Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Pragmatics of Model Checking: An STTT Special Section *

Rance Cleaveland

Department of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794-4400, USA

Abstract. Since its discovery in the early 1980s, model checking has emerged as one of the most exciting areas in the field of formal verification of system correctness. The chief reason for this enthusiasm resides in the fact that model checkers establish *in a fully automated manner* whether or not a system satisfies formal requirements; users need not construct proofs. Until the early 1990s, however, the practical impact of model checking was modest, in large part because the *state-explosion* problem limited the applicability of the technology. Recent advances in the field have dramatically expanded the scope of model checking, however, and interest in it on the part of practitioners is growing. This special section surveys several recent approaches to attacking the state explosion that are supporting the continued advancement of model checking as a viable technology for improved system design.

Key words: model checking – temporal logic – state explosion – system verification

1 Introduction

Model checking [20,22,57] has emerged as one of the most promising developments of the past two decades in the area of formal methods for system correctness. As the term suggests, in model checking one checks whether a mathematically precise description of a system is a model of, or *satisfies*, a mathematically precise description of its requirements. The crucial insight underpinning the interest in model checking is that when the system descriptions are *finite-state*, and the requirements are expressed in temporal logic [33,52], then the satisfaction check can be performed algorithmically, without requiring interaction with the user. This contrasts with more traditional approaches to system verification, which rely on user-generated proofs that systems meet their requirements. Indeed, the term "model checking" has become virtually synonymous with the (more precise) phrase "automatic temporal-logic model checking," although other automated analyses, including checks of refinement relations between systems [10,12,18,27,41,45,55,60], may also be seen as instances of the general notion. In the remainder of this note we use "model checking" to mean "automatic temporal-logic model checking."

Despite the obvious appeal of model checking it has only been within the past five years that it has begun attracting serious attention from engineers, such as hardware and communications-protocol designers, who build finite-state systems. In addition to pedagogical and cultural reasons, a key technical difficulty has impeded the uptake of this technology: the *state-explosion problem*. State explosion results from the fact that, in general, the size of a state space of a system grows exponentially in the size of the system description. This phenomenon has several causes, two of which include the following.

- If the system contains data elements (variables, latches, etc.) then the number of states is usually exponential in the number of such elements.
- If a system consists of several parallel components, then the number of system states is usually proportional to the product of the sizes of the state spaces of the individual processes.

Consequently, while the best traditional model-checking algorithms [22,29,57,63] are linear in the number of states of a system, their applicability is severely restricted by the prohibitive number of states systems can have.

Over the past decade, researchers have begun to develop effective techniques for ameliorating the consequences of state explosion. Some of these approaches have been implemented in different verification tools and exercised on various case studies, with promising results [26]. Industrial interest, especially in the hardware community but also in the telecommunications and embedded systems areas, has grown signif-

* Research supported in part by NSF grants CCR-9505562 and CCR-9705998, and AFOSR grants F49620-95-1-0508 and F49620-96-1-0087.

icantly [42,58] with companies such as Cadence, Chrysalis, CS Verilog and Telelogic marketing model checkers as part of their system design automation packages and other companies such as Intel, Lucent and HP supporting internal model-checking groups.

Despite this developing industrial interest, however, model checking represents only a small part of over-all industrial and governmental investment in system development, and new methods for coping with state explosion are needed for the technology to continue its advance. This special section surveys some of the recent promising approaches to this problem. Each of the topics presented has a sound mathematical basis, has been implemented in a tool, and has been used to undertake at least one significant case study.

The remainder of this note sets the stage for the papers to come by providing some general background in temporal logic and model checking. The next section reviews basic definitions needed to understand temporal logics and their use in specifying system requirements. The section following then describes the classic model-checking algorithm of [22] and points out how state explosion affects its behavior. The paper closes with brief descriptions of the results presented in the rest of the special section.

2 Temporal Logic and State Machines

The temporal-logic model-checking problem may be phrased abstractly as follows.

Given: A system description and a formula in temporal logic.
Determine: Whether or not the system satisfies the formula.

Model-checking algorithms take two inputs— a system and a formula— and return a yes/no answer, depending on whether or not the system satisfies ("is a model of") the formula. (Some model checkers also return diagnostic information in the event the answer is "no".) Generally speaking, model checking is only guaranteed to be decidable if the system in question contains at most a finite number of reachable states, although for some classes of infinite-state systems the problem may also be solved automatically [3–5,17,40].

Model-checking tools typically provide users with a programming-like notation for describing systems. The tool then "compiles" system descriptions into (finite-)state machines that are given, together with the temporal logic formula, to the model-checking algorithm. The intention behind the state machines is that they encode all the possible states and execution steps a system may exercise as it runs; the algorithms use this information together with the semantics of the temporal logic to calculate their answers.

2.1 State Machines

In the model-checking literature one may find two major classes of state machines: *Kripke structures*, and *labeled transition systems*. We present the mathematical definitions of these concepts and discuss briefly the kinds of systems they are generally used to model.

Kripke Structures A *Kripke structure* contains five kinds of information; hence, mathematically, a Kripke structure may be represented as a quintuple $\langle \mathcal{S}, \mathcal{P}, \ell, \longrightarrow, s_I \rangle$. These five components have the following interpretation.

- $\mathcal{S}$ is a set of states.
- $\mathcal{P}$ is a set of *atomic propositions* that may be true in some states and false in others.
- $\ell : \mathcal{S} \longrightarrow 2^{\mathcal{P}}$, the *labeling function*, maps states to sets of atomic propositions satisfied by the states. So if $p \in \ell(s)$ then proposition p holds in state s, and doesn't hold otherwise.
- $\longrightarrow \subseteq \mathcal{S} \times \mathcal{S}$, the *transition relation*, indicates which execution steps are possible. Intuitively, if $\langle s, s' \rangle \in \longrightarrow$ then when the system is in state s it may perform an execution step and evolve to state s'. Following the usual notational convention, we write $s \longrightarrow s'$ in lieu of $\langle s, s' \rangle \in \longrightarrow$.
- $s_I \in \mathcal{S}$ is the designated start state

One may find variants of this definition in the literature: sometimes $\mathcal{P}$ is left implicit, and in some situations a non-empty set of start states, rather than a single start state, is specified. It is also common to require that the transition relation $\longrightarrow$ be *total*, i.e. that every state has at least one outgoing transition. Some Kripke-structure-based formalisms are equipped with *fairness* constraints that disallow certain pathological execution sequences [52].

Kripke structures generally model systems that are *closed* in the sense that they execute without interacting with their environments. The atomic propositions typically refer to artifacts, such as values of variables, coming from the (user-level) system-modeling notation. For example, in the UNITY language [19] one represents systems as collections of if-then statements whose bodies assign values to variables. If `x` is such a variable, an example atomic proposition might be `x > 0`, which is true in some states and false in others.

Labeled Transition Systems Labeled transition systems differ from Kripke structures by allowing transitions as well as states to bear labels. The set of transition labels, or *actions*, must therefore be part of the description of a labeled transition system. Mathematically, a labeled transition system is a sextuple $\langle \mathcal{S}, \mathcal{P}, \mathcal{A}, \ell, \longrightarrow, s_I \rangle$.

- The interpretation attached to $\mathcal{S}, \mathcal{P}, \ell$, and s_I are the same as for Kripke structures.
- $\mathcal{A}$ is a set of *actions*.
- The transition relation $\longrightarrow \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ *labels* transitions with actions. Intuitively, if $\langle s, a, s' \rangle \in \longrightarrow$, and if the system is in state s, then it may perform an execution step and evolve to state s' *provided that* the environment enables action a. Traditionally, one writes $s \xrightarrow{a} s'$ in lieu of $\langle s, a, s' \rangle \in \longrightarrow$.

