An Overview of Model Checking

Ma Li
Center for Logic, Language and Cognition
Peking University

November 23, 2009

Abstract

This paper is devoted to investigate model checking. The main motivation is rather to check how much I know about model checking than to give a presentation itself. The paper is composed of four parts. In the first part, model checking is introduced, including the background, its advantages over other verifying techniques as well as its main disadvantage. In the second part, we will investigate how to do model checking, that is to say, the process of model checking. In the third part, temporal logic as a specifying logic for model checking will be discussed. In the last part, two main breakthroughs on the state space explosion will be concerned. They are symbolic model checking and partial order reduction.

1 Introduction

Computer takes a greater and greater role in our life, involving economics, traffic, spaceflight and many other important aspects. Usually, we design some complex system, then let it implemented by the computer. A problem has been haunting over us is that how can we ensure the correctness of the design. Sometimes, even a same failure can lead to a economical disaster or even life losses. To solve this problem, there are several approaches, such as simulation, testing and model checking. The former two are both informal while the latter is formal. While simulation and testing explore some of the possible behaviors and scenarios of the system, leaving open the question of whether the unexplored trajectories may contain the fatal bug, formal verification conducts an exhaustive exploration of all possible behaviors. Thus, when a design is pronounced by a formal verification method, it implies that all behaviors have been explored, and the questions of adequate coverage or a missed behavior become irrelevant. That is to say, by model checking, a desired behavior property of a reactive system is verified over a given system(the model) through exhaustive enumeration of all the states reachable by the system and the behaviors that traverse through them.

Model checking enjoys the following advantages for verification of circuits and protocols. First and the most important one is that it can be implemented completely automatically. Secondly, the model checking algorithm will either terminate with the answer true, indicating that the model satisfies the specification or give a counterexample execution that shows why the formula is not satisfied. The counterexamples are particularly important in finding subtle errors in complex transition systems. The third advantage of model checking is that it is fast. Partial specification can be checked, so it is unnecessary to specify completely the system before useful information can be obtained regarding its correctness. When a specification is not satisfied, other formulas-not part of the original specification- can be checked in order to locate the source of the error. Finally, the logic used for specification has strong expressive power. It can express many of the properties that are needed for reasoning about concurrent systems([1]).

Apart from the above advantages, model checking also has its own disadvantages. The main disadvantage of it is the state space explosion that can occur if the system being verified has many components that can make transitions in parallel. In this case, the number of global system states may grow exponentially with the number of processes.

2 How to Do Model Checking

The first step is to convert a design into a formalism accepted by a model checking tool. In many cases, this is simply a compilation task. Sometimes, for time and memory's sake, the modeling of design may require the use of abstraction to eliminate irrelevant or unimportant details.

The second step is to specify the properties that the designs must satisfy. The specification is usually given in some logical formulas. Both hardware and software systems can use temporal logic as their specification logic, which can assert how the behavior of the system evolves over time.

The last step is to verify whether the model obtained in the first step satisfied the specification got in the second step. Ideally the verification is fully automatic. However, in practice it often involves human assistance. One such manual activity is the analysis of the verification results. In case of a negative result, the user is often provided with an error trace. This tracking down where the error occurred. In this case, analyzing the error trace may require a modification to the system and reapplication of the model checking algorithm.

Because the verification relates both modeling and specification, the error trace may result from both incorrect modeling and incorrect specification. The error trace can help us to fix these two problems. Another possibility is that the verification algorithm may not terminate eventually, due to the size of the model, which is too large to fit into the computer memory. In this situation, we should remodel the design.

3 Temporal Logic

Temporal logic is a formalism for describing sequences of transition between states in a reactive system. In the temporal logics that we will consider, time is not mentioned explicitly; instead, a formula might specify that eventually some designated state is reached, or that an error state is never entered. Properties like *eventually* or *never* are specified using special temporal operators. These operators can also be combined with boolean connectives or nested arbitrarily. Temporal logics differ in the operators that they provide and the semantics of those operators. We will focus on a powerful logic called CTL^*[1][2].

