PROCEEDINGS

Southern African Computer Symposium 1991

6th de Rekenaarsimposium van Suider Afrika 1991

DE OVERBERGER HOTEL, CALEDON

2 - 3 JULY 1991

SPONSORED BY

ISM

FRD

GENMIN

EDITED BY

M H Linck

DEPARTMENT OF COMPUTER SCIENCE • UNIVERSITY OF CAPE TOWN

PROCEEDINGS / KONGRESOPSOMMINGS

6th
SOUTHERN AFRICAN COMPUTER SYMPOSIUM

6de
SUIDELIKE-AFRIKAANSE REKENAARSIMPOSIUM

De Overberger Hotel, Caledon

2 - 3 JULY 1991

SPONSORED by

ISM
FRD
GENMIN

EDITED by

M H LINCK

Department of Computer Science

University of Cape Town

TABLE OF CONTENTS

FOREWORD

The 6th Computer Symposium, organised under the auspices of SAICS, carries on the tradition of providing an opportunity for the South African scientific computing community to present research material to their peers.

It was heartening that 31 papers were offered for consideration. As before all these papers were refereed. Thereafter a selection committee chose 21 for presentation at the Symposium.

Several new dimensions are present in the 1991 symposium:

* The Symposium has been arranged for the day immediately after the SACLA conference.

* It is being run over only 1 day in contrast to the 2-3 days of previous symposia.

* I believe that it is first time that a Symposium has been held outside of the Transvaal.

* Over 85 people will be attending. Nearly all will have attended both events.

* A Sponsorship package for both SACLA and the Research Symposium was obtained. (This led to reduced hotel costs compared to previous symposia)

A major expense is the production of the Proceedings of the Symposium. To ensure financial soundness authors have had to pay the page charge of R20 per page.

A thought for the future would be consideration of a poster session at the Symposium. This could provide an alternative approach to presenting ideas or work.

I would sincerely hope that the twinning of SACLA and the Research Symposium is considered successful enough for this combination survive. As to whether a Research Symposium should be run each year after SACLA, or only every second year, is a matter of need and taste.

A challenge for the future is to encourage an even greater number of MSc & PhD students to attend the Symposium. Unlike this year, I would recommend that they be accommodated at the same cost as everyone else. Only if it is financially necessary should the sponsored number of students be limited.

I would like to thank the other members of the organising committee and my colleagues at UCT for all the help that they have given me. A special word of thanks goes to Prof. Pieter Kritzinger who has provided me with invaluable help and ideas throughout the organisation of this 6th Research Symposium.

M H Linck
Symposium Chairman

SYMPOSIUM CHAIRMAN

M H Linck, University of Cape Town

ORGANISING COMMITTEE

D Kourie, Pretoria University.

P S Kritzinger, University of Cape Town.

M H Linck, University of Capè Town.

SPONSORS

ISM

GENMIN

FRD

LIST OF REFEREES FOR 6th RESEARCH SYMPOSIUM

NAME	INSTITUTION
Barnard, E	Pretoria
Becker, Ronnie	UCT
Berman S	UCT
Bishop, Judy	Wits
Berman, Sonia	UCT
Brink, Chris	UCT
Bodde, Ryn	Networks Systems
Bornman, Chris	UNISA
Bruwer, Piet	UOFS
Cherenack, Paul	UCT
Cook Donald	UCT
de Jaeger, Gerhard	UCT
de Villiers, Pieter	Stellenbosch
Ehlers, Elize	RAU
Eloff, Jan	RAU
Finnie, Gavin	Natal
Gaynor, N	AECI
Hutchinson, Andrew	UCT
Jourdan, D	Pretoria
Kourie Derrick	Pretoria
Kritzinger, Pieter	UCT
Krzesinski, Tony	Stellenbosch
Laing, Doug	ISM
Labuschagne, Willem	UNISA
Levy, Dave	Natal

MacGregor, Ken	UCT
Machanick, Philip	Wits
Mattison Keith	UCT
Messerschmidt, Hans	UOFS
Mutch, Laurie	Shell
Neishlos, N	Wits
Oosthuizen, Deon	Pretoria
Peters Joseph	Simon Fraser
Ram, V	Natal, Pmb.
Postma, Stef	Natal, Pmb
Rennhackkamp, Martin	Stellenbosch
Shochot, John	Wits
Silverberg, Roger	Council for Mineral Technology
Smit, Riël	UCT
Smith, Dereck	UCT
Terry, Pat	Rhodes
van den Heever, Roelf	UP
van Zijl, Lynette	Stellenbosch
Venter, Herman	Fort Hare
Victor, Herna	Stellenbosch
von Solms, Basie	RAU
Wagenaar, M	UPE
Wentworth, Peter	Rhodes
Wheeler, Graham	UCT
Wood, Peter	UCT

6TH RESEARCH SYMPOSIUM - 1991

FINAL PROGRAM

TUESDAY 2nd July 1991

10h00 - 13h00 **Registration**

13h00 - 13h50 **PUB LUNCH**

14h00 - 15h30 SESSION 1A

Venue: Hassner

Chairman: Prof Basie von Solms

14h00 - 14h30
"A value can belong to many types. "
B H Venter, University of Fort Hare

