Model Checking and the Mu-calculus

E. Allen Emerson

University of Texas at Austin, Austin, Tx 78712, USA

Abstract. There is a growing recognition of the need to apply formal mathematical methods in the design of "high confidence" computing systems. Such systems operate in safety critical contexts (e.g., air traffic control systems) or where errors could have major adverse economic consequences (e.g., banking networks). The problem is especially acute in the design of many *reactive* systems which must exhibit correct ongoing behavior, yet are not amenable to thorough testing due to their inherently nondeterministic nature. One useful approach for specifying and reasoning about correctness of such systems is temporal logic *model checking*, which can provide an efficient and expressive tool for automatic verification that a finite state system meets a correctness specification formulated in temporal logic. We describe model checking algorithms and discuss their application. To do this, we focus attention on a particularly important type of temporal logic known as the *Mu-calculus*.

1 Introduction

There is a growing need for reliable methods of designing correct reactive systems. These systems are characterized by ongoing, typically nonterminating and highly nondeterministic behavior. Often such systems amount to parallel or distributed programs. Examples include operating systems, network protocols, and air traffic control systems.

There is nowadays widespread agreement that some type of temporal logic [Pn77] provides an extremely useful framework for reasoning about reactive programs. Basic temporal operators such as "sometimes" (F), "always" (G), and "nexttime" (X) make it possible to easily express many important correctness properties; e.g., $G(sent \Rightarrow Freceived)$ asserts that whenever a message is sent, it is eventually received.

When we introduce "path quantifiers" (A, E), meaning "for all possible future computations" and "for some possible future computation", respectively, we can distinguish between the inevitability of events (AFP) and their potentiality (EFP). Such a system is referred to as a branching time temporal logic.

One commonly used branching time logic is CTL (Computation Tree Logic) (cf. [EC80], [CE81]).

Another branching time logic is the (propositional) Mu-calculus [Ko83] (cf. [EC80], [Pr81]). The Mu-calculus may be thought of as extending CTL with a least fixpoint (μ) and greatest fixpoint (ν) operator. We note that $EFP \equiv P \vee EXEFP$, so that EFP is a fixed point, also known as a fixpoint, of the expression $\tau(Y) = P \vee EXY$. In fact, EFP is the least fixpoint, i.e., the least $Y \equiv P \vee EXY$. The least fixpoint of $\tau(Y)$ is ordinarily denoted as $\mu Y.\tau(Y)$. As this example suggests, not all of CTL is needed as a "basis" for the Mu-calculus, which can instead be defined in terms of atomic proposition constants and variables $(P, ..., ; Y, ...)$, boolean connectives $(\wedge, \vee, \neg)$, nexttime operators (AX, EX), and finally least and greatest fixpoint operators (μ, ν). The rest of the CTL operators can be defined in terms of these surprisingly simple primitives. In fact, most modal and temporal logics of interest can be defined in terms of the Mu-calculus. In this way, it provides a single, simple and uniform framework subsuming most other logics of interest for reasoning about reactive systems (cf. [EL86]).

The "classical" approach to the use of temporal logic for reasoning about reactive programs is a manual one, where one is obliged to construct by hand a proof of program correctness using axioms and inference rules in a deductive system. A desirable aspect of some such proof systems is that they may be formulated so as to be "compositional", which facilitates development of a program hand in hand with its proof of correctness by systematically composing together proofs of constituent subprograms. Even so, manual proof construction can be extremely tedious and error prone, due to the large number of details that must be attended to. Hence, correct proofs for large programs are often very difficult to construct and to organize in an intellectually manageable fashion. It seems clear that it is unrealistic to expect manual proof construction to be feasible for large-scale reactive systems. For systems with millions or even just tens of thousands of lines of codes, transcription and other clerical errors guarantee that the task of proof construction is beyond the ability of humans by themselves.

Hence, we have historically advocated an alternative, automated approach to reasoning about reactive systems (cf. [Em81], [CE81]). One of the more useful approaches for specifying and reasoning about correctness of such systems has turned out to be temporal logic *model checking* (cf. [CE81], [Em81], [QS82]), which can provide an efficient and expressive tool for automatic verification that a finite state reactive system meets a correctness specification formulated in propositional temporal logic. Empirically, it turns out that many systems of interest either are or can be usefully modeled at some level of abstraction as finite state systems. Moreover, the propositional fragment of temporal logic suffices to specify their important correctness properties.[1] The model checking problem can be formalized as:

> The Model Checking Problem - Given a finite state transition graph M, an initial state s_0 of M, and a temporal logic specification formula f, does $M, s_0 \models f$? i.e., is M at s_0 a model of f?

Variant formulations of the model checking problem stipulate calculating the set of all such states s_0 in M where f is true.

The remainder of this paper is organized as follows: Section 2 defines the Mu-calculus. Section 3 defines certain related logics including CTL. The expressiveness of the Mu-calculus is discussed in section 4. Algorithms for model checking in the Mu-calculus are described in section 5. Section 6 gives some concluding remarks.

2 The Mu-calculus

The (propositional) Mu-Calculus (cf. [Pa70], [EC80], [Ko83]) provides a *least fixpoint* operator (μ) and a *greatest fixpoint* operator (ν), which make it possible to give extremal *fixpoint characterizations* of correctness properties. Intuitively, the Mu-Calculus makes it possible to characterize the modalities in terms of recursively defined tree-like patterns. For example, the assertion that "along all computation paths p will become true eventually" can be characterized as $\mu Z.p \vee AXZ$, the least fixpoint of the functional $p \vee AXZ$ where Z is an atomic proposition variable (intuitively ranging over sets of states) and AX denotes the universal nexttime operator.

We first give the formal definition of the Mu-Calculus.

Syntax

The formulae of the propositional Mu-Calculus $L\mu$ are those generated by rules (1)-(6):

(1) Atomic proposition constants P, Q
(2) Atomic proposition variables $Y, Z, \ldots$
(3) EXp, where p is any formula.
(4) $\neg p$, the negation of formula p.
(5) $p \wedge q$, the conjunction of formulae p, q.

[1] These two assertions are related. Most propositional temporal logics satisfy the finite model property: if a specification is satisfiable, it has a finite model which may be viewed as a system meeting the specification.

(6) $\mu Y.p(Y)$, where $p(Y)$ is any formula syntactically monotone in the propositional variable Y, i.e., all occurrences of Y in $p(Y)$ fall under an even number of negations.

The set of formulae generated by the above rules forms the language $L\mu$. The other connectives are introduced as abbreviations in the usual way: $p \vee q$ abbreviates $\neg(\neg p \wedge \neg q)$, $p \Rightarrow q$ abbreviates $\neg p \vee q$, $p \equiv q$ abbreviates $p \Rightarrow q \wedge q \Rightarrow p$, AXp abbreviates $\neg EX\neg p$, $\nu Y.p(Y)$ abbreviates $\neg\mu Y.\neg p(\neg Y)$, etc. Intuitively, $\mu Y.p(Y)$ ($\nu Y.p(Y)$) stands for the least (greatest, resp.) fixpoint of $p(Y)$, EXp (AXp) means p is true at some (every) successor state reachable from the current state, $\wedge$ means "and", etc. We use $|p|$ to denote the *length* (i.e., number of symbols) of p.

We say that a formula q is a *subformula* of a formula p provided that q, when viewed as a sequence of symbols, is a substring of p. A subformula q of p is said to be *proper* provided that q is not p itself. A *top-level* (or *immediate*) subformula is a maximal proper subformula. We use $SF(p)$ to denote the set of subformulae of p.

The fixpoint operators μ and ν are somewhat analogous to the quantifiers $\exists$ and $\forall$. Each occurrence of a propositional variable Y in a subformula $\mu Y.p(Y)$ (or $\nu Y.p(Y)$) of a formula is said to be *bound*. All other occurrence are *free*. By renaming variables if necessary we can assume that the expression $\mu Y.p(Y)$ (or $\nu Y.p(Y)$) occurs at most once for each Y.

