Finiteness conditions and structural construction of automata for all process algebras

Eric Madelaine [**] and Didier Vergamini [*]

[**]INRIA
Route des Lucioles, Sophia Antipolis
06565 Valbonne Cedex (France)
email: madelain@mirsa.inria.fr

[*]CERICS
Rue Albert Einstein, Sophia Antipolis
06565 Valbonne Cedex (France)
email: dvergami@mirsa.inria.fr

Abstract

Finite automata are the basis of many verification methods and tools for process algebras. It is however undecidable in most process algebras whether the semantics of a given term is finite. We give sufficient finiteness conditions derived from the analysis of the operational rules of the algebra operators. From these rules we also generate the functions that compute automata from terms of the algebra. These constructions allow one to use our verification tools for programs written in many process algebras.

Keywords verification, concurrent systems, process algebra, lotos, structural operational semantics, finite automata.

1 Introduction

Verification methods for concurrent systems can be classified in at least three families: theorem proving methods, model-checking, and automata based methods. The first family holds the biggest theoretical power; it may be applied to many sort of undecidable problems and in some sense it can deal with infinite objects. However theorem proving methods have usually a high complexity and there is few hope to make these methods purely automatic. The ECRINS system ([MdSV89]) uses theorem proving methods, together with specialized algorithms, to check for the validity of bisimulation laws in process algebras. The approach is general enough to consider most usual process calculi from the literature; the semantics of the operators is defined in user-defined *calculus description files*, and used by the system to generate specialized behaviour-evaluation algorithms.

We want to apply this parameterized approach for building tools based on automata analysis. The system AUTO ([dSV89], [RS89]) is dedicated to *verification by reduction* of parallel and concurrent programs. AUTO deals only with terms that have finite automata representations. Its main activities are the construction of automata from terms of process algebras and the reduction and comparison of automata along a large family of equivalences. These activities are mostly intertwined, according to congruence properties of the equivalences that allow for reducing subterms of operators before building any global automaton. This approach cuts off partially the space explosion that causes the well-known limitation of such techniques.

The current AUTO system is using a subset of the MEIJE calculus ([Bo85]) as input language. To ensure that terms have finite representations, we use a two layers structure for input terms. In the lower layer, one can write recursive definitions directly encoding automata: recursive variables correspond to states and transitions are specified through action prefixing and non-deterministic

choice (see the example 1 in section 2.2); these are *dynamic* operators, for they build the behaviour of components of a system. In the upper layer, one builds networks of automata using *static* operators (asynchronous parallel composition, renamings of signals, and a restriction operator). Finiteness of automata is guaranteed by forbidding occurrences of the parallel and renaming operators inside the recursive definitions. Observational equivalence appears to be a congruence for the MEIJE parallel and restriction operators, so lower layer automata can be reduced before being composed.

Extending the structural construction of automata to parameterized process algebra, we need new finiteness conditions, computable from the very definitional rules of the operators. Here the splitting between static and dynamic operators makes less sense, as many operators can be used both in dynamic and static positions. Moreover there are operators that are asymmetric; recursion on the left argument of the *enable* and *disable* operators of LOTOS may generate infinite structures, whereas recursion on their right arguments can be used safely. We shall deduce from analysis of their rules which operators may in which position accept a recursive variable as one of its arguments, and which operators are preserving finiteness of automata. The rules analysis also provides us functions associated with each operator for building the automata.

In section 2, we give an overview of the concepts from process calculi theory we need in the paper, including a description of the syntax we allow for structural operational semantic rules. In section 3, we discuss finiteness conditions and explain the classification of operators obtained from the analysis of the rules. In section 4 we describe the algorithms for structural construction of the automata, and in the conclusion we describe a prototype system that uses this generic technique and discuss current work.

2 Process Calculi

Process calculi are now a well-accepted generic notion for designing a class of formalisms which share the same definitional principles : CCS [Mi80], SCCS [Mi83], MEIJE [Bo85], TCSP [Bro83], ACP [BK86], BasicLOTOS [BB88] to name a few. We shall assume reader's acquaintance with at least one of these languages and its definitional mechanisms.

Process calculi are based on two main types : *actions* and *processes*. Operators take actions and processes arguments into processes, providing a classical algebraic structure. Operational semantics provides interpretation of closed terms into transition systems, with actions as transition labels. Operators and non-closed expressions are then interpreted as transition system transformers; this semantics is defined through behaviour rules in a structural operational style, with a particular format (see [dSi85], [VG88]). We shall describe our format for rules in section 2.4.

Special operators are action renamings and recursive definitions. They are present in all process calculi.

2.1 Actions

In all process algebras actions are themselves structured. This structure is what allows for synchronization and further communication to be handled in relevant operators rules. Just recall the inverse signals in CCS, which meets in synchronization and produce a hidden action τ. In SCCS and MEIJE there is a full commutative monoid of potential simultaneous actions, again containing a group of invertible signals. Actions structure in ACP is more scarce and parametric, while in BasicLOTOS it is only a set of so-called gate names without structure, but for a distinguished termination action δ.

