An Implementation of a Translational Semantics for an Imperative Language

Lars-åke Fredlund
Bengt Jonsson
Joachim Parrow
Swedish Institute of Computer Science, Stockholm*

Abstract

We present a semantics for an imperative programming language, Lunsen, with constructs for concurrency and communication. The semantics is given through a translation into CCS. We have implemented this translation within the framework of the Concurrency Workbench, which is a tool for analysis of finite-state systems in CCS. The point of the translational semantics is that by imposing restrictions on Lunsen so that the semantics of a program is finite-state, we can analyze Lunsen programs automatically using the Concurrency Workbench. As an illustration we include an analysis of a mutual exclusion algorithm.

1 Introduction

Concurrent programs often exhibit complex behaviors, and it is therefore important to develop methods and tools for analyzing them rigorously. Many implementations have been developed for automatic analysis of concurrent programs [CPS89, BdSV88, CES86, RRSV87, GLZ89]. Most of these tools are designed for simple models of programs, e.g. finite-state transition systems [CPS89, BdSV88, GLZ89]. However, many concurrent algorithms are naturally formulated in some imperative programming language with constructs for concurrency. In order to analyze a program using the tools just mentioned, the program must first be translated manually into the appropriate model.

In this paper, we present an automated translation of a concurrent programming language with imperative features into CCS. The imperative language, Lunsen, is ALGOL-like and contains standard constructs for sequential programming — such as assignments, procedures, and arrays — as well as constructs for parallel execution of processes. Processes can communicate both via shared variables and through synchronous channels.

The formal semantics of Lunsen is defined through a translation into CCS [Mil89]. We have implemented this translation within the framework of the the Concurrency Workbench (CWB). The point of our implementation is that CWB can be used to analyze Lunsen programs. CWB is a versatile tool which can automatically decide e.g. whether two concurrent systems are equivalent, or whether a system satisfies a property formulated in a modal logic.

The main source of inspiration for the formal semantics of Lunsen is the semantics of a sequential language given by Robin Milner in Chapter 8 of his book [Mil89]. In order to make the programs effectively analyzable, we impose restrictions on the language Lunsen so that the semantics of a

*Address: SICS, Box 1263, S-164 28 Kista, SWEDEN.

program will be finite-state: this means e.g. that the types of variables must contain only finitely many elements, and that procedures cannot call each other recursively in an arbitrary manner.

There exist other automated tools for analyzing concurrent programs written in imperative languages. EMC [CES86] is a tool for checking that a program satisfies a formula formulated in a branching time temporal logic. EMC has a preprocessor which accepts programs written in a simple CSP-like language. Xesar [RRSV87] is a tool for checking similar properties for communication protocols defined in an extension of PASCAL with facilities for communication. Auto [BdSV88] and TAV [GLZ89] are tools for analysis of concurrent systems, which are related to the Concurrency Workbench. Other translations of imperative languages into CCS include a translation from Ada by Hennessy and Li [HL83], a translation from CSP by Astesiano and Zucca [AZ81], and a translation from NIL by Smolka and Strom [SS86].

In the next section, we define the syntax of Lunsen and give an informal semantics. The translation of Lunsen to CCS goes via an intermediate language, Typed CCS, which is an extension of CCS that is presented in Section 3. The translation itself is presented in Section 4. Section 5 discusses some optimizations to the translation, and Section 6 illustrates the analysis by an example: a mutual exclusion algorithm due to Peterson. Conclusions are found in Section 7.

2 Lunsen: Syntax and Informal Semantics

Lunsen is an imperative language belonging to the ALGOL family of strongly typed languages. This means that variables and imperative constructions such as while-loops are fundamental to the language. Lunsen does not include dynamic constructions such as pointers and creating of objects. We also place restrictions on procedure calling; these restrictions have the effect that Lunsen programs can be executed without a runtime stack or a heap area and ensure that programs in Lunsen will be *finite-state*.

Lunsen also contains non-sequential primitives. Commands are executed in parallel with the **par** command and may communicate either synchronously by sending messages over ports or via shared variables. Furthermore, nondeterministic choice can be expressed in the language.

Lunsen programs communicate with the outside world (the environment of programs) either by sending messages on ports that are visible to the environment or through global, or *visible*, variables of the program which can be accessed by the environment.

The syntax for Lunsen programs is given in Table 1 using a dialect of BNF. Objects written inside slanted brackets (*[ ]*) are optional, and we let the | symbol denote alternatives (instead of grouping them on different lines). We presuppose a set of *identifiers* partitioned into *constants* ranged over by I_c, *type identifiers* ranged over by I_t, *procedure identifiers* ranged over by I_p, *Lunsen variables* ranged over by I_v, *array variables* ranged over by I_a, *program identifiers* ranged over by I_s, and *port identifiers* ranged over by I_m.

We will now give an informal description of the meaning of the Lunsen constructions.

- Programs

  A program consists of a declaration section and a command which invokes the execution of the program.

- Declarations

  Procedures, types, variables and ports are defined in a declaration section. The order between definitions is not significant.

- Types

S::=	**program** I_s; D C **endprog**;	Program
D::=	ϵ	Empty declaration
	type $I_t = T$;	Type definition
	var I_v:[G] $I_t = E$;	Variable declaration
	procedure I_p[(I_v:A I_t,...,I_v:A I_t)] D C **endproc**;	Procedure definition
	port I_m:[**visible**] I_t;	Port declaration
	D D	
A::=	**in** $\mid$ **out** $\mid$ **inout**	Parameter usage
G::=	**read** $\mid$ **write** $\mid$ **readwrite**	Visibility of variables
T::=	$\{I_c,\ldots,I_c\}$	Enumerated type
	ordered $\{I_c,\ldots,I_c\}$	Ordered type
	array $[I_t,\ldots,I_t]$ **of** I_t	Array type
	$I_t * \ldots * I_t$	Tuple type
E::=	I_c	Constant
	I_v	Variable
	$I_a[E,\ldots,E]$	Array expression
	$(E,\ldots,E)$	Tuple expression
	$\# I_c\ E$	Tuple access
	if $E \rightarrow E \mid \ldots \mid E \rightarrow E$ [**else** E] **endif**	Deterministic choice
	$E = E$	Equality
	$E \leq E$	Less-than-or-equal
	succ E	Successor function
	pred E	Predecessor function
	not E $\mid$ E **and** E $\mid$ E **or** E	Boolean functions
C::=	**begin** C **end**	Compound statement
	skip	No action
	C ; C	Sequencing
	C **par** C	Parallel composition
	$I_v := E$	Assignment
	$I_a[E,\ldots,E] := E$	Array assignment
	I_p[$(E,\ldots,E)$]	Procedure call
	if $E \rightarrow C \mid \ldots \mid E \rightarrow C$ [**else** C] **endif**	Deterministic choice
	when $P \rightarrow C \mid \ldots \mid P \rightarrow C$ **endwhen**	Port synchronization choice
	while E **do** C **endwhile**	While loop
	P	Port command
P::=	τ	Invisible action
	$I_m\ !\ E$	Send value
	$I_m\ ?\ I_v$	Receive value into variable

