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 the 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 that uses Bryant's *Binary Decision Diagrams* (1986) 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. The fixed point computations for each decision procedure are sometimes complex, but can be concisely expressed in the Mu-Calculus. We illustrate the practicality of our approach to symbolic model checking by discussing how it can be used to verify a simple synchronous pipeline circuit.

*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.

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 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.

Our technique 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) (Bryant, 1986), which is widely used in various tools for the design and analysis of digital circuits. BDDs 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 described in the literature are limited to systems with at most 10^8 reachable states.

Several groups have applied this idea to different verification methods. Coudert, Berthet, and Madre (1989) describe a BDD-based system for showing equivalence between deterministic Moore machines. 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 (1989) 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.

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.

Another technique for reducing the state explosion problem is to exploit concurrency. Two actions x and y (e.g., program statements) are said to be concurrent if executing xy is equivalent to executing yx. By considering only one order of the concurrent actions, or considering the actions to be unordered, the state explosion can be reduced. Examples of such techniques are the stubborn sets method of Valmari (1989; 1990), the trace automaton method of Godefroid and Wolper (Godefroid, 1990; Godefroid and Wolper, 1991), the behavior machines method of Probst and Li (1990), and the Time Petri Nets method of Yoneda *et al.* (1989). These methods are limited in that they only address one source of the state explosion problem—the interleaving of concurrent actions. They are not effective, for example, on synchronous finite state machines, which do not involve interleaving of actions. The symbolic model checking technique, on the other hand, can be effective in dealing with the state explosion in the synchronous case, as demonstrated in section 10. Symbolic methods have also been shown to be effective for asynchronous finite state machines (Burch et al., 1990; Burch et al., 1991b). In practice, much of the state explosion that results from interleaving can be handled efficiently by symbolic methods.

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 BDDs 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 complicated 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 circuit.

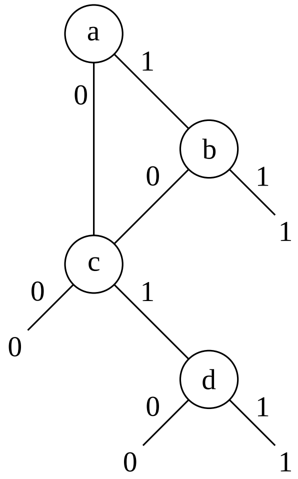

Figure 1: A Binary Decision Diagram

2 Binary Decision Diagrams

Binary decision diagrams (BDDs) are a canonical form representation for Boolean formulas (Bryant, 1986). They are often substantially more compact than traditional representations such as conjunctive normal form and disjunctive normal form. Hence, BDDs have found application in many computer aided design tasks, including symbolic verification of combinational logic. A BDD is similar to a binary decision tree, except that its structure is a directed acyclic graph rather than a tree, and there is a total order placed on the occurrence of variables as one traverses the graph from root to a 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.

Bryant showed that there is a unique BDD for a given Boolean function together with a given variable ordering. The size of the BDD representing a given function depends critically on the variable ordering. Bryant also described efficient algorithms for basic operations on BDDs, such as computing the BDD representations of $\neg f$ and $f \vee g$ given the BDDs for formulas f and g. The only other operations required for the algorithms that follow are quantification over Boolean variables and substitution of variable names. Bryant gives an algorithm for computing the BDD for a restricted formula of the form $f|_{a=0}$ or $f|_{a=1}$. The restriction algorithm allows us to compute the BDD for the formula $\exists v[f]$, where v is a Boolean variable and f is a 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,

$$f\langle v \leftarrow w\rangle = \exists v[(v \Leftrightarrow w) \wedge f].$$

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 BDDs being the same on both sides of the substitution.

BDDs can also be viewed as a form of deterministic finite automata (Kimura and Clarke, 1990). An n-argument Boolean function can be identified with the set of strings in $\{0, 1\}^n$ that represent valuations where the function is true. Since this is a finite language and all finite languages are regular, there is a minimal finite automaton that accepts this set. This automaton provides a canonical representation for the original Boolean function. Logical operations on Boolean functions can be implemented by set operations on the languages accepted by the finite automata. For example, conjunction corresponds to language intersection. Standard constructions from elementary automata theory can be used to compute these operations on languages.

3 The Mu-Calculus

A number of different versions of the Mu-Calculus have been proposed. In this paper we use the notation of Park (1974). It can be shown that this version of the Mu-Calculus can express any property expressible in other versions of the Mu-Calculus (Cleaveland, 1989; Emerson and Lei, 1986; Kozen, 1983; Stirling and Walker, 1989).

The Mu-Calculus is similar to standard first-order logic, with the following changes. First, as a simplifying assumption, we do not include function symbols or constant symbols. Also, relational symbols are replaced by relational variables. In formulas of the form $R(z_1, z_2, \ldots, z_n)$, the R can be a relational variable (analogous to a relational symbol in first-order logic), or it can be a *relational term* in one of two other forms. The first of the these forms is

$$\lambda y_1, y_2, \ldots, y_n[f],$$

where f is a formula and the y_i are individual variables. Most often the y_i are free in f, but this need not be the case. Also, the free variables of f need not be contained in the set of y_i. The other form for a relational term is $\mu P[R]$, where R is a relational term with some arity n and P is a relational variable, also with arity n. The term $\mu P[R]$ represents the least fixed point of R. To insure that the least fixed point exists, we require that R be *formally monotone* with respect to P, which means that all free occurrences of P in R fall under an even number of negations.

As an example, let (V, E) be a directed graph, and let V_0 and Q be subsets of V. The Mu-Calculus formula

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

is true if and only if the vertex y is in V_0 or is reachable in one step from a vertex in Q. The Mu-Calculus relational term

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

represents the smallest set Q such that

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

This is the set V_* of vertices reachable from V_0.

The above description of the syntax of the Mu-Calculus can be formalized as follows. We assume we are given a finite signature $\mathcal{S}$. Each symbol in $\mathcal{S}$ is either an *individual variable* or a *relational variable* with some positive arity. We recursively define two syntactic categories: *formulas* and *relational terms*. Formulas have the following form:

1. $R(z_1, z_2, \ldots, z_n)$, where R is an n-ary relational term and $z_1, z_2, \ldots, z_n$ are individual variables in $\mathcal{S}$ not free in R.

2. $\neg f$, $f \vee g$, $\exists z[f]$, where f and g are formulas and z is an individual variable in $\mathcal{S}$.

Also, relational terms of arity n have the following form:

1. P, where P is an n-ary relational 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. $\mu P[R]$, where P is an n-ary relational variable in $\mathcal{S}$ and R is an n-ary relational term that is formally monotone with respect to P.

The formal definition of when an individual variable or relational variable 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 relational 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 R and R' are n-ary relational terms we write $\neg R$ as an abbreviation for $\lambda z_1, \ldots, z_n[\neg R(z_1, \ldots, z_n)]$, and we write $R \vee R'$ as an abbreviation for

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

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

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

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

The truth or falsity of a formula is determined with respect to a *model* $\mathcal{M} = (D, I_R, I_D)$ where D is a non-empty set called the *domain* of the model, I_R 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 variable P, $I_R(P)$ is an n-ary relation on the set D. In this paper, the domain of a model will always be finite. For a given domain, let $\mathcal{I}_D$ and $\mathcal{I}_R$ be the set of all possible individual variable interpretations and the set of all possible relational variable interpretations, respectively.

