Symbolic Model Checking: 10^{20} States and Beyond

J. R. Burch E. M. Clarke K. L. McMillan
School of Computer Science
Carnegie Mellon University

D. L. Dill L. J. Hwang
Stanford University

Abstract

Many different methods have been devised for automatically verifying finite state systems by examining state-graph models of system behavior. These methods all depend on decision procedures that explicitly represent a state space, using a list or a table that grows in proportion to the number of states. We describe a general method that represents the state space *symbolically* instead of explicitly. The generality of our method comes from using a dialect of the Mu-Calculus as the primary specification language. We describe a *model checking* algorithm for Mu-Calculus formulas which uses Bryant's *Binary Decision Diagrams* [3] to represent relations and formulas symbolically. We then show how our new Mu-Calculus model checking algorithm can be used to derive efficient decision procedures for CTL model checking, satisfiability of linear-time temporal logic formulas, strong and weak observational equivalence of finite transition systems, and language containment for finite ω-automata. This eliminates the need to describe complicated graph-traversal or nested fixed point computations for each decision procedure. We illustrate the practicality of our approach to symbolic model checking by discussing how it can be used to verify a simple synchronous pipeline.

1 Introduction

Over the last decade, it has become apparent that finite-state systems can often be verified automatically by examining state-graph models of system behavior. A number of different methods have been proposed: temporal logic model checking, language containment algorithms for automata, "conformation checking" in trace theory, and testing for various equivalences and preorders between finite CCS-like models. Although each of these methods uses a different computational model and a different notion of verification, they all rely on decision algorithms that explicitly represent a state space, using a list or table that grows in proportion to the number of states. Because the number of states in the model may grow exponentially with the number concurrently executing components, the size of the state table is usually the limiting factor in applying these algorithms to realistic systems.

A recent idea for combating this "state explosion problem" is to represent the state space *symbolically* instead of explicitly. In many cases, the intuitive "complexity" of the state space is much less than the number of states would indicate. Often systems with a large number of components have a regular structure that would suggest a corresponding regularity in the state graph. Consequently, it may be possible to find more sophisticated representations of the state space that exploit this regularity in a way that a simple table of states cannot. One good candidate for such a symbolic representation is the *binary decision diagram* (BDD) [3], which is widely-used in various tools for the design and analysis of digital circuits. BDD's do not prevent a state explosion in all cases, but they allow many practical systems with extremely large state spaces to be verified — systems that would be impossible to handle with explicit state enumeration methods. Indeed, we present empirical results in this paper that show that the method can be applied in practice to verify models with in excess of 10^{20} states. Explicit state enumeration methods are limited to systems with at most 10^3 to 10^6 reachable states.

Several groups have applied this idea to different verification methods. Coudert, Berthet, and Madre

This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976. The National Science Foundation also sponsored this research effort under contract numbers CCR-8722633 and MIP-8858807. The third author is supported by an AT&T Bell Laboratories Ph.D. Scholarship. The fourth and fifth authors are supported by a CIS Seed Research Grant.

describe a BDD-based system for showing equivalence between deterministic Moore machines [8]. Their system performs a *symbolic breadth-first execution* of the state space determined by of the product of the two machines. This model is not generalized to models other than deterministic Moore machines, or notions of verification other than strict equivalence. Bose and Fisher [1] have described a BDD-based algorithm for CTL model checking that is applicable to synchronous circuits. However, their method is unable to handle asynchronous concurrency, or properties of infinite computations, such as liveness and fairness. Burch, Clarke, Dill, and McMillan [4] describe another BDD-based method for handling CTL model checking with *fairness constraints* and show this method can be applied to both synchronous and asynchronous examples.

All of these methods are based on iterative computation of fixed points. It seems clear that numerous additional papers could be generated by applying this technique to different verification methodologies. Our goal is to provide a unified framework for these results by showing that all can be seen as special cases of symbolic evaluation of Mu-Calculus formulas.

We describe the syntax and semantics of a dialect of the Mu-Calculus, and present a *model checking* algorithm for Mu-Calculus formulas that uses BDD's to represent relations and formulas. We then show how our new Mu-Calculus model checking algorithm can be used to derive efficient decision procedures for CTL model checking, satisfiability of linear-time temporal logic formulas, strong and weak observational equivalence of finite transition systems, and language containment for finite ω-automata. In each case, a Mu-Calculus formula can be directly derived from an instance of the problem. This formula can be evaluated automatically, eliminating the need to describe a complicated graph-traversal or nested fixed point computations for each decision procedure. We illustrate the practicality of our approach to symbolic model checking by discussing how it can be used to verify a simple synchronous pipeline.

2 Binary Decision Diagrams

Binary decision diagrams (BDD) are a canonical form representation for a boolean formulas. Bryant described algorithms for efficient manipulation of BDD's in [3]. BDD's are often substantially more compact than the traditional normal forms such as CNF and DNF, and hence have found application in symbolic verification of combinational logic, among other uses. A BDD is similar to a binary decision

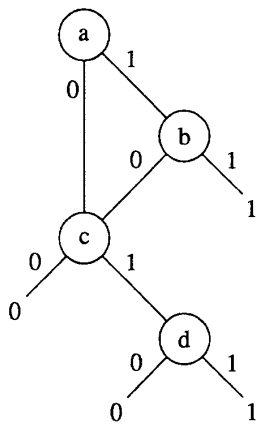

Figure 1: A Binary Decision Diagram

tree, except that its structure is a directed acyclic graph rather than a tree, and there is a strict total order placed on the occurrence of variables as one traverses the graph from root to leaf. Consider for example, the BDD of figure 1. It represents the formula $(a \wedge b) \vee (c \wedge d)$, using the variable ordering $a < b < c < d$. Given an assignment of Boolean values to the variables a, b, c and d, one can decide whether the assignment satisfies the formula by traversing the graph beginning at the root, branching at each node based on the assigned value of the variable which labels that node. For example, the assignment $\langle a \leftarrow 1, b \leftarrow 0, c \leftarrow 1, d \leftarrow 1 \rangle$ leads to a leaf node labeled 1, hence this assignment satisfies the formula.