Table 1: The syntax of Lunsen

The only standard type in the language is **boolean**, defining the constants **true** and **false**. New types can be defined by enumerating the constants of the type, or by forming array types or tuple types. For example,

```
type int4 = {1,2,3,4};
```

defines a new type **int4** and also the predicate = over that type (e.g. $2 = 2$). By adding the keyword **ordered** before the enumerated set of constants, the relation $\leq$ will also be defined on that type as well as the functions **succ**(x) (the successor function) and **pred**(x) (the predecessor function) (in this case, applying **succ** to 4 generates an error message). As an example of an array type,

```
type arrint4 = array[int4] of boolean;
```

defines an array type of four elements, assuming the previous definition of **int4**. Each element of such an array is capable of storing one of the values **true** or **false**. An example of a definition of a tuple type is

```
type tup = int4 * int4;
```

- Variables

  A variable is defined using the **var** declaration. A variable must be supplied with an initial value when it is declared. An example:

```
var a:int4 = 2;
```

  The optional keywords **read**, **write** and **readwrite** determine how and if the variable is visible to a potential observer (user) of the program. If a variable is declared as **read**-able the observer can inquire of the value of that variable. If a variable is **write**-able the outside observer can modify the value of that variable. The **readwrite** keyword combines the effects of **read** and **write**.

  Array variables are not full members of the language in that we place some restrictions on their usage. They cannot be passed as parameters to procedures, cannot be communicated via ports, nor can they be assigned to as a single entity. It is of course possible to assign values to elements of array variables, for example,

```
var a:arrint4 = {true, true, true, true};
a[1] := false
```

- Ports

  Concurrent processes may use ports to synchronize their activities. A process sends a value on a port using the *!* operator and receives values into variables using the *?* operator. A process attempting communication on a port will halt its execution until another process is also ready for communication. As an example:

```
port p:int4;
var  x:int4;

p?x par p!2 par p!3
```

The three commands p?x, p!2, and p!3 represent three processes that execute in parallel. Each process is suspended until another process is able to communicate with it. Communication is binary. In an execution of the three commands, either the value 2 or the value 3 is assigned to the variable x.

If a port is declared to be **visible**, an observer of the program will be able to communicate with the program through that port. Otherwise the port is only accessible within the program (and in the scope of the port declaration).

- Procedures

  A procedure contains a declaration section and a command. A procedure accepts zero or more parameters. **in** parameters are used to supply values to the procedure. **out** parameters communicate results from the procedure back to the caller. **inout** parameters combine the effects of **in** and **out**.

  The semantics for procedure calls is as follows. First **in** and **inout** parameters are evaluated (*call-by-value* style) and temporary copies of **out** and **inout** parameters are created. Then the command in the procedure body is executed. After the execution of the procedure body the values in the temporary **inout** and **out** variables are copied back to the variables supplied in the actual procedure call command.

  In order to ensure that Lunsen programs are finite-state, the following (syntactic) restrictions on admissible procedure calls are enforced:

  1. A parent may always call its child.
  2. A child may never call its parent.
  3. A sibling may call another sibling as long as the call is made *tail-recursively*.
  4. A procedure may call itself as long as the call is made tail-recursively.

  Here *tail-recursive* means that no command can occur after the call command in the calling procedure, i.e. after the end of the execution of the called procedure, the calling procedure does not have to be resumed. Furthermore we demand that if an **inout** or **out** variable occurs at position i in the enumerated list of **inout** and **out** formal parameters to the procedure in which the tail-recursive call is made, then it should occur at the same place in the corresponding enumerated list of **inout** and **out** actual parameters to the called procedure in the tail-recursive call. We also require that these two enumerated lists have the same number of elements, i.e. no extra **inout** or **out** parameters are allowed in the formal parameter list of the tail-recursively called procedure. To illustrate:

```
procedure p;
  procedure p1;
    p;              Illegal! P1 may not call its parent
    p2              But may call its sibling tail-recursively
  endproc;

  procedure p2;
    p1;             Illegal! p2 may not call its sibling non-tail-recursively
    p2              But may call itself tail-recursively
  endproc;

  p1; p1            P may call p1 (a child) non tail-recursively as well as tail-recursively
endproc;
```

- Expressions

  An expression represents a value which can be passed as an **in** parameter to procedures and assigned to variables.

  The **if** expression evaluates its conditional expressions in sequential order. The value of the **if** expression is the value of the expression corresponding to the first true boolean expression. If no boolean expression evaluates to true and there exists an **else** clause, the value of the **if** expression is the value of the **else** expression. Otherwise the **if** expression aborts. The projection function $\#I_c\ E$ will return the component number I_c of the tuple expression E. Tuple components are numbered consecutively starting from 1.

- Commands

  Standard commands such as sequencing (;), assignment (:=) exists in Lunsen and have their usual meaning. Note that the execution of the assignment command is non-atomic: the evaluation of the right-hand side is separate from storing the result into the left-hand side.

  The **if** command is similar to the **if** expression. The execution of the **when** command is suspended until one of its communication events can take place; then the corresponding command is executed. The τ event will take place spontaneously, without having to wait for communication with another process. Thus the **when** command may introduce explicit non-determinism in a program. As an example:

```
port synch:boolean;

when
  tau -> p1
| synch!true -> p2
endwhen
```

  The execution of the **when** command in the example can proceed in two ways: either through the spontaneous τ event, in which case p1 is executed, or by sending on the synch port, in which case p2 is executed; the last alternative requires that another process is ready to communicate on the synch port.

  The **par** command creates two processes that execute in parallel. Given

```
program Pvar;
  type int4 = ordered {1,2,3,4};
  var v:int4;

  v := 1;
  v := succ(v) par v := succ(v)
endprog;
```

  the value of v may become either 2 or 3, i.e. the execution of parallel commands is finely grained. This is due to the semantics of the assignment command, which is executed non-atomically.

