Model Checking

Automated Verification of Computational Systems

Madhavan Mukund

The ACM Turing Award for 2007 was awarded to Clarke, Emerson and Sifakis for their invention of model-checking, an automated technique for verifying finite-state computing systems. In this article, we describe the central ideas underlying their approach.

Madhavan Mukund is Professor at Chennai Mathematical Institute. His research interests are primarily in formal methods for specifying and verifying computing systems. He is the Secretary of the Indian Association for Research in Computing Science (IARCS) and the National Coordinator of the Indian Computing Olympiad.

1. Introduction

Each year, the Association for Computing Machinery (ACM) confers the A M Turing Award for "contributions of lasting and major technical importance to the computer field". This is the most prestigious award in computing science and can be considered the equivalent of the Nobel Prize for the subject.

The Turing award for 2007, announced in 2008, was shared by Edmund M Clarke (Carnegie-Mellon University, USA), E Allen Emerson (University of Texas at Austin, USA) and Joseph Sifakis (Verimag, Grenoble, France) for their "role in developing model-checking into a highly effective verification technology, widely adopted in the hardware and software industries."

In this article, we give a quick introduction to the central ideas behind model-checking.

2. Verification of Programs

The need to verify the correctness of programs is as old as programming itself. From the 1960s, it was realized that mathematical logic provides a suitably precise language to formulate properties of programs. The method proposed was to attach appropriate logical assertions about the values of variables and the relationships

Keywords
Automated verification, model checking, temporal logic.

In Hoare logic, each piece of program text *C* is annotated by two logical assertions, the precondition and the postcondition.

between them to different points in the text of a program and use these assertions to prove the program correct.

This approach was formalized by C A R Hoare, building on ideas of Robert Floyd, in a system called 'Hoare logic'. In Hoare logic, each piece of program text C is annotated by two logical assertions, the precondition and the postcondition, yielding a *Hoare triple* written as $\{Pre\}C\{Post\}$. The interpretation is that if Pre holds before the execution of C, then $Post$ is guaranteed to hold after the execution of C. Given a sequence of program blocks $C_1, C_2, \ldots, C_k$, one can then string together a proof of correctness by showing that $\{Pre_i\}C_i\{Post_i\}$ is a valid Hoare triple for each block C_i and that $Post_i$ implies Pre_{i+1} for each pair of consecutive blocks.

The main difficulty with this approach lies in automating the logical reasoning involved. Automation is required for the technique to scale up to large programs. Independently, there are difficulties applying this technique to concurrent programs, where components execute independently, communicating through mechanisms such as shared variables or message channels.

2.1 *Temporal Logic*

In 1977, Amir Pnueli proposed an alternative approach to reason about concurrent programs using another logical formalism called 'temporal logic'[1]. In Hoare logic, the proof of correctness works with the actual program text. Pnueli's approach was to build an abstract model of the program, capturing the features essential for verification. These features are then encoded as logical propositions. As the program executes, it goes through a sequence of different states, each described by a different combination of these logical propositions.

[1]Pnueli received the Turing award in 1996 for this contribution.

The main difficulty with this approach lies in automating the logical reasoning involved.

An important aspect of Pnueli's approach is that it is intended to deal with systems that are not expected to

terminate, such as operating systems, schedulers and controllers. For such *reactive systems*, each execution of the system normally generates an infinite sequence of states.

Temporal logic is a language for asserting properties about logical propositions as they vary over a sequence of states.

Temporal logic is a language for asserting properties about logical propositions as they vary over a sequence of states. For instance, in temporal logic we can write formulas of the form "in a future state f holds" and "henceforth f holds in all states".

There are two main flavours of temporal logic, linear-time and branching-time. In linear-time temporal logic, which was Pnueli's original formulation, each execution of the system is analyzed independently. In this interpretation, a system satisfies a formula f, if f holds along every execution.

Another approach is to combine all possible executions of the system into a single (infinite) tree. Each path in the tree represents one possible execution and the branching points capture the nondeterministic choices. These choices typically arise from the fact that we are dealing with concurrent programs, so we cannot always fix the order in which events occur across independent components in the system. In branching-time logic, assertions are interpreted over this single computation tree representing the overall behaviour of the system. Temporal assertions are quantified with respect to paths, so we can say "over all paths f holds" and "there exists a path over which f holds".