In [3], Bryant shows that there is a unique minimal BDD for a given formula with a given variable ordering. Since BDD's (with some modifications) can be viewed as deterministic finite automata, this result is not surprising. The size of the minimal BDD depends critically on the variable ordering. Bryant also gives algorithms of low complexity for computing the BDD representations of $\neg f$ and $f \vee g$ given the BDD's for formulas f and g. The only other operations which we will require for the algorithms that follow are quantification over Boolean variables (QBF, see [13]) and substitution of variable names. Bryant gives an algorithm for computing the BDD for a restricted formula of the form $f|_{a=0}$ or $f|_{a=1}$. Though Bryant does not deal with QBF formulas, the restriction algorithm allows us to compute the BDD for the QBF formula $\exists v[f]$, where v is a Boolean variable and f is a QBF formula, as $f|_{a=0} \vee f|_{a=1}$. The substitution of a variable w for a variable v in a formula f, denoted

$f\langle v \leftarrow w\rangle$ can be accomplished using quantification, that is,

$$f\langle v \leftarrow w\rangle \stackrel{\text{def}}{=} \exists v[(v \Leftrightarrow w) \wedge f].$$

More efficient algorithms are possible, however, for the case of quantification over multiple variables, or multiple renamings. In the latter case, efficiency depends on the ordering of variables in the BDD's being the same on both sides of the substitution.

3 The Mu-Calculus

A number of different versions of the Mu-Calculus have been proposed. In this paper we use Park's notation [16]. It can be shown that this version of the Mu-Calculus can express any property expressible in the other versions of the Mu-Calculus found in [7,9,11,16,17].

We assume we are given a finite signature $\mathcal{S}$. Each symbol in $\mathcal{S}$ is either an *individual variable* or a predicate variable with some positive arity. There are two syntactic categories: *formulas* and *relational terms*. Formulas have following form:

1. **True**, **False**.
2. $[z_1 = z_2]$, where z_1 and z_2 are individual variables in $\mathcal{S}$.
3. $\neg f$, $f \vee g$, $\exists z[f]$, where f and g are formulas and z is an individual variable in $\mathcal{S}$.
4. $P(z_1, z_2, \ldots, z_n)$, where P is an n-ary relational term and $z_1, z_2, \ldots, z_n$ are individual variables in $\mathcal{S}$ not free in P.

The syntax for the *n-ary relational terms* is given below:

1. Z, where Z is an n-ary predicate variable in $\mathcal{S}$.
2. $\lambda z_1, z_2, \ldots, z_n[f]$, where f is a formula and $z_1, z_2, \ldots, z_n$ are distinct individual variables in $\mathcal{S}$.
3. $\mu Z[P]$, where Z is an n-ary predicate variable in $\mathcal{S}$ and P is an n-ary relational term that is formally monotone in Z.

The n-ary relational term $\mu Z[P]$ represents the *least fixed point* of an n-ary relational term P. A relational term P is *formally monotone* in the predicate variable Z if all free occurrences of Z in P fall under an even number of negations. The formal definition of when a variable or predicate symbol is *bound* or *free* in some formula or relational term is standard, and will not be given here. Note, however, that individual variables can be bound by both the existential quantifier $\exists$ and by the abstraction operator λ, while predicate variables can only be bound by the fixed point operator μ.

We will assume that $\forall$, $\wedge$, $\Rightarrow$, and $\Longleftrightarrow$ are treated as abbreviations in the usual manner. If P and P' are n-ary relational terms we write $\neg P$ as an abbreviation for $\lambda z_1, \ldots, z_n[\neg P(z_1, \ldots, z_n)]$, and we write $P \vee P'$ as an abbreviation for

$$\lambda z_1, \ldots, z_n[P(z_1, \ldots, z_n) \vee P'(z_1, \ldots, z_n)].$$

The term $\nu Z[P]$ is introduced as an abbreviation for

$$\neg\mu Z[\neg P\langle Z \leftarrow (\neg Z)\rangle]$$

and denotes the *greatest fixed point* of an n-ary relational term P, where $P\langle Z \leftarrow (\neg Z)\rangle$ denotes relational term formed from P by substituting $\neg Z$ for Z.

The truth or falsity of a formula is determined with respect to a *structure* $\mathcal{M} = (D, I_P, I_D)$ where D is a non-empty set called the *domain* of the structure, I_P is the *relational variable interpretation* and I_D is the *individual variable interpretation*. More specifically, for each individual variable y, $I_D(y)$ is a value in D, and for each n-ary relational symbol Z, $I_P(Z)$ is an n-ary relation on the set D. We let $\mathcal{I}_D$ be the set of possible individual variable interpretations. Let $\mathcal{I}_P$ be similarly defined.

The semantic function $\mathcal{D}$ maps formulas to elements of

$$(\mathcal{I}_P \rightarrow (\mathcal{I}_D \rightarrow \{true, false\}));$$

and n-ary relational terms to elements of

$$(\mathcal{I}_P \rightarrow (\mathcal{I}_D \rightarrow 2^{(D^n)}))$$

where $2^{(D^n)}$ denotes the power set of the set D^n. The semantic function $\mathcal{D}$ is defined inductively on the structure of formulas and relational terms. First, we define $\mathcal{D}$ on formulas. The semantics of the formulas **True** and **False** are defined in the obvious way. If z_1 and z_2 are individual variables, then

$$\mathcal{D}([z_1 = z_2])(I_P)(I_D) \stackrel{\text{def}}{=} (I_D(z_1) = I_D(z_2)).$$

If f and g are formulas, then

$$\mathcal{D}(\neg f)(I_P)(I_D) \stackrel{\text{def}}{=} \neg(\mathcal{D}(f)(I_P)(I_D))$$

$$\mathcal{D}(f \vee g)(I_P)(I_D) \stackrel{\text{def}}{=} \mathcal{D}(f)(I_P)(I_D) \vee \mathcal{D}(g)(I_P)(I_D)$$

$$\mathcal{D}(\exists z[f])(I_P)(I_D) \stackrel{\text{def}}{=} \exists e \in D\,.\,[\mathcal{D}(f)(I_P)(I_D\langle z \leftarrow e\rangle)].$$

If P is an n-ary relational term, then

$$\mathcal{D}(P(z_1,\ldots,z_n))(I_P)(I_D) \stackrel{\text{def}}{=} \langle I_D(z_1),\ldots,I_D(z_n)\rangle \in \mathcal{D}(P)(I_P)(I_D).$$

Next, we define $\mathcal{D}$ on relational terms. A relational variable Z has the expected meaning,

$$\mathcal{D}(Z)(I_P)(I_D) \stackrel{\text{def}}{=} I_P(Z).$$

The final two cases are given by

$$\mathcal{D}(\lambda z_1,\ldots,z_n[f])(I_P)(I_D) \stackrel{\text{def}}{=} \{\langle e_1,\ldots,e_n\rangle : \mathcal{D}(f)(I_P)(I_D\langle z_1 \leftarrow e_1,\ldots z_n \leftarrow e_n\rangle)\}$$

$$\mathcal{D}(\mu Z[P]) \stackrel{\text{def}}{=} \text{lfp } \lambda Q \in 2^{(D^n)}.[\mathcal{D}(P)(I_P\langle Z \leftarrow Q\rangle)(I_D)].$$

where lfp denotes the least fixed point over the inclusion ordering. It is clear from elementary fixed point theory that the requirement that P be formally monotone in Z is sufficient to ensure the existence of the fixed point in the final definition above.

