South African
Computer
Journal
Number 8
November 1992

Suid-Afrikaanse
Rekenaar-
tydskrif
Nommer 8
November 1992

Computer Science and Information Systems

Rekenaarwetenskap en Inligtingstelsels

The South African Computer Journal
An official publication of the Computer Society of South Africa and the South African Institute of Computer Scientists

Die Suid-Afrikaanse Rekenaartydskrif
'n Amptelike publikasie van die Rekenaarverenging van Suid-Afrika en die Suid-Afrikaanse Instituut vir Rekenaarwetenskaplikes

Subscriptions

	Annual	Single copy
Southern Africa:	R45,00	R15,00
Elsewhere:	$45,00	$15,00

to be sent to:
Computer Society of South Africa
Box 1714 Halfway House 1685

Computer Science Department, University of Pretoria, Pretoria 0002
Phone: +27 (12) 420-2504 Fax: +27 (12) 43-6454

Sigma Press

Guest Contribution

The paper below was given as an invited address by Prof Roode at the July 1992 Conference of the South African Computer Lectures' Association. (Editor)

The Ideology, Struggle and Liberation of Information Systems

Dewald Roode
Department of Informatics, University of Pretoria

In 1989, Denning *et al* presented the final report of the Task Force on the Core of Computer Science in an article entitled "Computing as a Discipline" [3]. This was said to present a new intellectual framework for the discipline of computing and proposed a new basis for computing curricula.

In the words of the authors, "an image of a technology-based discipline is projected whose fundamentals are in mathematics and engineering." Algorithms are represented as the most basic objects of concern and programming and hardware design as the primary activities. Although there is wide consensus that computer science encompasses far more than programming, the persistent emphasis on programming "arises from the long-standing belief that programming languages are excellent vehicles for gaining access to the rest of the field" [3].

The new framework sets out to present the intellectual substance of the field in a new way, and uses three paradigms to provide a context for the discipline of computing. These paradigms are *theory*, rooted in mathematics; *abstraction*, rooted in the experimental scientific method and *design*, with its roots in engineering.

Programming, the report recommends, should still be a part of the core curriculum and programming languages should be seen and used as vehicles for gaining access to important aspects of computing.

The following short definition is offered of the discipline of computing [3]:

> The discipline of computing is the systematic study of algorithmic processes that describe and transform information: their theory, analysis, design, efficiency, implementation, and application. The fundamental question underlying all of computing is, *"What can be (efficiently) automated?"*

In the same issue of Communications, tucked away towards the end of the journal, an article by Banville and Landry asked the innocent question "Can the Field of MIS be disciplined?" [1]. It is not clear whether the use of the word "discipline" in both articles was purely coincidental – however, the implications were quite clear: computer science was able to talk about "computing as a discipline," and indeed, could present a report which, in a sense, was a culmination of more than twenty years' efforts. Yet, its sister discipline was still asking questions of a very introvertive nature about itself.

It has become quite clear that the fields (leaving aside for the moment the questions of "disciplines") of computer science and information systems (or MIS, informatics, or whatever other name we want to attach to it) have different aims and objectives, different problems that confront it, and, yes, if we want to be truly scientific, different paradigms. To support the latter statement, it is sufficient to contrast the three paradigms of computing with the four paradigms of information systems development described by Hirschheim and Klein [5]. It can be said that a central activity in information systems is the development of information systems, and that therefore, these paradigms have implications for the field of information systems. The four paradigms can be characterized briefly, as follows:

- The analyst as systems expert
- The analyst as facilitator
- The analyst as labour partisan
- The analyst as emancipator or social therapist.

In the same spirit, Lyytinen sees the "systems development process as an instrument in organizational change" [6] and remarks that analysts' principal problems are "in understanding the goals and contents of such change instead of solving technical problems." Already in 1987 Boland [2] observed that: "designing an information system is a moral problem because it puts one party, the designer, in the position of imposing an order on the world of another."

This is clearly a far cry from Denning *et al*'s statement that the fundamental question is "what can be automated?" At the same time, within the context of the field of computing, there is nothing wrong with this question, and it is probably the right question for practitioners of computing to continually ask themselves. But it is a disastrous question for a practitioner of informatics to ask. And it has taken us quite a long time to realise this – that the two disciplines have fundamentally different roles to play. These roles are complementary and supportive, and not destructively opposed.