We are dealing with concurrent programs, so we cannot always fix the order in which events occur across independent components in the system.

2.2 *Model Checking*

In mathematical logic, a traditional question is satisfiability – given a formula f, is there a structure in which f is true? However, the application of logic in verification gives rise to another natural question – given a structure M and a formula f, is f true in M? This corresponds to asking if a program meets its specification and is now

known as the model-checking problem[2].

[2]The term 'model-checking' was first used by Clarke and Emerson in [1].

The primary contribution of Clarke, Emerson and Sifakis [1, 2] was to pose model-checking as an algorithmic problem and provide an efficient solution in the setting of reactive systems with respect to temporal logic assertions.

In the rest of the article, we illustrate their approach using an example. We first show how to extract an abstract model from a concurrent program. We then formalize a version of branching-time temporal logic and describe how the corresponding model-checking algorithm works.

The primary contribution of Clarke, Emerson and Sifakis was to pose model-checking as an algorithmic problem and provide an efficient solution in the setting of reactive systems with respect to temporal logic assertions.

3. Modelling Concurrent Programs

Our running example will be Peterson's algorithm for solving the problem of mutually exclusive access to a shared resource by two concurrent processes [3]. The processes can communicate with each other using shared variables. However, no assumption is made about how the actions of the two processes interleave. For instance, if a process decides to move based on the current value of a shared variable, it may happen that the value of the variable is changed by the other process between the time it is tested and the next action is taken in the first process.

Peterson's algorithm is shown in *Figure* 1. The processes are numbered 0 and 1. The solution uses three shared

Process 0	Process 1
while($true$){	while($true$){
$request_0 \leftarrow true$;	$request_1 \leftarrow true$;
$turn \leftarrow false$;	$turn \leftarrow true$;
while ($request_1 \wedge \neg turn$){	while ($request_0 \wedge turn$){
// "Busy" wait	// "Busy" wait
}	}
// Enter critical section	// Enter critical section
// ...	// ...
// Leave critical section	// Leave critical section
$request_0 \leftarrow false$;	$request_1 \leftarrow false$;
}	}

Figure 1. Peterson's algorithm for mutual exclusion.

variables, $request_0$, $request_1$ and *turn*. The algorithm describes what Clarke and Emerson call the "synchronization skeleton" of the system – it only reflects the mechanism to coordinate mutually exclusive access to the shared resource, abstractly referred to as the 'critical section', without bothering about the actual nature of this shared resource. Notice that each process is in an infinite loop, signifying that this represents a reactive system that is not expected to terminate in a finite time.

The "synchronization skeleton" of the system reflects the mechanism to coordinate mutually exclusive access to the shared resource.

In our abstract model of this system, each state records the values of the three shared variables. In addition, we associate three phases with each process, *idle*, *trying* and *critical*. Initially both processes are idle. After assigning the variables $request_i$ and *turn*, process i goes from the idle phase to the trying phase, which corresponds to the "busy wait" loop in the program text that checks for appropriate values of *turn* and the other process's *request*. If the test succeeds, the process enters the critical section and hence the critical phase of its activity. On completing this phase, it resets $request_i$ and returns to the idle phase.

Thus, the state of the system is captured by nine Boolean values $\{r_0, r_1, u, i_0, i_1, t_0, t_1, c_0, c_1\}$. The values $\{r_0, r_1, u\}$ correspond to the shared variables $request_0$, $request_1$ and *turn*, respectively. The values $\{i_j, t_j, c_j\}$ describe which of the three phases – idle, trying or critical – process j is in.

We can represent the different states of the program in terms of a diagram, as shown in *Figure* 2. Each state is labelled by the Boolean values true in that state. Any value not shown is implicitly false in the given state. The arrows indicate the transitions that occur between states when one of the processes executes a statement. Solid arrows correspond to moves of process 0 while dashed arrows correspond to moves of process 1.