If $\mathcal{M}$ is a structure and f is a formula, then we will write $\mathcal{M} \models f$ to indicate that f is true in $\mathcal{M}$ according to the above semantics. In this paper the domain of a structure will always be finite.

4 Model Checking Algorithm

Model checking means determining whether a given formula f is valid in a given model M. In this section, we present a model checking algorithm for the Mu-Calculus which uses BDD's as its internal representation, in order to avoid enumerating the elements of the relations in the model. The algorithm is limited to the domain $D = \{0,1\}$. In other words, all individual variables are Boolean, and all relations are Boolean relations. Later we will show that a model in any finite domain can be encoded as a model in the Boolean domain, hence our model checking algorithm is fully general.

The algorithm is divided into two functions, BDD_f and BDD_R, which recurse over the structure of the formula. We assume here that the syntactic correctness of the formula has been checked, in particular the formal monotonicity requirement given in Section 3. The function BDD_f takes two arguments: a formula f and a relational variable interpretation I_P, which assigns values to the free relational variables in f. It returns a BDD which has the following property: $\text{BDD}_f(f, I_P)$ is satisfied by a given interpretation I_D for the individual variables if and only if f is satisfied by the model M that has interpretations I_P and I_D.

The value of each relational variable in the interpretation I_P is represented by a BDD, using a set of place-holder (dummy) variables not in the signature of the logic. We refer to these variables as $d_1, d_2, \ldots$, where d_i is used to stand for the i^{th} argument of a relation. Thus, an n-ary relation represented by a BDD is said to hold for some arguments $y_1,\ldots,y_n$ if and only if the interpretation $\langle d_1 \leftarrow y_1,\ldots,d_n \leftarrow y_n\rangle$ satisfies the BDD. In many practical instances, this representation of a relation is much more compact than an enumeration of its elements.

The function BDD_f is defined in Figure 2. The first four cases in the definition derive directly from the respective semantic definitions for BDD's and Mu-Calculus formulas and should require no explanation. For the implementation of BDDATOM, BDDAND and BDDNEGATE, see [3]. The last case, application of a relational term R, uses the function BDD_R to find a representation of the relational term R (under the interpretation I_P), then substitutes the argument variables $x_1,\ldots,x_n$ for the place-holder variables $d_1,\ldots,d_n$, producing a BDD which is satisfied if and only if R holds for $\langle x_1,\ldots,x_n\rangle$.

The function BDD_R (see Figure 2) takes as arguments a relational term R and a relational interpretation I_P. It returns a BDD which represents the the relational term in the manner described above. Since the relational term may have free individual variables, the BDD may contain both the place-holder variables and the individual variables of the logic. Thus, given an interpretation I_D for the individual variables, and an interpretation I_A for the place-holder variables, $\text{BDD}_R(R, I_P)$ is satisfied if and only if the relation $\mathcal{D}(R)(I_P)(I_D)$ holds for the n-tuple $\langle I_A(d_1),\ldots,I_A(d_n)\rangle$, where n is the arity of R.

The first case in the definition of BDD_R, a relation variable, simply returns the BDD representation of the variable in the interpretation I_P. The second case, lambda abstraction, produces a BDD with place-holder variables $d_1,\ldots,d_n$ substituted for the variables $x_1,\ldots,x_n$. This is the representation for an n-ary relation which holds if and only if its arguments satisfy the formula f when assigned to $x_1,\ldots,x_n$. The most interesting case is the last: the fixed point operator μ. To find the fixed point of a relational term with respect to a free relational variable Z, we use the standard technique for finding the least fixed point of a monotonic functional in a finite domain. This computes the fixed point by a series of approximations $T_0, T_1, \ldots$, beginning with the empty relation (which is represented by the BDD constant FALSEBDD). To compute T_{i+1}, we let the interpretation of Z be T_i, while evaluating the relational term R using BDD_R. Since the domain is finite and R is formally monotone

```
function BDD_f(f : formula, I_P : rel-interp) : BDD;
    case
        f an individual variable:
            return BDDATOM(f);
        f of the form f_1 ∧ f_2:
            return BDDAND(BDD_f(f_1, I_P), BDD_f(f_2, I_P));
        f of the form ¬f_1:
            return BDDNEGATE(BDD_f(f_1, I_P));
        f of the form ∃x[f_1]:
            return BDDEXISTS(x, BDD_f(f, I_P));
        f of the form Z(x_1, ..., x_n):
            return BDD_R(Z, I_P)⟨d_1 ← x_1⟩ ··· ⟨d_n ← x_n⟩;
    end case;

function BDD_R(R : rel-term, I_P : rel-interp) : BDD;
    case
        R a relational variable:
            return I_P(R);
        R of the form λx_1, ..., x_n[f]:
            return BDD_f(f, I_P)⟨x_1 ← d_1⟩ ··· ⟨x_n ← d_n⟩;
        R of the form μZ[R_1]:
            return FIXEDPOINT(Z, R_1, I_P, FALSEBDD);
    end case;

function FIXEDPOINT(Z : rel-var, R : rel-term, I_P : rel-interp, T_i : BDD) : BDD;
    let T_{i+1} = BDD_R(R, I_P⟨Z ← T_i⟩);
    if T_{i+1} = T_i return T_i
    else return FIXEDPOINT(Z, R, I_P, T_{i+1});
```

Figure 2: Mu-Calculus Model Checking Algorithm.

in Z, the series must converge to the least fixed point. Convergence is detected when $T_{i+1} = T_i$. Note that testing for convergence is easy, since testing BDD's for equivalence is a constant time operation.

A performance improvement can be realized in the above fixed point algorithm by observing that any subterms or subformulas of R which do not have Z as a free variable will not change in their evaluation from one iteration to the next. The evaluations of these terms can therefore be cached and do not need to be recomputed. For this reason, it is useful when possible to rewrite formulas so that fixed point subterms contain fewer free relational variables.