14h30 - 15h00
"A Transputer Based Embedded Controller Development System"
M R Webster, R G Harley, D C Levy & D R Woodward, University of Natal

15h00 - 15h30
"Improving a Control and Sequencing Language"
G Smit and C Fair, University of Cape Town

SESSION 1B

Venue: Hassner C

Chairman:Prof Roelf v d Heever

14h00 - 14h30
"Design of an Object Orientated Framework for Optimistic Parallel Simulation on Shared-Memory Computers" P Machanick, University of Witwatersrand

14h30 - 15h00
"Using Statecharts to Design and Specify the GMA Direct-Manipulation User Interface" L van Zijl & D Mitton, University of Stellenbosch

15h00 - 15h30
"Product Form Solutions for Multiserver Centres with Heirarchical Classes of Customers" A Krzesinski, University of Stellenbosch and R Schassberger, Technische Universität Braunschweig

15h30 - 16h00 **TEA**

16h00 - 17h30 SESSION 2A

Venue: Hassner

Chairman: Prof Derrick Kourie

16h00 - 16h30
"A Reusable Kernel for the Development of Control Software" W Fouché and P de Villiers, University of Stellenbosch

16h30 - 17h00
"An Implementation of Linda Tuple Space under the Helios Operating System" P G Clayton, E P Wentworth, G C Wells and F de-Heer-Menlah, Rhodes University

17h00 - 17h30
"The Design and Analysis of Distributed Virtual Memory Consistency Protocols in an Object Orientated Operating System" K MacGregor, University of Cape Town & R Campbell, University of Illinois at Urbana-Champaign

19h30 **PRE-DINNER DRINKS**

20h00 **GALA CAPE DINNER**
(Men: Jackets & ties)

WEDNESDAY 3rd July 1991

7h00 - 8h15 BREAKFAST

8h15 - 9h45 SESSION 3A

Venue: Hassner

Chairman: Assoc Prof P Wood

8h15 - 8h45
"Concurrency Control Mechanisms for Multidatabase Systems" A Deacon, University of Stellenbosch

8h45 - 9h15
"Extending Local Recovery Techniques for Distributed Databases" H L Victor & M H Rennhackkamp, University of Stellenbosch

9h15 - 9h45
"Analysing Routing Strategies in Sporadic Networks" S Melville, University of Natal

SESSION 3B

Venue: Hassner C

Chairman: Prof G Finnie

8h15 - 8h45
The Design of a Speech Synthesis System for Afrikaans" M J Wagener, University of Port Elizabeth

8h45 - 9h15
"Expert Systems for Management Control: A Multiexpert Architecture" V Ram, University of Natal

9h15 - 9h45
"Integrating Simularity-Based and Explanation-Based Learning" G D Oosthuizen and C Avenant, University of Pretoria

9h45 - 10h15 TEA

10h15 - 11h00 SESSION 4

Venue: Hassner

Chairman: Prof P S Kritzinger
Invited paper: E Coffman

11h00 - 11h10 BREAK

11h10 - 12h40 SESSION 5A

Venue: Hassner

Chairman: Prof C Bornman

11h10 - 11h40
"Efficient Evaluation of Regular Path Programs"
P Wood, University of Cape Town

11h40 - 12h10
"Object Orientation in Relational Databases"
M Rennhackkamp, University of Stellenbosch

12h10 - 12h40
"Building a secure database using self-protecting objects" M Olivier and S H von Solms, Rand Afrikaans University

SESSION 5B

Venue: Hassner C

Chairman: Prof A Krzesinski

11h10 - 11h40
"Modelling the Algebra of Weakest Preconditions"
C Brink & I Rewitsky, University of Cape Town

11h40 - 12h10
"A Model Checker for Transition Systems"
P de Villiers, University of Stellenbosch

12h10 - 12h40
"A New Algorithm for Finding an Upper Bound of the Genus of a Graph"
D I Carson and O R Oellermann, University of Natal

12h45-12h55 GENERAL MEETING of RESEARCH SYMPOSIUM ATTENDEES

Venue: Hassner

Chairman: Dr M H Linck

13h00 - 14h00 LUNCH

FINIS 6th COMPUTER SYMPOSIUM

PAPERS

of the

6TH RESEARCH SYMPOSIUM

A Model Checker for Transition Systems

P.J.A. de Villiers
Institute for Applied Computer Science
University of Stellenbosch, Stellenbosch 7600

July 1991

Abstract

A model checker automatically determines whether a model of a reactive system satisfies its specification. Temporal logic is used to specify the intended behaviour of a reactive system which is modelled as a transition system. Fast state space exploration is mandatory, the main problem being to determine the uniqueness of each newly generated state. Traditional model checkers can analyse about 10^4 states in an acceptable amount of time. A model checker which incorporates three new ideas has been implemented. (1) A *bit vector technique* used by Holzmann in a fast protocol validation system is combined with model checking to produce a system capable of analysing about 10^7 reachable states. (2) Since state spaces are sparse and clustered, larger problems are handled by using *paging techniques*. (3) Traditional model checkers often search subspaces unnecessarily when temporal operators are nested. A top-down technique called *subproblem detection* is used which avoids this.