For instance, the transition from the state $\{i_0, i_1\}$ to the state $\{r_0, i_0, i_1\}$ corresponds to the initial statement in the loop where process 0 sets $request_0$ to *true*. Similarly, the transition from $\{r_0, r_1, t_0, i_1\}$ to $\{r_0, r_1, u, t_0, t_1\}$ represents the step where process 1 sets *turn* to *true* and enters its trying phase (process 0 is already in its trying phase when this transition occurs). Notice that in states $\{r_0, r_1, u, t_0, t_1\}$ and $\{r_0, r_1, t_0, t_1\}$, where both processes are simultaneously in the trying phase, only one of the two processes can move. This graphically illustrates the clever way in which the three shared variables combine in Peterson's algorithm to guarantee mutual exclusion.

4. Branching-time Temporal Logic

We now define a temporal logic that can be used to assert properties of abstract models like the one in *Figure* 2. As we had mentioned earlier, we can work either in a linear-time framework or in a branching-time framework. Our exposition is based on the branching-time temporal logic used by Clarke and Emerson in [1] because it admits one of the most direct algorithms for the model-checking problem[3].

[3] Queille and Sifakis used a closely related logic in [2].

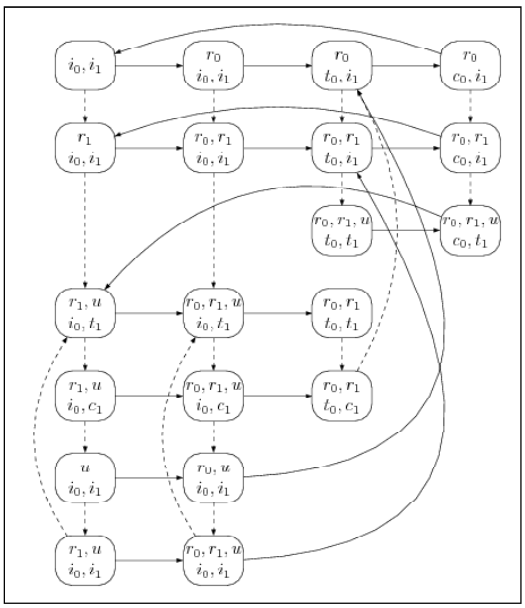

Figure 2. An abstract model of Peterson's algorithm.

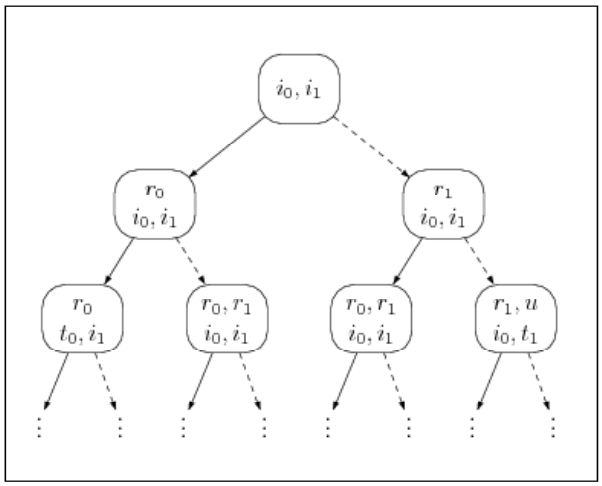

Figure 3. The computation tree for Peterson's algorithm.

The first step is to unwind our abstract model into a computation tree. Let us assume that the system in *Figure* 2 starts execution in the state $\{i_0, i_1\}$. From this state it can move to two different states, depending on whether process 0 or process 1 moves. From each of these states, we again have two possible moves. If we systematically explore the states we can reach, we obtain the tree shown in *Figure* 3. In the tree, we make a fresh copy of each state that we encounter. Notice, for example, that at the third level we have two separate copies of the state $\{r_0, r_1, i_0, i_1\}$ since these are reached by two different execution sequences of the global system. Each path in the tree describes one possible evolution of the system. Since the original program is designed to run forever, paths in the tree are, in general, infinite.

The computation tree is a single structure capturing the complete behaviour of the model. For instance, to identify if it is possible for both processes to be simultaneously in the trying phase, it suffices to check if a state containing $\{t_0, t_1\}$ is reachable in the tree. To further verify if the system can deadlock – for example, both processes are in the trying phase but each is stuck waiting for a condition to become true, so no progress can be made – we can check whether there is a node in the

The computation tree is a single structure capturing the complete behaviour of the model.