3 CCS and Typed CCS

Two versions of CCS (Calculus of Concurrent Systems) are used in this paper. The first one is the *basic* calculus. The second version is called TCCS (Typed CCS), and is closely related to the *value-passing calculus* in Chapter 2.8 of [Mil89]. TCCS extends basic CCS in that action prefixes are

S::=	$K\ E$ **where** $D;\ldots;D$	TCCS program
D::=	**type** $t = T$	Type definition
	port $p : t$	Port definition
	$K\ \tilde{v} : t \stackrel{\text{def}}{=} A$	Agent definition
T::=	$\{c,\ldots,c\}$	Enumerated type
	ordered $\{c,\ldots,c\}$	Ordered enumerated type
	$t * \ldots * t$	Tuple type
A::=	**nil**	The **nil** agent
	$K\ E$	Agent identifier expression
	$P.A$	Prefixing
	$A + A$	Choice
	$A \mid A$	Parallel composition
	$A\backslash\{p,\ldots,p\}$	Restriction
	$A[p/p,\ldots,p/p]$	Relabeling
	if E **then** A **else** A	If agent
E::=	c	Constant
	v	Variable
	$(E,\ldots,E)$	Tuple expression
	$\#c\ E$	Tuple access
	$E = E$	Equality
	$E \leq E$	Less than or equal
	not $E \mid E$ **and** $E \mid E$ **or** E	Boolean functions
P::=	τ	The τ action
	$p!E$	Send value E on port p
	$p?\tilde{v}$	Receive any value on port p
	$p?{=}E$	Receive value E on port p

Table 2: The syntax of TCCS

explicitly parameterized on data values. TCCS acts as an intermediary language in the translation from Lunsen to CCS. This enables us to separate the concerns of flow control in Lunsen from concerns related to typing and value-passing.

We first briefly review basic CCS. Let $A, B, \ldots$ range over *agents*, and let $a, b, \ldots$ range over port names. The complementary port of a is denoted by $\overline{a}$. Two agents can communicate if one of them has a port named a and the other a port named $\overline{a}$. We extend the set of port names with the silent action τ to form the set of CCS actions. We let α range over the set of actions. The operators used in the basic calculus are: prefixing ($\alpha.A$), choice ($A + B$), parallel composition ($A \mid B$), restriction ($A\backslash L$) on a set of ports L, and relabeling ($A[f]$) where f is a function that relabels the ports in A. As usual we write $\sum_{i=1}^{n} A_i$ for $A_1 + \cdots + A_n$.

For TCCS we presuppose a set of *types* ranged over by t, a set of *agent identifiers* ranged over by K, a set of *port names* ranged over by p, q, a set of TCCS *variables* ranged over by v, and a set of *constant values* ranged over by c. We write $\tilde{v}$ for a (possibly empty) tuple of variables $(v_1, \ldots, v_n)$ and similarly $\tilde{c}$ for a tuple of constants.

The syntax for TCCS is defined in Table 2 using the BNF dialect. There are two predefined types: **boolean** with the two elements **true** and **false**, and **unit** with the only element (). This element may be omitted in expressions; for example $p!.A$ is short for $p!().A$.

A TCCS variable v is *bound* in the input prefix $p?v.A$; more generally $p?\tilde{v}.A$ binds all variables in $\tilde{v}$. Similarly an agent definition $K\tilde{v} : t \stackrel{\text{def}}{=} A$ binds the variables $\tilde{v}$ in A. We only consider TCCS programs which are well typed and where all variables occur bound. Thus, in a TCCS program each agent identifier, port name, TCCS variable, and constant value is associated with a unique type as given in a TCCS definition (the type of a variable is considered the same as the type of the port or identifier where it is bound). In the following we write $\mathit{typeof}(X)$ for the type associated with such an object (or a tuple of such objects) X. We also write $A[\tilde{c}/\tilde{v}]$ to mean the TCCS agent gained by substituting each free occurrence of v_i by c_i.

The meaning of TCCS is defined by a function $\mathcal{T}[\![C]\!]$, which maps a TCCS construction C into basic CCS. In this definition we do not distinguish between a closed expression (an expression without variables) and the constant value it denotes when the operators "=", "**and**" etc. are given the obvious interpretations. We further assume that for each TCCS port p and constant c of the same type there is a basic CCS action p_c.

The first clause in the definition of $\mathcal{T}[\![]\!]$ is:

$$\mathcal{T}[\![K\ E\ \textbf{where}\ D_1; \ldots; D_n]\!] = \mathcal{T}[\![K\ E]\!]$$

where the right hand side is to be interpreted with respect to the basic CCS agent identifier definitions introduced by $D_1, \ldots D_n$ as follows. TCCS type definitions and TCCS port definitions do not result in any basic CCS identifier definitions. Each TCCS agent identifier definition $K\ \tilde{v} : t \stackrel{\text{def}}{=} A$ yields the set of CCS agent identifier definitions: $K_{\tilde{c}} \stackrel{\text{def}}{=} \mathcal{T}[\![A[\tilde{c}/\tilde{v}]]\!]$ forall $\tilde{c}$ of type t.

The translation of a TCCS agent is defined in Table 3. Note in particular the *determined* input construct $p?\!\!=\!E.A$. This results in a TCCS agent which can only accept a particular value (as determined by E) on port p; such a construct turns out to be useful in defining the semantics of Lunsen arrays.

A	$\mathcal{T}[\![A]\!]$
nil	**nil**
$K\ E$	K_E
$\tau.A$	$\tau.\mathcal{T}[\![A]\!]$
$p?\tilde{v}.A$	$\sum_{\mathit{typeof}(\tilde{c})=\mathit{typeof}(p)} p_{\tilde{c}}.\mathcal{T}[\![A[\tilde{c}/\tilde{v}]]\!]$
$p!E.A$	$\overline{p_E}.\mathcal{T}[\![\mathrm{A}]\!]$
$p?\!\!=\!E.A$	$p_E.\mathcal{T}[\![\mathrm{A}]\!]$
$A_1 + A_2$	$\mathcal{T}[\![A_1]\!] + \mathcal{T}[\![A_2]\!]$
$A_1 \mid A_2$	$\mathcal{T}[\![A_1]\!] \mid \mathcal{T}[\![A_2]\!]$
$A \backslash L$	$\mathcal{T}[\![A]\!] \backslash \{p_c : p \in L, c \in \mathit{typeof}(p)\}$
$A[f]$	$\mathcal{T}[\![A]\!][f']$ where $f'(p_c) = f(p)_c$
if true then A_1 **else** A_2	$\mathcal{T}[\![A_1]\!]$
if false then A_1 **else** A_2	$\mathcal{T}[\![A_2]\!]$