As with Kripke structures, one may find minor variations of this definition in the literature. In process algebras [7,11, 46,54], for instance, the set $\mathcal{P}$ is typically taken to be empty, or to contain only the propositions tt (for "true") and ff (for "false") which hold of all and no states, respectively. In both cases $\mathcal{P}$ and ℓ are omitted, and labeled transition systems become quadruples. Also, the start state s_I is sometime left out, in which case the labeled transition system actually defines a collection of systems differentiated on the basis of which state is the start state.

In contrast with Kripke structures, labeled transition systems are usually used to model *open* systems that interact with their environments. The actions encode these interactions and are often interpreted as *communications*, i.e. as sends and receives. So if $s \xrightarrow{a} s'$ and a is viewed as reception of an input, then the transition can "fire" only if the environment enables the input by providing the corresponding output. Mathematically, Kripke structures may be seen as labeled transition systems whose action set $\mathcal{A}$ contains exactly one element.

2.2 *Temporal Logics*

Temporal logics provide constructs for describing the properties systems have as they evolve over time. Different logics are generally used for Kripke structures and labeled transition systems, with notations like Computation Tree Logic (CTL) [22] and Linear-Time Temporal Logic (LTL) [52,56] being favored for the former and the modal mu-calculus (also known as the propositional mu-calculus) [49,61] being used for the latter.

2.2.1 Logics for Kripke Structures

Temporal logics for Kripke structures traditionally provide operators for describing properties of individual execution sequences of systems together with mechanisms for quantifying over such sequences. The number of such logics is vast, and a summary of them all lies beyond the scope of this note; the interested reader may consult the survey paper of Emerson [33] for an overview. Here we focus on the temporal logic CTL* [35] and various of its sublogics; these have received the bulk of attention from tool builders in the Kripke-structure community.

CTL* formulas fall into two categories. *Path* formulas permit properties to be defined for execution sequences (i.e. sequences of states) in a given Kripke structure, while *state* formulas allow properties to be defined for states. State formulas are the ones used to define properties of systems, the idea being that if a system's start state satisfies a state formula, then the system is deemed to have the given property.

Path formulas have the following form.

- Any state formula S is also a path formula; the intention is that a path satisfies S if the initial state on the path does.
- If P is a path formula, then so is $\neg P$, the "negation" of P. A path satisfies $\neg P$ if it does not satisfy P.
- If P and Q are path formulas, then so is $P \wedge Q$, the "conjunction" of P and Q. A path satisfies $P \wedge Q$ if it satisfies both P and Q.
- If P is a path formula, then $\mathsf{X}\,P$ is a path formula. X is often referred to as the "next-state" operator; a path satisfies $\mathsf{X}\,P$ if the suffix, or "tail", starting with the state after the initial one satisfies P.
- If P and Q are path formulas, then so is $P \,\mathsf{U}\, Q$. A path satisfies $P \,\mathsf{U}\, Q$ if it keeps P true "until" it makes Q true. More precisely, the path satisfies $P \,\mathsf{U}\, Q$ if some suffix of the path satisfies Q and every suffix up to this one satisfies P.

In addition to these operators, formulas typically use the following derived constructs.

- Additional propositional connectives such as $\vee$ ("or") and $\rightarrow$ ("implies") can be encoded in terms of $\neg$ and $\wedge$. Specifically, $P \vee Q$ may be rendered as $\neg((\neg P) \wedge (\neg Q))$ ("it is not the case that both P and Q are false") and $P \rightarrow Q$ as $(\neg P) \vee Q$.
- $\mathsf{F}\,P$ holds of paths in which path formula P eventually becomes true. It may be encoded as $\mathsf{tt}\,\mathsf{U}\,P$ ("true holds until P does"). Here tt, a formula that holds of any path, is defined as $S \vee (\neg S)$ for an arbitrary state formula S.
- $\mathsf{G}\,P$ holds of paths in which P remains true forever. This may be defined as $\neg\mathsf{F}(\neg P)$ ("it is not true that P is eventually false").

The operators X, U, F and G are often called *path modalities*; their meanings are illustrated in Fig. 1. In the literature one sometimes finds $\mathsf{X}\,P$, $\mathsf{F}\,P$ and $\mathsf{G}\,P$ written as $\bigcirc P$, $\Diamond P$ and $\Box P$, respectively.

State formulas are intended to be interpreted with respect to states in a Kripke structure. They may be defined as follows, where $\mathcal{P}$, a parameter to the definition, contains the set of atomic propositions.

- Any atomic proposition $p \in \mathcal{P}$ is a state formula. A state satisfies such a p if p belongs to the set that the labeling in the Kripke structure assigns to the state.
- If S is a state formula then so is its negation, $\neg S$. A state satisfies $\neg S$ if it does not satisfy S.
- If S and T are state formulas then so is their conjunction, $S \wedge T$. A state satisfies $S \wedge T$ if it satisfies both S and T.
- If P is a path formula then $\mathsf{A}\,P$ is a state formula. Intuitively, a state satisfies $\mathsf{A}\,P$ if every execution path beginning at the state satisfies path formula P.

The following derived constructs are also frequently used.

- Propositional operators such as $\vee$ and $\rightarrow$ may be encoded in the same way that their path-formula counterparts are.
- $\mathsf{E}\,P$ holds of states having some execution path emanating from them that satisfies path formula P. $\mathsf{E}\,P$ may be encoded as $\neg\mathsf{A}(\neg P)$ ("it is not the case that every path violates P").

The operators A and E are often called *path quantifiers*. Fig. 2 illustrates the meaning of the A quantifier.

As stated previously, CTL* state formulas are the ones used to define properties of Kripke structures; a Kripke structure satisfies such a formula if its start state does. Sample properties, and their natural language equivalents, include the following; propositions in *italics* denote atomic propositions.

