A Front-End Generator for Verification Tools *

Rance Cleaveland † Eric Madelaine ‡ Steve Sims †

April 28, 1995

Abstract

This paper describes the Process Algebra Compiler (PAC), a front-end generator for process-algebra-based verification tools. Given descriptions of a process algebra's concrete and abstract syntax and semantics as structural operational rules, the PAC produces syntactic routines and functions for computing the semantics of programs in the algebra. Using this tool greatly simplifies the task of adapting verification tools to the analysis of systems described in different languages; it may therefore be used to achieve source-level compatibility between different verification tools. Although the initial verification tools targeted by the PAC are MAUTO and the Concurrency Workbench, the structure of the PAC caters for the support of other tools as well.

1 Introduction

The past ten years have seen the development of a variety of automatic verification tools for finite-state systems expressed in process algebra; examples include MAUTO [6], the Concurrency Workbench [10], TAV [14], and Aldébaran [11]. In general, these tools support a specific language, such as CCS [19], Meije [1], or Basic Lotos [5], for describing systems and provide users different methods, such as equivalence checking, preorder checking, model checking, random simulation, and abstraction mechanisms, for analyzing their behavior. The utility of these tools has been demonstrated via several case studies [7, 18]. However, the impact on system design practice of such tools has been limited by the fact that the languages they support, while possessing nice theoretical properties, are not widely used by system engineers. In addition, as each tool in general supports a different language, it is difficult to compare the tools and to investigate approaches to using them in collaboration with one another.

This paper presents the Process Algebra Compiler (PAC), a system that substantially simplifies the task of changing the language supported by verification tools. The PAC is a "front-end generator"; given a description of the syntax and semantics of a language, it produces routines for parsing and unparsing programs in the language and for computing user-defined semantic relations. By providing users with high-level notations for defining languages and managing the difficult and

*This work is partially funded by NSF-INRIA collaboration # CCR-9247478, ESPRIT Basic Research Action CONCUR2, NSF/DARPA grant CCR-9014775, ONR Young Investigator Award N00014-92-J-1582, and NSF Young Investigator Award CCR-9257963.

†rance@csc.ncsu.edu, gr-sts@galois.csc.ncsu.edu, Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206, USA, (919) 515-7862.

‡Eric.Madelaine@sophia.inria.fr, INRIA, B.P. 93, 06902 Sophia Antipolis Cedex, France, (+33) 93 65 78 07.

technically tedious development of syntactic and semantic functions, the PAC provides the research community with a useful tool for expanding the repertoire of languages their tools can support.

The remainder of the paper is organized along the following lines. The next section sharpens the motivation for the PAC by presenting two verification tools, MAUTO and the Concurrency Workbench (CWB) , and the common semantic framework underlying the (different) languages each supports. The section following presents an overview of the architecture of the PAC and describes the specification language used for defining algebras and their semantics, while Section 4 discusses issues in generating semantic functions from their PAC specifications. Section 5 then gives experimental results obtained from PAC-produced front ends for the Concurrency Workbench; somewhat surprisingly, the PAC-generated code significantly outperforms existing hand-produced code built for this tool. The final section contains our conclusions and directions for future work.

2 Verification Tools and Structural Operational Semantics

This section presents an overview of two verification tools, MAUTO and the Concurrency Workbench. Although similar in intent, the tools differ markedly in terms of the analyses they support, and yet at the moment there is no way for a user to use the tools collaboratively. On the other hand, the languages supported by the two tools have a semantics that is given in a very similar style, which we also discuss at the end of this section. These observations provided the impetus for the development of the PAC.

2.1 Verification tools

Both MAUTO and the Concurrency Workbench provide utilities for verifying finite-state systems expressed in process algebra. The specific process algebras supported differ, however, as do the supported analyses. The following provides more detail about the systems.

MAUTO. MAUTO is a system for analyzing networks of finite-state systems. MAUTO builds automata from programs in the Meije process algebra and is capable of reducing and comparing them with respect to various bisimulation-based equivalences. It also provides a novel facility that enables users to define *abstract* transition relations on a given automaton, obtaining a new system, usually smaller and more tractable, that highlights specific behaviors of the original system. Much attention has been devoted to issues of efficiency. In particular, the building of automata from terms is mixed with the reduction of the automata using congruence properties of the semantic equivalences, thereby ensuring that automata are kept as small as possible. Facilities are also provided for explaining the results of analysis and for drawing the resulting automata in a graphical editor [21].

The Concurrency Workbench. The Concurrency Workbench (CWB) is an extensible tool for verifying systems written in the process algebra CCS . In contrast with other process algebra tools, the CWB supports the computation of numerous different semantic equivalences and preorders; it does so in a modular fashion in that generic equivalence- and preorder-checking routines are combined with suitable process transformations (see, e.g., [8]) in order to compute different relations. The CWB also includes a flexible *model-checking* facility for determining when a process satisfies a formula in a very expressive temporal logic, the propositional mu-calculus. Recently the CWB has been extended to deal with a discrete-time version of CCS (TCCS), and with the synchronous algebra SCCS.

