A semantics driven temporal verification system

G. D. Gough and H. Barringer*
Department of Computer Science
University of Manchester
Oxford Road
Manchester, M13 9PL

Abstract

We present an overview of SMG, a *generic* state machine generator, which interfaces to various temporal logic model checkers and provides a practical generic temporal verification system. SMG transforms programs written in user-definable languages to suitable finite state models. thus enabling fast verification of temporal properties of the input program. It can be applied, in particular, to the verification of temporal properties of concurrent and reactive systems.

1 Introduction

Over the past decade, it has been widely acknowledged that temporal logics can form a suitable basis for formal techniques for the analysis, specification and development of systems. In particular, temporal logics lend themselves well to the specification of both *safety* and *liveness* properties of concurrent computing systems. We refer the reader to [Pnu77,OL82,Lam83b,MP82] for extensive examples. More recently, compositional specification techniques based on temporal logics have been developed, for example [Lam83a,BKP84,Lam84,Bar87]; these techniques enable hierarchic (top-down) system development in the temporal framework.

To encourage the widespread use of such formally based development approaches, it is most important that support tools are available (cf. the requirement of interpreters, compilers, debuggers, source-code control systems, etc. for programming languages). Such tools range from the more mundane lexical and syntactic tools, e.g. syntax analysers, pretty printers and even proof checkers, through general book-keeping tools (at various levels), to semantic analysis tools, e.g. type checkers, interpreters, transformers and theorem provers. Not surprisingly, there are strong beliefs that formal development will only be widely adopted when practicable proof assistants and theorem provers exist to support the dischargement of proof obligations (because formal proof by hand is too tedious, time consuming and error prone for the average non-logician). It seems, therefore, crucial that mechanised verification support is investigated and developed for the logics underlying any potential/putative formal approach to system development. To this end, the TEMPLE project has been investigating the mechanisation of temporal logics. Indeed, in the report [BG87], we present a survey of different techniques for "mechanising" various forms of temporal logic; these range through decision procedures, model checkers, resolution-based theorem proving, direct execution and program synthesis.

In this article, we outline a *generic* system for the verification of temporal properties of finite state programming languages that we have developed at Manchester. The system couples the verification paradigm based on model checking [CES86] of finite state programs, with language presentation via formal semantic description such as Structural Operational Semantics [Plo81]. We have structured our

*Research supported under Alvey/SERC grant GR/D/57492

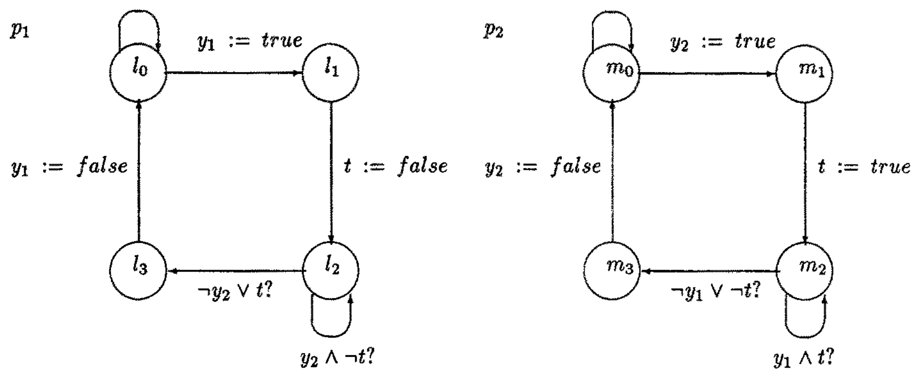

Figure 1: Mutual Exclusion Algorithm

presentation as follows. Section 2 reviews two approaches to temporal verification of programs; it highlights the impracticability of using decision procedures and the apparent viability of model checking. Section 3 provides a overview of the architecture of our verification system, in particular, it is concerned with the generic aspects of the model (or state machine) generator, SMG. Finally in section 4 we discuss the application domain for our prototype verification system, its current limitations and the future directions in which the work is proceeding.

2 Mechanised temporal verification

Given a program $\mathcal{P}$ in a finite state language $\mathcal{L}$ (see section 3.1.1) with given operational semantics, one approach to the problem of verifying its temporal behaviour, i.e. deciding whether its execution behaviour satisfies a given temporal formula ϕ, is to work entirely within the temporal framework. This means giving a temporal semantics for $\mathcal{L}$, equivalent to the operational semantics; the meaning of $\mathcal{P}$ is then a temporal formula ψ and the verification problem is then reduced to proving the validity of the formula $\psi \Rightarrow \phi$. The proof of this, usually lengthy, formula can either be tackled by hand or, if the logic used is a propositional linear time logic, can be proved mechanically by use of a decision procedure such as those of [Gou84]. Typically, a temporal logic decision procedure will validate a formula by attempting to create a model for the negation of the given formula; if a model does not exist then the formula is valid. Because of the complexities involved, this approach is not always viable.