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

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

and n-ary relational terms to elements of

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

where $2^{(D^n)}$ denotes the set of n-ary relations on D. The semantic function $\mathcal{D}$ is defined inductively on the structure of formulas and relational terms. First, we define $\mathcal{D}$ on formulas. If R is an n-ary relational term, then

$$\mathcal{D}(R(z_1, \ldots, z_n))(I_R)(I_D)$$

is true if and only if

$$\langle I_D(z_1), \ldots, I_D(z_n)\rangle \in \mathcal{D}(R)(I_R)(I_D).$$

If f and g are formulas, then

$$\begin{array}{l} \mathcal{D}(\neg f)(I_R)(I_D) = \neg(\mathcal{D}(f)(I_R)(I_D)) \\ \mathcal{D}(f \vee g)(I_R)(I_D) = \\ \quad \mathcal{D}(f)(I_R)(I_D) \vee \mathcal{D}(g)(I_R)(I_D) \\ \mathcal{D}(\exists z[f])(I_R)(I_D) = \\ \quad \exists e \in D\, [\mathcal{D}(f)(I_R)(I_D\langle z \leftarrow e\rangle)]. \end{array}$$

Next, we define $\mathcal{D}$ on relational terms. The first two cases are given by

$$\begin{array}{l} \mathcal{D}(P)(I_R)(I_D) = I_R(P), \\ \mathcal{D}(\lambda z_1, \ldots, z_n[f])(I_R)(I_D) = \\ \quad \{\langle e_1, \ldots, e_n\rangle \in D^n : \mathcal{D}(f)(I_R)(I_D\langle z_1 \leftarrow e_1, \ldots, z_n \leftarrow e_n\rangle)\}. \end{array}$$

Finally,

$$\mathcal{D}(\mu P[R]) = Z,$$

where Z is the subset of D^n that is the least fixed point (under the inclusion ordering) of the equation

$$Z = \mathcal{D}(R)(I_R\langle P \leftarrow Z\rangle)(I_D).$$

It is clear from elementary fixed point theory that the least fixed point exists, since R is formally monotone with respect to P.

If $\mathcal{M}$ is a model 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.

4 Model Checking Algorithm

Model checking is the process of determining whether a given formula f is true in a given model M. In this section, we present a model checking algorithm for the Mu-Calculus that uses BDDs as its internal representation. First, we describe the algorithm for the Boolean domain $D = \{0, 1\}$. Later we show that a model with any finite domain can be encoded as a model with the Boolean domain, hence our model checking algorithm is fully general.

The algorithm is divided into two functions, $\textsc{Bdd}_f$ and $\textsc{Bdd}_R$, which recurse over the structure of formulas and relational terms, respectively (Figure 2). We assume here that the syntactic correctness of the formula has already been checked, including the formal monotonicity requirement.

The value of each relational variable in a relational interpretation I_R is represented by a BDD, using a set of place-holder (dummy) variables not in the signature $\mathcal{S}$. We refer to these variables as $d_1, d_2, \ldots$, where d_i is used to stand for the ith argument of a relation. Thus, an n-ary relation represented by a BDD is said to hold for some arguments $x_1, \ldots, x_n$ if and only if the interpretation $\langle d_1 \leftarrow x_1, \ldots, d_n \leftarrow x_n \rangle$ satisfies the BDD. In many practical instances, this representation of a relation is much more compact than an enumeration of its elements.

The function $\textsc{Bdd}_f$ takes two arguments: a formula f and a relational variable interpretation I_R, which assigns values to the free relational variables in f. It returns a BDD which has the following property: $\textsc{Bdd}_f(f, I_R)$ is satisfied by a given interpretation I_D for the individual variables if and only if f is satisfied by the model $M = (D, I_R, I_D)$. The first case in the definition treats individual variables as formulas, which is possible because the domain D is Boolean. The function $\textsc{BddAtom}(v)$ returns a BDD that is true if and only if $v = 1$. The next three cases in the definition derive directly from the respective semantic definitions for BDDs and Mu-Calculus formulas and should require no explanation. The algorithms for $\textsc{BddAnd}$ and $\textsc{BddNegate}$ were described by Bryant (1986). The implementation of $\textsc{BddExists}$ in terms of disjunction and restriction was discussed in Section 2. The last case, application of a relational term R, uses the function $\textsc{Bdd}_R$ to find a representation of the relational term R (under the interpretation I_R), then substitutes the argument variables $x_1, \ldots, x_n$ for the place-holder variables $d_1, \ldots, d_n$.

The function $\textsc{Bdd}_R$ takes as arguments a relational term R and a relational interpretation I_R. It returns a BDD which represents 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, $\textsc{Bdd}_R(R, I_R)$ is satisfied if and only if the relation $\mathcal{D}(R)(I_R)(I_D)$

```
function $\textsc{Bdd}_f(f : \text{formula},\ I_R : \text{rel-interp}) : \text{BDD}$;
    case
        $f$ is an individual variable:
            return $\textsc{BddAtom}(f)$;
        $f$ is of the form $f_1 \wedge f_2$:
            return $\textsc{BddAnd}(\textsc{Bdd}_f(f_1, I_R),\ \textsc{Bdd}_f(f_2, I_R))$;
        $f$ is of the form $\neg f_1$:
            return $\textsc{BddNegate}(\textsc{Bdd}_f(f_1, I_R))$;
        $f$ is of the form $\exists x[f_1]$:
            return $\textsc{BddExists}(x, \textsc{Bdd}_f(f, I_R))$;
        $f$ is of the form $R(x_1, \ldots, x_n)$:
            return $\textsc{Bdd}_R(R, I_R)\langle d_1 \leftarrow x_1, \ldots, d_n \leftarrow x_n\rangle$;
    end case;

function $\textsc{Bdd}_R(R : \text{rel-term},\ I_R : \text{rel-interp}) : \text{BDD}$;
    case
        $R$ is a relational variable:
            return $I_R(R)$;
        $R$ is of the form $\lambda x_1, \ldots, x_n[f]$:
            return $\textsc{Bdd}_f(f, I_R)\langle x_1 \leftarrow d_1, \ldots, x_n \leftarrow d_n\rangle$;
        $R$ is of the form $\mu P[R']$:
            return $\textsc{FixedPoint}(P, R', I_R, \textsc{FalseBdd})$;
    end case;

function $\textsc{FixedPoint}(P : \text{rel-var},\ R : \text{rel-term},\ I_R : \text{rel-interp},$
            $Z : \text{BDD}) : \text{BDD}$;
    let $Z' = \textsc{Bdd}_R(R, I_R\langle P \leftarrow Z\rangle)$;
    if $Z' = Z$ then return $Z$
    else return $\textsc{FixedPoint}(P, R, I_R, Z')$;
```

Figure 2: Mu-Calculus Model Checking Algorithm.

