Some finite-graph models for Process Algebra

Paul Spruit & Roel Wieringa

Department of Mathematics and Computer Science
Vrije Universiteit
De Boelelaan 1081a
1081 HV, Amsterdam
The Netherlands
Email: pasprui@cs.vu.nl, roelw@cs.vu.nl

1 Introduction

In this paper, we present a number of closely related models of process algebra [2, 3, 4], called *finite-graph models*. In a finite-graph model of process algebra, each process is a bisimulation class of a particular kind of process graphs, called *recursive process graphs*. Just as in the standard graph model [1], each guarded recursive specification has exactly one solution in a finite-graph model, but in contrast to the standard graph model, this solution can be shown to contain a finite recursive process graph as element.

The finite-graph models were defined in order to be able to build an editor that can manipulate process graphs. It is well-known that there are finite guarded specifications that have no finite solution in the standard graph model; the specification of a stack is an example [1, page 63]. Figure 1 shows an approximation of a graph of a (terminating) stack process and figure 2 shows a recursive process graph that, in a finite-graph model, bisimulates with it. The intuitive reading of figure 2 is that after a $push_i$ event, there is a choice between a pop_i or doing process S itself. Details are given below. Figure 2 is a finite graph that can be drawn on a finite screen. The finite-graph models presented in this paper can represent more processes in a finite manner than the standard graph model. Note that recursive specifications can also be finitely represented in a graphical way, by means of parse trees of the terms. However, such parse trees have a structure that is quite different from the structure of recursive process graphs, and are more difficult to understand.

In section 2, we define the set $\Re$ of recursive process graphs, a bisimulation relation $\leftrightarrows$ on $\Re$, and prove that $\Re/\leftrightarrows$ is a model of BPA. It is shown that a subalgebra of $\Re/\leftrightarrows$ is isomorphic with the standard graph model of finitely-branching processes. Section 3 extends this to process algebra with the parallel composition operator. We also show the relation with some forms of true concurrency. Section 4 concludes the paper. All proofs are omitted from the paper, but are given in [9, 10].

2 Two finite-graph models for BPA

We fix a set Act of *atomic actions* and a countably infinite set $PVAR$ of *process variables*, disjoint from Act. Metavariables ranging over Act are $a, b, c, \ldots$ and metavariables ranging over $PVAR$ are $X, Y, Z, X_1, X_2, \ldots$. The two operators of Basic Process Algebra (BPA) are + (choice)

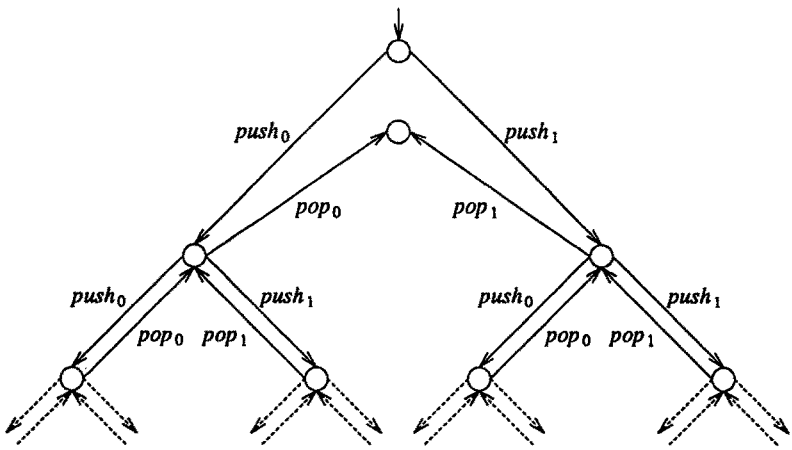


Figure 1: Part of a graph of a terminating stack in the standard model.

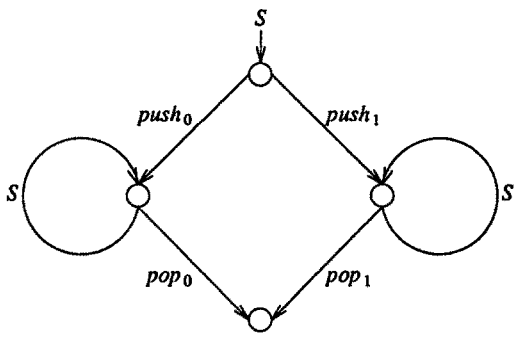


Figure 2: Recursive process graph of a terminating stack in a finite-graph model.

$X+Y=Y+X$	A1
$(X+Y)+Z=X+(Y+Z)$	A2
$X+X=X$	A3
$(X+Y)Z=XZ+YZ$	A4
$(XY)Z=X(YZ)$	A5

Table 1: The BPA axiom system.

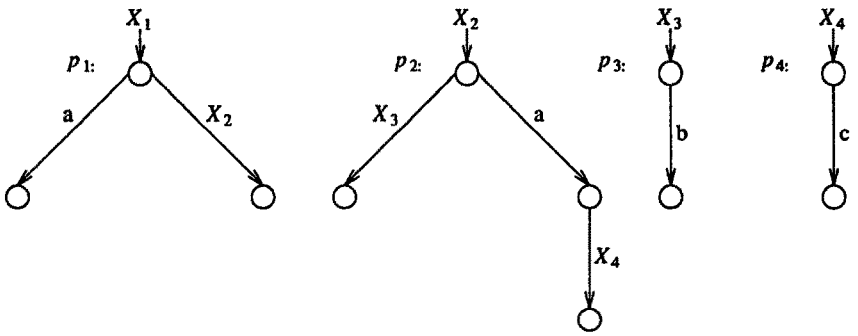


Figure 3: Some example recursive process components

and · (sequence). The · for sequence is often omitted from BPA terms. The axiom system of BPA is given in table 1 [1].