Table 3: Translation of TCCS agents into CCS agents

For example, $\mathcal{T}[\![K(x : \textbf{boolean}) \stackrel{\text{def}}{=} p?y.q!y.r?\!\!=\!x.\textbf{nil}]\!]$ yields the set of basic CCS agents identifier definitions

$$\begin{cases} K_{true} \stackrel{\text{def}}{=} p_{true}.\overline{q_{true}}.r_{true}.\textbf{nil} + p_{false}.\overline{q_{false}}.r_{true}.\textbf{nil} \\ K_{false} \stackrel{\text{def}}{=} p_{true}.\overline{q_{true}}.r_{false}.\textbf{nil} + p_{false}.\overline{q_{false}}.r_{false}.\textbf{nil} \end{cases}$$

4 A Formal Semantics for Lunsen

4.1 Combinators

When translating the Lunsen constructions into TCCS, we will use a number of basic combinators similar to the ones defined in Milner's book [Mil89]. The combinator *Before* will be used to model sequential composition of two agents; the first agent must signal that it has finished "running" by using the combinator *Done* before the other agent can start "running". The *Par* combinator is used to model two processes running in parallel.

Each expression will return its value by using the combinator *Result(value)*. Such a value can be bound to a TCCS variable in a process agent through the combinator *Into*, as in *expr* $Into(x)\,(ag)$: the TCCS variable x will here be bound to the value of *expr* in the TCCS process agent *ag*. The combinator *Into_l* is a variant of *Into*, allowing a list of variables to be bound to a list of expressions in an agent body. We use the syntax *hd* :: *tail* for a list consisting of the head *hd* and tail list *tl*. [] will denote the empty list. If we are certain that a list consists of a fixed number of elements, say e_1 and e_2, this is written as $[e_1, e_2]$. The combinators are defined in Table 4.

$$\begin{aligned}
Done &= done!.\mathbf{nil} \\
P\ Before\ Q &= (P[b/done]\,|\,b?.Q)\backslash\{b\} \\
P\ Par\ Q &= (P[d1/done]\,|\,Q[d2/done]\,|\,d1?.d2?.Done)\backslash\{d1, d2\} \\
Result(v) &= result!v.\mathbf{nil} \\
E\ Into(x)\,(A) &= (E[i/result]\,|\,i?x.A)\backslash\{i\} \\
\\
[]\ Into_l([])\,(A) &= A \\
E :: RestE\ Into_l(x :: Restx)\,(A) &= E\ Into(x)\,(RestE\ Into_l(Restx)\,(A))
\end{aligned}$$

Table 4: The basic combinators

4.2 Variables

A non-array variable v of type T is translated to the TCCS agent

$$Reg_v(y : T) \stackrel{\text{def}}{=} put_v?(x).Reg_v(x) + get_v!(y).Reg_v(y)$$

A value can be stored in the variable v by sending the new value on the port put_v. Reading of values from v is accomplished by receiving the current value from the port get_v. An array variable v of type **array** $[T_{index}]$ **of** T_{store} will be represented as the family of agents

$$\forall i \epsilon T_{index} : Reg_{v_i}(y : T_{store}) \stackrel{\text{def}}{=} get_v!(i, y).Reg_{v_i}(y) + \sum_{x \in T_{store}} put_v?{=}(i, x).Reg_{v_i}(x)$$

We will use $store(a)$ to denote the type of elements in an array a (T_{store} in the example), and $index(a)$ to denote the index type of the array (T_{index} in the example). So, reading the array element $a[2]$ is accomplished in the TCCS agent

$$\sum_{z \in store(a)} get_a?{=}(2, z).B$$

When we present the formal semantics of Lunsen below, we will for simplicity assume that no variables are of an array type (except in explicit array access expressions and commands). Each rule involving

registers should thus be extended with a case where the variable is an array, in which case a set of register agents Reg_{v_i} should be used instead of the single agent Reg_v.

4.3 Access and Restriction sorts

An access sort $\mathcal{ACC}$ for a declaration in Lunsen is the set of TCCS port names by which it it possible for other Lunsen declarations and commands to interact with that declaration. For a variable v, the access sort will consist of put_v and get_v.

The restriction sort $\mathcal{RAC}$ for an declaration contains the port names of that declaration which should not be accessible to an external observer of the program. It will be identical to $\mathcal{ACC}$ except when visible variables or ports occur in a program. The function $\mathcal{RAC}$ is presented in Table 5.

$$\begin{aligned}
\mathcal{RAC}(\textbf{var } I_v{:}I_t = E;) &= \{put_{I_v}, get_{I_v}\} \\
\mathcal{RAC}(\textbf{var } I_v{:}\textbf{read } I_t = E;) &= \{put_{I_v}\} \\
\mathcal{RAC}(\textbf{var } I_v{:}\textbf{write } I_t = E;) &= \{get_{I_v}\} \\
\mathcal{RAC}(\textbf{var } I_v{:}\textbf{readwrite } I_t = E;) &= \emptyset \\
\mathcal{RAC}(\textbf{procedure } I_p[(I_v{:}A\ I_t, \ldots, I_v{:}A\ I_t)]\ D\ C\ \textbf{endproc};) &= \emptyset \\
\mathcal{RAC}(\textbf{port } I_m{:}\textbf{visible } I_t;) &= \emptyset \\
\mathcal{RAC}(\textbf{port } I_m{:}I_t;) &= \{I_m\} \\
\mathcal{RAC}(D_1\ D_2) &= \mathcal{RAC}(D_1) \bigcup \mathcal{RAC}(D_2)
\end{aligned}$$

Table 5: The function $\mathcal{RAC}$ for computing restriction sorts

4.4 The translation

We define the translation $\mathcal{L}[\![]\!]$ from commands in Lunsen to TCCS agents in separate tables for declarations, expressions, and commands. Constructions not generating any TCCS "code" are not listed in the tables. These for example include definitions of types. First we show the translation of the **program** statement:

$$\mathcal{L}[\![\textbf{program } I_s;\ D\ C\ \textbf{endprog};]\!] = I_s, \text{ where } I_s \stackrel{\text{def}}{=} (\mathcal{L}[\![D]\!] \mid (\mathcal{L}[\![C]\!]\ \textit{Before}\ \textbf{nil})) \backslash \mathcal{RAC}_{\mathcal{D}}$$

Translation of declarations is listed in Table 6. In the rule for procedure translation **(1)**, $x_1^i, \ldots, x_n^i$ match the subset of formal parameters to the procedure $(I_{v1}^i, \ldots, I_{vn}^i)$ that are **in** or **inout** parameters. Similarly, $x_1^o, \ldots, x_n^o$ match the subset of formal parameters to the procedure $(I_{v1}^o, \ldots, I_{vn}^o)$ that are **out** or **inout** parameters. The intuition behind the translation of a procedure is that the **in** and **inout** actual parameter values $x_1^i, \ldots, x_n^i$ are supplied as parameters to the agent identifier I_p in the translation of the call command. Temporary copies of all parameters are made and the actual values of the parameters are stored in the registers $Reg_{I_{vi}}$. When the execution of the translated command $\mathcal{L}[\![C]\!]$ in the procedure has finished, the values of the temporary registers and returned as a TCCS tuple expression through the use of the *Result* combinator.

