

The Process Algebra Compiler

Version 1.0

November, 1999

User's Manual

Steve Sims

Reactive Systems, Inc.

www.reactive-systems.com

sims@reactive-systems.com

Copyright ©1999 by North Carolina State University and Reactive Systems, Inc.

Contents

1	Introduction	3
1.1	Revision History	3
1.2	Feedback and Bug Reports	4
1.3	Acknowledgments	4
2	Overview	4
2.1	Labeled Transition Systems	4
2.1.1	Actions	4
2.1.2	Definition	4
2.1.3	Example	5
2.2	Structural Operational Semantics	5
2.2.1	Abstract Syntax	6
2.2.2	Inductive Systems	7
2.2.3	Rule-Based Inductive Definitions	7
2.2.4	Defining the Transition Relation	8
2.2.5	Building Labeled Transition Systems	8
2.3	Retargeting the CWB-NC	9
2.4	PAC Architecture	11
2.5	Implementation	12
3	PAC Syntax Specifications	13
3.1	PAC Identifiers and Keywords.	13
3.2	Structure of the Syntactic Description	13
3.3	Abstract Syntax	13
3.3.1	Sorts	13
3.3.2	Cons	14
3.3.3	Funs	15
3.4	Semantic Relation Declarations	17
3.4.1	Rels	17
3.4.2	Inputs	17
3.5	Pragmas	18
3.5.1	<code>user files</code>	18
3.5.2	<code>parser entries</code>	18
3.5.3	<code>build_keyword_table</code>	18
3.5.4	<code>unparser entries</code>	18
3.5.5	<code>unparser info</code>	20
3.5.6	<code>sharing constraints</code>	20
3.5.7	<code>comments</code>	20
3.5.8	<code>sos coments</code>	21
3.5.9	<code>cache</code>	21
3.5.10	<code>naming convention</code>	21
3.6	Concrete Syntax	22
3.6.1	Tokens	22
3.6.2	Priorities	24
3.6.3	Nonterminals	24
3.6.4	Grammar	27

3.6.5	Lists	27
3.7	rules syntax Differences	28
4	PAC Semantic Specifications	31
4.1	Structure of a <code>RULE_SET</code> Module	31
4.2	Structure of an Inference Rule	31
4.3	Restrictions on the Variables in Rules	32
4.4	A <code>RULE_SET</code> Module for CCS	33
5	The Grungy Details	35
5.1	Obtaining the PAC	35
5.2	Installing the PAC	35
5.3	Running the PAC	36
5.4	Files Produced by the PAC	36
5.5	Compiling and Using the Generated Files	38
5.5.1	Building a New Version of the CWB-NC	38
6	Hints and Common Errors	39

1 Introduction

This document describes how to use the Process Algebra Compiler (PAC), a tool that eases the task of changing the design language accepted by the Concurrency Workbench for the New Century (CWB-NC) [5, 4], a process-algebra-based verification tool. The CWB-NC includes efficient implementations of decision procedures for computing a number of different behavioral equivalences and preorders between systems and for determining whether systems satisfy formulas written in an expressive temporal logic, the modal mu-calculus. The tool also includes a general-purpose reachability analysis routine. All design-language specific routines are encapsulated within a single module making it possible to change the language supported by the tool. The CWB-NC's modular design makes its analysis routines independent of the specific input language that the tool accepts. Therefore, changing the input language that the tool accepts is feasible; however, doing so is a tedious and time-consuming task. The PAC greatly simplifies this job. Figure 1 shows that, given a high-level description of the syntax and semantics of a design language, the PAC generates Standard ML source code implementing a front end that allows the CWB-NC to analyze systems specified in the new language.

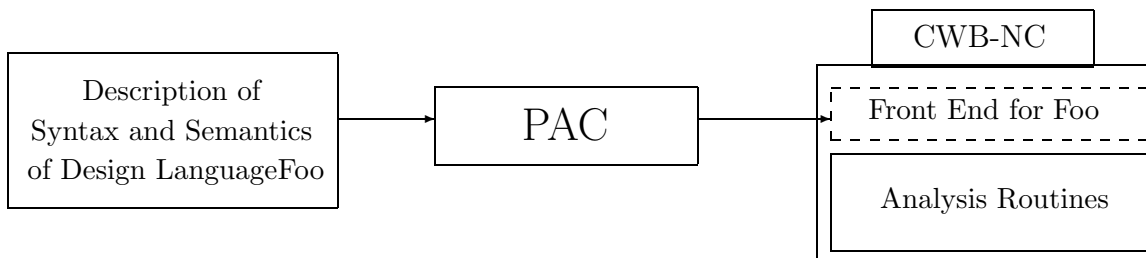


Figure 1: Using the PAC to generate a new language front end for the CWB-NC.

A PAC specification for a language includes a description of the syntax of the language (given as a Yacc-like grammar) and a description of the semantics of the language (given as sets of Plotkin-style SOS rules [8]). High-level, formal notations such as context free grammars and regular expressions have proved to be ideal input formats for parser and scanner generators. The PAC extends this notion by using SOS rules as input for generating routines for computing various user-defined semantic relations.

Although the initial purpose of the PAC was to change the specification languages that verification tools accept, other uses may be possible. For example, we would like to investigate the possibility of using the PAC to develop interpreters and compilers. The syntactic component of the PAC has also been used without the semantic component as a simple way to generate scanners, parsers, and unparsers. The most obvious advantage over using Lex and Yacc is that unparsers are also generated from the grammar given to specify the parser.

1.1 Revision History

Version 1.0 The original public release of the PAC was labeled Version 1.0 and occurred in November of 1999.

1.2 Feedback and Bug Reports

Please send comments and bug reports to pac@reactive-systems.com. All feedback is greatly appreciated.

1.3 Acknowledgments

Most of the effort to build the PAC occurred while the author was a student in the Department of Computer Science at North Carolina State University. The research was directed by Professor Rance Cleaveland. The Office of Naval Research and the National Science Foundation provided funding for the research for which we are grateful. Thanks also to Dr. Eric Madelaine (INRIA, France) for his collaboration on the project and for hosting us in his home during several grueling research visits to the south of France which, despite some pesky distractions (e.g. Cannes Film Festival, hikes in the Alps, caving), were very productive. We are also indebted to a number of early PAC users who gave valuable feedback. Some of these brave souls include: Joel Gray, Glenn Bruns, Marco Bernardo, Gerald Lüttgen, Girish Bhat, Alex Groce, and Samalam Arun-Kumar (a.k.a. SAK).

2 Overview

This section provides an overview of the PAC. We begin with the formal basis for the PAC by describing *labeled transition systems*, a mathematical model for system behavior, and *structural operational semantics*, a technique for defining the semantics of design languages. Do not worry if you do not follow all the technical details. Just as you can use a parser generator without completely understanding the theory of context free languages, it is possible to use the PAC without mastering the theory presented here. But, of course understanding the theory will make using the tool easier, so we recommend at least scanning this section. We go on to explain what the PAC must generate to retarget the CWB-NC to a new language and describe the input that a user must provide to the PAC in order to generate this new language interface. Finally, we give a high-level description of the structure of the PAC and conclude the section with some brief implementation details.

2.1 Labeled Transition Systems

A common theme in engineering is that to understand the behavior of a complex system, one builds a model of the system and studies the behavior of the model. This is the approach taken in the CWB-NC. One common mathematical model for concurrent systems are *labeled transition systems*, which resemble non-deterministic finite-state automata.

2.1.1 Actions

The building blocks for a labeled transition system (LTS) come from a set *Act* representing the set of actions the modeled system may perform. Typically, the actions correspond to communications offered by the system to its environment. We require that *Act* contains a distinguished action τ that models unobservable, internal computation.

2.1.2 Definition

A labeled transition system models the behavior of a concurrent system as follows. The nodes of the LTS correspond to the various *states* that the concurrent system may enter during execution.

An edge in the LTS from state s_1 to state s_2 and labeled by the action a indicates that when the system modeled by the LTS is in state s_1 it may perform an action a and evolve to state s_2 . One state in the LTS is designated as the start state indicating that this is the state the system is in when execution begins.

Definition 2.1 (Labeled Transition Systems (LTS)) *A labeled transition system is a triple $\langle S, s_0, \longrightarrow \rangle$ where*

- S is a set of states.
- s_0 is the start state.
- $\longrightarrow \subseteq S \times Act \times S$ is a transition relation.

We denote the set of all labeled transition systems as \mathcal{LTS} . When $\langle s, a, s' \rangle \in \longrightarrow$ we write $s \xrightarrow{a} s'$. We extend the transition relation to sequences of actions in the natural way by defining $\longrightarrow^* \subseteq S \times Act^* \times S$ as the least set containing:

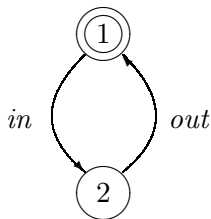
- $\{\langle s, [], s \rangle \mid s \in S\}$
- $\{\langle s, a :: t, s'' \rangle \mid s \xrightarrow{a} s' \wedge s' \xrightarrow{t}^* s''\}$

When no confusion arises, we write \longrightarrow for \longrightarrow^* .

2.1.3 Example

As an example consider the LTS for a one place buffer:

$$\langle \{1, 2\}, 1, \{\langle 1, in, 2 \rangle, \langle 2, out, 1 \rangle\} \rangle$$



This trivial system begins execution in state 1, may perform an input action (*in*) and evolve to state 2, after which it may perform an output action (*out*) and return to state 1. Note that the system does not terminate, but rather offers ongoing interaction with its environment.

2.2 Structural Operational Semantics

As with programming languages, a distinction is made between the syntax and semantics of a design language. The syntax of a language specifies which streams of characters constitute a valid term in the language, whereas the semantics of the language explains what a valid term in the language means. Structural Operational Semantics (SOS), introduced by Plotkin [8], is a particular approach to defining the semantics of languages. The name of this technique includes two adjectives *structural* and *operational*. The latter indicates that this style of defining semantics describes how a term in

the language executes and the former implies that this is inferred from the structure of the term (i.e. a term's behavior depends on the behavior of its sub-terms).