An alternative approach of much lower complexity is to use the operational semantics to build a finite state model of the program and then use a model checker to test the truth of ϕ on the model. This approach has been used with some success in a branching time logic framework [BCDM84,CES86,BC86,Bro86].

2.1 Verification Example

We exemplify the two approaches outlined above, and justify our choice of direction, with the verification of the exclusion property of a simple mutual exclusion algorithm [Pet81,Pnu84]. Figure 1 below presents the algorithm as two concurrent finite state machines. The states labelled l_0 and m_0 are the initial states and the states l_3 and m_3 are the critical regions. Each process p_i has a variable y_i used to signal its desire to enter its critical section. The processes "share" a variable t used to arbitrate in situations of conflict.

2.1.1 Using a Decision procedure

The semantics is given in terms of a propositional, discrete linear time, temporal logic. The logic consists of a set of propositions, the standard propositional connectives and the temporal operators $\Box, \Diamond$ and $\bigcirc$. The intuitive interpretation of these operators is that, if ϕ is a formula, then

- $\Box\phi$ is true if ϕ is true *always* in the future (including the current moment).
- $\Diamond\phi$ is true if ϕ is true *sometime* in the future (or at the current moment).
- $\bigcirc\phi$ is true if ϕ is true at the *next* moment in the future.

The semantics of the parallel composition of the concurrent state machines p_1 and p_2 of Figure 1 is encoded as follows.

- The form of the formula describing the semantics is

$$\begin{array}{l} (\text{Initial conditions})\wedge \\ \Box((w \Rightarrow (\text{Disjunction of } p_1 \text{ transitions}))\wedge \\ \quad (\neg w \Rightarrow (\text{Disjunction of } p_2 \text{ transitions}))\wedge \\ \quad \Diamond w \wedge \Diamond\neg w) \end{array}$$

  The auxiliary proposition w is used to determine which process makes a step, so that when w is true p_1 makes a step and when $\neg w$, p_2. The final clause, $\Diamond w \wedge \Diamond\neg w$, ensures that each process takes a step infinitely often, i.e. we have a *weakly fair* parallel composition.

- Each transition is described by a formula of the form

$$\text{Old location } \wedge \bigcirc(\text{New location}) \wedge \text{State change}$$

  The locations l_0, l_1, l_2 and l_3 are encoded using auxiliary propositions c_1 and c_2 as below.

$$\begin{array}{ll} l_0 \Leftrightarrow \neg c_1 \wedge \neg c_2 & l_1 \Leftrightarrow c_1 \wedge \neg c_2 \\ l_2 \Leftrightarrow \neg c_1 \wedge c_2 & l_3 \Leftrightarrow c_1 \wedge c_2 \end{array}$$

  The locations of the second process, i.e. m_0, m_1, m_2 and m_3, are similarly encoded using propositions d_1 and d_2.
  Thus, for example, the p_1 transition from l_1 to l_2 is described by the formula

$$((c_1 \wedge \neg c_2) \wedge \bigcirc(\neg c_1 \wedge c_2) \wedge \bigcirc\neg t \wedge (y_1 \Leftrightarrow \bigcirc y_1))$$

We now express the desired properties of the algorithm, in this case the mutual exclusion property, as a formula in the same logic. Thus, we wish to encode "Always not (in l_3 and in m_3)", which in terms of the auxiliary control propositions is

$$\Box\neg(c_1 \wedge c_2 \wedge d_1 \wedge d_2)$$

Hence the formula for validation is that presented in Figure 2. Establishing the validity of this formula (which has 159 subformulae) with our decision procedure executing on a Sun 3/50 took about 2 minutes. Our original temporal logic encoding for this example was a slightly less obvious one, in an attempt to reduce the size of the formula to be proved; this "clever" encoding did in fact result in a slightly smaller formula (145 subformulae), but unfortunately took over 20 minutes to prove!

The above example shows quite clearly the intractability of this use of decision procedures in full-scale program verification.