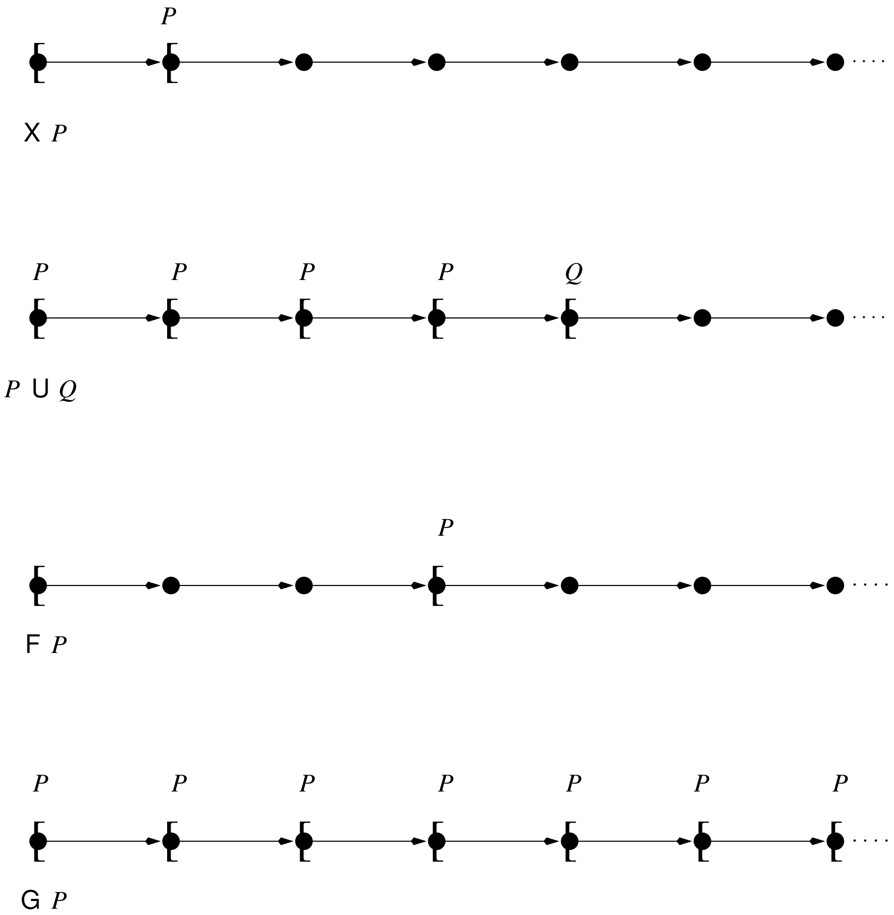

Fig. 1. An illustration of the path modalities.

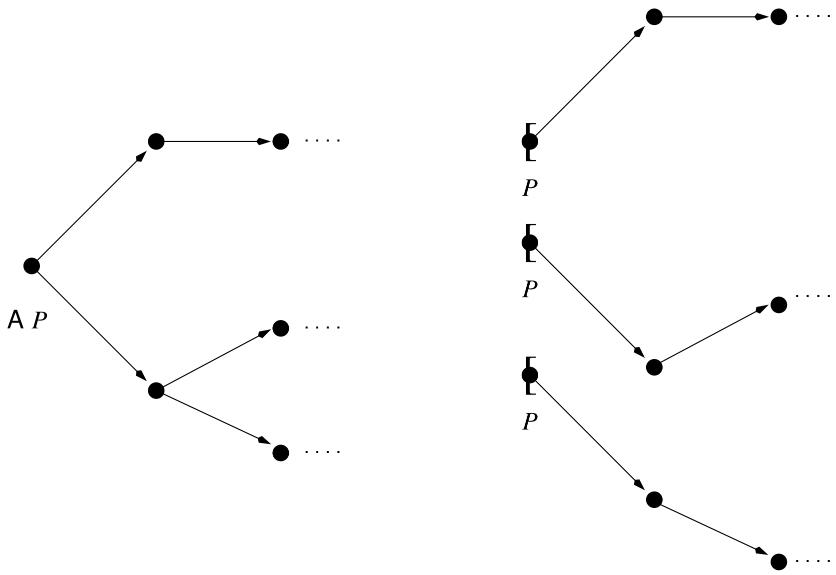

Fig. 2. An illustration of the A path quantifier.

– $\mathsf{AG}(\neg inCS1 \vee \neg inCS2)$
  "It is always the case that *inCS1* (which would hold when e.g. process 1 is in its critical section) or *inCS2* is false.
– $\mathsf{AG}(sent \rightarrow \mathsf{X}((\neg sent)\,\mathsf{U}\,received))$
  "It is always the case that if a message has just been sent, then another cannot be sent until a message has been received."
– $\mathsf{AGEF}\ reset$
  "It is always the case that the reset state is reachable."

The intuitive accounts of the operators given above can be formalized mathematically by giving a relation indicating when sequences of states satisfy path formulas and states satisfy state formulas. In order to do this we need to make precise the notion of "execution sequence". Some auxiliary notation on infinite sequences proves useful.

Definition 1. Let $\mathcal{K} = \langle \mathcal{S}, \mathcal{P}, \ell, \longrightarrow, s_I \rangle$ be a Kripke structure such that $\longrightarrow$ is total, let $\mathcal{S}^\omega$ be the set of infinite sequences of states, and let $\sigma = s_0 s_1 s_2 \ldots$ be an element of $\mathcal{S}^\omega$.

1. $\sigma[i] \in \mathcal{S}$ is defined to be s_i.
2. $\sigma[i\rangle \in \mathcal{S}^\omega$ is the infinite sequence $s_i s_{i+1} \ldots$.
3. σ is an *execution sequence emanating from* s if $s_0 = s$ and for all $i \geq 0$, $s_i \longrightarrow s_{i+1}$. That is, s must be the initial state in the sequence, and a state is reachable from its predecessor via an execution step. We use $E_{\mathcal{K}}(s) \subseteq \mathcal{S}^\omega$ to represent the set of execution sequences emanating from s in Kripke structure $\mathcal{K}$.

Given a Kripke structure $\mathcal{K} = \langle \mathcal{S}, \mathcal{P}, \ell, \longrightarrow, s_I \rangle$ we now define a relation $\models_{\mathcal{K}}$ relating paths (i.e. elements of $\mathcal{S}^\omega$) to path formulas and states to state formulas. If $\sigma \models_{\mathcal{K}} P$ then σ is deemed to satisfy P, and likewise for $s \models_{\mathcal{K}} S$. The definition is given below; it what follows p is an atomic proposition, P and Q are assumed to be path formulas, and S and T are state formulas.

$$
\begin{array}{lll}
\sigma \models_{\mathcal{K}} S & \text{if} & \sigma[0] \models_{\mathcal{K}} S \\
\sigma \models_{\mathcal{K}} \neg P & \text{if} & \sigma \not\models_{\mathcal{K}} P \\
\sigma \models_{\mathcal{K}} P \wedge Q & \text{if} & \sigma \models_{\mathcal{K}} P \text{ and } \sigma \models_{\mathcal{K}} Q \\
\sigma \models_{\mathcal{K}} \mathsf{X}\,P & \text{if} & \sigma[1\rangle \models_{\mathcal{K}} P \\
\sigma \models_{\mathcal{K}} P\,\mathsf{U}\,Q & \text{if} & \text{there is a } n \geq 0 \text{ such that } \sigma[n\rangle \models_{\mathcal{K}} Q \\
 & & \text{and for all } i \text{ with } 0 \leq i < n, \sigma[i\rangle \models_{\mathcal{K}} P \\
s \models_{\mathcal{K}} p & \text{if} & p \in \ell(s) \\
s \models_{\mathcal{K}} \neg S & \text{if} & s \not\models_{\mathcal{K}} S \\
s \models_{\mathcal{K}} S \wedge T & \text{if} & s \models_{\mathcal{K}} S \text{ and } s \models_{\mathcal{K}} T \\
s \models_{\mathcal{K}} \mathsf{A}\,P & \text{if} & \text{for all } \sigma \in E_{\mathcal{K}}(s), \sigma \models_{\mathcal{K}} P
\end{array}
$$

Finally, we write $\mathcal{K} \models S$ if $s_I \models_{\mathcal{K}} S$; a Kripke structure satisfies a state formula exactly when its initial state does.

*Sublogics of CTL** We close this section by commenting on fragments of CTL* that appear frequently in the literature. Linear-time temporal logic, or LTL [33,52], consists of path formulas containing no occurrences of the path quantifiers A and E. A system satisfies such a formula P if it satisfies the CTL* state formula $\mathsf{A}\,P$. Computation Tree Logic, or CTL [22], formulas require that every path modality ($\mathsf{X}, \mathsf{U}, \mathsf{F}$ or G) be immediately preceded by a path quantifier (A or E). For example, the formula $\mathsf{FG}\,p$ is an LTL formula, but $\mathsf{FEG}\,p$ is not, owing to the presence of the E operator. $\mathsf{AFEG}\,p$ is a CTL formula, but $\mathsf{AFG}\,p$ is not, since the G operator is not immediately preceded by either an A or an E.

