EINDHOVEN UNIVERSITY OF TECHNOLOGY

Algebraic specification and implementation of infinite processes

Citation for published version (APA):
Ramesh, S. (1989). *Algebraic specification and implementation of infinite processes*. (Computing science notes; Vol. 8911). Technische Universiteit Eindhoven.

Document status and date:
Published: 01/01/1989

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 16. Nov. 2023

Algebraic specification and implementation of infinite processes

by

S. Ramesh

89/11

September, 1989

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology. Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review. Copies of these notes are available from the author or the editor.

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
All rights reserved
editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.

Algebraic Specification and Implementation of Infinite Processes

by

S. Ramesh

August 1988

Algebraic Specification and Implementation of Infinite Processes

S. Ramesh*
Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O.Box 513
5600 MB Eindhoven
The Netherlands

August 88
Revised November 21, 1988

1 Introduction

Algebraic techniques have proved to be very fruitful in specifying data structures and sequential control structures. Starting from the mid seventies, lot of work has been done on algebraic specification of abstract data types (quoting a few references [GTW78,LZ74,GH78,EM85]). Following this, algebras have been successfully used for specifying more complex control structures, like non determinism and concurrency. These algebras are called process algebras and two such algebras are CCS [Mil80] and ACP [BK84b]. The basic objects of process algebras are processes which are constructed from certain primitive actions using some algebraic operations. Important operations used are sequencing, denoted by ;, non determinism + and concurrency ||. The semantics of these operations are given by defining an equivalence relation among processes. Different equivalence relations with different discriminative powers have been suggested in the literature [Mil80,BK84b].

Process algebras treat primitive actions of processes as *abstract and meaningless* entities and hence what they specify are purely the properties of control structures. But a complete theory of processes has to associate meaning to the atomic processes and study their effect. Such a theory can be developed by integrating process algebras with data algebras and interpreting the primitive actions of processes as well-defined operations on a data type. With the integration of process and data algebras, conventional notions of process equivalences will have to be refined; as a result of giving meaning to atomic actions,

*Supported by a fellowship from EUT and by the foundation for Computer Science Research in the Netherlands (NFI) with financial aid from the Netherlands Organization for the advancement of pure research (NWO).

certain processes, which are considered different by the conventional equivalence relations, may have to be identified.

An important use of integrating process and data algebras is that different techniques and methods available in one of these algebras can be extended to the other. For instance, the notion of implementation has been extensively studied in ADTs and many methods for verifying correctness of implementations are available. But this notion has not at all been studied in process algebras. Our main interest in integrating process and data algebras is to extend the notion of implementation to process algebras and to design methods for proving correctness of implementations of processes.

Recently, some attempts towards integration of process and data algebras have been made [KP87, EPB*88]. In both these approaches, processes have been specified on top of a data type specification. These processes are complex combinations of certain basic operations on the underlying data type. In [KP87,Kap88] the notion of implementation of processes is defined by directly extending the notion available in data algebras and a proof method has been suggested for a restricted class of processes. The class of processes considered are finite processes, i.e., processes that do not involve recursion. In [EPB*88], infinite processes (i.e., recursive processes) are also specified. They are distinguished by their finite projections and a model based upon projective algebras have been discussed. But atomic actions of processes are not interpreted and hence only the control structure aspects are specified. Furthermore, the problem of implementation is not considered at all.

The aim of this paper is to specify finite as well as infinite processes in which atomic actions are given meaning, to extend the notion of implementation to infinite processes and to develop a method for verifying the correctness of implementations. The proposed notion and method of implementation is based upon the ones given in [KP87]. Main problem in introducing infinite processes is defining an appropriate equivalence relation among processes that identifies processes that we want to identify. As in [KP87], finite processes can be distinguished based upon the values they produce upon application. But this can not be used to distinguish infinite processes for the obvious reason that the result of application of an infinite process on a data value is not defined. So we define a new equivalence relation that is based upon what processes do repeatedly rather than what they produce at the end. This relation, when restricted to finite processes coincides with the relation based on the final result of computations.

Our treatment of process specification should be considered as adding ADT flavor to process algebras rather than the other way around. This is because, we differ from conventional data type specifications in one important respect. In ADT specifications, semantics to objects are specified equationally (or in general axiomatically) and one (or more) of the models satisfying the equations is taken as the meaning of specification. Whereas, we do not specify the meanings of processes purely equationally. We take a particular model, and define an equivalence relation over this model. In this respect, our treatment is more like Milner's approach [Mil80]. The reason for doing so is that we find this approach more intuitive. Another reason is that no complete axiomatization of process algebras is possible.

The organization of this paper is as follows: In Section 2, basic materials of ADT techniques are

reviewed quickly and the extension to process specification is discussed. Section 3 is the main section of the paper, where the problems with infinite processes are discussed and a *new notion* of equivalence of processes is proposed. In section 4, the notion of implementation is extended to infinite processes and a method for verifying the correctness of implementation is then discussed. The method is then illustrated with an example.

2 Preliminaries

We shall now briefly review the basic concepts of abstract data type specifications. For the sake of conciseness, our treatment will be very informal and for a formal discussion readers are advised to refer to standard texts on ADTs, for instance [EM85].

Algebraic specification of an ADT is a triple $< S, \Sigma, E >$, where (S, Σ) is called the signature that defines the syntax of the data type while E is a set of equations defining its semantics.

S is a set of names of sorts or types one of which is the type being specified. This type is called the type of interest (TOI). For instance, consider the specification given in Figure 1. It has a sort set containing *queue*, *int*, *bool*. queue is the TOI, while *int*, *bool* are the names of integer and boolean data types respectively.

