leftrightarrow$, i.e. for all terms s,t from $\mathbb{T}(BPA_\delta)$ we have

$$A1,2,3,4^*,5,6,7^* \vdash s=t \quad \Leftrightarrow \quad \mathbb{T}(BPA_\delta)/\leftrightarrow \ \models s=t \quad \Leftrightarrow \quad s \leftrightarrow t.$$

If one takes a more denotational viewpoint, then Plotkin style rules are just a way to define function between labeled transition systems (process graphs). The last two rules of table 2, for instance, determine the operation of **sequencing**: given two process graphs g and h, $g \cdot h$ is the process graph obtained by appending a copy of h to each endnode of g. Sequencing is a simple and natural operation on process graphs. However, it turns out that in a setting with parallel composition and communication we often want to interpret the operator symbol $\cdot$ differently.

2.11 SEQUENTIAL COMPOSITION.

Consider a process $x \cdot y$, where x describes the behaviour of a system consisting of a number of processors which jointly perform some parallel computation. Then it may occur that at some point during the execution of x a state of **deadlock** is reached, i.e. all processors are waiting for each other, before the computation is finished. Usually, y is not allowed to start in such a situation, even though x has reached a state where no transitions are possible. Process y may start only when process x has terminated *successfully*. When we talk about sequential composition, we assume that there are two termination possibilities: successful termination and unsuccessful termination. The sequential composition of x and y starts with execution of x, followed by the execution of y upon successful termination of x. Now there are at different ways in which we can make this intuition more precise. We will present three alternatives, before focusing on the third alternative. Consecutively, we consider:

a. successful termination as a hidden signal (2.12);
b. successful termination as an attribute of actions (2.13);
c. successful termination with $\surd$-refinement (2.14).

2.12 SUCCESSFUL TERMINATION AS A HIDDEN SIGNAL.

Deadlock is considered as an unsuccessful form of termination. If deadlock is characterised by the absence of any possibility to proceed, it seems natural to introduce a special label to indicate successful termination. This special label is denoted $\surd$ (pronounced 'tick'). Thus, we will have an extra binary relation $\xrightarrow{\surd}$. Next, the behaviour of process $a \in A$ is described by the rules

$$a \xrightarrow{a} \varepsilon \qquad \varepsilon \xrightarrow{\surd} \delta.$$

Here, ε is a new constant symbol denoting the process that terminates immediately and successfully (ε first appears in KOYMANS & VRANCKEN [KV]). We see that the process a first performs an a-transition, and then terminates successfully. The process δ still has no outgoing transitions and therefore corresponds in this setting with the process which terminates immediately but unsuccessfully. Now what rules can we have for the sequential composition operator? First, we note that it would not be correct to have rules like

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

because then we could derive things like

$$(a \cdot b) \cdot c \xrightarrow{a} (\varepsilon \cdot b) \cdot c \xrightarrow{\surd} c \xrightarrow{c} \varepsilon,$$

which are clearly in contrast with the intended semantics of the sequential composition operator. Hence $\surd$-events performed by the first argument of the $\cdot$ operator cannot remain visible.

One possible view on sequential composition, which is taken in CCS (see MILNER [M]), is that $\surd$-events do occur, but that they are 'hidden from our view'. This can be expressed formally by the following rules:

$$\frac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y} \text{ (for } a \in A \cup \{\tau\}) \qquad \frac{x \xrightarrow{\surd} x'}{x \cdot y \xrightarrow{\tau} y}$$

Here τ is the silent move of MILNER [M]. Under this interpretation, the transitions of process $(a \cdot b) \cdot c$ are:

$$(a \cdot b) \cdot c \xrightarrow{a} (\varepsilon \cdot b) \cdot c \xrightarrow{\tau} b \cdot c \xrightarrow{b} \varepsilon \cdot c \xrightarrow{\tau} c \xrightarrow{c} \varepsilon \xrightarrow{\surd} \delta.$$

The introduction of τ leads to a number of difficult questions. For instance, should the process $(a \cdot b) \cdot c$ be considered equal to a process p with transitions

$$p \xrightarrow{a} q \xrightarrow{b} r \xrightarrow{c} s \xrightarrow{\surd} \delta.$$

In this paper we want to deal with *concrete* process algebra only, i.e. we do not want to consider the silent move and different alternatives for its axiomatisation and representation by means of action relations. Therefore, we will not pursue the above view on sequential composition any further in this paper.

2.13 SUCCESSFUL TERMINATION AS AN ATTRIBUTE OF ACTIONS.
In BRINKSMA [BR], a sequential composition operator is presented which is based on the idea that successful termination is a visible attribute of the last action of a process. Slightly simplified, this looks as follows: beside the actions in A, the set of labels also contains the elements of the set $A\surd = \{a\surd \mid a \in A\}$. The new action rules are $(a \in A)$:

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

This approach is comparable to the approach in VAN GLABBEEK [VG] (there, $a \xrightarrow{a\surd} \delta$ is written as $a \xrightarrow{a} \surd$). While this is a viable approach, the problem we have with it is, that there seems to be a mismatch with so-called 'True' concurrency and event structures. Many algebraic concurrency languages can be provided with a non-interleaved semantics. A reasonable criterion, put forward by DEGANO, DE NICOLA & MONTANARI [DDM] and OLDEROG [O], is that the interleaved semantics of a language must be *retrievable* from the non-interleaved semantics. Now consider the operator $\|$ of parallel composition without synchronisation. If we add such an operator to the current setting, the action rules will be

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

With these rules, the transition system for $a \| b$ becomes as shown in fig. 1.

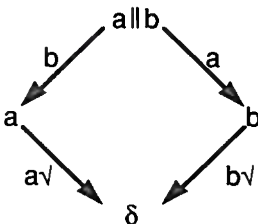

FIGURE 1.

It seems almost unavoidable that in a non-interleaved event structure semantics from which the above interleaving semantics is retrievable, there are 4 events $a, b, a\surd, b\surd$. Furthermore we do not see how to avoid that events a and b are conflicting, whereas event $b\surd$ is causally dependent on event a and event $a\surd$ is causally dependent on event b. But this would be in clear contradiction with the non-interleaved interpretation of $a \| b$ that one expects intuitively, where the two events a and b are not causally related.

Hence, we think that it will be difficult to give a non-interleaved event structure semantics which is compatible with this interpretation of sequential composition.

2.14 SUCCESSFUL TERMINATION WITH √-REFINEMENT.
A third view on sequential composition, the view we prefer, is that we have to do with an instance of **action refinement** as advocated e.g. by CASTELLANO, DE MICHELIS & POMELLO [CDP] and VAN GLABBEEK & GOLTZ [GG]. We again use √-labels to denote successful termination, and assume we have a process domain where √-events have no causal successors and are moreover not concurrent with any other event. This implies that a √-event, if it occurs, will always be the last action performed by a process. On such a domain the sequential composition of processes x and y can be implemented by *refining* every √-event of x to the process y. We refer to [GG] for a formal definition of refinement on the domain of event structures*. Here, we only present the action rules which correspond to the refinement view of sequential composition, in table 3.

$a \xrightarrow{a} \varepsilon$	$\varepsilon \xrightarrow{\surd} \delta$
$\dfrac{x \xrightarrow{u} x'}{x+y \xrightarrow{u} x'}$	$\dfrac{y \xrightarrow{u} y'}{x+y \xrightarrow{u} y'}$
$\dfrac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y}$	$\dfrac{x \xrightarrow{\surd} x' \quad y \xrightarrow{u} y'}{x \cdot y \xrightarrow{u} y'}$

TABLE 3. Sequential composition with action refinement.

In this table (and everywhere in the sequel), u stands for either a or √. This operational semantics can be found in BAETEN & VAN GLABBEEK [BG], only there $x \xrightarrow{\surd} \delta$ was written as $x\downarrow$. The present formulation is due to GROOTE & VAANDRAGER [GV].

Let $\mathbb{T}(BPA_{\delta\varepsilon})$ be the set of closed terms over the signature of BPA extended with the constants δ,ε. Since all rules in table 3 are in the *tyft* format of [GV], bisimulation is a congruence relation on $\mathbb{T}(BPA_{\delta\varepsilon})$. The rules of table 3 induce a model for BPA. In addition, we can also give an axiomatisation for the theory including the constants δ,ε. Let $BPA_{\delta\varepsilon}$ be the theory consisting of BPA together with the axioms in table 4.

$x + \delta = x$	A6
$\delta \cdot x = \delta$	A7
$\varepsilon \cdot x = x$	A8
$x \cdot \varepsilon = x$	A9

TABLE 4. Termination laws.