For a particular design language L , let $Proc_L$ be the set of design language terms denoting processes and let Act_L be the set of actions processes may perform. An SOS describes the execution of terms in L by inductively defining a *transition relation* $\longrightarrow_L \subseteq Proc_L \times Act_L \times Proc_L$ that relates a process to the single step transitions it is capable of performing. That is, if $\langle p, a, p' \rangle \in \longrightarrow_L$, written $p \xrightarrow{a}_L p'$, then process p is capable of performing the action a and subsequently behaving as p' . The rest of this section formalizes the technique for defining \longrightarrow_L .

2.2.1 Abstract Syntax

The transition relation is defined over the abstract syntax of terms in a language. Abstract syntax is given as closed terms over a *signature* as defined below. Intuitively terms are built up from a set of constructors and the abstract syntax of a term may be viewed as a tree whose nodes are labeled with constructors. The tree for a term t has a root node labeled with the top-level constructor in t and the children of the root node correspond to the top-level sub-terms of t .

Definition 2.2 (Signatures and Terms) *A many-sorted signature Σ is a pair $\langle S, C \rangle$ where S is a set of sorts and C is a set of constructors. A constructor is a triple $\langle \text{name}, \text{domain}, \text{codomain} \rangle$ where name is an element of a set of constructor names N , domain $\in S^*$, and codomain $\in S$. Let X be an S -sorted set of variables and $X \cap N = \emptyset$. We denote the set of variables in X having sort s as X_s . For every $s \in S$ the terms of sort s with free variables from X , denoted $T(\Sigma, X)_s$, is the least set containing:*

- all $x \in X_s$ (i.e. variables of sort s)
- all $c = \langle n, \epsilon, s \rangle \in C$ (i.e. nullary constructors of sort s)
- all $c(t_1, \dots, t_j)$ where $c = \langle n, (s_1, \dots, s_j), s \rangle \in C$ and $\forall i : 1 \leq i \leq j : t_i \in T(\Sigma, X)_{s_i}$.

The set of closed terms of sort s is denoted $T(\Sigma)_s$ and is the least set containing:

- all $c = \langle n, \epsilon, s \rangle \in C$ (i.e. nullary constructors of sort s)
- all $c(t_1, \dots, t_j)$ where $c = \langle n, (s_1, \dots, s_j), s \rangle \in C$ and $\forall i : 1 \leq i \leq j : t_i \in T(\Sigma)_{s_i}$.

The abstract syntax of CCS is defined by the signature $\Sigma = \langle S, C \rangle$ where

$$S = \{ \text{Proc}, \text{Act}, \text{ChannelSet}, \text{Relabeling}, \text{Ident} \}$$

$$C = \{ \begin{array}{l} \text{Nil} : \text{Proc} \\ \text{Var} : \text{Ident} \rightarrow \text{Proc} \\ \text{Pre} : \text{Act} \times \text{Proc} \rightarrow \text{Proc} \\ \text{Cho} : \text{Proc} \times \text{Proc} \rightarrow \text{Proc} \\ \text{Par} : \text{Proc} \times \text{Proc} \rightarrow \text{Proc} \\ \text{Res} : \text{Proc} \times \text{ChannelSet} \rightarrow \text{Proc} \\ \text{Rel} : \text{Proc} \times \text{Relabeling} \rightarrow \text{Proc} \end{array} \}$$

2.2.2 Inductive Systems

An *inductive system* is a technique for defining a set and will serve as the underpinning of our approach to defining the transition relation for a language. Intuitively an inductive system consists of a collection of rules of the form “if elements e_1, \dots, e_n are in the set then e is in the set” along with some base cases naming elements in the set.

Definition 2.3 (Inductive Systems) *An inductive system \hat{R} is a set of rule instances. A rule instance is a pair $\langle \widehat{\text{premises}}, \widehat{\text{conclusion}} \rangle$ where $\widehat{\text{premises}}$ is a set and $\widehat{\text{conclusion}}$ an element of a set. We assume that $\widehat{\text{premises}}$ is finite for all rule instances. An inductive system \hat{R} defines a set $I(\hat{R})$ as follows. A proof that $b \in I(\hat{R})$ is a sequence a_1, \dots, a_n , such that $a_n = b$ and $\forall i : 1 \leq i \leq n : \exists \langle p, c \rangle \in \hat{R} : c = a_i \wedge p \subseteq \{a_j \mid j < i\}$. When such a proof exists for b , we write $\hat{R} \vdash b$. The set inductively defined by \hat{R} is then $I(\hat{R}) = \{b \mid \hat{R} \vdash b\}$. If the elements of $I(\hat{R})$ are pairs $\langle b, s \rangle$, then $I(\hat{R})$ may be viewed as a family of S -indexed sets.*

2.2.3 Rule-Based Inductive Definitions

A rule-based inductive definition is a way of specifying an inductive system and its associated set. An SOS for a language will be given as a rule-based inductive definition defining the transition relation (and possibly other auxiliary relations). Like inductive systems an inductive definition is a collection of rules of the form “if elements e_1, \dots, e_n are in the set then e is in the set” and some base cases; however, here the elements named in the rules may be terms containing free *metavariables*. The idea is that the metavariables of a rule are instantiated to obtain a rule instance in the corresponding inductive system. Additionally each rule may have a *side condition* that is a predicate over the same metavariables in the rule. In order for a particular instantiation of a rule to be included in the associated inductive system the side condition must be satisfied.

Definition 2.4 (Rule-Based Inductive Definitions) *Let $\Sigma = \langle S, C \rangle$ be a signature and X an S -sorted set of variables. A rule-based inductive definition R defines a family of relations $\mathcal{R} = \mathcal{R}_1, \dots, \mathcal{R}_n$ where each $\mathcal{R}_i \subseteq T(\Sigma)_{s_1^i} \times \dots \times T(\Sigma)_{s_{m_i}^i}$ for $m_i \in \text{Nat}$, $s_1^i, \dots, s_{m_i}^i \in S$. We extend the notion of open terms to relations in the obvious fashion; if $\mathcal{R}_i \subseteq T(\Sigma)_{s_1^i} \times \dots \times T(\Sigma)_{s_{m_i}^i}$ then $\mathcal{R}_i(X) \subseteq T(\Sigma, X)_{s_1^i} \times \dots \times T(\Sigma, X)_{s_{m_i}^i}$. $\mathcal{R}(X)$ denotes $\bigcup_{i=1}^n \mathcal{R}_i(X)$. R consists of a set of inference rules of the form:*

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

where

- $\text{premises} \subseteq \mathcal{R}(X)$
- $\text{conclusion} \in \mathcal{R}(X)$
- side condition is a predicate over X

The associated inductive system \hat{R} is then defined as follows. Let $\Gamma(\Sigma, X)$ be the set of functions mapping $(\forall s \in S)$ variables in X_s to closed terms in $T(\Sigma)_s$. We also let $t[\gamma]$ represent the term obtained by the simultaneous substitution of each variable x in t by $\gamma(x)$ and we extend this notation to relations and predicates in the obvious way. If $r = \langle p, c, sc \rangle \in R$, then $\text{instances}(r) = \{\langle p[\gamma], c[\gamma] \rangle \mid \gamma \in \Gamma(\Sigma, X) \wedge \text{sc}[\gamma] = \text{true}\}$. The associated inductive system $\hat{R} = \{\text{instances}(r) \mid r \in R\}$ and $I(\hat{R})$ defines \mathcal{R} . We overload notation and say $R \vdash b$ when $\hat{R} \vdash b$.

2.2.4 Defining the Transition Relation

We now have the tools needed for defining SOS.

Definition 2.5 (SOS) *Let L be a design language whose abstract syntax is given by the signature $\Sigma = \langle S, C \rangle$. We assume that S includes distinguished sorts $Proc$ and Act such that closed terms over these sorts represent processes in L (i.e. $Proc_L = T(\Sigma)_{Proc}$) and actions in L (i.e. $Act_L = T(\Sigma)_{Act}$). A Structural Operational Semantics (SOS) for L is a rule-based inductive definition R specifying a family of relations \mathcal{R} that contains a relation $\mathcal{R}_i \in Proc_L \times Act_L \times Proc_L$ called the transition relation. When $\langle p, a, p' \rangle \in \mathcal{R}_i$ we write $p \xrightarrow{a} p'$.*

We now demonstrate the SOS approach by giving an SOS for Milner's Calculus of Communicating Systems (CCS) [7]. The CCS transition relation is defined by the rule-based inductive definition shown in Figure 2.2.4. Several auxiliary predicates and functions are used in the rules. The predicate $inverses : Act_{ccs} \times Act_{ccs} \rightarrow boolean$ is true for two actions if they are an input and an output over the same channel. The function $channel : Act_{ccs} \rightarrow \Lambda$ returns the channel component of a given action. The predicate $\notin : \Lambda \times 2^\Lambda \rightarrow boolean$ takes a channel name and a set of channel names and returns true when the channel name is not in the set. Finally a relabeling function $f : \Lambda \rightarrow \Lambda$ is lifted to a function over actions ($\hat{f} : Act_{ccs} \rightarrow Act_{ccs}$) as follows.

$$\hat{f}(a) = \begin{cases} f(\lambda) & \text{if } a \in \Lambda \text{ and } \text{labelof}(a) = \lambda \\ '(f(\lambda)) & \text{if } a = '\lambda, \lambda \in \Lambda \\ \tau & \text{if } a = \tau \end{cases}$$

$\frac{p \xrightarrow{a} p'}{X \xrightarrow{a} p'} (proc X = p)$	$\frac{}{a.p \xrightarrow{a} p} (true)$	$\frac{p \xrightarrow{a} p'}{p+q \xrightarrow{a} p'} (true)$
$\frac{q \xrightarrow{a} q'}{p+q \xrightarrow{a} q'} (true)$	$\frac{p \xrightarrow{a} p'}{p q \xrightarrow{a} p' q} (true)$	$\frac{q \xrightarrow{a} q'}{p q \xrightarrow{a} p q'} (true)$
$\frac{p \xrightarrow{a} p', q \xrightarrow{b} q'}{p q \xrightarrow{\tau} p' q'} (inverses(a, b))$	$\frac{p \xrightarrow{a} p'}{p \setminus L \xrightarrow{a} p' \setminus L} (channel(a) \notin L)$	$\frac{p \xrightarrow{a} p'}{p[f] \xrightarrow{\hat{f}(a)} p'[f]} (true)$

Figure 2: An SOS for CCS

2.2.5 Building Labeled Transition Systems

An SOS for L defines a mapping from process terms of the language to labeled transition systems.