Σ lists the names of operators and their arities; arity of an operator defines its domains and range. Typically, Σ is specified as shown in Figure 1. The operators in Σ are divided into two kinds: Constructors and derived operations. Constructors are operators with range TOI and any TOI object can be constructed purely using constructors. *new* and *enqueue* are the constructors of queues; any queue object can be constructed from these operators. Whereas *isempty*, *deque* are derived operators. Derived operators are definable in terms of constructors.

In Figure 1, it is assumed that *int* and *bool* are predefined by similar specifications on top of which the given specification is built. In general, there will be a whole hierarchy of specifications starting from some primitive types like integers and boolean and ending with the specification of TOI. Certain consistency and completeness criterion will have to be satisfied by the different specifications in the hierarchy about which we will not go into the details. Interested reader should consult [EM85].

Corresponding to the signature (S, Σ), there is a term model T_D which is the free algebra of well formed terms of sort TOI. These terms are constructed out of the objects of other sorts and operators in Σ whose range is the TOI. The objects of other sorts are, in turn, well formed terms involving the operators in the signatures of their specification. Each term in T_D corresponds to a particular syntactic object of TOI. The semantics to these objects are given by providing a congruence relation over T_D. This congruence relation is specified by means of the equations in E of the specification. These equations define the derived operators in terms of constructors and possibly relate constructor terms. Equations are of the form $M = N$, where M, N are well-formed terms involving the operators in Σ and variables of different sorts in S. The equations determine the quotient algebra T_D/E, which is the algebra consisting of all congruence classes of the terms in T_D. The set of all algebras isomorphic to the

DATA SPEC = Q_SPEC

Sorts(S):

TOI = Queue

Other types : int,bool,error.

Operations(Σ):

Constructors=$new, enqueue$

Derived operators=$dequeue, isempty, front$

$$
\begin{array}{lll}
new: & \rightarrow & queue \\
enqueue : queue \times int & \rightarrow & queue \\
dequeue : queue & \rightarrow & queue \\
isempty : queue & \rightarrow & bool \\
front : queue & \rightarrow & int \cup error
\end{array}
$$

Equations(E)

$$
\begin{array}{lll}
isempty(new) & = & true \\
isempty(enqueue(q,i)) & = & false \\
dequeue(new) & = & new \\
dequeue(enqueue(q,i)) & = & \text{if } isempty(q) \text{ then } new \\
 & & \text{else } enqueue(dequeue(q),i) \\
front(new) & = & error \\
front(enqueue(q,i)) & = & \text{if } isempty(q) \text{ then } i \\
 & & \text{else } front(q)
\end{array}
$$

Figure 1: Specification of Q_SPEC

quotient algebra is the intended semantics of the specification. For our example in Figure 1, equations define $isempty, dequeue$ in terms of the constructors; there is no relation between constructor terms.

Now consider the specification Q_IMPL given in Figure 2. It specifies the data type *implq* which is essentially the standard implementation of queues using an array with two pointers. Objects of type *implq* are quadruples consisting of an array object, a front pointer, a back pointer and a counter. Front and back pointers are indices denoting the front and back end of the queue. For our present purpose, the counter component is not essential; the reason for having this component will be clear when we discuss process specifications.

Let T_D' be the term algebra corresponding to this specification. Then, in order to prove that Q_IMPL indeed is an implementation of Q_SPEC of Figure 1, we make use of a standard notion of implementation. Given two signatures (S, Σ) and (S', Σ'), a *signature morphism* is a mapping $\psi : (S, \Sigma) \rightarrow (S'.\Sigma')$ such that for every sort $s \in S$ there is a sort $\psi(s)$ in S' and if $f : s_1 \times \cdots \times s_n \rightarrow s_n$ then $\psi(f) : \psi(s_1) \times \cdots \times \psi(s_n) \rightarrow \psi(s_n)$. Then given two specifications A and B as before, we say that B specifies an implementation of A, if there is a signature morphism ψ such that for each equation $M = N$ in E, $\psi(M) = \psi(N)$ logically follows from the equations of E', where the variables in M, N are

DATA SPEC = Q_IMPL

Sorts(S'):

TOI = implq.

Other types : array, int, nat, bool, error.

Operations(Σ'):

Constructors=$newa, asgn, tup$

Derived operators=$\overline{new}, \overline{enqueue}, acc$

$$
\begin{array}{lcl}
newa : & \rightarrow & array \\
asgn : array \times nat \times int & \rightarrow & array \\
acc : array \times nat & \rightarrow & int \\
tup : array \times nat \times nat \times nat & \rightarrow & implq \\
\overline{new} : & \rightarrow & implq \\
\overline{enqueue} : implq \times nat & \rightarrow & implq
\end{array}
$$

Equations(E')

$$
\begin{array}{lcl}
acc(newa, i) & = & error \\
acc(asgn(A, i, n), k) & = & \text{if } i = k \text{ then } n \\
& & \text{else } acc(A, k) \\
\overline{new} & = & tup(A, 0, 0, 0) \\
\overline{enqueue}(tup(A, fp, bp, k), n) & = & \text{if } fp \leq bp \text{ then } tup(asgn(A, bp, n), fp, bp + 1, cr) \\
& & \text{else } error \\
asgn(asgn(A, i, n), j, m) & = & \text{if } i = j \text{ then } asgn(A, i, m) \\
& & \text{else } asgn(asgn(A, j, m), i, n)
\end{array}
$$

Figure 2: Specification of Q_IMPL

mapped by ψ onto variables of appropriate types; variables of type *queue* are mapped on to variables of type *implq* and variables of other types are mapped on to those of respective types. It can be easily shown that the above condition is satisfied by our examples by taking ψ to be the mapping that maps the queue operators onto those operators in (S', Σ', E'), denoted by the same name with an overline.