2.1 Recursive process graphs

Definition 1 *A* recursive process component *is a directed, finitely branching, connected, rooted graph that has at least one edge. The edges of a recursive process component are labeled by elements of the set $PVAR \cup Act$. Each recursive process component has a name, which must be a process variable.*

The set of all recursive process components is called RPC. Metavariables ranging over RPC are $p, q, r, p_1, p_2, \ldots$ The set of all process variables occurring as a label or a name in a recursive process component p is called $pvar(p)$. The name of a recursive process component p is called $name(p)$. The root node of a recursive process component p is called $root(p)$.

Figure 3 shows some example recursive process components. Each standard process graph is a recursive process component (with an arbitrary name).

In what follows, we write $\mathcal{P}_f(A)$ for the set of all finite subsets of a set A, and $\mathcal{P}_{nf}(A)$ for all non-empty finite subsets of A.

Definition 2 *A* recursive process graph *is a pair (P, X) with $P \in \mathcal{P}_{nf}(RPC)$ and $X \in PVAR$ such that the following three conditions hold*

root existence $\exists p \in P : name(p) = X$

unique naming $\forall p, q \in P : p \neq q \Rightarrow name(p) \neq name(q)$

label existence $\forall p \in P : \forall Y \in pvar(p) : \exists q \in P : Y = name(q)$

The set of all recursive process graphs is called $\Re$. Metavariables ranging over $\Re$ are $g, h, g_1, g_2, \ldots$

For example, in figure 3 $(\{p_1, p_2, p_3, p_4\}, X_1)$, $(\{p_1, p_2, p_3, p_4\}, X_2)$ and $(\{p_4\}, X_4)$ are recursive process graphs. Each standard process graph is a recursive process graph (consisting of one component). We will assume the function $pvar$ is extended from recursive process components to recursive process graphs in the following way: if $g = (P, X)$ then $pvar(g) = \cup_{p \in P} pvar(p)$. Because of the root existence constraint we know for $g = (P, X)$ that $X \in pvar(g)$.

Definition 3 *For every recursive process graph* $g = (P, X)$ *the* recursive process component *function*

$$rpc_g : pvar(g) \rightarrow P$$

assigns to a process variable the recursive process component of which it is the name.

Using definition 2 we can see that for every recursive process graph g, the function rpc_g is correctly defined: every process variable in $pvar(g)$ is the name of a recursive process component because of the label existence constraint and this component is unique because of the unique naming constraint. Example: if in figure 3 we take $g = (\{p_1, p_2, p_3, p_4\}, X_1)$, then $rpc_g(X_i) = p_i$ (for $1 \leq i \leq 4$).

We extend the function $root$ from recursive process components to recursive process graphs by defining for a recursive process graph $g = (P, X)$ that $root(g) = root(rpc_g(X))$. Some more terminology: for a recursive process graph $g = (P, X)$, we call X the root variable of g and $rpc_g(X)$ the root component of g.

2.2 Recursive bisimulation

We want to give a semantics of BPA in terms of some form of observational equivalence between graphs of $\Re$, i.e. we want to define a bisimulation notion on elements of $\Re$ [7, 8]. A definition of bisimulation in $\Re$ is complicated by the fact that a node in a recursive process graph does not represent all state information of the process. Part of the state of a process consists of the sequence of "jumps" to process components that have been performed, so part of the remaining behavior of the process consists of a sequence of "pops" to performed. A bisimulation will not be a relation between the nodes of two recursive process graphs, but between pairs of the form (node, stack-of-nodes), where stack-of-nodes is a stack of "return addresses." In what follows we use strings to represent stacks, with ϵ for the empty stack and an invisible concatenation operation. The top of the stack is the leftmost element of the string. We use $k\ l, \ldots$ to denote a node of a recursive process graph, and $s, t, \ldots$ to denote a finite stack of nodes of a recursive process graph. Definitions 5 and 7 are illustrated in figure 4.

Definition 4 *A* state *of a recursive process graph* g *is a pair* (k, s) *with* k *a node of* g *and* s *a finite stack of nodes of* g.

Definition 5 *The relation* $\rightarrow_g^{push}$ *is defined for every recursive process graph* g *as a binary relation on the states of* g *in the following way:*

> $(k, s) \rightarrow_g^{push} (k', ls)$ *iff there is an edge in* g *starting at node* k*, ending at node* l *and labeled by some process variable* X *such that* $k' = root(rpc_g(X))$.

The relation $\rightarrow_g^{push*}$ *is defined for every recursive process graph* g *as the reflexive transitive closure of* $\rightarrow_g^{push}$.

Definition 6 *Node* k *of a recursive process graph* g *is called an* end node *iff there does not exists an edge in* g *starting at node* k.

Definition 7 *The relation* $\rightarrow_g^{pop}$ *is defined for every recursive process graph* g *as a binary relation on the states of* g *in the following way:*

> $(l, ks) \rightarrow_g^{pop} (k, s)$ *iff node* l *is an end node in* g.

The relation $\rightarrow_g^{pop*}$ *is defined for every recursive process graph* g *as the reflexive transitive closure of* $\rightarrow_g^{pop}$.