2.2 Recursive definitions

Most process algebras have some sort of recursion operator. In AUTO and ECRINS we use a common recursive definition mechanism for all process algebras. Here is an example of a recursive definition,

written with MEIJE operators:

Example 1

```
let rec {x = a:x + b:x +c:stop
     and y = a:y + b:z
     and z = a:z + c:c:y }    in x//y
```

Such recursive definitions are used in AUTO for building finite automata, perhaps composed later on by other operators of the algebra. One can also build infinite structures, not suitable for analysis in AUTO, such as:

Example 2

```
let rec {many-processes = one-process // many-processes}
    in many-processes
```

2.3 Definition of process algebras

A calculus description contains the concrete syntax and abstract syntax definitions of the calculus operators, together with *structural conditional rules* that gives them an operational semantics. The ECRINS *calculus compiler* uses the first part to produce a scanner, a parser and abstract syntax structures for expressions of the calculus. From the semantics part, the compiler will produce the functions for building and combining automata from terms. We now sketch the allowed syntax formats using examples. Full documentation will be found in [MdSV89] and is recalled shortly below.

```
operator hiding:: Process Label --> Process
syntax   \\   left 4
semantics
   hiding                  p -- a --> p'  &  (a equal s)
                           ----------------------------
                              p\\s -- tau --> p'\\s
   hiding-not            p -- a --> p'  &  not (a equal s)
                         ---------------------------------
                               p\\s -- a --> p'\\s
end
```

The first line declares the name and the signature of the operator. Then follows the concrete syntax description, along with associativity and priority declarations aimed at the parser generator. Finally the operational rules are listed. The end right upper part of the rules (e.g. `(a equal s)`) is a predicate over actions conditioning the applicability of the rule. Such predicates implement synchronization conditions over the actions of the operator subterms. The plain format of the structural conditional rules will be explain below. Here are two other examples, respectively from MEIJE and BASICLOTOS:

```
operator ticking :: Action Process --> Process
syntax   * right 3
semantics
        ticking       p -- a --> p'
                     -------------------
                     s*p -- a.s --> s*p'
end
```

```
operator disable :: Process Process --> Process
syntax   [> left 4
semantics
        disabling_1      p -- a --> p' & not(a equal d)
                         -------------------------------
                             p[>q -- a --> p'[>q
        disabling_2      p -- a --> p' & (a equal d)
                         ---------------------------
                             p[>q -- d --> p'
        disabling_3             q -- a --> q'
                                ----------------
                                p[>q -- a --> q'
end
```

2.4 The Conditional Rewrite Rules format

The rules must obey the following format:

$$\frac{\{x_j \xrightarrow{u_j} x'_j\}_{\mathcal{J}\subset[1..n]} \quad \& \quad P(\{u_j\}_{\mathcal{J}}, a_1, \ldots, a_m)}{Op(x_1, \ldots, x_n, a_1, \ldots, a_m) \xrightarrow{F(\{u_j\}_{\mathcal{J}}, a_1, \ldots, a_m)} T'(\{x_k\}_{[1..n]-\mathcal{J}}, \{x'_k\}_{\mathcal{J}}, a_1, \ldots, a_m)}$$

Many definitions in this paper rely on the syntax allowed for the various elements of conditional rules. Let us give precise names to these elements.

Definition 1 *We call:*

- *premises the upper part of the rule and conclusion its bottom part.*
- *subject the term at the left end part of the conclusion. The head operator Op of the subject has process arguments $x_1, \ldots, x_n$ and action parameters $a_1, \ldots, a_m$. Inside P, F and T' some other actions may appear: they are global constants of the calculus and had to be declared as such previously.*
- *formal hypothesis the part of the premises on the left of the & and working formal variables the x_j with $j \in \mathcal{J}$.*
- *actions predicate the part of the premises on the right of the &. P belongs to boolean operators closure of the following basic predicates :* equality, divisibility, set membership.. *over actions terms with synchronization product.*
- *resulting action the label over the arrow of the conclusion. The resulting action F is a function of the formal hypotheses actions (and of the operator action parameters).*
- *resulting process the right end part of the conclusion. The process arguments that appear as formal hypothesis must be transformed as x'_j in the resulting process. This condition does not allow to test a potential future behaviour of a process, without making it explicitly perform its action within the considered rule, therefore saving the possibility of choosing another future. This is a restriction to the format in [VG88]. Beside this condition, the resulting process is built from the action and process variables, and the operators of the calculus (excluding recursive definitions).*

3 Finiteness Conditions

The preceding conditional rules defines in a structural manner the semantics function that computes a *transition system* from a (closed) term of a process algebra. This definition is constructive: you can compute each transition of the transition system by building proof trees which nodes are instances of the rules.

Definition 2 *A term of a process algebra has* transition-system finiteness *(TSF) iff the transition system computed from the term using the operators' rules is strongly bisimular to some finite transition system (i.e. with a finite number of states and a finite number of transitions).*

In many process algebras with recursion, this property is undecidable. Sufficient syntactic conditions to ensure FTS will be given in this section, for any process algebra defined using the conditional rewrite rules from the preceding section. As far as possible, these conditions will be expressed in terms of the semantic rules of the operators. We shall use the following notions:

guarded recursion: It is possible to build infinite proof trees for terms containing recursive definitions. Can we found syntactic conditions to guaranty that all proof trees are finite?

non-growing operators: Making the assumption that the arguments of an operator have finite semantics, is it always true that their composition by this operator is a finite automaton? This property holds for most classical operators, but the rule format allows to define exotic operators that create infinitely many states.

sieves: Some unary operators have the nice property that the resulting automaton has exactly the same states than the argument automaton, only some transitions being transformed, erased, or added. We implement their semantics by *sieves*, that is functions that only modify the transitions of the system. Which operators may be implemented in this way?

switches: Inside a recursive definition, the use of recursive variables should be limited in some way, in order to avoid building infinitely many states, or states with infinitely many transitions. Clearly, parallel composition operators and non-alphabetical renamings should be somehow forbidden inside recursive definitions. At which places (defined as occurrences of operator arguments) is a recursive variable allowed to appear?

3.1 Guarded recursion

In order to avoid divergence in the proof tree construction, we introduce as usual a notion of guarded terms. We define here this notion in a rather abstract way, and we shall give a generic algorithm that computes it in a further section. The definition relies on the fact that if a proof tree is infinite, then either it contains a pattern that occurs infinitely, or the subject of its nodes are strictly growing along the infinite branches. The *guarded* property takes care of infinitely repeated patterns, while growing branches will be addressed later.