Thus, δ is the neutral element for alternative composition, ε is the neutral element for sequential composition. A7 is explained, since a deadlocked process can never perform any actions (notice the difference with A7*!). Note that if we added the dis-

* In fact, in [GG], actions are only refined by finite, conflict-free event structures. However, it can be easily seen that in case the actions which are refined have no causal successors, the definition of [GG] can be generalised to general event structures.

tributive law $x(y + z) = xy + xz$, then we could derive $ab = a(b + \delta) = ab + a\delta$, and so a process with no deadlock possibility would be equal to one that may deadlock, a clearly undesirable situation.

Now we have the following theorem, due to BAETEN & VAN GLABBEEK [BG].

2.15 THEOREM. $BPA_{\delta\varepsilon}$ is a complete axiomatisation of $\mathbb{T}(BPA_{\delta\varepsilon})/\leftrightarrows$, i.e. for all terms s,t from $\mathbb{T}(BPA_{\delta\varepsilon})$ we have

$$BPA_{\delta\varepsilon} \vdash s{=}t \quad \Leftrightarrow \quad \mathbb{T}(BPA_{\delta\varepsilon})/\leftrightarrows \ \models s{=}t \quad \Leftrightarrow \quad s \leftrightarrows t.$$

In the next section, we will extend this last view on sequential composition to a setting with process creation.

3. PROCESS CREATION.

3.1 MOTIVATION.

In 2.14, we restricted our attention to a domain of processes where a √-event is always the last event in an execution. This was a natural restriction since · was interpreted as sequential composition and √ as successful termination. Now we would like to consider the operation of √-refinement on a more general domain where √-events still do not have causal successors but with the possibility that a √-event is concurrent with a non-√-event.

These more general processes can for instance arise if one has an operation **new**(x) which removes all √-events in a process and introduces a new √-event which is concurrent with the remaining events of x. In such a setting every process can perform at most one √-event in its lifetime but this is not necessarily the last event. If we interpret a as a process which first does an a-event followed by a √-event, and the operator symbol · as √-refinement, then we can stepwise construct the interpretation of a·(**new**(b·c)·d) as in fig. 2.

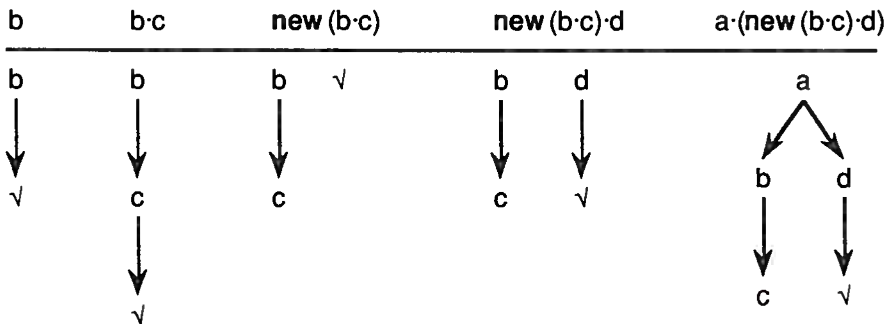

FIGURE 2.

One may think of fig. 2 as a graphical representation of a labeled prime event structure (see [W] or [GG] for the terminology). The arrows denote the causality relation. The traces of this process are

a b c d √   a b d c √   a b d √ c

a d √ b c   a d b √ c   a d b c √

The reader might notice that what we have achieved now is that we have informally given a semantics (essentially an event structure semantics) of a simple language

with process creation. Moreover, this semantics agrees with the intuitions concerning process creation that we presented in the introduction. In fact we claim that the semantics is compatible with the semantics given in AMERICA & DE BAKKER [AB] for a uniform and dynamic language (section 4). In the language of [AB], also alternative composition and μ-recursion are present. We have not described an interpretation of these operators on event structures because we do not want to become too technical here. Such an interpretation, however, is standard and described in many papers (see for instance [W]).

In [AB], the operator : is presented as an operator "which is able to decide dynamically whether it should act as sequential or parallel composition". We prefer a different intuition because we think that the operator : does not introduce a choice or conflict between sequential and parallel composition, but rather that it is a natural generalisation of the sequential composition operator · on a domain of processes where √ may occur in a non-final position.

A surprising thing about the above semantics is that, on the domain of event structures, we give exactly the *same* interpretation to the operator symbol · as in the case of sequential composition. The only difference is that the domain of processes is enlarged. When we work with the extended domain of processes, we will call this operation **continuation** and the √-event the **continuation action** (sequential composition and successful termination is not an appropriate terminology now).

3.2 CONTINUATION

We will now give Plotkin-style rules, which correspond to the above event structure semantics. It turns out that on this level we do have to change the rules for the · operator: since in a product x·y, the process x may continue after y has started, we have to introduce an auxiliary operator $\Vert\!\!\upharpoonright$ for describing those states where y has started but x is not yet finished. See table 5.

$\dfrac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y}$	$\dfrac{x \xrightarrow{\surd} x' \quad y \xrightarrow{u} y'}{x \cdot y \xrightarrow{u} x' \Vert\!\!\upharpoonright y'}$

TABLE 5. Continuation.

We can see, that the second rule here is a generalisation of the corresponding rule in table 3. There, if $x \xrightarrow{\surd} x'$, necessarily $x' \equiv \delta$, and the term $\delta \Vert\!\!\upharpoonright x$ has the same transitions as x.

The reader may think there is a possibility missing here, viz.

$$\frac{x \xrightarrow{\surd} x' \xrightarrow{a} x''}{x \cdot y \xrightarrow{a} x'' \Vert\!\!\upharpoonright y}.$$

However, this rule is not in accordance with our view on sequential composition with refinement of √-events: when y refines the √-event, any action in place of the √-event should involve an initial action of y. Moreover, the proposed rule leads to counter-intuitive behaviour: process x should behave the same as process x·ε, but if x can perform √ and then a, then x·ε can also perform a before √ with the rule above.

3.3 PARALLEL COMPOSITION.

The operator $\Vert\!\!\upharpoonright$ is just parallel composition with the additional restriction that only the process on the right-hand side may perform √-events. This operator is very similar to

the parallel composition operator in the theory ACP of [BK2,3]. In ACP, the parallel composition of x and y can perform a √-event only if both x and y can perform a √-event at the same time. When we compose processes x and y by means of our new combinator $\Vdash$, the composition can do a √-event when y can do a √-event.

With respect to interleaving, $\Vdash$ behaves as one would expect: if one component can perform a certain atomic action, the composition can also perform this action. In table 6, we present the action relation definition for $\Vdash$.

$\dfrac{x \xrightarrow{a} x'}{x \Vdash y \xrightarrow{a} x' \Vdash y}$	$\dfrac{y \xrightarrow{u} y'}{x \Vdash y \xrightarrow{u} x \Vdash y'}$

TABLE 6. Parallel composition.

We should note that all our operators are defined on the domain of processes that is described above. Thus, any composition of processes that have at most one √-event in every execution path, again gives such a process. If one has no objection to operators that lead outside this domain, a symmetric parallel composition can be used, and $x \Vdash y$ is represented by something like $\partial_{\{\surd\}}(x) \| y$ (where $\partial_{\{\surd\}}$ cancels all √'s).

In languages with process creation, parallel composition is mostly not included in the language. It is an auxiliary operator which is present only on a semantical level.

3.4 PROCESS CREATION.
The operator **new** can be defined by:

$$\mathbf{new}(x) = x \Vdash \varepsilon.$$

From this definition, it follows that **new** is characterised by the action rules in table 7.

$\mathbf{new}(x) \xrightarrow{\surd} x \cdot \delta$	$\dfrac{x \xrightarrow{a} x'}{\mathbf{new}(x) \xrightarrow{a} \mathbf{new}(x')}$

TABLE 7. Action rules for process creation.

We claim that this operator is essentially the same construct as the **new** of AMERICA & DE BAKKER [AB] (section 4).

3.5 EXAMPLE. The term $a \cdot \mathbf{new}(b \cdot c) \cdot d$ determines the transition diagram in fig. 3.

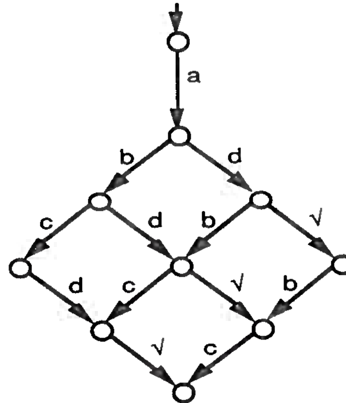

FIGURE 3.

3.6 COMMUNICATION.

Although the rules of tables 5-7 gave a simple and intuitive semantics to process creation, this semantics is not very practical. In any practical language with process creation there must be a possibility of *communication* between a newly created process and the rest of the system. Therefore, we add rules for $\Vert\!\!\urcorner$ and $\cdot$ which express the possibility of communication.