A *sentence* (or *closed* formula) is a formula that contains no free propositional variables, i.e., every variable is bound by either μ or ν. A formula is said to be in *positive normal form* (PNF) provided that no variable is quantified twice and all the negations are applied to atomic propositions only. Note that every formula can be put in PNF by driving the negations in as deep as possible using DeMorgan's Laws and the dualities $\neg\mu Y.p(Y) = \nu Y.\neg p(\neg Y), \neg\nu Y.p(Y) = \mu Y.\neg p(\neg Y)$. (This can at most double the length of the formula). *Subsentences* and *proper subsentences* are defined in the same way as subformulae and proper subformulae.

Let σ denote either μ or ν. If Y is a bound variable of formula p, there is a unique μ or ν subformula $\sigma Y.q(Y)$ of p in which Y is quantified. Denote this subformula by σY. Y is called a *μ-variable* if $\sigma Y = \mu Y$; otherwise, Y is called a *ν-variable*. A *σ-subformula* (*σ-subsentence*, resp.) is a subformula (subsentence) whose main connective is either μ or ν. We say that q is a *top-level* σ-subformula of p provided q is a proper σ-subformula of p but not a proper σ-subformula of any other σ-subformula of p. Finally, a *basic modality* is a σ-sentence that has no proper σ-subsentences.

Semantics

We are given a set Σ of atomic proposition constants and a set Γ of atomic proposition variables. We let AP denote $\Sigma \cup \Gamma$. Sentences of the propositional Mu-Calculus $L\mu$ are interpreted with respect to a structure $M = (S, R, L)$ where

S is the set of *states*,
R is a total *binary relation* $\subseteq S \times S$ (i.e., one where $\forall s \in S \exists t \in S (s,t) \in R$), and
L: $S \to 2^{AP}$ is a labeling which associates with each state s a set $L(s)$ consisting of all atomic proposition symbols in the underlying set of atomic propositions AP intended to be true at state s.

We may view M as a labeled, directed graph with node set S, arc set R, and node labels given by L. The size of M, $|M| = |S| + |R|$, where $|S|$ is the size of the state space S, the sum over $s \in S$ of the sizes of $L(s)$, and $|R|$ is the number of transitions.

The power set of S, 2^S, may be viewed as the complete lattice ($2^S, S, \emptyset, \subseteq, \cup, \cap$). Intuitively, we identify a predicate with the set of states which make it true. Thus, *false*, which corresponds to the empty set, is the bottom element, *true*, which corresponds to S is the top element, and implication ($\forall s \in S[P(s) \Rightarrow Q(s)]$), which corresponds to simple set-theoretic containment ($P \subseteq Q$), provides the partial ordering on the lattice.

Let functional $\tau : 2^S \to 2^S$ be given; then we say that $\tau(Y)$ is *monotonic* provided that $P \subseteq Q$ implies $\tau(P) \subseteq \tau(Q)$. We say P' is a *fixpoint* of functional $\tau(Y)$ provided $P' = \tau(P')$. Fixpoint P' is a *least* fixpoint provided that if P'' is a fixpoint then $P' \subseteq P''$. Note that any least fixpoint of $\tau(Y)$ is unique, since, if P' and P'' are least fixpoints, $P' \subseteq P''$ and $P'' \subseteq P'$. We use $\mu Y.\tau(Y)$ to denote the least fixpoint; analogously, $\nu Y.\tau(Y)$ denotes the greatest fixpoint. The existence of these of these "extremal" fixpoints is guaranteed by the following.

Theorem (Tarski-Knaster). Let $\tau : 2^S \to 2^S$ be a monotonic functional. Then

(a) $\mu Y.\tau(Y) = \cap \{Y : \tau(Y) = Y\} = \cap \{Y : \tau(Y) \subseteq Y\}$,
(b) $\nu Y.\tau(Y) = \cup \{Y : \tau(Y) = Y\} = \cup \{Y : \tau(Y) \supseteq Y\}$,
(c) $\mu Y.\tau(Y) = \cup_i \tau^i(\mathit{false})$ where i ranges over all ordinals of cardinality at most that of the state space S, so that when S is finite i ranges over $[0{:}|S|]$; $\mu Y.\tau(Y)$ is the union of the following ascending chain of approximations: $\mathit{false} \subseteq \tau(\mathit{false}) \subseteq \tau^2(\mathit{false}) \ldots$, and
(d) $\nu Y.\tau(Y) = \cap_i \tau^i(\mathit{true})$ where i ranges over all ordinals of cardinality at most that of the state space S, so that when S is finite i ranges over $[0{:}|S|]$; $\nu Y.\tau(Y)$ is the intersection of the following descending chain of approximations: $\mathit{true} \supseteq \tau(\mathit{true}) \supseteq \tau^2(\mathit{true}) \ldots$.

A formula p with free variables $Y_1, \ldots, Y_n$ is thus interpreted as a mapping p^M from $(2^S)^n$ to 2^S, i.e., it is interpreted as a predicate transformer. We write $p(Y_1, \ldots, Y_n)$ to denote that all free variables of p are among $Y_1, \ldots, Y_n$. A *valuation* $\mathcal{V}$, denoted $(V_1, \ldots, V_n)$, is an assignment of the subsets of S, $V_1, \ldots, V_n$, to free variables $Y_1, \ldots, Y_n$, respectively. We use $p^M(\mathcal{V})$ to denote the value of p on the (actual) arguments $V_1, \ldots, V_n$ (cf. [EC80], [Ko83]). The operator p^M is defined inductively as follows:

(1) $P^M(\mathcal{V}) = \{s : s \in S \text{ and } P \in L(s)\}$ for any atomic propositional constant $P \in AP$
(2) $Y_i^M(\mathcal{V}) = V_i$
(3) $(p \wedge q)^M(\mathcal{V}) = p^M(\mathcal{V}) \cap q^M(\mathcal{V})$
(4) $(\neg p)^M(\mathcal{V}) = S \backslash (p^M(\mathcal{V}))$
(5) $(EXp)^M(\mathcal{V}) = \{s : \exists t \in p^M(\mathcal{V}), (s,t) \in R\}$
(6) $\mu Y_1.p(Y_1)^M(\mathcal{V}) = \cap\{S' \subseteq S : p(Y_1)^M(S', V_2, \ldots, V_n) \subseteq S'\}$

Note that our syntactic restrictions on monotonicity ensure that least (as well as greatest) fixpoints are well-defined.

Usually we write $M, s \models p$ (respectively, $M, s \models p(\mathcal{V})$) instead of $s \in p^M$ (respectively, $s \in p^M(\mathcal{V})$) to mean that sentence (respectively, formula) p is true in structure M at state s (under valuation $\mathcal{V}$). When M is understood, we write simply $s \models p$.

3 Temporal Logics

In this section we define three representative systems of propositional temporal logic. The system PLTL (Propositional Linear temporal logic) is the "standard" linear time temporal logic (cf. [Pn77], [MP92]). The branching time logic, CTL (Computational Tree Logic), allows basic temporal operators of the form: a path quantifier—either A ("for all futures") or E ("for some future"—followed by a single one of the usual linear temporal operators G ("always"), F ("sometime"), X ("nexttime"), or U ("until") (cf. [CE81], [EC80]). Its syntactic restrictions limit its expressive power so that, for example, correctness under fair scheduling assumptions cannot be expressed. We therefore also consider the much richer language CTL*, which extends CTL by allowing basic temporal operators where the path quantifier (A or E) is followed by an arbitrary linear time formula, allowing boolean combinations and nestings, over F, G, X, and U (cf. [EH86]).
Syntax

We now give a formal definition of the syntax of CTL*. We inductively define a class of state formulae (true or false of states) using rules S1-3 below and a class of path formulae (true or false of paths) using rules P1-3 below:

S1 Each atomic proposition P is a state formula
S2 If p, q are state formulae then so are $p \wedge q, \neg p$
S3 If p is a path formula then Ep, Ap are state formulae
P1 Each state formula is also a path formula
P2 If p, q are path formulae then so are $p \wedge q, \neg p$
P3 If p, q are path formulae then so are Xp, pUq

The set of state formulae generated by the above rules forms the language CTL*. The other connectives can then be introduced as abbreviations in the usual way: $p \vee q$ abbreviates $\neg(\neg p \wedge \neg q)$, $p \Rightarrow q$ abbreviates $\neg p \vee q$, $p \equiv q$ abbreviates $p \Rightarrow q \wedge q \Rightarrow p$, Fp abbreviates $trueUq$, and Gp abbreviates $\neg F \neg p$. We also let $\overset{\infty}{F}\, p$ abbreviate GFp ("infinitely often"), $\overset{\infty}{G}\, p$ abbreviate FGp ("almost everywhere"), and (pBq) abbreviate $\neg((\neg p)Uq)$ ("before").