2.2 Structural Operational Semantics

MAUTO and the CWB are similar in that they analyze systems by converting them into finite automata and then invoking routines on these automata. However, the languages and forms of analysis they support, and the approaches they take to construct automata from systems, differ markedly. In the last case in particular, MAUTO adopts a "bottom-up" approach, with automata recursively constructed for subsystems and then assembled into a single machine for the entire system. The CWB, on the other hand, uses an "on-the-fly" approach, with transitions of components calculated and then combined appropriately into transitions for the over-all system.

One characteristic shared by MAUTO and the CWB, however, is that the languages they support have operational semantics given in the Structural Operational Semantic (SOS) [20] style; this fact motivates our inclusion in the PAC of capabilities for generating routines from SOS descriptions. A SOS for a language consists of rules for inferring the execution behavior of programs written in the language. Rules have the following general form.

$$\frac{premises}{conclusion}(side\ condition)$$

The intuitive reading of the rule is that if one is able to establish the premises, which typically involve statements about the execution behavior of subprograms of the one mentioned in the conclusion, and the side condition holds, then one may infer the conclusion. As an example, the following describes the synchronizations allowed by the parallel composition operation in CCS.

$$\frac{p \xrightarrow{a} p' \quad q \xrightarrow{b} q'}{p|q \xrightarrow{\tau} p'|q'}(a, b\ inverses)$$

The rule states that if p can engage in an action a and evolve to p' and q can engage in b and evolve to q', and a and b are inverses (i.e. constitute an input/output pair on the same communication channel), then $p|q$ can execute an internal action, τ, corresponding to the synchronized execution of a and b.

The SOS style has evolved in many ways since the Plotkin's seminal paper [20] and has been applied to many areas of language semantics. The SOS style is very flexible, as numerous languages with widely varying features have been given semantics using this framework. Recent work has focused on the metatheory of SOS [4, 17, 12, 2, 22]; in particular, researchers have shown that when SOS rules conform to different syntactic formats, the resulting languages have nice properties. In the area of process algebras, one important property is congruence of the behavioral equivalences with respect to the operators of the language; this allows one to reason about the system components in a compositional way.

3 Using the PAC

This section provides an overview of the PAC architecture and indicates how users specify process algebras for processing by the PAC.

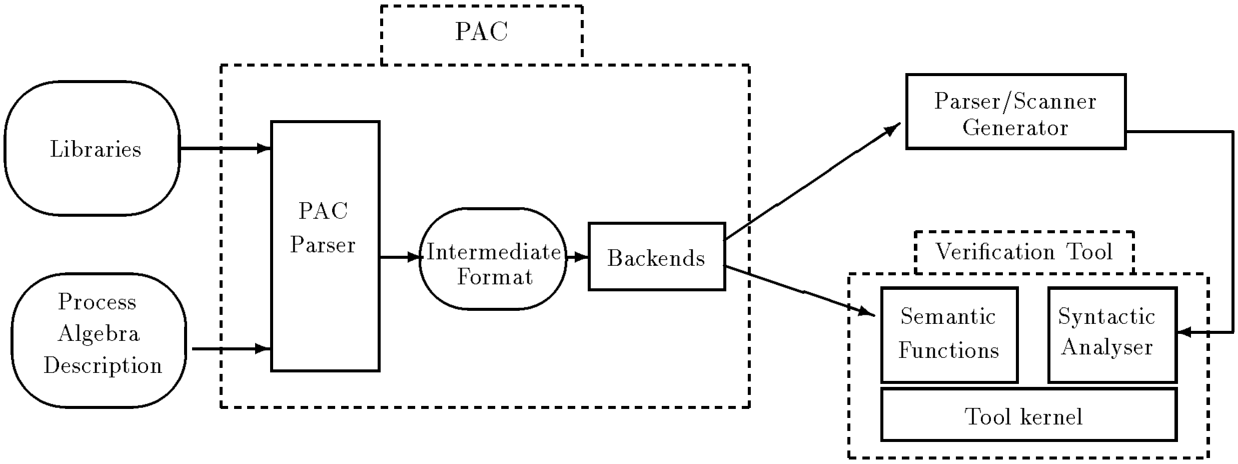

Figure 1: Architecture of the PAC

3.1 PAC Overview

Figure 1 sketches the organization of the PAC. The system takes as input files containing the syntactic and semantic description of a process algebra as well as libraries containing the definitions of any necessary auxiliary functions. It then produces two (sets of) files:

- A YACC/LEX[1] specification of the language's syntax.
- Semantic routines to analyze programs written in the language.

To specialize a target verification tool to the given language, the PAC user must run YACC/LEX on the first set of files to produce a parser and then insert the parser and the semantic routines into the verification engine. Provided that the target tool separates the syntactic analysis of programs from their verification, the language-independent part of the verification tool (its *kernel*) need not change at all. It should be noted that the PAC is in fact a "compiler": it takes a PAC specification of a language as input and produces source code which is compiled along with the kernel of the target verification tool. There is **no** PAC run-time system that becomes part of the target verification tool. The PAC can also be viewed as a "compiler compiler" since the generated code is a "compiler" which accepts a process algebra program as input and produces a labeled transition system as output.