Like in ACP, we have given a partial binary function γ on A, which is commutative and associative, the so-called **communication function**. If $\gamma(a,b) = c$, we say a and b communicate, and the result of the communication is c. If $\gamma(a,b)$ is undefined, we say that a and b do not communicate. In table 8, we present the new rules for $\Vert\!\!\urcorner$ and $\cdot$.

We will not discuss here the consequences of the change in the action rules on the level of non-interleaved event structure semantics. It will be clear that $\cdot$ can no longer be interpreted as just refinement of tick-events. The construction will now introduce a large number of new events which describe possible synchronisations between the original and the new processes.

$$\frac{x \xrightarrow{a} x' \quad y \xrightarrow{b} y'}{x \Vert\!\!\urcorner y \xrightarrow{c} x' \Vert\!\!\urcorner y'} \quad \text{if } \gamma(a,b) = c$$

$$\frac{x \xrightarrow{\surd} x' \xrightarrow{a} x'' \quad y \xrightarrow{b} y'}{x \cdot y \xrightarrow{c} x'' \Vert\!\!\urcorner y'} \quad \text{if } \gamma(a,b) = c$$

TABLE 8. Communication

As before, a,b,c range over A, u ranges over $A \cup \{\surd\}$.

3.7 ENCAPSULATION.

As in ACP (see [BK2,3]), we have the **encapsulation operator** ∂_H (where H is a set of atomic actions), which blocks actions from H. This operator is used to block communications with the environment, and remove 'halves' of communication (actions that should communicate). The action relation definition is straightforward.

$$\frac{x \xrightarrow{u} x'}{\partial_H(x) \xrightarrow{u} \partial_H(x')} \quad \text{if } u \notin H$$

TABLE 9. Encapsulation.

3.8 BISIMULATION.

Since all the action rules presented so far are in the *tyft* format of GROOTE & VAANDRAGER [GV], we can conclude that bisimulation remains a congruence, also with respect to the new operators ⫴, $\cdot$, ∂_H. Thus, if $\mathbb{T}(PC)$ is the set of terms built with the signature of $BPA_{\delta\varepsilon}$ extended with these operators, then $\mathbb{T}(PC)/\leftrightarrows$ is a well-defined structure. In the next section, we will proceed to find a complete axiomatisation for this structure.

4. AXIOMATISATION.

4.1 AUXILIARY OPERATORS.

In order to give a finite axiomatisation for the structure $\mathbb{T}(PC)/\leftrightarrows$ defined in 3.8, we will need some auxiliary operators, comparable to the operators ⌊⌊, | in ACP (see [BK2,3]). Since our parallel composition operator is asymmetric, we will need not two but three auxiliary operators: ⫴⌊, ⌋⫴, ⌈. These three operators will form the three components of the merge operator ⫴: ⫴⌊, the **left-merge**, will give the possibilities that the left-hand side performs an event, ⌋⫴, the **right-merge**, gives the possibilities that the right-hand side performs an event (together, these two operators give the interleaving), and finally, ⌈, the **communication merge**, gives the possibilities that a communication action occurs between the two processes.

The axiomatisation to be presented also uses an additional auxiliary operator $\surd$. The process $\surd(x)$ starts with a $\surd$-event. Next the process x is performed from which however all $\surd$-events have been removed. When no confusion can occur, we will write $\surd x$ instead of $\surd(x)$.

4.2 SIGNATURE.

Now we will present the language for our **Algebra for Process Creation** (APC). As parameters of the language, we have a finite set A of atomic actions, and a partial binary function γ on A, which is commutative and associative. Then, we have constants a (for each $a \in A$), constants δ, ε, binary operators $+, \cdot$, ⫴, ⫴⌊, ⌋⫴, ⌈, a unary operator $\surd$, and unary operators ∂_H (for each $H \subseteq A$).

4.3 AXIOMS.

The axioms of APC are presented in table 10. There, $a, b \in A$, $H \subseteq A$, and x, y, z are arbitrary processes. Notice that the constant ε becomes definable, by axiom PC1.

$x + y = y + x$	A1	$x \Vert\!\upharpoonright y = x \Vert\!\lfloor y + x \rfloor\!\upharpoonright y + x \upharpoonright y$	PCM
$x + (y + z) = (x + y) + z$	A2		
$x + x = x$	A3	$\delta \Vert\!\lfloor x = \delta$	PCL1
$(x + y)z = xz + yz$	A4	$ax \Vert\!\lfloor y = a(x \Vert\!\upharpoonright y)$	PCL2
$(xy)z = x(yz)$	A5	$\surd x \Vert\!\lfloor y = \delta$	PCL3
$x + \delta = x$	A6	$(x + y) \Vert\!\lfloor z = x \Vert\!\lfloor z + y \Vert\!\lfloor z$	PCL4
$\delta x = \delta$	A7		
$\varepsilon x = x$	A8	$x \rfloor\!\upharpoonright \delta = \delta$	PCR1
$x \varepsilon = x$	A9	$x \rfloor\!\upharpoonright ay = a(x \Vert\!\upharpoonright y)$	PCR2
		$x \rfloor\!\upharpoonright \surd y = \surd(x \Vert\!\upharpoonright y)$	PCR3
$\varepsilon = \surd\delta$	PC1	$x \rfloor\!\upharpoonright (y + z) = x \rfloor\!\upharpoonright y + x \rfloor\!\upharpoonright z$	PCR4
$\mathbf{new}(x) \cdot y = x \Vert\!\upharpoonright y$	PC2		
$\surd x = \surd(x\delta)$	PC3	$ax \upharpoonright by = \gamma(a,b) \cdot (x \Vert\!\upharpoonright y)$	
$(\surd x) \cdot y = x \rfloor\!\upharpoonright y + x \upharpoonright y$	PC4	if $\gamma(a,b)$ defined	PCC1
		$ax \upharpoonright by = \delta$ if undefined	PCC2
$\partial_H(\delta) = \delta$	PCD1	$\surd x \upharpoonright y = \delta$	PCC3
$\partial_H(ax) = a \cdot \partial_H(x)$ if $a \notin H$	PCD2	$x \upharpoonright \surd y = \delta$	PCC4
$\partial_H(ax) = \delta$ if $a \in H$	PCD3	$(x + y) \upharpoonright z = x \upharpoonright z + y \upharpoonright z$	PCC5
$\partial_H(\surd x) = \surd(\partial_H(x))$	PCD4	$x \upharpoonright (y + z) = x \upharpoonright y + x \upharpoonright z$	PCC6
$\partial_H(x+y) = \partial_H(x) + \partial_H(y)$	PCD5		

TABLE 10. APC.

When we write a specification in APC, we only use a part of the signature, not the auxiliary operators. Formally, we can declare part of the signature to be *hidden*, as explained e.g. in BERGSTRA, HEERING & KLINT [BHK] or VAN GLABBEEK & VAANDRAGER [VGV].

The visible signature of APC is $\Sigma = \{a \mid a \in A\} \cup \{\delta, \varepsilon, +, \cdot, \mathbf{new}\} \cup \{\partial_H \mid H \subseteq A\}$, whereas the hidden signature contains $\Vert\!\upharpoonright, \Vert\!\lfloor, \rfloor\!\upharpoonright, \upharpoonright, \surd$. We will write all specifications in section 5 in the signature Σ.

4.4 LEMMA. We list some useful identities that can be derived from APC.

i.	$\delta \Vert\!\upharpoonright x = x$	vii.	$\mathbf{new}(\varepsilon) = \varepsilon$
ii.	$\mathbf{new}(x) = x \Vert\!\upharpoonright \varepsilon$	viii.	$(\surd x) \cdot \delta = \delta$
iii.	$\mathbf{new}(\delta) = \varepsilon$	ix.	$\surd(\surd x) = \varepsilon$
iv.	$\varepsilon \Vert\!\lfloor x = \delta$	x.	$(x \Vert\!\upharpoonright y) \cdot z = x \Vert\!\upharpoonright (y \cdot z)$
v.	$\varepsilon \upharpoonright x = x \upharpoonright \varepsilon = \delta$	xi.	$\surd x = x \rfloor\!\upharpoonright \varepsilon$.
vi.	$\delta \upharpoonright x = x \upharpoonright \delta = \delta$		

PROOF. Mostly straightforward. We give proofs for the difficult identities.

i. $\delta \Vert\!\upharpoonright x = \delta \Vert\!\lfloor x + \delta \rfloor\!\upharpoonright x + \delta \upharpoonright x = \delta + \delta \rfloor\!\upharpoonright x + \delta \upharpoonright x =$
$= \delta \rfloor\!\upharpoonright x + \delta \upharpoonright x = (\surd\delta) \cdot x = \varepsilon \cdot x = x$;

vi. $\delta \upharpoonright x = \delta \upharpoonright x + \varepsilon \upharpoonright x = (\delta + \varepsilon) \upharpoonright x = \varepsilon \upharpoonright x = \delta$;

ix. $\surd(\surd x) = \surd((\surd x) \cdot \delta) = \surd\delta = \varepsilon$;

x. $(x \Vert\!\upharpoonright y) \cdot z = (\mathbf{new}(x) \cdot y) \cdot z = \mathbf{new}(x) \cdot (y \cdot z) = x \Vert\!\upharpoonright (y \cdot z)$;

xi. $\surd x = \surd x \cdot \varepsilon = x \rfloor\!\upharpoonright \varepsilon + x \upharpoonright \varepsilon = x \rfloor\!\upharpoonright \varepsilon$.