3.1 The Computation Tree Logic CTL^*

Conceptually, CTL^* formulas describe properties of *computation trees*. The tree is formed by designating a state in a Kripke structure as the *initial state* and then unwinding the structure into an infinite tree with the designated state at the root, as illustrated in Figure3.1. The computation tree shows all of the possible executions starting from the initial state.

In CTL^* formulas are composed of *path quantifiers* and temporal operators. The path quantifiers are used to describe the branching structure in the computation tree. There are two such quantifiers **A**("for all computation paths") and **E** ("for some computation path"). These quantifiers are used in a particular state to specify that all of the paths or some of the paths starting at that state have some property. The temporal operators describe properties of a path through the tree. There are five basic operators:

- **X**("next time") requires that a property holds in the second state of the path.
- The **F**("eventually" or "in the future") operator is used to assert that a property will hold at some state on the path.
- **G**("always" or "globally") specifies that a property holds at every state on the path.
- The **U** ("until") operator is a bit more complicated since it is used to combine two properties. It holds if there is a state on the path where the second property holds, and at every preceding state on the path, the first property holds.
- **R**("release") is the logical dual of the **U** operator. It requires that the second property holds along the path up to and including the first state where the first property hold. However, the first property is not required to hold eventually.

The remainder of this section contains a precise description of the syntax and semantics of CTL^*. There are two types of formulas in CTL^*: *state formulas*(which are true in a specific state) and *path formulas* (which are true along a specific path). Let AP be the set of atomic proposition names. The syntax of state formulas is given by the following rules:

- If $p\in AP$, then p is a state formula.
- I f and g are state formulas, then $\neg f$, $f \vee g$ and $f \wedge g$ are state formulas.

- f is a path formula, the $\mathbf{E}f$ and $\mathbf{A}f$ are state formulas.

Two additional rules are needed to specify the syntax of path formulas:

- If f is a state formula, then f is also a path formula.
- If f and g are path formulas, then $\neg f$, $f \vee g$, $f \wedge g$, $\mathbf{X}f$, $\mathbf{F}f$, $\mathbf{G}f$, $f\mathbf{U}g$, and $f\mathbf{R}g$ are path formulas.

CTL^* is the set of state formulas generated by the above rules.

We define the semantics of CTL^* with respect to a Kripke structure. Recall that a Kripke structure$M$ is a triple$\langle S, R, L\rangle$, where S is the set of states; $R \subseteq S \times S$ is the transition relation, which must be total (i.e., for all states $s \in S$ there exists a state $s' \in S$ such that $(s, s') \in R$); and $L : S \to 2^{AP}$ is a function that labels each state with a set of atomic propositions true in that state. A *path in M* is an infinite sequence of states, $\pi = s_0, s_1, \ldots$ such that for every $i \geq 0$, $(s_i, s_{i+1}) \in R$. (Alternatively, we can think of a path as an infinite branch in the computation tree that corresponds to the Kripke structure.)

We use π^i to denote the *suffix*of π starting ats^i. If f is a state formula, the notation $M, s \models f$ means that f holds at state s in the Kripke structure M. Similarly, is $f$is a path formula, $M, \pi \models f$ means that f holds along path π in the Kripke structure M. When the Kripke structure M is clear from the context, we will usually omit it. The relation $\models$is defined inductively as follows (assuming that f_1 and f_2 are two state formulas and g_1 and g_2 are path formulas):

1. $M, s \models p \quad \Leftrightarrow p \in L(s)$.
2. $M, s \models \neg f_1 \Leftrightarrow M, s$ un-satisfies f_1
3. $M, s \models f_1 \vee f_2 \Leftrightarrow M, s \models f_1$ or $M, s \models f_2$
4. $M, s \models f_1 \wedge f_2 \Leftrightarrow M, s \models f_1$ and $M, s \models f_2$
5. $M, s \models \mathbf{E}g_1 \quad \Leftrightarrow$ there is a path π from s such that $M, \pi \models g_1$.
6. $M, s \models \mathbf{A}g_1 \quad \Leftrightarrow$ for every path π starting from s, $M, \pi \models g_1$.
7. $M, \pi \models f_1 \Leftrightarrow s$ is the first state of π and $M, s \models f_1$
8. $M, s \models \neg g_1 \Leftrightarrow M, s g_1$
9. $M, \pi \models g_1 \vee g_2 \Leftrightarrow m, \pi \models g_1 or M, \pi \models g_2$
10. $M, \pi \models g_1 \wedge g_2 \Leftrightarrow m, s \models g_1 and M, s \models g_2$
11. $M, \pi \models \mathbf{X}g_1 \quad \Leftrightarrow M, \pi^1 \models g_1$.
12. $M, \pi \models \mathbf{F}g_1 \quad \Leftrightarrow$ there exists a k ≥ 0 such that $M, \pi^k \models g_1$.