The PAC itself is organized into several components centered around an internal representation of the syntax and semantics of the language being processed. This internal structure is produced by the PAC parser from files provided by the user. From it, *back ends* produce the required routines for the target systems. As target verification tools are typically written in different languages (MAUTO in Lisp, the Concurrency Workbench in SML, Aldébaran in C, for example), there will in general be several back ends in the PAC. The initially targeted systems are MAUTO and the Workbench; accordingly, the existing prototype includes back ends that generate Lisp and SML, respectively.

[1] Using YACC/LEX provides an easy way of guaranteeing the compatibility of parsers generated for a given algebra by different back ends. Other parser-generators may also be used at the discretion of the back-end writer.

The PAC parser. The PAC parser tries to factor out as much of the back-end-independent work as possible from the processing of user-supplied algebra descriptions; in particular, it checks the PAC input for syntactic correctness and performs certain consistency checks. If user files satisfy these criteria, the parser then produces an intermediate representation of the input which contains:

- A representation of the abstract syntax of the process algebra.
- A structured description of the concrete syntax from which specifications for scanners, parsers, and unparsers may be generated.
- A representation of the sets of SOS rules used for defining the semantics of the operators in the algebra.

Back ends. The back ends build the actual routines to be included in the verification tools. They accept as input the intermediate format generated by the PAC parser and generate as output a YACC grammar together with routines that compute the semantics of a system from its abstract syntax. The routines typically differ from one verification tool to another; typical examples include those for computing the single-step transitions of a process, generating the composition of several automata by a given composition operator, computing sufficient syntactic conditions for a process to be finite-state [17], and calculating whether or not a process is divergent.

Implementation. The PAC is implemented in Standard ML (SML). The system, which currently consists of roughly 15,000 lines of code, is batch-oriented; it processes inputs and either generates output files or reports error messages.

3.2 PAC Process Algebra Specifications

A PAC process algebra specification consists of two components. The `ALGEBRA` module contains descriptions of the concrete and abstract syntax of actions, processes and semantic relations. The `RULE_SET` modules contain SOS rules defining the relations used to define the semantics of processes. Users may also provide *library* files containing code that directly implements auxiliary structures (such as sets or environments) and operations (such as set membership or lookup functions) used in defining the semantics and for which users do not wish to provide SOS definitions. Back ends directly insert this code into the files they generate; consequently this code must be written in a language compatible with that of the target tool. The remainder of this section discusses each of these modules using as an example the CWB-6.0 version of Milner's CCS .

3.2.1 `ALGEBRA` Modules

`ALGEBRA` modules consist of several sections.

- In the `sorts` section, users define the syntactic categories for their language.
- The `cons` section defines the term constructors (i.e. the abstract syntax) to be used to build elements in the different syntactic categories.
- The `funs` sections introduces the names and "types" of functions that may be applied to elements of sorts. The implementations of these functions must be supplied by the PAC user.

```
ALGEBRA CCS
sorts
    id, act, id_set, restriction, ('a eqn), agent, ('a frame), ('a env), ...
cons
    Tau         : act
    Input       : id -> act
    Output      : id -> act
    ...
    Res_set     : id_set -> restriction
    Res_var     : id -> restriction
    Eqn         : id * 'a -> ('a eqn)
    ...
    Nil         : agent
    Bottom      : agent
    Ag_var      : id    -> agent
    Prefix      : act * agent -> agent
    Plus        : agent * agent -> agent
    Restriction : agent * restriction -> agent
    Fix         : agent * (agent frame) -> agent
    ...
funs
    id_parse    : string -> id
    id_eq       : id * id -> bool
    ...
    inverses    : act * act -> bool
    ...
    mk_id_set   : (id list) -> id_set
    member      : id * id_set -> bool
    ...
    mk_frame    : (('a eqn) list) -> ('a frame)
    mk_frame_inv: ('a frame) -> (('a eqn) list)
    empty       : 'a env
    push_frame  : ('a frame) * ('a env) -> ('a env)
    ...
rels
    transition  : (agent env) * (id_set env) * agent * act * agent -> bool
    diverges    : (agent env) * (id_set env) * agent -> bool
inputs
    transition is [1,2,3]
    diverges is [1,2,3]
pragmas
    ...
    CWB "parser entries:  act, agent, id_set"
    ...
SYNTAX
    ...
RULE_SYNTAX
    ...
end
```

Figure 2: An `ALGEBRA` module for CCS

- The `rels` section defines the names and "types" of the semantic relations to be defined by SOS rules and for which the PAC will generate an implementation. The `inputs` section then indicates what type the generated functions computing these relations should have (i.e. which positions in the relation should be "inputs" and which should be "outputs").
- The `pragmas` section contains back-end-specific directives, such as locations of library files and names to be assigned to functions generated by back ends.
- The `SYNTAX` and `RULE_SYNTAX` sections contain descriptions of the concrete syntax of both the process algebra and of the relations used to define the algebra's semantics.