$$
\begin{array}{l}
(t \wedge \neg y_1 \wedge \neg y_2 \wedge \neg c_1 \wedge \neg c_2 \wedge \neg d_1 \wedge \neg d_2) \wedge \\
\Box((w \Rightarrow ((d_1 \Leftrightarrow \bigcirc d_1) \wedge (d_2 \Leftrightarrow \bigcirc d_2) \wedge (y_2 \Leftrightarrow \bigcirc y_2) \wedge \\
\qquad\qquad (((\neg c_1 \wedge \neg c_2) \wedge \bigcirc(\neg c_1 \wedge \neg c_2) \wedge (y_1 \Leftrightarrow \bigcirc y_1) \wedge (t \Leftrightarrow \bigcirc t)) \vee \\
\qquad\qquad ((\neg c_1 \wedge \neg c_2) \wedge \bigcirc(c_1 \wedge \neg c_2) \wedge \bigcirc y_1 \wedge (t \Leftrightarrow \bigcirc t)) \vee \\
\qquad\qquad ((c_1 \wedge \neg c_2) \wedge \bigcirc(\neg c_1 \wedge c_2) \wedge \bigcirc \neg t \wedge (y_1 \Leftrightarrow \bigcirc y_1)) \vee \\
\qquad\qquad ((\neg c_1 \wedge c_2) \wedge \bigcirc(\neg c_1 \wedge c_2) \wedge (y_2 \wedge \neg t) \wedge (y_1 \Leftrightarrow \bigcirc y_1) \wedge (t \Leftrightarrow \bigcirc t)) \vee \\
\qquad\qquad ((\neg c_1 \wedge c_2) \wedge \bigcirc(c_1 \wedge c_2) \wedge (\neg y_2 \vee t) \wedge (y_1 \Leftrightarrow \bigcirc y_1) \wedge (t \Leftrightarrow \bigcirc t)) \vee \\
\qquad\qquad ((c_1 \wedge c_2) \wedge \bigcirc(\neg c_1 \wedge \neg c_2) \wedge \bigcirc \neg y_1 \wedge (t \Leftrightarrow \bigcirc t))))) \\
\quad \wedge \\
\quad (\neg w \Rightarrow ((c_1 \Leftrightarrow \bigcirc c_1) \wedge (c_2 \Leftrightarrow \bigcirc c_2) \wedge (y_1 \Leftrightarrow \bigcirc y_1) \wedge \\
\qquad\qquad (((\neg d_1 \wedge \neg d_2) \wedge \bigcirc(\neg d_1 \wedge \neg d_2) \wedge (y_2 \Leftrightarrow \bigcirc y_2) \wedge (t \Leftrightarrow \bigcirc t)) \vee \\
\qquad\qquad ((\neg d_1 \wedge \neg d_2) \wedge \bigcirc(d_1 \wedge \neg d_2) \wedge \bigcirc y_2 \wedge (t \Leftrightarrow \bigcirc t)) \vee \\
\qquad\qquad ((d_1 \wedge \neg d_2) \wedge \bigcirc(\neg d_1 \wedge d_2) \wedge \bigcirc t \wedge (y_2 \Leftrightarrow \bigcirc y_2)) \vee \\
\qquad\qquad ((\neg d_1 \wedge d_2) \wedge \bigcirc(\neg d_1 \wedge d_2) \wedge (y_1 \wedge t) \wedge (y_2 \Leftrightarrow \bigcirc y_2) \wedge (t \Leftrightarrow \bigcirc t)) \vee \\
\qquad\qquad ((\neg d_1 \wedge d_2) \wedge \bigcirc(d_1 \wedge d_2) \wedge (\neg y_1 \vee \neg t) \wedge (y_2 \Leftrightarrow \bigcirc y_2) \wedge (t \Leftrightarrow \bigcirc t)) \vee \\
\qquad\qquad ((d_1 \wedge d_2) \wedge \bigcirc(\neg d_1 \wedge \neg d_2) \wedge \bigcirc \neg y_2 \wedge (t \Leftrightarrow \bigcirc t))))) \\
\quad \wedge \\
\quad \Diamond w \wedge \Diamond \neg w)) \\
\\
\Rightarrow \ \Box \neg (c_1 \wedge c_2 \wedge d_1 \wedge d_2)
\end{array}
$$

Figure 2: Mutual Exclusion Verification Obligation

```
procedure $p_1()$
      $[\textbf{true} \rightarrow y_1 := \textbf{true}; t := \textbf{false}; p_{11}() \ \Box \ \textbf{true} \rightarrow p_1()]$.
procedure $p_{11}()$
      $[\neg y_2 \vee t \rightarrow p_{12}() \ \Box \ y_2 \wedge \neg t \rightarrow p_{11}()]$.
procedure $p_{12}()$
      $crit_1 := \textbf{true}; crit_1 := \textbf{false}; y_1 := \textbf{false}; p_1()$.

procedure $p_2()$
      $[\textbf{true} \rightarrow y_2 := \textbf{true}; t := \textbf{true}; p_{21}() \ \Box \ \textbf{true} \rightarrow p_2()]$.
procedure $p_{21}()$
      $[\neg y_1 \vee \neg t \rightarrow p_{22}() \ \Box \ y_1 \wedge t \rightarrow p_{21}()]$.
procedure $p_{22}()$
      $crit_2 := \textbf{true}; crit_2 := \textbf{false}; y_2 := \textbf{false}; p_2()$.

program
      $y_1 := \textbf{false}; y_2 := \textbf{false}; t := \textbf{true};$
      $crit_1 := \textbf{false}; crit_2 := \textbf{false};$
      $p_1() \parallel p_2()$.
```