contains 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 $\textsc{Bdd}_R$, a relational variable, simply returns the BDD representation of the variable in the interpretation I_R. The second case, lambda abstraction, produces a BDD with place-holder variables $d_1, \ldots, d_n$ substituted for the variables $x_1, \ldots, x_n$. The most interesting case involves the fixed point operator μ. To find the fixed point of a relational term with respect to a relational variable P, we use the standard technique for finding the least fixed point of a monotonic function with a finite domain. This computes the fixed point by a series of approximations $Z_0, Z_1, \ldots$, beginning with the empty relation (which is represented by the BDD constant $\textsc{FalseBdd}$). To compute Z_{i+1}, we let the interpretation of P be Z_i, while evaluating the relational term R using $\textsc{Bdd}_R$. Since the domain is finite and R is formally monotone with respect to P, the series must converge to the least fixed point. Convergence is detected when $Z_{i+1} = Z_i$. Note that testing for convergence is easy, since testing BDDs 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 P as a free variable will not change in their evaluation from one iteration to the next. Thus, the evaluations of these terms 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 each Boolean vector of length m to an element of D. This function must be surjective, but it need not be injective. The minimum possible value of m is $\lceil \log_2 |D| \rceil$, but encodings with a larger number of bits are also possible. Using such an 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 truth of a given formula f, we 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 an ordering for the BDD variables has a substantial impact on the efficiency of the model checking algorithm. For digital circuits, the choice of encoding 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 or relational term so that it can be analyzed more efficiently by the model checking algorithm. In this section we describe a systematic method for rewriting relational terms, called the *iterative squaring* transformation, that can result in an exponential reduction in the number of iterations necessary to compute fixed points. We begin by showing how the iterative squaring transformation can be applied to a particular relational term. Later we describe more general conditions under which the transformation can be applied.

5.1 Transitive Closure

Let W be the relational term

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

which describes the set V_* of vertices reachable in the directed graph (V, E) from the set of vertices V_0 (see section 3). When the model checking algorithm is applied to W, it requires n iterations to compute the set V_n of vertices reachable via a path of length n or shorter. Thus, the number of iterations is linear in the diameter of the subgraph (V_*, E'), where E' is the set of edges in E connecting only vertices in V_*. However, a standard technique can be used to rewrite W so that the model checking algorithm converges faster. The first step is to compute the transitive closure of E,

$$E_* = \mu P[\lambda x, y[E(x,y) \vee \exists w[P(x,w) \wedge P(w,y)]]].$$

Let E_n be the binary relation computed by the model checker after n iterations in the computation of E_*. The following theorem can be proved by induction on n.

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

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

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 BDDs 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 $y_1, \ldots, y_r$),

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

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 this form. However, nearly all the Mu-Calculus terms that we have used as specifications in practice can be written in this form.

The relational term $\mu Q[R]$ is analogous to the relational term W described above. Recall that W represented the set V_*, which is the set of vertices reachable from V_0 in the graph (V, E). The analogy is clear if we let V be the set of r-tuples over the domain D, let E be N, and let V_0 be S. Under this analogy, $\mu Q[R]$ represents V_*, the set of vertices reachable from S via N.

We can re-express $\mu Q[R]$ in terms of the transitive closure of N. This allows us to use iterative squaring to compute the least fixed point. Define the relational term T such that

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

which is the transitive closure of N. The set of vertices reachable from S via N can be expressed as

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

This observation provides the intuition behind the proof of the following theorem.

Theorem 2 $\mu Q[R] = \lambda \bar{y}[S(\bar{y}) \vee \exists \bar{x}[S(\bar{x}) \wedge T(\bar{x}, \bar{y})]]$.

There is a straightforward relationship between the least and greatest fixed points. We claim that $\bar{y}$ is in $\nu Q[R]$ if and only if $\bar{y}$ is in $\mu Q[R]$ or $\bar{y}$ can be reached from some $\bar{x}$ that is on a cycle in the graph of N. The formula $T(\bar{x}, \bar{x})$ is true if and only if $\bar{x}$ is on a cycle. Assuming that the domain D is finite, we have the following theorem:

Theorem 3 $\nu Q[R] = \mu Q[R] \vee \lambda \bar{y}[\exists \bar{x}[T(\bar{x}, \bar{x}) \wedge T(\bar{x}, \bar{y})]]$

Proof. Let

$$\begin{aligned} Z &= \mu Q[R] \vee \lambda \bar{y}[\exists \bar{x}[T(\bar{x}, \bar{x}) \wedge T(\bar{x}, \bar{y})]] \\ &= \lambda \bar{y}[S(\bar{y}) \vee \exists \bar{x}[S(\bar{x}) \wedge T(\bar{x}, \bar{y})] \vee \exists \bar{x}[T(\bar{x}, \bar{x}) \wedge T(\bar{x}, \bar{y})]] \end{aligned}$$

It is straightforward (but tedious) to show that Z is a fixed point of $R(Q)$, so we omit this argument. It remains to show that Z is the greatest fixed point, that is, if $Q = R(Q)$, then $Q \subseteq Z$. Suppose that $\bar{x}_0$ is an element of Q. It follows that $\bar{x}_0$ is an element of $R(Q)$, hence $S(\bar{x}_0) \vee \exists \bar{x}[Q(\bar{x}) \wedge N(\bar{x}, \bar{x}_0)$ holds. Thus, $\bar{x}_0$ is in S, or $\bar{x}_0$ has a predecessor in Q. Under the first condition, it follows immediately that $\bar{x}_0$ is in Z. Under the second condition, there exists an $\bar{x}_1$ such that $N(\bar{x}_1, \bar{x}_0)$ and $Q(\bar{x}_1)$ both hold. Since $\bar{x}_1$ is in Q, we can continue the above process, generating a sequence $\bar{x}_0, \bar{x}_1, \ldots$ where $N(\bar{x}_{i+1}, \bar{x}_i)$ holds for all i. Either this sequence terminates at some $\bar{x}_i$ in S, or it is infinite. In the terminating case, $T(\bar{x}_i, \bar{x}_0)$ holds, since there is a path from $\bar{x}_i$ to $\bar{x}_0$. Hence $\bar{x}_i$ is a witness for $\exists \bar{x}[S(\bar{x}) \wedge T(\bar{x}, \bar{x}_0)]$, so $\bar{x}_0$ is in Z. In the infinite case, there must exist $0 < m < n$ such that $\bar{x}_m = \bar{x}_n$, since we have assumed the domain is finite. In this case $T(\bar{x}, \bar{x})$ holds, where $\bar{x}$ is the common value of $\bar{x}_m$ and $\bar{x}_n$. Thus $\exists \bar{x}[T(\bar{x}, \bar{x}) \wedge T(\bar{x}, \bar{x}_0)$ holds, implying that $\bar{x}_0$ is in Z. We have shown that in all cases, if $Q = R(Q)$ and $\bar{x}_0$ is in Q, then $\bar{x}_0$ is in Z. Thus, Z is the greatest fixed point of $R(Q)$. □

The iterative squaring theorems can often be applied more than once to terms that have several fixed point operators. For example, consider the directed graph (V, E) described earlier. The relational term

$$R = \nu P\Big[V_0 \wedge \mu Q\left[\lambda y[\exists x[(P(x) \vee Q(x)) \wedge N(y, x)]]\right]\Big]$$

represents the set of vertices y in V_0 such that there is a path starting at y that passes through a vertex in V_0 infinitely often. Theorems 2 and 3 can be used twice to show that R is equal to

$$\lambda y[V_0(y) \wedge \exists x[V_0(x) \wedge T(x, x) \wedge T(y, x)]].$$

Unless otherwise noted, all the Mu-Calculus relational terms used in the remainder of this paper can be computed using the iterative squaring technique. As a result, the number of fixed point iterations can be made logarithmic in the cardinality of the domain.