The liberation of information systems lies in realising this elemental truth: that information systems are man-made objects designed to effect organisational change and that, as such, they can ill be studied using the paradigms of abstraction and engineering mentioned above.

What then is needed? Banville and Landry offer the consolation that we need not concern ourselves too much about the lack of discipline, and that we can indeed even pride ourselves in being a fragmented adhocracy. It is, in fact, even healthy to continue in all sorts of directions. During this process of finding itself, a discipline should be allowed a considerable degree of latitude, and many avenues should be explored. This obviously makes the field of information systems extremely exciting: it is in the process of discovering remarkable truths, discovering that there are in reality people out there using the systems which analysts design and build, and that the most intriguing problems centre around the role of people in all of this: the analyst, the user, their interaction, the impact of sytems on the work lives of workers on all levels, the impact on organizations. These are questions which have mostly been ignored or lightly treated over the years, but which have emerged as *the* problems to be solved. We do not have the tools to solve them – not yet; but a good starting point would certainly be to first understand more about our field and its research tools, for the empirical, positivist approach so often employed will not suffice to solve the above problems.

In the spirit of contributing to the liberation movement of information systems, we have embarked on a study of research on research in Information Systems, and will report on the results more fully in the near future. We define Information Systems as follows [4]:

> Information Systems is an inter-disciplinary field of scholarly inquiry, where information, information systems and the integration thereof with the organisation is studied in order to increase the effectiveness and efficiency of the total system (of technology, people, organisation and society).

In Information Systems then, we see the fundamental question underlying the entire discipline, to be the problem of balancing the need to contribute, through information systems, to the achievement of the mission of the organisation with the moral responsibility to develop and implement socially accepted information systems.

Each of the fields, computer science and information systems, benefits enormously from the activities of the other. Nonetheless, we must recognize the different approaches used by the two disciplines and allow them to complement each other. It should not be our business to convince one another that the universal truth is that which we use in our discipline – whether that be computer science or information systems. Instead, we should seek out the opportunities for synergy, and for complementing each other. If we succeed in doing this at SACLA, then we could indeed do ourselves proud.

References

1. C Banville and M Landry. 'Can the field of MIS be disciplined?'. *Communications of the ACM*, **32**(1):48–60, (1989).
2. R J Boland and R A Hirschheim, eds. *Critical Issues in Information Systems Research*. John Wiley & Sons Ltd., 1987.
3. P J Denning, D E Comer, D Gries, M C Mulder, A Tucker, A J Turner, and P R Young. 'Computing as a discipline'. *Communications of the ACM*, **32**(1):9–23, (1989).
4. N F Du Plooy, L D Introna, and J D Roode. 'Notes on research in information systems'. Unpublished research report, Department of Informatics, University of Pretoria, (1992).
5. R Hirschheim and H K Klein. 'Four paradigms of information systems development'. *Communications of the ACM*, **32**(10):1199–1216, (1989).
6. K Lyytinen. 'New challenges of systems development: A vision of the 90's'. *Data Base*, pp. 1–12, (Fall 1989).

Editor's Notes: To Compete or Collaborate

Human interaction invariably brings with it a blend of competition and collaboration. Competition means that one enjoys the exhilaration of winning while the other endures the shame of loosing. Because of this reward/punishment mechanism, it is a widely assumed that competition enhances performance and efficiency. This dogma pervades not only commerce, sport and politics, but is found in practically all areas of human endeavour, including research.

The competitive spirit in research is found in the well-known saga of Watson and Crick racing to unravel the double helix structure of DNA. Not so well-known, though equally illustrative, is the intensity of Newton's stratagems to oust Leibnitz from receiving any credit for differentiation. Recently there have been reports of scientists who have either tolerated or manufactured fraudulent results in order to win some or other scientific race. The space race, the arms race, the race for an AIDS cure, the scurry for faster smaller hardware, the race for awards, the drive for publications, Nobel prizes: all of this attests to a profoundly competitive international research culture.