1 Introduction

Model checking was introduced by Clarke *et al.*[3, 4]. The technique has been used successfully to verify non-trivial systems such as hardware modules[1] by representing the system to be verified by some mathematical abstraction which is automatically checked against its specification. A transition system is used to represent a reactive system and the branching time temporal logic CTL[3] as a specification language. In principle model checking is a powerful verification technique. The theoretical foundations of model checking have been investigated thoroughly[6, 5, 12] and efficient implementation techniques are now required to put it into practice.

Three ideas are proposed in this paper to improve the performance of model checkers. Firstly, a *bit vector technique* used by Holzmann[8] to implement a fast protocol validation system is combined with model checking to speed up state generation. Secondly, large state spaces are handled by using a *paging technique* and thirdly, a technique called *subproblem detection* is used to handle nested modalities more efficiently.

2 A Transition System as Computational Model

The transition system used here is similar in spirit to the system described in [11]. A transition system which illustrates the mutual exclusion problem for two concurrent processes is given in Figure 1. A simple modelling language is used which is sufficiently powerful for the current purpose. Two process variables $p1$ and $p2$ describe two concurrent processes. Each process can each be in states 0 (inside its non-critical section), 1 (trying to enter its critical section) or 2 (inside its critical section). The variable x, which can be in states 0 or 1, represents a semaphore which is used to enforce mutual exclusion between the processes. The state of the transition system is defined by its *state vector* $s = (p1, p2, x)$, with the start state being (0,0,1). Transitions are denoted by $(guard) \rightarrow (action)$. A transition may only be selected for execution when it is enabled, which means that its guard must evaluate to true in the current state—we say that the guard *matches* the current state. To evaluate the guard of a transition, its components are compared against the corresponding components of the current state, the symbol "*" meaning that the corresponding component may have any value allowed by its range definition. To execute a transition its action vector is added to the state vector. For example, if the current state is (0,1,1), the first transition in Figure 1 will be enabled and if the transition is executed, the state vector will change to (1,1,1). The components of action vectors may have any integer values as long as executing the transition does not violate the restrictions placed on the values of individual variables. If more than one transition is enabled in a given state the nondeterminism must be resolved by exploring all possibilities. The specification given after the keyword "SPEC" will be explained in Section 4.4.

Figure 2 shows the reachability graph of unique states which can be computed by executing the transition system of Figure 1. The reachability graph is a compact representation of the reachability tree which can often be infinite. When necessary the reachability tree is reduced to be finite as will be explained in Sections 3 and 4.

3 Temporal Logic

The language of *branching time propositional temporal logic* is used as a specification language for reactive systems. Broadly speaking, a temporal logic extends classical propositional logic by adding certain non-truthfunctional temporal operators, such as *always*, *sometimes*, *next* or *until*. Validity in such a logic is governed by the assumptions made about the nature of time. Since the execution of a reactive system is usually non-deterministic this means that at any given time instant there are various different "possible futures". Therefore it is natural to use a branching time temporal logic and the logic CTL (Computation Tree Logic) is adopted as first defined in [3]. Time is modelled as an infinite *tree* of discrete time instants, each time instant corresponding to a state in the execution of a transition system.

```
MODEL
  PROCESS
    p1: 0..2;
    p2: 0..2
  VAR
    x : 0..1
  TRANS
    (0, *, *) -> ( 1, 0, 0);
    (1, *, 1) -> ( 1, 0,-1);
    (2, *, *) -> (-2, 0, 1);
    (*, 0, *) -> ( 0, 1, 0);
    (*, 1, 1) -> ( 0, 1,-1);
    (*, 2, *) -> ( 0,-2, 1)
  START
    (0,0,1)
  SPEC
    AG((p1 = 1) => AF(p1 = 2))
END.
```