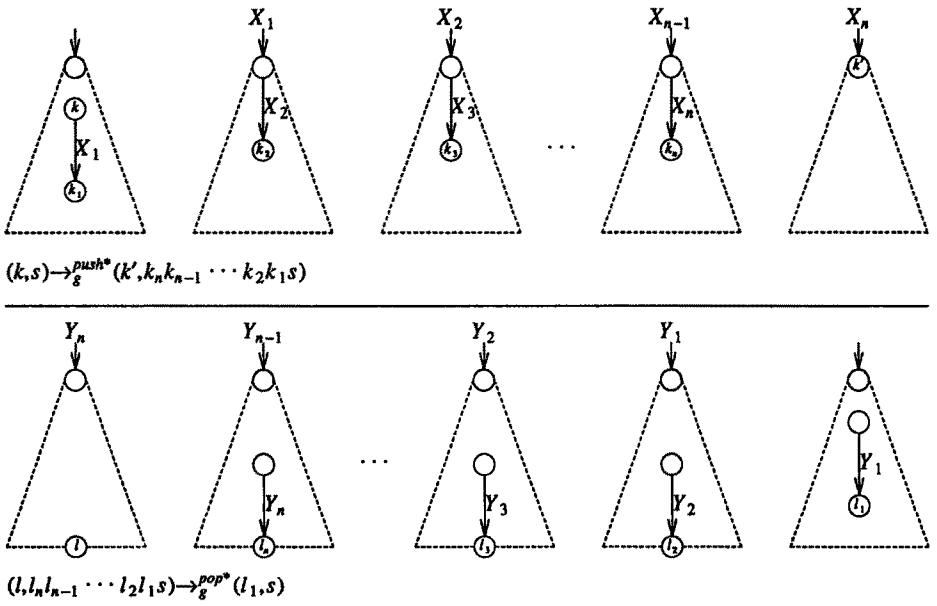


Figure 4: Illustration of the push and pop relations.

Definition 8 is illustrated in figure 5.

Definition 8 *The relation $\rightarrow_g^a$ is defined for every recursive process graph g and atomic action a as a binary relation on the states of g as follows: $(k,s) \rightarrow_g^a (k',s')$ iff there exist states (l,t) and (l',s') such that the following three conditions hold:*

1. $(k,s) \rightarrow_g^{pop*} (l,t)$
2. $(l,t) \rightarrow_g^{push*} (l',s')$
3. *There is an edge in g starting at l', ending at k' and labeled a.*

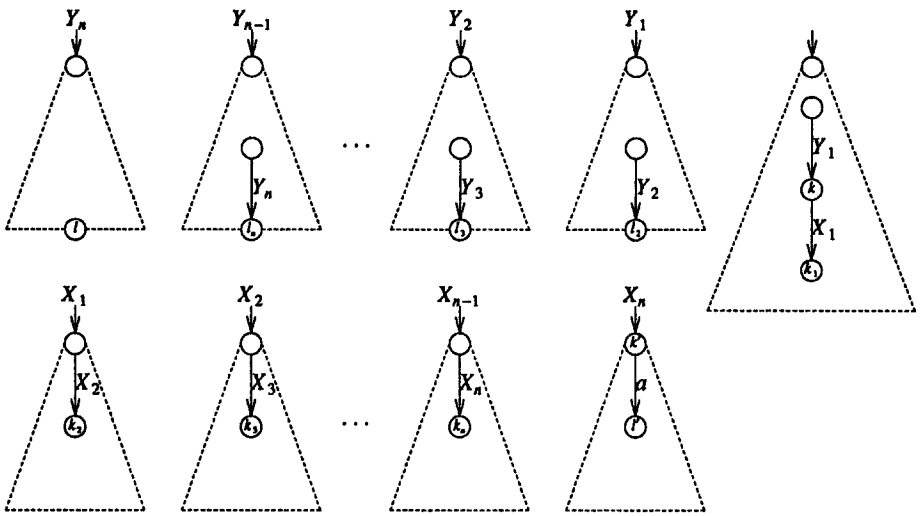
In figure 5, we combine the $\rightarrow_g^{pop*}$ and $\rightarrow_g^{push*}$ parts of figure 4 by taking $l_1 = k$. The relation $\rightarrow_g^a$ captures the idea of a "state change" in a recursive process graph. Note that the definition of $\rightarrow_g^a$ does not allow a "mix" of pops and pushes, but requires first a number of pops and then a number of pushes. This is not really a constraint, because after a push, we arrive in a root node of a recursive process component, and from this node no pop would be possible anyway (it can't be an end node because by definition, a recursive process component contains at least one edge).

Definition 9 *A state (k,s) of g is called a* reachable state *of g iff there is an $n \geq 1$, a sequence of states $(l_1,t_1), (l_2,t_2), \ldots, (l_n,t_n)$ and a sequence of atomic actions $a_1, a_2, \ldots, a_{n-1}$ (not necessarily different) such that:*

$$(root(g), \epsilon) = (l_1,t_1) \rightarrow_g^{a_1} (l_2,t_2) \rightarrow_g^{a_2} \cdots \rightarrow_g^{a_{n-1}} (l_n,t_n) = (k,s)$$

Definition 10 *The relation $\Rightarrow_g^a$ is defined for every recursive process graph g and atomic action a as a binary relation on states of g as follows: $(k,s) \Rightarrow_g^a (k',s')$ iff the following two conditions hold:*

1. $(k,s) \rightarrow_g^a (k',s')$



$(l, l_n l_{n-1} \cdots l_2 ks) \rightarrow_g^a (l', k_n k_{n-1} \cdots k_2 k_1 s)$

Figure 5: Illustration of the $\rightarrow_g^a$ relation.

2. *(k, s) is a reachable state of g.*

A trivial fact is that if (k, s) is a reachable state and $(k, s) \Rightarrow_g^a (k', s')$ then (k', s') is also a reachable state.