To illustrate what appears in these sections, consider the (elided) version of an `ALGEBRA` module for CCS in Figure 2. The `sorts` section declares the kinds of objects that appear in the definition of the algebra, including `act` (actions), `agent` (CCS processes), `'a eqn` (equations), and `'a frame` (frames, or mappings from identifiers to values). Note that sorts may be polymorphic: in the case

```
SYNTAX
tokens
        "nil"     => NIL
        "where"   => WHERE
        "and"     => AND
        "end"     => END
        ...
priorities
        ...
nonterminals
        agent of agent
        ag_eqn_list of (agent eqn) list
        ...
grammar
        agent : NIL                         (Nil())
              | agent WHERE agent_frame END   (Fix(agent, agent_frame))
              ...
        agent_frame : ag_eqn_list            (mk_frame(ag_eqn_list))
lists
        ag_eqn_list is non_empty_list EMPTY_STR COMMA AND EMPTY_STR of ag_eqn
        ...
RULE_SYNTAX
        ...
grammar
        relation : agent_env COMMA id_set_env COLON agent DASHDASH act ARROW agent
                   (transition (agent_env, id_set_env, agent1, act, agent2))
...
```

Figure 3: A `SYNTAX` and `RULE_SYNTAX` section for CCS

of `'a frame`, for instance, the `'a` may be instantiated with any well-formed sort. The PAC also includes three built-in sorts: `string` for character strings, `bool` for booleans, and `'a list` for polymorphic lists.

The next section of the example introduces the constructors used in CCS and their sorts. For example, `Tau` is introduced as a constructor taking no arguments and producing a value of sort `act`; that is, `Tau` is an action constant. `Input` and `Output` take an identifier (intuitively, a channel name) as an argument and produce an action. In the CWB version of CCS, users may bind identifiers to sets of actions and then use these identifiers in place of sets in the restriction operator. To cater for this possibility, the algebra introduces a sort `restriction` and two constructors, `Res_set` and `Res_var`, permitting sets of identifiers (i.e. a label set, in CCS terms) or a single identifier (a variable name bound to a set) to be viewed as "restrictions". `Eqn` is used to construct equations from identifiers and values, while the remaining constructors are used to build agents. Note that the `Fix` operator takes an agent and an agent frame as arguments; intuitively, the frame contains bindings for the free variables that may appear in the agent.

The `funs` section introduces operations that may be applied to elements of given sorts. These operations differ from constructors in that the PAC will generate implementations for the latter but not for the former; users must provide routines for these. This feature permits users to re-use existing code and to program efficient implementations of low-level data structures as appropriate.[2] Thus, to generate a CWB front end on the basis of the example algebra module a user would neet to provide implementations in Standard ML (the language in which the CWB is written) for operations such as `id_parse`, `mk_id_set` and `mk_frame`.

[2]PAC back ends also generate implementations of sorts having constructors declared for them; it relies on users to specify implementations of sorts for which no constructors have been specified. For example, `act` would have a PAC-supplied implementation, since three constructors have `act` as their return sort. The sort `'a frame`, on the other hand, does not have constructors defined for it; consequently, a user must supply code defining the data structure to be used to represent frames.

The `rels` component of the example introduces two semantic relations: `transition` and `diverges`. In this version of CCS, the transitions and potential for divergence of an agent depend on two environments: one to resolve free agent variables, and one to resolve free variables used in restrictions. Thus each relation includes an `agent environment` and an `id_set environment` argument.

For each relation the `inputs` section indicates the form the PAC-generated function for computing this relation should take. In the case of `transition`, for example, the input specification indicates that the generated function should have three inputs corresponding to the first three positions of the relation (here, two environment arguments and an agent). Given such a triple, the function will return the set of all action-agent pairs which, when combined with the triple, yield a quintuple in the relation. In the case of `diverges`, all places are mentioned in the input list; in this case, the PAC will generate a function taking three arguments and returning a boolean.

The `pragmas` section includes miscellaneous directives for specific back ends. In the above example, the given pragma indicates that the parser produced by the PAC back end for the CWB should have entries for agents, actions and identifier sets. These are needed since the CWB supports commands requiring users to provide information from these sorts. Other pragmas might be used to rename sorts appropriately (some tools might require a type `proc` rather than `agent`, for example) or supply names of library files.

Samples of the syntax sections of the CCS algebra specification appear in Figure 3. The `SYNTAX` component contains information needed to generate the parsers to be used by the target tool; this currently takes the form of a YACC-like grammar whose semantic actions consists of "sort-correct" expressions built using the constructors and functions declared previously. In the example, the syntax of the fixpoint agent operator is defined to be a `where` e_1 `and` ... `and` e_n `end`, where each e_i is an equation. Note that PAC grammars extend YACC grammars by permitting list specifications; the nonterminal `ag_eqn_list`, for example, yields lists of `ag_eqn` (agent equations) whose beginning and ending delimiters are the empty string and whose separator is the token AND (`and` in concrete syntax). The `RULE_SYNTAX` section enriches this syntactic specification with information needed to parse the SOS rules that define the semantics of processes; in particular, it includes definitions of the concrete syntax of relations. This example defines the syntax of the `transition` relation to be ae`,` se `:` p `--`a`-->` q. The PAC fits this information into a general "rule template" in order to produce a grammar which is processed and then used to parse the user-supplied rules.