Note that from axiom PC2 and lemma 4.4.ii it follows that the operators **new** and $\mathbin{\|\!\!\lceil}$ can be defined in terms of each other. Also, by 4.4.xi, the auxiliary operator $\surd$ is definable in terms of $\mathbin{\rfloor\!\rfloor\!\lceil}$ and ε.

4.5 ACTION RELATIONS.
We can also give action relation definitions for the auxiliary operators. We give the full set in table 11.

$$a \xrightarrow{a} \varepsilon \qquad \varepsilon \xrightarrow{\surd} \delta \qquad \surd x \xrightarrow{\surd} x\cdot\delta$$

$$\mathbf{new}(x) \xrightarrow{\surd} x\cdot\delta \qquad \frac{x \xrightarrow{a} x'}{\mathbf{new}(x) \xrightarrow{a} \mathbf{new}(x')}$$

$$\frac{x \xrightarrow{u} x'}{x+y \xrightarrow{u} x'} \qquad \frac{y \xrightarrow{u} y'}{x+y \xrightarrow{u} y'}$$

$$\frac{x \xrightarrow{a} x'}{x\cdot y \xrightarrow{a} x'\cdot y} \qquad \frac{x \xrightarrow{\surd} x' \quad y \xrightarrow{u} y'}{x\cdot y \xrightarrow{u} x' \mathbin{\|\!\!\lceil} y}$$

$$\frac{x \xrightarrow{\surd} x' \xrightarrow{a} x'' \quad y \xrightarrow{b} y'}{x\cdot y \xrightarrow{c} x'' \mathbin{\|\!\!\lceil} y'} \quad \text{if } \gamma(a,b) = c$$

$$\frac{x \xrightarrow{a} x'}{x \mathbin{\|\!\!\lceil} y \xrightarrow{a} x' \mathbin{\|\!\!\lceil} y \qquad x \mathbin{\lfloor\!\!\lfloor\!\lceil} y \xrightarrow{a} x' \mathbin{\|\!\!\lceil} y}$$

$$\frac{y \xrightarrow{u} y'}{x \mathbin{\|\!\!\lceil} y \xrightarrow{u} x \mathbin{\|\!\!\lceil} y' \qquad x \mathbin{\rfloor\!\!\rfloor\!\lceil} y \xrightarrow{u} x \mathbin{\|\!\!\lceil} y'}$$

$$\frac{x \xrightarrow{a} x' \quad y \xrightarrow{b} y'}{x \mathbin{\|\!\!\lceil} y \xrightarrow{c} x' \mathbin{\|\!\!\lceil} y' \qquad x \mathbin{|\!\lceil} y \xrightarrow{c} x' \mathbin{\|\!\!\lceil} y'} \quad \text{if } \gamma(a,b) = c$$

$$\frac{x \xrightarrow{u} x'}{\partial_H(x) \xrightarrow{u} \partial_H(x')} \quad u \notin H$$

TABLE 11. Action relations for APC.

Again, all these action rules are in *tyft* format, so bisimulation remains a congruence. Let us call the set of process expressions over this extended set of operators $\mathbb{T}$(APC). We will prove that the axiom system APC is a complete axiomatisation of $\mathbb{T}$(APC)/$\leftrightarrows$. First, we will need some other results.

4.6 DEFINITION. We define some useful sets of terms.

i. The set of **bottom terms** is defined inductively by:
   - δ is a bottom term;
   - if t,s are bottom terms, then so are $a\cdot t$ and $t + s$.

ii. The set of **basic terms** is defined inductively by:

- δ is a basic term;
- if t is a bottom term, then $\surd t$ is a basic term;
- if t,s are basic terms, then so are at and t + s.

We see that a basic term is a closed term built from the signature δ, +, $\surd$, a, such that a $\surd$ occurs at most once in every execution sequence, and such that we have *only prefix multiplication* (as defined below).

4.7 DEFINITION. We say a term has **only prefix multiplication** if for each subterm of the form $t \cdot s$, t is an atomic action, and s is *not* an atomic action. This means that for these terms, instead of having constants a and general multiplication $\cdot$, we could also use a signature with only unary operators $a\cdot$. Notice that this is the usual situation in CCS (MILNER [M]) and CSP (HOARE [H]).

4.8 LEMMA. Let t be a basic term. Then there exists a bottom term t' such that APC $\vdash t \cdot \delta = t'$.

PROOF: Straightforward induction on the structure of basic terms.

4.9 THEOREM. (*Elimination Theorem*)
Let t be a closed APC-term. Then there exists a basic term t' such that APC $\vdash t = t'$.

PROOF: By an inductive argument, it is enough to prove the following claim:
Let q,q' be basic terms and let p be syntactically equal to δ, ε, a, q+q', $q \cdot q'$, $q \Vert\!\upharpoonright q'$, $q \lfloor\!\lfloor q'$, $q \rfloor\!\rfloor\!\upharpoonright q'$, $q \upharpoonright q'$, $\surd q$, new(q) or $\partial_H(q)$. Then there exists a basic term r such that APC $\vdash$ p=r.
To prove this claim, we use induction on the *size* of a term. We define size inductively by:

- size(δ) = size(ε) = 1
- size(a) = 2
- size(t+t') = size($t \cdot t'$) = size($t \lfloor\!\lfloor t'$) = size($t \rfloor\!\rfloor\!\upharpoonright t'$) = size($t \upharpoonright t'$) = size(t) + size(t')
- size($t \Vert\!\upharpoonright t'$) = size(t) + size(t') + 1
- size($\surd t$) = size(t) + 2
- size(new(t)) = size(t) + 5
- size(∂_H(t)) = size(t) + 1.

In the cases $p \equiv \delta$, $p \equiv q+q'$, we already have the required form.

- $p \equiv \varepsilon$: use PC1;
- $p \equiv a$: use A9 and PC1;
- $p \equiv q \cdot q'$: here we use induction on the structure of q. If $q \equiv \delta$, use A7; if $q \equiv a \cdot q''$, use A5 and the induction hypothesis; if $q \equiv \surd q''$, with q'' a bottom term, use PC4 and the induction hypothesis; if $q \equiv q'' + q^*$, use A4 and the induction hypothesis;
- $p \equiv q \Vert\!\upharpoonright q'$: use PCM and the induction hypothesis;
- $p \equiv q \lfloor\!\lfloor q'$: here we use induction on the structure of q. If $q \equiv \delta$, use PCL1; if $q \equiv a \cdot q''$, use PCL2 and the induction hypothesis; if $q \equiv \surd q''$, use PCL3; if $q \equiv q'' + q^*$, use PCL4 and the induction hypothesis;
- $p \equiv q \rfloor\!\rfloor\!\upharpoonright q'$: here we use induction on the structure of q'. If $q' \equiv \delta$, use PCR1; if $q' \equiv a \cdot q''$, use PCR2 and the induction hypothesis; if $q' \equiv \surd q''$, use PCR3 to write $p = \surd(q \Vert\!\upharpoonright q'')$, by induction $p = \surd q^*$ for some basic q^*, by PC3 $p = \surd(q^* \cdot \delta)$, and then

by lemma 4.8 $p = \surd q^{\circ}$ for some bottom q°; if $q \equiv q'' + q^{*}$, use PCR4 and the induction hypothesis;

- $p \equiv q \upharpoonright q'$: by simultaneous induction on the structure of q and q'; left to the reader;
- $p \equiv \surd q$: use PC3 and lemma 4.8;
- $p \equiv \mathbf{new}(q)$: write $\mathbf{new}(q) = \mathbf{new}(q) \cdot \varepsilon = q \Vert\!\upharpoonright \varepsilon = q \Vert\!\upharpoonright \surd\delta$ and apply induction;
- $p \equiv \partial_H(q)$: similar to $p \equiv q \rfloor\!\upharpoonright q'$; left to the reader.