Branching-time temporal logic is a formalism for expressing properties of the computation tree generated by an abstract model.

tree that has no successor nodes. In other words, a system never deadlocks if and only if every path in its computation tree is infinite.

Branching-time temporal logic is a formalism for expressing properties of the computation tree generated by an abstract model. The specific version of branching-time temporal logic that we work with is called Computation Tree Logic, or CTL.

The basic assertions in CTL are the logical propositions that we use to describe the states of the system – in the case of Peterson's algorithm, we have nine basic assertions $\{r_0, r_1, u, i_0, i_1, t_0, t_1, c_0, c_1\}$.

Formulas in CTL are evaluated at states in the computation tree. Remember that each state is defined in terms of the logical propositions that hold in that state. Thus, a basic assertion is true at a state whenever the corresponding logical proposition holds at that state. For instance, at the root node of the tree in *Figure* 3, the basic assertions i_0 and i_1 are true and all other basic assertions are false.

We can qualify formulas with *temporal modalities* asserting where the formula holds along a path. These come in two basic forms.

- **Next** The formula $\mathsf{X}f$ holds at a state s along a path in the computation tree if f holds at its successor state along the path.
- **Until** The formula $f\mathsf{U}g$ holds at a state s along a path in the computation tree if g holds at some future state along the path and, starting at the current state, f holds at each state until the state where g holds. Note that $f\mathsf{U}g$ holds in a trivial way at s if g already holds at s.

We can qualify formulas with *temporal modalities* asserting where the formula holds along a path.

Two useful special cases of the *Until* modality are worth

recording explicitly.

- **Eventually** The formula $\mathsf{F}f$ holds at a state s along a path in the computation tree if f holds at some future state along the path.

- **Henceforth** The formula $\mathsf{G}f$ holds at a state s along a path in the computation tree if f holds at every future state along the path.

A temporal formula is evaluated with respect to a fixed *path* in the tree. In CTL, every temporal modality must be accompanied by a *path quantifier* describing whether the property holds along all paths or along some path. We write A to denote the quantifier "over all paths" and E to denote the quantifier "there exists a path".

A temporal formula is evaluated with respect to a fixed *path* in the tree.

Once we pair up each a temporal formula with a path quantifier, we are back to evaluating formulas at states rather than over paths. Thus, for example, $\mathsf{AX}f$ holds at a state s in the tree if f holds in every successor state of s, and $\mathsf{E}f\mathsf{U}g$ holds at a state s if there is some path originating at s along which $f\mathsf{U}g$ holds.

We are, of course, also allowed to combine formulas using the usual Boolean connectives (and ($\wedge$), or ($\vee$) and not ($\neg$)), as well as derived connectives such as logical implication ($\Rightarrow$). Note, however, that the strong restriction in CTL requiring us to pair path quantifiers with temporal modalities prevents us from directly writing assertions such as $\mathsf{E}(\mathsf{GF}f)$, which captures the property that there is a path along which f holds infinitely often – $\mathsf{GF}f$ says that at every point along the path, there is a further point where f holds, so after every f we must see one more f in the future, yielding an infinite subsequence of states where f holds.

Here are some examples of properties that we can express in CTL.

- **Deadlock** Recall that a state is deadlocked if it has no successor state. Let $\mathtt{tt}$ denote the formula "true" which is true at every state (for instance, $\mathtt{tt}$ can be an abbreviation for the tautology $r_0 \vee \neg r_0$). Then $\mathsf{EX}\mathtt{tt}$ asserts that a state has a successor, from which it follows that $\neg\mathsf{EX}\mathtt{tt} \equiv \mathsf{AX}\neg\mathtt{tt}$ asserts that a state has no successor. Thus the formula $\mathsf{EF}(\mathsf{AX}\neg\mathtt{tt})$ holds at the root of the computation tree precisely when there is a reachable deadlocked state.

