Model Checking with Multi-Valued Temporal Logics

Marsha Chechik Steve Easterbrook Benet Devereux
Department of Computer Science
University of Toronto
6, King's College Rd, Toronto, Canada M5S 3H5
{chechik,sme,benet}@cs.toronto.edu

Abstract

Multi-valued logics support the explicit modeling of uncertainty and disagreement by allowing additional truth values in the logic. Such logics can be used for verification of dynamic properties of systems where complete, agreed upon models of the system are not available. This paper presents a symbolic model checker for multi-valued temporal logics. The model checker works for any multi-valued logic whose truth values form a quasi-boolean lattice. Our models are generalized Kripke structures, where both atomic propositions and transitions between states may take any of the truth values of a given multi-valued logic. Properties to be model checked are expressed in CTL, generalized with a multi-valued semantics. The design of the model checker is based on the use of MDDs, a multi-valued extension of Binary Decision Diagrams.

1. Introduction

In the past few years, *model checking* [10] has become established as one of the most effective automated techniques for analyzing correctness of software and hardware designs. A model checker checks a finite-state system against a correctness property expressed in a propositional temporal logic such as LTL or CTL. These logics can express safety (e.g. "No two processes can be in the critical section at the same time") and liveness (e.g. "Every job sent to the printer will eventually print") properties. Model-checking has been effectively applied to reasoning about correctness of hardware, communication protocols, software requirements, etc. A number of industrial model checkers have been developed, including SPIN [16], SMV [18], and Murϕ [11].

Despite their variety, existing model-checkers are typically limited to reasoning in classical logic. However, there are a number of problems for which classical logic is insufficient. One of these is reasoning under *uncertainty*. This can occur either when complete information is not known or cannot be obtained (e.g., during requirements analysis), or when this information has been removed (abstraction). Classical model-checkers typically deal with uncertainty by creating extra states, one for each value of the unknown variable and each feasible combination of values of known variables. However, this approach adds significant extra complexity to the analysis.

Classical reasoning is also insufficient for models that contain *inconsistency*. Models may be inconsistent because they combine conflicting points of view, or because they contain components developed by different people [13]. Conventional reasoning systems cannot cope with inconsistency because the presence of a single contradiction results in trivialization — anything follows from $A \wedge \neg A$. Hence, faced with an inconsistent description and the need to perform automated reasoning, we must either discard information until consistency is achieved again, or adopt a non-classical logic [17].

Multi-valued logics provide a solution to both reasoning under uncertainty and under inconsistency. For example, we can use "unknown" and "no agreement" as logic values. In fact, model-checkers based on three-valued and four-valued logics have already been studied. For example, [5] used a three-valued logic for interpreting results of model-checking with abstract interpretation, whereas [15] used four-valued logics for reasoning about abstractions of detailed gate or switch-level designs of circuits.

For reasoning about dynamic properties of systems, we need to extend existing modal logics to the multi-valued case. Fitting [14] explores two different approaches for doing this: the first extends the interpretation of atomic formulae in each world to be multi-valued; the second also allows multi-valued accessibility relations between worlds. The latter approach is more general, and can readily be applied to the temporal logics used in model checking [9].

We use different multi-valued logics to support different types of analysis. For example, to model information from multiple sources, we may wish to keep track of the origin of each piece of information, or just the majority vote, etc. Thus, rather than restricting ourselves to any particular multi-valued logic, our approach is to extend classical symbolic model-checking to arbitrary multi-valued logics, as long as conjunction, disjunction and negation of the logical values are well defined. The model, together with the description of the desired multi-valued logic and the set of correctness properties, expressed in CTL, become input to our model-checker, χchek. χchek returns the truth value(s) that each property has in the initial state(s).

The paper is organized as follows. Section 2 defines the quasi-boolean logics we use in this work, and shows how we specify each logic as a lattice. Section 3 introduces CTL model checking. Section 4 describes our multi-valued generalization of CTL model checking. Section 5 outlines the implementation of our model checker. We conclude the paper with a summary of results in section 6.