13. $M, \pi \models \mathbf{G}g_1 \Leftrightarrow$ for all i $\geq$0, $M, \pi^i \models g_1$.
14. $M, \pi \models g_1\mathbf{U}g_2 \quad \Leftrightarrow$ there exists a k $\geq$0 such that $M, \pi^k \models g_2$ and for all 0$\leq$ $j < k$, $M, \pi^j \models g_1$.
15.$M, \pi \models g_1\mathbf{R}g_2 \Leftrightarrow$ for all $j \geq 0$, if for every $i < j$, M, π^i un-satisfies g_1 then $M, \pi^j \models g_2$.

It is easy to see that the operators $\vee$, $\neg$, **X**, **U**, and **E** are sufficient to express any other CTL^* formulas.

- $f \wedge g \equiv \neg(\neg f \vee \neg g)$
- $f\mathbf{R}g \equiv \neg(\neg f\mathbf{U}\neg g)$
- $\mathbf{F}f \equiv \mathit{True}\ \mathbf{U}\ f$
- $\mathbf{G}f \equiv \neg\mathbf{F} \neg f$
- $\mathbf{A}(f) \equiv \neg\mathbf{E} \neg (f)$

3.2 CTL and LTL

In this section we consider two useful sublogics of CTL^*: one is a branching-time logic and one is a linear-time logic. The distinction between them is in how they handle branching in the underlying computation tree.In branching-time temporal logic the temporal operators quantify over the paths that are possible from a given state. In linear-time temporal logic, operators are provided for described for describing events along a single computation path.

Computation Tree Logic(CTL) is a restricted subset of CTL^* in which each of the temporal operators **E**, **F**, **G**, **U** and **R** must be immediately preceded by a path quantifier. More precisely, CTL is the subset of CTL^* that is obtained by restricting the syntax of path formulas using the following rule.

- If f and g are state formulas, then $\mathbf{X}f$, $\mathbf{F}f$, $\mathbf{G}f$, $f\mathbf{U}g$ and $f\mathbf{R}g$ are path formulas.

Linear Temporal Logic(LTL), on the other hand, will consist of formulas that have the form $\mathbf{A}f$ where f is a path formula in which the only state subformulas permitted are atomic propositions. More precisely, an LTL path formula is either

- If $p \in AP$, then p is a path formula.
- If f and g are path formulas, then $\neg f$, $f \vee g$, $f \wedge g$, $\mathbf{X}f$, $\mathbf{F}f$, $\mathbf{G}f$, $f\mathbf{U}g$, and $f\mathbf{R}g$ are path formulas.

It can be shown that the three logics that we have discussed have different expressive powers. For example, there is no CTL formula that is equivalent to the LTL formula $\mathbf{A}(\mathbf{FG}p)$. This formula express the property that along every path, there is some state from which p will hold forever. Likewise, there is no LTL formula that is equivalent to the CTL formula $\mathbf{AG}(\mathbf{EF}p)$. The disjunction of these two formula $\mathbf{A}(\mathbf{FG}p)\vee$ $\mathbf{AG}(\mathbf{EF}p)$ is a CTL^* formula that is not expressible in either CTL or LTL.