Definition 11 *A state (k, s) of recursive process graph g is called an* end state *iff k is an end node in g and every node in s is an end node in g.*

Using the $\Rightarrow_g^a$ relation and the notion of end state, the notion of bisimulation can be defined. In the following definition the notation xRy is used for $(x, y) \in R$.

Definition 12 *Let g and h be recursive process graphs. A relation R between reachable states of g and reachable states of h is called a* bisimulation relation *(notation $R : g \leftrightarrows h$) iff the following five conditions hold*

1. $(root(g), \epsilon) R (root(h), \epsilon)$
2. *if $(k, s) R (l, t)$ and $(k, s) \rightarrow_g^a (k', s')$ then there must exist a state (l', t') (in h) such that $(k', s') R (l', t')$ and $(l, t) \rightarrow_h^a (l', t')$*
3. *if $(k, s) R (l, t)$ and $(l, t) \rightarrow_h^a (l', t')$ then there must exist a state (k', s') (in g) such that $(k', s') R (l', t')$ and $(k, s) \rightarrow_g^a (k', s')$*
4. *if $(k, s) R (l, t)$ and (k, s) is an end state in g, then (l, t) is an end state in h*
5. *if $(k, s) R (l, t)$ and (l, t) is an end state in h, then (k, s) is an end state in g*

Note that as R is a relation between reachable nodes of g and reachable nodes of h, we could change all $\rightarrow_g^a$ by $\Rightarrow_g^a$ and $\rightarrow_h^a$ by $\Rightarrow_h^a$ in the above definition, without really changing the meaning of the definition.

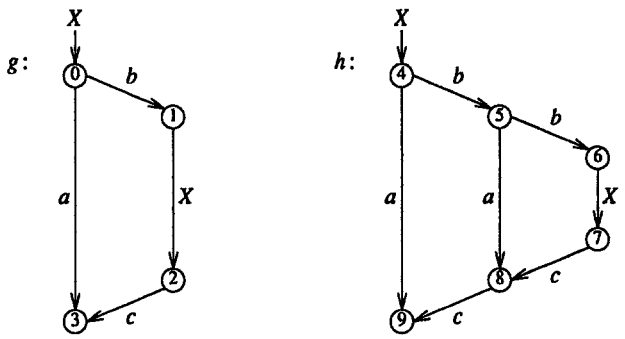


Figure 6: Two bisimilar recursive process graphs.

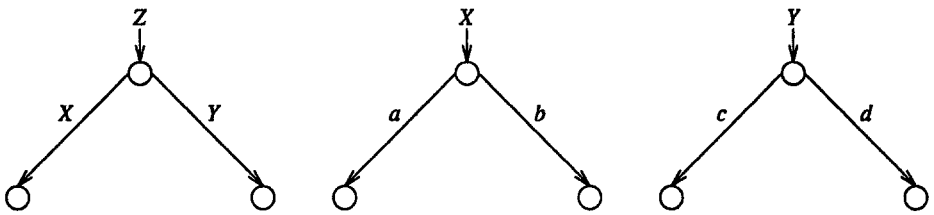


Figure 7: A four-way branch.

Definition 13 *The recursive process graphs g and h are called* bisimilar *(g and h* bisimulate*) if a bisimulation relation R exists such that R : g$\leftrightarrows$ h. (Notation if g and h are bisimilar: g$\leftrightarrows$ h).*

Figure 6 shows two bisimilar recursive process graphs g and h (both g and h consist of one recursive process component). The nodes of the graphs have been numbered to make it possible to refer to them. The relation below is a bisimulation relation between g and h (in the following relation the abbreviation k^j represents a stack consisting of j nodes which are all k):

$$\{((0,\epsilon),(4,\epsilon))\} \cup (\bigcup_{i=0}^{\infty} \{ \; ((1,2^{2i}),(5,7^i)),((1,2^{2i+1}),(6,7^i)), \\ ((3,2^{2i}),(9,7^i)),((3,2^{2i+1}),(8,7^i))\})$$

Discussion. In our definition of bisimulation, we have chosen an $\Rightarrow_g^a$ step as the smallest unit of execution. The "substeps" that together form such a step (see definitions 8 and 10) are not considered execution steps. Because of this choice, the model we get has a number of useful properties, one of which is that it is isomorphic to the standard graph model (see proposition 26). If we had chosen to take as atomic steps all the pop and push substeps that make up one $\Rightarrow_g^a$ step, we would get a totally different model for which this isomorphism does not hold.

To explain the difference further, consider figures 7 and 8. In both figures a recursive process graph is drawn, with Z as the root variable. Figure 7 may suggest that there is first a choice between X and Y, and then depending either a choice between a and b or between c and d. This would be correct if we would consider a jump to be an invisible action like Milner's τ [7, 8]. However, jumps are not even invisible atomic actions, they are not atomic at all, and the branching structure is bisimilar with that in figure 8. **End of discussion.**

Proposition 14 *Bisimulation is an equivalence relation on* $\Re$.

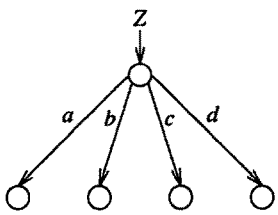


Figure 8: A four-way branch.

2.3 BPA operations on recursive process graphs

To define operations on recursive process graphs, we need to take care of some trivial problems with name clashes by using a renaming function.

Definition 15 *A renaming τ is a bijective function $\tau : PVAR \to PVAR$.*

Any renaming can be extended to a function on recursive process components and to a function recursive process graphs.