Definition 3

- *A proof tree is* unguarded *iff the* subject *of its root is equal to the subject of one of its subtree, or if it has an unguarded subtree.*
- *A term of a process algebra is* unguarded *if it has an unguarded proof tree, or if at least one of its possible reconfigurations is unguarded.*
- *A term of a process algebra is* guarded *if it is not unguarded.*

3.2 Non-growing operators

Growing operators build infinite structures from arguments having finite automata representations. Though no operators of usual process algebras have such a nasty behaviour, we need a syntactical way to ensure that no growing operator is used in a term. In order to obtain infinitely many states in the resulting automaton, one would have to introduce a rule that produces new terms ab infinitum. A natural sufficient condition for ensuring finiteness is to be able to find an order on expressions such that for all rules, the resulting process is not strictly greater than the subject.

It is possible to adapt here many results from the term rewrite system theory, with the difference that we are looking for a non-strict order compatible with our rewrite relation, whereas usual rewrite systems need a strict decreasing order. We give here a simple definition that covers nearly all interesting cases:

We consider families of operators closed under reconfiguration: if an operator belongs to the family, then all operators that occur in the resulting processes of all its rules also belong to the family.

Definition 4 *A family of operators* $\{Op_k\}$ *is* non-growing *iff there exist a simplification ordering* $\ll$ *such that:*

For each rule of each operator, let us denote $Op_k(x_i)$ *the subject of the rule and* $T_{i_j}[x'_i/x_i]$ *its resulting process in which all resulting working variables* x'_i *have been replaced by their corresponding* x_i*, then:*

$$either\ T_{i_j}[x'_i/x_i] \ll Op_k(x_i)$$

$$or\ T_{i_j}[x'_i/x_i] = Op_k(x_i)$$

where $=$ *is the syntactic equality on terms.*

We say then that all Op_k *are* $\ll$*-non-growing.*

Theorem 1 *Given a family of non-growing operators* $F = \{Op_k\}$*, an operator* $Op_k(x_i)$ *of arity* n*, and* n *terms* T_i *having TSF, then* $Op_k(T_i)$ *has TSF.*

Lemma 1 *Any process expression containing no recursive definition has a finite number of behaviour rules*

This follows trivially from our format for the operators' rules: building the behaviour proof trees, the premise at each node is strictly smaller than the subject of the node, w.r.t. the subterm order. So the depth of the proof tree is bounded by the depth of the expression. Then at each node you can choose among a finite number of rules to apply, the number of proof trees is finite, so the number of behaviours (= successful proof trees) is finite.

Proof of theorem 1:

Consider the set $\mathcal{R}$ of all possible reconfigurations R of $Op_i(T_i)$. The behaviours of each R are computed from instances of the rules of the operators $\{Op_k\}$. As simplification orderings are substitutive, the rules of each R are themselves compatible with the ordering. From lemma 1, each R have only a finite set of rules. Simplification orderings are well-founded, so $\mathcal{R}$ is finite.

Now say the subterms T_i have finite automata A_i. Then we can build the transition system of $Op_k(T_i)$, with states of the form $s =< R, st_{j_1}, ..., st_{j_n} >$, where the st_{j_i} are states from A_k, and initial state $< Op_k(T_i), st_{0_1}, ..., st_{0_n} >$. The number of those states is bounded by $|\mathcal{R}|*|A_1|*...*|A_n|$. The transitions of any such s are computed from the rules of R (finite from lemma 1), and from all combinations of the transitions of the st_{j_i}; they are in finite number.

■

This proof gives hints toward implementation: A pre-compilation phase can compute the reconfiguration set, and even build the functions that compute the composed transitions. Such a pre-compilation is subsumed in the algorithm in section 4.3.

3.3 Sieves

Definition 5 *A* sifting operator, *or* sieve, *is an operator with exactly one process argument, which rules obey the following conditions: each rule has exactly one working formal variable (the process argument), may have predicates and any form of resulting action, and the resulting process is obtained from the subject of the rule by substituting the working formal variable by the corresponding resulting process variable.*

This definition includes the *renaming*, *restriction*, and *ticking* operators of MEIJE, the *hiding* operator of TCSP and LOTOS.

Sieves are defined in fact as (partial) functions on actions, rather than on automata. More precisely they are functions that map actions to finite sets of actions, for a sieve may have a (finite) number of rules that can apply on a a given action. As such, they can be easily composed. They can also be combined with the automata building functions, leading to efficient implementations where no intermediate automata are built for such operators. This is especially important when the sieve discards some transitions: it is possible to save producing and exploring the corresponding target states.

Another issue is the introduction of sieves inside recursive definitions. We need define here a subclass of sieves such that the language generated by their compositions, modulo some idempotence property, remains finite. Then the states of the generated automaton will be obtained as pairs of a recursive variable and a composition of sieves.

This applies e.g. for alphabetical renamings, hiding, and restriction (for the alphabet of action labels in a term is finite, and the set of all restriction compositions is a finite commutative group).

Of course, it does not apply to ticking or to non-alphabetical renamings, and the MEIJE term: `let rec x= a:y and y = b*x in x` generates infinitely many states `x`, `b*x`, `b*b*x`, etc.

We need here a *non-growing* definition for resulting action functions:

Definition 6 *An operator rule is* action-non-growing *iff its resulting action is an action term built only from the following items: the formal action of the rule, the action parameters of the subject, the constant actions of the calculus, and alphabetical renamings (including renaming a label by an invisible action).*

This definition is trivially fulfilled by all operators of BASICLOTOS and CCS, but not by the ticking operator of MEIJE, nor by non-alphabetical renamings.

Definition 7 *A* non-growing sieve *is a sifting operator which rules are action-non-growing.*

Proposition 1 *Given a finite alphabet of actions, the algebra of all compositions of non-growing sieves has a finite model.*

Proof: Consider the *action-sifting* functions associated to sieves: Given an action (the label of a transition), an action-sifting function returns the (finite) set of possible transformations of the action by the various rules of the sieve that may be applied to this particular action.