As LTL formulas may only quantify over the executions leaving the start state of a system, they often prove easier to formulate and conceptualize; consequently, most of the non-model-checking literature on temporal logic deals with LTL. Until relatively recently [48], however, effective model-checking tools have not existed for LTL, owing to worries about the inefficiency of the model-checking procedures for it [51,63]. On the other hand, efficient model-checking algorithms for CTL and similar logics were developed in the early 1980s [21,57], and consequently the earliest model-checkers supported these "pure branching-time" logics. Most current model checkers for Kripke structures still provide CTL as their temporal logic for this reason. For a discussion on the relative merits of branching-time and linear-time temporal logics, the interested reader is referred to [35,37,50]

2.2.2 Logics for Labeled Transition Systems

The *modal mu-calculus* [49,61], the logic most frequently used in the setting of labeled transition systems, differs substantially in form from the temporal logics discussed above. Specifically, it replaces the two-level syntax of CTL* with a single syntax of state formulas, and instead of a fixed collection of path modalities it provides a family of action-labeled next-state operators together with a general facility for defining properties *recursively*.

The definition of the mu-calculus is parameterized with respect to three sets: a set $\mathcal{P}$ of atomic propositions, a set $\mathcal{A}$ of actions, and a set $\mathcal{X}$ of propositional variables. These sets are assumed to be disjoint. Formulas are intended to describe properties of states in a given labeled transition system, and may have one of the following forms.

– Any atomic proposition $p \in \mathcal{P}$ is a mu-calculus formula. A state satisfies such a formula if the state labeling in the labeled transition system indicates this is the case.
– Any variable $x \in \mathcal{X}$ is a mu-calculus formula. To interpret formulas of this form, one needs an environment indicating what variables mean. Assuming such an environment is given, a state satisfies a variable if the environment indicates this is the case.
– If S is a mu-calculus formula, then so is its negation, $\neg S$. A state satisfies $\neg S$ if it does not satisfy S.
– If S and T are mu-calculus formulas, then their conjunction, $S \wedge T$, is a formula. A state satisfies $S \wedge T$ if it satisfies both S and T.
– If $a \in \mathcal{A}$ is an action and S is a mu-calculus formula, then $[a]S$ is a mu-calculus formula. A state satisfies $[a]S$ if all the a-transitions emanating from the state lead to states in which S holds. Fig. 3 illustrates the meaning of $[a]S$.
– If $x \in \mathcal{X}$ is a variable and S is a mu-calculus formula, then $\mu x.S$ is a mu-calculus formula. This formula is in-

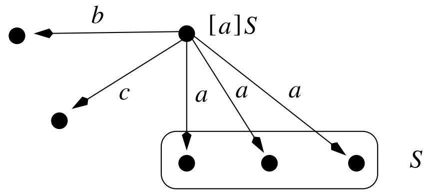

Fig. 3. The meaning of mu-calculus formula $[a]S$.

tended to represent a "recursively defined" formula that "solves" the equation $x = S$.

In addition to adhering to the forms outlined above well-formed mu-calculus formulas must satisfy two additional constraints. Firstly, a formula must be *closed*: every occurrence of a variable x must fall within the scope of a μx construct. Secondly, in formulas of the form $\mu x.S$ any (free) occurrence x within S must be *positive* in the sense that it falls within the scope of an even number of negations. The first requirement ensures that variables are only used in support of recursive definitions. The second restriction is necessary to ensure that equations such as $x = S$ indeed have solutions.

The following derived constructs also prove useful.

- The propositional operators tt, ff, $\vee$ and $\rightarrow$ are encoded in the usual fashion.
- $\langle a \rangle S$ holds of a state if it has an outgoing transition labeled by a and leading to a state satisfying S. It may be encoded as $\neg[a](\neg S)$ ("it is not the case that every a-transition leads to a state violating S").
- $\nu x.S$ represents another "solution" to the equation $x = S$. It may be encoded as $\neg\mu x.(\neg S[\neg x/x])$, where $S[\neg x/x]$ represents the formula obtained by replacing all (free) occurrences of x in S by $\neg x$. We comment on $\nu x.S$ and its relationship with $\mu x.S$ below.

The $[a]$ and $\langle a \rangle$ constructs are *single-step* modalities in that they only permit the expression of properties that look one execution step into the future. Nevertheless, the mu-calculus turns out to be more expressive than other known temporal logics. This expressiveness results from the inclusion of the μ construct, which is often referred to as a *least fixpoint* operator. Semantically, $\mu x.S$ represents the "least solution" to the equation $x = S$, where "least" means "satisfied by the fewest possible states." Intuitively, at least for finite-state systems, $\mu x.S$ may be thought of as an infinitary disjunction,

$$S_0 \vee S_1 \vee S_2 \vee \cdots,$$

where S_0 represents the formula ff that holds of no states, and S_{i+1} is $S[S_i/x]$ (i.e. S with all free occurrences of x replaced by S_i). As an example, consider the formula $\mu x.[a]x$. According to the interpretation just given, this is equivalent to

$$\mathsf{ff} \vee [a]\mathsf{ff} \vee [a][a]\mathsf{ff} \vee \cdot.$$

A state satisfying this formula must satisfy either ff (which no state can satisfy); or $[a]$ff, which holds if the state has no a-transitions (this is the only way that "every a-transition" can lead to a state satisfying ff); or $[a][a]$ff (i.e. every a-transition leads to a state having no a-transitions); etc. So for this formula to hold of a state in a finite-state system, the state must be incapable of an infinite sequence of a-transitions. The set of states satisfying this property may be seen as a "solution" to $x = [a]x$.

By way of contrast, $\nu x.S$ represents the greatest solution to $x = S$ and is consequently referred to as the *greatest fixpoint* operator. Using the definition ν together with DeMorgan's laws, one sees that in the case of finite-state systems, $\nu x.S$ can be viewed as an infinite conjunction,

$$\hat{S}_0 \wedge \hat{S}_1 \wedge \hat{S}_2 \wedge \cdots,$$

where $\hat{S}_0$ is tt and $\hat{S}_{i+1} = S[\hat{S}_i/x]$. As an example, consider $\nu x.[a]x$. A state in a finite-state system that satisfies this formula must satisfy:

$$\mathsf{tt} \wedge [a]\mathsf{tt} \wedge [a][a]\mathsf{tt} \wedge \cdots.$$

So a state satisfying this formula satisfy tt (which every state does); and $[a]$tt (i.e. every a-transition must lead to a state satisfying tt, but every state has this property); etc. It turns out that $\nu x.[a]x$ is equivalent to tt in that every state can satisfy it. Like $\mu x.[a]x$, this may be seen as a reasonable "solution" to $x = [a]x$.

The following examples illustrate the kinds of properties one can express in the mu-calculus.

- $\mu x.[\mathit{login}]x$
  "Only finitely many *login* attempts are possible."
- $\nu x.([\mathit{receive}]\mathsf{ff} \wedge [\mathit{idle}]x)$
  "A *receive* becomes enabled only after a *send* action occurs." (Here we assume that the set of actions includes only *send*, *receive* and *idle*.) Recall that a state satisfies $[a]$ff exactly when it has no outgoing a-transitions.