The translation of Lunsen expressions into TCCS agents is shown in Table 7 and translation of commands in Table 8.

In the translation of a non-tail-recursive procedure call **(2)** in Table 8, we assume that **in** and **inout** parameters are denoted by E_1^i through E_n^i and the **out** and **inout** parameters by E_1^o through E_n^o. Note that the actual parameters corresponding to **out** and **inout** formal parameters must be variables (I_v). In the translation we evaluate the **in** expressions, continue with the execution of the

translated procedure I_p which will as previously described end its execution by returning the values of the "out" parameters. We then store back those values in the variables (I_v) passed as actual parameters to the procedure call.

The translation of the tail-recursive call (**3**) is more straightforward because we do not have to store back the results of the procedure call into the actual parameters.

Our semantics for scoping of procedures and procedure calling follow the informal rules presented in 2. Since the informal rules are nonproblematic we omit a formal specification here. Likewise, we do not present the function which determines if a procedure call is a tail-recursive one.

We will examplify the translation rules by informally discussing the result of applying them to the `Pvar` program in Section 2, in the introduction to Lunsen Commands. The resulting CCS specification will consist of two agents in parallel, the first representing the variable `v` in the form of a register agent, the second the execution flow in the program. The agent representing the execution flow starts by assigning `1` to `v`, then splits into two new agents composed by the CCS parallel operator. Each of these two agents will perform two atomic actions: first compute the value `succ(v)`, and then assign that value to `v`. Since the translation of the assignment command consists of two atomic actions, the resulting value assigned to `v` can be either `2` or `3`:

- if the computation of `succ(v)` by the two agents are performed directly after each other, the result will be `2`
- if one of these agents gets to both compute `succ(v)` and then store that value in `v` before the other agent computes `succ(v)`, the result will be `3`

After both agents have performed their assignment to `v`, the execution of the program terminates.

5 Implementation details

As mentioned in the introduction, one aim is to use the output from the Lunsen compiler in analyses performed by the Concurrency Workbench. This means that care has to be taken to ensure that all generated agents are finite-state. In this section we briefly comment on some problems in this respect.

5.1 Problems

The basic Combinators from Section 4.1 may introduce agents which are syntactically non-finite-state, although they are equivalent with finite state agents.

For example, the TCCS agent definition $A \stackrel{\mathrm{def}}{=} p.Done\ Before\ A$ is translated into the CSS agent $A = (p.\overline{done}.nil[b/done]|b.A)\backslash\{b\}$. The expansion of this agent using the *expansion law* yields the agent definition $A = p.(nil[b/done]|A)\backslash\{b\}$. The CWB could here reduce the agent expression to $p.A$ using a rule equating $(nil[Z]|X)\backslash Y$ to $X\backslash Y$. Instead, CWB chooses the strategy of continuing to expand the $(nil[b/done]|A)\backslash\{b\}$ expression, resulting in the expression $p.(nil[b/done]|p.(nil[b/done]|A)\backslash\{b\})\backslash\{b\}$. Obviously, the expansion process will never terminate.

To avoid these problems the basic combinators are implemented in the following way:

- P *Before* Q is implemented as the agent obtained by substituting each occurrence of *done*!.**nil** in P with Q.
- E *Into*$(x)\,(A)$ is implemented as: the agent obtained by substituting each occurrence of a *result*!*v*.**nil** expression in E with $A\{v/x\}$ (A where we substitute v for x).

$$\mathcal{L}[\![D_1\ D_2]\!] \quad = \quad \mathcal{L}[\![D_1]\!] \,|\, \mathcal{L}[\![D_2]\!]$$

$$\mathcal{L}[\![\textbf{var } I_v{:}[G]\ I_t = E;]\!] \quad = \quad Reg_{I_v}(\mathcal{L}[\![E]\!]) \text{ where } E \text{ is a constant expression}$$

$$\mathcal{L}[\![\textbf{procedure } I_p(I_{v1}{:}A\ I_{t1},\ldots,I_{vn}{:}A\ I_{tn});\ D\ C\ \textbf{endproc};]\!] = \textbf{nil}$$

We define the TCCS agent

$$I_p((x^i_1,\ldots,x^i_n)) \stackrel{\text{def}}{=} (\underbrace{Reg_{I_{v1}}(_) \mid \ldots \mid Reg_{I_{vn}}(_)}_{\text{Temp. copies of parms.}} \mid$$
$$\underbrace{put^i_{I_{v1}}!x^i_1.\ldots.put^i_{I_{vn}}!x^i_n}_{\text{In(out) parms.}}.\ \mathcal{L}[\![C]\!]\ Before \qquad (1)$$
$$\underbrace{get^o_{I_{v1}}?x^o_1.\ldots.get^o_{I_{vn}}?x^o_n}_{\text{(in)Out parms.}}.Result((x^o_1,\ldots,x^o_n)) \mid$$
$$\mathcal{L}[\![D]\!]) \backslash ACC_{\mathcal{D}} \bigcup ACC_{I_{v1}} \bigcup \ldots \bigcup ACC_{I_{vn}}$$