It is enough finding a finite model for the compositions of action-to-action transformation: the extension to subsets will keep finiteness. By the way, the set of action-to-action transformations listed in definition 6 is closed under composition (u_p is the formal action of a sieve rule):

- $u_p \rightarrow u_p$ (the identity) is neutral element for the composition,
- $u_p \rightarrow a$ with a either an action parameter or constant of the calculus is trivially left absorbing for the composition,
- $u_p \rightarrow u_p < a_1/b_1, \ldots, a_n/b_n >$ where a_i and b_i are action labels (excluding composed actions) form a finite monoid for the composition.

■

3.4 Switching operators

The simplest way to generate an arbitrary finite automaton using recursive definition is to write an equation for each state of the automaton that describe the (finite) set of transitions, and of target states, of this particular state. This is achieved usually using recursive equations which right-hand-sides contain finite sums of action prefix operators (this was the case in the preceding version of AUTO).

For the sake of conciseness of the terms and of generality with respect to the now usual operators of process algebras, we want to extend as much as possible the possible language for these recursive equations, while keeping syntactic conditions on this language for TSF. Such extensions have been already published e.g. for Basic Lotos in [Ai86],[GN], allowing for example the following BasicLotos-like term, where ":" stands for the action prefixing operator, "[]" for the choice operator, and ">>" for the enabling operator:

Example 3

```
let rec x = a:exit >> y
    and y = b:y [] c:x   in x
```

Definition 8 *Given a family of operators $\mathcal{O}$ and a well founded simplification ordering $\ll$ on expressions generated from $\mathcal{O}$,*

An operator Op in $\mathcal{O}$ is a switching operator *(or simply a* switch*) w.r.t. one of its process arguments "p" iff:*

- *All rules in which "p" is a working formal variable verify the following properties: it has no other premise ("p" works alone) and the resulting process is exactly p'.*
- *All rules where the resulting process contains an occurrence of "p" are non-growing for $\ll$ and are their resulting processes are themselves switches for "p".*

Remarks:

- This includes non-growing operators with no premise at all.
- The name "*switch*" may refer to the fact that a switching rule selects (at most) one of its process arguments.
- This definition could be extended by allowing rules in which "p" is working to have as resulting processes T such that $T \ll Op(..., p', ...)$, with T being also in some sense a switch for p. However, this would complicate to much both the definition and the related proofs, whereas all classical operators fit the restricted definition we have just given.

The *sum* operator of Sccs, the binary *choice* of BasicLotos are switches, but also the *delay* operator of Meije and the *disabling* operator of Lotos for its second argument.

Usual prefixing operators are also switches, including the *action prefix* operators of Meije and Lotos, of course, but also the *enabling* operator of Lotos for its second process argument. The *internal choice* of Tcsp is a switch.

Yet the *external choice* operator of Tcsp is not a *switching* operator (see its rules in the annex), because for each of its arguments, it has a rule looking as a *switching* rule, and a rule resembling a *sieve* rule. By the way this operator is one we do not want to be involved in a recursive definition:

Example 4

```
let rec x = (tau : x) ext-choice a:y in x
```

This term generates the following sequence of resulting processes:

```
let rec {x = tau:x ext-choice a:x}
        in x ext-choice a:x
let rec {x = tau:x ext-choice a:x}
        in (x ext-choice a:x) ext-choice a:x
let rec {x = tau:x ext-choice a:x}
        in (x ext-choice a:x) ext-choice a:x ext-choice a:x
...
```

Though this specific case could be reduced (to a finite set of terms) by semantical arguments, the finiteness property may no more be guaranteed at a syntactical level. Semantical arguments for finiteness are out of the scope of this paper.

Definition 9 *Given a family of variables V, a term from a process algebra is called a* term suitable for recursion on V *iff either*

1. *it is a variable from V,*
2. *or it does not contain any variable from V and it has a finite automaton semantics,*
3. *or its head operator is a switching operator for some of its arguments, these arguments are subterms suitable for recursion on V, and all other arguments contains no occurrences of variables in V and have finite automaton semantics,*
4. *or its head operator is a non-growing sieve and its argument is suitable for recursion on V.*

Remark: this definition can be extended to handle nested recursive definitions, by adding an item for any recursive declaration "let rec $\{x_i = e_i\}$ in x_0" such that all e_i are suitable for recursion on $V \cup \{x_i\}$. Such an extension preserves the following theorem, though the proof is still more tedious.

Theorem 2 *Let Proc be a recursive definition* "let rec $\{x_i = e_i\}$ in x_0"*.*
If Proc is guarded, and if all expressions e_i are terms suitable for recursion on $\{x_i\}$, then the recursive definition has a finite automaton semantics.

Proof: We define a finite set of *states* by induction on the structure of the term, then we prove that the transition system of the term maps in this state space, with a finite number of transitions from each state. The full proof is in the annex.

This property allows us to guarantee that some recursive definition have a finite semantics. In any process algebra, it permits using any combination of nested recursive definitions, and arbitrary closed terms inside recursive definitions. In the case of MEIJE-SCCS, it naturally includes the classical "well-guarded" condition (sums of action-prefix operators). In the case of BASICLOTOS, it allows the occurrence of recursive variables as second arguments of the *enabling* operator and well-guarded occurrences of recursive variables within the second argument of the *disabling* operator.

3.5 Finite operators

This notion is used to ease the application of definition 9 for simple cases.

Definition 10 Finite *operators are non-growing operators with no process argument.*

Most process algebras have a constant operator without any possible behaviour (the *nil* of CCS, the *stop* of LOTOS). All these are finite operators. Another example is:

```
operator clock : Process
syntax h prefix
semantics
           ----------------------
             h(a) -- a --> h(a)
```

Finite operators are trivially accepted by case 1 of definition 9

3.6 Accessibility

All preceding conditions can be restricted to the *accessible* parts of the term. This is not only an optimization issue: considering only accessible parts allows for rejecting less programs, for any violations of a condition inside a non-accessible part of a term will have no consequence on its semantics.