Proposition 16 *For every renaming τ and recursive process graph g: $g \leftrightarrows \tau(g)$.*

Definition 17 *A recursive process graph g is a* variant *of recursive process graph h if a renaming τ exists such that $g = \tau(h)$.*

Definition 18 *Given recursive process graphs g and h, the recursive process graph $g[h]$ is a variant of g such that there are no name clashes between variables in $g[h]$ and h (this means $pvar(g[h]) \cap pvar(h) = \emptyset$). We assume that some algorithm exists to determine $g[h]$ uniquely.*

Since there are infinitely many variables, there is always a $g[h]$ for any pair of recursive process graphs g and h.

Definition 19 *The BPA operations and constants are interpreted in $\Re$ in the following way.*

- *A constant (atomic action) a is interpreted as the recursive process graph $(\{p\}, Z)$ with Z the first element of $PVAR$ and p a recursive process component with name Z and one edge starting at the root node with label a.*
- *The operation $+$ is interpreted as: $g + h$ (with $g[h] = (P, X)$ and $h = (Q, Y)$) is the recursive process graph $(P \cup Q \cup \{p\}, Z)$, where Z is the first process variable in $PVAR \setminus (pvar(g[h]) \cup pvar(h))$ and p the recursive process component with name Z and two edges starting at the root node, one labeled X and the other labeled Y.*
- *The operation $\cdot$ is interpreted as: $g \cdot h$ (with $g[h] = (P, X)$ and $h = (Q, Y)$) is the recursive process graph $(P \cup Q \cup \{p\}, Z)$, where Z is the first process variable in $PVAR \setminus (pvar(g[h]) \cup pvar(h))$ and p the recursive process component with name Z and two edges, the first one starting at the root node labeled X and the second edge starting at the endpoint of the first edge and labeled Y.*

It is easy to see that the above three constructions indeed result in recursive process graphs (which must have the root existence, unique naming and label existence properties). In figure 9 definition 19 is illustrated. (In this figure the root components of the three constructions of definition 19 are drawn.)

Proposition 20 *Bisimulation is a congruence relation with respect to the operations of definition 19.*

Proposition 21 *$\Re/\leftrightarrows$ is a model of BPA.*

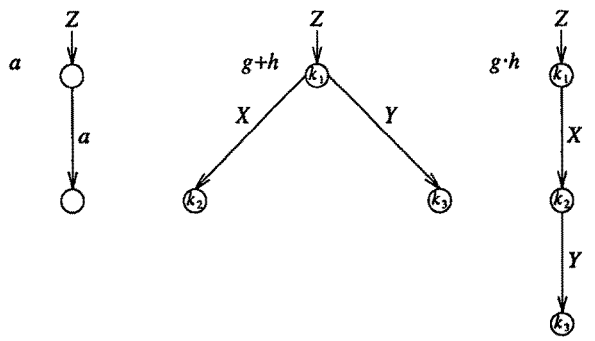


Figure 9: Operations on recursive process graphs.

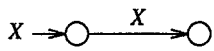


Figure 10: A circular process graph.

2.4 Circularity

Figure 10 shows a special kind of recursive process graph that we call *circular*. In this section, we define these graphs formally and study the models we get by allowing them and by disallowing them.

Definition 22 *The* one-step process variables function

$$pvar1 : RPC \to \mathcal{P}_f(PVAR)$$

assigns to every recursive process component the set of process variables occurring as labels on edges starting at the root of the recursive process component.

We only consider finitely branching graphs, so $pvar1(p)$ is always finite. We now define the set of variables that are reachable, in a recursive process graph, from the root of a recursive process component, without doing an atomic action.

Definition 23 *For every recursive process graph $g = (P, X)$ define the function $dep_g : P \to \mathcal{P}(pvar(g))$ as*

$$\begin{aligned} dep_g(p) \;=\; & \{X \mid \exists n \geq 2 : \exists X_1, X_2, \ldots, X_n \in pvar(g) : X_1 = name(p) \wedge \\ & X = X_n \wedge \forall i \in \{1, \ldots, n-1\} : X_{i+1} \in pvar1(rpc_g(X_i))\} \end{aligned}$$

For example, in figure 3, if we let $g = (\{p_1, p_2, p_3, p_4\}, X_1)$, then $dep_g(p_1) = \{X_2, X_3\}$.

Definition 24 *A recursive process graph $g = (P, X)$ is called* circular *iff*

$$\exists p \in P : name(p) \in dep_g(p)$$

The set of all non-circular recursive process graphs is denoted $\Re^{\emptyset}$.

In figure 3 the recursive process graph $(\{p_1, p_2, p_3, p_4\}, X_1)$ is not circular, but would be circular if the label X_3 in p_2 would be replaced by X_1.

Proposition 25 *$\Re^{\emptyset}/\leftrightarrow$ is a model of BPA*

$X + \delta = X$	A6
$\delta X = \delta$	A7

Table 2: Deadlock axioms.

Proposition 26 *The models $\Re^{\emptyset}/\leftrightarrow$ and $G/\leftrightarrow$ are isomorphic.*

$G/\leftrightarrow$ is the standard graph model as described in [1]. Because every guarded recursive specification has a unique solution in $G/\leftrightarrow$ and the "unique solution" property is preserved under isomorphisms, we can conclude that every guarded recursive specification has a unique solution in $\Re^{\emptyset}/\leftrightarrow$. We can then prove:

Proposition 27 *The solution of every finite guarded recursive specification in $\Re^{\emptyset}$ contains a finite element.*

Turning now to the model $\Re/\leftrightarrow$ that includes circular recursive process graphs, we would like to find a graph model that is isomorphic to it. One suggestion where to look is given by the following proposition.

Proposition 28 *$\Re/\leftrightarrow$ is a BPA_δ model.*

BPA_δ is BPA extended with the constant δ (deadlock) and the axioms shown in table 2.