The formal semantics of the mu-calculus is given in terms of a function, $[\![-]\!]_{\mathcal{L}}$, mapping formulas to sets of state in labeled transition system $\mathcal{L}$. Intuitively, $[\![S]\!]_{\mathcal{L}}$ returns the set of states in $\mathcal{L}$ satisfying S. In order to define $[\![-]\!]_{\mathcal{L}}$ inductively on the structure of formulas, we need to cater for free occurrences of variables that arise when giving the semantics of $\mu x.S$ in terms of S. This may be done by augmenting $[\![-]\!]_{\mathcal{L}}$ so that it also takes a second argument, e, which is an environment assigning meanings to free variables. Thus $[\![S]\!]_{\mathcal{L}} e$ returns a set of states satisfying S, with e supplying the interpretation of any free variables in S. The following notions make this precise.

Definition 2. Let $\mathcal{L} = \langle \mathcal{S}, \mathcal{P}, \mathcal{A}, \ell, \longrightarrow, s_I \rangle$ be a labeled transition system, and let $\mathcal{X}$ be a set of propositional variables.

1. Let $s \in \mathcal{S}$ and $a \in \mathcal{A}$ be a state and action, respectively. Then $\{s \xrightarrow{a} \bullet\} \subseteq \mathcal{S}$ is the set of states reachable from s via an a-transition:
$$\{s \xrightarrow{a} \bullet\} = \{\, s' \in \mathcal{S} \mid s \xrightarrow{a} s' \,\}.$$
2. An *environment* over $\mathcal{X}$ and $\mathcal{L}$ is a function $e \in \mathcal{X} \longrightarrow 2^{\mathcal{S}}$ mapping variables to sets of states.

3. Let e be an environment over $\mathcal{X}$ and $\mathcal{L}$, and let $x \in \mathcal{X}$ and $\mathcal{S}' \subseteq \mathcal{S}$ be a variable and set of states, respectively. Then $e[x \mapsto \mathcal{S}']$ is a new environment defined as follows.

$$(e[x \mapsto \mathcal{S}'])(y) = \begin{cases} \mathcal{S}' & \text{if } x = y \\ e(y) & \text{otherwise} \end{cases}$$

In other words, $e[x \mapsto \mathcal{S}']$ behaves exactly like e except on argument x, in which case it returns $\mathcal{S}'$.

Let $\mathcal{L} = \langle \mathcal{S}, \mathcal{P}, \mathcal{A}, \ell, \longrightarrow, s_I \rangle$ and e be an environment over $\mathcal{X}$ and $\mathcal{L}$. We may now define $[\![-]\!]_{\mathcal{L}}$ as follows.

$$\begin{aligned}
[\![p]\!]_{\mathcal{L}} e &= \ell(p) \\
[\![x]\!]_{\mathcal{L}} e &= e(x) \\
[\![\neg S]\!]_{\mathcal{L}} e &= \mathcal{S} - ([\![S]\!]_{\mathcal{L}} e) \\
[\![S \wedge T]\!]_{\mathcal{L}} e &= ([\![S]\!]_{\mathcal{L}} e) \cap ([\![T]\!]_{\mathcal{L}} e) \\
[\![[a]S]\!]_{\mathcal{L}} e &= \{\, s \in \mathcal{S} \mid \forall s' \in \{s \xrightarrow{a} \bullet\}. s' \in [\![S]\!]_{\mathcal{L}} e \,\} \\
[\![\mu x.S]\!]_{\mathcal{L}} e &= \bigcap \{\, \mathcal{S}' \subseteq \mathcal{S} \mid [\![S]\!]_{\mathcal{L}} (e[x \mapsto \mathcal{S}']) \subseteq \mathcal{S}' \,\}
\end{aligned}$$

The definition of $[\![\mu x.S]\!]_{\mathcal{L}} e$ relies on results from lattice theory, and we comment more on it here. Given formula $\mu x.S$ and environment e, we may define a function $f_e : 2^{\mathcal{S}} \longrightarrow 2^{\mathcal{S}}$ as follows.

$$f_e(\mathcal{S}') = [\![S]\!]_{\mathcal{L}} e[x \mapsto \mathcal{S}']$$

Because of the syntactic restrictions on x in S (occurrences of x must appear within the scope of an even number of negations), f_e is guaranteed to be *monotonic*: if $\mathcal{S}' \subseteq \mathcal{S}''$ then $f_e(\mathcal{S}') \subseteq f_e(\mathcal{S}'')$. In addition, $2^{\mathcal{S}}$ forms a complete lattice with respect to the subset ordering: any set $\mathcal{T} \subseteq 2^{\mathcal{S}}$ of subsets of $\mathcal{S}$ has a greatest and least upper bound given by $\bigcup \mathcal{T}$ and $\bigcap \mathcal{T}$, respectively. Consequently, the Tarski-Knaster Theorem from lattice theory [62] ensures that f_e has a least fixpoint $\mu f_e \subseteq \mathcal{S}$ and greatest fixpoint $\nu f_e \subseteq \mathcal{S}$ given as follows.

$$\mu f_e = \bigcap \{\, \mathcal{S}' \subseteq \mathcal{S} \mid f_e(\mathcal{S}') \subseteq \mathcal{S}' \,\}$$
$$\nu f_e = \bigcup \{\, \mathcal{S}' \subseteq \mathcal{S} \mid \mathcal{S}' \subseteq f_e(\mathcal{S}') \,\}$$

That is, μf_e is the smallest set of states that is a "solution" to the "equation" $x = f_e(x)$ in the sense that $f_e(\mu f_e) = \mu f_e$; similarly, νf_e is the largest such solution. From the definition of $[\![\mu x.S]\!]_{\mathcal{L}} e$, one may see that

$$[\![\mu x.S]\!]_{\mathcal{L}} e = \mu f_e;$$

consequently, $\mu x.S$ may be seen as the "smallest," or "least permissive," property satisfying the equation $x = S$. A similar line of reasoning establishes the dual result for $\nu x.S$: it is the "most permissive" property satisfying $x = S$.

When a formula S contains no free variables, it follows that the meaning of S cannot depend on an environment; formally, $[\![S]\!]_{\mathcal{L}} e = [\![S]\!]_{\mathcal{L}} e'$ for any environments e and e'. In this case we write $[\![S]\!]_{\mathcal{L}}$ to represent this set of states. We also write $\mathcal{L} \models S$ if $s_I \in [\![S]\!]_{\mathcal{L}}$; so a labeled transition system satisfies formula S exactly when its start state does.

Researchers have identified useful variants and fragments of the mu-calculus that we briefly touch on here. Some accounts label the $[-]$ and $\langle - \rangle$ modalities with *sets* of actions rather than single actions [61]. Assuming that $A \subseteq \mathcal{A}$ is such a set, a state satisfies $[A]S$ if every transition labeled by an action in A leads to a state satisfying S, and dually for $\langle A \rangle S$. Other accounts use systems of equations to define formulas rather than the linear syntax presented here [6]. Specifically, a system such as:

$$\mathsf{min} \left\{ \begin{array}{c} X_1 = S_1 \\ \vdots \\ X_n = S_n \end{array} \right\}$$

defines n formulas, one for each equation, where the min indicates that "smallest" solutions are intended. This notation is no more expressive than the one given here, although it can be more concise. Finally, the full mu-calculus presents efficiency problems for model checking [13], and researchers have thus focused on the development of algorithms for fragments of the logic. The *alternation-free* fragment forbids formulas of the following form:

$$\mu x.(\cdots \nu y.S \cdots)$$

where both x and y appear free in S. In essence, a formula of this form introduces a "mutual dependency" between x and y in that assigning a meaning to S requires values for both x and y. If the type of fixpoint for x and y is the same this does not present problems; however, if (as in the example) one fixpoint is greatest while the other is least, then the time-complexity of model checking can be affected. The alternation-free mu-calculus [38] forbids such situations by requiring that fixpoints of different types be strictly nested within one another. Another fragment, L_2 [8,36], copes with problems of alternation by restricting the ways in which variables may appear in the context of the $\wedge$ and $[-]$ operators.