Figure 1: A transition system representing the mutual exclusion problem

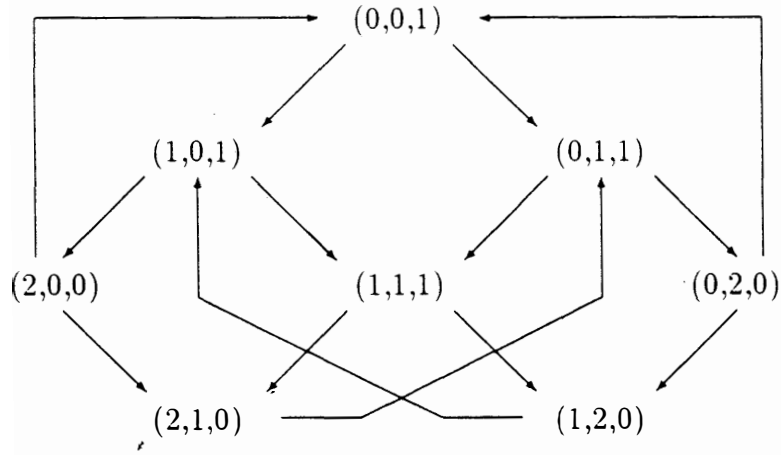

Figure 2: Reachability graph of a transition system

3.1 Basic Concepts

The language of classical propositional logic is adopted wholesale. As is customary in modal-type logics it is next assumed that there is a set S of *states* such that every atomic proposition is or is not true at a particular state. The states are related to each other by an *accessibility relation.* In modal logics the set of states, together with the accessibility relation and the assignment of truth values to atomic propositions in every state, is called a *Kripke structure.* Some properties are usually ascribed to the accessibility relation; depending on what these are, different modal logics arise. Here the accessibility relation structures the set of states into a tree, the branches representing all possible execution sequences from some initial state. The truth value of a compound formula at a given state may depend on the truth values of some of its subformulae at other states further down the tree. To express this it is necessary to quantify over the *states* in any particular execution sequence, and also over *execution sequences.* For the first purpose the notation of modal logic is used—"$\Box$" for "always", "$\Diamond$" for "sometimes" and "$\bigcirc$" for "next". For the second purpose the standard notation of first-order logic is used—"$\forall$" for "for all" and "$\exists$" for "there is". Finally, these different quantifiers are combined to obtain six different modalities: $\forall\Box$, $\forall\Diamond$, $\forall\bigcirc$, $\exists\Box$, $\exists\Diamond$ and $\exists\bigcirc$. Thus "$\forall\Box\alpha$" would say that formula α is true for all states in every execution sequence, "$\forall\Diamond\alpha$" says that in every execution sequence there is some state in which α is true, "$\forall\bigcirc\alpha$" says that α is true in every immediate successor state, and so on.

In the more technical section 3.2 below an *until* operator is introduced to increase the expressiveness of the language. Roughly, "$\alpha\ \mathcal{U}\ \beta$" is the claim that β will be true at some point in the future, and up to the immediately preceding point α will be true. Again, this may be preceded by $\forall$ or $\exists$, indicating respectively that the claim holds for all or only some execution sequences.

3.2 The Branching Time Logic CTL

The alphabet of CTL comprises: a set of variables $P = \{p_1, p_2, \ldots, p_n, \ldots\}$; the constants *true* and *false*; the connectives $\neg$ and $\wedge$; the temporal operators $\forall, \exists, \bigcirc$ and $\mathcal{U}$ with parentheses and square brackets as punctuation symbols.
Any variable by itself is a formula. If α and β are formulae, then so are:

$$\neg\alpha,\ \alpha \wedge \beta,\ \forall\bigcirc\alpha,\ \exists\bigcirc\alpha,\ \forall[\alpha\ \mathcal{U}\ \beta],\ \exists[\alpha\ \mathcal{U}\ \beta],$$

and nothing else is a formula. To define validity our assumptions concerning time are packed into the formal definition of a *Kripke structure.* It is a triple $K = (S, R, v)$ such that:

- S is a finite set, the elements of which are called *states.* There is some distinguished state $s_0 \in S$, called the *initial state.*

- R is an *adjacency relation* over S, such that (S, R) is a *tree*, with root node s_0.
- $v : P \times S \rightarrow \{0, 1\}$ is an assignment of some truth value (0 *or* 1) to every variable at every state.

A *path* is a branch of the tree (S, R)—an infinite sequence of states $(s_0, s_1, \ldots)$ starting with the root node s_0 and such that $(\forall i)[(s_i, s_{i+1}) \in R]$. A path *from* a given state s is a branch of the subtree with s as root.

Having assumed an assignment of truth values to every atomic formula at every state an inductive definition can be given of what it means for any compound formula α to be true at some state s. Such a fact is indicated by "$s \models \alpha$" and the fact that α is *not* true at s by "$s \not\models \alpha$". Inductively then, for any state s: $s \models \mathit{true}$, and $s \not\models \mathit{false}$; for any atomic formula p, and any state s: $s \models p$ iff $v(p, s) = 1$; for any formulae α and β, and any state s:

- $s \models \neg\alpha$ iff $s \not\models \alpha$
- $s \models \alpha \wedge \beta$ iff $s \models \alpha$ and $s \models \beta$
- $s \models \forall\bigcirc\alpha$ iff for every state t such that $(s, t) \in R$ we have $t \models \alpha$
- $s \models \exists\bigcirc\alpha$ iff there exists a state t such that $(s, t) \in R$ and $t \models \alpha$
- $s \models \forall[\alpha \mathcal{U} \beta]$ iff for all paths $(t_0(= s), t_1, t_2, \ldots)$ from s $(\exists i)[i \geq 0$ and $t_i \models \beta$ and $(\forall j)[0 \leq j < i$ implies $t_j \models \alpha]]$
- $s \models \exists[\alpha \mathcal{U} \beta]$ iff there exists a path $(t_0(= s), t_1, t_2, \ldots)$ from s such that $(\exists i)[i \geq 0$ and $t_i \models \beta$ and $(\forall j)[0 \leq j < i$ implies $t_j \models \alpha]]$