4.10 THEOREM. (*Soundness Theorem*)
The structure $\mathbb{T}(APC)/\leftrightarroweq$ is a model of APC.

PROOF: To prove the theorem, we need to check that each axiom of APC holds in $\mathbb{T}(APC)/\leftrightarroweq$. As an example, consider axiom A5 (by far the most difficult one!).

Consider the relation R on $\mathbb{T}(APC)$, that relates all terms with themselves, and moreover relates each term of the form $(x \cdot y) \cdot z$ with $x \cdot (y \cdot z)$ (and vice versa), every term $(x \Vert\!\upharpoonright y) \cdot z$ with $x \Vert\!\upharpoonright (y \cdot z)$ (and v.v.), and every term $(x \Vert\!\upharpoonright y) \Vert\!\upharpoonright z$ with $x \Vert\!\upharpoonright (y \Vert\!\upharpoonright z)$ (and v.v.). We claim that R is a bisimulation on $\mathbb{T}(APC)$. To prove this, we need to check that the transfer property holds. This proof has a large number of cases. We will give some of these cases.

In principle, this part of the proof could have been done mechanically also. In fact, the tool ECRINS (see MADELAINE & DE SIMONE [MDS]) has been designed for doing this type of proofs. Unfortunately, ECRINS is not able to deal with Plotkin style rules with a lookahead of more than one, such as the third rule for the $\cdot$ operator.

Suppose from $(x \cdot y) \cdot z$, we can perform a step. This fact is proved by a proof following the rules for $\cdot$ in table 11. Now look at the last step in this proof.

CASE 1. The last step uses the first rule. Thus, $x \cdot y$ can do an a-step. Now look at the last step in the proof of this fact.

SUBCASE 1.1. This last step uses the first rule. Thus x can do an a-step, to a term x', say. We have $x \xrightarrow{a} x'$, and so the steps in the proof were $x \cdot y \xrightarrow{a} x' \cdot y$ and $(x \cdot y) \cdot z \xrightarrow{a} (x' \cdot y) \cdot z$. From the first rule and $x \xrightarrow{a} x'$, we derive immediately that $x \cdot (y \cdot z) \xrightarrow{a} x' \cdot (y \cdot z)$, and $(x' \cdot y) \cdot z$ and $x' \cdot (y \cdot z)$ are again related.

SUBCASE 1.2. This last step uses the second rule. Then, we must have $x \xrightarrow{\surd} x'$ and $y \xrightarrow{a} y'$, and so $x \cdot y \xrightarrow{a} x' \Vert\!\upharpoonright y'$ and $(x \cdot y) \cdot z \xrightarrow{a} (x' \Vert\!\upharpoonright y') \cdot z$. By the first rule, $y \xrightarrow{a} y'$ implies $y \cdot z \xrightarrow{a} y' \cdot z$, and by the second rule, using $x \xrightarrow{\surd} x'$, we derive $x \cdot (y \cdot z) \xrightarrow{a} x' \Vert\!\upharpoonright (y' \cdot z)$. Now $(x' \Vert\!\upharpoonright y') \cdot z$ and $x' \Vert\!\upharpoonright (y' \cdot z)$ are again related.

SUBCASE 1.3. This last step uses the third rule. Then, we must have $x \xrightarrow{\surd} x' \xrightarrow{a} x''$, $y \xrightarrow{b} y'$ and $\gamma(a,b) = c$, whence $x \cdot y \xrightarrow{c} x'' \Vert\!\upharpoonright y'$ and $(x \cdot y) \cdot z \xrightarrow{c} (x'' \Vert\!\upharpoonright y') \cdot z$. By the first rule, $y \xrightarrow{b} y'$ implies $y \cdot z \xrightarrow{b} y' \cdot z$, and by the third rule, using $x \xrightarrow{\surd} x' \xrightarrow{a} x''$, we derive $x \cdot (y \cdot z) \xrightarrow{c} x'' \Vert\!\upharpoonright (y' \cdot z)$. Now $(x'' \Vert\!\upharpoonright y') \cdot z$ and $x'' \Vert\!\upharpoonright (y' \cdot z)$ are again related.

CASE 2. The last step uses the second rule. Thus, $x \cdot y$ can do an $\surd$-step and $z \xrightarrow{u} z'$ for some u, z'. Now the only possibility that $x \cdot y$ can do an $\surd$-step, is as a result of rule 2, with $x \xrightarrow{\surd} x'$ and $y \xrightarrow{\surd} y'$, and so, we had $x \cdot y \xrightarrow{\surd} x' \Vert\!\upharpoonright y'$ and $(x \cdot y) \cdot z \xrightarrow{u} (x' \Vert\!\upharpoonright y') \Vert\!\upharpoonright z'$. By rule 2, using $y \xrightarrow{\surd} y'$ and $z \xrightarrow{u} z'$, we obtain $y \cdot z \xrightarrow{u} y' \Vert\!\upharpoonright z'$, and by rule 2 again, using $x \xrightarrow{\surd} x'$, we obtain $x \cdot (y \cdot z) \xrightarrow{u} x' \Vert\!\upharpoonright (y' \Vert\!\upharpoonright z')$. Now $(x' \Vert\!\upharpoonright y') \Vert\!\upharpoonright z'$ and $x' \Vert\!\upharpoonright (y' \Vert\!\upharpoonright z')$ are again related.

CASE 3. The last step uses the third rule. Thus, $x \cdot y$ can do a $\surd$-step followed by an a-step, $z \xrightarrow{b} z'$ and $\gamma(a,b) = c$ (for some a,b,c,z'). Now, as in case 2, $x \cdot y \xrightarrow{\surd}$ implies $x \xrightarrow{\surd} x'$ and $y \xrightarrow{\surd} y'$ and $x \cdot y \xrightarrow{\surd} x' \|\!\ulcorner y'$. This means that $x' \|\!\ulcorner y'$ can do an a-step. This must be the result of one of the three rules for $\|\!\ulcorner$.

SUBCASE 3.1. The first rule for $\|\!\ulcorner$ was used. Then, $x' \xrightarrow{a} x''$ for some x", and so $x' \|\!\ulcorner y' \xrightarrow{a} x'' \|\!\ulcorner y'$ and $(x \cdot y) \cdot z \xrightarrow{c} (x'' \|\!\ulcorner y') \|\!\ulcorner z'$. By the second rule for $\cdot$, using $y \xrightarrow{\surd} y'$ and $z \xrightarrow{b} z'$, we obtain $y \cdot z \xrightarrow{b} y' \|\!\ulcorner z'$. Then apply the third rule for $\cdot$, using $x \xrightarrow{\surd} x' \xrightarrow{a} x''$, to get $x \cdot (y \cdot z) \xrightarrow{c} x'' \|\!\ulcorner (y' \|\!\ulcorner z)$. Now $(x'' \|\!\ulcorner y') \|\!\ulcorner z$ and $x'' \|\!\ulcorner (y' \|\!\ulcorner z)$ are again related.

SUBCASE 3.2. The second rule for $\|\!\ulcorner$ was used. Then, $y' \xrightarrow{a} y''$ for some y", and so $x' \|\!\ulcorner y' \xrightarrow{a} x' \|\!\ulcorner y''$ and $(x \cdot y) \cdot z \xrightarrow{c} (x' \|\!\ulcorner y'') \|\!\ulcorner z'$. Apply the third rule for $\cdot$, using $y \xrightarrow{\surd} y' \xrightarrow{a} y''$ and $z \xrightarrow{b} z'$, to get $y \cdot z \xrightarrow{c} y'' \|\!\ulcorner z'$. Then use the second rule for $\cdot$ with $x \xrightarrow{\surd} x'$ to obtain $x \cdot (y \cdot z) \xrightarrow{c} x' \|\!\ulcorner (y'' \|\!\ulcorner z')$. Now $(x' \|\!\ulcorner y'') \|\!\ulcorner z'$ and $x' \|\!\ulcorner (y'' \|\!\ulcorner z')$ are again related.