6 Computation Tree Logic

Computation Tree Logic (CTL) is a propositional, branching-time, temporal logic (Clarke et al., 1986). 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 holds in a state provided that f holds at all points along all possible computation paths starting from that state, and $\mathbf{EF}f$, which holds in a state provided that there is a computation path such that f holds at some point 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 A be the set of atomic propositions, then:

1. Every atomic proposition p in A 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* $M = (A, S, L, N, S_0)$, where A is a set of atomic propositions, S is a finite set of states, $L : S \rightarrow 2^A$ 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. $\mathbf{EX}f$ is true in a state s of M if and only if s has a successor t such that f is true at t. **U** is the *until* operator. $\mathbf{E}[f\mathbf{U}g]$ is true in a state s of M if and only if there exists a path starting at 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.

EGf is true at a state s if there is a path starting at s such that f holds at each state on the path.

We also use the following syntactic abbreviations for CTL formulas:

- **AX**$f \equiv \neg$**EX**$\neg f$ which means that f holds at all successor states of the current state (f must hold at the *next* state).
- **EF**$f \equiv$ **E**[*true***U**f] which means that for some path, there exists a state on the path at which f holds (f is *possible* in the future).
- **AF**$f \equiv \neg$**EG**$\neg f$ which means that for every path, there exists a state on the path at which f holds (f is *inevitable* in the future).
- **AG**$f \equiv \neg$**EF**$\neg f$ which means that for every path, at every node on the path f holds (f holds *invariantly* along all paths).
- **A**[f**U**g] $\equiv \neg$**E**[$\neg g$**U**$\neg f \wedge \neg g$] $\wedge \neg$**EG**$\neg g$ which means that for every path, there exists 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 (f holds *until* g holds, along all paths).

6.1 CTL Model Checking

Checking whether a CTL formula f is true of a Kripke structure $M = (A, S, L, N, S_0)$ can be reduced to checking whether a Mu-Calculus formula f' is true of a structure $M' = (S, I_R, I_D)$. In the reduction, I_R provides the obvious interpretations for N and S_0; it also interprets each atomic proposition p in A to be a unary relation such that $I_R(p)(s)$ is true if and only if $p \in L(s)$. The individual variable interpretation I_D is not relevant since f' is defined to have no free individual variables.

The reduction of a CTL formula f to a Mu-Calculus formula f' is best understood by viewing CTL formulas as abbreviations for Mu-Calculus relational terms. In this view, 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. If f has no temporal operators, then it represents the relational term R that has exactly the same syntax as f. It remains only to consider CTL formulas of the form **EX**f, **EG**f or **E**[f**U**g]. For the remainder, we identify a CTL formula f with the Mu-Calculus relational term that it represents.

The CTL formula **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 **EX**f

to be a syntactic abbreviation for the Mu-Calculus relational term

$$\lambda s[\exists t[f(t) \wedge N(s,t)]].$$

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 derived from the identity

$$\mathbf{EG}f = f \wedge \mathbf{EX}\,\mathbf{EG}f.$$

It is straightforward to show that not only does **EG**f satisfy this equation, it is the greatest fixed point of the equation. Thus,

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

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:

$$\begin{aligned} \mathbf{E}[f\mathbf{U}g] &= g \vee (f \wedge \mathbf{EX}\,\mathbf{E}[f\mathbf{U}g]) \\ &= \mu Q[g \vee (f \wedge \mathbf{EX}Q)] \\ &= \mu Q[\lambda s[g(s) \vee (f(s) \wedge \exists t[Q(t) \wedge N(s,t)])]]. \end{aligned}$$

Once a CTL formula f has been transformed into a Mu-Calculus relational term R, it is still necessary to construct a Mu-Calculus formula f' that is true if and only if f is true of all the states in S_0. One such f' is

$$f' = \forall s[S_0(s) \Rightarrow f(s)].$$

As described in section 4, the Mu-Calculus model checking algorithm requires encoding the domain in terms of a Boolean domain. For Mu-Calculus formulas derived from CTL formulas, it is convenient to encode each state in the domain with the set of atomic propositions that are true for that state. This requires that no two distinct states have the same labeling of atomic propositions.

6.2 Fairness Constraints

Next, we consider the issue of *fairness*. In many cases, we are only interested in correctness along fair computation paths. For example, we may wish to consider only those computations in which some resource that is continuously requested by a process will eventually be granted to the process. This type of property cannot be expressed directly in CTL. In order to handle such properties we must modify the semantics of the logic slightly. A *fairness constraint* can be an arbitrary CTL formula. A path is said to be *fair* with respect to a set of fairness constraints if each constraint holds *infinitely often* along the path. The path quantifiers in CTL formulas are now restricted to fair paths. In the remainder of this section we describe how to translate CTL formulas to Mu-Calculus relational terms that reflect the modified semantics. We assume the fairness constraints are given by a set of CTL formulas $C = c_1, \ldots, c_n$. We write $\mathbf{E}_C\mathbf{X}f$ and $\mathbf{E}_C[f\mathbf{U}g]$, for example, to denote temporal operators with fairness constraints C.

Consider the formula $\mathbf{E}_C\mathbf{G}f$, which is true of a state s when there exists a path beginning at s in which f holds globally (invariantly) and each formula in C holds infinitely often. The set of such states Z is the largest set satisfying the following two conditions:

1. All of the states in Z satisfy f, and
2. for all $c_k \in C$, for all $s \in Z$, there is a path of length one or greater from s to a state satisfying c_k such that all states on the path satisfy f.

It is easy to show that if these conditions hold, each state in the set is the beginning of an infinite path on which f is always true, and every formula in C holds infinitely often. This gives us the characterization

$$\mathbf{E}_C\mathbf{G}f = \nu Z[f \wedge \bigwedge_{k=1}^{n} \mathbf{EX}\,\mathbf{E}[f\mathbf{U}(Z \wedge c_k)]].$$

The unfair CTL operators on the right side of the equations can be translated into Mu-Calculus relational terms as described above. Note that in this case, there is a nested fixed point since $\mathbf{EU}$ is an abbreviation for a least fixed point.