Figure 3: SMG program for Peterson Algorithm

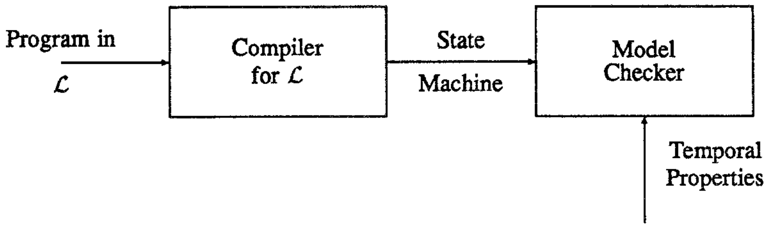

Figure 4: Architecture of Model Checking approach.

2.1.2 Using a Model checker

The alternative approach is to describe the algorithm in a high level language (see section 3.2) as shown in Figure 3. In this program, the variables $crit_1$ and $crit_2$ are used to flag the entry and exit from the critical sections; the variables y_1, y_2 and t are as above. This program is then compiled into a state machine with 32 states and we can verify that the program has the required mutual exclusion property using a temporal logic model checker. One such is mcb [Bro86], a model checker for *computation tree logic* (CTL), a branching time temporal logic. In CTL the mutual exclusion property is given by the formula $AG(\neg(crit_1 \wedge crit_2))$. The CTL operator AG is analogous to the linear time operator $\Box$, and the formula $AG\phi$ is true if and only if for every path, at every node on that path ϕ is true. The compilation time, again on a Sun 3/50, was less than 5 seconds and the mutual exclusion property for the resulting state machine was established in considerably less than 1 second. We have also checked the linear time form of the mutual exclusion property, i.e. $\Box\neg(crit_1 \wedge crit_2)$, using a model checker for linear time temporal logic; again the time taken for the verification was less than 1 second. Of course, once the state machine is generated we are able to use the model checker to test other temporal properties without recompilation, whereas the decision procedure approach would entail proving each property entirely separately, a substantial task in each case.

3 Verification System Architecture

We agree with the conclusions of [BC86,BCDM84,CES86] that the use of model checkers provides an attractive and tractable approach to automatic verification of temporal properties. This then leads to the basic architecture of figure 4. Indeed, this is the basic architecture underlying the verification system of Clarke et al. and the system CESAR (and later XESAR) of Sifakis et al. [QS82]. In both cases, a specific high-level programming language, SML [BC86] and CESAR respectively, have been devised for describing finite state systems. The compiler for SML (ltd) produces a state machine as output; this then serves as input not only to mcb but also to various VLSI design tools. It is possible, of course, to interface model checkers for different temporal logics to the compiler. Properties of CESAR programs are also verified by use of a model checker for a branching time temporal logic.

The formal semantics for SML is presented in terms of *conditional rewrite rules* in the style of S.O.S. [Plo81]. In our system, rather than produce specific compilers for various languages, we use such a semantics directly to drive a *generic* state machine generator. This means that the system can operate on programs written in a language for which the user can supply both the parser and semantics.

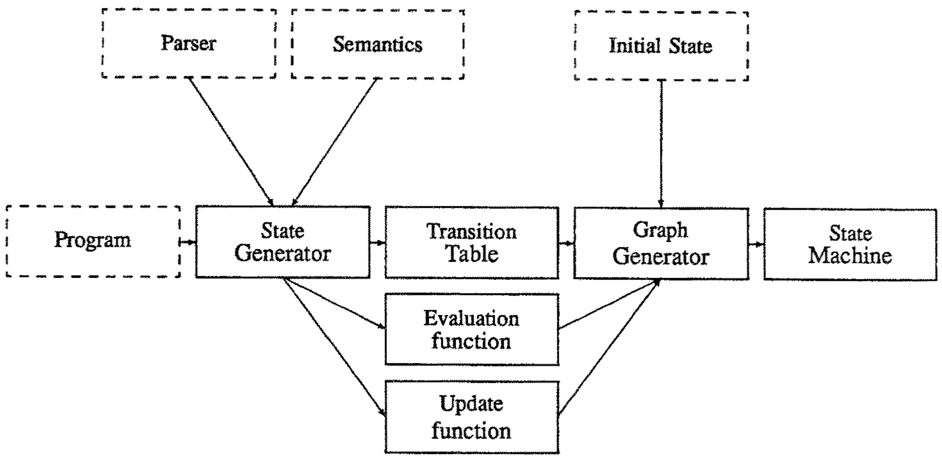

Figure 5: SMG architecture

3.1 State Machine Generator: SMG

Figure 5 outlines the basic structure of the state machine generator (compiler).