In order to do model checking over a non-Boolean (but finite) domain D, we use an encoding function $\phi : \{0,1\}^m \rightarrow D$ which maps a each Boolean vector of length m onto an element of D. This function may be many-to-one, but must be onto. Note that the minimum value of m is $\lceil \log_2 |D| \rceil$, but encodings with a larger number of bits are also possible. Using this encoding, we construct a corresponding model M' over the Boolean domain. If R is an n-ary relation symbol in the model M, then R' is a relation of arity mn in M', constructed by the following rule:

$$R'(\bar{x}_1, \ldots, \bar{x}_n) \Leftrightarrow R(\phi(\bar{x}_1), \ldots, \phi(\bar{x}_n))$$

where $\bar{x}_i$ is a shorthand for m boolean variables encoding x_i. In order to check the validity of a given formula f, we then replace each individual variable in the formula with a vector of m boolean valued variables, and check the resulting formula f' in the model M'. The homomorphism between M and M' guarantees that $M \models f$ if and only if $M' \models f'$.

The choice of an encoding function ϕ, and of the ordering of the variables in the BDD's has a substantial impact on the efficiency of the model checking. In a later section, we describe a of digital circuit design who's specification can be checked in time polynomial in the number of circuit components, for the right choice of variable ordering. In the area of digital

circuits, the choice of encodings is generally trivial, since all components of the state are Boolean valued to begin with.

5 Iterative Squaring

It is often possible to rewrite a Mu-Calculus formula in order to obtain an equivalent formula that can be analyzed more efficiently by the model checking algorithm. In this section we describe a systematic method for rewriting formulas, called the *iterative squaring* transformation, that can result in an exponential reduction in the number of iterations necessary to compute fixed points in the Mu-Calculus. We begin by showing how the iterative squaring transformation can be applied to a particular formula. Later we describe more general conditions under which the transformation can be applied.

5.1 Transitive Closure

Let (V, E) be a directed graph, and let V_0 be some subset of V. The set V_* of vertices reachable from V_0 is expressed by the Mu-Calculus relational term R given by

$$\mu Q[\lambda y[V_0(y) \vee \exists x[Q(x) \wedge E(x, y)]]].$$

When the model checking algorithm is applied to R, it requires n iterations to compute the set V_n of vertices reachable in n transitions. Thus, the number of iterations is linear in the diameter of the subgraph (V_*, E). However, the iterative squaring transformation can be used rewrite R so that the model checking algorithm converges faster. The first step is to compute the transitive closure E_* of E,

$$\mu Z[\lambda x, y[E(x, y) \vee \exists w[Z(x, w) \wedge Z(w, y)]]].$$

Let E_n be the binary relation computed by the model checker after n iterations in the computation of E_*.

Theorem 1 *For all vertices y and non-negative integers n,*

$$\exists x[V_0(x) \wedge E_{n+1}(x, y)] \iff V_{2^n}(y).$$

Thus, the number of iterations necessary to compute E_* is logarithmic in the diameter of (V, E). If the diameters of (V, E) and (V_*, E) are roughly the same (the usual case in practice), this leads to a significant reduction in the number of iterations needed to compute V_*. However, iterative squaring can be impractical if the BDD's needed to represent the intermediate computations become too large.

5.2 General Transformation

We consider r-ary relational terms of the form $\mu Q[R]$ or $\nu Q[R]$, where R is some r-ary relational term. We further restrict R to be of the form (using $\bar{y}$ as a shorthand for $y_1, \ldots, y_r$),

$$\lambda \bar{y}[S(\bar{y}) \vee \exists \bar{x}[Q(\bar{x}) \wedge N(\bar{x}, \bar{y})]]. \qquad (1)$$

where S and N are relational terms that do not have Q as a free variable. It may seem overly restrictive to require that terms be of the form (1). However, nearly all the Mu-Calculus terms that we have used as specifications in practice can be written in this form. Define the relational term T such that

$$T = \mu Z[\lambda \bar{x}, \bar{y}[N(\bar{x}, \bar{y}) \vee \exists \bar{w}[Z(\bar{x}, \bar{w}) \wedge Z(\bar{w}, \bar{y})]]],$$

which is the transitive closure of N. The general form of the iterative squaring transformation is given by the following theorem.

Theorem 2 *Let T be as defined above, and let R be a relational term of the form (1). Then $\mu Q[R]$ can be transformed into the equivalent term*

$$\lambda \bar{y}[S(\bar{y}) \vee \exists \bar{x}[S(\bar{x}) \wedge T(\bar{x}, \bar{y})]].$$

Also, $\nu Q[R]$ can be transformed into the equivalent term

$$\lambda \bar{y}[S(\bar{y}) \vee \exists \bar{x}[(S(\bar{x}) \vee T(\bar{x}, \bar{x})) \wedge T(\bar{x}, \bar{y})]].$$

The iterative squaring transformation can often be applied more than once to terms that have several fixed point operators. For example, consider the relational term

$$\nu L[B \wedge \mu Q[\lambda x[\exists y[(L(y) \vee Q(y)) \wedge N(x, y)]]]],$$

where B, L and N are relational variables. This relational term is used in CTL model checking with fairness constraints [4]. Using Theorem 2, the reader can check that the above relational term is equivalent to

$$\lambda x[B(x) \wedge \exists y[B(y) \wedge T(y, y) \wedge T(x, y)]],$$

for T as defined above. The reduction from two levels of fixed point operators to one level is possible because the transitive closure of T is just T.

Unless otherwise noted, all the Mu-Calculus relational terms used in the remainder of this paper can be computed using the iterative squaring transformation.

6 CTL

CTL or Computation Tree Logic [6] is a propositional, branching-time, temporal logic. Each of the usual forward-time operators of linear temporal logic (**G** *globally* or *invariantly*, **F** *sometime in the future*, **X** *nexttime* and **U** *until*) must be directly preceded by a *path quantifier*. The path quantifier can either be an **A** (for all computation paths) or an **E** (for some computation path). Thus, some typical CTL operators are $\mathbf{AG}f$, which will hold in a state provided that f holds at all points (globally) along all possible computation paths starting from that state, and $\mathbf{EF}f$, which will hold in a state provided that there is a computation path such that f holds at some point in the future on the path.

In our description of the syntax and semantics of CTL, we specify the existential path quantifiers directly and treat the universal path quantifiers as syntactic abbreviations. Let P be the set of atomic propositions, then:

1. Every atomic proposition p in P is a formula in CTL.
2. If f and g are CTL formulas, then so are $\neg f$, $f \wedge g$, $\mathbf{EX}f$, $\mathbf{E}[f\mathbf{U}g]$ and $\mathbf{EG}f$.

The semantics of a CTL formula is defined with respect to a labeled state transition graph or *Kripke structure* $\mathcal{M} = (P, S, L, N, S_0)$ where P is a set of atomic propositions, S is a finite set of states, $L : S \rightarrow 2^P$ is a function labeling each state with a set of atomic propositions, $N \subseteq S \times S$ is a total transition relation, and S_0 is the set of initial states. A *path* is an infinite sequence of states $s_0, s_1, s_2, \ldots$ such that $N(s_i, s_{i+1})$ is true for every i.