The cases of $\mathbf{E}_C\mathbf{X}f$ and $\mathbf{E}_C[f\mathbf{U}g]$ are a bit simpler. Define the set of all states which are on some fair computation as $h = \mathbf{E}_C\mathbf{G}\,true$. Then,

$$\mathbf{E}_C\mathbf{X}(f) = \mathbf{EX}(f \wedge h),$$
$$\mathbf{E}_C[f\mathbf{U}g] = \mathbf{E}[f\mathbf{U}(g \wedge h)].$$

7 Propositional Linear Temporal Logic

The tableau method for testing the satisfiability of propositional linear temporal logic (PTL) formulas (Manna and Wolper, 1981) 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 (1989) 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, they represent the states of the automaton explicitly, so their technique still suffers from the state explosion problem.

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

1. *atomic propositions* A (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 be extended easily 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 formalize this, let $\sigma \in [A \rightarrow \{0,1\}]^\omega$ be a sequence of truth assignments to the atomic propositions, and let σ_i be the ith suffix of σ (i.e., $\sigma_i(j) = \sigma(j+i)$ for all $j \in \omega$). The semantics of PTL formulas can be defined as follows:

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

The *tableau* associated with a PTL formula f is a Kripke structure whose atomic propositions represent the truth values of the particular formulas constructed from f. By representing the tableau symbolically, we can use the symbolic CTL model checking procedure to determine whether the formula f is satisfiable. A state of the tableau is a Boolean vector $\bar{x}$. With each formula f, we associate a component x_f of the state vector. A function $\alpha(f)$ associates a relational term in the Mu-Calculus with each PTL formula f.

This term represents the set of states of the tableau labeled with the formula f. The function α is defined recursively over the structure of PTL formulas as follows:

$$\begin{array}{rcl}
\alpha(p) & = & \lambda\bar{x}[x_p] \quad \text{if } p \in A, \\
\alpha(\neg f) & = & \neg\alpha(f), \\
\alpha(f \vee g) & = & \alpha(f) \vee \alpha(g), \\
\alpha(\mathbf{X}f) & = & \lambda\bar{x}[x_{\mathbf{X}f}], \\
\alpha(f\mathbf{U}g) & = & \alpha(g) \vee \\
 & & (\alpha(f) \wedge \lambda\bar{x}[x_{\mathbf{X}(f\mathbf{U}g)}])
\end{array}$$

Notice that for a given formula f, the only components of the state vector used in $\alpha(f)$ are the atomic propositions and the formulas $\mathbf{X}g$, where $\mathbf{X}g$ is a subformula of f, and $\mathbf{X}(g\mathbf{U}h)$, where $g\mathbf{U}h$ is a subformula of f. We call these subformulas the *elementary* subformulas of f, or $el(f)$. Using only the elementary formulas in the tableau reduces the number of Boolean state variables. The elementary subformulas can be defined recursively as follows (where f and g are any PTL formulas):

$$\begin{array}{rcl}
el(p) & = & \{p\} \quad \text{if } p \in A, \\
el(\neg f) & = & el(f), \\
el(f \vee g) & = & el(f) \cup el(g), \\
el(\mathbf{X}f) & = & \{\mathbf{X}f\} \cup el(f), \\
el(f\mathbf{U}g) & = & \{\mathbf{X}(f\mathbf{U}g)\} \cup el(f) \cup el(g).
\end{array}$$

The transition relation R of the tableau is defined such that the elementary formula $\mathbf{X}f$ is true in the current state if and only if f is true in the next state. Thus,

$$R = \lambda\bar{x}, \bar{x}' \bigwedge_{\mathbf{X}g \in el(f)} \alpha(\mathbf{X}g)(\bar{x}) \Leftrightarrow \alpha(g)(\bar{x}').$$

The set S_0 of initial states of the tableau is the set satisfying f. Thus, $S_0 = \alpha(f)$. The formula f is satisfiable if and only if there is an infinite path in the tableau such that

- f is true in the initial state, and

- for all subformulas $g\mathbf{U}h$, if $g\mathbf{U}h$ is true in some state, then eventually h is true in some later state.

This is equivalent to the CTL formula

$$\mathbf{E}_C\mathbf{G}\,\mathit{true}$$

with the set of fairness constraints

$$C = \{\neg\alpha(g\mathbf{U}h) \vee \alpha(h) \mid g\mathbf{U}h \text{ occurs in } f\}.$$

If there is an infinite path satisfying all of the formulas in C infinitely often, then for all subformulas $g\mathbf{U}h$, it is not the case that $g\mathbf{U}h$ holds forever after some point while h remains false. Hence, there is a path satisfying f.

The test for satisfiability of a formula f proceeds in the following steps. The set of elementary formulas of f is computed using its recursive definition. The symbolic (BDD) representations of R and S_0 are computed, using the recursive definition of α. The set C of fairness constraint formulas is constructed. Finally, the CTL formula $\mathbf{E}_C\mathbf{G}\mathit{true}$ is translated into the Mu-Calculus using the procedure of section 6.2. This formula is evaluated using the symbolic Mu-Calculus model checking procedure of section 4 to determine whether the formula f is satisfiable.

8 Observational Equivalence

In this section, we describe how to use the Mu-Calculus for expressing strong equivalence and weak equivalence of finite transition systems. This makes it possible to use the BDD-based Mu-Calculus model checking algorithm described earlier for deciding these equivalences. A finite transition system is a 4-tuple (S, s_0, Σ, Δ), where S is a finite set of *states*, s_0 is the *initial state*, Σ is a finite set of *actions*, and $\Delta \subseteq S \times \Sigma \times S$ is the *transition relation* (Milner, 1980; Milner, 1983).

8.1 Strong Equivalence

Let M_S and M_T be two finite transition systems on the same set of actions Σ. That is, let $M_S = (S, s_0, \Sigma, \Delta_S)$ and $M_T = (T, t_0, \Sigma, \Delta_T)$. The strong equivalence relation (written '$\sim$') is a subset of $S \times T$. The two finite transitions systems M_S and M_T are said to be strongly equivalent if and only if $s_0 \sim t_0$. The strong equivalence relation is the greatest fixed point of the function

$$F : 2^{S \times T} \rightarrow 2^{S \times T}$$

such that $F(R)$ is the set of all pairs (s, t) for which

- $\forall \sigma \, \forall s'$, if $\Delta_S(s, \sigma, s')$ then $\exists t'$ such that $\Delta_T(t, \sigma, t')$ and $R(s', t')$, and
- $\forall \sigma \, \forall t'$, if $\Delta_T(t, \sigma, t')$ then $\exists s'$ such that $\Delta_S(s, \sigma, s')$ and $R(s', t')$.

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. Let the domain D be the union of S, T and Σ (which are assumed to be disjoint). The relational interpretation I_R consists of the relations Δ_S and Δ_T, and the individual interpretation I_D consists of s_0 and t_0. Let F' be the Mu-Calculus relational term

$$\begin{aligned} \lambda s, t[&\forall \sigma, s'[\Delta_S(s, \sigma, s') \Rightarrow \\ &\quad \exists t'[\Delta_T(t, \sigma, t') \wedge R(s', t')]] \\ &\wedge \forall \sigma, t'[\Delta_T(t, \sigma, t') \Rightarrow \\ &\quad \exists s'[\Delta_S(s, \sigma, s') \wedge R(s', t')]]]. \end{aligned}$$

Then $F'(s, t)$ is true if and only if (s, t) is an element of $F(R)$. Thus, M_S and M_T are strongly equivalent if and only if $\nu R[F'](s_0, t_0)$ holds. This can be evaluated with the BDD-based model checking algorithm, although the iterative squaring transformation cannot be used.