As was remarked at the end of Section 2.1, Kripke structures may be seen as labeled transition systems whose action set $\mathcal{A}$ contains only one element. With $\mathcal{A}$ thus restricted, one may then compare the expressive power of CTL* and the mu-calculus. The later turns out to be strictly more expressive [33], and effective translation procedures exist from CTL* to the (L_2 fragment of the) mu-calculus [9,31,36]. It also turns out that the CTL fragment of CTL* can be efficiently encoded in the alternation-free mu-calculus [34].

3 Traditional Model Checking

This section explains the basic workings of traditional model-checking procedures by examining in some detail the classic algorithm for CTL presented in [22].

3.1 The Syntax of CTL

We begin by providing a more systematic account of the syntax of CTL than the one given in Section 2.2.1. As CTL formulas are special cases of CTL* formulas, the reader is referred to Section 2.2.1 for a discussion of their semantics.

The syntax of CTL is parameterized with respect to a set $\mathcal{P}$ of atomic propositions. CTL formulas then have the following form.

– If $p \in \mathcal{P}$ then p is a CTL formula.
– If S is a CTL formula then $\neg S$ and $\mathsf{AX}\, S$ are CTL formulas.
– If S and T are CTL formulas then $S \wedge T$, $\mathsf{A}(S \,\mathsf{U}\, T)$ and $\mathsf{E}(S \,\mathsf{U}\, T)$ are CTL formulas.

It should be noted that CTL formulas are all *state* formulas in the parlance of CTL*. Also, both the $\mathsf{E}(\_\,\mathsf{U}\,\_)$ and the $\mathsf{A}(\_\,\mathsf{U}\,\_)$ operators are needed; one cannot be encoded in terms of the other.

3.2 CTL Model Checking

A CTL model-checking algorithm takes two inputs: a Kripke structure $\mathcal{K} = \langle \mathcal{S}, \mathcal{P}, \ell, \longrightarrow, s_I \rangle$, and a CTL formula S. It returns the boolean *true* if $\mathcal{K} \models S$ and false otherwise. In what follows, we assume that $\mathcal{K}$ is fixed.

The model-checking procedure of [22] follows a "global strategy" in that it calculates *all* states satisfying formula S. Then $\mathcal{K} \models S$ exactly when s_I is in this set of states. For notational convenience we write $[\![S]\!]_{\mathcal{K}} \subseteq \mathcal{S}$ to represent the set of states satisfying formula S; so $s \in [\![S]\!]_{\mathcal{K}}$ if and only if $s \models_{\mathcal{K}} S$. We may therefore characterize the model checker [22] as consisting of two steps.

– Calculate $[\![S]\!]_{\mathcal{K}}$.
– Determine whether or not $s_I \in [\![S]\!]_{\mathcal{K}}$.

Clearly, the difficult step involves the calculation of $[\![S]\!]_{\mathcal{K}}$. The procedure is syntax-directed; it uses the structure of formula S to guide the computation. The specific strategy may be summarized as follows.[1]

– For each immediate subformula T of S, recursively calculate $[\![T]\!]_{\mathcal{K}}$.
– Use the topmost operator of S to compute $[\![S]\!]_{\mathcal{K}}$ from the $[\![T]\!]_{\mathcal{K}}$.

Most of the CTL operators present no difficulty and are summarized below.

$$\begin{aligned} S = p &: [\![S]\!]_{\mathcal{K}} = \ell(p) \\ S = \neg T &: [\![S]\!]_{\mathcal{K}} = \mathcal{S} - [\![T]\!]_{\mathcal{K}} \\ S = T \wedge U &: [\![S]\!]_{\mathcal{K}} = [\![T]\!]_{\mathcal{K}} \cap [\![U]\!]_{\mathcal{K}} \end{aligned}$$

The set difference operator $-$ and the intersection operator $\cap$ can be implemented in obvious ways in time proportional to the sizes of the sets being manipulated.

The AX operator is only slightly more difficult to handle. Suppose that $S = \mathsf{AX}\, T$ and that $[\![T]\!]_{\mathcal{K}}$ has been computed. To calculate $[\![S]\!]_{\mathcal{K}}$, we scan through all states in $s \in \mathcal{S}$; for each such s, we traverse each transition emanating from s and check whether the target state t is in $[\![T]\!]_{\mathcal{K}}$. If this holds for each such t then s may be added to $[\![S]\!]_{\mathcal{K}}$; otherwise, it is omitted. In general, computing $[\![S]\!]_{\mathcal{K}}$ in this case requires that each state in $\mathcal{K}$ be visited and that each transition be traversed once.

The complicated operators to handle are $\mathsf{A}(\_\,\mathsf{U}\,\_)$ and $\mathsf{E}(\_\,\mathsf{U}\,\_)$; as the issues are similar for the two, we only study the former in depth. So assume that $S = \mathsf{A}(T \,\mathsf{U}\, U)$ and that $[\![T]\!]_{\mathcal{K}}$ and $[\![U]\!]_{\mathcal{K}}$ have been computed. An iterative, approximation-based procedure is then used to compute $[\![S]\!]_{\mathcal{K}}$. Roughly speaking, the routine generates a series $S_0, S_1, \ldots$ of approximations to $[\![S]\!]_{\mathcal{K}}$. Each S_i contains more states than its predecessor, and each is guaranteed to contain only states that are also in $[\![S]\!]_{\mathcal{K}}$. More formally, assuming S_n is the most recently generated approximation, the invariants maintained are the following.

1. For all i with $0 \le i < n, S_i \subset S_{i+1}$.
2. For all i with $0 \le i \le n, S_i \subseteq [\![S]\!]_{\mathcal{K}}$.

The procedure stops when no more states can be added to the most recent approximation F_n without invalidating invariant 2. The S_i are calculated as follows.

1. S_0 is $\emptyset$
2. From the definition of the semantics of A and U, every state that satisfies U is guaranteed to satisfy $\mathsf{A}(T \,\mathsf{U}\, U)$. Consequently, S_1 is set to $[\![U]\!]_{\mathcal{K}}$, if it is nonempty. If it is empty, we stop and set $[\![S]\!]_{\mathcal{K}} = S_0 = \emptyset$.
3. For $i \ge 2$, we generate S_{i+1} from S_i by doing the following.
   (a) We calculate the set
   $$\begin{aligned} S_i' = \{\, s \in \mathcal{S} \mid s \in [\![T]\!]_{\mathcal{K}} \wedge s \notin S_i \\ \wedge \forall t \in \{\, t \in \mathcal{S} \mid s \longrightarrow t \,\}.t \in S_i \,\} \end{aligned}$$
   Informally, S_i' contains those states that satisfy T and are not in S_i but all of whose transitions lead to states in S_i.
   (b) If S_i' is empty we set halt, setting $[\![S]\!]_{\mathcal{K}} = S_i$.
   (c) Otherwise, we set $S_{i+1} = S_i \cup S_i'$.

Fig. 4 illustrates how this works.