But while competition might be the handmaiden of commerce and sport, it is the harlot of research – an unfortunate concomitant of the silly side of human nature. The archetypal researcher not only rises above the incidentals of human accolades; he disdains them. By tradition, the definitive research qualification is a PhD – a Doctor of Philosophy – a lover of thought. Discovery and thought are not only by their very nature rewarding, they are also humbling. When the archetypal researcher moves outside his interior thought-world, it is to share his discoveries. If he is childish, it is not the little boy flexing his biceps and saying: "I'm stronger than you" but the child rushing to

tell everyone: "Wow – look at this!" He is forgetful of self: Pythagoras, oblivious of the invading enemy and his impending death while he researches in the sand; Archimedes shouting "Eureka" without care for his nudity. The competitive spirit is a crass intrusion into this ancient legacy of innocence and selflessness.

By its nature, collaboration thrives in a climate of easy social intercourse. It may initially feel uncomfortable for researchers, who are inclined to be socially inept and are wont to bury themselves in work away from society. However, once the plunge to collaborate is taken there is ample evidence that it leads to successful research. In maximizing the use of available talent, it brings about a synergy in which two heads are better than one. All participants enjoy its rewards and no individual has to endure the full weight of its failures. In fact, the notion of collaboration is now so commonplace that significant research seems impossible without it. The tendency, however, is to encourage research collaboration within an organisation, but to emphasize competition in relation to outside organisations.

During a forum discussion at the July South African Computer Lecturers' Association (SACLA) conference, an appeal was made for greater collaboration between universities. Not surprisingly, the information technology disciplines at local universities have always had both a competitive and a collaborative relationship. The competitiveness usually takes the form of friendly rivalry, while the very existence of SACLA bears testimony to a rather unique collaborative relationship. In latter years the competitiveness seems to have intensified, while electronic mail and other developments have improved the prospects for collaboration. At issue, then, is whether there is an imbalance between these dual forces. The appeal at the SACLA forum implied that there is, and I would strongly agree. It is my view (my prejudice, if you will) that competition between universities is a self-indulgent and wasteful dissipation of energy.

Those who are inclined to compete should seriously examine what is to be gained. It is unconvincing to argue that winning makes a significant impact on the way in which students select universities: in the main, this is a matter of geography and language preference. To some extent, the same might be said about staff, although research reputation perhaps plays a more important role here. Neither are research funding agencies (e.g. the FRD) influenced by whether X is "better" in some or other sense than Y. On the contrary, it has wisely been decided to fund on the basis of criteria that are believed to be objective, without any reference whatsoever to the performance of competitors. True enough, funds are limited, but it is precisely for this reason that it is wasteful to divide the little there is between divergent research efforts.

It seems to me that there is a wealth of research talent out there, but that each researcher selects an area of interest almost as a matter of whim. There is an urgent need for well-coordinated collaboration on focussed research areas that have been carefully selected as directly relevant to the country. It is especially incumbent on those who finance, manage and lead research to identify such areas and to encourage collaboration in every possible way.

I look forward to the manifestation of such collaboration in SACJ publications authored by researchers from different university departments. To date there have been none of consequence. If we fail to collaborate, we are in danger of becoming little Don Quixotes who spend our lives attacking windmills and defending castles of xenophobia and irrelevance.

A Model Checker for Transition Systems

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

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.*

Keywords: *Verification, Transition Systems, Branching time temporal logic, Model checking*
Computing Review Categories: *D2.1, D2.4, F3.1*

Received January 1992, Accepted August 1992.

1 Introduction

Model checking was introduced by Clarke *et al.* [4, 5]. 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 reactive system is represented by a transition system which is used to generate all possible execution sequences of the system. The branching time temporal logic CTL [4] is used as a specification language to specify various properties the system should have. In principle model checking is a powerful verification technique. The theoretical foundations of model checking have been investigated thoroughly [7, 6, 14] 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 [9] 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 Using a Transition System as Computational Model