Most of the specifications in the following part of this article will be written in the logic CTL. There are ten basic CTL operators:

- **AX** and **EX**,
- **AF** and **EF**
- **AG** and **EG**
- **AU** and **EU**
- **AR** and **ER**

Each of the ten operators can be expressed in terms of three operators **EX**, **EG** and **EU**:

- $\mathbf{AX}f = \neg\ \mathbf{EX}(\neg f)$
- $\mathbf{EF}f = \mathbf{E}[\mathit{True}\mathbf{U}f]$
- $\mathbf{AG}f = \neg\ \mathbf{EF}(\neg f)$
- $\mathbf{A}[f\mathbf{U}g] = \neg\ \mathbf{E}[\neg g\mathbf{U}(\neg f \wedge \neg g)] \wedge \neg \mathbf{EG}\neg g$
- $\mathbf{A}[f\mathbf{R}g] = \neg\ \mathbf{E}[\neg f\mathbf{U}\neg g]$

4 Some Breakthrough on State Space Explosion (Mainly Focus on the Symbolic Model Checking)

As mentioned in the first part, the main disadvantage of model checking is the state space explosion problem. Because of this problem, many researchers in formal verification predicted that model checking would never be practical for large systems. However, in the late 1980s, the size of the transition systems that could be verified by model-checking techniques increased dramatically. Much of the increase has been due to some breakthrough on the model checking, such as symbolic model checking, partial order reduction, bounded model checking and localization reduction. We will focus on the symbolic model checking in this article.

4.1 Binary Decision Diagrams

Before the introduction of symbolic model checking, we first have a look at the OBDDs, which is used in the symbolic model checking.Ordered binary decision diagrams (OBDDs) are a canonical form representation for boolean formulas. They are often substantially more compact than traditional normal forms such as conjunction normal form and disjunction normal form, and they can be manipulated very efficiently. Hence, they have become widely used for a variety of applications in computer aided design, including symbolic simulation, verification of combinational logic and, more recently, verification of finite-state concurrent systems.

A binary decision tree is a rooted, directed tree that consists of two types of vertices, terminal vertices and nonterminal vertices. Each nonterminal vertex v is labeled by a variable *var(v)* and has two successors: *low(v)* corresponding to the case where the variable is assigned 0, and *high(v)* corresponding the case where v is assigned 1. If the variable v is assigned 0, then the next vertex on the path from the root to the terminal vertex will be *low(v)*. If v is assigned 1 then the next vertex on the path will be *high(v)*. The value that labels the terminal vertex will be the value of the function for this assignment.

Binary decision trees do not provide a very concise representation for boolean functions. In fact, they are essentially the same size as truth tables.

Fortunately, there is usually a lot of redundancy in such trees, such as isomorphic subtrees. Thus, we can obtain a more concise representation for the boolean function by merging isomorphic subtrees. This results in a directed acyclic graph (DAG) called a *binary decision diagram.* More precisely, a binary decision diagram is a rooted, directed acyclic graph with two types of vertices, terminal vertices and nonterminal vertices. As in the case of binary decision trees, each nonterminal vertex v is labeled by a variable *var(v)* and has two successors, *low(v)* and *high(v).* Each terminal vertex is labeled by either 0 or 1. Every binary decision diagram B with root v determines a boolean function $f_v(x_1, \ldots, x_n)$ in the following manner:
1. If v is a terminal vertex
(a) If *value(v)* =1 then $f_v(x_1, \ldots, x_n)$=1.
(b) If *value(v)* =0 then $f_v(x_1, \ldots, x_n)$=0.
2. v is a nonterminal vertex with *var(v)*=x_i then f_v is the function
$f_v(x_1, \ldots, x_n)$=$(\neg\, x_i \wedge\, f_{low(v)}(x_1, \ldots, x_n)) \vee (\, x_i \wedge\, f_{high(v)}(x_1, \ldots, x_n))$

In practical applications it is desirable to have a canonical representation for boolean functions. Such a representation must have the property that two boolean functions are logically equivalent if and only if they have isomorphic representations. This property simplifies tasks like checking equivalence of two formulas and deciding if a given formula is satisfiable or not. Two binary decision diagrams are isomorphic if there exists a one-to-one and onto function h that maps terminals of one to terminals of the other and nonterminals of one to nonterminals of the other, such that for every terminal vertex v, *var(v)=var(h(v))*, *h(low(v))=low(h(v))*, and *h(high(v))=high(h(v)).*