Definition 2.6 (Ψ_{SOS_L}) *Let $SOS_L = R$ be an SOS for design language L defining \longrightarrow . The mapping $\Psi_{SOS_L} : Proc_L \rightarrow LTS$ is defined as follows. For $p \in Proc_L$, $\Psi_{SOS_L}(p) = \langle S, s_0, Tr \rangle$, where*

- $S = reachable(p)$

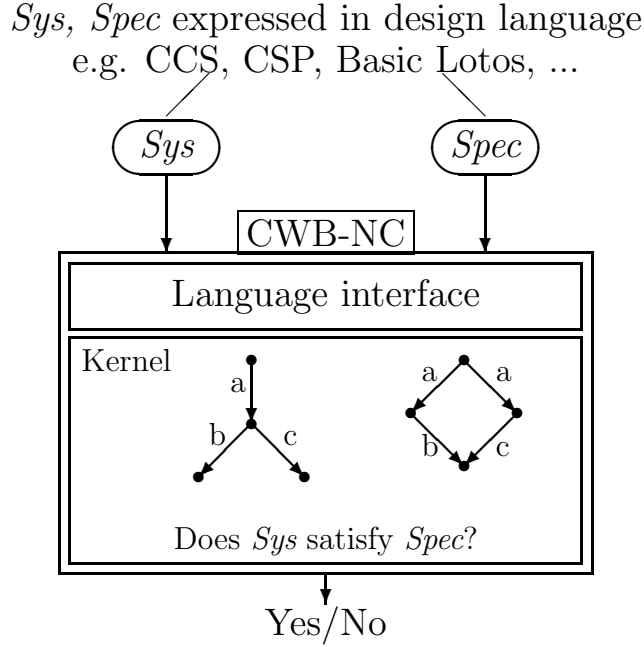


Figure 3: Structure of the CWB-NC

- $s_0 = p$
- $Tr = \{\langle s_1, a, s_2 \rangle \mid s_1, s_2 \in S, a \in Act_L, R \vdash s_1 \xrightarrow{a} s_2\}$

The set $reachable(p)$ includes all processes that p may evolve to during execution and is defined as the least set containing:

- p
- all $p'' \in Proc_L : \exists a \in Act_L, p' \in Proc_L : R \vdash p \xrightarrow{a} p' \wedge p'' \in reachable(p')$

An SOS for a language L and the induced mapping Ψ_{SOS_L} enable the many types of verification implemented in the CWB-NC over LTS to be applied to terms in L .

2.3 Retargeting the CWB-NC

The CWB-NC employs the basic design sketched in Figure 3. To use the CWB-NC, one describes a system using a design language such as CCS, CSP, or Basic Lotos, feeds this system description along with a specification into the tool, then performs various types of analysis. For each type of analysis, the first step that the CWB-NC must perform is to *compile* the system description into a labeled transition system (LTS) that models the behavior of the system. The *language interface* that performs this compilation may be viewed as a compiler with the design language as its source and LTS as its target. When given a system to compile, the language interface must also perform syntactic and semantic checks to ensure that the given input is a well-formed design language term.

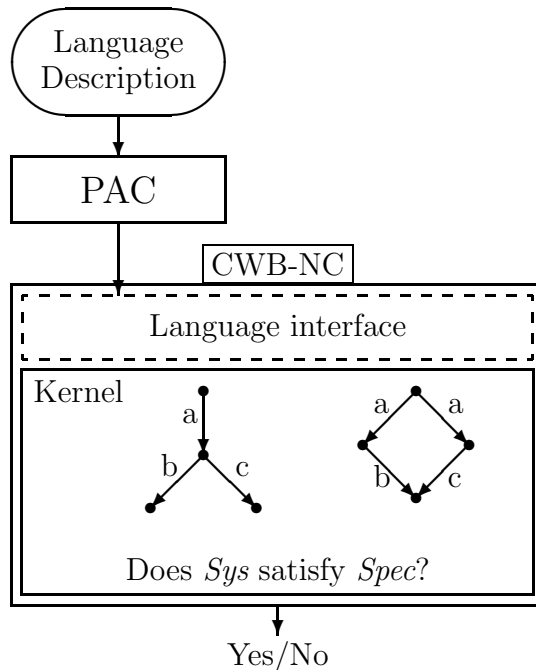


Figure 4: The purpose of the PAC is to generate new CWB-NC language interfaces.

Once an LTS for a system has been compiled, the user may analyze the system in a number of different ways. Note that the analysis routines in the CWB-NC’s *kernel* directly manipulate the LTS and thus do not depend on the particular design language used for describing systems.

To retarget the CWB-NC to a new design language, the language interface must be replaced. The purpose of the PAC is to generate new language interfaces for the CWB-NC. As shown in Figure 4, given a high-level description of a design language, the PAC generates a new language interface. 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 CWB-NC. There is no PAC run-time system that becomes part of the CWB-NC. The PAC can also be viewed as a “compiler compiler” since the generated code is a “compiler” which accepts a design language program as input and produces a labeled transition system as output.

Figure 5 shows that a PAC-generated language interface is structured much like a traditional compiler and consists of three components:

1. The *parser* reads in a system description and checks for syntactic correctness.
2. A *parse tree* stores an intermediate representation of the system.
3. A collection of *semantic routines* takes a parse tree as input and builds a LTS.

Figure 5 also illustrates that a PAC language specification includes sections corresponding to each of the three components of a language interface.

1. A description of the language’s concrete syntax, given as a Yacc-like grammar, is used to generate the parser.

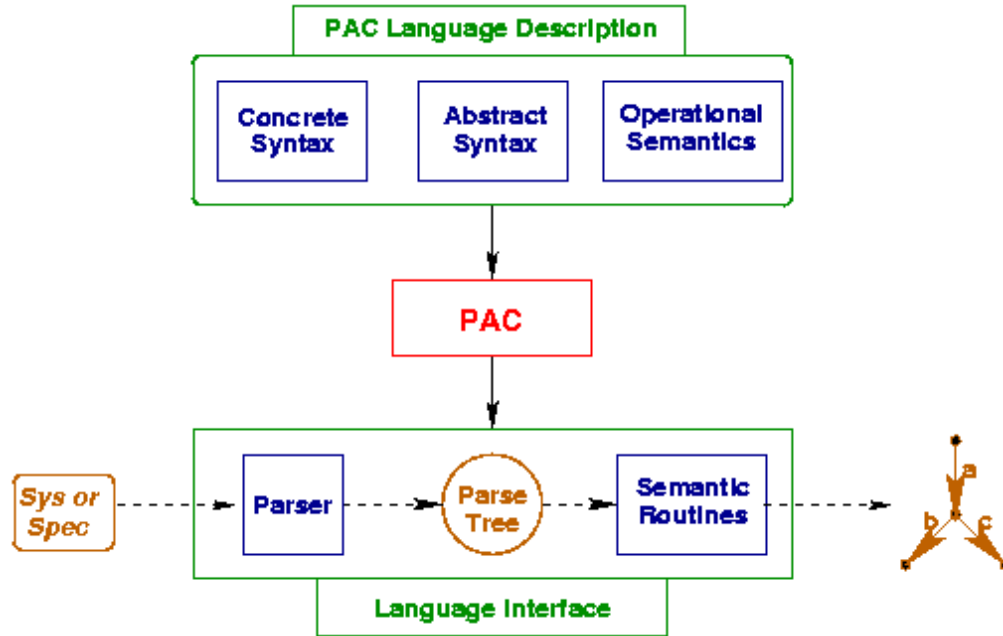


Figure 5: Structure of a PAC language specification and a PAC-generated language interface.

2. A description of the language’s abstract syntax, specified as a many-sorted signature, is used to generate the data structures for parse trees.
3. An SOS for the language is used to generate the semantic routines of the language interface.

High-level, formal notations such as context free grammars and regular expressions have proven to be ideal input formats for parser and scanner generators. The PAC extends this notion by using SOS rules (introduced in Section 2.2) as input for generating routines for computing various user-defined semantic relations. For example, a commonly defined semantic relation is one which determines the set of single step transitions that a term in the design language is capable of performing. A PAC language definition completely and concisely defines a design language and using these high-level notations makes implementing a language interface with the PAC much simpler than implementing one by hand.

2.4 PAC Architecture

Figure 6 contains a diagram showing the architecture of the PAC. A PAC language specification is provided in two files. The first file, whose name has the `.syn` suffix, contains the descriptions of the abstract and concrete syntax of the language; while, the second, whose name has the `.sos` suffix, includes the SOS rules defining the semantics of the language. Sections 3 and 4 contain detailed descriptions of the syntactic and semantic components of a PAC language specification.

A language interface is generated from the two input files as follows. First the `.syn` file is parsed and checked for syntactic correctness by the PAC *syntax specification parser* and, if the file contains no errors, an internal representation of the abstract and concrete syntax of the specified language is produced. The syntax description is then input into the *rules parser generator* which generates a parser to be used to process the `.sos` file. The rules parser must be generated on-the-fly because

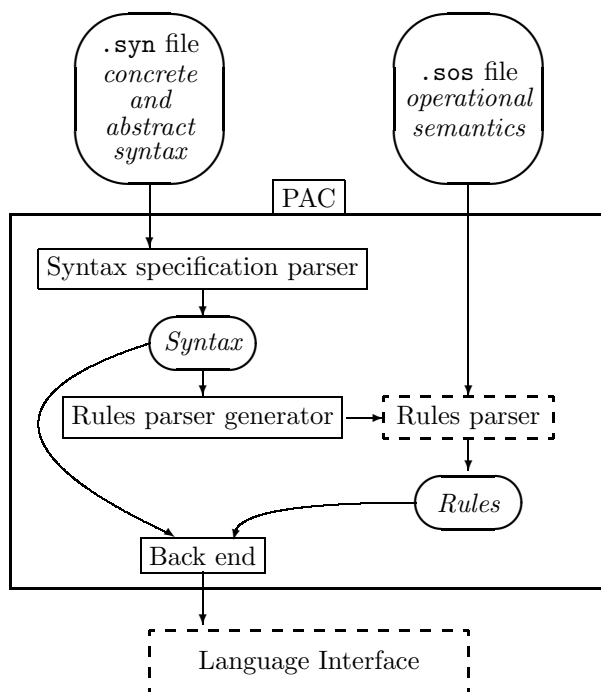


Figure 6: Architecture of the PAC.

the syntax of the language as given in the `.syn` file is used in the SOS rules defining the semantics of the language. This allows the SOS rules to be written in a very natural way as they appear in the literature. Next the newly generated rules parser processes the `.sos` file and creates an internal representation of the SOS rules.