2. Quasi-Boolean Logics

Our approach to modeling makes use of an entire family of multi-valued logics. Since our goal is model-checking as opposed to theorem proving, we do not need to give a complete axiomatization for each logic. Rather, we simply give a semantics by defining conjunction, disjunction and negation on the truth values of the logic, and restrict ourselves to those logics where these operations are well-defined, and where conjunction and disjunction are idempotent, commutative, etc. Such properties are guaranteed by ensuring that the truth values of the logic form a finite, quasi-boolean lattice. In this section we give a brief introduction to quasi-boolean lattices.

Definition 1. *A* lattice *is a partial order ($\mathcal{L}$, $\sqsubseteq$) for which a unique greatest lower bound and least upper bound, denoted $a \sqcap b$ and $a \sqcup b$, respectively, exist for each pair of elements (a, b).*

$a \sqcap b$ and $a \sqcup b$ are referred to as *meet* and *join*, respectively. $a \sqsubseteq b$ means that "b is at least as true as a". The following properties hold of lattices:

$$\begin{array}{rcll} a \sqcup a & = & a & \\ a \sqcap a & = & a & \text{(idempotence)} \\ a \sqcup b & = & b \sqcup a & \\ a \sqcap b & = & b \sqcap a & \text{(commutativity)} \\ a \sqcup (b \sqcup c) & = & (a \sqcup b) \sqcup c & \\ a \sqcap (b \sqcap c) & = & (a \sqcap b) \sqcap c & \text{(associativity)} \end{array}$$

Definition 2. *A lattice is* complete *if it includes a meet and a join for for every subset of its elements. Every complete lattice has a top ($\top$) and a bottom ($\bot$).*

$$\begin{array}{rcll} \bot & = & \sqcap \mathcal{L} & (\bot \text{ characterization}) \\ \top & = & \sqcup \mathcal{L} & (\top \text{ characterization}) \end{array}$$

We only use lattices that have a finite number of elements. Every finite lattice is complete.

Definition 3. *A finite lattice ($\mathcal{L}$, $\sqsubseteq$) is* quasi-boolean *[2] (also called* De Morgan *[12]) if there exists a unary operator $\neg$ defined for it, with the following properties (a, b are elements of $\mathcal{L}$):*

$$\begin{array}{rcll} \neg(a \sqcap b) & = & \neg a \sqcup \neg b & \text{(De Morgan)} \\ \neg(a \sqcup b) & = & \neg a \sqcap \neg b & \\ a \sqsubseteq b & \Leftrightarrow & \neg a \sqsupseteq \neg b & (\neg \text{ antimonotonic}) \\ \neg\neg a & = & a & (\neg \text{ involution}) \end{array}$$

Thus, $\neg a$ is a quasi-complement *of a.*

The family of multi-valued logics we use are exactly those logics whose truth values form a quasi-boolean lattice. Meet and join in the lattice of truth values define conjunction and disjunction respectively, and we assume that an appropriate negation operation is defined that has the properties required by Definition 3. The identification of a suitable negation operator is greatly simplified by the observation that quasi-boolean lattices are symmetric about their horizontal axes:

Definition 4. *A lattice ($\mathcal{L}$, $\sqsubseteq$) is* horizontally-symmetric *if there exists a bijective function $H : \mathcal{L} \to \mathcal{L}$ such that for every pair $a, b \in \mathcal{L}$,*

$$\begin{array}{rcll} a \sqsubseteq b & \Leftrightarrow & H(a) \sqsupseteq H(b) & (\text{order}-\text{embedding}) \\ H(H(a)) & = & a & \text{(H involution)} \end{array}$$

Theorem 1. [9] *Horizontal symmetry is a necessary and sufficient condition for a lattice to be quasi-boolean with $\neg a = H(a)$ for each element of the lattice.*

The negation of each element is then defined as its image through horizontal symmetry[1]. Finally, we define an operator $\to$ as follows:

$$a \to b \equiv \neg a \sqcup b \qquad (\text{definition of } \to)$$

3. CTL Model-Checking

CTL model-checking is an automatic technique for verifying properties expressed in a propositional branching-time temporal logic called *Computational Tree Logic* (CTL) [10]. The system is defined by a Kripke structure, and properties are evaluated on a tree of infinite computations produced by the model of the system. The standard notation $M, s \models P$ indicates that a formula P holds in a state s of a model M. If a formula holds in the initial state, it is considered to hold in the model.