The other propositional connectives, $\vee$ ("or"), $\Rightarrow$ ("implies") and $\Longleftrightarrow$ ("iff") may be defined from $\neg$ and $\wedge$ in the ordinary textbook way. The remaining four of the six modalities mentioned in Section 3.1 can now be introduced by definition: $\forall\Diamond\alpha$ iff $\forall[\mathit{true}\ \mathcal{U}\ \alpha]$; $\exists\Diamond\alpha$ iff $\exists[\mathit{true}\ \mathcal{U}\ \alpha]$; $\forall\Box\alpha$ iff $\neg\exists\Diamond\neg\alpha$ and $\exists\Box\alpha$ iff $\neg\forall\Diamond\neg\alpha$.

3.3 Specification

By the inductive definition of $\models$ every CTL formula α is or is not true at any state s in a Kripke structure K. Because of the forward looking nature of the temporal operators it is necessary, for unspecified α, to have full knowledge of the distribution of truth values to atomic formulae at all states in the subtree with root s in order to deduce the truth value of α at s. Conversely, of course, if we do know that α is true at s we know something about the subtree with root s. The convention is adopted of saying that a subtree with root s has property α if the state s itself has property α, which is to say that α is true at s. In particular then, a Kripke structure K has property α iff α is true at the root node s_0.

A combination of transition system and logic is now possible. A reactive system is represented by a transition system from which an execution tree can be computed. Repetitive sequences of states are discarded according to the rules of fairness (see Section 4.3) and thus the tree can be reduced to be finite without losing important information. Simple tests on the variables of the transition system are used to determine the truth value of atomic propositions. The reduced tree may thus be regarded as a Kripke structure, and desirable properties of the reactive system are expressed as CTL formulae. The CTL formula therefore represents a property the system should have while the (finite) execution tree of the reactive system represents a Kripke structure. The model checker can now determine whether the given reactive system has the specified property by checking whether the formula is true at the root node of the tree. Temporal logic can be used to express important properties of reactive systems such as freedom from deadlock, absence of starvation and responsiveness. The CTL formula appearing after the keyword "SPEC" in Figure 1 captures absence of starvation for process 1 of the given transition system. (For practical reasons a slightly different notation is used in the computerised system for CTL—"AG" meaning "$\forall\Box$" and "AF" meaning "$\forall\Diamond$"). CTL is therefore seen as a query language to formulate questions about the system being studied, as suggested by Everitt[7]. A list of properties which are relevant for reactive systems is given in [10].

4 An Efficient Model Checker

The model checker described here executes the transition system in order to explore various paths while determining the truth value of the CTL formula. The paths explored are determined by the specific CTL formula. For example, the formula $\forall\Box\alpha$ will force the model checker to explore all paths leading from the initial state (unless the formula is found to be violated before all paths have been explored), while the formula $\exists\Diamond\alpha$ will allow the model checker to stop as soon as some path is found which leads to a state in which α is true.

Although in theory efficient model checking algorithms exist for some suitably restricted temporal logics such as CTL, little has been reported about the performance of model checker implementations. Even recently published algorithms such as the algorithm given in [12] are inefficient and it was therefore decided to explore the various possibilities of designing a model checker which would be efficient enough to verify real reactive systems such as large protocols. The success of model checking depends on efficient state space exploration techniques. Such techniques have been investigated thoroughly in the field of protocol validation and this experience influenced the design of the model checker described here.

Many protocol validators generate states dynamically, testing each state against a predefined set of correctness criteria known as state properties. To avoid analysing unnecessary states it is necessary to determine whether each newly generated state

had been visited before. It is thus necessary to compare each new state to all previously generated states. Therefore all unique states must be stored and as the number of states increases it takes progressively longer to determine the uniqueness of a state. It was determined experimentally that *state comparison* is the most time consuming operation in traditional protocol validators[8]. About 100 states per second can be processed on medium scale machines, rendering the technique impractical for systems which generate more than about 10^4 states.

Traditional model checkers[3, 4, 1, 12] compute a reachability graph which is stored in memory. Computation of this graph leads to a similar efficiency problem: each new state must be compared against all previous states to ensure uniqueness. Although few measurements of the performance of model checkers exist, this similarity between model checkers and protocol validators suggests that systems which generate more than about 10^4 states cannot be analysed by traditional model checkers. Burch *et al.*[2] showed that one large problem could be analysed by representing the state space symbolically. However, only the *potential* size of the state space (10^{20}) is given and not the number of actually *reachable* states. Furthermore, the technique is not generally applicable and more research will be required to determine its usefulness in practice.

Fortunately Holzmann recently found a new method of searching large state spaces[8, 9] which leads to a significant improvement in performance. States are generated dynamically by representing the system by a *state vector model.* The traditional method of determining uniqueness of states is replaced by a very efficient one. The new technique requires a large vector of bits to be maintained in memory to keep track of previously generated states but, even so, much larger systems can be analysed before space becomes a problem. Holzmann measured the performance of the technique and showed that it can be used to analyse protocols generating up to 10^7 states.