Finally, the intermediate representations of the language syntax and semantics are fed into the PAC *back end* which generates the actual implementation of the language interface. As was described in the previous section, the generated interface has three components: a parser, data structures for parse trees, and semantic routines. Lex and Yacc specifications are generated to implement the parser, while SML source code is generated to implement the data structures for parse trees and the various semantic routines.

The PAC may also be used in its current form with tools other than the CWB-NC; however, the tools must be implemented in SML. To use the PAC with tools implemented in other languages, a new back end must be written to generate code in the new implementation language, but only the back end need change. Using the the PAC to generate language interfaces for different tools from the same PAC language specification also provides a way to make collaborative use of different tools possible.

2.5 Implementation

The PAC is implemented in Standard ML (SML). The system, which currently consists of roughly 18,000 lines of code, is batch-oriented; it processes inputs and either generates output files or reports error messages.

3 PAC Syntax Specifications

The previous section explained that a PAC language specification consists of a syntactic and a semantic component. This section describes the format of the syntactic component and the following section gives the details of the semantic component. Throughout the two sections, we use a definition of CCS [7] (introduced in Section 2.2.4) as a running example.

3.1 PAC Identifiers and Keywords.

Both the syntactic and semantic components of a PAC language description include numerous uses of identifiers. In the PAC, an identifier is any sequence of (upper and lower) case letters, numbers, and the symbols `-`, `_` and `'` that begins with a letter and is not a keyword. The following are the PAC keywords.

<code>Int</code>	<code>EMPTY_STR</code>	<code>RULE_SET</code>	<code>String</code>	<code>add</code>
<code>and</code>	<code>bool</code>	<code>cons</code>	<code>cwb</code>	<code>empty</code>
<code>empty_list</code>	<code>end</code>	<code>funcs</code>	<code>grammar</code>	<code>head</code>
<code>inputs</code>	<code>is</code>	<code>isNull</code>	<code>language</code>	<code>left</code>
<code>list</code>	<code>lists</code>	<code>noassoc</code>	<code>non_empty_list</code>	<code>nonterminals</code>
<code>not</code>	<code>of</code>	<code>or</code>	<code>pragmas</code>	<code>priorities</code>
<code>rels</code>	<code>right</code>	<code>rules</code>	<code>sorts</code>	<code>string</code>
<code>syntax</code>	<code>tail</code>	<code>tokens</code>	<code>true</code>	<code>unit</code>
<code>vars</code>				

3.2 Structure of the Syntactic Description

A PAC syntactic description defines the abstract and concrete syntax of a language and must be contained in a file whose name includes the suffix `.syn`. Figure ?? contains the (elided) version of a syntactic description for CCS found in a file named `ccs.syn`. Note that comments in PAC specifications begin with the `%` character and end with a newline. A `.syn` file begins with the `ALGEBRA` keyword and is followed a PAC identifier corresponding to the name of the design language being defined; the file is terminated with the keyword `end`. By convention, the base name of the file containing the `ALGEBRA` module (i.e. the name with the `.syn` extension removed) and the name of the algebra are the same. The remainder of this section is devoted to a more detailed explanation of the contents of each section of the `.syn` file.

3.3 Abstract Syntax

Recall from Section 2.2 that the SOS approach defines a transition relation for a language over the abstract syntax of terms in the language. The notation employed by the PAC for defining abstract syntax is based on the machinery defined in Section 2.2.1. The abstract syntax of a language is specified by a many-sorted signature and parse trees for terms in the design language are represented as closed terms over the signature. A many-sorted signature consists of a set of sorts, defined in the `sorts` section, and a set of constructors, defined in the `cons` and `funcs` sections.

3.3.1 Sorts

In the `sorts` section, users define the syntactic categories for their language. The sorts for the CCS example appear in Figure 8 and include declarations for sorts `act` (actions), `agent` (CCS

```

% ccs.syn
% 2-12-96
% This file contains a PAC specification for CCS.

language ccs

sorts
...
cons
...
funs
...
rels
...
inputs
...
pragmas
...
syntax
...
rules syntax
...
end % language ccs

```

Figure 7: An language module for CCS

```

sorts

  id, act , rename , relabeling , id_set, restriction,
  automaton, state, trans, int,
  agent, binding, ('a frame), ('a env), spec

```

Figure 8: A sorts section for CCS.

processes), `'a frame` (frames, or mappings from identifiers to values), and others. Note that sorts may be polymorphic: in the case of `'a frame`, for instance, the `'a` may be instantiated with any well-formed sort. The PAC also includes four built-in sorts: `string` for character strings, `bool` for booleans, `'a list` for polymorphic lists, and `unit`. Note that the parentheses around polymorphic sorts are required.

3.3.2 Cons

The `cons` section defines the term constructors to be used to build elements in the different syntactic categories. Implementations for these constructors are generated by the PAC. Each constructor definition includes a name, domain, and codomain

$$name : domain \rightarrow codomain$$

where *name* is a PAC identifier, *domain* is a list of sorts delimited by the character `*`, and *codomain* is a sort.

The `cons` section of the CCS example, given in Figure 9, defines the constructors used in CCS. For example `Nil` is introduced as a constructor taking no arguments and producing a value of

```

cons

Nil      : unit      -> agent
Bottom  : unit      -> agent
Ag_var   : id        -> agent
Prefix  : act * agent -> agent
Plus     : agent * agent -> agent
Par      : agent * agent -> agent
Res      : agent * restriction -> agent
Rel      : agent * relabeling -> agent
Aut      : automaton -> agent
Fix      : agent * (agent frame) -> agent

Res_set  : id_set -> restriction
Res_var  : id     -> restriction

Rename  : id * id     -> rename

Ag_bind  : id * agent -> binding
Set_bind : id * id_set -> binding
Spec     : (binding list) -> spec

```

Figure 9: A `cons` section for CCS.

sort `agent`; that is, `Nil` is an agent constant (which represents a terminated agent). The `Prefix` constructor takes an action and an agent as arguments and produces an agent. In this version of CCS, users may bind identifiers to sets of identifiers and then use these identifiers in place of sets in the restriction operator. To cater for this possibility, the language introduces a sort `restriction` and two constructors, `Res_set` and `Res_var`, permitting sets of identifiers (i.e. sets of channel names), or a single identifier (a variable name bound to a set) to be viewed as restrictions. The `binding` sort is used to bind an identifier to either an agent or an identifier set and a CCS specification consists of a list of such bindings. 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.

3.3.3 Funs

The `funs` section introduces additional constructors that differ from those in the `cons` section 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. The syntax of declarations in this section is exactly the same as in the `cons` section.

Figure 10 contains the `funs` section for CCS. To generate a CWB-NC language interface on the basis of this, a user would need to provide implementations in Standard ML (the language in which the CWB-NC is written) for each of the operations declared here. The PAC also generates implementations of sorts having constructors declared for them but relies on users to specify implementations of sorts for which no constructors have been specified. For example, `agent` would have a PAC-supplied implementation, since several constructors have `agent` 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.

```

funcs

tau      : unit -> act
input    : id   -> act
output   : id   -> act

id_parse : string -> id

label_of : act     -> id
inverse  : act     -> act
inverses : act * act -> bool

mk_relabeling : (rename list) -> relabeling
apply         : relabeling * act -> act
compose      : relabeling * relabeling -> relabeling

mk_set      : (id list) -> id_set
member      : id * id_set -> bool
insert      : id * id_set -> id_set
id_set_union : id_set * id_set -> id_set

empty_set_binding_list : unit -> (binding list)
mk_ag_frame : (binding list) -> (agent frame)
mk_set_frame : (binding list) -> (id_set frame)

empty      : unit -> ('a env)
push_frame : ('a frame) * ('a env) -> ('a env)
lookup     : id * ('a env) -> 'a

atoi      : string -> int

mk_automaton : int * (state list) -> automaton
State       : int * (trans list) -> state
Trans      : act * (int list) -> trans

```

Figure 10: A `funcs` section for CCS.

```

rels

transitions : (agent env) * (id_set env) * agent * act * agent -> bool
diverges    : (agent env) * (id_set env) * agent -> bool
aut_trans   : automaton * act * automaton -> bool

```

Figure 11: A `rels` section for CCS.

```

inputs

transitions is [1,2,3]
diverges is [1,2,3]
aut_trans is [1]

```

Figure 12: An `inputs` section for CCS.

3.4 Semantic Relation Declarations

The `rels` and `inputs` sections contain information about each semantic relation to be defined or used in the SOS rules of the semantic component of the language description. Like constructors the relations declared here fall into one of two categories: PAC-generated or user-defined. The user provides code to implement relations in the later group; whereas, SOS rules are given for those in the former from which the PAC generates a function implementing the relation.

3.4.1 Rels

The `rels` section defines the names and “types” of the semantic relations that will be either defined by or used in the SOS rules for the language. The syntax for relation declarations is the same as that for constructors and functions, with the additional requirement that the codomain must be `bool`. The `rels` section of file `ccs.syn` appears in Figure 11. The `rels` component of the example introduces three semantic relations: `transitions`, `diverges` and `aut_trans`. 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 of these two relations includes an `agent` environment and an `id_set` environment argument. The third relation is used to manipulate the user-defined sort `automaton`.

3.4.2 Inputs

The `inputs` section indicates what types the functions computing the relations declared in the `rels` section should have (i.e. which positions in the relation should be “inputs” and which should be “outputs”). Each relation has an entry in the the `inputs` section of the following form:

$$rel \text{ is } [i_1 , \dots , i_n]$$

Each i_j is an integer index into the domain of rel that indicates that the i_j^{th} position of the domain corresponds to an input to the the generated function. Note that indexing begins with one.