SMG takes as input a parser and operational rewrite rule semantics for the user's language $\mathcal{L}$, together with a program in $\mathcal{L}$ and an initial state assignment for the program variables. It then constructs a *transition table* for the program $\mathcal{P}$ by instantiating, for the parse tree of $\mathcal{P}$ the transition rules given in the semantics. Using the given initial states of $\mathcal{P}$, together with evaluation and state update functions obtained from the rewrite rule semantics, a *state transition graph* is then generated for $\mathcal{P}$. An illustration of this process is given in section 3.2.3.

The semantics is given, in S.O.S. style, as a set of labelled transition rules of the form

$$S_1 \xrightarrow{ec,\ sm,\ em} S_2$$

and inference rules of the form

$$\frac{R_1 \dots R_n}{R}$$

where S_1,S_2 are program phrases, ec is a boolean expression defining the enabling condition for the transition, sm is the state modification effected by the transition, em the environment modification and R, $R_1 \dots R_n$ are transition rules.

The parser input in the current system is given via a yacc grammar [Joh79].

The output format of the state transition graph can be chosen as appropriate for the model checker to be used, in particular for mcb or a linear time temporal logic model checker.

3.1.1 Input Language Restriction

Clearly, we can only build state transition graphs for finite state programs, i.e. programs that will only use, during their finite or infinite execution, a finite state space. Typically we refer to those languages, whose programs are always finite state, as finite state languages (e.g. as we did in section 2). In this first prototype of SMG we make no checks on finiteness of input language and assume that all variables are global, and all procedures use a call by name mechanism for parameter substitution.

3.2 An example language for SMG

The Peterson algorithm given in Fig. 3 was written in a language generated for small demonstrations of the SMG system. It consists of Boolean variables and expressions, assignment, guarded commands, sequential composition, parallel composition and tail-recursive procedures; its syntax and semantics are given below.

3.2.1 Language Syntax

We describe the syntax (omitting the syntax of expressions) using a BNF-like notation. Assuming the syntactic classes Var, $Expression$ and $Name$ for the obvious entities, we have

$$\begin{array}{lll}
Prog & ::= & Decl\text{-}list\ Body \\
Decl\text{-}list & ::= & Decl \mid Decl\ Decl\text{-}list \\
Decl & ::= & \textbf{procedure}\ Name\ (Var\text{-}list)\ Statement. \\
Var\text{-}list & ::= & \mid NVar\text{-}list \\
NVar\text{-}list & ::= & Var \mid Var, NVar\text{-}list \\
Body & ::= & \textbf{program}\ Statement. \\
Statement & ::= & Var := Expression\ ;\ Statement \mid \\
 & & [Choice] \mid Call \mid Parallel \\
Choice & ::= & Expression \rightarrow Statement \mid \\
 & & Expression \rightarrow Statement\ \square\ Choice \\
Call & ::= & Name(Var\text{-}list) \\
Parallel & ::= & Call \parallel Call
\end{array}$$

Note that this is not the current form of parser input, as mentioned above, SMG currently requires the user to define or modify a Yacc based parser. However it is intended to interface a parser-generator to SMG that will accept such BNF-like input.

3.2.2 Language Semantics

The dynamic semantics intended for the language's use in Fig. 3, is given below in the SOS style described in section 3.1. We assume

$$\begin{array}{c}
S, S_i \in Statement, x \in Var, \overline{x} \in Var\text{-}list \\
e, e_i \in Expression \text{ and } p \in Name
\end{array}$$

For clarity of exposition, we omit the environment component.

$$\textbf{program}\ S. \xrightarrow{\textbf{true},\ [\,]} S$$

$$x := e; S \xrightarrow{\textbf{true},\ [x/e]} S$$

$$[\square\ e_i \rightarrow S_i] \xrightarrow{e_i,\ [\,]} S_i$$

$$[\square\ e_i \rightarrow S_i] \xrightarrow{\bigwedge \neg e_i,\ [\,]} [\square\ e_i \rightarrow S_i]$$

$$p(\overline{x}) \xrightarrow{\textbf{true},\ [\,]} \mathcal{E}(p)(\overline{x})$$

(where $\mathcal{E}(p)$ is the body of p in the current environment) and the inference rules

$$\frac{S_1 \xrightarrow{ec,\ sm} S_1'}{S_1 \parallel S_2 \xrightarrow{ec,\ sm} S_1' \parallel S_2}$$

$$\frac{S_1 \xrightarrow{ec,\ sm} S_1'}{S_2 \parallel S_1 \xrightarrow{ec,\ sm} S_2 \parallel S_1'}$$

These rules are fairly self explanatory, however, we briefly explain the assignment and guarded choice rules.