Remark: We could take the view that Ap abbreviates $\neg E \neg p$, and give a more terse syntax in terms of just the primitive operators $E, \wedge, \neg, X$, and U. However, the present approach makes it easier to give the syntax of the sublanguage CTL below.

The restricted logic CTL is obtained by restricting the syntax to disallow boolean combinations and nestings of linear time operators. Formally, we replace rules P1-3 by

P0 If p, q are state formulae then Xp, pUq are path formulae.

The set of state formulae generated by rules S1-3 and P0 forms the language CTL. The other boolean connectives are introduced as above while the other temporal operators are defined as abbreviations as follows: EFp abbreviates $E(trueUp)$, AGp abbreviates $\neg EF \neg p$, AFp abbreviates $A(trueUp)$, and EGp abbreviates $\neg AF \neg p$. (Note: this definition can be seen to be consistent with that of CTL*.)

Finally, the set of path formulae generated by rules S1,P1-3 define the syntax of the linear time logic PLTL.

Semantics

A formula of CTL* is interpreted with respect to a structure $M = (S, R, L)$ as is the Mu-calculus.

A *fullpath* of M is an infinite sequence $s_0, s_1, s_2, \ldots$ of states such that $\forall i$ $(s_i, s_{i+1}) \in R$. We use the convention that $x = (s_0, s_1, s_2, \ldots)$ denotes a fullpath, and that x^i denotes the suffix path $(s_i, s_{i+1}, s_{i+2}, \ldots)$. We write $M, s_0 \models p$ (respectively, $M, x \models p$) to mean that state formula p (respectively, path formula p) is true in structure M at state s_0 (respectively, of fullpath x). We define $\models$ inductively as follows:

S1 $M, s_0 \models P$ iff $P \in L(s_0)$
S2 $M, s_0 \models p \wedge q$ iff $M, s_0 \models p$ and $M, s_0 \models q$
$M, s_0 \models \neg p$ iff it is not the case that $M, s_0 \models p$
S3 $M, s_0 \models Ep$ iff $\exists$ fullpath $x = (s_0, s_1, s_2, \ldots)$ in M, $M, x \models p$
$M, s_0 \models Ap$ iff $\forall$ fullpath $x = (s_0, s_1, s_2, \ldots)$ in M, $M, x \models p$
P1 $M, x \models p$ iff $M, s_0 \models p$
P2 $M, x \models p \wedge q$ iff $M, x \models p$ and $M, x \models q$
$M, x \models \neg p$ iff it is not the case that $M, x \models \neg p$
P3 $M, x \models pUq$ iff $\exists i\ [M, x^i \models q$ and $\forall j\ (j < i$ implies $M, x^j \models p)]$
$M, x \models Xp$ iff $M, x^1 \models p$

A formula of CTL is also interpreted using the CTL* semantics, using rule P3 for path formulae generated by rule P0.

Similarly, a formula of PLTL, which is a "pure path formula" of CTL* is interpreted using the above CTL* semantics.

We say that a state formula p (resp., path formula p) is *valid* provided that for every structure M and every state s (resp., fullpath x) in M we have $M, s \models p$ (resp., $M, x \models p$). A state formula p (resp., path formula p) is *satisfiable* provided that for some structure M and some state s (resp., fullpath x) in M we have $M, s \models p$ (resp., $M, x \models p$).

A formula of CTL* is a *basic modality* provided that it is of the form Ap or Ep where p itself contains no A's or E's, i.e., p is an arbitrary formula of PLTL. Similarly, a basic modality of CTL is of the form Aq or Eq where q is one of the *single* linear temporal operators F, G, X, or U applied to pure propositional arguments. A CTL* (respectively, CTL) formula can now be thought of as being built up out of boolean combinations and nestings of basic modalities (and atomic propositions).

4 Expressiveness

The Mu-calculus is of considerable importance for several reasons, which, overall, relate to its expressiveness. First, the Mu-calculus provides a single, elegant, uniform logical framework of great raw expressive power that subsumes most modal and temporal logics of programs, and related formalisms. Both CTL and CTL* can be translated into the Mu-calculus, as well as most other commonly used modal and temporal logics of programs. It can be shown that the Mu-calculus, over infinite binary trees, coincides in expressive power with finite state tree automata; in fact, Mu-calculus formulas are really alternating finite state automata on infinite trees (cf. [EJ91]). Second, the semantics of the Mu-calculus is firmly anchored in the fundamental Tarski-Knaster theorem and the basic notion of inductive definability. This provides a ready means to do model checking, i.e., check whether a given structure defines a model of a given specification as discussed in the next section. Finally, the translation of most logics and formalisms turns out to require only small syntactic fragments of the Mu-calculus. This has implications for the complexity of model checking as also discussed in the next section.

The above-mentioned syntactic fragments are determined by "alternation depth" of a Mu-calculus formula. Intuitively, the alternation depth refers to the depth of "significant" nesting of alternating μ's and ν's. An alternation of either of the following forms is "insignificant"

$$(*)\ \mu Y.f(Y, \nu Z.g(Z)) \qquad \text{or} \qquad \nu Y.f(Y, \mu Z.g(Z))$$

as the inner σ-formula is a sentence and is not influenced by the surrounding σ-formula of opposite "polarity". In contrast, an alternation of either of the following forms is significant

$$(**)\ \mu Y.f(\nu Z.g(Y, Z)) \qquad \text{or} \qquad \nu Y.f(\mu Z.g(Y, Z))$$

as a free occurrence of μ variable Y appears within the scope of νZ or a free occurrence of ν Y occurs within the scope of a μZ.

We can give the technical definition of the alternation depth $ad(f)$ of formula f as follows (cf. [EL86], [An93]). We assume that f is initially placed in positive normal form.

$ad(P) = ad(Y) = 0$ for atomic proposition constants P and variables Y.
$ad(f \wedge g) = ad(f \vee g) = \max\ \{ad(f), ad(g)\}$
$ad(\neg f) = ad(f)$
$ad(\mu Y.f) = 1 + \max\ \{ad(\nu Z.g) : \nu Z.g$ is a subformula of f in which Y occurs free $\}$
$ad(\nu Y.f) = 1 + \max\ \{ad(\mu Z.g) : \mu Z.g$ is a subformula of f in which Y occurs free $\}$

Let $L\mu_k$ denote the Mu-Calculus $L\mu$ restricted to formulas of alternation depth at most k. Most modal or temporal logics of programs can be translated into $L\mu_1$ or $L\mu_2$, often succinctly (cf. [EL86]).

For example, below we give characterizations of CTL basic modalities in terms of least or greatest fixpoints. Note that each is a formula of $L\mu_1$.

$$EFP \equiv \mu Z.P \vee EXZ$$
$$AGP \equiv \nu Z.P \wedge AXZ$$
$$AFP \equiv \mu Z.P \vee AXZ$$
$$EGP \equiv \nu Z.P \wedge EXZ$$
$$A(P\ U\ Q) \equiv \mu Z.Q \vee (P \wedge AXZ)$$
$$E(P\ U\ Q) \equiv \mu Z.Q \vee (P \wedge EXZ)$$

These fixpoint characterizations are simple and plausible. They also turn out to be important in applications as they underly the original CTL model checking algorithm of [CE81] as well as the "symbolic" approaches developed later and discussed in section 5.5.