Proposition 28 can be explained intuitively by considering the process in figure 10. This process can never do an atomic action, and furthermore, the start state $(root(g), \epsilon)$ is not an end state. Because successful termination means reaching an endstate, the process can never terminate. This explains intuitively that this process has the same properties as deadlock.

A second suggestion where to look for a model isomorphic to $\Re/\leftrightarrow$ is found in the proof of proposition 26 (given in [9]), where non-circularity is used essentially to prove that $\Re/\leftrightarrow$ is isomorphic to the model $G/\leftrightarrow$ of finitely-branching processes. A reasonable place to look for a model of BPA_δ that is isomorphic to $\Re/\leftrightarrow$, would therefore be $G_\delta^\infty/\leftrightarrow$. Figure 11 illustrates this suggestion, by giving an examples of an infinite branching graph (an element of G_δ^∞) and a finitely branching (circular) recursive process graph (an element of $\Re$) that represent the same process (i.e. are bisimilar).

However, a simple argument shows that $G_\delta^\infty/\leftrightarrow$ has $2^{\aleph_0}$ elements, but $\Re/\leftrightarrow$ has $\aleph_0$ elements. These models can therefore not be isomorphic. We surmise, however, that the following two claims are true:

- There is a homomorphism from $\Re/\leftrightarrow$ to $G_\delta^\infty/\leftrightarrow$.
- Every guarded recursive specification has a unique solution in $\Re/\leftrightarrow$.

Further research is needed on these points.

3 A finite-graph model for PA

Process Algebra (PA) is an extension of BPA with operators $\|$ (merge) and $\between$ (left-merge) for parallel composition. $X\|Y$ is the parallel merge of X and Y, and $X \between Y$ is the merge in which the first event is a first event from X. Table 3 gives the axioms with which BPA is extended to get PA. We must extend the recursive process graph model to deal with parallel composition, because

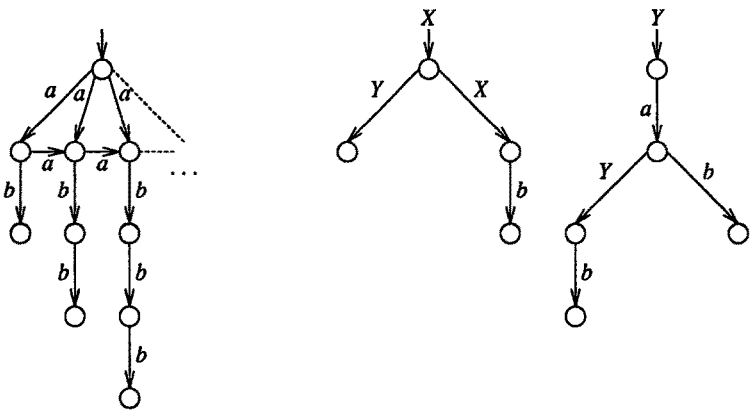


Figure 11: Infinite branching.

$X \| Y = X \mathbb{L} Y + Y \mathbb{L} X$	M1
$a \mathbb{L} X = aX$	M2
$aX \mathbb{L} Y = a(X \mathbb{L} Y)$	M3
$(X + Y) \mathbb{L} Z = X \mathbb{L} Z + Y \mathbb{L} Z$	M4

Table 3: Merge axioms.

processes exist (e.g. a bag) that can be finitely specified in PA, but not in BPA. These processes can therefore not be finitely represented in the recursive process graph model.

By $\mathcal{M}_f(A)$ we mean the set of finite multisets of elements from a set A, and by $\mathcal{M}_{nf}(A)$ we mean the set of non-empty finite multisets of elements from A. To cater for process merge, we now allow labeling an edge with a multiset of process variables,

Definition 29 *A* multi-recursive process component *is a directed, finitely branching, connected, rooted graph that has at least one edge. The edges of a recursive process component are labeled by elements of the set $\mathcal{M}_{nf}(PVAR) \cup Act$ such that any two edges labeled by a multiset, have different end-points. Each multi-recursive process component has a name, which must be a process variable. The set of all multi-recursive process components is called MPC. Metavariables ranging over MPC are $p, q, r, p_1, p_2, \ldots$*

The definitions of $pvar(p)$, $name(p)$ and $root(p)$ go through virtually unchanged. The definition of a *multi-recursive process graph* then is identical to definition 2. Figure 12 gives an example. The set of all multi-recursive process graphs is called $\mathfrak{R}^M$. Using definition 4 of *state*, we can then define:

Definition 30 *A* multistate *of a multi-recursive process graph g is a multiset of states of g. Metavariables ranging over multistates are $A, B, C, \ldots$*

In the following definitions we use $\uplus$ as a binary infix operator denoting multiset union (yielding a multiset). Definitions 31 and 32 are illustrated in figure 13.

Definition 31 *The relation $\rightarrow_g^{push}$ is defined for every multi-recursive process graph g as a binary relation on the multistates of g in the following way:*

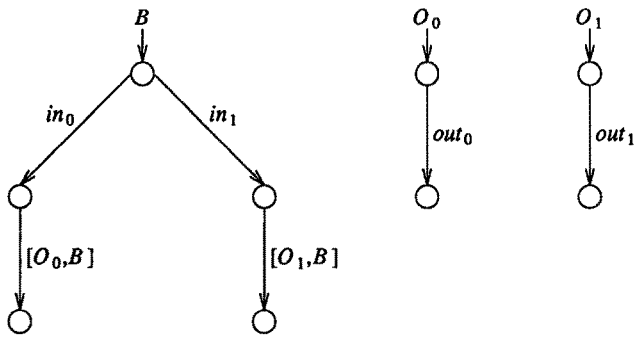


Figure 12: An example multi-recursive process graph: a bag

$A \uplus [(k,s)] \rightarrow_g^{push} A \uplus [(k_1,ls),(k_2,ls),\ldots,(k_n,ls)]$ iff there is an edge in g starting at node k, ending at node l and labeled by some multiset $[X_1,X_2,\ldots,X_n]$ (with $n \geq 1$) such that $\forall 1 \leq i \leq n : k_i = root(rpc_g(X_i))$.