The model checker described here uses this efficient method to speed up the computation of the reachability graph. In addition, it is unnecessary to store the reachability graph because model checking can be done *while generating the reachability graph.* This approach has several important advantages. Firstly, space is saved because no reachability graph is stored explicitly. Secondly, the truth value of many temporal formulae can be determined without generating the entire reachability graph. An example of such a formula is $\exists\Diamond\alpha$ which will be satisfied as soon as a state is found in which α is true. This makes the model checker faster. Thirdly, problems which generate a reachability graph which is too large for the available memory space could sometimes be analysed because it may be unnecessary to generate the entire graph. To put these ideas into practice two problems had to be solved, namely, how to handle fairness constraints and nested formulae *without storing the reachability graph explicitly.*

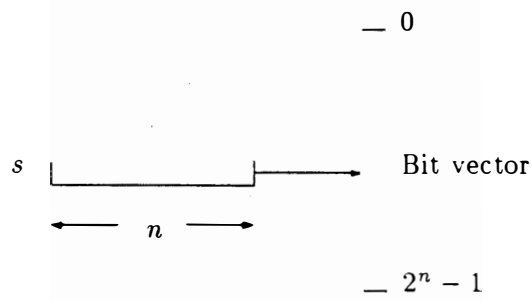

Figure 3: The bit vector technique

4.1 Fast State Comparison

As explained in Section 2 the state of a transition system is described by its state vector. The state vector as well as the guards and actions are represented by bitstrings of fixed length. The start state is assigned to the state vector to initialise a given transition system. To execute the transition system the guard of each transition is compared to the contents of the state vector until one is found to match. That transition is then selected and a new state is generated by adding the corresponding action to the state vector, and the process is repeated to execute a depth-first search of the state space. Each state is viewed as a (unique) index into a large array of bits (called a *bit vector*) as illustrated in Figure 3. As each new state is generated the corresponding bit in the bit vector is set to record the fact that such a state has been generated. The bit vector is used to determine whether a newly generated state is unique or not.

Unfortunately it is not enough to keep track of states which have been visited. The *current execution path* needs to be recorded in order to detect loops. This is important to handle fairness as will be explained in Section 4.3. A stack of state records is therefore kept to record the current execution path. To decide whether a state is on the stack, the entire stack must be searched—a relatively expensive operation. However, a state can only be on the stack if it has been visited before and therefore the bit vector provides a fast way of avoiding a stack search when it is unnecessary. Because loops are detected, the model checking algorithm will always terminate and when a specification is violated a counterexample is given which helps the user to find his error.

4.2 Model Checking Algorithm

The language used for presenting the algorithm is (more or less) standard Modula-2, for which see [13].

For brevity only a description of the various procedures used in the code is given and declarations are left out. With respect to the algorithm of Figure 4, note that

- There are two modes of operation: *trans* denoting that transitions are being

```
LOOP
  CASE mode OF
  trans:
    IF (i <= transMax) THEN
      IF Enabled(i) THEN
        stack[depth].index:= i+1;
        s:= Action(i);
        IF StateVisitedBefore THEN
          IF Stacked(s) THEN (* loop detected *)
            mode:= pred
          ELSE (* visited on earlier path *)
            WITH stack[depth] DO
              s:= state; i:= index
            END
          END
        ELSE (* unique state *)
          value:= TruthValue(sf);
          MarkStateAsVisited;
          IF value = T THEN
            UpdateStack(sf); mode:= pred
          ELSIF value = F THEN
            mode:= pred
          END
        END
      ELSE (* next transition *)
        INC(i); mode:= trans
      END
    ELSE (* no more transitions *)
      value:= FinalValue(sf);
      DEC(depth);
      mode:= pred
    END |
  pred:
    IF StackEmpty THEN RETURN value = T
    ELSE
      WITH stack[depth] DO
        s:= state; i:= index
      END;
      AdaptControlInfo
    END
  END
END
```

Figure 4: Model Checking Algorithm

tested in order to generate new states and *pred* indicating a backtrack operation to a predecessor state.

- A stack of state records is kept, each stack entry representing a state along the current path. Each record contains two fields: *state*—a state descriptor and *index*—indicating which transition to try next.
- Each state is mapped onto a unique bit in a vector of bits and *StateVisited* is a test to determine whether the bit corresponding to the state s is set.
- Procedure *Enabled* returns TRUE if the guard of transition i evaluates to true in the current state s, and FALSE otherwise.
- Procedure *Action* returns a new state which is derived from the current state s by adding each component of the action vector of transition i to each component of s.
- Procedure *Stacked* returns TRUE if state s has been visited before along the *current path*, and FALSE otherwise.
- Procedure *TruthValue* returns the truth value of formula *sf* in the current state, three values being possible: F (false), T (true) or U (undefined).
- For every unique state generated the mode is changed depending on the truth value of *sf*. When no further state exploration is necessary the mode is changed to *pred*. While *value* is U state exploration is allowed to proceed. Procedure *UpdateStack* has to do with fairness for which see Section 4.3.
- Procedure *FinalValue* determines the final truth value of the formula in the current state once it is known that all paths leading from state s have been explored. It depends on *value* and on the formula. For example if *value* is U and the formula is $\exists\Diamond\alpha$, *value* is changed to F.
- Whenever a predecessor state is entered *AdaptControlInfo* changes the values of *mode*, *value* and *depth* depending on whether more state exploration is necessary or not.