2.1 Process specification

Our discussion on process specification is essentially from [KP87,Kap88] and for more formal details refer to this. A process specification specifies on top of a data type specification, a set of processes which are complex operations on this data type. Processes are constructed from certain *atomic processes* and some composite processes using the following operators: sequencing ;, non deterministic choice +, parallelism ||. Atomic processes include two special processes ϵ, δ which are called null process and deadlocking process respectively. Other atomic processes are operations on the underlying data type. Composite processes are processes constructed out of atomic ones and the operators. In addition to the process-constructing operators, there is a process application operator :: which applies a process

on a data value and yields a data value as a result. Atomic processes, $;, +$ are constructors while $||, ::$ are derived operators. The meaning of ϵ, δ and the operators $;, +, ||, ::$ are fixed and independent of the data type while other atomic processes and composite processes are dependent upon the underlying data type. For this reason, in process specification, only atomic and composite processes will be specified. The meaning of all the derived operators are given in Figure 3 which includes the axiom system BPA due to [BK84b]. Note that in the definition of $||$, a hidden operator $\underline{||}$ is used. It is assumed that data type underlying a process specification has two operators $\cup, \emptyset$ with the properties mentioned in Figure 4. These two properties are required to give meaning to $+$ and δ.

$p\|\|q = p\underline{\|\|}q + q\underline{\|\|}p$	$(a;b);p = a;(b;p)$
$(p+q)\underline{\|\|}r = p\underline{\|\|}r + q\underline{\|\|}r$	$\delta;p = \delta$
$(a;p)\underline{\|\|}q = a;(p\|\|q)$	$\epsilon;p = p$
	$p;\epsilon = p$
$(p+q)+r = p+(q+r)$	$(p+q) :: d = (p :: d) \cup (q :: d)$
$p+q = q+p$	$(a;p) :: d = p :: (a :: d)$
$p+p = p$	$\delta :: d = \emptyset$
$p+\delta = p$	$\epsilon :: d = d$

Figure 3: Basic Process Axioms

$\cup : TOI \times TOI$	$\rightarrow$	TOI
$\emptyset :$	$\rightarrow$	TOI
$d_1 \cup d_2$	$=$	$d_2 \cup d_1$
$d_1 \cup (d_2 \cup d_3)$	$=$	$(d_1 \cup d_2) \cup d_3$
$d_1 \cup d_1$	$=$	d_1
$d_1 \cup \emptyset$	$=$	d_1

Figure 4: Additional operations over Data

Figure 5 gives a process specification PC_SPEC specifying a collection of producer-consumer processes that apply upon queues. These processes make use of two atomic actions *put*, *get* which correspond respectively to producing a value and putting in the queue and consuming a value from the queue respectively. Note that atomic processes are defined by giving their effect on the underlying data type. Whereas, composite processes are defined in terms of atomic processes. PC_SPEC does not have any composite processes.

As in the case of data type specifications, the signature of a process specification defines a term algebra. This term algebra consists of the data terms of the underlying data type and the process terms constructed using atomic processes and the operators $;, +, ||$. The semantics of the data type specification defines a congruence relation over all the data terms. Let us call this relation $\equiv_D$. The equations of Figure 3 define a congruence over the set of all process terms. There is a well-known tree

PROCESS SPEC = PC_SPEC

DATA SPEC = Q_SPEC + $\{\cup, \emptyset\}$

Atomic Processes=*put*, *get*

$put : int \;\; \rightarrow \;\; process$

$get : \;\; \rightarrow \;\; process$

Equations

$put(n) :: q \;\; = \;\; enqueue(q, n)$

$get :: q \;\; = \;\; dequeue(q)$

Figure 5: Specification of PC_SPEC

model of processes [Mil80,HM85], that is isomorphic to the term algebra of process terms. This tree model, denoted by T_P, consists of finite rooted trees whose edges are labeled by the names of atomic processes. In this model, atomic processes are trees with one branch which is labeled by the name of the processes. The operation + is defined as merging the roots of the trees corresponding to its argument. Sequential composition of trees t and t' corresponds to merging the leaves of t_1 with the root of t_2. $||$ is the cross product of the two trees concerned. For every process, there is a unique tree in T_P such that branches from the same father node (i) have distinct labels and (ii) are ordered from left to right based upon an (arbitrary but fixed) ordering of their labels. Hereafter, we shall not distinguish a process from its corresponding tree in T_P that satisfies the above two properties.

As mentioned in the introduction, our intention of integrating process specification with data specifications is to interpret atomic processes and refine the notion of existing process equivalences. The equations of Figure 3 alone are not sufficient to satisfy our intentions as it defines the derived operators, like $||, ::$ but leaves the constructor terms uninterpreted and distinct. Consequently, we define an additional congruence relation over these process terms that identifies more terms. We use the following relation suggested in [KP87,Kap88]. For any two processes p, q,

$$p \equiv_f q \text{ iff } \forall C \forall d : C[p] :: d \equiv_D C[q] :: d$$

where d ranges over underlying data objects while C, called context, ranges over all process expressions involving a free variable of type *processes*. $C[p], C[q]$ are process expressions obtained from C by replacing all occurrences of the free variable in C by p, q respectively. According to this relation, two processes are congruent iff under any context, they produce equivalent values when applied to same data values. Thus two processes are distinguished based upon the values they produce rather than what their syntactic structure is. *The meaning of a process specification is taken to be the set of all algebras isomorphic to the quotient algebra* $T_P / \equiv_f$.