The `inputs` section of `ccs.syn` appears in Figure 12. In the case of `transitions`, 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 should return the set of all action-agent pairs which, when combined with the input 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 should generate a function taking three arguments and returning a boolean.

3.5 Pragmas

The PAC is structured to support different back ends targeted to different verification tools. The current version of the tool includes a back end for the CWB-NC. The `pragmas` section includes miscellaneous directives for specific back ends. Each pragma consists of a keyword indicating the back end to which the pragma is directed (the only current option is `cwb`) followed by a string delimited by the double quote character (`"`). The text in the string is passed through to the appropriate back end for processing. The PAC supports the following `cwb` pragmas demonstrated in Figure 13. In general each of the pragmas includes one or more keywords, followed by a colon, then some arguments.

3.5.1 user files

The `user files` pragma gives the names of files that include user-provided code (implementations of user-defined sorts, of functions listed in the `funcs` section, and of relations for which no `RULE_SET` module is provided). See Section 5.5 for a description of precisely what code must be provided by the user. The files named here are included in the generated Compilation Manager file.

3.5.2 parser entries

The `parser entries` pragma indicates what entries the parser produced by the PAC should have. Each argument is the name of a nonterminal declared in the `syntax` section of a `.syn` file, which is described in Section 3.6. For each nonterminal `foo` of (SML) type `foo-type` listed here, a parsing function `parse_foo` of type `string → foo-type` is generated.

3.5.3 build_keyword_table

The `build_keyword_table` pragma indicates that the generated scanner should use a keyword table to reduce its size. In such a scheme, all alpha-numeric tokens are recognized by a single regular expression. When such a string is recognized, the keyword table is consulted to determine if the token is a keyword, otherwise it is assumed to be an identifier. When the `build_keyword_table` pragma is used, the `tokens` section of the `.syn` file should include the definition of a token named `ID`. The regular expression for this `ID` token should match all keywords in the language. Keywords are declared as standard tokens as follows:

```
"begin"           => BEGIN
"end"             => END
"[a-zA-Z][a-zA-Z0-9' _-]*" => ID  of String
```

3.5.4 unparser entries

The `unparser entries` pragma indicates what unparsers will be generated by the PAC. Each argument is the name of a nonterminal declared in the `syntax` section of a `.syn` file, which is described

```

pragmas

cwb "parser entries: act, agent, spec, id_set"
cwb "build_keyword_table"
cwb "unparser entries: act, agent, id_set"
cwb "unparser info:
    space(1) PLUS space(1),
    space(1) PAR space(1),
    PREFIX no_break,
    space(1) END space(1),
    space(1) WHERE space(1),
    space(1) AND space(1),
    break LBRACE no_break,
    space(1) EQUALS space(1) no_break,
    ID no_break,
    space(1) LCBRACE,
    RCBRACE space(1),
    COLON space(1)
"
cwb "sharing constraints:
    user_decls_sig {
        where type int = int
        where type id_set = Id.Set.set
        where type id = Id.id
        where type 'a frame = 'a F.frame
        where type 'a env = 'a E.env
        where type relabeling = Relabeling.relabeling
        where type act = CcsBasicAct.act
        where type automaton = CcsAut.automaton
        where type trans = CcsAut.trans
        where type state = CcsAut.state
    }
"
cwb "comments: eoln {\*}"
cwb "sos comments: eoln {\*}"
cwb "cache transitions: parallel_1,parallel_2,parallel_3"
cwb "naming convention: underscore"

```

Figure 13: A pragmas section for CCS.

in Section 3.6. For each nonterminal *foo* of type *foo_type* listed here, an unparser *unparse_foo* of (SML) type *foo_type* \times *int* \rightarrow *string* is generated. The integer argument indicates the width (in characters) of the area onto which the unparsed string will be printed. This allows the generated unparsers to insert newline characters appropriately.

3.5.5 unparser info

The `unparser info` pragma is used to *tune* the generated unparsers by specifying the number of spaces that should appear before and after tokens and where line breaks are allowed. The default is for a line break to be allowed after a token and for no spaces to be inserted either before or after a token. These settings may be changed for any token by giving an entry in an `unparser info` pragma of the following form: `<space(n1)> token_name <space(n2)> <no_break>`. The brackets (`<` `>`) indicate that all arguments except the token name are optional. The arguments have the following meaning:

- *n*₁ is an integer indicating the number of spaces preceding the token.
- *n*₂ is an integer indicating the number of spaces following the token.
- A `no_break` argument indicates that line breaks should not occur after the token.

3.5.6 sharing constraints

The `sharing constraints` pragma indicates that SML sharing constraints should be inserted into PAC-generated code. The text between the open (`{`) and close (`}`) delimiters is simply inserted into PAC-generated code at the location indicated by the keyword preceding the braced string. Using `ccs.syn` as an example, the four currently supported location keywords ¹ indicate the following locations in various PAC-generated files.

Keyword	Location
<code>decls</code>	functor header of <code>CcsDeclsFn</code> in <code>ccs-decls-fn.sml</code>
<code>grm</code>	functor header of <code>CcsLrValsFn</code> in <code>ccs.grm</code>
<code>pac</code>	functor header of <code>CcsTopFn</code> of <code>ccs-pac-fn.sml</code>
<code>user_decls_sig</code>	after the end of signature <code>CCS_USER_DECLS</code> in <code>ccs-user-decls-sig.sml</code>

It is the user's responsibility to ensure that the text is a correctly formed SML sharing constraint.

3.5.7 comments

The `comments` pragma specifies how comments are declared in the design language being defined. Two styles of comments are supported.

- A pragma of the form

```
comments:  eoln bracketed-string
```

¹Note that since pragmas are simply passed through to the back end for processing, the location keywords are not recognized as keywords in the `.syn` file and therefore are not listed in Section 3.1.

indicates that all text between the token specified by *bracketed-string* and an end of line character should be ignored.

- A pragma of the form

`comments: balanced bracketed-string1 bracketed-string2`

indicates that all text between the token specified by *bracketed-string1* and the token specified by *bracketed-string2* should be ignored.

3.5.8 `sos comments`

The `sos comments` pragma specifies how comments are declared in the `sos` file for the language. The format for the `sos comments` pragma is exactly as described above for the `comments` pragma.

3.5.9 `cache`

The `cache` pragma relates to one of the optimizations performed by the PAC when generating code. The PAC-generated LTS compilers proceed in a breadth first manner. To construct an LTS for a process p create an LTS state for p , compute the transitions of p , add an edge for each transition, then repeat the process for each target state of a transition. The basic idea of the caching optimization implement by the PAC is to maintain a hash table storing the results of recursive calls to the transitions function; 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 compilation of the CCS process $p|q$ proceeds. First, a call is made to the transitions function with $p|q$ as input. After making the recursive calls to compute the transitions of p and q , the transitions function saves the results of these calls in a table. From the semantics of CCS, 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 LTS for $p|q$ will include computing the transitions of each $p'|q$; but, in this case instead of making recursive calls to re-compute the transitions of q , the transitions function 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.

The PAC user indicates which results to save with a `cache` pragma . The arguments to the pragma are the names of SOS rules for which the results of the recursive calls from its premises should be stored. We noted above that the semantics of the parallel composition operator guarantees that the transitions of subterms will be re-computed during LTS compilation. The question then arises as to whether other operators of a language yield similar gains if semantic information is stored for them as well. One approach would be to simply cache the results of all recursive calls to the transitions function. Experiments have shown however that the optimal strategy is to save only results guaranteed to be re-computed. In the future we would like to incorporate routines in the PAC to automatically determine which results to cache based on the SOS of the language.

3.5.10 `naming convention`

The `naming convention` pragma indicates how multi-word function names generated by the PAC should be formulated. The two supported conventions are:

`underscore` separate words with underscores, e.g.

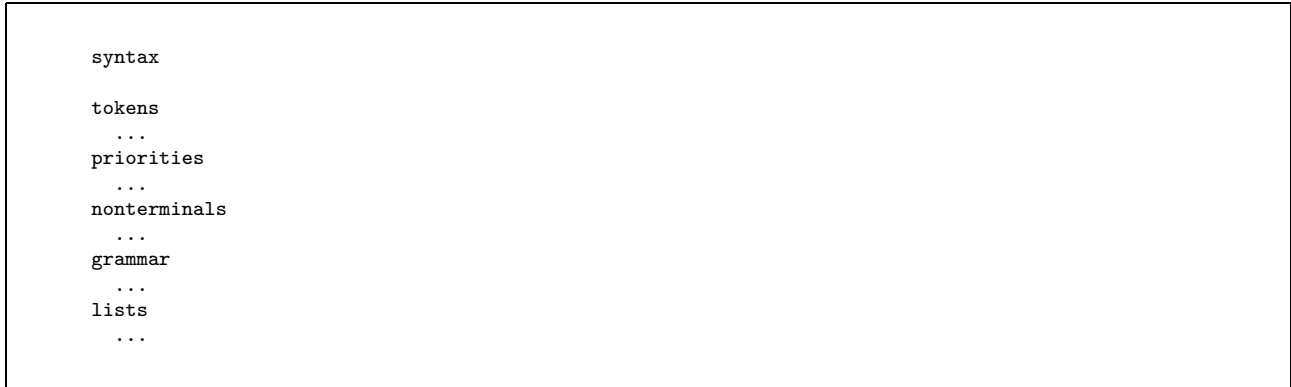


Figure 14: Structure of the `syntax` and `rules syntax` sections of a PAC language description.

```
is_foo, foo_inv, bar_eq, bar_hash, parse_baz, unparse_baz
```

`capitalize` capitalize first letter of each word after the first word, e.g.

```
isFoo, fooInv, barEq, barHash, parseBaz, unparseBaz.
```

3.6 Concrete Syntax

The `syntax` and `rules syntax` sections contain descriptions (as yacc-like grammars) of the concrete syntax of

- the design language (`syntax` section) and
- the relations used to define the language’s semantics (`rules syntax` section).

The over-all structure of these two sections is identical and is depicted in Figure 14; in what follows we discuss the contents of each subsection of the `syntax` section and summarize at the end the (minor) differences between the `syntax` and `rules syntax` sections. Note that the `priorities` and `lists` sections may be omitted from a `syntax` or `rules syntax` component.