Accessibility is an *undecidable* property, for it may involves deciding whether some subterms are dead-locked. We only want to give here a reasonable sufficient characterization of *potentially accessible* parts of a term. The main idea is that whenever some equation in a recursive definition is not referenced from an accessible part of the term, anything inside this equation has no influence at all on the semantics of the term. We define here by structural induction whether a sub-term, a recursive variable, or a recursive equation is accessible:

Definition 11

The root of a term is accessible.

If a subterm is accessible and this subterm has the form $Op(arg_i)$, and if the union of the sets of working formal variables in all the rules of operator Op is $\{arg_j\}$, then all the $\{arg_j\}$ are accessible (for all operators of classical process algebras, $\{arg_j\} = \{arg_i\}$).

If a recursive definition `let rec xi = ti in T` *is accessible, then all the variables* `xi` *that are free in* `T` *and in the accessible right-hand-sides* `ti` *are accessible, the corresponding recursive equations are accessible, and the right-hand-sides of these equations are accessible.*

If a local definition `let x = t in T` *is accessible, and if* `x` *is free in* `T`*, then the variable* `x` *is accessible, and the sub-term* `t` *is accessible.*

4 Algorithms

We have implemented a new version of AUTO using the preceding results. From the rules of the operators, and from their classification in finite, sifting, switching, and non-growing operators, we derive functions that test the syntactical conditions for finiteness, then build the automaton of a term.

More precisely, given a closed term of a given process algebra, the system goes through the following steps:

1. Computes the accessible part of the term. This is an opportunity to emit diagnostics on potential user errors.
2. Checks whether the accessible part is guarded. If not, rejects the term.
3. Checks whether all accessible recursive definitions are suitable for recursion. If not, rejects the term.
4. Generates the corresponding automaton. This step, as in the original AUTO system, is parameterized by a reduction function (hopefully a congruence) to be applied on each sub-automaton before building their combination.

4.1 Guarded recursion test

Within the framework of AUTO, the only possibility for building an unguarded proof tree is through a recursive declaration: there exist a path in any unguarded proof tree such that the subjects at both ends of the path are identical, and there is an unfolding of (at least) a recursive definition on this path, and all other nodes correspond to rules having at least one premise.

Let us test whether a term of is guarded or not. Informally, we look for a cycle through the immediate dependencies among recursive variables. As such a cycle may appear through several levels of nested recursive declarations, the dependency graph is to be computed for the whole accessible part of the term.

The algorithm builds a directed graph whose vertices are either recursive variables or subterms of the term, and the initial vertices are the accessible recursive variables. There is an arrow between states **T1** and **T2** if the head operator of **T1** has a rule with a premise for its n-th argument, and **T2** is the n-th son of **T1**; there is also an arrow between states **T1** and **T2** if **T1** is a recursive variable, and **T2** is the right-hand-side part of the equation defining **T1**, and between states **T1** and **T2** if **T1** is a recursive definition `let rec xi = ti in T2`

The term is unguarded iff there exists a cycle in this graph.

4.2 Automata Generators and Combinators

We describe now the algorithms used for the structural construction of automata.

We call *tta* (standing for *term to automaton*) the function computing an automaton from a term of a given process algebra. This function builds automata in a bottom-up fashion. A special case raises when a recursive definition is encountered: under the hypotheses of definition 9, the semantics of a recursive definition is a finite automaton (theorem 2). The algorithm *let-rec:tta* checks for the finiteness hypotheses and build the automaton. On all non-growing operators the algorithm *non-growing:tta* builds a resulting automaton from the automata corresponding to the operator parameters. The *tta* algorithms make also a special case for sieves: as the states of `tta`(<sieve>(p)) are included in those of `tta`(p), we propagate sieves as second argument of *tta* and use them each time a transition is generated. This implies that we can compose sieves and that we apply them as deep as possible inside the computations. Thus, *tta* takes two arguments:

- a term which is a non-growing operator applied on several subterms,
- a composed sieve (inherited from the recursive calls).

The basic *tta* algorithms are generic. They are adapted to each algebra through a specific set of functions derived from the operational semantics definition of the operators. The following sections describe the two algorithms *non-growing:tta* and *letrec:tta*, and the form of the specialized functions.

We use the object-oriented convention *type:name* for naming a generic algorithm and its specialized versions (called *methods*) depending on the type of its first argument.

4.3 The residual algorithm for any kind of non-growing operator

We use the same generic algorithm for all non-growing operator but for sieves. The algorithm for sieves is straightforward: $\mathit{sieve{:}tta}(\text{<sieve>}_1(p), \text{<sieve>}_2) = \mathit{tta}(p, \text{<sieve>}_1 \circ \text{<sieve>}_2)$

For other non-growing operators, we have to compose the argument automata. In a first step we recursively compute the automata denoted by all subterms. The most difficult point in this recursive application is building the composed sieve to give as second argument for each recursive call: if we want to optimize the size of the subsystems, we need to be able to compute how a composed sieve is modified when going through an operator. For the moment, we only treat simple cases such as the classical sum operator of Ccs for which we transmit the inherited sieve without modification. Other operators transmit the trivial identity sieve.

The second step of the algorithm is a classical residual exploration. Each state of the global system is a structure containing the states of each subsystem plus the operator combining them. The initial state of the global system is composed by the operator and the initial states of the subsystems. The unexplored state list is initialized to this initial state. Then the algorithm is a loop taking one state in the unexplored state list, computing all the transitions of this state using the specific functions associated to its operator and the sieve. This computation may produced some states that are not yet discovered and thus added to the unexplored state list. The computation finishes when the unexplored state list is empty. As we only use non-growing operators, the number of reconfigurations of a given operator is finite, and the termination of the algorithm stems from theorem 2.