It may be noted that the above congruence is the natural congruence relation one defines when the process specification is considered as an hierarchical extension of the underlying data specifications. Such a view is taken in [KP87,Kap88] but we do not subscribe to this view since it is no longer applicable when infinite processes are considered.

PROCESS SPEC = PC_IMPL

DATA SPEC = Q_SPEC + $\{\cup, \emptyset\}$

Atomic Processes = $inc, upd, \overline{get}$

Composite Processes = $\overline{put}$

$$\begin{array}{lll} inc: & \rightarrow & process \\ upd: int & \rightarrow & process \\ \overline{put}: int & \rightarrow & process \\ \overline{get}: & \rightarrow & process \end{array}$$

Equations

$$\begin{array}{lll} inc :: tup(A, fp, bp, cr) & = & \text{if } fp \leq bp \text{ then } tup(A, fp, bp, cr+1) \\ & & \text{else } \emptyset \\ upd :: tup(A, fp, bp, cr) & = & \text{if } (fp \leq bp) \wedge (cr > 0) \\ & & \text{then } tup(asgn(A, bp, n), fp, bp+1, cr-1) \text{ else } \emptyset \\ \overline{get} :: tup(A, fp, bp, cr) & = & \text{if } fp < bp \text{ then } tup(A, fp+1, bp, cr) \\ & & \text{else } \emptyset \\ \overline{put}(n) & = & inc; upd \end{array}$$

Figure 6: Specification of PC_IMPL

The notion of implementation can be extended in a natural manner to process specifications. For this extend the notion of signature morphism to include process signatures. Given two process specifications, say, A and B, a signature morphism from A to be B is a mapping that when restricted to the data parts is a signature morphism as defined earlier and additionally it maps every atomic process of A onto a process in B and the process operators of A are mapped onto operators in B with the same arity. let T_A, T_B be the corresponding tree models and $\equiv_A$ and $\equiv_B$ be the additional congruence relations defined over T_A and T_B respectively. To avoid notational complexity, we shall not distinguish the fixed part of the signatures (namely, $::, ;, +, ||$ etc.) of different process specifications.

Definition 2.1 *A is an implementation of B if there exists a signature morphism ψ from B onto A such that (i) when it is restricted to the data parts of B the necessary conditions for the data part of A being an implementation of the data part of B are satisfied and (ii) for every atomic process a, data object d and any pair of processes in B,*

$$\begin{array}{rcl} \psi(a :: d) & \equiv_D & \psi(a) :: \psi(d) \\ \psi(p; q) & \equiv_B^A & \psi(p); \psi(q) \\ \psi(p+q) & \equiv_B^A & \psi(p) + \psi(q) \\ \psi(p||q) & \equiv_B^A & \psi(p)||\psi(q), \end{array}$$

where $\equiv_B^A$ is the congruence relation defined over T_B as follows: for two processes p, q in B, $p \equiv_B^A q$ if

for any contexts *in* A,

$$\psi(C)[p] :: d \equiv_D \psi(C)[q]$$

where $\psi(C)$ is obtained from C by replacing all operators in A by their corresponding operators and the variable by a distinct variable. Note that, in general, this relation is different from $\equiv_B$ which considers all contexts formed by its processes. Only a subset of processes in B are relevant as far as A is concerned and hence the use of $\equiv_B^A$ rather than $\equiv_B$.

(i) is the classical problem of showing the correctness of data type implementation. Proving (ii) is dealt in detail in [KP87,Kap88]. It is easy to show that the first three equivalences hold. But the forth equivalence does not hold in general. We shall illustrate this with the help of a simple example. Let $p = a, q = b$ and let $\psi(a)$ and $\psi(b)$ be $a_1; a_2$ and $b_1; b_2$ respectively. Then we have

$$\begin{aligned}
p||q &\equiv a; b + b; a \\
\psi(p||q) &\equiv a1; a2; b1; b2 + b1; b2; b1; b2 \\
\psi(p)||\psi(q) &\equiv a1; a2; b1; b2 + a1; b1; a2; b2 + a1; b1; b2; a2 + \\
&\quad b1; a1; b2; a2 + b1; a1; a2; b2 + b1; b2; a1; a2
\end{aligned}$$

For the sake of notational simplicity, the subscripts to the equivalence relation are dropped in the above. In going from specification to implementation atomicity of operations is changed: in the specification p was atomic but it is implemented as a non atomic operation. Consequently $\psi(p)||\psi(q)$ contain certain 'interleaved' terms that are not in $\psi(p||q)$. In general, $\psi(p||q) \not\equiv \psi(p)||\psi(q)$. However, if all the interleaved terms are equivalent to δ or can be 'regrouped' into one of the non interleaved terms then the required equivalence follows. This is the basic idea behind the proof method proposed in [KP87, Kap88].

We shall now formally give the regroupability condition proposed in [KP87,Kap88], in our notation. We restrict ourselves to the case where any atomic action is implemented by finite sequential terms, i.e., terms involving only ;. The condition for the general case, in which atomic actions are implemented by arbitrary finite terms, is given in [KP87,Kap88].

Given n sequential terms $A_i, i = 1, \cdots, n$ let $Merge(A_1, \cdots, A_n)$ be the set of all terms in which (i) only the actions from $A_i, i = 1, \cdots, n$ occur and (ii) the actions from each A_i occur in the same order in which they occur in A_i; Let Seq be the set of all finite *sequential terms* involving zero, one or more number of atomic actions and the operator ';'. Then let

$$\psi(Seq) = \{\psi(t) | t \in Seq\}$$

For any two sequential terms A, B, let $A \leq B$ stand for the fact that A is an initial subsequence of B. Let $A, B \in \psi(Seq)$. Then the regroupability condition is given by

(REG) $\forall t \in Merge(A, B) : (t \equiv t') \vee (t = \delta)$

where $t' \in Merge(A, B)$ and $t' \in \psi(Seq)$.