3.6.1 Tokens

The `tokens` section includes a sequence of token definitions of the following form:

```
" regexp " => token_name < of String >
```

where *regexp* is an SML-Lex regular expression as described in the SML-Lex documentation [1] and *token_name* is a PAC identifier. By default tokens carry no value; however, the user may indicate that a string value should be carried by the token with the `of String` option. The type of the carried value will be the built-in sort `string`. The `"` character may be used within a token definition by repeating it `""`. Figure 15 lists the `tokens` section found in `ccs.syn`; note that SML-Lex reserved characters

```
? * + | ( ) ^ $ / ; . = < > [ { " \
```

```
tokens

"t"      => TAU
"'"      => PRIME
";"      => COLON
"\"      => SLASH
"nil"    => NIL
"@"      => BOTTOM
"\"      => PREFIX
"\"      => PLUS
"\"      => PAR
"\"      => LBRACE
"]"      => RBRACE
"where"  => WHERE
"and"    => AND
"end"    => END
"proc"   => PROC
"set"    => SET
"\"      => LCBRACE
"}"      => RCBRACE
"\"      => LPAREN
"\"      => RPAREN
","      => COMMA
"\"      => EQUALS
"\"      => BACKSLASH
"Aut"    => AUT
"start"  => START
"[a-zA-Z][a-zA-Z0-9' _-]*" => ID of String
"[1-9][0-9]{0,8} | 0"    => INT of String
```

Figure 15: A tokens section for CCS.

```

priorities

noassoc 40 WHERE
right   50 PLUS
right   60 PAR
noassoc 70 BACKSLASH
noassoc 80 PREFIX
noassoc 85 LBRACE
noassoc 90 NIL BOTTOM

```

Figure 16: A `priorities` section for CCS.

must be “escaped” with the `\` character.

As is usually the case in scanner definitions, the order of declarations is significant. When the generated scanner processes a string matching more than one regular expression, the token listed first in the declaration list is selected. For instance, in the example, if the `NEWID` token had been declared before the keyword tokens (e.g. `t`, `nil`, `end` ...) then the keyword tokens would never be matched.

3.6.2 Priorities

In the `priorities` section the user may assign associativity and priority values to tokens. These values enable non-parenthesized terms to be disambiguated. For example, in standard arithmetic expressions multiplication has a higher priority than addition. Therefore $a + (b \times c)$ may unambiguously be written $a + b \times c$. Tokens may also be declared as either left or right associative. Each priority declaration is of the form:

$$\textit{associativity} \quad \textit{priority} \quad \textit{token_name}$$

where

- *associativity* is `noassoc`, `leftassoc` or `rightassoc`
- *priority* is an integer. A larger value means a higher priority.
- *token_name* is the name of a token declared in the `tokens` section

The `priorities` section of `ccs.syn` is shown in Figure 16.

3.6.3 Nonterminals

The `nonterminals` section includes a declaration for each nonterminal used in the ensuing grammar. Declarations have the form:

$$\textit{nonterm_name} \textit{ of } \textit{nonterm_type}$$

where *nonterm_name* is a PAC identifier and *nonterm_type* is a *monomorphic* sort (i.e. no occurrences of sort variable such as `'a` are allowed). For example, nonterminals may have type `agent`, `act`, or `(agent frame)`, but not `('a frame)`. Figure 17 lists the nonterminals section of `ccs.syn`.

```
nonterminals

spec of spec
binding_list of (binding list)
binding of binding

agent of agent
act of act
restriction of restriction
relabeling of relabeling
rename of rename
id_set of id_set
id of id
rename_list of (rename list)
id_list of (id list)
agent_frame of (agent frame)
agent_binding of binding
agent_binding_list of (binding list)

automaton of automaton
state of state
state_list of (state list)
trans of trans
trans_list of (trans list)
int of int
int_list of (int list)
```

Figure 17: A nonterminals section for CCS.

```

grammar

spec : binding_list          (Spec(binding_list))

binding : PROC id EQUALS agent  (Ag_bind(id,agent))
        | SET id EQUALS id_set  (Set_bind(id,id_set))

agent : NIL                  (Nil())
      | BOTTOM                (Bottom())
      | id                    (Ag_var(id))
      | act PREFIX agent      (Prefix(act,agent))
      | agent PLUS agent      (Plus(agent1,agent2))
      | agent PAR agent       (Par(agent1,agent2))
      | agent BACKSLASH restriction (Res(agent,restriction))
      | agent LBRACE relabeling RBRACE (Rel(agent,relabeling))
      | agent WHERE agent_frame END (Fix(agent,agent_frame))
      | automaton             (Aut(automaton))
      | LPAREN agent RPAREN    (agent)

act : TAU                    (tau())
    | id                      (input(id))
    | PRIME id                 (output(id))

restriction : id_set          (Res_set(id_set))
            | id              (Res_var(id))

id_set : LCBRACE id_list RCBRACE (mk_set(id_list))

relabeling : rename_list      (mk_relabeling(rename_list))

rename : id SLASH id          (Rename(id1,id2))

agent_frame : agent_binding_list (mk_ag_frame(agent_binding_list))

agent_binding : id EQUALS agent  (Ag_bind(id,agent))

automaton : AUT LPAREN START EQUALS int COMMA state_list RPAREN
           (mk_automaton(int,state_list))

state : int COLON trans_list    (State(int,trans_list))

trans : act int_list           (Trans(act,int_list))

id : ID                        (id_parse(ID))

int : INT                      (atoi(INT))

```

Figure 18: A grammar section for CCS.

3.6.4 Grammar

The `grammar` section includes a yacc-like context free grammar for the language; the grammar in `ccs.syn` appears in Figure 18. The grammar consists of a list of productions, each of which has the form:

$$lhs : rhs_list$$

where

- *lhs* is a nonterminal declared in the `nonterminals` section.
- *rhs_list* is a nonempty list delimited by the character `|` where items of the list have the form:

$$particle_list (return_expression)$$

where

- *particle_list* is a possibly empty list delimited by spaces and having either nonterminals or tokens as items.
- *return_expression* is a sort-correct expression formed from constructors and functions defined in previous sections of the `language` module. The sort or the return expression must agree with the sort declared for *lhs* in the `nonterminals` section. The particles of a *rhs* serve as the atoms for building its return expression. For example, in the productions involving `PLUS` and `PAR`, an integer index is appended to a nonterminal name when a *rhs* includes more than one occurrence of a nonterminal.

3.6.5 Lists

The `lists` section gives users a shorthand for specifying the syntax of lists. Each list definition has the form:

$$list_name \text{ is } \{ non_empty_list \mid empty_list \} \text{ open separator close of } item$$

where

- *list_name* is the name of a nonterminal declared in the `nonterminals` section; it must be of type `(foo list)` for some sort *foo*.
- The second argument indicates whether or not the list may be empty.
- *open*, *separator*, and *close* are tokens declared in the tokens section. A built-in “empty” token `EMPTY_STR` may be used for any of the delimiters.
- *item* is a nonterminal declared in the `nonterminals` section. If *list_name* has type `(foo list)`, then type of *item* must be *foo*.

The `lists` section for `ccs.syn` is shown in Figure 19.

```

lists

% for top-level bindings
binding_list is non_empty_list EMPTY_STR EMPTY_STR EMPTY_STR of binding

% for fixpoint operator
agent_binding_list is non_empty_list EMPTY_STR AND EMPTY_STR of
  agent_binding

rename_list is non_empty_list EMPTY_STR COMMA EMPTY_STR of rename
id_list is non_empty_list EMPTY_STR COMMA EMPTY_STR of id

% for automata
state_list is non_empty_list EMPTY_STR EMPTY_STR EMPTY_STR of state
trans_list is empty_list EMPTY_STR EMPTY_STR EMPTY_STR of trans
int_list is empty_list LCBRACE COMMA RCBRACE of int

```

Figure 19: A `lists` section for CCS

3.7 rules syntax Differences

As indicated at the beginning of this section, `syntax` and `rules syntax` sections are structured in exactly the same manner; the difference is that the `rules syntax` component describes the “extra” syntax that needs to be added to the `syntax` grammar in order for the `.sos` file to be parsed. Declarations made in the `syntax` section may also be used in the `rules syntax` section. `rules syntax` grammars also include two built-in nonterminals: `bool`, which is used to parse expression of sort `bool`, and `relation`, which is used to parse relation expressions. The PAC provides the following built-in `bool`-sorted operations: `and`, `or`, `not`, `true`. The return expression in a production for `bool` may be any `bool`-sorted expression involving user-defined functions or relations whose input list includes all positions in the relation. Figures 20 and 21 contain the `rules syntax` portion of `ccs.syn`.

The return expression for a `relation` production must consist of the application of a relation name declared in the `rels` section to a number of arguments matching its arity (not the number of inputs). For example, the return expression for the `transition` production applies `transition` to five arguments.