- If a program remaining to be executed is of the form $x := e; S$ then, since the enabling condition of the matching rule is **true**, it can *unconditionally* rewrite to the program S, using the state modification $[x/e]$ to update the global state by overwriting the current value of x by that of e.
- A guarded choice may nondeterministically rewrite to any of its choices whose guarding condition is open, i.e. true. Thus, in the graph construction, the node corresponding to the guarded choice construct in this state may have several outgoing edges, each corresponding to an open choice. If none of the guarding conditions evaluate to true, then the statement rewrites to itself; a blocked process thus appears to be idling. Alternative semantics for guarded choice are given in section 3.2.4 below.

The current version of SMG has a built in evaluation mechanism for handling Boolean expressions.This may appear to be a limitation on the user's ability to alter the input language semantics. However, although it is possible to present the semantics of Boolean expression evaluation within the SOS framework, the state explosion that would occur seems an unnecessary price to pay for such a common semantic entity. This strategy of mixed compilation and interpretation will be extended for handling other common semantic entities, however the user will be given some ability to modify or override the built in mechanisms.

3.2.3 State machine generation

To illustrate the operation of SMG, consider the program fragment

$$t := \mathbf{false}; p_{11}()$$

of the example program of Fig. 3. Matching this fragment with the semantic rewrite rules given above results in the transition rules

$$\begin{array}{lrcl}
1) & t := \mathbf{false}; p_{11}() & \xrightarrow{\mathbf{true},\ [t/\mathbf{false}]} & p_{11}() \\
2) & p_{11}() & \xrightarrow{\mathbf{true},\ [\,]} & [\neg y_2 \lor t \rightarrow p_{12}() \ \Box\ y_2 \land \neg t \rightarrow p_{11}()] \\
3) & [\neg y_2 \lor t \rightarrow p_{12}() \ \Box\ y_2 \land \neg t \rightarrow p_{11}()] & \xrightarrow{\neg y_2 \lor t,\ [\,]} & p_{12}() \\
4) & [\neg y_2 \lor t \rightarrow p_{12}() \ \Box\ y_2 \land \neg t \rightarrow p_{11}()] & \xrightarrow{y_2 \land \neg t,\ [\,]} & p_{11}() \\
5) & [\neg y_2 \lor t \rightarrow p_{12}() \ \Box\ y_2 \land \neg t \rightarrow p_{11}()] & \xrightarrow{\neg(\neg y_2 \lor t) \land \neg(y_2 \land \neg t),\ [\,]} & [\neg y_2 \lor t \rightarrow p_{12}() \ \Box\ y_2 \land \neg t \rightarrow p_{11}()]
\end{array}$$

The procedure calls $p_{11}()$ occurring in rules 1 and 4 are replaced by the procedure body as given by rule 2, similarly for the call to $p_{12}()$. The final rule is of course never actually used, since its transition condition is always false; such redundant rules can be detected by use of a propositional calculus decision procedure and eliminated. This gives the following rules which are installed in the *transition table* mentioned in section 3.

$$\begin{array}{lrcl}
1) & t := \textbf{false}; p_{11}() & \xrightarrow{\textbf{true},\ [t/\textbf{false}]} & [\neg y_2 \vee t \rightarrow p_{12}() \ \Box\ y_2 \wedge \neg t \rightarrow p_{11}()] \\
2) & [\neg y_2 \vee t \rightarrow p_{12}() \ \Box\ y_2 \wedge \neg t \rightarrow p_{11}()] & \xrightarrow{\neg y_2 \vee t,\ [\,]} & \text{Body of } p_{12}() \\
3) & [\neg y_2 \vee t \rightarrow p_{12}() \ \Box\ y_2 \wedge \neg t \rightarrow p_{11}()] & \xrightarrow{y_2 \wedge \neg t,\ [\,]} & [\neg y_2 \vee t \rightarrow p_{12}() \ \Box\ y_2 \wedge \neg t \rightarrow p_{11}()]
\end{array}$$

The transition table is used together with the evaluation and state update functions to generate the final state graph.

Consider the graph generation from the above fragment, given the a state in which

$$y_1 = \textbf{true},\ y_2 = \textbf{true},\ t = \textbf{true},\ crit_1 = \textbf{false},\ crit_2 = \textbf{false}$$

which we abbreviate to

$$S_0 = tttff,\ t := \textbf{false}; p_{11}()$$

Applying the first transition rule gives the new state

$$S_1 = ttfff,\ [\neg y_2 \vee t \rightarrow p_{12}() \ \Box\ y_2 \wedge \neg t \rightarrow p_{11}()]$$

Rules 2 and 3 now match, but rule 2 cannot be used since its enabling condition is false. Applying rule 3 then yields a state, say S_2,

$$S_2 = ttfff,\ [\neg y_2 \vee t \rightarrow p_{12}() \ \Box\ y_2 \wedge \neg t \rightarrow p_{11}()]$$

which is of course the same as state S_1. Thus the graph construction for this initial fragment terminates.