The propositional connectives $\neg$ and $\wedge$ have their usual meanings of negation and conjunction. The other propositional operators can be defined in terms of these. **X** is the *nexttime* operator. $\mathbf{EX}f$ will be true in a state s of $\mathcal{M}$ if and only if s has a successor t such that f is true at t. **U** is the *until* operator. $\mathbf{E}[f\mathbf{U}g]$ will be true in a state s of $\mathcal{M}$ if and only if there exists a computation path starting in s and an initial prefix of the path such that g holds at the last state of the prefix and f holds at all other states along the prefix. The operator **G** is used to express the *invariance* of some property over time. $\mathbf{EG}f$ will be true at a state s if there is a path starting at s such that f holds at each state on the path.

6.1 Representing Kripke Structures

Using the Mu-Calculus algorithm for CTL model checking is quite straightforward. In fact, given an appropriate domain and interpretation, the CTL operators $\mathbf{EX}f$, $\mathbf{EG}f$ and $\mathbf{E}[f\mathbf{U}g]$ can be viewed as syntactic abbreviations for Mu-Calculus relational terms. If the CTL formula f is an abbreviation for the Mu-Calculus relational term R, then f is true at state s if and only if $R(s)$ is true. Let the domain D be the set of states S of the Kripke structure, and the interpretation I_P consist of the transition relation N and one unary relation for each atomic proposition $p \in P$, such that $I_P(p)(s)$ is true if and only if $p \in L(s)$. We assume that no two distinct states have the same labeling. There is no loss of generality in this assumption, since extra atomic propositions can be added to P without affecting the truth of CTL formulas.

The CTL formula $\mathbf{EX}f$ is true of a state s if and only if there exists a state t such that f is true of t and $N(s,t)$ is true. We therefore define $\mathbf{EX}f$ to be a syntactic abbreviation for the Mu-Calculus relational term

$$\lambda u[\exists\, v[f(v) \wedge N(u,v)]].$$

The Mu-Calculus expansions for **EG** and **EU** are based on a characterization of the CTL operators as fixed points of predicate transformers. The fixed points can be computed using either direct iteration or iterative squaring.

The fixed point characterization for **EG** is given by

$$\begin{aligned} \mathbf{EG}f &= \nu Q[f \wedge \mathbf{EX}Q] \\ &= \nu Q[\lambda u[f(u) \wedge \exists v[Q(v) \wedge N(u,v)]]]. \end{aligned}$$

The intuition should be clear: For every state s, there is a path starting at s along which f holds globally if and only if f holds at s and s has a successor t such that there is a path starting at t along which f holds globally.

The operator **EU** has a fixed point characterization that is similar to the one for **EG**. However, this time the characterization is the *least fixed point* of the corresponding predicate transformer rather than the greatest.

$$\mathbf{E}[f\mathbf{U}g] = \mu Q[g \vee (f \wedge \mathbf{EX}Q)]$$

which is equivalent to

$$\mu Q[\lambda u[g(u) \vee (f(u) \wedge \exists v[Q(v) \wedge N(u,v)])]].$$

Starting at some state s, there will be a path such that $f\mathbf{U}g$ holds if and only if g holds at s or f holds at s and s has a successor t such that there is a path starting at t along which $f\mathbf{U}g$ holds.

In [4] we show how this method can also be applied to CTL with *fairness constraints*, and that iterative squaring can also be used in this case.

7 PTL

The tableau method for testing the satisfiability of propositional linear temporal logic (PTL) formulas [12] can be implemented by translating a PTL formula into a Mu-Calculus formula which is true if and only if the PTL formula is satisfiable. This gives a symbolic procedure with the advantage that, in some cases, a large tableau can be represented by a relatively small BDD.

Fujita and Fujisawa [10] describe a verification procedure based on linear temporal logic that uses binary decision diagrams to represent the transition conditions in automata derived from temporal logic formulas. However, their technique still suffers from a form of the state explosion problem. This is a result of their representing states explicitly in automata derived from temporal formulas.

There are many dialects of PTL depending on the modal connectives that are defined. We choose a small, generic dialect.

1. *atomic propositions* AP (written p, q, etc.),
2. $\neg f$, $f \vee g$, $\mathbf{X}f$, and $f\mathbf{U}g$ when f and g are PTL formulas.

Our technique can easily extended to additional or alternative modal connectives.

As in CTL, $\mathbf{X}f$ means that f holds in the next state and $f\mathbf{U}g$ means that f is true in every state until g holds. To define this formally, let $\sigma \in [AP \to \{0,1\}]^\omega$ be a sequence of truth assignments to the atomic propositions, and let σ_i be the ith suffix of σ ($\sigma_i(j) = \sigma(j+i)$ for all $j \in \omega$).

$$
\begin{array}{lll}
\sigma \models p & \text{iff} & \sigma(0)(p) = 1 \quad \text{when } p \in AP, \\
\sigma \models \neg f & \text{iff} & \sigma \not\models f, \\
\sigma \models f \vee g & \text{iff} & \sigma \models f \text{ or } \sigma \models g, \\
\sigma \models \mathbf{X}f & \text{iff} & \sigma_1 \models f, \\
\sigma \models f\mathbf{U}g & \text{iff} & \exists i : (\sigma_i \models g \text{ and } \forall j < i : \sigma_j \models f).
\end{array}
$$

The tableau procedure for PTL transforms a PTL formula f into a Kripke structure that encodes a set of paths. Every state is labeled with a set of subformulas of f. The tableau is constructed by iterating a two-step process: The first step takes a set of PTL formulas and breaks them down into a set of states. The states are labeled with elementary formulas: propositional formulas characterizing what is true "now", and $\mathbf{X}$-formulas that determine the possible successor states. The second step removes the $\mathbf{X}$ from each $\mathbf{X}$-formula in the state; this gives a set of formulas to which the first step can be applied, yielding the successors to the state. An explicit tableau construction unwinds the tableau from the initial formula until all reachable states have been found.

The tableau construction here is somewhat different from the usual one, so as to reduce the number of different kinds of state labels (which become propositional variables).