- **Progress** We would like to ensure that whenever a process attempts to enter the critical region, it eventually makes progress and succeeds. We have used the logical propositions t_i and c_i to denote that process i is in its trying phase and its critical phase, respectively. The progress property we want for process i is captured by asserting the temporal formula $t_i \mathsf{U} c_i$ along every path originating from each state where t_i holds. This is equivalent to requiring that the logical implication $t_i \Rightarrow \mathsf{A}t_i\mathsf{U}c_i$ holds at every state in the tree. Thus, Peterson's algorithm satisfies the progress condition if the formula $\mathsf{AG}(t_0 \Rightarrow \mathsf{A}t_0\mathsf{U}c_0) \wedge \mathsf{AG}(t_1 \Rightarrow \mathsf{A}t_1\mathsf{U}c_1)$ holds at the root of the computation tree.

5. Model-checking CTL Formulas

We now have all the ingredients available to formulate and describe the model-checking problem for CTL. We assume we have an abstract model of the system at hand – a graph consisting of states and transitions in which each state is labelled by the logical propositions that hold at that state, like the model for Peterson's algorithm in *Figure* 2. We fix a state $\hat{s}$ in this model M and ask whether a given CTL formula f holds at the root of the computation tree obtained by unravelling the model M starting with state $\hat{s}$.

Recall that the computation tree rooted at $\hat{s}$ is typically an infinite object, so we cannot exhaustively verify whether formula f holds by explicitly constructing this tree and examining all its paths. Instead, we proceed by observing that we can reduce the question of checking whether a formula holds at a state to questions about its constituent parts, or *subformulas*. For instance, a formula of the form $f \wedge g$ holds at a state s if and only if both f and g, which are subformulas, hold at s. This observation forms the basis for an inductive procedure by which we label the model with subformulas of f so that a state s is labelled by a subformula g of f whenever g holds at f.

We can reduce the question of checking whether a formula holds at a state to questions about its constituent parts, or *subformulas*.

The simplest subformulas of f are the basic assertions corresponding to the logical propositions attached to states in the abstract model. Since we have explicit information about which logical propositions are true in every state, it is a simple matter to label each state by the set of basic assertions that hold at that state.

The Boolean connectives $\wedge$, $\vee$ and $\neg$ are easy to handle. A state s can be labelled with the formula $g \wedge g'$ provided it has already been labelled with both g and g'. A state s can be labelled with the formula $g \vee g'$ provided it has already been labelled with either g or g'. A state s can be labelled with the formula $\neg g$ provided we have finished assigning the label g to all states and the state s has not been labelled g.

The interesting point is that this labelling algorithm can be extended to temporal assertions in CTL.

- $\mathsf{EX}g$, $\mathsf{AX}g$: We label a state s with the subformula $\mathsf{EX}g$ provided it has some successor that has already been labelled with g. Similarly, we label a state s with the subformula $\mathsf{AX}g$ provided that every successor of s has already been labelled with g.

- $\mathsf{E}g\mathsf{U}g'$: There are two cases to consider.

1. If a state s has already been labelled with g', we can immediately assert that $g\mathsf{U}g'$ holds along every path originating at s, so we directly label s with the formula $\mathsf{E}g\mathsf{U}g'$.

2. Otherwise, we observe that it must be the case that g holds at s and $g\mathsf{U}g'$ holds along some path originating from s. This, in turn, amounts to requiring that $\mathsf{E}g\mathsf{U}g'$ holds at some successor of s. In other words, if g' does not hold at s, then the formula $g \wedge \mathsf{EX}(\mathsf{E}g\mathsf{U}g')$ must hold at s in order for $\mathsf{E}g\mathsf{U}g'$ to hold at s.

Putting these two observations together, we have the following iterative procedure to label states with a subformula of the form $\mathsf{E}g\mathsf{U}g'$.

1. For every state s labelled with g', add the label $\mathsf{E}g\mathsf{U}g'$.

2. For every state s labelled with g that has a successor state labeled $\mathsf{E}g\mathsf{U}g'$, label s with $\mathsf{E}g\mathsf{U}g'$.