Theorem 2.2 (Kaplan) *If (REG) holds then for any p, q*

$$\psi(p)||\psi(q) \equiv \psi(p||q)$$

Let $A_i \in \psi(Seq), i = 1, \cdots, n$ for some n. Then from the definition of congruence we have:

Corollary 2.3 *For any p, q,*

$$\psi(p)||\psi(q) \equiv \psi(p||q)$$

if for every $t \in Merge(A_1, \cdots, A_n)$

$$\forall d \in TOI : (t :: s = \emptyset) \vee (t :: s = t' :: s)$$

where $t' \in Merge(A_1, \cdots, A_n)$ and $t' \in \psi(Seq)$.

3 Infinite processes

Processes specified in the last section are finite processes as they involve finite number of application of atomic actions. Processes involving infinite number of applications of atomic actions can be specified using recursive specifications. A recursive specification is, in general, a finite set of equations of the form $X_i = E_i, i = 1, \cdots, n$, where X_i are variables of type process and E_i are expression of type process that involve one or more variables X_i. A simple example of recursive specifications is $X = a; X$. This specification denotes a process X that does an infinite number of action a. We consider only a sub class of recursive terms, referred in the literature [BK84a] as *guarded* recursive terms. A recursive term is guarded if its body is guarded. A term is guarded iff it is *not* unguarded. A term is unguarded, iff either it is a process variable or it is of the form $t; t', t + t'$ or $t||t'$, where t, t' are unguarded.

To add recursive terms to the process specification, we add a countable number of zero-ary operators, denoted by $X_i, i = 1, 2, \cdots$. Furthermore, to our original set of equations given in Table 1, we add a countable number of finite set of equations, of the form mentioned above, each of which involving a finite number of X_i's.

As in the case of finite processes, a term algebra of processes can be defined for the present specification. This algebra includes the finite term algebra and, in addition, has an infinite number of objects corresponding to the different X_i's. The equations of Table 1 and the newly added equations define a congruence relation over these terms and induces a quotient algebra. The tree model discussed in the last section, can be extended, by including infinite trees, to get a model that is isomorphic to the quotient algebra. This model, which also we denote by T_P, consists of finitely branching rooted trees. The trees may be finite or infinite. Finite trees correspond to finite processes while infinite ones to the infinite processes. As before, we identify a process with a tree such that branches from a common father node have distinct labels and are arranged from left to right in an order based upon an order on the atomic actions. An example is the tree corresponding to the term $X = (a + b)X$ shown in Figure 7. As before, for every process term there is a unique tree in T_P and we do not distinguish a process term from the corresponding tree in T_P.

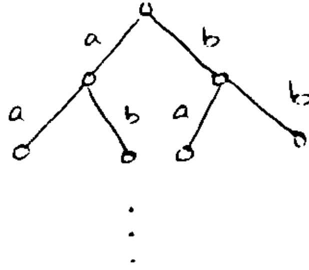

Figure 7: $X = (a + b)X$

As in the finite process case, we define an additional congruence relation over processes for the same reason. In the case of finite processes, processes were distinguished based upon the the values they yield upon application on the underlying data type. Obviously, this can not be used for infinite processes since the result of application of an infinite sequence of operations is undefined. So we propose a new congruence relation for processes. Since processes can be distinguished only based on their effect on the underlying data type and the effect of an infinite process is undefined, a natural way out is: define finite approximations of processes, which are finite processes and relate two processes by comparing their finite approximations. There is one standard method suggested in the literature [BK84b], for obtaining finite approximations: use of *Projection* function Pr. Given a process (tree) p and a natural number n, $Pr(p, n)$ is the subprocess of p obtained by 'cutting' p at depth n. It is defined as follows:

$Pr(p, 0) = \epsilon$

$Pr(p, n + 1) = a; Pr(p, n)$

$Pr(p + p', n) = Pr(p, n) + Pr(p', n)$

Then given two processes, say, p and q, p and q are equivalent iff for every n, $Pr(p, n) \equiv_f Pr(q, n)$, where $\equiv_f$ is the relation used for comparing finite processes. We shall call this relation *Projection Equivalence*[1].

But we find projection equivalence to be more discriminating than we desire and based almost on the syntactic structure of processes, although it makes use of the relation used for finite processes. For instance, projection equivalence is, in general, *incompatible* with the finite process equivalence relation $\equiv_f$ defined earlier, i.e., two finite processes can be equivalent with respect to $\equiv_f$ but may not be projection equivalent. A simple example shows this. Consider two processes a and $b; b$ such that $a \not\equiv_f b$ but $a \equiv_f b; b$. Obviously these two processes are not projection equivalent. Projection equivalence is too restrictive since it requires each and every finite approximations to be equivalent. In general, two processes are projection equivalent iff they are identical (up to the equivalence induced by Table 1) at

[1] Projection equivalence as defined in [BK84b] is slightly different from our equivalence as the relation used for comparing finite approximations are different.

all finite depths (except possibly differing at infinity).

So we proceed to define a new equivalence relation that is less restrictive and compatible with the finite process equivalence. For this purpose, we associate with each process t in T_P, a set of functions OBS_t, whose domain is natural numbers and whose range is the set of all finite terms; OBS stands for observers. Each function in OBS_t gives for each natural number, a finite subtree of t . To formally describe OBS_t we require a few definitions:

A *trace* of a tree is a finite path, in the tree, starting from the root of the tree. Any trace will be denoted by the finite sequential term involving the labels of the edges in the trace.