The key insight in the algorithm involves the efficient calculation of set S_i' in step 3a. A naive approach would require scanning the entire Kripke structure $\mathcal{K}$. Using such an scheme would lead to a worst-case running time quadratic in the size of $\mathcal{K}$ to calculate $[\![\mathsf{A}(T \,\mathsf{U}\, U)]\!]_{\mathcal{K}}$, since the number of approximations is bounded by the number of states in $\mathcal{K}$ and the generation of each approximation would require processing every state and transition in $\mathcal{K}$. It turns out, however, that the computation of set S_i' can be amortized over the computation of the preceding approximations by remembering, for each state, how many of its transitions lead to states *not* contained the current approximation. When this number becomes 0 for a state $s \in [\![T]\!]_{\mathcal{K}}$, s has all its transitions contained in the current approximation S_i and should then be added to S_i'. As this set will then be merged into S_{i+1}, states having a transition *into* s must have their counts decremented. It may be shown that each transition is traversed once in this fashion throughout the calculation of all the approximations, and hence the

[1] It should be noted that this account of the algorithm differs in style, although not in substance, from the one given in [22].

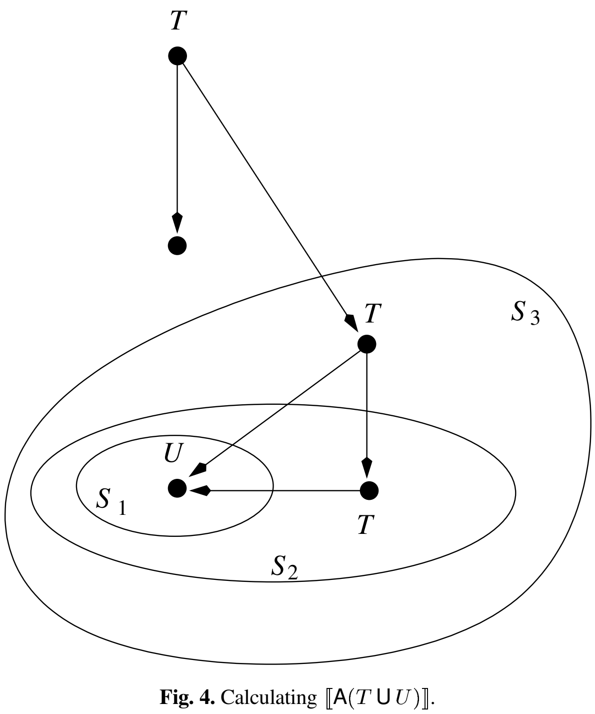

Fig. 4. Calculating $[\![\mathsf{A}(T\,\mathsf{U}\,U)]\!]$.

computation of $[\![\mathsf{A}(T\,\mathsf{U}\,U)]\!]$ can be done in time proportional to the size of $\mathcal{K}$.

Implementing this approach requires some additional storage in the form of a counter for each state (which may be reused), and the data structure for $\mathcal{K}$ must allow transitions to be traversed backwards from target to source so that counters can be decremented efficiently. At the beginning of the approximation procedure the counters must also be initialized; this requires a scan of $\mathcal{K}$.

3.3 Practical Impacts

The previous algorithm may be shown to have a running time of $O(|\mathcal{K}| \cdot |S|)$, where $|\mathcal{K}|$ is the total number of states and transitions in the system and $|S|$ is the number of subformulas of S. In practice, the latter quantity is much smaller than the former, and thus the routine is essentially linear in the size of $\mathcal{K}$. The efficiency of the algorithm led Clarke and colleagues to experiment with model checking on actual systems; the EMC system was built and used to analyze a variety of hardware designs [14,15]. At the same time, Sifakis and coworkers built the CESAR/XESAR model checker and applied it to the analysis of several communications protocols [57,59]. It may be safely said that these groups' work marked the emergence of model-checking as an interesting and useful technology for system design.

Nevertheless, the limitations of the traditional approach to model checking quickly became apparent, because even though the algorithms exhibited running times proportional to the size of the state machines, in practice these state machines can be enormous. For example, a (quite small) hardware design containing only 100 latches could have 2^{100}, or over 10^{30}, states. Thus, while it generated excitement in the formal methods research community in the 1980s, model checking did not attract much attention from practicing engineers during this time.

4 The Special Section

In the 1990s, advances in model-checking technology have improved its practical utility to the point that it is becoming commercialized. These developments have come about because of research aimed at alleviating the *state-explosion* problem referred to in the introduction.

The papers in this special section describe approaches to coping with state explosion. The methods presented may be categorized on the following basis.

- Some techniques use compact data structures or efficient search techniques to limit the storage needed to analyze a Kripke structure or labeled transition system.
- Other techniques attempt to exploit structure in system descriptions before they are converted into Kripke structures or labeled transition systems in order to reduce the sizes of the structures analyzed.

A number of other promising directions exist besides the ones addressed in this special section; these include abstraction-based approaches [25,43,44], the exploitation of system symmetries and architectures [23,24,39], and the use of semantic quotienting to eliminate redundant states [32]. Interested readers may consult [26,28] for more complete overviews of research in the area.

The first paper in the section, "Model Checking: A Hardware Design Perspective," describes how model checking may be used in traditional hardware design methodologies as a means of getting early feedback on the correctness of a design. The model-checking technology the authors describe uses *ordered binary decision diagrams* [16] (OBDDs) to encode the Kripke structures associated with designs. OBDDs afford extremely compact representations of state spaces and transition relations; the authors discuss how OBDDs can significantly expand the scope of designs that may be analyzed using CTL model checking.

The second paper, "A Minimized Automaton Representation of Reachable States," presents an alternative means of efficiently encoding state spaces that is based on the use of minimized deterministic automata. The authors show how operations on sets of states may be implemented efficiently, and they give a wide array of experimental results for their technique, which they have implemented in the SPIN verification tool [47].

The next paper, "State Space Reduction using Partial Order Techniques," describes the use of *partial order* information as a means of eliminating redundant states during LTL model checking. The name of the method derives from the fact that when a system consists of several parallel components, the transitions of individual subsystems may be *independent* in the sense that the occurrence of one does not affect the other, and vice versa. When this is the case, instead of considering the states resulting from interleaving these transitions in all possible ways, one may instead consider only one interleaving without sacrificing correctness. The paper describes the mechanics of partial-order reduction and discusses practical experience with the technique in the context of the SPIN model checker [48].

"Partial Model Checking of Modal Equations" also advocates an analysis of the "concurrency structure" of a system in order to reduce state-space size. The basic approach is to use the labeled transition systems for individual components to "factor" a mu-calculus formula into a formula that the rest of the system needs to satisfy for the whole system to be correct. Consequently, the global labeled transition system for the whole system need never be constructed. The paper describes several heuristics for reducing the sizes of the intermediate mu-calculus formulas generated and presents experimental results obtained from an implementation of the method.

The final paper in the section, "Local Model Checking and Protocol Analysis," describes a *local*, or *on-the-fly*, technique for comparing labeled transition systems to formulas in the alternation-free modal mu-calculus. The hallmark of local algorithms is that they compute partial semantics of subformulas in a demand-driven manner; this stands in contrast to traditional, global techniques. This demand-driven aspect ensures that irrelevant information is not computed; it also enables the labeled transition system itself to be computed in a demand-driven manner. The potential savings of this technique are analyzed in the context of a case study involving a real-time modification to the Ethernet protocol.