The set of elementary formulas associated with a PTL formula can be defined recursively (f and g are any PTL formulas):

$$
\begin{aligned}
el(p) &= \{p\} \quad \text{when } p \in AP, \\
el(\neg f) &= el(f), \\
el(f \vee g) &= el(f) \cup el(g), \\
el(\mathbf{X}f) &= \{\mathbf{X}f\} \cup el(f), \\
el(f\mathbf{U}g) &= \{\mathbf{X}(f\mathbf{U}g)\} \cup el(f) \cup el(g).
\end{aligned}
$$

The set of states of the tableau generated from the formula f is $2^{el(f)}$. Intuitively, each state corresponds to the set of formulas that may hold at a some time in some model. If a formula does not appear in a state, the negation of the formula holds.

We use $2^{el(f)}$ as the domain D over which Mu-Calculus formulas are to be interpreted. The domain D is encoded as all the bit vectors of length $|el(f)|$. For every formula g in $el(f)$, there is a unary relation $P(g)$. Read $P(g)(\boldsymbol{x})$ as "g appears in state $\boldsymbol{x}$". The interpretation of $P(g)$ is simply the set of bit-vectors that have a 1 bit in the position associated with g.

The first step of the iterative tableau construction, breaking a formula into a set of states, is captured in a recursive transformation α that maps a PTL formula f and the name of an individual variable $\boldsymbol{x}$ to a formula representing a set of states.

$$
\begin{aligned}
\alpha(p)(\boldsymbol{x}) &= P(p)(\boldsymbol{x}) \quad p \in AP, \\
\alpha(\neg f)(\boldsymbol{x}) &= \neg P(f)(\boldsymbol{x}), \\
\alpha(f \vee g)(\boldsymbol{x}) &= \alpha(f)(\boldsymbol{x}) \vee \alpha(g)(\boldsymbol{x}), \\
\alpha(\mathbf{X}f)(\boldsymbol{x}) &= P(\mathbf{X}f)(\boldsymbol{x}), \\
\alpha(f\mathbf{U}g)(\boldsymbol{x}) &= \alpha(g)(\boldsymbol{x}) \vee \\
&\quad [\alpha(f)(\boldsymbol{x}) \wedge P(\mathbf{X}(f\mathbf{U}g))(\boldsymbol{x})].
\end{aligned}
$$

To find the set of successors of a particular state, we remove the $\mathbf{X}$ operators from in front of the next-time formulas in the state and then apply α. The resulting formula holds for a state if and only if the state is an immediate successor of the original state. The next state relation can be written as a propositional formula over two sets of variables:

$$
N(\boldsymbol{u}, \boldsymbol{v}) = \bigwedge_{\mathbf{X}g \in el(f)} P(\mathbf{X}g)(\boldsymbol{u}) \Leftrightarrow \alpha(g)(\boldsymbol{v}).
$$

In the special case where f contains no next-time formulas, $N(u,v)$ is identically true.

The tableau has a distinguished set of *initial states* which give the temporal conditions that must be satisfied by a model σ. We define a formula representing this set: $S_0(u) = \alpha(f)(u)$.

An infinite sequence ρ of states of the tableau is called a *path* if $S_0[\rho(0)]$ and $N[\rho(i), \rho(i+1)]$ for all $i \in \omega$ and

A formula f is satisfiable if and only if there exists a path satisfying the additional property that whenever $\rho(i)$ contains a formula of the form $\mathbf{X}(g\mathbf{U}h)$, there is some $j > i$ such that ρ_j satisfies h.

By the definition of N, when $\mathbf{X}(g\mathbf{U}h)$ appears in $\rho(i)$, it either appears also in $\rho(i+1)$ or $\rho(i+1)$ satisfies $\alpha(h)$. So this condition is a fairness constraint on *consecutive pairs* of states in ρ: infinitely often, $\rho(i)$ is *not* labeled with $\mathbf{X}(g\mathbf{U}h)$ or $\rho(i+1)$ satisfies $\alpha(h)$.

For states u and v to satisfy such fairness constraint, they must satisfy

$$[P(\mathbf{X}(g\mathbf{U}h))(u) \Rightarrow \alpha(h)(v)] \wedge N(u,v),$$

which we abbreviate $F(\mathbf{X}(g\mathbf{U}h))(u,v)$. Let $N_*(v,w)$ express "there is a finite path from v to w". Thus N_* is the transitive closure of N. Expressing transitive closure in the Mu-Calculus was discussed in Section 5. The set of states belonging to a fair cycle is given by the relational term *Fair* defined as

$$\begin{aligned}&\lambda u[\exists u_1, v_1, \ldots, u_k, v_k \\ &\quad [F(\mathbf{X}(g_1\mathbf{U}h_1))(u_1,v_1) \wedge N_*(v_1,u_2) \wedge \ldots \wedge \\ &\quad F(\mathbf{X}(g_k\mathbf{U}h_k))(u_k,v_k) \wedge N_*(v_k,u) \wedge N_*(u,u_1)]],\end{aligned}$$

assuming there are k formulas $\mathbf{X}(g_i\mathbf{U}h_i)$ in $el(f)$. Therefore, the formula is satisfiable only if

$$\exists u, v[S_0(u) \wedge N_*(u,v) \wedge Fair(v)]$$

is true in the appropriate model.

8 Observational Equivalences

In this section, we describe BDD based algorithms for deciding strong and weak equivalence of finite transition systems (FTS's). In [15,14], a finite transition system is defined as a 4-tuple $\langle K, p_0, \Sigma, \Delta\rangle$ where K is a finite set of *states*, p_0 is the *initial state*, Σ is a finite set of *actions*, and $\Delta \subseteq K \times \Sigma \times K$ is the *transition relation*. We use $p_i \xrightarrow{\sigma} p_f$ to denote $(p_i, \sigma, p_f) \in \Delta$.

8.1 Strong Equivalence

Let P and Q be two finite transition systems on the same actions Σ. That is, let $P = \langle K_p, p_0, \Sigma, \Delta_p\rangle$ and $Q = \langle K_q, q_0, \Sigma, \Delta_q\rangle$. Strong equivalence is a relation $\sim \subseteq K_p \times K_q$. The two finite transitions systems P and Q are said to be strongly equivalent if and only if $p_0 \sim q_0$. In [14], it is shown that $\sim$ is the greatest fixed point of the function $F : 2^{K_p \times K_q} \rightarrow 2^{K_p \times K_q}$ where $F(R)$ is the set of all pairs (p,q) such that

- $\forall \sigma, p'$: if $p \xrightarrow{\sigma} p'$ then $\exists q' : q \xrightarrow{\sigma} q'$ and $R(p',q')$,
- $\forall \sigma, q'$: if $q \xrightarrow{\sigma} q'$ then $\exists p' : p \xrightarrow{\sigma} p'$ and $R(p',q')$.