Below we sketch representative proofs of correctness for some of these fixpoint characterizations (cf. [EC80], [EL86]). We assume that each proof is conducted in the context of an arbitrarily chosen underlying structure M.

Proposition. $EFP \equiv \mu Z.P \vee EXZ$

Proof idea. Let $g(Z) = P \vee EXZ$. Then $g^i(\mathit{false})$ corresponds to the set of states from which it is possible to reach a state satisfying P by a path of length at most i states. Plainly, s satisfies EFP iff $s \in g^i(\mathit{false})$ for some i. □

Proposition. $AGP \equiv \nu Z.P \wedge AXZ$.

Proof Idea. We use duality, interchanging the connective A with E, F with G, μ with ν, $\vee$ with $wedge$, and X with itself. Thus AGP is converted into its dual EFP while $\mu Z.P \vee EXZ$ is converted into its dual $\nu Z.P \wedge AXZ$. The result follows by the preceeding proposition. □

Proposition. $AFP \equiv \mu Z.P \vee AXZ$.

Proof Idea. Establish the dual claim that $EGP \equiv \nu Z.P \wedge EXZ$. Let $f(Z)$ = $P \wedge EXZ$. First note that EGP is a fixpoint of f(Z), viz. $EGP \equiv f(EGP)$. Now suppose Y is an arbitrary fixpoint of $f(Z)$, so that $Y \equiv f(Y)$. Let s_0 be an arbitrary state of Y. As $s_0 \in Y$, then by virtue of f, s_0 is in (the set of states satisfying) P. Moreover, s_0 has a successor $s_1 \in Y$. Apply to s_1 the argument. We get an infinite computation path comprised of consecutive states $s_0, s_1, s_2, \ldots$ each of which is in Y and also satisfies P. This path witnesses the truth of EGP at s_0. Thus every state in Y satisfies EGP and Y is a subset of (the set of states satisfying) EGP. Hence, any fixpoint of f is a subset of EGP, which must be the greatest fixpoint, thereby establish the dual claim. □

These fixpoint characterizations of CTL basic modalities provide the key for translating all of CTL into the Mu-calculus, viz., $L\mu_1$ ([EC80], [EL86]). For instance, the CTL formula $AG(AFP \wedge EFQ)$ can be seen to be comprised of basic modalities of the form EF, AF, and AG. Expanding the fixpoint characterizations and doing appropriate substitutions we get that $AG(AFP \wedge EFQ) \equiv \nu Y.((\mu Z.P \vee AXZ) \wedge (\mu Z'.Q \vee EXZ')) \wedge AXY$. Observe that the result is still of alternation depth 1.

The translation of CTL* into the Mu-calculus is more involved. Each basic modality of CTL* is of the form Eh where h is a PLTL formula. After first being converted to an automaton on infinite strings, h can be converted an equivalent ω-regular expression h' (cf. [ES83], [VW83]). Eh' is readily rendered in $L\mu_2$. This permits us to translate all of CTL* into $L\mu_2$ as above [EL86].

For example, if h is the temporal logic formula $G(PUQ)$ the corresponding ω-regular expression is $(P^*Q)^\omega$, denoting the set of all infinite strings that are of this form: an (ω) infinite repetition of finite strings comprised of finitely many consecutive P's followed by a single Q. Then $E((P^*Q)^\omega) \equiv \nu Y.\mu Z.(Q \wedge EXY) \vee (P \wedge EXZ)$. If we take P to be *true*, this property simplifies, in ω-regular expression notation, to $E(true^*Q)^\omega$. This is equivalent to $E\overset{\infty}{F}Q$ meaning "along some path Q occurs infinitely often", whose fixpoint characterization may be simplified to $\nu Y.\mu Z.EX(Q \wedge Y \vee Z)$ in $L\mu_2$. As suggested , such properties associated with fairness can be expressed in alternation depth 2. In fact, they require alternation depth 2. It can be shown that $E\overset{\infty}{F}Q$ is not expressible by any alternation depth 1 formula (cf. [EC80] [EL86]).

The existence of these translations witnesses the generality of the Mu-calculus. The translations are important in practice because correctness specifications written in logics such as CTL or CTL* are often more readable than specifications written directly in the Mu-calculus. In fact, it turns out to be rather easy to write down highly inscrutable Mu-calculus formulae for which there is no readily apparent intuition regarding their intended meaning. Since Mu-calculus formula are really alternating tree automata, perhaps this is not so surprising. After all, even such basic automata as *deterministic* finite state automata on *finite strings* can be highly complex, incomprehensible "bowls of spaghetti". On the other hand, many Mu-calculus characterizations of correctness properties are elegant, and the formalism seems to have found increasing favor, especially in Europe, owing to its simple and elegant underlying mathematical structure. In any event, many people find that the translations serve to "tame" the Mu-calculus, making its expressive power more useful.

For many years it was not known if the higher alternation depths form a true hierarchy of expressive power. Recently, affirmative solutions to this open problem were reported in [Br96] and [Le96]. In practice, it seems to make little difference, since it does appear that everything practical is in alternation depth 2. However, it is of theoretical interest. Moreover, the question has some bearing on the complexity of model checking in the overall Mu-calculus as discussed next.

5 Model Checking

What has turned out to be one of the more useful techniques for automated reasoning about reactive systems began with the advent of efficient temporal logic *model checking* [CE81] (cf. [Em81], [QS82], [CES86]). The basic idea is that the global state transition graph of a finite state reactive system defines a (Kripke) structure in the sense of temporal logic (cf. [Pn77]), and we can give an efficient algorithm for checking if the state graph defines a model of a given specification expressed in an appropriate temporal logic. While earlier work in the protocol community had addressed the problem of analysis of simple reachability properties, model checking provided an expressive, uniform specification language in the form of temporal logic along with a single, efficient verification algorithm which automatically handled a wide variety of correctness properties.

5.1 Taxonomy of Model Checking Approaches

It is possible to give a rough taxonomy of model checking methods according to certain criteria:

Explicit State Representation versus Symbolic State Representation. In the explicit state approach the Kripke structure is represented extensionally using conventional data structures such as adjacency matrices and linked lists so that each state and transition is enumerated explicitly. In contrast, in the symbolic approach boolean expressions denote large Kripke structures implicitly. Typically, the data structure involved is that of Binary Decision Diagrams (BDDs), which can, in many applications, although not always, manipulate boolean expressions denoting large sets of states efficiently.

The distinction between explicit state and symbolic representations is to a large extent an implementation issue rather than a conceptual one. The original model checking method was based on an algorithm for fixpoint computation it was implemented using explicit state representation. The subsequent symbolic model checking method uses the same fixpoint computation algorithm, but now represents sets of states implicitly. However, the succinctness of BDD data structures underlying the implementation can make a significant practical difference.

Global Calculation versus Local Search. In the global approach, we are given a structure M and formula f. The algorithm calculates $f^M = \{s : M, s \models f\}$, the set of all states in M where f is true. This necessarily entails examining the entire structure. Global algorithms typically proceed by induction on the formula structure, calculating g^M for the various subformulae g of f. The algorithm can be presented in recursive form; as the recursion "unwinds" the values of the shortest subformula are calculated first, then the next shortest, etc.

In contrast, in the local approach, we are given a specific state s_0 in M along with M and f. We wish to determine whether $M, s_0 \models f$. The computation proceeds by performing a search of M starting at s_0. The potential advantage is that, many times in practice, only a portion of M may need to be examined to settle the question. In the worst case, however, it may still be necessary to examine all of M. (cf. [SW89]).

Monolithic Structures versus Incremental Algorithms. To some extent this is also more of an implementation issue than a conceptual one. Again, however, it can have significant practical consequences. In the monolithic approach, the entire structure M is built and represented at one time in computer memory. While conceptually simple and consistent with standard conventions for judging the complexity of graph algorithms, in practice this may be highly undesirable because the entire graph of M may not fit in computer memory at once. In contrast, the incremental approach (also referred to as the "on-the-fly" or "on-line" approach) entails building and storing only small portions of the graph of M at any one time (cf. [JT89]).