3.2.2 `RULE_SET` Modules

The second part of a PAC process algebra specification consists of the SOS rules needed to define the semantics of processes. In general, a user must supply a collection of rules for each relation introduced in the `ALGEBRA` module. Each rule in turn consists of four components: a name, a list of *premises*, a *side condition*, and a *conclusion*. In general, premises and conclusions involve relations, while side conditions can be any expression generated using the following grammar,

$$be ::= \texttt{true} \mid \texttt{not}\ be \mid be\ \texttt{and}\ be \mid be\ \texttt{or}\ be \mid P(t_1, ..., t_n)$$

where P is a predicate: any boolean-sorted function declared in the `funs` section or any relation in the `rels` section all of whose positions are input positions. The t_i should be terms in the appropriate sort, based on the definition of P. A fragment of the rules for the `transition` relation

```
RULE_SET transition
vars
    a, b                    : act
    p, p', p1, p1', p2, p2' : agent
    s                       : id_set
    ae                      : (agent env)
    se                      : (id_set env)
    ...
rules
    prefix
                  ------------------------(true)
                   ae, se : a.p -- a --> p
    ...
    parallel_1
                        ae, se : p1 -- a --> p1'
                  -----------------------------------(true)
                   ae, se : p1 | p2 -- a --> p1' | p2
    parallel_2
                        ae, se : p2 -- a --> p2'
                  -----------------------------------(true)
                   ae, se : p1 | p2 -- a --> p1 | p2'
    parallel_3
                   ae, se : p1 -- a --> p1', ae, se : p2 -- b --> p2'
                  ---------------------------------------------------(inverses(a,b))
                          ae, se : p1 | p2 -- t --> p1' | p2'
    ...
end
```

Figure 4: A `RULE_SET` module for the CCS `transition` relation

for CCS appears in Figure 4. Note that premises appear above, and conclusions below, a line of hyphens, with the side condition appearing in parentheses after the hyphens. All expressions in the rules are written using the concrete syntax declared in the `ALGEBRA` module; this enables the rules to look very close to what appears in the literature. In addition, functions defined in the `ALGEBRA` module may be used in the rules; for example, `parallel_3` contains a reference to the `inverses` predicate, which intuitively should hold when the given actions represent an input and output on the same channel (note that `t` is the concrete syntax that has been defined for the CCS internal action). Rule sets can refer to relations defined in other rule sets, although no such reference is made in this example.

4 PAC Back Ends

The PAC currently includes back ends dedicated to the CWB and to MAUTO. The former generates code in SML, the language in which the CWB is written, while the latter, which is still under construction, produces LeLisp, the programming language in which MAUTO is implemented. In each case, the produced code contains a parser, some unparsing functions, and a number of semantic functions encoding the SOS rules of the algebra. The parsers (generated using respectively LeLisp-Yacc and SML-Yacc) are fully compatible, meaning that PAC-generated front ends for MAUTO and the CWB handle the same syntax. As Section 2 indicated, however, the analysis functions of the target tools are different, as are the semantic functions. The following discusses what semantic functions the different back ends must produce and how they are generated from SOS specifications.

4.1 The CWB Back End

In addition to various parsing and unparsing routines, the CWB requires that its front end include implementations of types `act` and `agent` and functions `transition`, `diverges` and `sort`. The functions each take an agent and return a set of action-agent pairs, a boolean, and a set of actions, respectively. This section describes how the CWB back end generates code from the SOS definitions of semantic relations. Generally speaking, given a rule set for a particular semantic relation, the technique constructs a function whose inputs correspond to the places in the relation declared as inputs in the `inputs` section of the `ALGEBRA` module. On a given input, the generated routine produces a set of tuples as outputs; the idea is that each output tuple, when combined with the input tuple, yields an element in the relation. As an example, in the case of the `transition` relation defined in Section 3.2, positions 1, 2 and 3 of `transition` are declared as inputs; the procedure that is produced will therefore accept an $\langle \texttt{agent env}, \texttt{id\_set env}, \texttt{agent} \rangle$ triple as input and produces a set of $\langle \texttt{action}, \texttt{agent} \rangle$ pairs as output with the property that if the input is $\langle ae, se, p \rangle$, then pair $\langle a, q \rangle$ is in the set of outputs if and only if $\langle ae, se, p, a, q \rangle$ is in relation `transition`.

In order for the procedure described below to work, the rules used to define semantic relations must obey certain syntactic restrictions. Recall from Sections 2.2 and 3.2.2 that SOS rules have the following general form:

$$\frac{premises}{conclusion}(side\ condition)$$

where *conclusion* is an element of the relation being defined, *premises* is a list of elements of the relation being defined or of other relations declared in the `rels` section, and *side condition* is a boolean expression that may involve predicate expressions of the form $P(t_1, \ldots, t_n)$, with the t_i being terms that may involve variables. For the code produced by the CWB back end to compile, each rule must satisfy the following constraints.

1. All variables appearing in the input positions of the premises must appear in the input positions of the conclusion.
2. All variables appearing in the output positions of the conclusion or in the side condition must appear either in the input positions of the conclusion or in the output positions of a premise.
3. All variables appearing in the input positions of the conclusion or the output positions of a premise are distinct.