In order to compute this equivalence using the BDD-based Mu-Calculus checking algorithm, it remains only to assemble the appropriate domain and interpretations, and to express the above condition in the Mu-Calculus. It is useful in this case to think of the domain D as having three sorts: K_p, K_q and Σ. The relational interpretation I_P consists of the relations Δ_p and Δ_q, and the individual interpretation I_D consists of p_0 and q_0, symbols for the two initial states. The function F is given by the Mu-Calculus relational term

$$\begin{aligned}&\lambda p, q[\forall \sigma, p'[\Delta_p(p,\sigma,p') \Rightarrow \\ &\quad \exists q'[\Delta_q(q,\sigma,q') \wedge R(p',q')]] \\ &\wedge \forall \sigma, q'[\Delta_q(q,\sigma,q') \Rightarrow \\ &\quad \exists p'[\Delta_p(p,\sigma,p') \wedge R(p',q')]]].\end{aligned}$$

The two FTS's P and Q are thus strongly equivalent if and only if $\nu R[F](p_0,q_0)$ holds. This can be evaluated using the BDD-based model checking algorithm, though the iterative squaring transformation can not be applied.

8.2 Weak Equivalence

Let τ be a distinguished action in the set Σ, and let the relation $\Delta^{\tau *}$ be the reflexive transitive closure of $\lambda x, y[\Delta(x,\tau,y)]$. That is, $\Delta^{\tau *}(p,q)$ is true if and only if there is a path from p to q labeled by a sequence of zero or more τ actions. Further, for every $\sigma \in \Sigma$, let $\Delta^{\tau * \sigma}$ be the relational composition of $\Delta^{\tau *}$ and Δ,

$$\Delta^{\tau * \sigma} \equiv \lambda x, z[\exists y[\Delta^{\tau *}(x,y) \wedge \Delta(y,\sigma,z)]].$$

That is, $\Delta^{\tau * \sigma}(p,q)$ is true if and only if there is a path from p to q labeled by a sequence of zero or more τ actions followed by a single action σ. Weak observational equivalence ($\approx$) is the greatest fixed point of the function $G : 2^{K_p \times K_q} \rightarrow 2^{K_p \times K_q}$, where $G(R)$ is the set of all pairs (p,q) such that

- $\forall p'$: if $\Delta^{\tau *}(p,p')$ then $\exists q' : \Delta^{\tau *}(q,q')$ and $R(p',q')$,
- $\forall q'$: if $\Delta^{\tau *}(q,q')$ then $\exists p' : \Delta^{\tau *}(p,p')$ and $R(p',q')$,

- $\forall p', \sigma$: if $\Delta^{\tau * \sigma}(p, p')$ then $\exists q' : \Delta^{\tau * \sigma}(q, q1)$ and $R(p', q')$,
- $\forall q', \sigma$: if $\Delta^{\tau * \sigma}(q, q')$ then $\exists p' : \Delta^{\tau * \sigma}(p, p')$ and $R(p', q')$.

From here, it is a straightforward exercise (which we omit) to translate the definition of G into the Mu-Calculus. The two FTS's P and Q are weakly equivalent if and only if $\nu R[G](p_0, q_0)$. This can be evaluated using the BDD-based model checking algorithm, though the iterative squaring transformation can not be applied.

9 ω-Automata

Finally, we discuss symbolic Mu-Calculus based algorithms for deciding language containment between finite ω-automata. Although there are many types of ω-automata, we will consider here only the simplest case, that of deterministic finite Buchi automata. Algorithms for other types of automata can be derived in a similar fashion from results in [5].

A finite Buchi automaton is an ordered 5-tuple $(K, p_0, \Sigma, \Delta, A)$, where K is a finite set of *states*, $p_0 \in K$ is the *initial state*, Σ is a finite *alphabet*, $\Delta \subseteq K \times \Sigma \times K$ is the *transition relation*, and $A \subseteq K$ is the *acceptance set*. The automaton is *deterministic* if for all $p \in K$ and $\sigma \in \Sigma$, there exists at most one distinct $q \in K$ such that $\Delta(p, \sigma, q)$ holds. An infinite sequence of states $p_0, p_1, p_2, \ldots \in K^\omega$ is a *path* of a Buchi automaton if there exists an infinite sequence $a_0, a_1, a_2, \ldots \in \Sigma^\omega$ such that

$$\forall i \geq 0 : \langle s_i, a_i, s_{i+1} \rangle \in \Delta.$$

A sequence $a_0, a_1, a_2, \ldots$ is *accepted* by a Buchi automaton if the corresponding path $p_0, p_1, p_2, \ldots$ goes through a one or more members of A infinitely often. The set of sequences accepted by an automaton M is called the language of M and denoted $\mathcal{L}(M)$.

Let M and M' be two Buchi automata over the same alphabet Σ. Let $K(M, M')$ be a Kripke structure $(AP, K \times K', \langle p_0, p_0' \rangle, L, R)$, where

- $AP = \{\alpha, \alpha'\}$ is the set of atomic propositions
- $\langle s, s' \rangle \models \alpha$ *iff* $s \in A$
- $\langle s, s' \rangle \models \alpha'$ *iff* $s' \in A'$
- $\langle s, s' \rangle R \langle r, r' \rangle$ *iff* $\exists a \in \Sigma : \langle s, a, r \rangle \in \Delta$ and $\langle s', a, r \rangle \in \Delta'$.

Recall that in Section 6 we showed how to encode Kripke structures symbolically. In [5], it is shown that, if M is deterministic,

$$\mathcal{L}(M) \subseteq \mathcal{L}(M') \Leftrightarrow K(M, M') \models \mathbf{A}(\mathbf{GF}\alpha \Rightarrow \mathbf{GF}\alpha')$$

Note that the above is not a CTL formula. However, it belongs to a class of formulas which can be evaluated in polynomial time using fixed point algorithms. In fact, $\mathbf{A}(\mathbf{GF}\alpha \Rightarrow \mathbf{GF}\alpha')$ is equivalent to $\mathbf{AGAF}\alpha'$ under the fairness constraint "infinitely often α". Checking the above formula with the given fairness constraint can be reduced to checking the truth of a particular Mu-Calculus formula, using the techniques described in Section 4 and [4].

10 Empirical Results

The BDD method for testing boolean satisfiability is only efficient in a heuristic sense. The problem is, of course, NP-complete in general; the only claim that is made for BDD's is that they perform well for certain useful classes of boolean functions. Likewise, the BDD method for representing state sets in the CTL model checking problem is only of heuristic value, and does not improve the asymptotic complexity of model checking. Therefore, in order to evaluate the method, we need empirical results showing the performance of the method for some problems of practical interest. Here we present briefly some performance results for CTL model checking on a class of simple synchronous pipelines, which include data path as well as control circuitry. The number of states in these systems is far too large to apply traditional model checking techniques, but we have obtained very encouraging results using the BDD method.