3.2.4 Alternative semantics

Given the flexibility or tailorability of SMG, it is easy for a user developing his own application language to experiment with different semantics and its effects on verification. For example, if we impose a restriction that procedures may only modify variables that they own, i.e. a distributed variables language [BKP86], then the semantics of the parallel construct can be altered to that of lock-step parallelism by replacing the two derived rules by the single rule

$$\frac{S_1 \xrightarrow{ec_1,\ sm_1} S_1',\ S_2 \xrightarrow{ec_2,\ sm_2} S_2'}{S_1 \parallel S_2 \xrightarrow{ec,\ sm} S_1' \parallel S_2'}$$

where

$$\begin{array}{rcl} ec & = & ec_1 \wedge ec_2 \\ sm & = & sm_1 \cup sm_2 \end{array}$$

and $sm_1 \cup sm_2$ is union of maps.

With the semantics given above, a set of guarded commands all of whose guards are false will idle until one of them becomes true. An alternative approach is given by introducing a new statement **skip** , with the semantics

$$\textbf{skip}\ ; S \xrightarrow{\textbf{true},\ [\,]} S$$

and replacing the existing rules for guarded commands with

$$[\,\Box\ e_i \rightarrow S_i] \xrightarrow{x_i,\ [\,]} S_i$$

$$[\,\Box\ e_i \rightarrow S_i] \xrightarrow{\bigwedge \neg x_i,\ [\,]} \textbf{skip}$$

3.3 Fairness

With the interleaving model of parallelism implied by the first SMG semantics shown, the generated state transition graph will contain all possible interleavings, even though the desired language semantics might include some notion of fairness. For example the program in figure 3 could always take a p_1 step and completely ignore p_2. In this example the presence or absence of fairness does not affect mutual exclusion, a safety property, but would affect liveness properties such as ensuring entry to each critical section. To ensure that only fair execution paths are considered by the model checker we need some mechanism for describing such paths. One approach is to describe such paths by use of additional variables. The model checker mcb includes a mechanism for expressing fairness constraints as a set of CTL formulae that are infinitely often true on each "fair" path, and such constraints can be expressed directly within the logic if a linear time model checker is being used. In the present implementation of SMG it is necessary to explicitly include these extra variables within the program. In the above example we could replace the definition of p_1 by

```
procedure $p_1()$
        $f_1$ := true ; $f_1$ := false ;
        $y_1$ := true ; $t$ := false ; $p_{11}()$.
procedure $p_2()$
        $f_2$ := true ; $f_2$ := false ;
        $y_2$ := true ; $t$ := true ; $p_{21}()$.
```

Fair paths are then those on which the formulae f_1,$\neg f_1$,f_2 and $\neg f_2$ are true infinitely often. Thus using mcb we impose the fairness constraints f_1,$\neg f_1$,f_2 and $\neg f_2$, and using a linear time model checker to check a property ϕ we need to check the formula

$$\Box(\Diamond f_1 \wedge \Diamond\neg f_1 \wedge \Diamond f_2 \wedge \Diamond\neg f_2) \Rightarrow \phi$$

The obligation on the programmer to include extra information that is actually a consequence of the language semantics is obviously unsatisfactory. Two approaches to overcome this limitation are currently under investigation. The first is to use the above approach but to generate automatically the necessary extra variables and fairness conditions for transmission to the model checker. This has the disadvantage of increasing the size of each state in the finite state machine and increasing the number of states. The second approach is to attach some form of process labelling to the edges of the state machine and to modify the model checker to use this labelling to restrict that search space to fair paths.

4 Discussion

SMG coupled with a temporal logic model checker provides a powerful tool for the verification of temporal properties of (concurrent) programs. Of course the combination is not intended to replace existing validation tools, but to supplement the tools that the systems engineer has at his disposal. The restriction, mentioned earlier, on finiteness may seem severe; however, we feel that most system structures that require temporal verification fall into this category. Applications to which we believe the tool most appropriate range from communications protocol verification (at software, e.g. LOTOS, and hardware, e.g. ELLA, levels), through process control verification to verification of temporal aspects of hardware systems. Existing experience gained with SML has certainly demonstrated the practicality of model checking and our approach of using a generic front end to model checker quickens and simplifies state model generator or "compiler" construction in much the same way as compiler generators aid compiler construction. SMG is also a most useful tool for teaching environments where it is desirable to give students the ability to design their own languages for particular verification applications.

SMG is, at present, a prototype and was constructed to investigate the feasibility and usefulness of a semantics driven approach to state transition graph generation. As such, there are several unnecessary limitations, which will not be present in future implementations. For example, all variables must be Boolean, the parameter mechanism for procedures is by name, the input language parser is given as YACC grammar and fairness is handled crudely.