3. Repeat Step 2 until no further states are labelled $\mathsf{E}g\mathsf{U}g'$.

Each iteration of Step 2 propagates the label to one more state. This procedure must terminate in a finite number of steps because we only have a finite number of states in our model to which the label $\mathsf{E}g\mathsf{U}g'$ can be added.

$\mathsf{A}g\mathsf{U}g'$: This case is similar to the previous one. Here we observe that $\mathsf{A}g\mathsf{U}g'$ holds a state s if either g' holds at s or g holds at s and $\mathsf{A}g\mathsf{U}g'$ holds at every successor of s. This immediately yields an iterative labelling procedure like the one we just described for $\mathsf{E}g\mathsf{U}g'$.

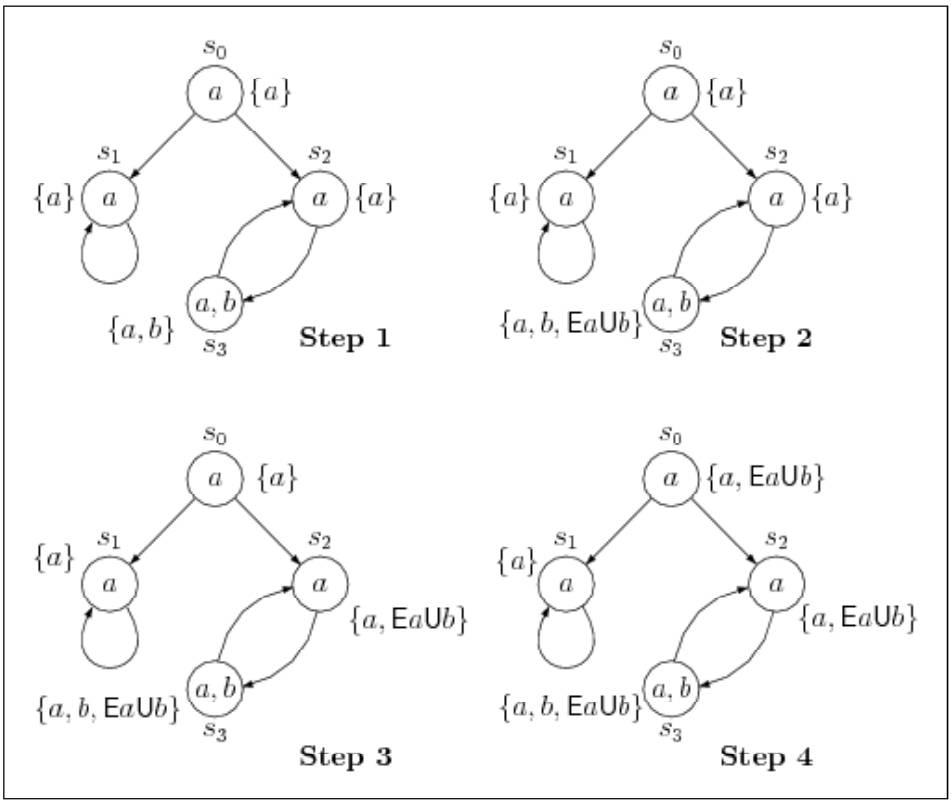

Figure 4. The CTL model-checking algorithm for EaUb

As we had mentioned earlier, F and G are special cases of the modality U. Formally, Ff is the same as ttUf and Gf is the same as $\neg$F$\neg f$. This means that we can translate all formulas of the form EFf, AFf, EGf and AGf in terms of EU and AU and apply the labelling procedure described above.

Figure 4 illustrates the labelling procedure on a simple model with four states. In this example, we would like to check whether the assertion EaUb holds at the state s_0, where a and b are basic assertions. Here is the sequence in which labels are generated by our model-checking algorithm.

1. Label each state with the basic assertions corresponding to the logical propositions that hold at that state.

One extremely useful feature of model-checking is that when the algorithm fails, it yields some information about why the formula failed to hold.

Suggested Reading

[1] E M Clarke and E A Emerson, Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic, in D Kozen (ed.) *Workshop on Logic of Programs*, Springer Lecture Notes in Computer Science, Vol.131, pp.52–71, 1981.