The transition system used here is similar in spirit to the system described in [13]. 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

```
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

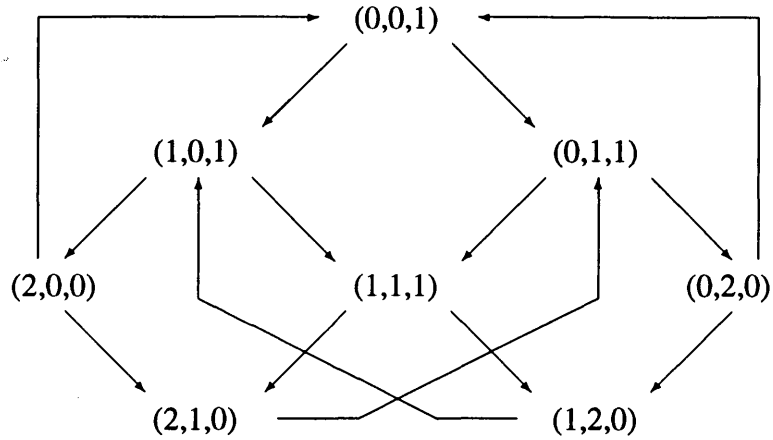

Figure 2. Reachability graph of a transition system

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 CTL formula following the keyword "SPEC" captures a property that should hold for this particular transition system as will be explained in Section 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 [4]. 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.

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 below an *until* operator is introduced to increase the expressiveness of the language. Roughly, "$\alpha\ U\ \beta$" 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.

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: $s \models true$, and $s \not\models false$; 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 $(t_0, t_1, t_2, \ldots)$ from $s(= t_0)$ 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 can now be introduced by definition: $\forall\Diamond\alpha$ iff $\forall[true\,\mathcal{U}\,\alpha]$; $\exists\Diamond\alpha$ iff $\exists[true\,\mathcal{U}\,\alpha]$; $\forall\Box\alpha$ iff $\neg\exists\Diamond\neg\alpha$ and $\exists\Box\alpha$ iff $\neg\forall\Diamond\neg\alpha$.

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) 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 [8]. A list of properties which are relevant for reactive systems is given in [12].

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 [14] 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 effi-

cient 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 [9]. 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 [4, 5, 1, 14] 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.* [3] 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, efficient symbolic representations do not always exist [2] and more research will be needed to determine the effectiveness of the technique in general.

Recently Holzmann found a new method of searching large state spaces [9, 10] 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 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.

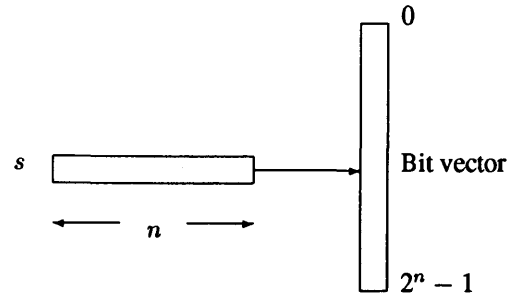

Figure 3. The bit vector technique

Jard *et. al.* also developed a technique to generate the reachability graph dynamically [11]. However, they used a linear time logic and suggested that it could be difficult to transport the idea towards branching time logics. To do it, two problems had to be solved, namely, how to handle fairness constraints and nested formulae *without storing the reachability graph explicitly* as will be explained below.

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 (the contents of the state vector $s = (p1, p2, x)$) is viewed as a (unique) index into a large array of bits (called a *bit vector*) as illustrated in Figure 3. For example, the transition system shown in Figure 1 generates the states $(0,0,1)$, $(1,0,1)$, $(0,1,1)$, $(2,0,0)$, $(1,1,1)$, $(0,2,0)$, $(2,1,0)$ and $(1,2,0)$ with corresponding indices 1, 9, 3, 16, 11, 4, 18 and 12 into the bit vector.

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. 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.

```
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

Model Checking Algorithm

The algorithm is presented in (more or less) standard Modula-2, for which see [15].

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 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).
- The mode is changed depending on the truth value of *sf* for every unique state generated. 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.
- 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.

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 π leads back to s without reaching a state in which α is true, it seems to invalidate the property $\forall\Diamond\alpha$. However, when we insist on a fair transition system and there is another path π' leading from s

to a state in which α holds, *this other path must eventually be followed.* We may therefore circulate a finite number of times through the path π without reaching a state in which α holds, but eventually path π' will lead us to a state in which α holds. When a potentially unfair path (leading from s back to s without satisfying α) is discovered it is thus necessary to know whether some other path starting at s can lead to a state in which α holds 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.

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 [14] 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.

Handling Large State Spaces

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 may require several hours of processor time and therefore cannot even be considered.

It is possible to analyse much larger state spaces in an acceptable amount of time by using the bit vector technique. But for larger problems the bit vector quickly becomes too big (as dictated by the larger state vector) to accommodate directly. Holzmann uses a hashing technique to map large state vectors onto a bit vector of an acceptable size [9]. Unfortunately there is a price to be paid: because hash collisions cannot be detected some errors may be missed.