There is, however, a limitation that is rather more serious, but for which we believe there there may be some hope in particular cases. The major problem with the model checking approach to program verification is *state explosion.* Consider a system consisting of 12 parallel asynchronous processes, each process represented by a 10 state automaton. The combined automaton would have an upper bound of a 10^{12} states, well beyond our current capabilities. The work of [CGB86] on concurrent system that are composed of many identical processes suggests that special techniques can be applied in certain commonly occurring circumstances. At present, we are investigating the use of compositional and inductive techniques as a possible means to control the explosion.

In summary, though, we have been sufficiently encouraged by our early experience with SMG for us to continue its development. In particular, we are interfacing the system to propositional linear-time temporal logic model checkers (enabling greater flexibility with respect to fairness), extending its language capabilities and improving the parser input mechanism.

References

[Bar87] H. Barringer.
Using Temporal Logic in the Compositional Specification of Concurrent Systems.
In A. P. Galton, editor, *Temporal Logics and their Applications*, chapter 2, pages 53–90, Academic Press Inc. Limited, London, December 1987.

[BC86] M.C. Browne and E.M. Clarke.
SML - a high level language for the design and verification of finite state machines.
In *From H.D.L. descriptions to guaranteed correct circuit designs*, IFIP, September 1986.

[BCDM84] M.C. Browne, E.M. Clarke, D. Dill, and B. Mishra.
Automatic Verification of Sequential Circuits using Temporal Logic.
Technical Report CS–85–100, Department of Computer Science, Carnegie-Mellon University, 1984.

[BG87] H. Barringer and G.D. Gough.
Mechanisation of Temporal Logics. Part 1: Techniques.
Temple internal report, Department of Computer Science, University of Manchester, 1987.

[BKP84] H. Barringer, R. Kuiper, and A. Pnueli.
Now You May Compose Temporal Logic Specifications.
In *Proceedings of the Sixteenth ACM Symposium on the Theory of Computing*, 1984.

[BKP86] H. Barringer, R. Kuiper, and A. Pnueli.
A Really Abstract Concurrent Model and its Temporal Logic.
In *Proceedings of the Thirteenth ACM Symposium on the Principles of Programming Languages*, St. Petersberg Beach, Florida, January 1986.

[Bro86] M.C. Browne.
An improved algorithm for the automatic verification of finite state systems using temporal logic.
Technical Report , Department of Computer Science, Carnegie-Mellon University, December 1986.

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

[CGB86] E. M. Clarke, O. Grümberg, and M. C. Browne.
Reasoning about networks with many identical finite-state processes.
In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, ACM, August 1986.

[Gou84] G. D. Gough.
Decision Procedures for Temporal Logic.
Master's thesis, Department of Computer Science, University of Manchester, October 1984.

[Joh79] Stephen C. Johnson.
Yacc: Yet another compiler-compiler.
Unix Programmer's Manual Vol 2b, 1979.

[Lam83a] L. Lamport.
Specifying concurrent program modules.
ACM Transactions on Programming Languages and Systems, 5(2):190–222, July 1983.

[Lam83b] L. Lamport.
What good is temporal logic.

In R. E. A. Mason, editor, *Information Processing 83*, pages 657–668, IFIP, Elsevier Science Publishers B.V. (North-Holland), 1983.

[Lam84] L. Lamport.
An Axiomatic Semantics of Concurrent Programming Languages.
In Krysztof Apt, editor, *Logics and Models of Concurrent Systems*, pages 77–122, NATO, Springer-Verlag, La Colle-sur-Loup, France, October 1984.

[MP82] Z. Manna and A. Pnueli.
Verification of Concurrent Programs: The Temporal Framework.
In Robert S. Boyer and J. Strother Moore, editors, *The Correctness Problem in Computer Science*, Academic Press, London, 1982.

[OL82] S. Owicki and L. Lamport.
Proving Liveness Properties of Concurrent Programs.
ACM Transactions on Programming Languages and Systems, 4(3):455–495, July 1982.

[Pet81] G. L. Peterson.
Myths about the mutual exclusion problem.
Information Processing Letters, 12(3):115–116, 1981.

[Plo81] G. D. Plotkin.
A structural approach to operational semantics.
Technical Report DAIMI FN-19, Department of Computer Science,Aarhus University, September 1981.

[Pnu77] A. Pnueli.
The Temporal Logic of Programs.
In *Proceedings of the Eighteenth Symposium on the Foundations of Computer Science*, Providence, November 1977.

[Pnu84] A. Pnueli.
In transition from global to modular temporal reasoning about programs.
In Krysztof Apt, editor, *Logics and Models of Concurrent Systems*, pages 123–144, NATO, Springer-Verlag, La Colle-sur-Loup, France, October 1984.

[QS82] J. P. Queille and J. Sifakis.
Specification and verification of concurrent systems in CESAR.
Lecture Notes in Computer Science, 137, April 1982.