In order to construct the parser for the `.sos` file, the PAC in essence “merges” the grammars in `syntax` and `rules syntax` and adds some built-in notation for rules. This notation is described in the next section.

```

rules syntax

tokens

"labelof"      => LABELOF
"inverses"     => INVERSES
"apply"        => APPLY
"compose"      => COMPOSE
"same"         => SAME
"mkRelabeling" => MKRELABELING
"id_set_union" => UNION

"in"           => IN
"insert"       => INSERT
"mkSet"        => MKSET
"mkenv"        => MKENV

"\=\=\>"      => BIGARROW
"--"          => DASHDASH
"--\>"        => ARROW
"_"           => DASH
"->"          => SHORTARROW
""            => HAT
"insort"       => INSORT
"lookup"       => LOOKUP
"push_frame"  => PUSH_FRAME
"Diverges"    => DIVERGES

priorities

left 110 SLASH
left 120 DASH

nonterminals

agent_env of (agent env)
id_set_env of (id_set env)
ag_lookup of agent
id_set_frame of (id_set frame)
id_set_eqn of binding
id_set_eqn_list of (binding list)

```

Figure 20: A rules syntax section for CCS (Part 1).

```

grammar

id : LABELOF LPAREN act RPAREN          (label_of(act))

bool : INVERSES LPAREN act COMMA act RPAREN  (inverses(act1,act2))
      | id IN id_set                       (member(id,id_set))
      | DIVERGES LPAREN agent_env COMMA id_set_env COMMA agent RPAREN
        (diverges(agent_env,id_set_env,agent))
      | act EQUALS act                       (act_eq(act1,act2))

relation : agent_env COMMA id_set_env COLON agent DASHDASH act ARROW agent
          (transitions(agent_env,id_set_env,agent1,act,agent2))
          | agent_env COMMA id_set_env COLON agent HAT
            (diverges(agent_env,id_set_env,agent))
          | automaton DASH act SHORTARROW automaton
            (aut_trans(automaton1,act,automaton2))

agent_env : LBRACE agent_frame RBRACE
           (push_frame(agent_frame,empty()))
           | PUSH_FRAME LPAREN agent_frame COMMA agent_env RPAREN
             (push_frame(agent_frame,agent_env))

id_set_eqn : id EQUALS id_set  (Set_bind(id,id_set))

id_set_frame : id_set_eqn_list (mk_set_frame(id_set_eqn_list))

id_set_env : LBRACE id_set_frame RBRACE
            (push_frame(id_set_frame,empty()))

agent : LOOKUP LPAREN id COMMA agent_env RPAREN (lookup(id,agent_env))

act : APPLY LPAREN act COMMA relabeling RPAREN (apply(relabeling,act))

relabeling : COMPOSE LPAREN relabeling COLON relabeling RPAREN
            (compose(relabeling1,relabeling2))
            | MKRELABELING LPAREN rename_list RPAREN
              (mk_relabeling(rename_list))

id_set : LOOKUP LPAREN id COMMA id_set_env RPAREN (lookup(id,id_set_env))
        | INSERT LPAREN id COMMA id_set RPAREN   (insert(id,id_set))
        | UNION LPAREN id_set COMMA id_set RPAREN (id_set_union(id_set1,id_set2))

lists

id_set_eqn_list is non_empty_list EMPTY_STR EMPTY_STR EMPTY_STR of id_set_eqn

end % language CCS

```

Figure 21: A rules syntax section for CCS (Part 2).

4 PAC Semantic Specifications

PAC semantic specifications are based on structural operational semantics (SOS) which was introduced in Section 2.2. The SOS approach uses a rule-based inductive definition to define the *transition relation* (and possibly other relations) for a language. The transition relation defines the execution steps a term in the language is capable of taking.

PAC semantic specifications are defined in a file whose name has the suffix `.sos`. This file consists of a sequence of `RULE_SET` module declarations. A user supplies a `RULE_SET` module for each relation introduced in the `.syn` file for which the PAC should generate an implementation. If no `RULE_SET` module is included for a relation, then an implementation of the relation must be provided by the user.

4.1 Structure of a `RULE_SET` Module

Each `RULE_SET` module includes a name, some variable declarations, and a list of inference rules and is structured as follows:

```
RULE_SET relation_name
vars
  var_decl_list
rules
  rule_list
end
```

where

- *relation_name* is the name of a relation declared in the `rels` section of the `.syn` file.
- *var_decl_list* is a list of variable declarations of the form:

$$\textit{variable_list} : \textit{sort}$$

where

- *variable_list* is a list of PAC identifiers with comma as the separator.
 - *sort* is a sort declared in the `sorts` section of the `ALGEBRA` module and indicates the type that each variable in *variable_list* should have. As was the case for nonterminals, a variable's type must be a monomorphic sort or an instantiated polymorphic sort.
- *rule_list* is a list of inference rules that defines *relation_name*. The structure of a rule is defined below.

4.2 Structure of an Inference Rule

Recall from Section 2.2.3 that an inference rule in a rule-based inductive definition consists of three components: a set of *premises*, a *side condition*, and a *conclusion*. For convenience, in the PAC notation, each rule also has a name. The intuitive reading of a rule is that if one is able to establish the premises, which typically involve statements about the execution behavior of sub programs of the one mentioned in the conclusion, and the side condition holds, then one may infer the conclusion. The syntax of a rule is:

$$\frac{\textit{rule_name} \quad \frac{\textit{list_of_prems_and_bexps}}{\textit{conclusion}}}{\textit{conclusion}}$$

where

- *rule_name* is an PAC identifier. Each rule in a rule set must have a unique name.
- *list_of_prems_and_bexps* contains the premises and side condition. In general, premises involve relations, while side conditions consist of zero or more boolean expressions. The items of the list are separated by commas and for convenience a side condition may be written as several boolean expressions interspersed throughout the list. When a rule is processed, a conjunction of all boolean expressions in the list becomes the side condition and the remaining relations are the premises. If the list includes no boolean expressions, then the side condition is taken to be **true**. The boolean expressions in the list are generated using the following grammar,

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

where P is a predicate: any boolean-sorted function declared in the **funcs** 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 . Functions and relations used in a rule must have been assigned a concrete syntax in the **ALGEBRA** module. The concrete syntax for any predicate to be used in a side condition should be defined in the production for the built-in nonterminal **bool** and any relation used in a premise (or conclusion) should be defined in the production for the built-in nonterminal **relation**

- The horizontal bar separating *list_of_prems_and_bexps* and *conclusion* is formed by concatenating four or more hyphens (-).
- *conclusion* is an instance of the relation being defined by the rule set.

4.3 Restrictions on the Variables in Rules

Obviously all variables appearing in a rule must be declared in the **vars** section. There are three additional requirements. Recall that SOS rules have the following conceptual ² form:

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

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 $P(t_1, \dots, t_n)$, with the t_i being terms that may involve variables. For the code produced by the PAC 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.

²As explained in the previous section, the PAC syntax requires the side condition to be above rather than beside the horizontal bar.

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 [3] while being incomparable to the tyft/tyxt pattern of [6] 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 [3, 9] and are incomparable to ours.

4.4 A RULE_SET Module for CCS

A fragment of the rules for the transition relation for CCS appears in Figure 22. All expressions in the rules are written using the concrete syntax declared in the `.syn` file; this enables the rules to look very close to the presentation of the SOS for CCS in Section 2.2, the only difference being that the PAC input must be in ASCII format. In addition, functions defined in the `.syn` file 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 τ). Rule sets can refer to relations defined in other rule sets, although no such reference is made in this example.

```

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

    -----
    ae, se : a.p -- a --> p

parallel_1
    ae, se : p1 -- a --> p1'
    -----
    ae, se : p1 | p2 -- a --> p1' | p2

parallel_2
    ae, se : p2 -- a --> p2'
    -----
    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 22: A RULE_SET module for the CCS transition relation

5 The Grungy Details

5.1 Obtaining the PAC

The PAC and CWB-NC are freely available and may be downloaded from their respective web pages:

```
www.reactive-systems.com/pac
www.cs.sunysb.edu/~cwb
```

5.2 Installing the PAC

You should have downloaded a file named `pac.tar.gz` from the web site. To compile and use the PAC requires the Standard ML of New Jersey compiler. The version that this distribution was tested with was the latest public release 110. We have compiled and used the PAC with many other versions of SML-NJ and suspect it will work with the forthcoming 111 release, but of course we can not guarantee compatibility. You will also need ML-Yacc, ML-Lex, and the Compilation Manager (CM) which are a part of the standard SML-NJ distribution. To install the PAC proceed as follows:

1. Place `pac.tar.gz` in the directory in which you would like the PAC directory to reside.
2. Uncompress `pac.tar.gz` yielding `pac.tar`.

```
% gunzip pac.tar.gz
```

3. Untar `pac.tar` to create the `pac` directory containing the PAC distribution.

```
% tar xvf pac.tar
```

4. Edit `pac/src/sources.cm` and set the two site specific SML-NJ library file names to point to the proper directories on your system.
5. Make sure the SML-NJ compiler is in your path and available as `sml`. Change to the newly created `pac/src` directory and run the `mkpac` build script.

```
% mkpac
Compiling The PAC
Standard ML of New Jersey v110 ...
...
```

If all goes well this script should compile the PAC, export a heap image named `pac-heap.arch-os` (where *arch* is your system architecture, e.g. `x86` and *os* is your operating system, e.g. `linux`), and move the heap image to the `pac/bin` directory.

6. Next edit `pac/bin/pac` (the script that launches the PAC) and set the two site specific variables to point to the proper directories as described in the script file.
7. Finally, either copy or link the SML runtime to the `pac/bin` directory. The link or copy of the file should be named `run-sml-arch-os` where *arch* and *os* are the names of your system architecture and operating system as produced when you run `pac/bin/arch-n-opsys`.

```
% cd ../pac/bin
% ln -s /usr/local/sml-110/bin/.run/run.x86-linux run-sml-x86-linux
```

Of course the location of the SML runtime may differ on your system.

The installation is now complete and you may run the PAC as described in the next subsection.

5.3 Running the PAC

The following describes the steps a user must take to generate an interface (parsers, unparsers, semantic routines) for a language *foo*.

1. Create a PAC specification for *foo* that consists of `foo.syn`, which contains the syntactic description of *foo*, and `foo.sos`, which presents the semantics of the language as a collection of structural operational semantic (SOS) rules.
2. Invoke the PAC by executing:

```
% pac
```

3. This script launches an SML session with the PAC routines preloaded. At the SML prompt, issue the following command:

```
- pac "foo" [];
```

The second argument is a list of strings where each string in the list is a flag. There are currently two legal flags each of which turns off an optimization that is on by default. The two flags are:

- `catching_off`
- `tree_flattening_off`

The PAC is run with all optimizations off with the command:

```
- pac "foo" ["catching_off", "tree_flattening_off"];
```

5.4 Files Produced by the PAC

Running the PAC with input `foo.syn` and `foo.sos` generates the following files. All but `foo.lex`, `foo.grm`, and `foo.tex` contain SML code in the form of signatures, functors and structures; this is necessitated by the fact that the CWB-NC is written in SML and that the PAC-generated files will ultimately be compiled together with the CWB-NC.

`foo-decls-sig.sml` A signature describing the PAC-implemented sorts and functions. For each sort which has constructors defined for it, an implementation of the sort will be included in the the structure matching this signature. For each PAC-implemented sort *bar* the following is included in the structure:

- An SML type: *bar*.
- An equality function: `bar_eq : bar * bar → bool`

- A hash function : $bar_hash : bar \rightarrow int$

For each constructor $Baz : s_1 * \dots * s_n \rightarrow s$ declared in the cons section of `foo.syn`, the following access functions are included in this structure:

- $Baz : s_1 * \dots * s_n \rightarrow s$
- $is_Baz : s \rightarrow bool$
- $Baz_inv : s \rightarrow s_1 * \dots * s_n$

The signature also includes the signature given in file `foo-user-decls-sig.sml`.

`foo-decls-fn.sml` A functor implementing the signature in `foo-decls-sig.sml`. The argument to the functor is a structure matching the signature in `foo-user-decls-sig.sml`.

`foo-user-decls-sig.sml` A signature describing the sorts to be implemented by the user. For each sort bar the following must be provided:

- An SML type: bar .
- An equality function: $bar_eq : bar * bar \rightarrow bool$
- A hash function : $bar_hash : bar \rightarrow int$

`foo-user-lib-sig.sml` A signature describing functions to be implemented by the user. For each function $Baz : s_1 * \dots * s_n \rightarrow s$ declared in the `funcs` section of `foo.syn`, the following functions must be provided by the user:

- $Baz : s_1 * \dots * s_n \rightarrow s$
- $is_Baz : s \rightarrow bool$
- $Baz_inv : s \rightarrow s_1 * \dots * s_n$

`foo-pac-fn.sml` A functor producing a structure containing “top-level” information: a function for each SOS rule set, implementations of sorts, and parsing and unparsing functions. The argument for the functor includes four structures: one for a (built-in) PAC library; one matching the signature in `foo-user-lib-sig.sml`; one matching the signature in `foo-decls-sig.sml`; and one matching a signature produced by SML-yacc.

`foo.lex` The lexical specification of `foo` for processing by the scanner generator SML-lex.

`foo.grm` The syntactic specification of `foo` for processing by the parser generator SML-yacc.

`foo.link.sml` A file containing the `FooPac` structure which assembles all of the generated code. All generated code is accessed via this structure.

`foo-syntax.tex` A concise description of the syntax of `foo` that can be included in latex documents.

`foo-sources.cm` A file for use by the Compilation Manager to compile the generated code stand alone (without the CWB-NC).

`foo-rules*` These are intermediate files used during the processing of the `foo.sos` file. They should be deleted by the tool, but this has not yet been implemented. You can safely delete them.

5.5 Compiling and Using the Generated Files

This section describes how to build a version of the CWB-NC supporting language `foo` using the PAC-generated files listed above. The PAC allows users to indicate that implementations of certain types and functions are to be provided by users; in order to use the PAC-generated files, one must first do the following

1. Implement a functor named `FooUserDeclsFn`. This functor should have no arguments and return a structure matching the signature in `foo-user-decls-sig.sml`.
2. Implement a functor named `FooUserLibFn`. This functor should take one argument matching the signature in file `foo-decls-sig.sml` and return a structure matching the signature in `foo-user-lib-sig.sml`.

If you list the names of the source files containing these implementations (and all supporting code as well) in a “user files” pragma, then these file names will be included in the generated `foo-sources.cm` file.

5.5.1 Building a New Version of the CWB-NC

To compile the CWB-NC your installation of SML must include support for the Compilation Manager (CM), a tool distributed with the New Jersey SML compiler that greatly simplifies the task of building and maintaining large systems. The steps for building a retargetted version of the CWB-NC are as follows.

We suggest you use one of the previously implemented interfaces (CCS for example) as a model for your new interface. Structuring your code in the same way is strongly recommended. In the following we let `.../cwb/src` denote the CWB-NC source directory. The existing interfaces are located in `.../cwb/src/interfaces`. For `foo` a subdirectory `.../cwb/src/interfaces/foo` should be created to contain `foo.syn`, `foo.sos` and the PAC-generated files. You should also create the following files and store them in the `.../cwb/src/interfaces/foo` directory.

`cwb-nc-foo-sources.cm` Use `cwb-nc-ccs-sources.cm` as your template. Do a global replace of `ccs` with `foo`. This will show you all the files that you (or the PAC) must create to retarget the CWB-NC.

`foo-act.sml` This file contains a structure matching the `BASIC_ACT` signature (`.../cwb/src/act/act-basic-sig.sml`) and an application of the `BasicActToActFn` functor to create a structure `FooAct` matching the `ACT` signature.

`foo-act-io.sml` This file includes a structure which adds some IO routines to the `FooAct` structure yielding a structure matching the `ACT_IO` signature (`.../cwb/src/act/act-io-sig.sml`). Note that the CWB-NC expects this structure to be named `ActIO`.

`foo-config.sml` Set the two strings to “foo”.

`foo-automaton.sml` Apply the two functors as done for CCS.

`foo-agent-fn.sml` This file should contain a functor implementing a structure matching the `AGENT_BASIC` signature (contained in `.../cwb/src/agent/agent-basic-sig.sml`) using structure `FooPac` as a basis (this structure is constructed in the file `foo.link.sml`). The means for doing this will vary, depending on taste; one means is to write a functor taking the `FooPac` structure as an argument and yielding a structure matching `AGENT_BASIC`.

foo-agent.sml This file applies the `AgentBasicToAgentFn` functor (`.../cwb/src/agent/agent-basic-to-agent-fn.sml`) to build a structure matching the `AGENT` signature, given the structure matching the `AGENT_BASIC` signature. The resulting Agent structure is the primary vehicle for “plugging the new interface into the CWB-NC” so the name “Agent” must be used for this structure.

Next the model checker parser must be updated to handle the syntax of actions in the new language. The Mu-Calculus parser is also implemented with the PAC. Update it as follows.

1. Create a new subdirectory `.../cwb/src/mucalc/interfaces/ccs`.
2. Copy `.../cwb/src/mucalc/interfaces/ccs/mucalc.syn` into the new directory.
3. Update the syntax of actions in `mucalc.syn` to coincide with the syntax you have defined for actions in `foo`.
4. Run the PAC.

The following files elsewhere in the CWB-NC source tree also need to be modified or created.

`.../cwb/src/make-cwb` Add `foo` as a new language.

`.../cwb/bin/cwb-nc` Add `foo` as a new language.

`.../cwb/bin/cwb-nc-tui` Add `foo` as a new language.

Finally, you are ready to compile! To do so simply go to `.../cwb/src` and run `make-cwb`. If all goes well this should generate `cwb-nc-foo.x86-linux` (if you compiled on a linux box). Newer versions of `make-cwb` move the heap image to its final destination `.../cwb/bin/heaps-x86-linux`, but if you have an older version you should move it there yourself. You are now ready to run the sucker, see the CWB-NC User’s manual for details on that.

6 Hints and Common Errors

- No keyword should be defined which coincides with a sort name. For example if there is a sort named “process”, then there should be no keyword “process” in the language. Violating this restriction results in very strange errors when parsing the `.sos` file.
- Parentheses around the names of polymorphic sorts are required. For example,
 `"Spec : (binding list) -> spec"` is a well-formed `cons` definition, but
 `"Spec : binding list -> spec"` yields an error.

References

- [1] Andrew W. Appel, James S. Mattson, and David R. Tarditi. A lexical analyzer generator for standard ml. Included in Standard ML of New Jersey Distribution.
- [2] J. C. M. Baeten and C. Verhoef. A congruence theorem for structured operational semantics with predicates. In E. Best, editor, *Proceedings CONCUR 93*, Hildesheim, Germany, volume 715 of *Lecture Notes in Computer Science*, pages 477–492. Springer-Verlag, 1993.

- [3] 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.
- [4] R. Cleaveland, T. Li, and S. Sims. The concurrency workbench of the new century. URL <http://www.cs.sunysb.edu/cwb/>.
- [5] R. Cleaveland and S. Sims. The NCSU Concurrency Workbench. In R. Alur and T. Henzinger, editors, *Computer Aided Verification (CAV '96)*, Lecture Notes in Computer Science, pages 394–397, New Brunswick, New Jersey, July 1996. Springer-Verlag.
- [6] Jan Friso Groote and Frits Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100(2):202–260, October 1992.
- [7] R. Milner. *Communication and Concurrency*. Prentice-Hall, London, 1989.
- [8] G Plotkin. A structural approach to operational semantics. Technical report, University of Aarhus, Denmark, 1981.
- [9] C. Verhoef. A congruence theorem for structured operational semantics with predicates and negative premises. In B. Jonsson and J. Parrow, editors, *CONCUR '94*, volume 836 of *Lecture Notes in Computer Science*, pages 433–448, Uppsala, Sweden, August 1994. Springer-Verlag.

Index

- .sos file, 10
- .syn file, 10, 11
 - comments, 11
 - cons section, 12
 - funs section, 12
 - grammar section, 18
 - inputs section, 13
 - lists section, 18
 - nonterminals section, 17
 - pragmas section, 13
 - priorities section, 17
 - rels section, 13
 - sorts section, 11
 - syntax section, 16
 - tokens section, 17
- abstract syntax, 5
- architecture, 10
- back end, 13
- CCS, 7
- compilation manager file, 13
- concrete syntax
 - defining in PAC specification, 16
- CWB-NC
 - design languages
 - changing, 9
- implementation, 10
- inductive systems, 6
- keywords
 - table, 14
- labeled transition systems, 3
- language module, 11
- LTS, 3
- many-sorted signature, 5
- naming convention, 16
- parsers
 - indicating which to generate, 14
 - keywords, 14
- pragmas, 13
 - cache, 15, 16
 - comments, 15
 - naming convention, 16
 - parser entries, 14
 - sharing constraints, 15
 - sos comments, 15
 - unparser entries, 14
 - unparser info, 14
- relation
 - user-defined, 20
- rule-based inductive definitions, 6
- RULE_SET module, 20
- scanners
 - keywords, 14
- semantic specification, 20
- sorts
 - built-in, 12
 - generated by the PAC, 12
 - user-defined, 12
- SOS, 4, 7
- sos rules
 - CCS example, 22
 - format, 21
 - variable restrictions, 21
- structural operational semantics, 4
- syntax specifications, 11
- unparsers
 - indicating which to generate, 14
 - tuning, 14
- user files pragma, 13