Because the bit vector is extremely sparse and clustered, virtual memory techniques can be considered to handle large bit vectors. However, a virtual address space supported by disk was found to be too inefficient [9].

We use a different technique: instead of implementing a virtual memory system (supported by disk), paging is used as an *addressing* mechanism to enable us to allocate *only the (small) portion of the bit vector that is actually referenced.* The state vector is viewed as a virtual address that

Table 1. Performance of Paging Technique

Unique states	Memory (bytes) allocated	Fraction of potential memory needed
644	1154.5K	0.76
1042	1872.5K	0.23
1686	3054.5K	0.09
2728	5022K	0.04
4414	8349.5K	0.02

is mapped to a physical address (a 32-bit pointer) through a series of page tables. The state vector is thus seen as an index into a large (virtual) bit vector which is subdivided into pages. The advantage is that only pages which are actually referenced need be allocated. This technique works surprisingly well because the state space tends to be clustered and extremely sparse. Thus the speed advantages of the bit vector technique can be retained without resorting to hashing.

The well-known *dining philosophers* problem (combined with a specification that forces the model checker to generate every possible state) was used to measure the efficiency of the paging technique. This problem was selected for two reasons: it causes sub-optimal behaviour because states are evenly distributed throughout the potential state space and the number of states generated can be controlled easily by changing the number of philosophers. The actual amount of memory allocated was measured for a number of problems of different size. In Table 1 this is expressed as a fraction of the potential memory needed if the bit vector were to be allocated directly without using paging. Although in the largest case only a few thousand unique states can be reached, the potential state space is larger than 10^9. The bit vector in this case would be far too big to be allocated directly—more than 500 Mbytes! The paging technique makes it possible to handle this problem in just over 8 Mbytes of memory.

5 Conclusion

A transition system is used to model the dynamic properties of a system. Figure 1 shows typical low-level input accepted by the model checker. In practice a special high-level modelling language is translated to this low-level form. The execution tree corresponding to 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 had to be developed to handle fairness. To handle nesting another technique (called *subproblem detection*) was developed. We propose a new memory management technique as an alternative to hashing to handle large problems.

A model checker based on the suggested design has been implemented and used to verify several systems. Systems which generate no more than a few thousand states can be analysed by using a personal computer but a more powerful machine (more memory) is necessary to analyse larger systems. The model checker can process about 3000 states per second on a workstation based on the Motorola 88K processor. Little has been reported about the efficiency of comparable model checkers but 100 states per second seems to be the norm.

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*, pp. 260–266, Washington D.C., (June 16-18 1986). IEEE Computer Society Press.
2. R Bryant. 'Graph-Based Algorithms for Boolean Function Manipulation'. *IEEE Transactions on Computers*, C-35(8):677–691, (August 1986).
3. 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*, pp. 428–439, Philadelphia, (June 1990).
4. E Clarke and E Emerson. 'Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic'. In D Kozen, ed., *Proceedings of IBM Workshop on Logic of Programs*, pp. 52–71. Lecture Notes in Computer Science, 131, (1981).
5. 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).
6. E A Emerson and J Y Halpern. ''Sometimes' and 'not never' revisited: on branching versus linear time temporal logic.'. *Journal of the ACM*, 33(1):151–178, (January 1986).
7. E Emerson and C Lei. 'Modalities for Model Checking: Branching Time Strikes Back'. *Science of Computer Programming*, (8):275–306, (1987).
8. H Everitt. 'Temporal Logic as an Aid to Validating Communication Protocols'. In *Proceedings of the Australian Software Engineering Conference*, pp. 293–305, Canberra, (May 11-13 1988).
9. G Holzmann. 'An Improved Reachability Analysis Technique'. *Software Practice and Experience*, 18(2):137–161, (February 1988).
10. G Holzmann. 'Algorithms for Automatic Protocol Verification'. *AT&T Technical Journal*, pp. 32–44, (January/February 1990).
11. C Jard and T Jeron. 'On-Line Model Checking for Finite Linear Temporal Logic Specifications'. In J Sifakis, ed., *International Workshop on Automatic Verification Methods for Finite State Systems*, pp. 189–196, Grenoble, France, (June 12-14 1989).
12. Z Manna and A Pnueli. 'Completing the Temporal Picture'. Research Report STAN-CS-89-1296, Department of Computer Science, Stanford, California 94305, (December 1989).
13. A Pnueli. 'Applications of temporal logic to the specification and verification of reactive systems: A survey