5.2 Extensional Model Checking Algorithms

Technically, the Tarski-Knaster theorem can be understood as providing a systematic basis for model checking. The specifications can be formulated in the Mu-calculus or in other logics such as CTL which, as noted above, are readily translatable into the Mu-calculus. For example, to calculate the states where the CTL basic modality EFP holds in structure $M = (S, R, L)$, we use the fixpoint characterization $EFP \equiv \mu Z.\tau(Z)$, with $\tau(Z) \equiv P \vee EXZ$. We successively calculate the ascending chain of approximations

$$\tau(false) \subseteq \tau^2(false) \subseteq \ldots \subseteq \tau^k(false)$$

for the least $k \leq |S|$ such that $\tau^k(false) = \tau^{k+1}(false)$. The intuition here is just that each $\tau^i(false)$ corresponds to the set of states which can reach P within at most distance i; thus, P is reachable from state s iff P is reachable within i steps from s for some i less than the size of M iff $s \in \tau^i(false)$ for some such i less than the size of M. This idea can be easily generalized to provide a straightforward model checking algorithm for all of CTL and even the entire Mu-calculus. The Tarski-Knaster theorem handles the basic modalities. Compound formulae built up by nesting and boolean combinations are handled by recursive descent.

Iterative Fixpoint Algorithms. Building on this simple idea of iterative fixpoint calculation using the Tarski-Knaster Theorem, we can get a number of successively faster (global) model checking algorithms.

Naive Algorithm. We give below an algorithm to calculate, given structure M, formula f, and valuation $\mathcal{V}$, $set(f) = \{s : M, s \models f(\mathcal{V})\}$, the set of states in M where formula f is true under valuation $\mathcal{V}$.

A straightforward implementation runs in time complexity $O((|M||f|)^{k+1})$ for input structure M and input formula p with μ, ν formulas nested k deep.

Basic Algorithm. The naive algorithm can be significantly improved by utilizing the monotonicity among consecutive least fixpoints and consecutive greatest fixpoints, together with a simple generalization of the Tarski-Knaster theorem. The original algorithm for model checking in the Mu-calculus (cf. [EL86]) exploited this basic optimization.

Initialize all atomic proposition constants P and variables Y:

```
    $set(P) := \{s : M, s \models P\}$;
    $set(Y) := \mathcal{V}(Y)$;
```

Inductively calculate $set(f)$ using

```
recursive procedure $set(f)$
    case on the form of $f$:
        $f = P$ : return $set(f)$ unchanged;
        $f = Y$ : return $Y$ unchanged;
        $f = EXg$ : return $set(f) := \{s : \exists t \in S\ (s,t) \in R$ and $t \in set(g)\}$;
        $f = AXg$ : return $set(f) := \{s : \forall t \in S\ (s,t) \in R$ implies $M, t \in set(g)\}$;
        $f = g \wedge h$ : return $set(f) := set(g) \cap set(h)$;
        $f = g \vee h$ : return $set(f) := set(g) \cup set(h)$;
        $f = \neg g$ : return $set(f) := S \setminus set(g)$;
        $f = \mu Y.g(Y)$ : [ $Y := \mathit{false}$;
                          repeat
                              $Y' := Y$;
                              $Y := set(g(Y))$;
                          until $Y' = Y$;
                          return $set(f) := Y$ ]
        $f = \nu Y.g(Y)$ : [ $Y := \mathit{true}$;
                          repeat
                              $Y' := Y$;
                              $Y := set(g(Y))$;
                          until $Y' = Y$;
                          return $set(f) := Y$ ]
    endcase
```

Fig. 1. The Naive Algorithm

Theorem (Generalized Tarski-Knaster). Let $\tau : 2^S \to 2^S$ be a monotonic functional. Then

(a) $\mu Y.\tau(Y) = \cup_i\ \tau^i(Y_0)$ for any $Y_0 \subseteq \tau(Y_0) \cap \mu Y.\tau(Y)$, where i ranges over all ordinals of cardinality at most that of the state space S, so that when S is finite i ranges over $[0{:}|S|]$; $\mu Y.\tau(Y)$ is the union of the following ascending chain of approximations:
$Y_0 \subseteq \tau(Y_0) \subseteq \tau^2(Y_0) \ldots$, and

(b) $\nu Y.\tau(Y) = \cap_i\ \tau^i(Y_0)$ for any $Y_0 \supseteq \tau(Y_0) \cap \nu Y.\tau(Y)$, where i ranges over all ordinals of cardinality at most that of the state space S, so that when S is finite i ranges over $[0{:}|S|]$; $\nu Y.\tau(Y)$ is the intersection of the following descending chain of approximations: $Y_0 \supseteq \tau(Y_0) \supseteq \tau^2(Y_0) \ldots$.

Let us first consider a formula with alternating μ's and ν's. For example, $\nu Y.f(Y, \mu Z.g(Y, Z))$ can take $|M|$ iterations of Y. Initially, $Y^0 = true$. Each subsequent iteration $Y^{i+1} = f(Y^i, \mu Z.g(Y^i, Z))$ involves calculating an ascending chain of iterations of Z, starting with $Z = \mathit{false}$, of length up to $|M|$. Thus, in total, Z is iterated $|M|^2$ times.

Now, consider $\mu Y.f(Y, \mu Z.g(Y, Z))$ where there are consecutive nested μ's, but no alternation of μ's and ν's. It can be computed as above, resetting Z to *false* each time the outer Y changes. In the above case, this resetting is apparently essential because Y is shrinking while Z is growing. In this case, however, monotonicity ensures that both Y and Z are growing, so that it is unnecessary to reset Z to *false* when Z changes. Let

$Y^0 = \mathit{false}$

$Y^{i+1} = f(Y^i, \mu Z.g(Y^i, Z))$

$Z^{i,0} =$ the initial value of Z in the context of Y^i

$Z^{i,j+1} = g(Y^i, Z^j)$

We use Y^ω to denote the first $Y^i = Y^{i+1}$ where Y stabilizes, and similarly for $Z^{i,\omega}$.

The computation proceeds as follows. $Y^0, Z^{0,0} = \mathit{false}$ initially. To compute $Y^1 = f(Y^0, \mu Z.g(Y^0, Z))$, one computes $\mu Z.g(Y^0, Z)$ as the limit $Z^{0,\omega}$ of the ascending chain of approximations $\mathit{false} = Z^{0,0} \subset Z^{0,1} \subset \ldots \subset Z^{0,\omega} = Z^{0,\omega+1}$ and then applies f. To compute, $Y^2 = f(Y^1, \mu Z.g(Y^1, Z))$, one must compute $\mu Z.g(Y^1, Z)$. One way to do this entails simply computing an ascending chain of approximations $\mathit{false} = Z^{1,0} \subset Z^{,1} \subset \ldots \subset Z^{1,\omega} = Z^{1,\omega+1}$ and then applies f, having reinitialized Z to *false*. However, since $Y^0 \subseteq Y^1$, by monotonicity $Z^{0,\omega} = \mu Z.g(Y^0, Z) \subseteq \mu Z.g(Y^1, Z)$. Moreover, $Z^{0,\omega} = g(Y^0, Z^{0,\omega}) \subseteq g(Y^1, Z^{0,\omega})$. Hence, by the generalized Tarski-Knaster Theorem we are permitted to start the computation of $\mu Z.g(Y^1, Z)$ with $Z^{1,0} = Z^{0,\omega}$. In general, we have $\mu Z.g(Y^i, Z) \subseteq \mu Z.g(Y^{i+1}, Z)$ and $Z^{i,\omega} = g(Y^i, Z^{i,\omega}) \subseteq g(Y^{i+1}, Z^{i,\omega})$. so that we can take $Z^{i+1,0} = Z^{i,\omega}$. On this basis, we see that not only are at most $|M|$ iterations required for Y but also at most $|M|$ iterations for Z.