Table 6: Translation of declarations (D)

$$\mathcal{L}[\![I_c]\!] \quad = \quad Result(I_c)$$

$$\mathcal{L}[\![I_v]\!] \quad = \quad get_{I_v}?x.Result(x)$$

$$\mathcal{L}[\![I_a[E_1,\ldots,E_n]]\!] \quad = \quad [\mathcal{L}[\![E_1]\!],\ldots,\mathcal{L}[\![E_n]\!]]\ Into_l([x_1,\ldots,x_n])\ (\textstyle\sum_{y \in store(I_a)}(get_{I_a}?{=}((x_1,\ldots,x_n),y).Result(y)))$$

$$\mathcal{L}[\![(E_1,\ldots,E_n)]\!] \quad = \quad [\mathcal{L}[\![E_1]\!],\ldots,\mathcal{L}[\![E_n]\!]]\ Into_l([x_1,\ldots,x_n])\ Result((x_1,\ldots,x_n))$$

$$\mathcal{L}[\![\#I_c\ E]\!] \quad = \quad \mathcal{L}[\![E]\!]\ Into(x)\ (\#\ I_c\ x)$$

$$\mathcal{L}[\![\textbf{if } E_{b1} \rightarrow E_1 \mid \ldots \mid E_{bn} \rightarrow E_n \textbf{ endif}]\!] \quad = \quad \mathcal{L}[\![E_{b1}]\!]\ Into(x_1)\ (\textbf{if } x_1 \textbf{ then } \mathcal{L}[\![E_1]\!] \textbf{ else } \ldots \textbf{ else } \mathcal{L}[\![E_{bn}]\!]\ Into(x_n)\ (\textbf{if } x_n \textbf{ then } \mathcal{L}[\![E_n]\!] \textbf{ else nil}))$$

$$\mathcal{L}[\![\textbf{if } E_{b1} \rightarrow E_1 \mid \ldots \mid E_{bn} \rightarrow E_n \textbf{ else } E \textbf{ endif}]\!] \quad = \quad \mathcal{L}[\![E_{b1}]\!]\ Into(x_1)\ (\textbf{if } x_1 \textbf{ then } \mathcal{L}[\![E_1]\!] \textbf{ else } \ldots \textbf{ else } \mathcal{L}[\![E_{bn}]\!]\ Into(x_n)\ (\textbf{if } x_n \textbf{ then } \mathcal{L}[\![E_n]\!] \textbf{ else } \mathcal{L}[\![E]\!]))$$

$$\mathcal{L}[\![E_1 = E_2]\!] \quad = \quad [\mathcal{L}[\![E_1]\!],\mathcal{L}[\![E_2]\!]]\ Into_l([x_1,x_2])\ (Result(x_1 = x_2))$$

$$\mathcal{L}[\![E_1 \le E_2]\!] \quad = \quad [\mathcal{L}[\![E_1]\!],\mathcal{L}[\![E_2]\!]]\ Into_l([x_1,x_2])\ (Result(x_1 \le x_2))$$

$$\mathcal{L}[\![\text{succ } E]\!] \quad = \quad \mathcal{L}[\![E]\!]\ Into(x)\ (Result(\text{succ}(x)))$$

$$\mathcal{L}[\![\textbf{pred } E]\!] \quad = \quad \mathcal{L}[\![E]\!]\ Into(x)\ (Result(\textbf{pred}(x)))$$

$$\mathcal{L}[\![\textbf{not } E]\!] \quad = \quad \mathcal{L}[\![E]\!]\ Into(x)\ (Result(\textbf{not}(x)))$$

$$\mathcal{L}[\![E_1 \textbf{ and } E_2]\!] \quad = \quad \mathcal{L}[\![E_1]\!]\ Into(x_1)\ (\textbf{if } x_1 = \textbf{false then } Result(\textbf{false}) \textbf{ else } \mathcal{L}[\![E_2]\!])$$

$$\mathcal{L}[\![E_1 \textbf{ or } E_2]\!] \quad = \quad \mathcal{L}[\![E_1]\!]\ Into(x_1)\ (\textbf{if } x_1 = \textbf{true then } Result(\textbf{true}) \textbf{ else } \mathcal{L}[\![E_2]\!])$$

Table 7: Translation of expressions (E)

$$\mathcal{L}\llbracket \textbf{begin } C \textbf{ end} \rrbracket = \mathcal{L}\llbracket C \rrbracket$$

$$\mathcal{L}\llbracket \textbf{skip} \rrbracket = \mathit{Done}$$

$$\mathcal{L}\llbracket C_1 \; ; \; C_2 \rrbracket = \mathcal{L}\llbracket C_1 \rrbracket \; \mathit{Before} \; \mathcal{L}\llbracket C_2 \rrbracket$$

$$\mathcal{L}\llbracket \mathrm{C} \textbf{ par } C \rrbracket = \mathcal{L}\llbracket C_1 \rrbracket \; \mathit{Par} \; \mathcal{L}\llbracket C_2 \rrbracket$$

$$\mathcal{L}\llbracket I_v := E \rrbracket = \mathcal{L}\llbracket E \rrbracket \; \mathit{Into}(x) \, (\mathit{put}_{I_v}!x.\mathit{Done})$$

$$\begin{aligned}\mathcal{L}\llbracket I_a[E_1,\ldots,E_n] := E_f \rrbracket = \; & [\mathcal{L}\llbracket E_f \rrbracket, \mathcal{L}\llbracket E_1 \rrbracket, \ldots, \mathcal{L}\llbracket E_n \rrbracket] \; \mathit{Into_l}([x_f, x_1, \ldots, x_n]) \\ & (\mathit{put}_{I_a}!((x_1,\ldots,x_n), x_f).\mathit{Done})\end{aligned}$$

Non tail-recursive procedure call (**2**):

$$\begin{aligned}\mathcal{L}\llbracket I_p[(E_1,\ldots,E_n)] \rrbracket = \; & [\mathcal{L}\llbracket E_1^i \rrbracket, \ldots, \mathcal{L}\llbracket E_n^i \rrbracket] \; \mathit{Into_l}([x_1^i, \ldots, x_n^i]) \\ & I_p(x_1^i,\ldots,x_n^i) \; \mathit{Into}(y_1^o,\ldots,y_n^o) \; (\mathit{put}_{E_1^o}!y_1^o.\ldots.\mathit{put}_{E_n^o}!y_n^o.\mathit{Done})\end{aligned}$$