We say that in the above step we have expanded (k,s) to $[(k_1,ls),(k_2,ls),\ldots,(k_n,ls)]$.

The relation $\rightarrow_g^{push*}$ is defined for every recursive process graph g as the reflexive transitive closure of $\rightarrow_g^{push}$.

Definition 32 *The relation $\rightarrow_g^{pop}$ is defined for every multi-recursive process graph g as a binary relation on the multistates of g in the following way:*

$A \uplus [(l_1,ks),(l_2,ks),\ldots,(l_n,ks)] \rightarrow_g^{pop} A \uplus [(k,s)]$ iff $\forall 1 \leq i \leq n : l_i$ is an end node in g and $\forall (l',t') \in A : \nexists t'' : t''ks = t'$.

We say that in the above step we have combined $[(l_1,ks),(l_2,ks),\ldots,(l_n,ks)]$ to (k,s).

The relation $\rightarrow_g^{pop*}$ is defined for every recursive process graph g as the reflexive transitive closure of $\rightarrow_g^{pop}$.

Comparing this with the definition of the pop-relation for BPA process graphs, we see that an important difference is the extra condition $\forall (l',t') \in A : \nexists t'' : t''ks = t'$. This condition states that a parallel composition can only terminate if all of its components have terminated.

Definition 33 *The relation $\rightarrow_g^a$ is defined for every multi-recursive process graph g and atomic action a as a binary relation on the multistates of g as follows: $A \rightarrow_g^a D$ iff there exist multistates B and C such that the following three conditions hold:*

1. $A \rightarrow_g^{pop*} B$
2. $B \rightarrow_g^{push*} C$
3. *$C = C' \uplus [(k,s)]$, $D = C' \uplus [(k',s)]$ and there is an edge in g starting at k, ending at k' and labeled a.*

The definition of bisimulation is virtually identical to definition 12 (taking multistates instead of states and using the action relation on multistates).

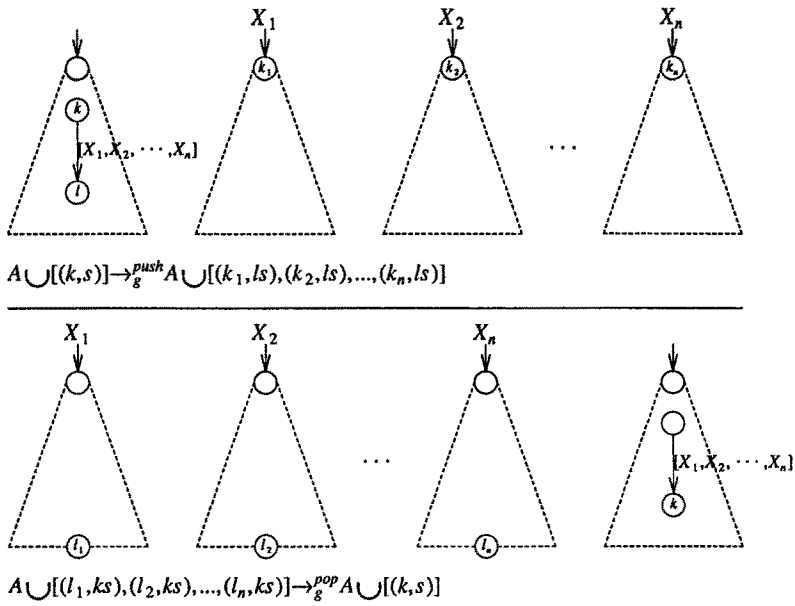


Figure 13: Illustration of the multipush and multipop relations.

Discussion. With the notion of multisets, we can easily define a "multiset step" relation: an action relation that instead on one, can perform a multiset of actions in every step. We define the relation $\rightarrow_g^{[a_1,a_2,\ldots,a_n]}$ on multistates of g as: $A \rightarrow_g^{[a_1,a_2,\ldots,a_n]} D$ iff there exist multistates B and C such that $A \rightarrow_g^{pop*} B$, $B \rightarrow_g^{push*} C$, $C = C' \uplus [(k_1,s_1),(k_2,s_2),\ldots,(k_n,s_n)]$, $D = C' \uplus [(k'_1,s_1),(k'_2,s_2),\ldots,(k'_n,s_n)]$ and for every i such that $1 \leq i \leq n$ there is an edge in g starting at k_i, ending at k'_i and labeled a_i. (Note the similarity between this definition and the definition of the $\rightarrow_g^a$ relation above.)

With the "multistep" relation, we can show an interesting connection between multi-recursive process graphs and (conflict free, prime) event structures [5]. To give this correspondence in some detail:

- Configurations in event structures correspond to multistates in multi-recursive process graphs. Note that there is one important difference: a configuration contains all information about which events have occurred, which is not true for multistates (because a graph may contain cycles, so it is possible to perform some actions and end up in the same multistate one started with.)