This algorithm can be straightforwardly implemented to run in time $O((|M| \cdot |f|)^{ad+1})$ and space $O(|M| \cdot |f|)$. This can be improved to $O((|M| \cdot |f|)^{ad})$ by avoiding redundant computation, computing the successive differences $Y^{i+1} \setminus Y^i$). [2]

Deluxe Algorithm. More extensive monotonicity considerations can be exploited (cf. [Lo+94]) for yet more improvement. The key idea is that in a formula f such as

$$\mu Y_1.\nu Z_1.\mu Y_2.\nu Z_2.\ldots.\mu Y_n.g(Y_1, Z_1, Y_2, Z_2, \ldots, Y_n)$$

of alternation depth $ad = 2n - 1$ the μ variables turn out to be monotonic with

[2] With better accounting we can obtain sharper multi-parameter bounds such as $O(|S|^{ad-1} \cdot |R| \cdot |f|)$, but these are adequate for our exposition.

respect to each other, and the innermost μ variable is monotonic in its various instantiations. For instance, in computing the formula $\mu Y_1.\nu Z_1.\mu Y_2.g(Y_1, Z_2, Y_2)$, with $ad = 3$, we have $\mu Y_2.g(Y_1^i, Z_1^{i,j}, Y_2) \subseteq \mu Y_2.g(Y_1^{i+1}, Z_1^{i+1,j}, Y_2)$ and we can avoid reinitializing Y_2 by taking $Y_2^{i+1,0} = Y_2^{i,\omega}$. What varies as the computation proceeds are the number of iterations of surrounding ν variables. For a fixed tuple of ν variable indices, the number of iterations of the innermost μ variable is $|M|$. The number of such tuples is about $|M|^{ad/2}$. So the dominant term in the complexity corresponds to about $|M|^{1+ad/2}$ iterations. This can be implemented to run in time at most $O((|M| \cdot |f|)^{2+ad/2})$. However, a careful examination reveals that an exponential number of intermediate results, roughly proportional to the number of tuples of ν indices must be stored. Thus the space complexity is also exponential.

While this algorithm is of theoretical interest, it should be noted that the time complexity is still $(|M| \cdot |f|)^{O(ad)}$ as is the basic algorithm above. Moreover, the exponential space complexity here does not compare favorably with the polynomial space complexity of the basic algorithm. Exponential space complexity is especially problematic, in practice, because it is usually the space complexity rather than the time complexity that is the limiting factor. For most applications, if the computation will not even fit within main computer memory, then performing it quickly is out of the question. For formulas of alternation depth 1 or 2, this algorithm thus yields no advantage. Since no practical example of a correctness property requiring alternation depth 3 or more is known, it is not clear that there is ever an actual situation where the algorithm could be helpful. Still, it provides mathematical insight into the nature of the Mu-calculus.

Other Algorithms. There are a variety of other types of (extensional) model checking algorithms which we will just briefly discuss. In contrast to the "bottom up" iterative approaches above, which are associated with global model checking, in which satisfaction of fixpoints radiates outward, we also have "top down" approaches which are associated with local model checking. For instance, given a specific state s_0 in structure M, if we wish to know whether $M, s_0 \models AFQ$, we perform a depth first search starting at s_0 keeping on the stack the sequence of states visited; if a cycle is detected without seeing Q then AFQ is *false*. Otherwise, the search will eventually return AFQ to be *true*.

A related approach is to form the product of $|M|$ with the syntax diagram of $|f|$ and view the result as a tree automaton. Then test the tree automaton for nonemptiness. Testing it for nonemptiness, can be done by model checking certain restricted formulas [EJS93] (cf. [BVW94]). The tree automaton approach captures the essence of the *boolean graph* approach (cf. [CS93], [An93]), since the boolean AND/OR graphs correspond to the transition diagrams of tree automata.

5.3 Complexity of Explicit State Model Checking

The complexity results we summarize here are for explicit state model checking. The best known algorithm corresponds to the deluxe algorithm above. It is exponential time in the worst case, but polynomial time for any bounded alternation depth. Since all practical correctness properties seem to be of alternation depth 1 or 2, we have a low order polynomial time algorithm as explained above.

Litchenstein and Pnueli advanced the following argument (cf. [LP85]): in practice, it is typically the structure size rather than the formula size that is the dominant factor in the complexity, because structures are usually extremely large while specifications are often rather short. Hence, it is highly desirable to have an algorithm whose complexity grows *linearly* in the structure size, while even exponential growth in the specification size may be tolerable. For CTL and $L\mu_1$, we can get algorithms of time and space complexity $O(|M| \cdot |f|)$ which is linear in the size of both inputs (cf. [CES86], [CS93]). For CTL* the problem is PSPACE-complete, and we can get a model checker of time complexity $O(|M| \cdot exp(|f|))$ [EL85] (cf. [LP85]).

Among significant unsolved problems with practical implications we thus have:

Open Question. Is there a model checking algorithm for $L\mu_2$ that runs in time linear in the structure size?

Finally, terms of complexity classes, we have this following result [EJS93] (cf. [BVW94], [Lo+94], [Va95]):

Proposition. Model checking in the Mu-calculus is in NP $\cap$ co-NP.

Proof idea. Given structure M and formula f guess an annotation of M with the subformulae of f true at each state; this annotation also provides a "rank" for each μ variable Y indicating how many times the associated μ-formula $\mu Y.g(Y)$, is unwound. These ranks correspond to the indices in the Tarski-Knaster sequence of approximations. Thus, we might have ranked μ variable Y^5, which is equivalent to $g(Y^4)$, at state s depending on Y^4, equivalent to $g(Y^3)$, at state t depending on Y^3 at state u and so forth; the "depending on" relation should be well-founded as $\mu Y.g(Y)$ is a *least* fixpoint and can only be unwound a a bounded (viz. $|M|$) times. In general the ranks will be tuples of natural numbers; each tuple is of length at most $|f|$ and each tuple component is a natural number of value at most $|M|$. The ranks are ordered lexicographically. After guessing the ranked, threaded annotation simply verify that it is well-founded. This shows membership in NP. Membership in co-NP follows from the fact that the Mu-calculus is trivially closed under complementation. $\square$

Open Question. Is there a polynomial model checking algorithm for the (entire) Mu-calculus (over extensionally represented structures)?

5.4 State Explosion

We emphasize that the above discussion focuses on *extensional* model checking, where it is assumed that the structure M including all of its nodes and arcs explicitly represented using data structures such as adjacency lists or adjacency matrices. An obvious limitation then is the combinatorial *state explosion* problem. Given a reactive system composed on n sequential processes running in parallel, its global state graph will be essentially the product of the individual local process state graphs. The number of global states thus grows exponentially in n. For particular systems it may happen that the final global state graph is of a tractable size, say a few hundred thousand states plus transitions. A number of practical systems can be modeled at a useful level of abstraction by state graphs of this size, and extensional model checking can be a helpful tool.

On the other hand, it can quickly become infeasible to represent the global state graph for large n. Even a banking network with 100 automatic teller machines each having just 10 local states, could yield a global state graph of astronomical size amounting to about 10^{100} states.

Plainly, for such astronomical size systems it is out of the question to perform model checking over them even using algorithms that run in time and space linear in the size of the state space. Various approaches to ameliorating state explosion are currently under investigations. One approach is to use *abstraction*. The basic idea here is to replace a large, detailed system M by a small, less detailed system M' where inessential information has been suppressed. If an appropriate correspondence between the large and small systems can be established, then correctness of the small system may be used to ensure correctness of the large system. For instance, suppose there is a homomorphism $h : M \longrightarrow M'$ such that s and $h(s)$ agree on atomic propositions in linear time formula f and such that if $s \to t$ is a transition in M then $h(s) \to h(t)$ is a transition in M'. We may then conclude that if there is a path satisfying $\neg f$ in M then there is an image path satisfying $\neg f$ in M'. Hence, if in the small system $M', h(s_0) \models Af$ then in the large system $M, s_0 \models Af$. Another approach is to represent transition relations and sets of states symbolically as decribed below.