Tail recursive procedure call (**3**):

$$\mathcal{L}\llbracket I_p[(E_1,\ldots,E_n)] \rrbracket = [\mathcal{L}\llbracket E_1^i \rrbracket, \ldots, \mathcal{L}\llbracket E_n^i \rrbracket] \; \mathit{Into_l}([x_1^i, \ldots, x_n^i]) \, (I_p(x_1^i,\ldots,x_n^i))$$

$$\begin{aligned}\mathcal{L}\llbracket \textbf{if } E_{b1} \rightarrow C_1 \mid \ldots \mid E_{bn} \rightarrow C_n \textbf{ endif} \rrbracket = \; & \mathcal{L}\llbracket E_{b1} \rrbracket \; \mathit{Into}(x_1) \; (\textbf{if } x_1 \textbf{ then } \mathcal{L}\llbracket C_1 \rrbracket \textbf{ else } \ldots \\ & \textbf{else } \mathcal{L}\llbracket E_{bn} \rrbracket \; \mathit{Into}(x_n) \; (\textbf{if } x_n \textbf{ then } \mathcal{L}\llbracket C_n \rrbracket \textbf{ else nil}))\end{aligned}$$

$$\begin{aligned}\mathcal{L}\llbracket \textbf{if } E_{b1} \rightarrow C_1 \mid \ldots \mid E_{bn} \rightarrow C_n \textbf{ else } C \textbf{ endif} \rrbracket = \; & \mathcal{L}\llbracket E_{b1} \rrbracket \; \mathit{Into}(x_1) \; (\textbf{if } x_1 \textbf{ then } \mathcal{L}\llbracket C_1 \rrbracket \textbf{ else } \ldots \\ & \textbf{else } \mathcal{L}\llbracket E_{bn} \rrbracket \; \mathit{Into}(x_n) \; (\textbf{if } x_n \textbf{ then } \mathcal{L}\llbracket E_n \rrbracket \textbf{ else } \mathcal{L}\llbracket C \rrbracket))\end{aligned}$$

$$\mathcal{L}\llbracket \textbf{when } P_1 \rightarrow C_1 \mid \ldots \mid P_n \rightarrow C_n \textbf{ endwhen} \rrbracket = \mathcal{L}\llbracket P_1 \rrbracket \; \mathit{Before} \; \mathcal{L}\llbracket C_1 \rrbracket + \ldots + \mathcal{L}\llbracket P_n \rrbracket \; \mathit{Before} \; \mathcal{L}\llbracket C_n \rrbracket$$

$$\begin{aligned}\mathcal{L}\llbracket \textbf{while } E \textbf{ do } C \textbf{ endwhile} \rrbracket = \; & W, \textbf{ where } W \stackrel{\text{def}}{=} \mathcal{L}\llbracket E \rrbracket \; \mathit{Into}(x) \\ & (\textbf{if } x \textbf{ then } \mathcal{L}\llbracket C \rrbracket \; \mathit{Before} \; W \textbf{ else } \mathit{Done})\end{aligned}$$

$$\mathcal{L}\llbracket I_m \; ! \; E \rrbracket = \mathcal{L}\llbracket E \rrbracket \; \mathit{Into}(x) \, (I_m!x.\mathit{Done})$$

$$\mathcal{L}\llbracket \tau \rrbracket = \tau.\mathit{Done}$$

$$\mathcal{L}\llbracket I_m \; ? \; I_v \rrbracket = I_m?x.\mathit{put}_{I_v}!x.\mathit{Done}$$

Table 8: Translation of commands (C and P)

- A *Par* B is defined as $p_1!.p_2!p_1?.p_2?.Done$, where p_1 and p_2 are ports unique to the program. Two new agents are also added, $A' \stackrel{\text{def}}{=} p_1?.A$ *Before* $p_1!.A'$ and $B' \stackrel{\text{def}}{=} p_2?.B$ *Before* $p_2!.B'$.

An example: assuming $A \stackrel{\text{def}}{=} a.A + done!.\mathbf{nil}$ and $B \stackrel{\text{def}}{=} b.B$, then A *Before* B is equivalent to A' where A' is a new agent: $A' \stackrel{\text{def}}{=} a.A + b.B$.

The compiler furthermore "lifts up" all parallel compositions (and restrictions) into the main program agent (the one named in the program statement). In this process unique names for restricted objects are created. Thus the main program agent will appear:

$$Spec \stackrel{\text{def}}{=} (Ag_1 | \dots | Ag_n) \backslash \{p_1, \dots, p_n\}$$

and the TCCS agents Ag_i will contain only + (choice), . (prefixing) and agent identifiers (K's).

5.2 Optimizations

We also do some further optimizations for the sake of efficiency of the analysis of the generated CCS agents in the CWB:

- Lunsen variables which can be accessed by at most one concurrent process do not need associated Register agents (cf. Section 4.2). As an example,

```
procedure p;
  var v:boolean = false;
  q!v
endproc;
```

  The variable v in the example can only be used in the procedure p and since the procedure contains no **par** command, the variable cannot be accessed by multiple processes concurrently. Therefore we need not actually use a register agent for v but may encode it directly in the TCCS agent corresponding to the translation of procedure p.

- Some useful algebraic manipulations: simplify $A + A$ into A, $A + \mathbf{nil}$ into A and $X.\tau.Y$ into $X.Y$.

6 An Example

In this section we will present an example of a distributed algorithm which is normally, and perhaps most clearly, formulated in an imperative language. Our example is an algorithm for mutual exclusion due to Peterson [PS85]. The algorithm will be defined in Lunsen and formally verified with the Concurrency Workbench. In the following we assume the reader to have some familiarity with the modal logic supported by the Workbench. An introduction to this logic and its use for verifying mutual exclusion algorithms can be found in a recent article by Walker [Wal89]; we will use the same algorithm and correctness criteria.

First we will formulate Peterson's mutual exclusion algorithm in Lunsen, assuming two concurrent processes competing to enter their critical sections. Then we will minimize the resulting CCS agents w.r.t. observation equivalence using the Workbench. Finally we will determine whether the algorithm meets the safety and liveness demands. This will be done by checking if the minimized agents satisfy a pair of propositions formulated in HML (Hennessy-Milner logic). These checks are done automatically by the Workbench.

In the formulation of the algorithm in Lunsen, we will use port communications to signal that a process requests to enter, enters and leaves its critical section (*req!i*, *enter!i* and *exit!i*, where i is the name of one of the two concurrent processes). These are the only actions which will be visible to an observer of the program. Peterson's algorithm as formulated in Lunsen is presented in Figure 1. The state graph of the Lunsen program after translation into basic CCS, and minimization w.r.t. observation equivalence by the Concurrency Workbench is displayed in Figure 2.

```
program Peterson;
  type int2 = {1, 2};
  type b_arr = array[int2] of boolean;

  var b:b_arr = {false, false};
  var k:int2;

  -- Ports visible to an observer
  port enter :visible int2;
  port exit  :visible int2;
  port req   :visible int2;

  procedure p(i:in int2, j:in int2);
    while true do
      b[i] := true; req!i;
      k := j;
      while (b[j] and (k = j)) do skip endwhile;
      enter!i;        -- Enter critical region
      exit!i;         -- Exit critical region
      b[i] := false
    endwhile
  endproc;

  p(1,2) par p(2,1) -- Start two processes running in parallel
endprog; -- Peterson
```