8.2 Weak Equivalence

Let τ be a distinguished action in the set Σ, and let the relation H be the reflexive transitive closure of $\lambda x, y[\Delta(x, \tau, y)]$. That is, $H(s, t)$ is true if and only if there is a path from s to t labeled by a sequence of zero or more τ actions. Also, let Δ^* be such that

$$\Delta^*(s, \sigma, t) = \exists x \exists y[H(s, x) \wedge \Delta(x, \sigma, y) \wedge H(y, t)].$$

The weak observational equivalence relation is the greatest fixed point of the function

$$G : 2^{S \times T} \rightarrow 2^{S \times T},$$

such that $G(R)$ is the set of all pairs (s, t) for which

- $\forall s' \forall \sigma$, if $\Delta_S^*(s, \sigma, s')$ then $\exists t'$ such that $\Delta_T^*(t, \sigma, t')$ and $R(s', t')$, and
- $\forall t' \forall \sigma$, if $\Delta_T^*(t, \sigma, t')$ then $\exists s'$ such that $\Delta_S^*(s, \sigma, s')$ and $R(s', t')$.

From this point, the translation of weak equivalence into the Mu-Calculus is completely analogous to the translation for strong equivalence.

9 ω-Automata

Finally, we discuss symbolic Mu-Calculus based algorithms for deciding language containment between finite ω-automata. We consider Büchi automata in detail, and also discuss a general method that is applicable to a large class of ω-automata.

A finite Büchi automaton is an ordered 5-tuple $(S, s_0, \Sigma, \Delta, B)$, where S is a finite set of *states,* $s_0 \in S$ is the *initial state,* Σ is a finite *alphabet,* $\Delta \subseteq S \times \Sigma \times S$ is the *transition relation,* and $B \subseteq S$ is the *acceptance set.* The automaton is *deterministic* if for all $s \in S$ and $\sigma \in \Sigma$, there exists exactly one $t \in S$ such that $\Delta(s, \sigma, t)$ holds. An infinite sequence of states $t_0, t_1, t_2, \ldots \in S^\omega$ is a *path* of a Büchi automaton if there exists an infinite sequence $\sigma_0, \sigma_1, \sigma_2, \ldots \in \Sigma^\omega$ such that

$$\langle t_i, \sigma_i, t_{i+1} \rangle \in \Delta$$

for all $i \geq 0$. A sequence $\sigma_0, \sigma_1, \sigma_2, \ldots$ is *accepted* by a Büchi automaton if the corresponding path $t_0, t_1, t_2, \ldots$ goes through a one or more elements of B infinitely often. The set of sequences accepted by an automaton M is called the language of M and denoted $\mathcal{L}(M)$.

To determine whether the language of a Büchi automaton M is contained in the language of a Büchi automaton M' (with the same alphabet), we define a Kripke structure M'' representing the product of M and M', and write a formula in CTL which is true of M'' if and only if every sequence accepted by M is also accepted by M'. This formula can be translated into the Mu-Calculus and evaluated using the symbolic model checking algorithm.

Let M'' be a Kripke structure $(A, S \times S', L, R, S_0'')$, where

- $A = \{p, p'\}$ is the set of atomic propositions,
- $S_0'' = \{\langle s_0, s_0' \rangle\}$,
- $\langle s, s' \rangle \models p$ iff $s \in B$,

- $\langle s, s' \rangle \models p'$ iff $s' \in B'$,
- $\langle s, s' \rangle R \langle t, t' \rangle$ iff $\exists \sigma \in \Sigma$ such that $\langle s, \sigma, t \rangle \in \Delta$ and $\langle s', \sigma, t' \rangle \in \Delta'$.

Recall that in Section 6 we showed how to encode Kripke structures symbolically. The transition relation of the Kripke structure M'' is

$$R = \lambda s, s', t, t'[\exists \sigma[\Delta(s, \sigma, t) \wedge \Delta'(s', \sigma, t')]].$$

The atomic proposition p can identified with the Mu-Calculus relational term $\lambda s, s'[B(s)]$ that represents that set of states that satisfy p. Similarly, p' is identified with the relational term $\lambda s, s'[B'(s')]$. The set of initial states is

$$S_0'' = \lambda s, s'[(s = s_0) \wedge (s' = s_0')].$$

In (Clarke et al., 1990), it is shown that, if M' is deterministic, then $\mathcal{L}(M) \subseteq \mathcal{L}(M')$ if and only if

$$M'' \models \mathbf{A}(\mathbf{GF}p \Rightarrow \mathbf{GF}p').$$

Note that the formula above is not a CTL formula since there are path operators that are not immediately preceded by path quantifiers. However, it is equivalent to $\mathbf{AG}\,\mathbf{AF}p'$ under the fairness constraint "infinitely often p". Thus, $\mathcal{L}(M) \subseteq \mathcal{L}(M')$ holds if and only if the formula $\mathbf{A}_C\mathbf{G}\,\mathbf{A}_C\mathbf{F}p'$ holds, where $C = \{p\}$. Using the results of section 6.2, and the above definitions for R, S_0'', p and p', this formula can be translated into a Mu-Calculus formula that can be evaluated using the Mu-Calculus model checking algorithm of section 4.

Another possible approach to the language containment problem makes use of the iterative squaring technique for computing transitive closures. Let T^* be the set of all pairs of states of the Kripke structure such that the second state can be reached from the first without passing through B'. This is the transitive closure of

$$T = \lambda s, s', t, t'[R(s, s', t, t') \wedge \neg B'(s') \wedge \neg B'(t')].$$

Using iterative squaring,

$$T^* = \mu Q[\lambda s, s', t, t'[T(s, s', t, t') \vee \exists u, u'[Q(s, s', u, u') \wedge Q(u, u', t, t')]]].$$

The language of M is contained in the language of M' iff there is no path to a state $\langle s, s' \rangle$ in B such that $\langle s, s' \rangle$ is on a cycle not passing through

B'. That is, $\mathcal{L}(M) \subseteq \mathcal{L}(M')$ if and only if $\neg \mathbf{EF}\, \lambda s, s'[T^*(s, s', s, s')]$. The operator **EF** can also be evaluated using iterative squaring. This technique reduces the number of iterations to the log of the diameter of the transition relation R. Using the technique based on CTL model checking with fairness constraints, the number of iterations may be as high as the square of the diameter, because of the nested fixed point operators. However, in many cases the BDDs needed to construct the transitive closure are impractically large. As a result, if the diameter of the state space is small, the nested fixed point method may be preferable.

While deterministic Büchi automata cannot express the complete class of ω-regular languages, algorithms for language containment for more expressive types of deterministic ω-automata (*e.g.,* Muller automata) can be derived in a similar fashion from results in (Clarke et al., 1990). These algorithms require a more expressive class of fairness constraints than we have considered here. Mu-Calculus based algorithms for this class of fairness constraints exist, and can be derived either from the PTL satisfiability algorithm, or from results of Emerson and Lei (1986).