5.5 Symbolic Approaches

A noteworthy advance has been the introduction of *symbolic* model checking techniques (cf. [McM92], [BCMDH90], [Pi90], [CM90]) which are – in practice – often able to succinctly represent and model check over state graphs of size 10^{100} states and even considerably larger. The basic algorithms used for symbolic model checking are the *same* as those used for extensional model checking, and are based on iterative calculation of (a representation of) the set of states where each temporal basic modality holds using fixpoint computation justified by the Tarski-Knaster Theorem. The key distinction is that the state graph of the Kripke structure and sets of states where formulae are true in it are represented in terms of a boolean characteristic function which is in turn represented by an (ordered) Binary Decision Diagram (BDD) (cf. [Br86]). These BDDs can in practice be extremely succinct. BDD-based model checkers have been remarkably effective and useful for debugging and verification of hardware circuits. For reasons not well understood, BDDs are often able to exploit the regularity that is readily apparent even to the human eye in many hardware designs. Because software typically lacks this regularity, BDD-based model checking seems much less helpful for software verification. We refer the reader to [McM92] for an extended account of the utility of BDDs in hardware verification.

It should be emphasized, however, that BDD based model checking methods, are, in worst case, still intractably inefficient. On the one hand, for some structures M of astronomical size there are small BDDs representing them, and this is exploited in applications as noted above. But for other structures M, sometimes those derived from applications such as software, the BDD representation is intractably large. Plainly, a counting argument shows that most structures do not have a small BDD representation. In any event, checking simple graph reachability in a structure M, e. g. $M, s_0 \models EFQ$ where M is represented by a BDD is PSPACE-complete (cf. [GW83], [Br86]). The disparity between theoretical, worst case results for symbolic checking and its surprisingly good performance in practice, has so far militated against the development of an associated complexity theory for this application.

5.6 Debugging versus Verification

Model checkers are a type of decision procedure and provide yes/no answers. It turns out that, in practice, model checkers are often used for debugging as well as verification. In industrial environments it seems that the capacity of a model checker to function as a debugger is perhaps better appreciated than their utility as a tool for verifying correctness.

Consider the empirical fact that most designs are initially wrong and must go through a sequence of corrections/refinements before a truly correct design is finally achieved. Suppose one aspect of correctness that we wish to check is that a simple invariance property of the form $AG\mathit{good}$ holds provided the system M is started in the obviously *good* initial state s_0. It seems quite likely that the invariance may in fact not hold of the initial faulty design due to conceptually minor but tricky errors in the fine details. Thus, during many iterations of the design process, we have that in fact $M, s_0 \models EF\neg \mathit{good}$.

It would be desirable to circumvent the global strategy of examining all of M to calculate the set $EF\neg \mathit{good}^M$ and then checking whether s_0 is a member of that set. If there does exist a $\neg\mathit{good}$ state reachable from s_0, once it is detected it is no longer necessary to continue the search examining M. This is the heuristic motivating local model checking algorithms. Many of them involve searching from the start state s_0 looking for confirming or refuting states or cycles; once found, the algorithm can terminate often "prematurely" having determined that the formula must be true or must be false at s_0 on the basis of the portion M examined during the limited search.

Of course, it may be that all states must be examined before finding a refutation to $AGgood$. Certainly, once a truly correct design is achieved, all states reachable from s_0 must be examined. But in many practical cases, a refutation may be found quickly after limited search.

We note in passing that some symbolic model checkers have been adapted to provide some sort of counter example facility for debugging.

6 Conclusion

Reactive systems are becoming increasingly important in our society. There is an undeniable and growing need to find effective methods of constructing correct reactive systems. One factor these systems have in common beyond their nondeterministic, ongoing, reactive nature is that they are highly complex, even though they are typically finite state. While it may be relatively easy to express informally and in general terms what such a system is supposed to do (e.g., provide an air traffic control system), it appears quite difficult to provide a formal specification of correct behavior and to prove that the implementation actually satisfies the specification. The Mu-calculus and associated temporal logics such as CTL provide a good handle on precisely stating just what behavior is to occur when, at a variety of levels of detail. The fully automated type of reasoning provided by model checking provides a convenient tool for both verifying correctness and for automatic debugging. Moreover, a number of interesting mathematical problems arise in connection with model checking in the Mu-calculus.

Acknowledgments This work was supported in part by NSF grant CCR-9415496 and by SRC contract 95-DP-388.

We thank Neil Immerman, Kedar Namjoshi, Richard Trefler and Moshe Vardi for most helpful comments.

References

[An93] Anderson, H. R., Verification of Temporal Properties of Concurrent Systems, Ph. D. Dissertation, Computer Science Department, Aarhus Univ., Denmark, June 1993.

[BVW94] Bernholtz, O., Vardi, M., and Wolper, P. An Automata-theoretic Approach to Branching Time Model Checking, Computer Aided Verification Conference (CAV94), pp. 142 -155, June 1994.

[Br96] Bradfield, J. The Mu-calculus Alternation Hierarchy is Strict, to appear in CONCUR96.

[BFHRR92] Bouajjani, A., Fernandez, J, Halbwichs, N., Raymond, P., and Ratel, C., Minimal State Graph Generation, *Science of Computer Programming*, 1992.

[BS92] Bradfield, J., and Stirling, C., "Local Model Checking for Infinite State Spaces", *Theor. Comp. Sci.*, vol. 96, pp. 157-174, 1992.

[BCD85] Browne, M., Clarke, E. M., and Dill, D. Checking the Correctness of sequential Circuits, Proc. 1985 IEEE Int.. Conf. Comput. Design, Port Chester, NY pp. 545-548

[BCDM86a] Browne, M., Clarke, E. M., and Dill, D., and Mishra, B., Automatic verification of sequential circuits using Temporal Logic, IEEE Trans. Comp. C-35(12), pp. 1035-1044, 1986

[Br86] Bryant, R., Graph-based algorithms for boolean function manipulation, IEEE Trans. on Computers, C-35(8), 1986.

[BCMDH90] Burch, J., Clarke, E., McMillan, M., Dill, D. and Hwang, L., Symbolic Model Checking:10^{20} States and Beyond, Proceedings of 5th Annual IEEE-CS Symposium on Logic in Computer Science, pp. 428 – 439, June 1990

[CE81] Clarke, E. M., and Emerson, E. A., Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic, Logics of Programs Workshop, IBM Yorktown Heights, New York, Springer LNCS no. 131., pp. 52-71, May 1981.

[CES86] Clarke, E. M., Emerson, E. A., and Sistla, A. P., Automatic Verification of Finite State Concurrent System Using Temporal Logic, 10th ACM Symp. on Principles of Prog. Lang., Jan. 83; journal version appears in *ACM Trans. on Prog. Lang. and Sys.*, vol. 8, no. 2, pp. 244-263, April 1986.

[CFJ93] Clarke, E. M., Filkorn, T., Jha, S. Exploiting Symmetry in Temporal Logic Model Checking, 5th International Conference on Computer Aided Verification, Crete, Greece, June 1993.

[CS93] Cleaveland, R. and Steffan, B., A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-calculus, Formal Methods in System Design, vol. 2, no. 2, pp. 121-148, April 1993.

[Cl93] Cleaveland, R., Analyzing Concurrent Systems using the Concurrency Workbench, Functional Programming, Concurrency, Simulation, and Automated Reasoning Springer LNCS no. 693, pp. 129-144, 1993.

[CM90] Coudert, O., and Madre, J. C., Verifying Temporal Properties of Sequential Machines without building their State Diagrams, Computer Aided Verification '90, E. M. Clarke and R. P. Kurshan, eds., DIMACS, Series, pp. 75-84, June 1990.

[DGG93] Dams, D., Grumberg, O., and Gerth, R., Generation of Reduced Models for checking fragments of CTL, CAV93, Springer LNCS no. 697, 1993.

[DC86] Dill, D. and Clarke, E.M., Automatic Verification of Asynchronous Circuits using Temporal Logic, IEEE Proc. 133, Pt. E 5, pp. 276-282, 1986.

[Em81] Emerson, E. A., Branching Time Temporal Logics and the Design of Correct Concurrent Programs, Ph. D. Dissertation, Division of Applied Sciences, Harvard University, August 1981.