- The $\rightarrow_\epsilon$ relation for event structures [5] corresponds to the $\rightarrow_g^{[\ldots]}$ relation in the following way: $X \rightarrow_\epsilon X'$ corresponds to $A \rightarrow_g^{[a_1,a_2,\ldots,a_n]} B$ provided that $X' \setminus X = \{e_1,e_2,\ldots,e_n\}$ and $l(e_i) = a_i$ (for $1 \leq i \leq n$).

Now we can directly transfer four of the semantics defined by van Glabbeek [5] (interleaving trace equivalence, interleaving bisimulation equivalence, step trace equivalence and step bisimulation equivalence) to our model. In fact, the bisimulation relation defined above corresponds exactly to the interleaving bisimulation relation defined by van Glabbeek. The other three equivalence listed above can also be transferred to our model.

Van Glabbeek also defines partial order semantics. These cannot be transferred to our model, because in order to define such semantics, we would have to find a connection between multistates

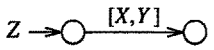


Figure 14: The merge of X and Y

and pomsets. However, multistates do not carry enough information to make such a correspondence possible. **End of discussion.**

The interpretation of the BPA constants and operations $+$ and $\cdot$ is virtually identical to the interpretation in the recursive graph model, just take a multiset of one process variable as a label of an edge when in definition 19 a process variable is used as a label. The operation $\|$ is interpreted as: $g\|h$ (with $g[h] = (P, X)$ and $h = (Q, Y)$) is the recursive process graph $(P \cup Q \cup \{p\}, Z)$, where Z is the first process variable in $PVAR \setminus (pvar\ (g[h]) \cup pvar(h))$ and p the recursive process component with name Z and one edge labeled my the multiset $[X, Y]$. The merge operation is illustrated in figure 14.

Trying to define the left-merge operation for multi-recursive process graphs yields a problem. In [10] it is argued that a left-merge of two (finitely-branching) multi-recursive process graphs may result in an infinitely branching multi-recursive process graph. So left-merge would not be well-defined, as our model only contains finitely branching graphs. The core of the problem is the fact that a finitely branching graph may represent a process containing an infinite choice (see figure 11).

There are two possible solutions to this problem: allow infinite branching or constrain the graphs to be non-circular (which effectively results in deleting the "implicit" infinite branching). We take the second route here. The definitions of the one step process variables function (definition 22), dependency set (definition 23) and circularity (definition 24) for recursive graphs can be directly transferred to the multi-recursive case. We call the set of non-circular process graphs $\mathfrak{R}^{M,\emptyset}$. Now the interpretation of the left merge is simple: we "unwind" both arguments of the left merge to (finitely branching) graphs with no edges labeled by process variables (details are given in [10]). Then we just take the left merge of the two graphs as defined in the standard graph model [1].

Proposition 34 *Bisimulation is a congruence relation on $\mathfrak{R}^{M,\emptyset}$ with respect to the operations of PA.*

Proposition 35 *$\mathfrak{R}^{M,\emptyset}/\leftrightarrows$ is a model of PA.*

Proposition 36 *$\mathfrak{R}^{M,\emptyset}/\leftrightarrows$ and $G/\leftrightarrows$ are isomorphic models of PA.*

Every guarded specification therefore has a unique solution in $\mathfrak{R}^{M,\emptyset}$.

Proposition 37 *Each finite guarded recursive specification not using $\mathbb{L}$ has a finite element in its solution.*

4 Concluding remarks

We have defined some finite-graph models for PA and BPA. Noncircular finite-graph models are isomorphic to known models, and therefore each guarded specification has a unique solution in them. The solution is, as usual, a congruence class of graphs, but in the finite-graph model, this class can be shown to contain a finite element. This result is important for the construction of an editor for process graphs, for it allows the representation of more processes on a finite screen.

An open problems is the question to which standard graph model $\mathfrak{R}^{\emptyset}/\leftrightarrows$ is isomorphic. We plan research on this in the future. Work will also be done on bisimulation-preserving operations on multi-recursive process graphs. These operations will be implemented in the planned editor. An

interesting problem in this respect is the decidability of bisimilarity of recursive process graphs. Finally, user interface matter will be attended to. For example, multi-recursive process graphs can be presented as a simple kind of state chart (see Harel [6]). We will explore these matters in the future.

Acknowledgements. The idea of recursive process graphs arose during a project done by Paul Spruit and Cees Duivenvoorde on a syntax-directed editor for formal specifications [11]. Jan Willem Klop encouraged the writing of this paper and suggested some improvements. Thanks are due to the anonymous referees, who gave some constructive criticism of the paper. Furthermore, one referee pointed out a mistake in definition 29.

References

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

[2] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60:109–137, 1984.

[3] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[4] J.A. Bergstra and J.W. Klop. Algebra of communicating processes. In J.W. de Bakker, M. Hazewinkel, and J.K. Lenstra, editors, *Mathematics and Computer Science (CWI Monographs 1)*, pages 89–138. North-Holland, 1986.

[5] R. J. van Glabbeek. *Comparative Concurrency Semantics and Refinement of Actions.* PhD thesis, Vrije Universiteit/Centrum voor Wiskunde en Informatica, Amsterdam, 1990.

[6] D. Harel. On visual formalisms. *Communications of the ACM*, 31:514–530, 1988.

[7] R. Milner. *A Calculus of Communicating Systems.* Springer, 1980. Lecture Notes in Computer Science 92.

[8] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[9] P.A. Spruit. Two finite-graph models for basic process algebra. Technical report, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, December 1990.

[10] P.A. Spruit. Finite-graph models for Process Algebra with parallel composition. Technical report, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, In preparation.

[11] P.A. Spruit and C.J.A. Duivenvoorde. A syntax- and semantics-directed editor for conceptual model specifications. Technical report, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, April 1990.