10 Empirical Results

Using BDDs for testing Boolean satisfiability is only efficient in a heuristic sense. The satisfiability problem is, of course, NP-complete; the only claim that is made for BDDs is that they perform well for certain useful classes of Boolean functions. Likewise, using BDDs for representing relations in Mu-Calculus model checking 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 on some problems of practical interest.

Here we briefly present 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 are pipeline circuit 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: the operands are read from the register file, then an ALU (Arithmetic Logic Unit) operation is performed, then the result is written

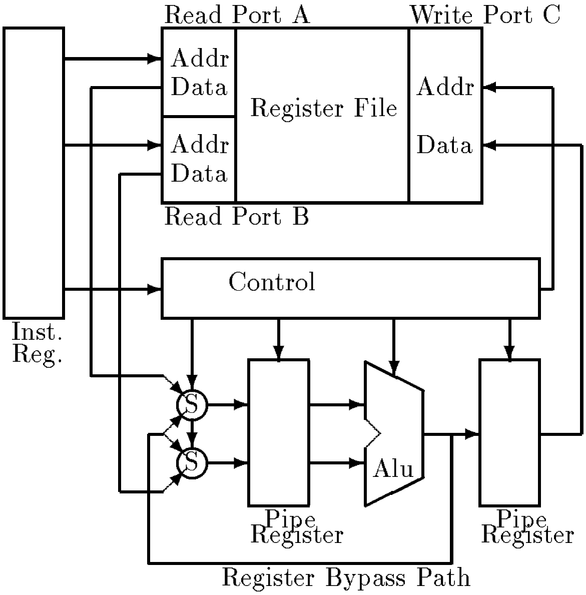

Figure 3: Block diagram of simple pipeline design

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 no instruction is available. When this occurs, a "no-operation" is propagated through the pipe. A functional block diagram of a typical pipeline is given in figure 3.

Since vectors of Boolean values are used to represent binary numbers in these designs, it is useful to introduce some notation for vectors of propositions in logical formulas. First, we extend the standard logical and modal operators to vectors of propositions in a component-wise manner. For example,

$$\begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{bmatrix} \wedge \begin{bmatrix} q_1 \\ q_2 \\ \vdots \\ q_n \end{bmatrix} \equiv \begin{bmatrix} p_1 \wedge q_1 \\ p_2 \wedge q_2 \\ \vdots \\ p_n \wedge q_n \end{bmatrix}$$

and

$$F\begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{bmatrix} \equiv \begin{bmatrix} Fp_1 \\ Fp_2 \\ \vdots \\ Fp_n \end{bmatrix}.$$

The latency in the example pipelines is three clock cycles. For this reason, the specification of the pipeline cannot be given in a straightforward manner using simply pre-conditions and post-conditions on operations. We can, however, use temporal operators and the above notation to specify the behavior of the pipeline, taking into account the pipe latency. When we specify a register transfer level operation for the pipeline, it is understood that the results of the operation will not affect the register file until three clocks cycles in the future, and that the inputs to the operation correspond to the state of the register file two clock cycles in the future. The state of the register file n clock cycles in the future can be expressed as $\mathbf{X}^n\mathbf{R}$. A register transfer specification such as $\mathbf{R_c} \leftarrow \mathbf{R_a} \oplus \mathbf{R_b}$ means that register $\mathbf{c}$ receives the exclusive-or of registers $\mathbf{a}$ and $\mathbf{b}$. Taking into account the pipe latency, this register transfer level specification can be expressed as a temporal formula in the following way:

$$(\mathbf{X}^3\mathbf{R})_\mathbf{c} = (\mathbf{X}^2\mathbf{R})_\mathbf{a} \oplus (\mathbf{X}^2\mathbf{R})_\mathbf{b},$$

where $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ are each bit-fields in the operation code. As similar formulas can be derived for other register transfer level expressions, we will write register transfer expressions in our specifications, with the understanding that they are to be interpreted as abbreviations for temporal logic formulas in the above way. Since $\mathbf{X}^n p$ is a path formula and not a state formula, it cannot be evaluated directly by the CTL model checker (which can only evaluate state formulas). We can show, however, that the state of the register file $\mathbf{R}$ two or three clock cycles in the future is uniquely determined by the current state of the system. We can show this by automatically checking the CTL formulas

$$\mathbf{AG}((\mathbf{EX})^2\mathbf{R} \equiv (\mathbf{AX})^2\mathbf{R})$$

and

$$\mathbf{AG}((\mathbf{EX})^3\mathbf{R} \equiv (\mathbf{AX})^3\mathbf{R}).$$

Thus, we can substitute the state formula $(\mathbf{EX})^2\mathbf{R}$ for the path formula $\mathbf{X}^2\mathbf{R}$, since the two are equivalent. Likewise, we can substitute $(\mathbf{EX})^3\mathbf{R}$ for $\mathbf{X}^3\mathbf{R}$.

Using the above temporal interpretation for register transfer level specifications, we write the specification for our simplest pipeline (which has only an exclusive-or instruction) as follows:

$$\mathbf{AG}(\neg STALL \Rightarrow (\mathbf{R_c} \leftarrow \mathbf{R_a} \oplus \mathbf{R_b})) \tag{1}$$

and

$$\mathbf{AG}\forall \mathbf{c}'(\mathbf{c} \neq \mathbf{c}' \vee STALL \Rightarrow (\mathbf{R_{c'}} \leftarrow \mathbf{R_{c'}})).$$

Recall that the register assignments are abbreviations for CTL formulas. The latter formula specifies that non-destination registers do not change, and that if a stall occurs, no registers change.

Figure 4 graphs the performance we obtained when checking formula 1 on a variety of pipelines of this type. The graphs show the total execution time and the size of the BDD needed to represent the transition relation. In all cases the register file had four registers. The number of bits per register varied from 1 to 12. We considered two ALU operations: exclusive-or and addition. In two cases the ALU performed just one of these operations. In the third case, the ALU performed both operations. The verifier operated directly on CTL formulas, which reduces the overhead that would result from first translating CTL formulas to Mu-Calculus formulas.

A pipeline with 12 bits has approximately 1.5×10^{29} reachable states, which puts it far outside the range of model checkers like the one reported by Browne *et al.* (1986). An 8-bit exclusive-or pipeline required a BDD with 42,000 nodes to represent the transition relation, and approximately 22 minutes to verify on a Sun 3/60. The execution times in the graph are for a single processor of an Encore Multimax, which is approximately half as fast as a Sun 3. The most interesting result is that the number of nodes in the transition relation BDD is asymptotically linear in the number of bits per register. As a result, the verification time is polynomial in the number of bits. The BDD variables were ordered so that all variables in a given bit position were grouped together. A fixed number of signals, consisting of the control bits and the ALU carry bit pass from one group to the next. It is this property of the system that results in the linear growth of the transition relation as represented by a BDD.