Figure 1: Peterson's algorithm

The mutual exclusion property we wish the algorithms to preserve can be formulated:

$$Mutex = \nu Z.(\neg(< exit!1 > true \wedge < exit!2 > true) \wedge [K]Z),$$

that is both processes should not be able to leave their critical sections at the same time, which implies that not both processes *are* in their critical section. $[K]Z$ in the previous formula is an abbreviation for $[K]Z \equiv \wedge_{a \in K}[a]Z$, where K is a set of actions. We will use

$$K = \{enter!1, enter!2, exit!1, exit!2, req!1, req!2\}$$

for verifying the mutual exclusion property. The liveness property can be formulated:

$$Live \equiv Live_1 \wedge Live_2$$

where

$$Live_i \equiv \nu Z.([req!i]\mu Y.(< exit!i > true \vee [K]Y) \wedge [K]Z)$$

This formula expresses that if a process i has requested execution of its critical section by *req!i* then there shall be no infinite path of actions not consisting of an *enter!i* action in the corresponding transition system. As pointed out by Walker this is just one interpretation of the liveness properties of the algorithm. When we check the safety property we find that $Peterson \models Mutex$ as hoped, i.e. two processes cannot be in their critical sections at the same time. We also find that it satisfies *Live*.

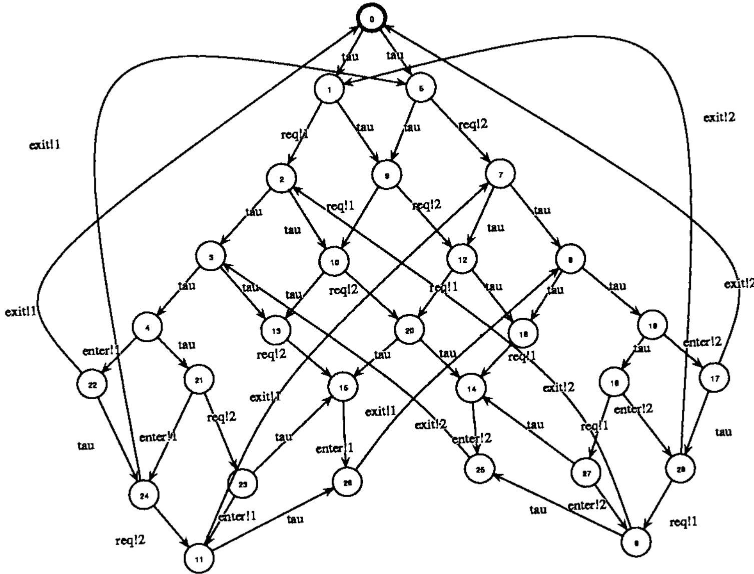

Figure 2: The graph of the minimized Peterson algorithm

7 Conclusions

We have presented an imperative language with constructs for sequential programming as well as constructs for parallel execution of processes, and an automated tool for analysis of concurrent programs written in the language. A few restrictions were imposed on the language to ensure that the semantics of a program is finite-state: e.g.: the types of variables must contain only finitely many elements, and that procedures cannot call each other recursively in an arbitrary manner. Many concurrent algorithms can be naturally described in this language. Our experience includes several other mutex algorithms (e.g. in [Lam86]) and two versions of the Alternating-Bit protocol [BSW69].

We have presented a formal semantics for the language, which represents the execution of a program on a multiprocessor with shared memory, without assuming that e.g. assignment statements are atomic. This means that the results of analysis are valid for a direct implementation of the algorithm.

The semantics was given through a translation to CCS. The efficiency of the resulting "code" is on par with the results of hand translations. The advantage of this is that CWB and associated systems can be used to carry out different forms of analysis. A disadvantage of the present implementation is that properties of programs must be formulated in terms of communication events and not in terms of predicates over the state of the program.

Desirable extensions of the language include a module concept to structure large programs and enable several instantiations of processes. This leads to problems with conflicting access sorts: a newly instantiated process needs to know on which ports it should communicate. One way to achieve this is to use CCS restriction and relabeling operators, but if these are used recursively, the resulting agents will not in general be finite state. Another way is to define the translation into the π-calculus [MPW89a, MPW89b] rather than into CCS (in the π-calculus an agent is explicitly parametrized on its free port names).

An interesting topic of further research is to give Lunsen a more direct semantics in terms of states and transitions between states, and to compare this semantics with the translational semantics in this paper.

References

[AZ81] E. Astesiano and E. Zucca. Semantics of CSP via translation into CCS. In *Mathematical Foundations of Computer Science*, volume 118 of *LNCS*, pages 172–182. Springer Verlag, 1981.

[BdSV88] G. Boudol, R. de Simone, and D. Vergamini. Experiment with Auto and Autograph on a simple case sliding window protocol. Technical Report 870, Inria, July 1988.

[BSW69] K. Bartlett, R. Scantlebury, and P. Wilkinson. A note on reliable full-duplex transmissions over half duplex lines. *Communications of the ACM*, 2(5):260–261, 1969.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specification. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CPS89] R. Cleaveland, J. Parrow, and B. Steffen. A semantics-based verification tool for finite-state systems. In *Protocol Specification, Testing, and Verification IX*, pages 287–302, 1989. North-Holland.

[GLZ89] J.C. Godskesen, K.G. Larsen, and M. Zeeberg. TAV users manual. In *Proc. Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, 1989.

[HL83] M. Hennessy and W. Li. Translating a subset of Ada into CCS. In D. Bjoerner, editor, *Formal Description of Programming Concepts II*, pages 227–249, Amsterdam, 1983. North-Holland.

[Lam86] L. Lamport. The mutual exclusion problem part II – statement and solutions. *Journal of the ACM*, 33(2), 1986.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[MPW89a] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I. Technical Report ECS-LFCS-89-85, Department of Computer Science, University of Edinburgh, 1989.

[MPW89b] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part II. Technical Report ECS-LFCS-89-86, Department of Computer Science, University of Edinburgh, 1989.

[PS85] J.L. Peterson and A. Silberschatz. *Operating System Concepts.* Addison-Wesley, 1985.

[RRSV87] J. Richier, C. Rodriguez, J. Sifakis, and J. Voiron. Verification in XESAR of the sliding window protocol. In *Protocol Specification, Testing, and Verification VII.* North-Holland, 1987.

[SS86] S.A. Smolka and R.E. Strom. A CCS semantics for NIL. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 347–368, Amsterdam, 1986. North-Holland.

[Wal89] D.J. Walker. Automated analysis of mutual exclusion algorithms using CCS. *Formal Aspects of Computing*, 1:273–292, 1989.