4.3 Fairness

In the present context fairness means that if a transition is enabled it should eventually be allowed to occur. Fairness concerns the behaviour of the transition system when certain paths are executed repeatedly and non-deterministic choices occur. Consider Figure 2 again. The execution path $(0,0,1) \rightarrow (1,0,1) \rightarrow (1,1,1) \rightarrow (1,2,0) \rightarrow (1,0,1) \rightarrow (1,1,1) \cdots$ is an example of an unfair path. If the transition which leads to state $(1,1,1)$ is always chosen at state $(1,0,1)$, process 1 will never be able to execute. The system will therefore never reach state $(2,0,0)$. Under these

circumstances the given specification will not be satisfied. The model checker must ignore such unfair behaviour because the transition system is meant to be fair.

The various temporal formulae can be classified into two groups according to their behaviour with respect to fairness:

1. $\forall\Box\alpha$, $\exists\Diamond\alpha$ and $\exists(\alpha\mathcal{U}\beta)$
2. $\exists\Box\alpha$, $\forall\Diamond\alpha$ and $\forall(\alpha\mathcal{U}\beta)$

The formulae in group 1 need no special treatment. To handle the second group, however, fairness must be considered. Consider the formula $\forall\Diamond\alpha$. Suppose the truth value of the given formula must be determined in state s. Therefore each path leading from state s must lead to a state in which α holds. If some path leads back to s without reaching a state in which α is true, it is necessary to know whether some other path starting at s can lead to a state in which α is true because, if so, the first path represents an unfair path and can be ignored. On the other hand if a state in which α holds cannot be reached from s, the given formula is false in s. To keep track of this is simple if the reachability graph is kept in memory. If the reachability graph is generated on the fly fairness is handled by keeping information about fairness on the stack.

4.4 Subproblem Detection

Sometimes the truth value of a temporal formula depends on the truth value of another temporal formula. For example The CTL formula given in Figure 1 specifies absence of starvation for process 1: whenever process 1 is trying to enter its critical section ($p1 = 1$), it will eventually reach it ($p1 = 2$). Nested formulae can be handled in different ways. A simple method is to break formulae down into subformulae which are then handled in a bottom-up way but much unnecessary work is normally done that way.

The problem can be solved more efficiently in a top-down fashion by computing truth values only when necessary. Consider the CTL formula given in Figure 1 again. To determine the truth value of this formula in state s the model checker will explore all paths leading from s while checking that in each state along every path the argument to $\forall\Box$ is true. The implication makes it unnecessary to determine the truth value of the nested modality in any state in which $p1 \neq 1$. If $p1 = 1$ however, the truth value of $\forall\Diamond(p1 = 2)$ is needed. Tuominen gives a top-down model checking algorithm[12] but the truth values of some subformulae are still computed unnecessarily. For example, the truth value of the nested modality in the given example will be recomputed in all states while it is often possible to deduce its value from some earlier state. Figure 2 provides an example. The truth value of the subformula $\forall\Diamond(p1 = 2)$ will be needed in states $(1, 0, 1)$, $(1, 1, 1)$ and $(1, 2, 0)$. However, since the latter two states are reachable from the state $(1, 0, 1)$,

it is unnecessary to determine the truth value of the nested modality in all three states. If the subformula is true in state $(1, 0, 1)$ it is bound to be true in the other two states. If it is false in state $(1, 0, 1)$ the main formula is invalidated and therefore the other two states can be ignored.

A new technique called *subproblem detection* is proposed which exploits such contextual information to avoid a significant amount of unnecessary processing. Whenever the truth value of some subformula cannot be determined directly in state s the subformula and state s are remembered as a *subproblem* to be analysed at a later stage. The subformula is assumed to be true and the model checker proceeds. In the given example it was necessary to analyse the *same* subformula in three different states. For each unique subformula a list of states is kept in which its truth value must be determined. Several simplifications are now possible. For example, as soon as the specification is found to be violated any subproblems which have been generated need not be analysed any further. Furthermore, the set of states V_s visited in order to compute the truth value of a subformula in state s is recorded in the bit vector. The truth value of the same subformula in some other state s' can then often be deduced from its truth value in s if $s' \in V_s$. The technique has been implemented and found to improve the speed of the model checker significantly.

Another advantage of the technique of subproblem detection is that it provides a natural way to parallelise the model checker: a main processor can be used to detect subproblems while several "worker" processors are used to solve subproblems. Results are returned to the main processor which keeps track of everything in order to determine the final result. Communication overhead is low since little information needs to be exhanged among processors. To solve a particular subproblem a worker processor needs to know only the start state and the particular subformula. The returned result is simply the truth value of the particular subformula in the start state. As an added bonus, a parallel version of the model checker will be using the memory of several machines as a combined resource. States generated in order to detect a particular subproblem need not be regenerated by the processor which is used to solve the subproblem. Similarly different state spaces are usually generated to solve different subproblems. A parallel version of the model checker based on a number of interconnected workstations is currently being developed.