Given two trees t_1 and t_2, we define a relation $\prec$ such that $t_1 \prec t_2$ iff t_1 is strictly a subtree of t_2 having the same root as t_2.

Then given a tree $t \in T_P$, any function $f \in OBS_t$ satisfies the following conditions:

(F1) for every n, $f(n)$ is a *finite* subtree of t rooted at the root of t

(F2) For any k, $f(k) \prec f(k+1)$.

(F3) Every trace of t is also a trace of $f(k)$ for some k.

Intuitively, f gives an arbitrarily larger and larger subtrees of t for increasing values of k and in the limit yields a tree containing all its traces.

Now we can define our equivalence relation. We say that p and q are related by a relation $\mathcal{R}$ iff there exist functions $f_1 \in OBS_p, f_2 \in OBS_q$ such that

$$\forall n : f_1(n) \equiv_f f_2(n).$$

The intuitive meaning of this relation is that p and q are related by $\mathcal{R}$ if they have an infinite number of equivalent finite subprocesses of arbitrarily large size. We shall illustrate this definition with an example: Consider the terms $X = aX$ and $Y = bY$, where $a \equiv_f b;b$. These two terms are related by $\mathcal{R}$ since there exist two functions f_1, f_2 satisfying the required condition. These functions are:

$$f_1(n) = a^{2n}$$

$$f_2(n) = b^n$$

$\mathcal{R}$ is obviously reflexive and symmetric but it is not immediately clear whether it is transitive or not. We safely take the transitive closure of $\mathcal{R}$ to get the required equivalence. Let us denote the transitive closure of $\mathcal{R}$ by $\mathcal{R}^*$. Then

Fact 3.0.1 *$\mathcal{R}^*$ is an equivalence relation*

Hereafter we shall denote $\mathcal{R}^*$ by $\equiv$. It is not very difficult to prove that

Fact 3.0.2 *For finite terms t_1 and t_2, $t_1 \equiv t_2$ iff $t_1 \equiv_f t_2$.*

Thus $\equiv$ is compatible with the earlier equivalence relation. Next we have the following important result:

Lemma 3.1 *$\equiv$ is a congruence relation.*

A rough sketch of the proof of this lemma is as follows: Let $t_1 \equiv t_2$. To prove congruence we have to show that for any term t it is the case that $(t_1\, op\, t) \equiv (t_2\, op\, t)$, where $op = +, ;$ or $||$. The proof for $+$ and ; are straightforward. So we prove the congruence for $||$. By assumption we have that there exist functions $f_1 \in OBS_{t_1}, f_2 \in OBS_{t_2}$, such that $f_1(k) \equiv_f f_2(k)$. The proof is complete once we show that there exist $f_1' \in OBS_{t_1||t}, f_2' \in OBS_{t_2||t}$ such that $\forall k : f_1'(k) \equiv_f f_2'(k)$. We claim that the following function satisfies the required condition:

$$f_1'(k) = Pr(t_1||t, k)||f_1(k)$$

$$f_2'(k) = Pr(t_2||t, k)||f_1(k)$$

where Pr is the projection function defined earlier. It can be easily seen that f_1', f_2' satisfy (F1) and (F2). The fact that (F3) also holds follows from the fact that every finite trace of t is also trace of $Pr(t, n)$ for some n.

It is easy to see that $\equiv$ is coarser than projection equivalence: For any $t \in T_p$, the function f, defined as $f(n) = Pr(t, n)$ is a member of OBS_t and hence projection equivalence is contained in $\equiv$. Two terms that are not projection equivalent can be related by $\equiv$.

Now we look at the problem of implementation.

4 Implementation of Infinite processes

The notion of implementation can be directly extended to infinite process specifications: Let A and B be two process specifications with $\equiv_A$ and $\equiv_B$ as the corresponding equivalence relations. Then we say a mapping ψ from A to B, a signature morphism, if it is an extension of a signature morphism of finite process specifications of A and B such that it maps every set of variables X_i in A defined by $X_i = E_i$, to a set of process variables Y_i satisfying $Y_i = E_i'$, where the E_i' is obtained from E_i by replacing all occurrences of process variables, operators and basic processes by the corresponding variables, operators and processes in B respectively. Then,

Definition 4.1 *B specifies an implementation of A if there exists a signature morphism from A to B such that*

(a) *all the equations of Table 1 are satisfied.*

(b) *for any p, q in A, if $p \equiv_A q$ then $\psi(p) \equiv_B^A \psi(q)$, where the latter relation is as defined in the last section.*

In order to prove that (a) holds we have to show exactly the same four equivalences mentioned in Section 3, but with the difference that the process terms involved may be infinite. As in the case of finite processes, it is easy to show that the first three equivalences hold. It is also easy to show that (b) holds. The argument is as follows: Let $t_1 \equiv_A t_2$ for two terms t_1, t_2 in A. Then there exist functions $f_1 \in$

$OBS_{t_1}, f_2 \in OBS_{t_2}$ such that $\forall n : f_1(n) \equiv_A f_2(n)$. In order to show $\psi(t_1) \equiv_B^A \psi(t_2)$, we have to show that there exist functions $f_1' \in OBS_{\psi(t_1)}, f_2' \in OBS_{\psi(t_2)}$, such that $\forall n : f_1(\psi(t_1), n) \equiv_B^A f_2(\psi(t_2), n)$. We claim that the following functions satisfy the necessary conditions:

$$f_i'(n) = \psi(f_i(n)), i = 1, 2.$$

The proof is complete once we show that these functions satisfy the conditions (F1) - (F3) given in the previous section. It is straightforward to show that (F1) and (F2) holds. (F3) also holds follows from the following argument: Take any trace τ in $\psi(t_i)$. From the definition of ψ it follows that there is a longer trace τ' which is a ψ - image of a trace in t_i, which is in $f_i(t_i, n)$ for some n. Hence τ' as well as τ is in $f_i'(t_i, n)$.