These constraints place restrictions on the "flow of data" through a rule: information flows from the inputs of the conclusion to the premises, and the outputs of premises flow (together with inputs of the conclusion) to the side condition and the outputs of the conclusion. Note also that patterns of arbitrary depth can appear in the input or output positions of the conclusion or premises. It should be noted that this rule format subsumes the positive GSOS format of [4] while being incomparable to the tyft/tyxt pattern of [12] and the path scheme of [2]. However, restriction 1 can be relaxed without too much difficulty to allow variables appearing in the output positions of premises to appear in the input positions of other premises; with this generalization, our format would subsume pure and well-founded tyft/tyxt and path. Other formats allow negative premises [4, 22] and are incomparable to ours.

The basic strategy used by the code generated from rules involves pattern matching: given a tuple of inputs, a generated function in essence determines which rules have conclusions whose input positions match the input tuple. Using the premises of these rules, appropriate (recursive) calls are issued, and the results which satisfy the side condition are combined into a set of result tuples using the form of the conclusion. To illustrate this idea, consider the rules given for CCS in Section 3.2.2, and suppose that the generated semantic function is given an input of the form

$\langle \texttt{ae}, \texttt{se}, p|q\rangle$. In this case, three rules are applicable— `parallel_1`, `parallel_2` and `parallel_3`. Each of the rules mentions the transitions of p or q in the premises; consequently, the generated code would include recursive calls to calculate the transitions for $\langle \texttt{ae}, \texttt{se}, p\rangle$ and $\langle \texttt{ae}, \texttt{se}, q\rangle$. On the basis of the first rule, transitions of p would be combined appropriately with q, while the second rule would transform transitions of q by combining them with p. The final rule combines transitions of the form $\langle a, p'\rangle$ and $\langle b, q'\rangle$ into $\langle \tau, p'|q'\rangle$ *provided that* the predicate $\texttt{inverses}(a, b)$ is satisfied. The results of these combinations are collected into one set and returned.

To improve the performance of the generated code, the CWB back end also employs several optimizations. For example, in order to minimize matching overhead, rules with the same input pattern in their conclusions are grouped and processed simultaneously. Also, the generated routines cache results of recursive calls in a hash table; before issuing a recursive call, this table is consulted to determine if the call has been made before. To demonstrate the savings from this technique, consider how the CWB would compile the agent $p|q$ into a labeled transition system. First, a call is made to the generated CCS transition function with $\langle ae, se, p|q\rangle$ as input (ae and se are the current agent and set environments). After making the recursive calls to compute the transitions of p and q, the generated `transition` function saves the results of these calls in a table. From *par_rule1*, it follows that $p|q$ has a transition $\langle a, p'|q\rangle$ for every transition $\langle a, p'\rangle$ of p. The next step in compiling the labeled transition system for $p|q$ will include computing the transitions of each $p'|q$; but, in this case instead of making recursive calls to recompute the transitions of q, `transition` would simply look this up in the transitions table. This strategy leads to significant time savings when computing the finite-state representation of a system; somewhat surprisingly, it can also lead to substantial space savings as well, since sharing becomes possible in the computation of output tuples.

4.2 The MAUTO Back End

MAUTO uses a "compositional" (bottom-up) approach to building automata; language constructs are interpreted as automaton transformations, and thus it is not in general possible to use directly the transition function described above. The same SOS rules that yield the *transition* function for the CWB are interpreted as specifying these automata transformers. Thus, the semantic functions generated by the PAC for MAUTO encode *transition system transducers* in the sense of Larsen and Xinxin [15]. Computing the automaton for $p|q$, for example, involves computing separately the finite automata describing the full behaviors of p and q, eventually reducing each of them according to any congruence at hand, then combining them using the transducer for parallel composition. In practice, the rule format required by this interpretation is more restrictive than the one in the preceding section: it ensures that the transducer generated for any context expression is finitely represented, and that the combination of finite automata always yields a finite global automaton. The structure of the functions produced for MAUTO is also very different from the structure of those produced for the CWB. In general, there are two functions for each operator, one describing the recursive structure of the bottom-up traversal, and one describing how to combine the transitions of a tuple of argument automata.

A static analysis of the structure of the SOS rules of the transition relation allows us to classify the process operators in the algebra. This is used to produce optimized automata-constructing routines, to ensure the finiteness of the produced transducers, and to guarantee *a priori* the termination of automata construction. The classification is a generalization of the notions defined in [17], and distinguishes between:

- *Combinators*, which are typically operators used for parallel composition. The format ensures that they do not generate infinite transition systems from finite arguments.
- *Switches*, which have only one process argument active at a time, and will eventually select one of them (sum, sequence). They are used for defining static conditions for finiteness of recursive definitions.
- *Sieves*, which have exactly one process argument and act as action transformers, keeping their structure unchanged (hiding, restriction, relabeling). Identifying sieves enables various run-time optimizations to be employed that avoid some intermediate automaton constructions.

We have produced code using these ideas that has proved to be very efficient and flexible, and easy to integrate with other compositional approaches.