4.5 Using Paging to Save More Space

Traditional model checkers keep information about each unique state in memory in the form of a state graph. For large state spaces this graph will be too large to fit into memory. However, because state generation is so slow when traditional methods are used—typically about 100 states per second—a space problem is not encountered in practice; problems large enough to cause a space problem cannot even be considered because too much processing time will be required.

Model checkers capable of processing at least several thousand states per second on a modern workstation can be built by using the bit vector technique. It

is thus possible to analyse much larger state spaces in an acceptable amount of time. But larger problems require huge virtual address spaces to accomodate the bitvector—too large to be supported directly by currently available operating systems. Fortunately the bit vector is extremely sparse and clustered[8] and therefore multi-level paging techniques can be used to handle large bit vectors. A virtual address space supported by disk slows down the model checker to an unacceptable speed[8] and therefore pages are allocated dynamically in memory as needed. The technique is used to save memory. It depends on the fact that only a small fragment of the virtual address space is usually needed. Results obtained thus far are encouraging but more experimentation will be required to determine the success of this technique.

5 Conclusion

A transition system is used to model the dynamic properties of a system. Figure 1 shows typical input accepted by the model checker. The execution tree of the transition system is generated dynamically while it is being verified whether the given specification is satisfied. To avoid storing the reachability graph explicitly a new technique was developed to handle fairness. To handle nesting another technique (called *subproblem detection*) was developed. It has several other advantages related to model checking in general. A paging technique is used to save space in order to handle even larger problems.

A model checker based on the suggested design has been implemented and used to verify several systems, the largest thus far being a model of the X.21 protocol. Systems which generate no more than a few thousand states can be analysed by using a personal computer, but a more powerful machine is necessary to analyse larger systems. The suggested techniques enable us to process about 3000 states per second on a workstation based on the Motorola 88K processor. Little has been reported about the efficiency of other model checkers except that about 100 states per second seems to be the norm using a typical workstation.

6 Acknowledgements

Several people have contributed ideas to the model checker described in this paper. Specific acknowledgements are due to Professor Chris Brink of the University of Cape Town, and the following people from the University of Stellenbosch: Dieter Barnard, Werner Fouché, Pieter Muller and Willem Visser.

References

[1] M. C. Browne, "An Improved Algorithm for the Automatic Verification of Finite State Systems Using Temporal Logic", in *Proceedings of the Symposium on*

Logic in Computer Science, (Washington D.C.), pp. 260–266, IEEE Computer Society Press, June 16-18 1986.

[2] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond", in *Proceedings of the 5-th IEEE Symposium on Logic in Computer Science*, (Philadelphia), pp. 428–439, June 1990.

[3] E. Clarke and E. Emerson, "Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic", in *Proceedings of IBM Workshop on Logic of Programs*, (D. Kozen, ed.), pp. 52–71, Lecture Notes in Computer Science, 131, 1981.

[4] E. Clarke, E. Emerson, and A. Sistla, "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach", *Proceedings 10th ACM Symposium on Principles of Programming Languages*, pp. 117–126, 1983.

[5] E. A. Emerson and J. Y. Halpern, "'Sometimes' and 'not never' revisited: on branching versus linear time temporal logic.", *Journal of the ACM*, vol. 33, no. 1, pp. 151–178, January 1986.

[6] E. Emerson and C. Lei, "Modalities for Model Checking: Branching Time Strikes Back", *Science of Computer Programming*, no. 8, pp. 275–306, 1987.

[7] H. Everitt, "Temporal Logic as an Aid to Validating Communication Protocols", in *Proceedings of the Australian Software Engineering Conference*, (Canberra), pp. 293–305, May 11-13 1988.

[8] G. J. Holzmann, "An Improved Reachability Analysis Technique", *Software Practice and Experience*, vol. 18, no. 2, pp. 137–161, February 1988.

[9] G. Holzmann, "Algorithms for Automatic Protocol Verification", *AT&T Technical Journal*, pp. 32–44, January/February 1990.

[10] Z. Manna and A. Pnueli, "Completing the Temporal Picture", Research Report STAN-CS-89-1296, Department of Computer Science, Stanford, California 94305, December 1989.

[11] A. Pnueli, "Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends", in *Current Trends in Concurrency*, (J. de Bakker, W. de Roever, and G. Rozenberg, eds.), pp. 510–584, Lecture Notes in Computer Science, 224, Springer Verlag, 1986.

[12] H. Tuominen, "Logic in Petri Net Analysis", Research Report 5, Helsinki University of Technology, Department of Computer Science, Digital Systems Laboratory, Otaniemi, Otakaari 5 A SF-02150 ESPOO, FINLAND, January 1988.

[13] N. Wirth, *Programming in Modula-2*. Springer-Verlag, 2 ed., 1983.