Bryant [2] showed how to obtain a canonical representation for boolean functions by placing two restrictions on binary decision diagrams. Firs, the variables should appear in the same order along each from the root to a terminal. Second, there should be no isomorphic subtrees or redundant vertices in the diagram. The first requirement is achieved by imposing a total ordering $<$ on the variables that label the vertices in the binary decision diagram and requiring that for any vertex u in the diagram, if u has a nonterminal successor$v$, then *var(u)*$<$ *var(v).* The second requirement is achieved by the diagram:

- *Remove duplicate terminals* Eliminate all but one terminal vertex with a given label and redirect all arcs to the eliminated vertices to the remaining one.

- *Remove duplicate nonterminals* If two nonterminals *var(u)* and *var(v)* have *var(u)=var(v)* , *low(u)< low(v)* and *high(u)< high(v)*, then eliminate u or v and redirect all incoming arcs to the other vertex.

- *Remove redundant tests* If nonterminal v has *low(v)=high(v)*, then eliminate v and redirect all incoming arcs to *low(v)*.

Starting with a binary decision diagram satisfying the ordering property, the canonical form is obtained by applying the transformation rules until the size of the diagram can no longer be reduced. If OBDDs are used as a canonical form for boolean functions, then checking equivalence is reduced to checking isomorphism between binary decision diagrams. Similarly, satisfiability can be determined by checking equivalence to the trivial OBDD that consists of only one terminal labeled by 0.

OBDDs are extremely useful for obtaining concise representations of relations over finite domains. We will see later how to use such representations to describe Kripke structures and to analyze them. If Q is an n-ary relation over $\{0,1\}$, then Q can be represented by the OBDD for its characteristic function

$f_Q(x_1,\ldots,x_n)$ =1 iff $Q(x_1,\ldots,x_n)$.

Otherwise, let Q be an n-ary relation over the finite domain D. Without loss of generality we assume that D has 2^m elements for some $m > 1$. In order to represent Q as an OBDD, we encode elements of D, using a bijection ϕ: $\{0,1\}^m \to$ D that maps each boolean vector of lenth m to an element of D. Using the encoding ϕ, we construct a boolean relation Q of arity $m \times n$ according to the following rule:

$\hat{Q}(\bar{x}_1,\ldots,\bar{x}_n)= Q(\phi(\bar{x}_1),\ldots,\phi(\bar{x}_n))$

where $\bar{x}_i$ is a vector of m boolean variables that encodes the variable x_i, which takes values in D. Qcan now be represented as the OBDD determined by the characteristic function $f_{\hat{Q}}$ of $\hat{Q}$. This technique can be easily extended to relations over different domains D_1, ..., D_n. Moreover, because sets can be viewed as unary relations, the same technique can be used to represent sets as OBDDs.

Consider now the Krikpe structure *M=(S, R, L)*. To represent this structure, we must describe the set S, the relation R, and the mapping L. For the set S, we first need to encode the states; for simplicity, we assume that there are exactly 2^m states. As above, we let ϕ: $\{0,1\}^m \to S$ be a function mapping boolean vectors to states. Since each assignment is the encoding of a state in S, the characteristic function representing S is the OBDD for 1. For the transition relation R, we use the same encoding for the states. We need two sets of boolean variables, one to represent the starting state and another to represent the final state of a transition. If the transition relation R is encoded by the boolean relation $\hat{R}(\bar{x}, \bar{x}')$, then R is represented by the characteristic function $f_{\hat{R}}$. Finally, we consider the mapping L. Although L is defined as a mapping from atomic propositions, it will be more convenient to consider it as a mapping from states to subsets of propositions to subsets of states. The atomic proposition p is mapped to the set of states that satisfy it: $\{s|p \in L(s)\}$. Call this set of states L_p; it can be represented using the encoding ϕ as above. We represent each atomic proposition separately in this way.