SUBCASE 3.3. The third rule for $\|\!\ulcorner$ was used. Then a is the result of a communication, say between a' and a". We find $x' \xrightarrow{a'} x''$, $y' \xrightarrow{a''} y''$, and so $x' \|\!\ulcorner y' \xrightarrow{a} x'' \|\!\ulcorner y''$ and $(x \cdot y) \cdot z \xrightarrow{c} (x'' \|\!\ulcorner y'') \|\!\ulcorner z'$. Now use the third rule for $\|\!\ulcorner$ with $y \xrightarrow{\surd} y' \xrightarrow{a''} y''$ and $z \xrightarrow{b} z'$, to get $y \cdot z \xrightarrow{\gamma(a'',b)} y'' \|\!\ulcorner z'$. Now notice that by associativity of γ we have that $\gamma(a', \gamma(a'',b)) = c$. Applying this in the third rule for $\|\!\ulcorner$ again, with $x \xrightarrow{\surd} x' \xrightarrow{a'} x''$, leads to $x \cdot (y \cdot z) \xrightarrow{c} x'' \|\!\ulcorner (y'' \|\!\ulcorner z')$. Now $(x'' \|\!\ulcorner y'') \|\!\ulcorner z'$ and $x'' \|\!\ulcorner (y'' \|\!\ulcorner z')$ are again related.

Thus, we see that the transfer property holds from $(x \cdot y) \cdot z$ to $x \cdot (y \cdot z)$. All information, needed to prove the converse implication is available above. In a similar fashion, we can prove the transfer property between $(x \|\!\ulcorner y) \cdot z$ and $x \|\!\ulcorner (y \cdot z)$, and between $(x \|\!\ulcorner y) \|\!\ulcorner z$ and $x \|\!\ulcorner (y \|\!\ulcorner z)$. We conclude that the relation R is a bisimulation, and thus that law A5 holds in $\mathbb{T}(APC)/\leftrightarrow$. Also, we have shown that the laws $(x \|\!\ulcorner y) \cdot z = x \|\!\ulcorner (y \cdot z)$ and $(x \|\!\ulcorner y) \|\!\ulcorner z = x \|\!\ulcorner (y \|\!\ulcorner z)$ hold in $\mathbb{T}(APC)/\leftrightarrow$.

Another interesting case in the soundness proof that we would like to mention is axiom PC4: $(\surd x) \cdot y = x \rfloor\!\ulcorner y + x \ulcorner y$. In the soundness proof of this axiom (similar to, but much simpler than the proof for A5), we need the soundness of the law $x\delta \|\!\ulcorner y = x \|\!\ulcorner y$.

4.11 LEMMA. Let p be a basic term and let q be an APC-term.

i. If, for some $a \in A$ $p \xrightarrow{a} q$, then there exists a basic term q' with size(q') < size(p) such that APC $\vdash$ $p = a \cdot q' + p$ and APC $\vdash$ $q = q'$;

ii. If $p \xrightarrow{\surd} q$, then there exists a bottom term q' with size(q') < size(p) such that APC $\vdash$ $p = \surd q' + p$ and APC $\vdash$ $q = q'$.

PROOF: Straightforward induction on the structure of p.

4.12 THEOREM. *(Completeness Theorem)*
The axiom system APC is a complete axiomatisation of $\mathbb{T}(APC)/\leftrightarrow$.

PROOF: Let $p,q \in \mathbb{T}$ with $p \leftrightarrow q$. We have to prove that APC $\vdash$ $p = q$. Since $\mathbb{T}(APC)/\leftrightarrow$ is a model for APC, the elimination theorem 4.9 tells us that we only have to prove this for basic terms p,q. A simple argument gives that it is even enough to show that for basic terms p,q

$$p+q \leftrightarrow q \quad \Rightarrow \quad APC \vdash p+q = q.$$

Assume $p+q \leftrightarrow q$. We prove APC $\vdash$ $p+q = q$ with induction on size(p) + size(q).

- $p \equiv \delta$: use A1 and A7.
- $p \equiv a \cdot p'$: we have $p \xrightarrow{a} \varepsilon \cdot p'$. Since $p+q \leftrightarrow q$, there is a q' such that $q \xrightarrow{a} q'$ and $\varepsilon \cdot p' \leftrightarrow q'$. By lemma 4.11 there is a basic term q'' with $size(q'') < size(q)$, $q = a \cdot q'' + q$ and $q'' = q'$. Since $p' \leftrightarrow q''$, $p'+q'' \leftrightarrow q''$ and $q''+p' \leftrightarrow p'$. Thus, by induction, $p'+q'' = q''$ and $q''+p' = p'$, and hence $p' = q''$. But now we derive that $p+q = a \cdot p' + q = a \cdot q'' + q = q$.
- $p \equiv \surd p'$: this case is similar to the previous case.
- $p \equiv p'+p''$: since $p+q \leftrightarrow q$, we also have $p'+q \leftrightarrow q$ and $p''+q \leftrightarrow q$. By induction $p'+q = q$ and $p''+q = q$. Hence $p+q = p'+p''+q = p'+q+p''+q = q+q = q$.

4.13 STANDARD CONCURRENCY.
As a consequence of the completeness theorem, all equations that hold in the model $\mathbb{T}(APC)/\leftrightarrow$ can be proven to hold in APC for all closed terms. We list a few of these equations in table 12. A name often given to such sets of equations is *Standard Concurrency*.

$(x \Vdash y) \Vdash z = x \Vdash (y \Vdash z)$
$x \Vdash y \cdot \delta = y \Vdash x \cdot \delta$
$x \Vdash \delta = x \cdot \delta$

TABLE 12. Standard concurrency.

As consequences of these axioms, we mention the identities $(x \Vdash y) \Vdash z = (y \Vdash x) \Vdash z$ and $x \Vdash y = x \cdot \delta \Vdash y$.

Using these axioms, we can prove a variant of the well-known *Expansion Theorem*, that is very useful to break down the parallel composition of many processes. Since our parallel operator is not in the visible signature, we will not bother to state it here.

5. EXAMPLES.

In order to give some interesting examples of process definitions in APC, we will say a few words about recursive definitions (more can be found in [BK2,3, vG]).

5.1 DEFINITIONS. A **recursive specification** over APC is a (countable) set of equations $\{X = t_X \mid X \in V\}$, where V is a set of variables, and t_X is a term over APC, possibly using variables from V, but no other variables. There is exactly one equation $X = t_X$ for each variable X.

A **solution** of the recursive specification E in a certain domain is an interpretation of the variables of V as processes such that all equations of E are satisfied.

The **Recursive Definition Principle (RDP)** says that every recursive specification has a solution. In the action relation model of APC, RDP holds, if we add for each recursive specification $E = \{X = t_X \mid X \in V\}$ and for each $X \in V$ a constant $\langle X \mid E \rangle$ to the language, together with an action rule

$$\frac{\langle t_X \mid E \rangle \xrightarrow{u} y}{\langle X \mid E \rangle \xrightarrow{u} y}.$$

Here $\langle t_X \mid E \rangle$ denotes the term obtained from t_X by replacing each variable $Y \in V$ by $\langle Y \mid E \rangle$. These rules still fit the *tyft* format, and so bisimulation remains a congru-

ence. Moreover, one can see that all axioms of APC remain valid in the extended setting.

Recursive specifications are used to define (specify) processes. Note that not every recursive specification has a *unique* solution, for $\{X = X\}$ has every process as a solution. In order to get a class of processes with unique solutions, we formulate the condition of guardedness.

5.2 DEFINITIONS.
i. Let t be an APC-term, and X a variable in t. We call an occurrence of X in t **guarded** if X is preceded by an atomic action, i.e. t has a subterm of the form $a \cdot s$, with $a \in A$, and the X in question occurs in s. Otherwise, we call the occurrence of X **unguarded.**
ii. A recursive specification $\{X = t_X \mid X \in V\}$ is **guarded** if each occurrence of a variable in each t_X is guarded.
iii. The **Recursive Specification Principle (RSP)** is the assumption that every guarded recursive specification has at most one solution. We can prove that the extended model of APC satisfies RSP.

In the remainder of this section, we give a number of examples of recursive specifications in APC.

5.3 EXAMPLE 1: SYSTOLIC SORTING.
Systolic systems are characterised by a regular configuration of simple components or cells. Systolic systems have turned out to be useful in VLSI design (see KUNG [K]).

We describe a sorting machine, that can is always ready to input numbers (less than some upper bound N), and is always ready to output the smallest number it contains. This machine consists of a number of cells that each can contain two numbers, and will dynamically create more cells as they become needed. Our description is based on the description in KOSSEN & WEIJLAND [KW], where also a correctness proof can be found (in the setting of ACP_τ). Consider the configuration in fig. 4.

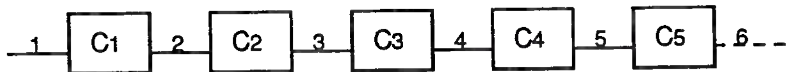

FIGURE 4.

The squares in fig. 4 represent the cells, the lines interconnecting them communication ports. We use the following atomic actions:

- $s_i(d)$ send number d along port i
- $r_i(d)$ read number d along port i
- $c_i(d)$ communicate number d along port i.

The communication function on these atomic actions is defined by: $\gamma(r_i(d), s_i(d)) = c_i(d)$, and γ is undefined on all other pairs.