[2] J P Queille and J Sifakis, Specification and Verification of Concurrent Systems in CESAR, in M Dezani-Ciancaglini and U Montanari (eds.), *International Symposium on Programming*, Springer Lecture Notes in Computer Science, Vol.137, pp.337–351, 1982.

[3] G L Peterson, Myths about the Mutual Exclusion Problem, *Information Processing Letters*, Vol.12(3), pp.115–116, 1981.

[4] O Grumberg and H Veith (eds.), *25 Years of Model Checking – History, Achievements, Perspectives*, Springer Lecture Notes in Computer Science, Vol. 5000, 2008.

[5] E Allen Emerson, The Beginning of Model Checking: A Personal Perspective, in [4], pp.27–45.

[6] E M Clarke, O Grumberg and D A Peled, *Model Checking*, MIT Press 1999.

2. Label state s_3 with $\mathsf{E}a\mathsf{U}b$ since it is already labelled b.

3. Label state s_2 with $\mathsf{E}a\mathsf{U}b$ since it is already labelled a and it has a successor labelled $\mathsf{E}a\mathsf{U}b$.

4. Label state s_0 with $\mathsf{E}a\mathsf{U}b$ since it is already labelled a and it has a successor labelled $\mathsf{E}a\mathsf{U}b$.

At this point s_0 has been labelled by the formula that we are attempting to model-check, so the algorithm reports success.

It is easy to verify that the algorithm fails for the formula $\mathsf{A}a\mathsf{U}b$. We are able to label s_3 and s_2 with $\mathsf{A}a\mathsf{U}b$, but not s_1. When we reach s_0 we find that it is labelled a but not all its successors are labelled $\mathsf{A}a\mathsf{U}b$, so we cannot add the label $\mathsf{A}a\mathsf{U}b$ to s_0.

One extremely useful feature of model-checking is that when the algorithm fails, it yields some information about why the formula failed to hold. For instance, in the previous example, we can identify that $\mathsf{A}a\mathsf{U}b$ fails to hold at s_0 because it also fails to hold at one of the successors of s_0, namely s_1. This in turn is due to the fact that no state labelled b is reachable from s_1. This kind of diagnostic information makes model-checking a valuable tool for debugging concurrent programs.

The model-checking algorithm for CTL works in time proportional to $|M| \cdot |f|$, where $|M|$ denotes the size of the abstract model (as a graph) and $|f|$ is the length of the formula f. This is because we have to label the model with all subformulas in f, so we have $|f|$ rounds of labelling. In each round, we may have to traverse the entire graph to propagate the subformula label to all states, so each round of labelling takes time proportional to $|M|$.

6. Beyond CTL Model-checking

After the pioneering work of Emerson, Clarke and Sifakis, model-checking algorithms have been devised for other variants of temporal logic, including linear-time temporal logic. Model-checking algorithms systematically explore all executions of the model that are relevant for interpreting formulas in the underlying temporal logic. However, the algorithms for other temporal logics are typically more complicated than the model-checking algorithm for CTL, which is unexpectedly simple, both conceptually and from the point of view of computational complexity.

The main bottleneck to the practical applicability of model-checking is the fact that the size of the abstract model is typically very large.

The main bottleneck to the practical applicability of model-checking is the fact that the size of the abstract model is typically very large. This is particularly true for concurrent systems with a large number of independent components. The global state space of such a concurrent system is the product of the local state spaces and hence grows exponentially in the number of components. This is often referred to as the *state explosion problem* in model-checking.

A lot of research has gone into techniques for overcoming the state explosion problem. One important development has been the invention of symbolic model-checking, in which the model is described implicitly using logical formulas, rather than explicitly representing all reachable states and transitions as a graph. Another fruitful area of current research is to extend model-checking to systems with an infinite number of states.

A recent collection of articles surveying the state of the art in model-checking can be found in [4]. In particular, this collection includes an informative article by Emerson on the history of model-checking [5]. An in-depth treatment of model-checking can be found in the textbook by Clarke, Grumberg and Peled [6].

Address for Correspondence
Madhavan Mukund
Chennai Mathematical Institute
H1 SIPCOT IT Park
Padur PO
Siruseri 603 103, India.
Email:madhavan@cmi.ac.in
http://www.cmi.ac.in/~madhavan