More formally, we sketch in figure 1 the residual algorithm that computes `tta`($Op(t_1, \ldots, t_n$, <actions parameters>), <sieve>) where <sieve> is the inherited composed sieve, and t_i are the process arguments of the operator Op. In this figure, we call transition a couple of an action and a resulting state.

Algorithm for
`non-growing:tta`($Op(t_1, \ldots, t_n$, <actions parameters>), <sieve>)

$\forall i, T_i = tta(t_i,$ mk-sieve(Op, <sieve>, <actions parameters>))
$\text{state}_0 = \langle Op, \text{initial}(T_1), \ldots, \text{initial}(T_n)\rangle$
STATES $= \{\text{state}_0\}$
UNEXPLORED $= \{\text{state}_0\}$
TRANSITIONS $= \emptyset$
while (UNEXPLORED $\neq \emptyset$) loop
  take s in UNEXPLORED
  UNEXPLORED = UNEXPLORED $- \{s\}$
  transitions $= \emptyset$
  foreach $\langle$action, state$\rangle \in$ mk-trans(s, <actions parameters>) do
    action = <sieve>(action)
    if action then
      transitions = transitions $\cup \{\langle$action, state$\rangle\}$
      if state $\notin$ STATES then
        STATES = STATES $\cup$ {state}
        UNEXPLORED = UNEXPLORED $\cup$ {state}
  TRANSITIONS = TRANSITIONS $\cup \{\langle s,$ transitions$\rangle\}$
return $\langle$STATES, state_0, TRANSITIONS$\rangle$

Figure 1: The residual algorithm for non-growing operators

The methods *mk-trans* and *mk-sieve* are specific to each operator. For instance, in CCS the method `parallel:mk-sieve` for the parallel operator simply produces the identity sieve. The method `parallel:mk-trans` is sketched in figure 2:
Here the notation $< p'|q >$ stand for the composed state where the first sub-automaton has advanced by one step, and the second sub-automaton is still in its original state.

foreachmove $< p \xrightarrow{a} p' >$ means that we repeat the same group of operations for each transition of p.

4.4 The residual algorithm for recursive declaration

We sketch in figure 3 the algorithm dedicated to recursive declarations.

The *scan* methods uses the classification of operators in *variables*, *switches*, and *non-growing* operators. The method `variable:scan` fails, i.e. indicates that there is a non-guarded variable

Algorithm for `parallel:mk-trans(p|q,<actions parameters>)`

transitions = $\emptyset$

foreachmove $< p \xrightarrow{a} p' >$
  transitions = transitions $\cup < a, p'|q >$
  foreachmove $< q \xrightarrow{b} q' >$
  if a is inverse of b then transitions = transitions $\cup < \tau, p'|q' >$
foreachmove $< q \xrightarrow{b} q' >$
    transitions = transitions $\cup < b, p|q' >$
return transitions

Figure 2: `parallel:mk-trans`

Algorithm for `let-rec:tta`(let rec $\{x_i = t_i\}_{i=1,n}$ in x_k, <sieve>)

Equations = $\{\langle x_i, t_i, \text{<sieve>}\rangle\}_{i=1,n}$
$\text{state}_0 = \langle x_k, t_k, \text{<sieve>}\rangle$
STATES = $\{\text{state}_0\}$
UNEXPLORED = $\{\text{state}_0\}$
TRANSITIONS = $\emptyset$
while (UNEXPLORED $\neq \emptyset$) loop
  take $s = \langle x, t, \text{<sieve>}\rangle$ in UNEXPLORED
  UNEXPLORED = UNEXPLORED $- \{s\}$
  transition = $\emptyset$
  scan(t, <sieve>)
  TRANSITIONS = TRANSITIONS $\cup \{\langle s, \text{transitions}\rangle\}$
return $\langle$STATES, state_0, TRANSITIONS$\rangle$

Figure 3: The residual algorithm for recursive definitions

in the term. The methods `<sieve>:scan` recursively call `scan` on its process argument after modifying its sieve argument. Last, the methods `<switch>:scan` call recursively `scan` on each allowed subterm, producing some transitions, leading to some subterms: when these subterms do not correspond to some variable in the set Equations, the function `check` generates new equations to be added to the set. The function `add-transition` add a transition to the list *transitions*, and add the new states to the STATES and UNEXPLORED lists. Figure 4 sketches the *scan* method for some simple switches and for sieves.

5 Conclusion

The AUTO system we currently distribute is using specific hand-coded algorithms for the operators of the MEIJE0 calculus (stop, prefixing, sum, parallel, restriction, renaming). These algorithms were carefully optimized in order to avoid building parts of product automata that were to be deleted by some restriction operators.

We have built a new prototype of the AUTO system using the generic algorithms of this paper. Tests have been made both for the MEIJE0 calculus and for BASICLOTOS (the prototype has been presented in [MV89]). The MEIJE0 operators are correctly classified by our definitions: the prototype accepts stricty more MEIJE0 programs than the preceeding AUTO system. Some other MEIJE-SCCS operators (see [dSi85]) can be added easily to this syntax, including *ticking*,

operator	parameters	`scan` algorithm

`sequence`	(<action> : p, <sieve>)	action = <sieve>(action)
		if action then
		  state = check(p, <sieve>)
		  add-transition(action, state)

`sum`	($p + q$, <sieve>)	`scan`(p, <sieve>) `scan`(q, <sieve>)

Sccs `delay`	($delay(p)$, <sieve>)	action = <sieve>(tau-action)
		if action then
		  state = check(p, <sieve>)
		  add-transition(action, state)

<sieve>	(<sieve>$_1(p)$, <sieve>$_2$)	`scan`(p, <sieve>$_2 \circ$ <sieve>$_1$)

Figure 4: `scan algorithms`

interleaving and the *synchronized product* as non-growing operators. The results are good also for BasicLotos operators: the usual finiteness conditions are correctly deduced from the rules. Some limitations of our conditions are listed in the annex. We also obtained efficiency mesurements: this version appears to have the same order of performances than the old version. Moreover, it should be clear that in many cases it allows to build a smaller number of automata (for sieves never require to copy an automaton) and to apply sieves on smaller automata. No optimizations have been done in the first prototype, so the new version is potentially much more efficient than the specialized Meije0 version.