5 Results

The current prototype of the PAC includes an algebra description parser and a back end for the CWB, with the development of the MAUTO back end in progress. In this section we describe our experience with using the PAC to generate front ends for the CWB. The experiments take two forms. In the first, we compare the efficiency of a PAC-generated front end for CCS with existing hand-coded front ends for CCS, while in the second we investigate the performance of front ends produced by the PAC for other languages. Our initial results suggest that the PAC does indeed ease the task of changing the language supported by the CWB and that the generated interfaces perform well. Our tests used a version of the Concurrency Workbench under development at North Carolina State University and were run on a Sun Sparc 5 with 128 megabytes of RAM. The functionality of this version of the CWB is similar to the Edinburgh CWB [10], but the NCSU version includes more efficient graph-construction and equivalence-checking routines.

Table 1 compares the performance of a PAC-generated front end with two hand-coded front ends for CCS. The first of these is the front end included with Version 6.0 of the CWB, while the second is a hand-tuned version of the first one developed at NCSU. The numbers describe the amount of processor time in seconds (time needed for system activities and for garbage collection have been omitted) needed by the NCSU CWB to build finite-state automata from different CCS sample programs using the transitions function supplied by the given interface. The example programs we used to test the interfaces included:

- Two communications protocols: an implementation of the Alternating Bit Protocol (ABP) and an implementation of part of the data link control layer of IEEE 802.2 (802-2).
- Two solutions of the two-process critical-section problem: an implementation of Dekker's algorithm (Dekker-2) and an implementation using semaphores (Semaphore-2).
- Milner's Jobshop example [19] (Jobshop).
- A specification of the Edinburgh mail system (Mail-system).

In addition, we tried some examples consisting of the parallel composition of these examples in order to assess the performance of the front ends on systems with large state spaces. In the table these examples have the form $\text{System}_1 | \text{System}_2$. As the table indicates, the PAC-generated

Example	*Number of states*	*Interface* CWB 6.0	NCSU CWB	PAC-generated
ABP	57	0.12	0.13	0.14
Jobshop	77	0.14	0.14	0.12
Dekker-2	127	0.38	0.35	0.39
802-2	331	1.67	1.33	1.83
Semaphore-2	468	2.66	2.44	2.25
Mail-system	1616	9.12	8.68	7.59
ABP | Jobshop	4389	18.82	13.76	10.73
Dekker-2 | Semaphore-2	59436	522.82	288.19	101.88
ABP | Mail-system	92112	Ran out of memory	Ran out of memory	340.90
Dekker-2 | Mail-system	205232	Ran out of memory	Ran out of memory	779.03

Table 1: This table shows the program time in seconds required by the different interfaces to construct automata for various CCS examples.

CCS interface actually performs substantially better than existing CCS interfaces while using less memory; the main reason for this lies in the caching of recursive calls outlined in Section 4.1.

We have also used the PAC to generate CWB front ends for several other languages as well. Examples have included a simple language of regular expressions and a version of CCS in which actions take priority [9]; the latter is noteworthy in that its semantic account requires the use of auxiliary semantic relations. In general, the amount of effort required has been much less than what would be required to generate interfaces by hand, although more experience with the tool is necessary to substantiate this claim. However, the fact that the notations the tool provides for expressing semantic and syntactic specifications of languages are more abstract than those provided by standard programming languages lead us to believe that the PAC will greatly simplify the production of front ends for verification tools.

Our most involved example has been the generation of a CWB front end for Basic Lotos, which is more complex, both syntactically and semantically, than the others we have tried. We have analyzed a number of Basic Lotos examples with the generated interface. Since no Basic Lotos interface existed previously for the CWB it is harder to evaluate the efficiency of the generated code than it was in the case of CCS. One crude measure, however involves comparing the states generated per unit time from LOTOS programs against a similar figure for the CCS front ends described previously. The states-per-second measures for the CCS front ends were computed from the first eight examples in the table above (the ones that all interfaces were able to handle), while the figure for the Basic Lotos interface was calculated based on timing results from the compilation of 8 examples ranging in size from 20 states to 45,000 states. The results are shown in Table 2, which shows that the front end generated for Basic Lotos is roughly 8 times slower than the one generated for CCS. This difference is not necessarily due to the inadequacy of the code-generating scheme used by the PAC, but rather arises from the fact that Basic Lotos is syntactically and semantically more complex than CCS.

	Interface			
	CWB 6.0	NCSU CWB	PAC-generated CCS	PAC-generated Basic Lotos
States per second	119.66	211.10	532.31	65.47

Table 2: Average number of states generated per second for four different interfaces.

6 Conclusions

In this paper we have presented the Process Algebra Compiler, a tool for generating front ends for verification tools. The PAC allows users to specify the syntax and semantics of a language they wish their verification tool to support; the system then produces the syntactic and semantic routines needed to specialize the given tool for the language. Experimental results indicate that PAC-generated routines exhibit performance that can in fact improve on that of hand-coded routines.

Regarding future work, our most immediate goal is to complete the MAUTO back end so that it and the CWB may become source-level compatible. Back ends for other verification tools could also be built and our experiences in building the CWB and MAUTO back ends would certainly ease this task. We anticipate that this will be possible for most tools based on transition system semantics, although some reorganization of the target tools may be necessary.