Cell number i has three types of states, depending on whether it contains 0,1 or 2 numbers. The recursive specification of cell i is given in table 13.

Then, the sorting machine is given by:

$$SORT = \partial_H(C_1^0),$$

where $H = \{r_i(d), s_i(d) \mid i>1, d \leq N\}$. Note that SORT has a guarded recursive specification.

$$C_i^0 = \sum_{d \le N} r_i(d) \cdot \mathbf{new}(C_{i+1}^0) \cdot C_i^1(d) + r_i(stop) + s_i(empty) \cdot C_i^0$$

$$C_i^1(d) = \sum_{e \le N} r_i(e) \cdot C_i^2(min(d,e), max(d,e)) + s_i(d) \cdot s_{i+1}(stop) \cdot C_i^0 \qquad (d \le N)$$

$$C_i^2(d,e) = \sum_{f \le N} r_i(f) \cdot s_{i+1}(e) \cdot C_i^2(min(d,f), max(d,f)) +$$

$$+ s_i(d) \cdot [\sum_{f \le N} r_{i+1}(f) \cdot C_i^2(min(e,f), max(e,f)) + r_{i+1}(empty) \cdot C_i^1(e)] \qquad (d \le e \le N)$$

TABLE 13. Systolic sorting.

5.4 EXAMPLE 2: QUEUE.
The specification of the unbounded (FIFO) queue is one of the recurring issues in process algebra. Examples of recursive specifications can be found in BAETEN & BERGSTRA [BB], VAN GLABBEEK & VAANDRAGER [vGV]. We will give two recursive specifications in APC involving the **new** construct: the first has an infinite number of equations, the second a finite number. To start with, we give the standard infinite specification of the queue in table 14. We denote the queue with contents σ (σ is a sequence of data elements, $\sigma \in D^*$ for some data set D) by Q_σ. λ is the empty sequence, d (for $d \in D$) also stands for a one element sequence, and $\sigma\rho$ denotes the concatenation of sequences σ and ρ.

$$Q_\lambda = \sum_{d \in D} in(d) \cdot Q_d$$

$$Q_{\sigma d} = \sum_{e \in D} in(e) \cdot Q_{e\sigma d} + out(d) \cdot Q_\sigma \qquad (\sigma \in D^*, d \in D)$$

TABLE 14. Queue, standard specification.

5.5 The second specification in APC will use an unlimited number of cells as in 5.3. This specification is inspired by a similar specification in DE SIMONE [DS]. Each cell can contain one data element; this element can be output when the permission for doing so is received: the permission go(i) will communicate with the *potential* output action pout(d,i) with as result the output out(d). Thus, we have a communication function γ given by:

$$\gamma(go(i), pout(d,i)) = out(d),$$

and γ is undefined otherwise. The definition of the cells and the queue is given in table 15. Note that this is a guarded specification. The encapsulation set is $H = \{go(i), pout(d,i) \mid d \in D, i \ge 1\}$ (D is the set of data elements).

$$C_i = \sum_{d \in D} in(d) \cdot \mathbf{new}(C_{i+1}) \cdot pout(d,i) \cdot go(i+1) \qquad (i \ge 1)$$

$$Q^1 = \partial_H(\mathbf{new}(C_1) \cdot go(1) \cdot \delta)$$

TABLE 15. Queue, first APC specification.

5.6 THEOREM. $Q^1 = Q_\lambda$.

PROOF: Define, for each $n \ge 1$ and each $\sigma \in D^*$, with $\sigma \equiv d_1 \ldots d_k$, the process R_σ^n by:

$$R_\sigma^n = \partial_H(C_{n+k} \Vert\!\!\lceil pout(d_1, n+k-1) \cdot go(n+k) \Vert\!\!\lceil \ldots \Vert\!\!\lceil pout(d_k, n) \cdot go(n+1) \Vert\!\!\lceil go(n) \cdot \delta).$$

(Note that we assume Standard Concurrency of 4.13 in this proof.) We will prove that, for each $n\geq 1$, $R^n_\sigma = Q_\sigma$. We do this by showing that the R^n_σ satisfy the specification in table 14. As a consequence, we derive

$$Q^1 = \partial_H(\mathrm{new}(C_1)\cdot go(1)\cdot\delta) = \partial_H(C_1 \| go(1)\cdot\delta) = R^1_\lambda = Q_\lambda.$$

Now the verification:

$$R^n_\lambda = \partial_H(C_n \| go(n)\cdot\delta) =$$
$$= \sum_{d\in D} in(d)\cdot\partial_H(\mathrm{new}(C_{n+1})\cdot pout(d,n)\cdot go(n+1) \| go(n)\cdot\delta) =$$
$$= \sum_{d\in D} in(d)\cdot\partial_H(C_{n+1} \| pout(d,n)\cdot go(n+1) \| go(n)\cdot\delta) =$$
$$= \sum_{d\in D} in(d)\cdot R^n_d.$$

Next, if $\sigma \equiv d_1 \ldots d_{k-1}$,

$$R^n_{\sigma d} = \partial_H(C_{n+k} \| pout(d_1,n+k-1)\cdot go(n+k) \| \ldots$$
$$\ldots \| pout(d_{k-1},n+1)\cdot go(n+2) \| pout(d,n)\cdot go(n+1) \| go(n)\cdot\delta) =$$
$$= \sum_{e\in D} in(e)\cdot\partial_H(\mathrm{new}(C_{n+k+1})\cdot pout(e,n+k)\cdot go(n+k+1) \| \ldots$$
$$\ldots \| pout(d,n)\cdot go(n+1) \| go(n)\cdot\delta) +$$
$$+ out(d)\cdot\partial_H(C_{n+k} \| pout(d_1,n+k-1)\cdot go(n+k) \| \ldots \| go(n+1) \| \delta) =$$
$$= \sum_{e\in D} in(e)\cdot\partial_H(C_{n+k+1} \| pout(e,n+k)\cdot go(n+k+1) \| \ldots$$
$$\ldots \| pout(d,n)\cdot go(n+1) \| go(n)\cdot\delta) +$$
$$+ out(d)\cdot\partial_H(C_{n+k} \| pout(d_1,n+k-1)\cdot go(n+k) \| \ldots \| go(n+1)\cdot\delta) =$$
$$= \sum_{e\in D} in(e)\cdot R^n_{e\sigma d} + out(d)\cdot R^{n+1}_\sigma.$$

Using RSP (see 5.2), we can show that the R^n_σ satisfy the specification in table 14.

5.7 THIRD SPECIFICATION OF QUEUE.

Next, we will give a finite recursive specification for the queue. In this specification, we will use action renaming. For each function $f: A \to A$, we introduce a unary operator ρ_f, that will rename atoms a into $f(a)$, and do nothing else. This operator is axiomatised in table 16. Action rules are quite easy to formulate.

$\rho_f(\delta) = \delta$
$\rho_f(ax) = f(a)\cdot\rho_f(x)$
$\rho_f(\surd x) = \surd(\rho_f(x))$
$\rho_f(x + y) = \rho_f(x) + \rho_f(y)$

TABLE 16. Action renaming.

From BAETEN & BERGSTRA [BB] we know that the queue can be finitely specified in ACP plus renaming. In table 17, we give a finite specification in APC plus renaming.

$\mathrm{Cell} = \sum_{d \in D} \mathrm{in}(d) \cdot \partial_H(\mathbf{new}(\rho_f(\mathrm{Cell})) \cdot \mathrm{pre}(d) \cdot \mathrm{go})$
$Q^2 = \partial_H(\mathbf{new}(\rho_f(\mathrm{Cell}) \cdot \mathrm{go} \cdot \delta)$

TABLE 17. Queue, second specification.

Here, we use the renaming function f that renames each pre(d) into near(d), and leaves all other atoms unchanged. The communication function is specified by $\gamma(\mathrm{go},\mathrm{near}(d)) = \mathrm{out}(d)$ (undefined otherwise). The encapsulation set is $H = \{\mathrm{go}\} \cup \{\mathrm{near}(d) \mid d \in D\}$.

In order to see that the specification in table 17 indeed describes a FIFO-queue, it might be illustrative to consider fig. 5.

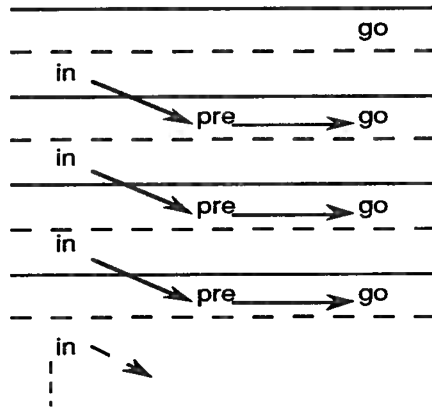

FIGURE 5.