It is also interesting to note that adding an exclusive-or operation to the addition pipeline roughly doubles the number of nodes in the transition relation. In general, the transition relation increases in size linearly with the number of instructions (Burch et al., 1991a). In addition, if the ALU were able to perform a multiply operation, a barrel shift, or some other complex

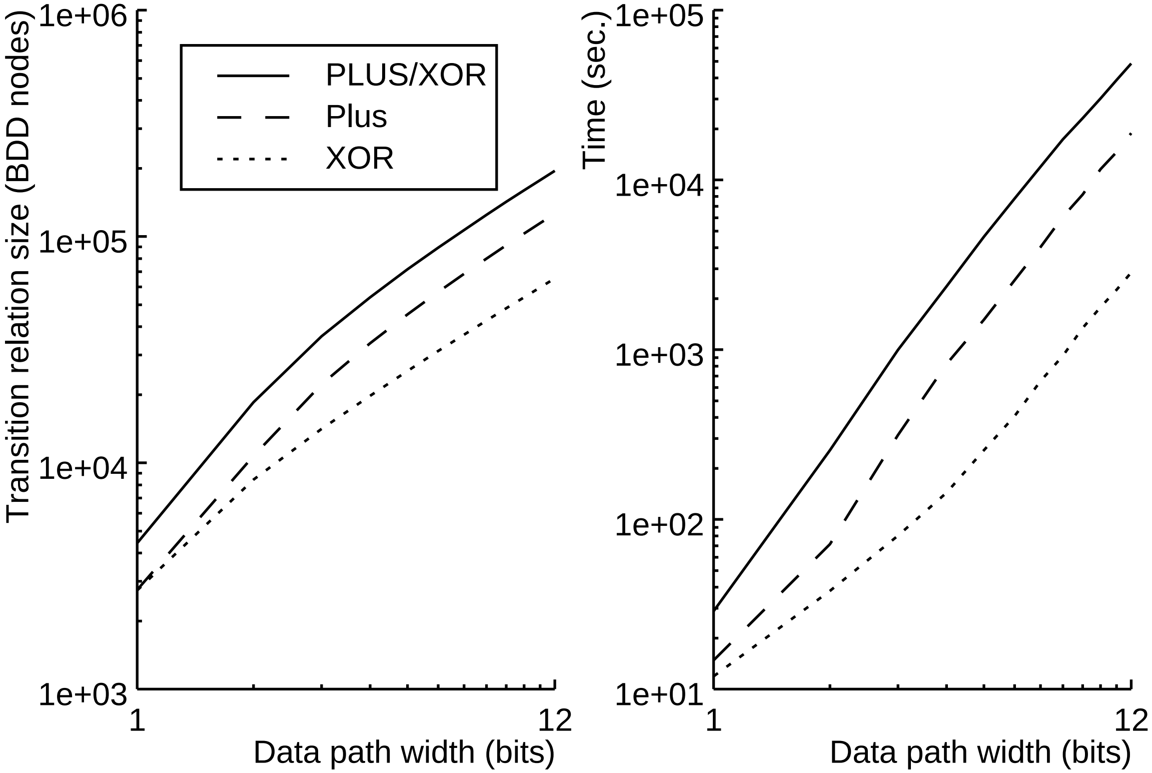

Figure 4: Performance of BDD model checking algorithm on simple pipelines

operation which has more than a constant amount of information passing from one bit position to the next, then the size of the BDD representation would quickly become unmanageable.

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 (Bryant, 1991). 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

Bose, S. and Fisher, A. L. (1989). Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic. In Claesen, L., editor, *Proceedings of the IMEC-IFIP International Workshop on Applied Formal Methods For Correct VLSI Design*, pages 759–764.

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

Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8).

Bryant, R. E. (1991). On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–213.

Burch, J. R., Clarke, E. M., and Long, D. E. (1991a). Representing circuits more efficiently in symbolic model checking. In *28th ACM/IEEE Design Automation Conference*.

Burch, J. R., Clarke, E. M., and Long, D. E. (1991b). Symbolic model checking with partitioned transition relations. In *Proceedings of the International Conference on Very Large Scale Integration*, Edinburgh, Scotland.

Burch, J. R., Clarke, E. M., McMillan, K. L., and Dill, D. L. (1990). Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE Design Automation Conference*.

Clarke, E. M., Draghicescu, I. A., and Kurshan, R. P. (1990). A unified approach for showing language containment and equivalence between various types of ω-automata. In Arnold, A. and Jones, N. D., editors, *15th Colloquium on Trees in Algebra and Programming*, volume 431 of *Lecture Notes in Computer Science*, Copenhagen, Denmark. Springer-Verlag.

Clarke, E. M., Emerson, E. A., and Sistla, A. P. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263.

Cleaveland, R. (1989). Tableau-based model checking in the propositional mu-calculus. Technical Report 2/89, University of Sussex.

Coudert, O., Berthet, C., and Madre, J. C. (1989). Verification of synchronous sequential machines based on symbolic execution. In Sifakis, J., editor, *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag.

Emerson, E. A. and Lei, C.-L. (1986). Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the First Annual Symposium on Logic in Computer Science*, Boston, Mass.

Fujita, M. and Fujisawa, H. (1989). Specification, verification, and synthesis on control circuits with propositional temporal logic. In Darringer, J. A. and Rammig, F. J., editors, *Proceedings of the Ninth International Symposium on Computer Hardware Description Languages and their Applications*, Washington, D.C. North-Holland.

Godefroid, P. (1990). Using partial orders to improve automatic verification methods. In (Kurshan and Clarke, 1990). Also in Springer-Verlag LNCS 531.

Godefroid, P. and Wolper, P. (1991). A partial approach to model checking. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science.*

Kimura, S. and Clarke, E. M. (1990). A parallel algorithm for constructing binary decision diagrams. In *Proceedings: IEEE International Conference on Computer Design.*

Kozen, D. (1983). Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354.

Kurshan, R. and Clarke, E. M., editors (1990). *Computer-Aided Verification, Proceedings of the 1990 Workshop*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society. Also in Springer-Verlag LNCS 531.

Manna, Z. and Wolper, P. (1981). Synthesis of communicating processes from temporal logic specifications. In Kozen, D., editor, *Logic of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*, Yorktown Heights, New York. Springer-Verlag.

Milner, R. (1980). *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag.

Milner, R. (1983). Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310.

Park, D. (1974). Finiteness is mu-ineffable. Theory of Computation Report No. 3, The University of Warwick.

Probst, D. K. and Li, H. F. (1990). Using partial order semantics to avoid the state explosion problem in asynchronous systems. In (Kurshan and Clarke, 1990). Also in Springer-Verlag LNCS 531.

Stirling, C. and Walker, D. J. (1989). Local model checking in the modal mu-calculus. In Diaz, J. and Orejas, F., editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 351–352 of *Lecture Notes in Computer Science*. Springer-Verlag.

Valmari, A. (1989). Stubborn sets for reduced state space generation. In *Tenth International Conference on Application and Theory of Petri Nets*.

Valmari, A. (1990). A stubborn attack on the state explosion problem. In (Kurshan and Clarke, 1990). Also in Springer-Verlag LNCS 531.

Yoneda, T., Nakade, K., and Tohma, Y. (1989). A fast timing verification method based on the independence of units. In *Proceedings of the Nineteenth International Symposium on Fault-Tolerant Computing*.