[EC80] Emerson, E. A., and Clarke, E. M., Characterizing Correctness Properties of Parallel Programs as Fixpoints. Proc. 7th Int. Colloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science #85, Springer-Verlag, 1981.

[EH86] Emerson, E. A., and Halpern, J. Y., 'Sometimes' and 'Not Never' Revisited: On Branching versus Linear Time Temporal Logic, *JACM*, vol. 33, no. 1, pp. 151-178, Jan. 86.

[EJ91] Emerson, E. A., and Jutla, C. S. "Tree Automata, Mu-Calculus, and Determinacy", *Proc. 33rd IEEE Symp. on Found. of Comp Sci.*, 1991.

[EJS93] Emerson, E. A., Jutla, C. S., and Sistla, A. P., On Model Checking for Fragments of the Mu-calculus, Proc. of 5th Inter. Conf. on Computer Aided Verification, Elounda, Greece, Springer LNCS no. 697, pp. 385-396, 1993.

[EL86] Emerson, E. A., and Lei, C.-L., Efficient Model Checking in Fragments of the Mu-calculus, IEEE Symp. on Logic in Computer Science (LICS), Cambridge, Mass., June, 1986.

[EL85] Emerson, E. A., and Lei, C.-L.m Modalities for Model Checking: Branching Time Strikes Back, pp. 84-96, ACM POPL85; journal version appears in Sci. Comp. Prog. vol. 8, pp 275-306, 1987.

[ES93] Emerson, E. A., and Sistla, A. P., Symmetry and Model Checking, 5th International Conference on Computer Aided Verification, Crete, Greece, June 1993full version to appear in *Formal Methods in System Design.*

[ES83] Emerson, E. A. and Sistla, A. P., Deciding Branching Time Logic, Logics of Programs Workshop, Springer LNCS no. 164, pp. 176 – 192, 1983.

[Em90] Emerson, E. A., Temporal and Modal Logic, in Handbook of Theoretical Computer Science, vol. B, (J. van Leeuwen, ed.), Elsevier/North-Holland, 1991.

[GW83] Galperin, H. and Wigderson, A., Succinct Representation of Graphs, *Information and Control*, vol. 56, pp. 183–198, 1983.

[GS92] German, S. M. and Sistla, A. P. Reasoning about Systems with many Processes, Journal of the ACM, July 1992, Vol 39, No 3, pp 675-735.

[Ho78] Hoare, C. A. R., Communicating Sequential Processes, CACM, vol. 21, no. 8, pp. 666-676, 1978.

[JT89] Jard, C., and Thierron, J., On-line model checking for Finite Linear Time Specifications, International Workshop on Automatic Verification Methods for Finite State Systems, LNCS 407, 1989.

[Ko83] Kozen, D., Results on the Propositional Mu-Calculus, Theor. Comp. Sci., pp. 333-354, Dec. 83

[KT87] Kozen, D. and Tiuryn, J., Logics of Programs, in Handbook of Theoretical Computer Science, (J. van Leeuwen, ed.), Elsevier/North-Holland, 1991.

[Ku86] Kurshan, R. P., "Testing Containment of omega-regular Languages", Bell Labs Tech. Report 1121-861010-33 (1986); conference version in R. P.

Kurshan, "Reducibility in Analysis of Coordination", LNCIS 103 (1987) Springer-Verlag 19-39.

[Ku94] Kurshan, R. P., *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach* Princeton University Press, Princeton, New Jersey 1994.

[LR86] Ladner, R. and Reif, J. The Logic of Distributed Protocols, in Proc. of Conf. On Theor. Aspects of reasoning about Knowledge, ed. J Halpern, pp. 207-222, Los Altos, Cal., Morgan Kaufmann

[La80] Lamport, L., Sometimes is Sometimes "Not Never"—on the Temporal Logic of programs, 7th Annual ACM Symp. on Principles of Programming Languages, 1980, pp. 174-185.

[La83] Lamport, L., What Good is Temporal Logic?, Proc. IFIP, pp. 657-668, 1983.

[Le96] Lenzi, G., A Hierarchy Theorem for the Mu-calculus, to appear ICALP96.

[LP85] Litchtenstein, O., and Pnueli, A., Checking That Finite State Concurrent Programs Satisfy Their Linear Specifications, POPL85, pp. 97-107, Jan. 85.

[Lo+94] Long, D., Browne, A., Clarke, E., Jha, S., Marrero, W., An Improved Algorithm for the Evaluation of Fixpoint Expressions, Proc. of 6th Inter. Conf. on Computer Aided Verification, Stanford, Springer LNCS no. 818, June 1994.

[MP92] Manna, Z. and Pnueli, A., Temporal Logic of Reactive and Concurrent Systems: Specification, Springer-Verlag, 1992

[McM92] McMillan, K., Symbolic Model Checking: An Approach to the State Explosion Problem, Ph. D. Thesis, Carnegie-Mellon University, 1992.

[Niw84] Niwinski, D., unpublished manuscript.

[Niw88] Niwinski, D., Fixed Points versus Infinite Generation, Proc. 3rd IEEE Symp. on Logic in Computer Science, pp. 402–409, 1988.

[Pa70] Park, D., Fixpoint Induction and Proof of Program Semantics, in Machine Intelligence, eds. B. Meltzer and D. Michie, vol. 5, pp. 59-78, Edinburgh Univ. Press, Edinburgh, 1970.

[Pi90] Pixley, C., A Computational Theory and Implementation of Sequential Hardware Equivalence, CAV'90 DIMACS series, vol.3 (also DIMACS Tech. Report 90-31), June 1990.

[Pi92] Pixley, C., A Theory and Implementation of Sequential Hardware Equivalence, IEEE Transactions on Computer-Aided Design, pp. 1469–1478, vol. 11, no. 12, 1992.

[Pn77] Pnueli, A., The Temporal Logic of Programs, 18th annual IEEE-CS Symp. on Foundations of Computer Science, pp. 46-57, 1977.

[Pn81] Pnueli, A., The Temporal Semantics of Concurrent Programs, Theor. Comp. Sci., vol. 13, pp 45-60, 1981.

[Pr81] Pratt, V., A Decidable Mu-Calculus 22nd Annual IEEE-CS Symposium on Foundations of Computer Science, pp. 421 – 427, May 1981.

[QS82] Queille, J. P., and Sifakis, J., Specification and verification of concurrent programs in CESAR, Proc. 5th Int. Symp. Prog., Springer LNCS no. 137, pp. 195-220, 1982.

[deR76] de Roever, W. P., Recursive Program Schemes: Semantics and Proof Theory, Math. Centre Tracts no. 70, Amsterdam, 1976.

[SW89] Stirling, C., and Walker, D., Local Model Checking in the Mu-calculus, pp. 369–383, Springer LNCS no. 351, 1989.

[St93] Stirling, C., Modal and Temporal Logics. in Handbook of Logic in Computer Science, (D. Gabbay, ed.) Oxford, 1993

[Ta55] Tarksi, A., A Lattice-Theoretical Fixpoint Theorem and its Applications, Pacific. J. Math., 55, pp. 285-309, 1955.

[Va95] Vardi, M., On the Complexity of Bounded Variable Queries, Proceedings of the 14th ACM Sympsium on Principles of Database Symposium, June 1995.

[Va9?] Vardi, M., Why is Modal Logic So Decidable? This Volume, 199?

[VW83] Vardi, M. Y., and Wolper, P., Yet Another Process Logic, Logics of Programs Workshop, Springer LNCS no. 164., pp. 501 – 512, June 1983.

[VW84] Vardi, M. Y., and Wolper, P., Automata Theoretic Techniques for Modal Logics of Programs, STOC 84; journal version in *JCSS*, vol. 32, pp. 183-221, 1986.

[VW86] Vardi, M. Y., and Wolper, P. , An Automata-theoretic Approach to Automatic Program Verification, Proc. IEEE LICS, pp. 332-344, 1986.

This article was processed using the LaTeX macro package with LLNCS style