# Finiteness conditions and structural construction of automata for all process algebras

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

DIDIER VERGAMINI
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.

# 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 ([DMdS90]) 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 ([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

[1]This work was partially supported by ESPRIT BRA (n°3006) CONCUR.

[2]The full version of this paper is published in the AMS-DIMACS volume of the Computer Aided Verification Workshop proceedings, R.P.Kurshan ed., 1990

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.

Similar conditions have been given by other authors for other languagaes, e.g. by D. Taubner for CCS, by H. Garavel for Lotos, and by M. Barbeau for Lotos also, but in the case of 1-place Petri-nets models, that may represent strictly more programs than finite automata.

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. Of course the finiteness conditions rely deeply on the format allowed for the operator rules.

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], 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 $\tau$. 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 $\delta$.

## 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. Here is an example taken from Basic-Lotos:

```
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 ([DMdS90]):

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

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:*

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

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

*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.*

*All proofs are in the full paper.*

## 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.

The introduction of sieves is two-folded:

- They are an interesting family of automata transformers, acting only on transitions. As such, they can be easily composed and combined with the automata building functions, leading to efficient implementations where no intermediate automata are built for such operators, and only accessble parts are considered.
- They may be used also inside recursive definitions. We need define here a sub-class 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.*

## 3.4 Switching operators

We introduce the family of *switching operators* as a generalization of the usual sum operator of CCS/MEIJE.

**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.
- 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 3**

```
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
...
```

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.

## 3.5 Main result

**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.*

*Hint :* 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 proof is in the full paper.

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.

## 3.6 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.

We give in the full paper a sufficient characterization of *potentially accessible* parts of a term, based on the analysis of recursive variable occurrences in the recursive definitions.

# 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. Due to size constraints, we cannot give here a full description of the generated algorithms. The main ideas are:

- A given term of a process algebra is first checked for guardeness and suitability of its accessible part. Terms that do not satisfy the syntactic conditions are rejected.
- The automaton is built structurally in a bottom-up manner, starting with the recursive definitions at the leaves. There are two main algorithms, one dealing with recursive definitions, and the other one combining several arguments-automata w.r.t. a given operator. Both algorithms take a (composition of) sieve and a reduction congruence as additional parameters.
- The algorithm dealing with a recursive definition maintains a set of states that are pairs of a recursive variable and a sieve, each state associated to a subterm containing recursive variable occurrences. The analysis of such a term generates transitions towards states that have to be compared to already existing states, or added to the set. The finiteness conditions ensure the termination of this computation.
- The composition algorithm is a residual algorithm that traverses both arguments combining their transitions in a manner that depends on the combination operator.

# 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*, *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.

This prototype in its BASICLOTOS version is currently integrated in the LOTOS tool environment of the ESPRIT project LOTOSPHERE.

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.

# References

[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

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

[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

[DMdS90] G.Doumenc, E. Madelaine, R. de Simone, *"Proving Process Calculi Translations in ECRINS"*, Technical Repport INRIA RR1192, 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"*, this volume.

[VG88] F.W. Vaandrager, J.F. Groote, "Structured operational semantics and bisimulation as a congruence" 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