4.2 Symbolic Model Checking

Symbolic model checking is based on the manipulation of boolean formulas. Because the OBDDs represent sets of states and transitions, we need to operate on entire sets rather than on individual states and transitions. For this purpose, we use *fixpoint* characterization of the temporal logic operators. A set $S' \subseteq S$ is a fixpoint of a function τ: $\wp(S) \rightarrow \wp(S)$ if $\tau(S')$ $=S'$.

4.2.1 Fixpoint Representations

Let *M=(S, R, L)* be an arbitrary finite Kripke structure. The set $\wp(S)$ of all subsets of S forms a lattice under the set inclusion ordering. In this section, we will use $\wp(S)$ to denote the lattice. Each element S' of the lattice can also be thought of as a *predicate* on S, where the predicate is viewed as being true for exactly the states in S'. The least element in the lattice is the empty set, which we also refer to as False, and the greatest element in the lattice is the set S, which we sometimes write as True. A function that maps $\wp(S)$ to $\wp(S)$ will be called a predicate transformer. Let $\tau : \wp(S) \rightarrow \wp(S)$ be such a function; then

- 1. τ is *monotonic* provided that $P \subseteq Q$ implies $\tau(P) \subseteq \tau(Q)$;
- τ is $\cup -$ *continuous* provided that $P_1 \subseteq P_2 \subseteq \ldots$ implies $\tau(\cup_i P_i) = \cup_i \tau(P_i)$;

- τ is $\cap -$ *continuous* provided that $P_1 \supseteq P_2 \supseteq \dots$ implies $\tau(\cap_i P_i) = \cap_i \tau(P_i)$;

We write $\tau^i(Z)$ to denote i applications of τ to Z. More formally, $\tau^i(Z)$ is defined recursively by $\tau^0(Z)$=Z and $\tau^{i+1}(Z)$= $\tau(\tau^i(Z))$. A monotonic predicate transformer τ on $\wp(S)$ always has a least fixpoint, $\mu Z.\tau Z$, and a greatest fixpoint, $\nu Z.\ \tau Z$: $\mu Z.\tau Z$=$\cap$ $\{Z|\tau(Z) \subseteq Z\}$ whenever τ is monotonic, and $\mu Z.\tau Z$=$\cup_i \tau^i(False)$ whenever τ is also $\cup$-continuous. Similarly, $\nu Z.\tau Z$=$\cup$ $\{Z|\tau(Z) \supseteq Z\}$ whenever τ is monotonic, and $\nu Z.\tau(Z)$= $\cap_i \tau^i(True)$ whenever τ is also $\cap$-continuous.

Here are procedure for computing least fixpoints and greatest fixpoints:

```
function Lfp(Tau: PredicateTransformer): Predicate
    Q: = False;
    Q':= Tau(Q);
    while ($Q \neq Q'$) do
        $Q = Q'$;
        $Q' = Tau(Q')$;
    end while;
    return($Q$);
end function
```

Figure 4.2.1
Procedure for computing least fixpoints.

```
function Gfp(Tau: PredicateTransformer): Predicate
    Q: = True;
    Q':= Tau(Q);
    while ($Q \neq Q'$) do
        $Q = Q'$;
        $Q' = Tau(Q')$;
    end while;
    return($Q$);
end function
```

Figure 4.2.2
Procedure for computing least fixpoints.

If we identify each CTL formula f with the predicate $\{s|M, s \models f\}$ in $\wp(s)$ then each of the basic CTL operators may be characterized as a least or greatest fixpoint of an appropriate predicate transformer.

- **AF** f = μZ. $f_1\vee$ **AXZ**
- **EF** f = μZ. $f_1\vee$ **EXZ**
- **AG** f = νZ. $f_1\wedge$ **AXZ**
- **EG** f = νZ. $f_1\wedge$ **EXZ**
- **A**$[f_1$ **U**f_2]= μZ. $f_2\vee(f_1\wedge$ **AXZ**)
- **E**$[f_1$ **U**f_2]= μZ. $f_2\vee(f_1\wedge$ **EXZ**)
- **A**$[f_1$ **R**f_2]= νZ. $f_2\wedge(f_1\vee$ **AXZ**)
- **A**$[f_1$ **R**f_2]= νZ. $f_2\wedge(f_1\vee$ **EXZ**)

Intuitively,least fixpoints correspond to eventualities while greatest fixpoints correspond to properties that should hold forever. Thus, **AF**f_1 has a least fixpoint characterization and **EG**f_1 has a greatest fixpoint characterization.