The circuits we have used as examples of this category are very simple pipelines that perform three-address logical and arithmetic operations on a register file. The complete state of the register file and pipe registers are modeled. The pipelines have three stages. In the first stage, the operands are read from the register file, in the second stage an ALU operation is performed, and in the third stage the result is written back to the register file. The ALU has a register bypass path, which allows the result of an ALU operation to be used immediately as an operand on the next clock cycle, as is typical in RISC instruction pipelines. The inputs to the circuits are an instruction code, containing the register addresses of the source and destination operands, and a STALL signal, which indicates that the instruction stream is stalled. When this occurs, a "no-operation" is propagated through the pipe.

A discussion of the CTL specification of the synchronous pipelines can be found in [4]. Here we discuss only the performance results. Table 1 summarizes the results we obtained in verifying a variety of pipelines of this type. We varied the number of bits

per register, and the type of operation(s) performed by the ALU, to see how these affected the size of the BDD used to represent the transition relation, the total execution time required to check the specification, and the total storage used. The most complex pipeline we verified had approximately 5×10^{20}, states, which puts it far outside the range of model checkers like the one reported in [2]. It required a BDD with 42000 nodes to represent the transition relation, and approximately 22 minutes on a Sun 3 workstation to verify. The most interesting result is that the number of nodes in the transition relation BDD increases *linearly* in the number of bits per register, while the running time increases roughly quadratically. This is somewhat surprising on it's face, since the number of states increases exponentially in the number of bits, as does the complexity of boolean satisfiability (as far as we know). On consideration, however, this is perhaps not very surprising. Intuitively, the complexity of the BDD is a function of how much information must be remembered as one passes from one level of the BDD to the next (i.e., from one variable to the next). In the pipeline examples, the information stored from one "bit slice" of the data path to the next is rather small; it amounts to the state of the control bits plus at most the value of the ALU "carry" bit. In particular, this amount of information does not increase as one increases the number of bits, so the BDD becomes deeper, but no "wider". This result lends some support to the notion that the complexity of model checking and other state enumeration based verification techniques can in fact be reduced in practice by using an efficient representation of the state space rather than enumerating it explicitly.

ALU ops	width (bits)	register file size	BDD nodes	Verification time (secs)
$\oplus$	1	4	2737	9
$\oplus$	2	4	8430	46
$\oplus$	3	4	14123	145
$\oplus$	4	4	19816	306
$\oplus$	8	4	41000	1349
$+$	1	4	2737	9
$+$	2	4	10734	45
$+$	3	4	22276	179
$+$	4	4	33818	492
$+$	8	4	79986	3709
$+, \oplus$	2	4	18429	188
$+, \oplus$	3	4	36239	690
$+, \oplus$	4	4	53924	1706

Table 1: Performance of BDD model checking algorithm on simple pipelines.

11 Conclusions

We have shown, that by choosing a suitable encoding of the model domain, and using a compact representation for relations, the complexity of various graph-based verification algorithms can be greatly reduced in practice (if not in the worst case). Along the way, we have shown how several of these algorithms can be concisely expressed in a form of the Mu-Calculus, and how these expressions can be used to derive efficient BDD-based verification algorithms. In the circuit examples we studied, the regular structure of the data path logic was captured by the BDD representation, resulting in a space complexity which was linear in the number of circuit components rather than exponential.

The current state of this research, however, leaves open several important and interesting questions. First, more work is needed in order to characterize the models for which the BDD Mu-Calculus checker is efficient. It is known, for example, that combinational multiplier circuits do not have efficient BDD representations [3]. On the other hand, the model checking algorithm is easily adapted to use other representations, if such are found to be compact for a useful class of relations. The problem of finding more efficient structures for representing Boolean formulas has attracted much attention of late; any results obtained in this area would be immediately applicable to Mu-Calculus model checking, and hence to the various verification methodologies treated in this paper.

The second open question is whether the techniques described here could be profitably extended to other common graph algorithms whose results can be expressed as relations, such as minimum spanning trees, graph isomorphism, etc. For example, if $E(u, v)$ is the edge relation of a directed graph, then the equivalence relation

$$\lambda u, v[E'(u, v) \wedge E'(v, u)]$$

is true of two vertices if and only if they are in the same strongly connected component, where E' is a relational term representing the reflexive transitive closure of E. Practical algorithms that could handle very large graphs (compared to current computer storage limitations) would certainly be of interest.

References

[1] S. Bose and A. Fisher. Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic. In *IMEC-IFIP International Workshop on Applied Formal Methods For Correct VLSI Design*, 1989.

[2] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Trans. Comput.*, C-35(12):1035–1044, 1986.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, C-35(8), 1986.

[4] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *ACM/IEEE Design Automation Conference*, June 1990. To Appear.

[5] E. M. Clarke, I. A. Draghicescu, and R. P. Kurshan. A unified approach for showing language containment and equivalence between various types of ω-automata. In *Proceedings of the Fifteenth Colloquium on Trees in Algebra and Programming*, 1990. To Appear.

[6] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Prog. Lang. Syst.*, 8(2):244–263, 1986.

[7] R. Cleaveland. *Tableau-Based Model Checking in the Propositional Mu-Calculus*. Technical Report 2/89, University of Sussex, March 1989.

[8] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France*, Springer-Verlag, June 1989.

[9] E. A. Emerson and C. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the First Annual Symposium on Logic in Computer Science*, 1986.

[10] M. Fujita and H. Fujisawa. Specification, verification, and synthesis on control circuits with propositional temporal logic. In *Ninth International Symposium on Computer Hardware Description Languages and their Applications*, North-Holland, June 1989.

[11] D. Kozen. Results on the propositional mu-calculus. *Theoretical Comput. Sci.*, 333–354, Dec. 1983.

[12] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. In *Proceedings of the Workshop on Logic of Programs*, 1981.

[13] A. R. Meyer and L. J. Stockmeyer. Word problems requiring exponential time. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, 1973.

[14] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Comput. Sci.*, 25:267–310, 1983.

[15] R. Milner. *A Calculus of Communicating Systems*. Volume 92 of *Lecture Notes in Computer Science*, Springer-Verlag, 1980.

[16] D. Park. *Finiteness is Mu-Ineffable*. Theory of Computation Report No. 3, The University of Warwick, 1974.

[17] C. Stirling and D. J. Walker. Local model checking in the modal mu-calculus. In *TAPSOFT*, 1989.