We would also like to investigate the addition of features in the PAC specification language. In particular, the lexical specifications supported by the PAC can be made more flexible, and providing some facility for modularity in the algebra section would be desirable. We have experimented with the latter; defining concrete syntax in a modular way, however, appears to be very difficult. We have also experimented with a less flexible, but much easier to use, format for expressing concrete syntax and plan to study this issue more. We have also worked on and would like to further investigate routines in the PAC for analyzing a rule set and reporting to the user whether it satisfies a given rule format, such as those mentioned in Section 2.2.

We also would like to explore the possibility of using the PAC for activities other than generating front ends for verification tools. Given the widespread use of SOS rules for defining the semantics of languages, it might be possible to use the PAC to automatically generate interpreters and compilers. We are also examining the feasibility of using the PAC as an implementation engine for generating on-the-fly verification routines, as these may often be formulated using SOS-style rules [3]. Obviously, these uses are greatly different from the PAC's initial purpose, and it remains to be seen if they are indeed practical.

Related Work. Other verification tools have also aimed at providing some parametricity with respect to the language analyzed. The ECRINS system [16] permitted users to specify the SOS semantics of their algebra, and to prove algebraic laws of their operators. MAUTO allows users to extend the syntax of the language it supports, although semantic routines must be altered by hand. As a compiler for syntactic and semantic specifications, the PAC is closely related to the CENTAUR system, and in particular to its semantic component TYPOL [13]. TYPOL provides a general framework for defining languages, interpreters, and compilers, using SOS rules. The more restrictive PAC rule format allows for the generation of simpler and more efficient code.

References

[1] D. Austry and G. Boudol. Algèbre de processus et synchronisation. *Theoretical Computer Science*, 30:91–131, 1984.

[2] J.C.M. Baeten and C. Verhoef. A congruence theorem for structured operational semantics with predicates. Technical Report 93/05, Eindhoven University of Technology, 1994.

[3] G. Bhat, R. Cleaveland, and O. Grumberg. Efficient on-the-fly model checking for CTL*. In *Tenth Annual Symposium on Logic in Computer Science (LICS '95)*, San Diego, July 1995. IEEE Computer Society Press.

[4] B. Bloom, S. Istrail, and A. Meyer. Bisimulation can't be traced. In *Fifteenth Annual ACM Symposium on Principles of Programming Languages (PoPL '88)*, pages 229–239, San Diego, January 1988. IEEE Computer Society Press.

[5] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In P.H.J.van Eijk, C.A.Vissers, and M.Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–76. North-Holland, 1989.

[6] G. Boudol, V. Roy, R. de Simone, and D. Vergamini. Process calculi, from theory to practice: Verification tools. Rapport de Recherche RR1098, INRIA, October 1989.

[7] R. Cleaveland. Analyzing concurrent systems using the Concurrency Workbench. In P.E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *Lecture Notes in Computer Science*, pages 129–144. Springer-Verlag, 1993.

[8] R. Cleaveland and M.C.B. Hennessy. Testing equivalence as a bisimulation equivalence. In *Proceedings of the Workshop on Automatic Verification Methods for Finite-State Systems*, pages 11–23. Springer-Verlag, 1989.

[9] R. Cleaveland and M.C.B. Hennessy. Priorities in process algebra. *Information and Computation*, 87(1/2):58–77, July/August 1990.

[10] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics-based tool for the verification of finite-state systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[11] J.C. Fernandez. Aldébaran: A tool for verification of communicating processes. Technical Report Spectre-c 14, LGI-IMAG, Grenoble, 1989.

[12] J.F. Groote and F. Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 2(100):202–260, 1992.

[13] G. Kahn. Natural semantics. Technical Report RR601, INRIA, 1987.

[14] K.G. Larsen, J.C. Godskesen, and M. Zeeberg. TAV, tools for automatic verification, user manual. Technical Report R 89-19, Dep^t of Mathematics and Computer Science, Ålborg university, 1989.

[15] K.G. Larsen and L. Xinxin. Compositionality through an operational semantics of contexts. In M.S. Paterson, editor, *Automata, Languages and Programming (ICALP '90)*, volume 443 of *Lecture Notes in Computer Science*, pages 526–539, Warwick, England, July 1990. Springer-Verlag.

[16] E. Madelaine, R. de Simone, and D. Vergamini. *ECRINS*, user manual, 1988. Technical Documentation.

[17] E. Madelaine and D. Vergamini. Finiteness conditions and structural construction of automata for all process algebras. In R. Kurshan, editor, *proceedings of Workshop on Computer Aided Verification*, New-Brunswick, June 1990. AMS-DIMACS.

[18] E. Madelaine and D. Vergamini. Specification and verification of a sliding window protocol in LOTOS. In K. R. Parker and G. A. Rose, editors, *Formal Description Techniques, IV*, volume C-2 of *IFIP Transactions*, Sydney, December 1991. North-Holland.

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

[20] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, September 1981.

[21] V. Roy and R. de Simone. Auto and autograph. In R. Kurshan, editor, *proceedings of Workshop on Computer Aided Verification*, New-Brunswick, June 1990. AMS-DIMACS.

[22] C. Verhoef. A congruence theorem for structured operational semantics with predicates and negative premises. In B. Jonsson and J. Parrow, editors, *Proceedings CONCUR 94*, Uppsala, Sweden, volume 836 of *Lectures Notes in Computer Science*, pages 433–448. Springer-Verlag, 1994.