In general, the fourth equivalence namely,

$$\psi(p||q) \equiv_B^A \psi(p)||\psi(q)$$

does not hold. However, we have

Lemma 4.2 *If the forth equivalence holds for finite terms and $\psi(a)$ is a finite term for any atomic process a then it holds for infinite terms as well.*

The proof of this lemma is as follows: Let f_1, f_2 be two arbitrary functions in OBS_p and OBS_q respectively. Consider f and g defined as follows:

$$f(n) = \psi(f_1(n))||\psi(f_2(n))$$

$$g(n) = \psi(f_1(n)||f_2(n))$$

Since $f_1(p, n)$ and $f_2(q, n)$ are finite terms, if the fourth equivalence holds for finite terms then $f(n) = g(n)$. The proof is complete once we show that f and g as defined above satisfy the conditions (F1) - (F3). This proof is straightforward.

When ψ is such that for any atomic action a, $\psi(a)$ is a finite sequential term, then from Theorem 2.2, we have that

Corollary 4.3 *If (REG) holds then the fourth equivalence holds for finite as well as infinite terms.*

We shall conclude this section by illustrating the proof method on a simple example. Consider the specification PC_IMPL given in Figure 6. We will prove that PC_IMPL specifies an implementation of PC_SPEC given in Figure 5. There is a morphism ψ from PC_SPEC to PC_IMPL given as follows:

$$\psi(Q_SPEC) = Q_IMPL$$

$$\psi(\overline{put}) = \overline{put}$$

$$\psi(\overline{get}) = \overline{get}$$

ψ maps other operators of PC_SPEC onto corresponding operators in PC_IMPL. IMPL_Q has been shown to be an implementation of SPEC_Q. Hence what remains to be done is to check that the

regroupability condition holds. To check this we take any term in $Merge((\overline{put})^m, (\overline{get})^n)$ for some m, n and show that this term is equivalent to a term in $\psi(Merge((put)^m, (get)^M))$. This indeed is the case follows from a simple inductive argument using the following equivalences:

$$inc; get^k; upd \equiv_B get^k; inc; upd$$

$$inc; get^k; inc \equiv_B get^k; inc; inc$$

for any k. The reason why these equivalences hold is that *inc* and *get* changes disjoint components of *tup*.

5 Conclusion

We have extended the ADT techniques to specify and verify (infinite) processes. This extension required a new definition of equivalence of processes. Using this new definition of equivalence, we have shown that the sufficient condition for one process specification being an implementation of another is exactly same as the one required when only finite processes are involved.

There are many more extensions possible. In this paper we have considered implementations in which an atomic action is implemented by a finite process. It is interesting to consider implementations in which atomic actions are implemented by infinite processes. This is not merely a theoretical extension: Conventional synchronization primitives like semaphores are considered as atomic at one level but implemented by an infinite wait at a lower level. We are presently working on this.

6 Acknowledgements

The author is grateful to Rob Gerth for bringing to his attention Kaplan's work [KP87].

References

[BK84a] J. A. Bergstra and J. W. Klop. The algebra of recursively defined processes and the algebra of regular processes. In *ICALP '84*, pages 82 – 95, Springer Verlag, July 1984.

[BK84b] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1), 1984.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications 1.* Springer Verlag, 1985.

[EPB*88] H. Ehrig, F.P- Presicce, P. Boehm, C. Rieckhoff, C. Dimitrovici, and M. G-. Rhode. *Algebraic Data Type and Process Specifications Based on Projection Spaces.* Technical Report, Technical University Berlin, West Germany, 1988.

[GH78] J.V. Guttag and J.J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10(1), 1978.

[GTW78] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An initial approach to the specification, correctness and implementation of abstarct data types. In *Current Trends in Programming Methodology*, pages 80–144, Prentice-Hall, New Jersey, 1978.

[HM85] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32, 1985.

[Kap88] S. Kaplan. *Algebraic Specification of Concurrent Systems.* Technical Report, Department of Computer Science, Hebrew University of Jerusalem, Israel, May 1988.

[KP87] S. Kaplan and A. Pnueli. Specification and implementation of concurrently accessed data structures: an abstract data type approach. In *Proc. Symposium on Theoretical Aspects of Computer Science*, pages 220–244, Springer Verlag, 1987.

[LZ74] B. Liskov and S. Zilles. Programming with abstract data types. *SIGPLAN Notices*, 9(4), 1974.

[Mil80] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1980.

In this series appeared :

No.	Author(s)	Title
85/01	R.H. Mak	The formal specification and derivation of CMOS-circuits
85/02	W.M.C.J. van Overveld	On arithmetic operations with M-out-of-N-codes
85/03	W.J.M. Lemmens	Use of a computer for evaluation of flow films
85/04	T. Verhoeff H.M.J.L. Schols	Delay insensitive directed trace structures satisfy the foam rubber wrapper postulate
86/01	R. Koymans	Specifying message passing and real-time systems
86/02	G.A. Bussing K.M. van Hee M. Voorhoeve	ELISA, A language for formal specifications of information systems
86/03	Rob Hoogerwoord	Some reflections on the implementation of trace structures
86/04	G.J. Houben J. Paredaens K.M. van Hee	The partition of an information system in several parallel systems
86/05	Jan L.G. Dietz Kees M. van Hee	A framework for the conceptual modeling of discrete dynamic systems
86/06	Tom Verhoeff	Nondeterminism and divergence created by concealment in CSP
86/07	R. Gerth L. Shira	On proving communication closedness of distributed layers
86/08	R. Koymans R.K. Shyamasundar W.P. de Roever R. Gerth S. Arun Kumar	Compositional semantics for real-time distributed computing (Inf.&Control 1987)
86/09	C. Huizing R. Gerth W.P. de Roever	Full abstraction of a real-time denotational semantics for an OCCAM-like language
86/10	J. Hooman	A compositional proof theory for real-time distributed message passing
86/11	W.P. de Roever	Questions to Robin Milner - A responder's commentary (IFIP86)
86/12	A. Boucher R. Gerth	A timed failures model for extended communicating processes