The set of programs accepted by the generic version is of course larger than in the former system, and programs may be written in a much more permissive way: for example parallel compositions may be done in many different ways using various operators and nested recursive definitions are allowed.

The congruence properties of some equivalences versus Meije composition operators are also to be generalized. It is very important for space efficiency reasons to apply reductions as deep as possible in the term, in order to create and compose smaller automata. We plan to have the Ecrins system proving congruence laws for various equivalences and various operators, so that the congruence properties can be automatically used in Auto during the automata construction.

6 Annexes

Not all the operators mentionned in this paper are widely known. Let us give the rules of some of them.

In [Bo85] is defined the Meije-Sccs calculus, with operators from both Boudol's Meije calculus and Milner's Sccs calculus. The operators *hiding*, *ticking*, *clock* that occur in the paper are taken from the Meije-Sccs calculus. Let us give other examples:

```
operator delta:: Process --> Process
syntax delta
semantics
  delayed        ---------------------------
                 delta(p) -- tau --> delta(p)
  undelayed              p -- a --> p'
                     --------------------
                     delta(p) -- a --> p'
end
operator DELTA:: Process --> Process
syntax DELTA
semantics
  DELTA_1                   p -- a --> p'
                   ----------------------------------
                   DELTA(p) -- a --> delta (DELTA(p'))
end
operator desynchro:: Process --> Process
syntax desynchro
semantics
  desynchronised        ------------------------------------
                        desynchro(p) -- tau --> desynchro(p)
  undesynchronised              p -- a --> p'
                        ------------------------------------
                        desynchro(p) -- a --> desynchro(p')
end
```

The *delta* operator is a switch.

The *DELTA* operator does not match our conditions for non-growing operators, though it could be managed by had-hoc methods.

The *desynchro* operator is a degenerated form of sieve, that does not match our conditions. An extension of the sieves definition towards this type of operators would be possible.

The *external-choice* of TCSP, as mentionned in section 8 have both switching and sifting rules. This prevent us from using it in recursive definitions:

```
operator external-choice :: Process Process --> Process
syntax ext-choice left 6
semantics
p -- a -->p' & (not (a equal tau))      p -- a -->p' & (a equal tau)
---------------------------------   ------------------------------------
    p ext-choice q -- a -->p'       p ext-choice q -- a --> p' ext-choice q
q -- a -->q' & (not (a equal tau))      q -- a -->q' & (a equal tau)
---------------------------------   ------------------------------------
    p ext-choice q -- a -->q'       p ext-choice q -- a --> p ext-choice q'
end
```

Proof of theorem 2: We define a finite set of *states* by induction on the structure of the term, then we prove that the transition system of the term maps in this state space, with a finite number of transitions from each state.

Let us call *state* a pair $St =< P, S >$ where P is a process expression and S is a (composition of) sieve.

We build recursively the set $\mathcal{S}$ of states by:

St1: The *initial state* $St_0 =< e_0, I >$, where I is the identity sieve, is in $\mathcal{S}$.

St2: Given a state $< P, S >$ in $\mathcal{S}$, with $P = x_i$, then $< e_i, S >$ is in $\mathcal{S}$.

St3: Given a state $< P, S >$ in $\mathcal{S}$, with P having TSF, then for all states st of the automaton of P, extending $\ll$ such that st is smaller than all operators, then $< st, S >$ is in $\mathcal{S}$,

St4: Given a state $< P, S >$ in $\mathcal{S}$, with $P = Op(p_i, q_j)$, Op is a switching operator for the x_i arguments corresponding to the p_i, and the q_j have TSF, let $\{r_i\}$ be the rules in which the x_i are working, $\{r_j\}$ the other rules of Op, $\{rp_i\}$ and $\{rq_j\}$ their respective resulting processes, then:

St4.1: for all r_i, $< p_i, S >$ is in $\mathcal{S}$,

St4.2: for each r_j, call q_g the subset of $\{q_j\}$ working in r_j, introduce new constants c_{g_k} of type process, one for each state of each automaton of q_g, stating that these new constants are smaller than all operators w.r.t. $\ll$, then for all terms $q' \ll Op(p_i, q_j)[c_{g_k}/q_g]$ for some combination of the c_{g_k}, $< q', S >$ is in $\mathcal{S}$ (if the rule is an axiom, then take $q' \ll Op(p_i, q_j)$).

St5: Given a state $< P, S >$ in $\mathcal{S}$, with $P =$ <sieve>(P'), then $St' =< P', S \circ$ <sieve> $>$ is in $\mathcal{S}$.

The set of process expressions in $\mathcal{S}$ is finite, because they all are $\ll$ to one of the e_i (cases 1 and 2 introduce some e_i, cases 3 and 4 introduce expressions $\ll$ to some expression in $\mathcal{S}$, case 5 introduces subterms of expressions in $\mathcal{S}$, and $\ll$ is compatible with the subterm ordering). The set of sieves compositions in $\mathcal{S}$ is finite by proposition 1. So $\mathcal{S}$ is finite.

Remark that any process expression can be written $s_1(\ldots(s_k(P))\ldots)$, where all s_i are sieves, and the head operator of P is not a sieve; let us write this decomposition $S(P)$.

Now we prove that all reconfigurations of $Proc$ decompose into a $S(P)$ such as $< P, S >$ is in $\mathcal{S}$, and have finitely many transitions. This will achieve the proof.