In this diagram, we have abstracted from data d in actions in(d), out(d), etc. With arrows the 'causal' links between events are denoted. A black line stands for an encapsulation operator ∂_H and a dashed line for a renaming operator ρ_f.

One may imagine that in an execution, events 'bubble' upwards until they have passed through the surface of the topmost encapsulation line. An events cannot move before all its causal predecessors have occurred. A pre-event can pass through both types of lines. However, when it passes through a dashed line, it is renamed into a near-event. near-events and go-events are blocked by a black line. The synchronisation of a near-event and a go-event, however, gives a out-event. out-events, like in-events, can pass through both types of lines.

Along these same lines, we can give a recursive specification for the stack in APC.

5.8 THEOREM. $Q^2 = Q_\lambda$.

PROOF (sketch): Similar to 5.6. We define processes S_σ, that satisfy the specification in table 14. The S_σ are defined by using auxiliary processes T_σ, that, in turn, are defined inductively:

$$T_\lambda = \mathrm{Cell}$$
$$T_{d\sigma} = \partial_H(\rho_f(T_\sigma \| \mathrm{pre}(d) \cdot \mathrm{go}))$$
$$S_\sigma = \partial_H(\rho_f(T_\sigma \| \mathrm{go} \cdot \delta)).$$

The proof that the S_σ satisfy the specification in table 14 makes use of *alphabet information*. For more information on this type of argument, see BAETEN, BERGSTRA & KLOP [BBK].

5.9 EXAMPLE 3: BAG.
Along the same lines as for queue, we can give a simple recursive specification for the bag (an *unordered* channel; a state of the bag can be considered as a multiset of objects). We give the recursive specification in table 18, without further comment.

$$\text{Bag} = \sum_{d\in D} \text{in}(d)\cdot\textbf{new}(\text{out}(d))\cdot\text{Bag}$$

TABLE 18. Bag.

ACKNOWLEDGEMENT. The idea for an event structure semantics for the **new** operator arose following an inspiring discussion with Henk Goeman.

REFERENCES.

[A] P. AMERICA, *Definition of the programming language Pool-T*, ESPRIT project 415, nr. 0091, Philips Research Laboratories, Eindhoven 1985.

[AB] P. AMERICA & J.W. DE BAKKER, *Designing equivalent semantic models for process creation*, TCS 60, 1988, pp. 109-176.

[BB] J.C.M. BAETEN & J.A. BERGSTRA, *Global renaming operators in concrete process algebra*, I&C 78, 1988, pp. 205-245.

[BBK] J.C.M. BAETEN, J.A. BERGSTRA & J.W. KLOP, *Conditional axioms and α/β-calculus in process algebra*, in: Proc. IFIP Conf. Formal Descr. of Progr. Concepts III (M. Wirsing, ed.), North-Holland, Amsterdam 1987, pp. 53-75.

[BG] J.C.M. BAETEN & R.J. VAN GLABBEEK, *Merge and termination in process algebra*, in: Proc. 7th FST&TCS, Pune (K.V. Nori, ed.), Springer LNCS 287, 1987, pp. 153-172.

[B] J.A. BERGSTRA, *A process creation mechanism in process algebra*, in: Applications of Process Algebra (J.C.M. Baeten, ed.), CWI Monograph 8, North-Holland, Amsterdam 1989, pp. 81-88.

[BHK] J.A. BERGSTRA, J. HEERING & P. KLINT, *Module algebra*, report CS-R8844, Centre for Math. & Comp. Sci., Amsterdam 1988. To appear in JACM.

[BK1] J.A. BERGSTRA & J.W. KLOP, *Fixed point semantics in process algebras*, report IW 206, Math. Centre, Amsterdam 1982.

[BK2] J.A. BERGSTRA & J.W. KLOP, *Process algebra for synchronous communication*, I&C 60, 1984, pp. 109-137.

[BK3] J.A. BERGSTRA & J.W. KLOP, *Process algebra: specification and verification in bisimulation semantics*, in: Math. & Comp. Sci. II (eds. M. Hazewinkel, J.K. Lenstra & L.G.L.T. Meertens), CWI Monograph 4, North-Holland, Amsterdam 1986, pp. 61-94.

[BIM] B. BLOOM, S. ISTRAIL & A.R. MEYER, *Bisimulation can't be traced: preliminary report*, in: Proc. 15th POPL, San Diego Ca. 1988, pp. 229-239.

[BC] G. BOUDOL & I. CASTELLANI, *Permutation of transitions: an event structure semantics for CCS and SCCS*, report 798, INRIA, Sophia Antipolis 1988. To appear in Proc. REX School (J.W. de Bakker, W.-P. de Roever & G. Rozenberg, eds.), Springer LNCS.

[BR] E. BRINKSMA, *On the design of extended LOTOS*, Ph.D. thesis, University of Twente, 1988.

[CDP] L. CASTELLANO, G. DE MICHELIS & L. POMELLO, *Concurrency vs. interleaving: an instructive example*, Bulletin of the EATCS 31, 1987, pp. 12-15.

[DDM] P. DEGANO, R. DE NICOLA & U. MONTANARI, *A distributed operational semantics for CCS based on condition/event systems*, Acta Informatica 26 (1/2), 1988, pp. 59-91.

[vG] R.J. VAN GLABBEEK, *Bounded nondeterminism and the approximation induction principle in process algebra*, in: Proc. STACS 87 (F.J. Brandenburg, G. Vidal-Naquet & M. Wirsing, eds.), Springer LNCS 247, 1987, pp. 336-347.

[GG] R.J. VAN GLABBEEK & U. GOLTZ, *Equivalence notions for concurrent systems and refinement of actions*, report 366, GMD, Sankt Augustin 1989.

[vGV] R.J. VAN GLABBEEK & F.W. VAANDRAGER, *Modular specifications in process algebra (with curious queues)*, report CS-R8821, Centre for Math. & Comp. Sci., Amsterdam 1988.

[GV] J.F. GROOTE & F.W. VAANDRAGER, *Structured operational semantics and bisimulation as a congruence*, report CS-R8845, Centre for Math. & Comp. Sci., Amsterdam 1988.

[H] C.A.R. HOARE, *Communicating sequential processes*, Prentice-Hall 1985.

[KW] L. KOSSEN & W.P. WEIJLAND, *Correctness proofs for systolic algorithms: palindromes and sorting*, in: Applications of Process Algebra (J.C.M. Baeten, ed.), CWI Monograph 8, North-Holland, Amsterdam 1989, pp. 89-125.

[KV] C.J.P. KOYMANS & J.L.M. VRANCKEN, *Extending process algebra with the empty process ε*, report LGPS 1, Faculty of Philosophy, State University of Utrecht 1985.

[K] K.T. KUNG, *Let's design algorithms for VLSI systems*, in: Proc. Conf. on VLSI: architecture, design, fabrication, California Institute of Technology, 1979.

[MDS] E. MADELAINE & R. DE SIMONE, *ECRINS un laboratoire de preuve pour les calculs de processus*, report R.R. 672, INRIA, Sophia Antipolis 1987.

[M] R. MILNER, *A calculus for communicating systems*, Springer LNCS 92, 1980.

[O] E.-R. OLDEROG, *Operational Petri net semantics for CCSP*, in: Advances in Petri Nets (G. Rozenberg, ed.), Springer LNCS 266, 1987, pp. 196-223.

[PA] D.M.R. PARK, *Concurrency and automata on infinite sequences*, in: Proc. 5th GI (P. Deussen, ed.), Springer LNCS 104, 1981, pp. 167-183.

[PL] G.D. PLOTKIN, *A structural approach to operational semantics*, report DAIMI FN-19, Comp. Sci. Dept., Aarhus University 1981.

[DS] R. DE SIMONE, *Higher-level synchronising devices in MEIJE-SCCS*, TCS 37, 1985, pp. 245-267.

[SS] S.A. SMOLKA & R.E. STROM, *A CCS semantics for NIL*, in: Formal Descr. of Progr. Concepts III (M. Wirsing, ed.), North-Holland, Amsterdam 1987, pp. 347-367.

[VA] F.W. VAANDRAGER, *Process algebra semantics of POOL*, in: Applications of process algebra (J.C.M. Baeten, ed.), CWI Monograph 8, North-Holland, Amsterdam 1989, pp. 173-236.

[W] G. WINSKEL, *Event structures*, in: Petri Nets: Applications and Relationships to Other Models of Concurrency, Bad Honnef 1986 (W. Brauer, W. Reisig & G. Rozenberg, eds.), Springer LNCS 255, 1987, pp. 325-392.