86/13	R. Gerth W.P. de Roever	Proving monitors revisited: a first step towards verifying object oriented systems (Fund. Informatica IX-4)
86/14	R. Koymans	Specifying passing systems requires extending temporal logic
87/01	R. Gerth	On the existence of sound and complete axiomatizations of the monitor concept
87/02	Simon J. Klaver Chris F.M. Verberne	Federatieve Databases
87/03	G.J. Houben J.Paredaens	A formal approach to distri- buted information systems
87/04	T.Verhoeff	Delay-insensitive codes - An overview
87/05	R.Kuiper	Enforcing non-determinism via linear time temporal logic specification.
87/06	R.Koymans	Temporele logica specificatie van message passing en real-time systemen (in Dutch).
87/07	R.Koymans	Specifying message passing and real-time systems with real-time temporal logic.
87/08	H.M.J.L. Schols	The maximum number of states after projection.
87/09	J. Kalisvaart L.R.A. Kessener W.J.M. Lemmens M.L.P. van Lierop F.J. Peters H.M.M. van de Wetering	Language extensions to study structures for raster graphics.
87/10	T.Verhoeff	Three families of maximally nondeter- ministic automata.
87/11	P.Lemmens	Eldorado ins and outs. Specifications of a data base management toolkit according to the functional model.
87/12	K.M. van Hee and A.Lapinski	OR and AI approaches to decision support systems.
87/13	J.C.S.P. van der Woude	Playing with patterns, searching for strings.
87/14	J. Hooman	A compositional proof system for an occam- like real-time language

87/15	C. Huizing R. Gerth W.P. de Roever	A compositional semantics for statecharts
87/16	H.M.M. ten Eikelder J.C.F. Wilmont	Normal forms for a class of formulas
87/17	K.M. van Hee G.-J.Houben J.L.G. Dietz	Modelling of discrete dynamic systems framework and examples
87/18	C.W.A.M. van Overveld	An integer algorithm for rendering curved surfaces
87/19	A.J.Seebregts	Optimalisering van file allocatie in gedistribueerde database systemen
87/20	G.J. Houben J. Paredaens	The R^2 -Algebra: An extension of an algebra for nested relations
87/21	R. Gerth M. Codish Y. Lichtenstein E. Shapiro	Fully abstract denotational semantics for concurrent PROLOG
88/01	T. Verhoeff	A Parallel Program That Generates the Möbius Sequence
88/02	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable Specification for Information Systems
88/03	T. Verhoeff	Settling a Question about Pythagorean Triples
88/04	G.J. Houben J.Paredaens D.Tahon	The Nested Relational Algebra: A Tool to handle Structured Information
88/05	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable Specifications for Information Systems
88/06	H.M.J.L. Schols	Notes on Delay-Insensitive Communication
88/07	C. Huizing R. Gerth W.P. de Roever	Modelling Statecharts behaviour in a fully abstract way
88/08	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	A Formal model for System Specification
88/09	A.T.M. Aerts K.M. van Hee	A Tutorial for Data Modelling

No.	Author(s)	Title
88/10	J.C. Ebergen	A Formal Approach to Designing Delay Insensitive Circuits
88/11	G.J. Houben J.Paredaens	A graphical interface formalism: specifying nested relational databases
88/12	A.E. Eiben	Abstract theory of planning
88/13	A. Bijlsma	A unified approach to sequences, bags, and trees
88/14	H.M.M. ten Eikelder R.H. Mak	Language theory of a lambda-calculus with recursive types
88/15	R. Bos C. Hemerik	An introduction to the category theoretic solution of recursive domain equations
88/16	C.Hemerik J.P.Katoen	Bottom-up tree acceptors
88/17	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable specifications for discrete event systems
88/18	K.M. van Hee P.M.P. Rambags	Discrete event systems: concepts and basic results.
88/19	D.K. Hammer K.M. van Hee	Fasering en documentatie in software engineering.
88/20	K.M. van Hee L. Somers M.Voorhoeve	EXSPECT, the functional part.
89/1	E.Zs.Lepoeter-Molnar	Reconstruction of a 3-D surface from its normal vectors.
89/2	R.H. Mak P.Struik	A systolic design for dynamic programming.
89/3	H.M.M. Ten Eikelder C. Hemerik	Some category theoretical properties related to a model for a polymorphic lambda-calculus.
89/4	J.Zwiers W.P. de Roever	Compositionality and modularity in process specification and design: A trace-state based approach.
89/5	Wei Chen T.Verhoeff J.T.Udding	Networks of Communicating Processes and their (De-)Composition.
89/6	T.Verhoeff	Characterizations of Delay-Insensitive Communication Protocols.
89/7	P.Struik	A systematic design of a paralell program for Dirichlet convolution

89/10	S.Ramesh	A new efficient implementation of CSP with output guards.
89/11	S.Ramesh	Algebraic specification and implementation of infinite processes.