Let us call subreconfigurations of $Proc$ the smaller set R of process expressions containing e_0, closed under *switching argument* relation (if $Op(p_i, q_j) \in \mathcal{R}$ and Op is a switch for the p_i, then the p_i are in $\mathcal{R}$), and closed under the transition relation (if $P' \in \mathcal{R}$ and $P' \xrightarrow{a} P''$ then $P'' \in \mathcal{R}$).

Let us prove by induction that:

$$P' \in \mathcal{R} \Longrightarrow P' \text{ is suitable for recursion on } \{x_i\} \text{ and } P' = P^*(S') \text{ with } < P^*, S' > \in \mathcal{S}$$

Ind0: e_0 verifies our property: it is suitable for recursion by hypotheses of theorem 2, and its decomposition is in $\mathcal{S}$, using construction St1, together with item 5 as many times as the head operator of e_0 is a sieve.

Ind1: For an operator Op that is a switch for p_i, the subterm corresponding to p_i is suitable for recursion by definition 9.3, and from $Op \in \mathcal{R}$ we get $< Op, I > \in \mathcal{S}$, so $< p_i, I > \in \mathcal{S}$ by St4.

Ind2: Consider a transition $P' \xrightarrow{a} P''$, with with $P' \in \mathcal{R}$. By induction hypotheses, P' is suitable for recursion, and its decomposition $< P^*, S' >$ is in $\mathcal{S}$. Trivially, P^* is suitable for recursion. Let us show that $P'' = P^{**}(S'')$ By case analysis on definition 9:

Ind2.1: $P^* = x_i$: then its transitions are those of e_i, that is suitable for recursion, and $< e_i, S' >$ is in $\mathcal{S}$ by St2 leading to the conclusion by application of the induction step on $< e_1, S' >$ (there may be no loop in the proof here, because of the *guarded* hypotheses).

Ind2.2: P^* is closed w.r.t $\{x_i\}$, and has TSF: all its reconfigurations are similarly suitable for recursion (by definition 9.2), and they map to the $< st_i, S' >$ states introduced by ST3.

Ind2.3: $P^* = Op(p_i, q_j)$ where Op is a switch for the x_i arguments corresponding to the p_j; its transitions are built:

Ind2.3.1: either using a rule with one of the p_i working; the resulting process P^{**} is got from a transition of p_i itself, so it verifies our property (apply the induction step on $< p_i, S' >$),

Ind2.3.2: or using a rule in which no p_i are working; by the definition of suitability, all q_j have TSF. Consider the product automaton of the q_j working in this rule (if the rule is an axiom, this is a one state blocked automaton); each state of this compound automaton correspond to a reconfiguration of $Op(p_i, q_j)$ that may involve some of the p_i, but these are unchanged. Those states map to some of the q' of St4.2. Their transitions are those of the compound automaton, plus some new transitions using r_i rules, leading to a p_i' reconfiguration. In the first case, $< q', S' >$ is in $\mathcal{S}$, and we keep suitability (for all subsequent reconfigurations are still switches for the p_i). In the second case, as in Ind2.3.1, we keep our property.

Ind2.4: P^* = <sieve>(p) is not possible by definition of the decomposition.

■

References

[Ai86] G. Ailloux, "*Verification in Ecrins of Lotos Programs*", ESPRIT/SEDOS/C2/N45 (1986)

[BB88] T. Bolognesi, E. Brinksma, "Introduction to the ISO Specification Language LOTOS", in *The Formal Description Technique LOTOS*, North-Holland, 1988

[BK86] J.A. Bergstra, J.W. Klop, "Process Algebra: Specification and Verification in Bisimulation Semantics", *CWI Monographs*, North-Holland, 1986

[BS87] T. Bolognesi, S. A. Smolka, "Fundamental Results for the Verification of Observational Equivalence: a Survey", proc. of the IFIP 7^{th} Internaional Symposium on Protocol Specification, Testing, and Verification, North-Holland, 1987

[Bo85] G. Boudol, "Notes on Algebraic Calculi of Processes", *Logics and Models of Concurrent Systems*, NATO ASI series F13, K.Apt ed., 1985

[Bro83] S. Brookes, "*A Model for Communicating Sequential Processes*", PhD Thesis, University of Oxford, 1983

[dSi85] R. De Simone, "Higher-Level Synchronising Devices in Meije-Sccs", *Theoretical Computer Science* 37, p245-267, 1985

[dSV89] R. De Simone, D. Vergamini, *"Aboard AUTO"*, Technical Report INRIA RT111, 1989

[GN] H. Garavel, E. Najm, "TILT: From LOTOS to Labelled Transition Systems", in *The Formal Description Technique LOTOS*, North-Holland, 1988

[MV89] E. Madelaine, D. Vergamini, "AUTO, a verification tool for distributed systems using reduction of automata", in proceedings of *Forte'89 conference*, Vancouver, North-holland, 1989

[MdSV89] E. Madelaine, R. de Simone et D. Vergamini, *"ECRINS, A Proof Laboratory for Process Calculi, User Manual"*, to appear INRIA, 1990

[Mi80] R. Milner, *"A Calculus for Communicating Systems"*, Lectures Notes in Comput. Sci. 92, 1980

[Mi83] R. Milner, "Calculi for Synchrony and Asynchrony", *Theoretical Computer Science* 25, p267-310, 1983

[RS89] V. Roy, R. De Simone, "*AUTO - AUTOGRAPH*, submitted to Workshop on Computer-Aided Verification, Princeton, N.J., 1990

[VG88] F.W. Vaandrager, J.F. Groote, "Structured operational semantics and bisimulation as acongruence" CWI report CS-R8845, 1988

[Ve89] D. Vergamini, "Verification of Distributed Systems: an Experiment", in *Formal Properties of Finite Automata and Applications*, LNCS 386, 1990