of current trends'. In J de Bakker, W de Roever, and G Rozenberg, eds., *Current Trends in Concurrency*, pp. 510–584. Lecture Notes in Computer Science, 224, Springer Verlag, (1986).
14. 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).
15. N Wirth. *Programming in Modula-2*. Springer-Verlag, 2nd edition, 1983.

Acknowledgements: I wish to thank Dr Gerard Holzmann of AT & T for valuable discussions about the techniques used to implement this model checker and Prof Chris Brink of the University of Cape Town for advice regarding the logic CTL. I am also grateful to the first serious users of the system: Dieter Barnard, Pieter Muller and Willem Visser of the University of Stellenbosch—they completed several non-trivial case studies and found a number of implementation errors.

Notes for Contributors

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research papers. However, non-refereed review and exploratory articles of interest to the journal's readers will be considered for publication under sections marked as Communications or Viewpoints. While English is the preferred language of the journal, papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted in triplicate to the editor.

Form of Manuscript

Manuscripts for *review* should be prepared according to the following guidelines.

- Use wide margins and $1\frac{1}{2}$ or double spacing.
- The first page should include:
  - title (as brief as possible);
  - author's initials and surname;
  - author's affiliation and address;
  - an abstract of less than 200 words;
  - an appropriate keyword list;
  - a list of relevant Computing Review Categories.
- Tables and figures should be numbered and titled. Figures should be submitted as original line drawings/printouts, and not photocopies.
- References should be listed at the end of the text in alphabetic order of the (first) author's surname, and should be cited in the text in square brackets [1, 2, 3]. References should take the form shown at the end of these notes.

Manuscripts accepted for publication should comply with the above guidelines (except for the spacing requirements), and may be provided in one of the following formats (listed in order of preference):

1. As (a) LaTeX file(s), either on a diskette, or via e-mail/ftp – a LaTeX style file is available from the production editor;
2. As an ASCII file accompanied by a hard-copy showing formatting intentions:
   - Tables and figures should be on separate sheets of paper, clearly numbered on the back and ready for cutting and pasting. Figure titles should appear in the text where the figures are to be placed.
   - Mathematical and other symbols may be either handwritten or typed. Greek letters and unusual symbols should be identified in the margin, if they are not clear in the text.

   Further instructions on how to reduce page charges can be obtained from the production editor.
3. In camera-ready format – a detailed page specification is available from the production editor;
4. In a typed form, suitable for scanning.

Charges

Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect scanning, typesetting, reproduction and other costs. Currently, the minimum rate is R20-00 per final page for LaTeX or camera-ready contributions and the maximum is R100-00 per page for contributions in typed format.

These charges may be waived upon request of the author and at the discretion of the editor.

Proofs

Proofs of accepted papers in categories 2 and 4 above will be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Note that, in the case of camera-ready submissions, it is the author's responsibility to ensure that such submissions are error-free. However, the editor may recommend minor typesetting changes to be made before publication.

Letters and Communications

Letters to the editor are welcomed. They should be signed, and should be limited to less than about 500 words.

Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book reviews

Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement

Placement of advertisements at R1000-00 per full page per issue and R500-00 per half page per issue will be considered. These charges exclude specialized production costs which will be borne by the advertiser. Enquiries should be directed to the editor.

References

1. E Ashcroft and Z Manna. 'The translation of 'goto' programs to 'while' programs'. In *Proceedings of IFIP Congress 71*, pp. 250–255, Amsterdam, (1972). North-Holland.
2. C Bohm and G Jacopini. 'Flow diagrams, turing machines and languages with only two formation rules'. *Communications of the ACM*, 9:366–371, (1966).
3. S Ginsburg. *Mathematical theory of context free languages*. McGraw Hill, New York, 1966.

South African Computer Journal

Number 8, November 1992
ISSN 1015-7999

Suid-Afrikaanse Rekenaar-tydskrif

Nommer 8, November 1992
ISSN 1015-7999

Contents

Sigma Press