References

1. *Twelfth Annual ACM Symposium on Principles of Programming Languages (POPL '85)*, New Orleans, Louisiana, January 1985. ACM Press.
2. *Symposium on Logic in Computer Science (LICS '86)*, Cambridge, Massachusetts, June 1986. IEEE Computer Society Press.
3. P. A. Abdulla and B. Jonsson. Verifying programs with unreliable channels. *Information and Computation*, 127(2):91–101, June 1996.
4. R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real time. *Information and Computation*, 104(1):2–34, May 1993.
5. R. Alur, C. Courcoubetis, T. Henzinger, N. Halbwachs, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, February 1995.
6. H.R. Andersen. Model checking and boolean graphs. *Theoretical Computer Science*, 126(1):3–30, April 1994.
7. J.C.M. Baeten, editor. *Applications of Process Algebra*, volume 17 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, England, 1990.
8. G. Bhat and R. Cleaveland. Efficient local model checking for fragments of the modal μ-calculus. In Margaria and Steffen [53], pages 107–126.
9. G. Bhat and R. Cleaveland. Efficient model checking via the equational μ-calculus. In *Eleventh Annual Symposium on Logic in Computer Science (LICS '96)*, pages 304–312, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.
10. B. Bloom and R. Paige. Transformational design and implementation of a new efficient solution to the ready simulation problem. *Science of Computer Programming*, 24(3):189–220, June 1995.
11. T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.
12. A. Bouali, A. Ressouche, V. Roy, and R. de Simone. The FC2TOOLS set. In Margaria and Steffen [53], pages 396–397.
13. A. Browne, E. Clarke, S. Jha, D. Long, and W. Marrero. An improved algorithm for the evaluation of fixpoint expressions. *Theoretical Computer Science*, 178(1–2):237–255, May 1997.
14. M.C. Browne, E.M. Clarke, and D. Dill. Automatic circuit verification using temporal logic: Two new examples. In *Formal Aspects of VLSI Design*, 1985.
15. M.C. Browne, E.M. Clarke, D. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computing*, C-35(12):1035–1044, December 1986.
16. R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.
17. O. Burkart and B. Steffen. Model-checking the full-modal mu-calculus for infinite sequential processes. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Automata, Languages and Programming (ICALP '97)*, volume 1256 of *Lecture Notes in Computer Science*, pages 419–429, Bologna, Italy, July 1997. Springer-Verlag. Full version to appear in *Theoretical Computer Science*.
18. U. Celikkan and R. Cleaveland. Generating diagnostic information for behavioral preorders. *Distributed Computing*, 9:61–75, 1995.
19. K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, Massachusetts, 1988.
20. E.M. Clarke and E.A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In D. Kozen, editor, *Logics of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, May 1981. Springer-Verlag.
21. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *Tenth Annual ACM Symposium on Principles of Programming Languages (POPL '83)*, pages 117–126, Austin, Texas, January 1983. ACM Press.
22. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.
23. E.M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in model checking. In Courcoubetis [30], pages 450–462.
24. E.M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks. *ACM Transactions on Programming Languages and Systems*, 19(5):726–750, September 1997.
25. E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
26. E.M. Clarke and J.M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

27. R. Cleaveland and M.C.B. Hennessy. Testing equivalence as a bisimulation equivalence. *Formal Aspects of Computing*, 5:1–20, 1993.
28. R. Cleaveland and S. Smolka. Strategic directions in concurrency research. *ACM Computing Surveys*, 28(4):607–625, December 1996.
29. R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2:121–147, 1993.
30. C. Courcoubetis, editor. *Computer Aided Verification (CAV '93)*, volume 697 of *Lecture Notes in Computer Science*, Elounda, Greece, June/July 1993. Springer-Verlag.
31. M. Dam. CTL* and ECTL* as fragments of the modal mu-calculus. *Theoretical Computer Science*, 126(1):77–96, April 1994.
32. W. Elseaidy, R. Cleaveland, and J.W. Baugh Jr. Modeling and verifying active structural control systems. *Science of Computer Programming*, 29(1–2):99–122, July 1997.
33. E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. North-Holland, 1990.
34. E.A. Emerson and E.M. Clarke. Characterizing correctness properties of parallel programs as fixpoints. In J. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming (ICALP '80)*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181, Utrecht, July 1980. Springer-Verlag.
35. E.A. Emerson and J.Y. Halpern. 'Sometime' and 'not never' revisited: On branching versus linear time temporal logic. *Journal of the Association for Computing Machinery*, 33(1):151–178, January 1986.
36. E.A. Emerson, C. Jutla, and A.P. Sistla. On model-checking for fragments of μ-calculus. In Courcoubetis [30], pages 385–396.
37. E.A. Emerson and C.-L. Lei. Modalities for model checking: Branching time strikes back. In *Twelfth Annual ACM Symposium on Principles of Programming Languages (POPL '85)* [1], pages 84–96.
38. E.A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Symposium on Logic in Computer Science (LICS '86)* [2], pages 267–278.
39. E.A. Emerson and A.P. Sistla. Symmetry and model checking. In Courcoubetis [30], pages 463–478.
40. J. Esparza. Decidability of model-checking for infinite-state concurrent systems. *Acta Informatica*, 34:85–107, 1997.
41. J.-C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13:219–236, 1989/1990.
42. R. Goering. Model checking expands verification's scope. *EE Times*, 1997. Issue 939, February 3, 1997.
43. S. Graf. Characterization of a sequentially consistent memory and verification of a cache memory by abstraction. *Distributed Computing*, 12(2+3):75–99, 1999.
44. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Computer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997. Springer-Verlag.
45. J.F. Groote and F. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In M.S. Paterson, editor, *Automata, Languages and Programming (ICALP '90)*, volume 443 of *Lecture Notes in Computer Science*, pages 626–638, Warwick, England, July 1990. Springer-Verlag.
46. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985.
47. G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
48. G.J. Holzmann and D. Peled. The state of SPIN. In R. Alur and T. Henzinger, editors, *Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 385–389, New Brunswick, New Jersey, July 1996. Springer-Verlag.
49. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, December 1983.
50. L. Lamport. 'sometimes' is sometimes 'not never'. In *Seventh Annual ACM Symposium on Principles of Programming Languages (POPL '80)*, pages 174–185, Las Vegas, Nevada, January 1980. ACM Press.
51. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Twelfth Annual ACM Symposium on Principles of Programming Languages (POPL '85)* [1], pages 97–107.
52. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, Berlin, 1992.
53. T. Margaria and B. Steffen, editors. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96)*, volume 1055 of *Lecture Notes in Computer Science*, Passau, Germany, March 1996. Springer-Verlag.
54. R. Milner. *Communication and Concurrency*. Prentice-Hall, London, 1989.
55. R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, December 1987.
56. A. Pnueli. The temporal logic of programs. In 18^{th} *Annual Symposium on Foundations of Computer Science*, pages 46–57, Providence, Rhode Island, October/November 1977. IEEE.
57. J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Proceedings of the International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351, Turin, April 1982. Springer-Verlag.
58. T. Rathje and S. Sandler. CPU formal verification receives a boost. *EE Times*, 1996. Issue 927, November 11, 1996.
59. J. Richier, C. Rodgriguez, J. Sifakis, and J. Voiron. Verification in XESAR of the sliding window protocol. In *Proceedings of the IFIP Symposium on Protocol Specification, Testing and Verification*, pages 235–250, Zurich, May 1987. North-Holland.
60. A.W. Roscoe. Model-checking CSP. In A.W. Roscoe, editor, *A Classical Mind: Essays in Honour of CAR Hoare*, chapter 21, pages 353–378. Prentice-Hall, February 1994.
61. C. Stirling. Modal and temporal logics. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 477–563. Oxford University Press, 1992.
62. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 25(2):285–309, 1955.
63. M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science (LICS '86)* [2], pages 332–344.