4.2.2 The Symbolic Model Checking Algorithm for CTL

The symbolic model-checking algorithm is implemented by a procedure *Check* that takes the CTL formula to be checked as its argument and returns an OBDD that represents exactly those states of the system that satisfy the formula. Of course, the output of *Check* depends on the OBDD representation of the transition relation of the system being checked; this parameter is implicit in the discussion below. We define *Check* inductively over the structure of CTL formulas. If f is an atomic proposition a, then *Check(f)* is the OBDD representing the set of states satisfying a. If $f = f_1 \wedge f_2$ or $f = \neg f_1$, then *Check(f)* will be easily obtained according to *Check*(f_1) and *Check*(f_2). Formulas of the form **EX** f, **E**$[f$ **U**$g]$, and **EG**f are handled by the procedures:

Check(**EX** f) = *CheckEX(Check(f))*, *Check*($\mathbf{E}$ [f $\mathbf{U}g$] = *CheckEU(Check(f), Check(g))*,

Check($\mathbf{EG}$ f) = *CheckEG(check(f))*.
Notice that these intermediate procedures take OBDDs as their arguments, whereas *Check* takes a CTL formula as its argument. The cases of CTL fomulas of the form $f \vee g$ or $\neg f$ are handled using the standard algorithm for computing boolean connectives with OBDDs. Because the other temporal operators can all be rewritten using just the ones above, this definition of *Check* covers all CTL formulas.

The procedure for *CheckEX* is straightforward in that the formula $\mathbf{EX}\ f$ is true in a state if the state has a successor in which f is true.

CheckEX $(f(\bar{v}))$=$\exists \bar{v}'\ [f(\bar{v}') \wedge R(\bar{v}, \bar{v}')]$.

where $R(\bar{v}, \bar{v}')$ is the OBDD representation of the transition relation. If we have OBDs for f and R, then we can compute an OBDD for

$\exists \bar{v}'\ [f(\bar{v}') \wedge R(\bar{v}, \bar{v}')]$.

by using the operation of QBF (omitted here).

The procedure for *CheckEU* is based on the least fixpoint characterization for the CTL operator $\mathbf{EU}$:

$\mathbf{E}\ [f_1\ \mathbf{U} f_2] = \mu \mathrm{Z}.\ f_2 \vee (f_1 \wedge\ \mathbf{EX}\mathrm{Z})$.
We use the function Lfp to compute a sequence of approximations

$Q_0, Q_1, \ldots, Q_i, \ldots$

that converges to $\mathbf{E}$ [f $\mathbf{U}g$] in a finite number of steps. If we have OBDDs for *f*, *g*, and the current approximation Q_i , then we can compute an OBDD for the next approximation Q_i. Because OBDDs provide a canonical from of boolean functions, it is easy to test for LFP terminates. The set of states corresponding to $\mathbf{E}$ [f $\mathbf{U}g$]will be represented by the OBDD for Q_i.

CheckEG is similar. In this cade the procedure is based on the greatest fixpoint characterization for the CTL operator $\mathbf{EG}$:

$\mathbf{EG}\ f = \nu \mathrm{Z}.\ f_1 \wedge\ \mathbf{EX}\mathrm{Z}$

If we have an OBD for f , then the function GFP can be used to compute an OBDD representation for the set of states that satisfy $\mathbf{EG}$ *f*.

5 Conclusion

As the title shows, I want to give an overview of model checking. However, for the lacking of time, I haven't done a satisfiable work. I will indulge myself to the model checking in a coming long term.

References

[1] Edmund M. Clarke, Jr., Oma Grumberg, and Doron A. Peled. *Model Checking*, MIT Press, 1999

[2] E.M. Clarke, E.A. Emerson, and A.P.Sistla. Automatic verifivation of finite-state concurrent systems using branching time temporal logic specifications. In *Proceedings of the 10th Annual ACM Symposium on Principles of Programming Language, January 1983.*