A Kripke structure consists of a set of states S, a transition relation $R \subseteq S \times S$, an initial state $s_0 \in S$, a set of atomic propositions A, and a labeling function $L : S \to \mathcal{P}(A)$. R must be total, i.e, $\forall s \in S,\ \exists t \in S : (s,t) \in R$. If a state s_n has no successors, we add a self-loop to it, so that $(s_n, s_n) \in R$. For each $s \in S$, the labeling function provides a list of atomic propositions which are *True* in this state.

[1]Note that we can still choose how to negate any elements that fall on the axis of symmetry.

CTL is defined as follows:

1. Every atomic proposition $a \in A$ is a CTL formula.
2. If φ and ψ are CTL formulae, then so are $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $EX\varphi$, $AX\varphi$, $EF\varphi$, $AF\varphi$, $EG\varphi$, $AG\varphi$, $E[\varphi U \psi]$, $A[\varphi U \psi]$.

The logic connectives $\neg$, $\wedge$ and $\vee$ have the usual meanings. The temporal quantifiers have two components: A and E quantify over paths, while X, F, U and G indicate "next state", "eventually (*future*)", "until", and "always (*globally*)", respectively. Hence, $AX\varphi$ is true in state s if φ is true in the *next* state on *all paths* from s. $E[\varphi U \psi]$ is true in state s if *there exists* a path from s on which φ is true at every step *until* ψ becomes true. Formally,

$$
\begin{array}{rcl}
M, s_0 \models a & \text{iff} & a \in L(s_0) \\
M, s_0 \models \neg\varphi & \text{iff} & M, s_0 \not\models \varphi \\
M, s_0 \models \varphi \wedge \psi & \text{iff} & M, s_0 \models \varphi \ \wedge \ M, s_0 \models \psi \\
M, s_0 \models \varphi \vee \psi & \text{iff} & M, s_0 \models \varphi \ \vee \ M, s_0 \models \psi \\
M, s_0 \models EX\varphi & \text{iff} & \exists t \in S,\ (s_0, t) \in R \ \wedge \ M, t \models \varphi \\
M, s_0 \models AX\varphi & \text{iff} & \forall t \in S,\ (s_0, t) \in R \rightarrow M, t \models \varphi \\
M, s_0 \models E[\varphi U \psi] & \text{iff} & \exists \text{ some path } s_0, s_1, \ldots, \text{ s.t.} \\
 & & \exists i,\ i \geq 0 \ \wedge \ M, s_i \models \psi \ \wedge \\
 & & \forall j,\ 0 \leq j < i \rightarrow M, s_j \models \varphi \\
M, s_0 \models A[\varphi U \psi] & \text{iff} & \text{for every path } s_0, s_1, \ldots, \\
 & & \exists i,\ i \geq 0 \ \wedge \ M, s_i \models \psi \ \wedge \\
 & & \forall j,\ 0 \leq j < i \rightarrow M, s_j \models \varphi.
\end{array}
$$

where the remaining operators are defined as follows:

$$
\begin{array}{rclcrcl}
AF(\varphi) & \equiv & A[\top U \varphi] & & EF(\varphi) & \equiv & E[\top U \varphi] \\
AG(\varphi) & \equiv & \neg EF(\neg\varphi) & & EG(\varphi) & \equiv & \neg AF(\neg\varphi)
\end{array}
$$

Definitions of AF and EF indicate that we are using a "strong until", that is, $E[\varphi U \psi]$ and $A[\varphi U \psi]$ are true only if ψ eventually occurs.

4. Multi-Valued Model Checking Semantics

In this section we give the semantics used in our multi-valued CTL model checker [9].

4.1. Defining the Model

A state machine M is a *multi-valued Kripke (χKripke) structure* if $M = (S, S_0, R, I, A, L)$, where:

- L is a quasi-boolean logic given as a lattice $(\mathcal{L}, \sqsubseteq)$.
- A is a (finite) set of atomic propositions
- S is a (finite) set of states. States are not explicitly labeled – each state is uniquely identified by its variable/value mapping. Thus, two states cannot have the same mapping.
- $S_0 \subseteq S$ is the non-empty set of initial states.
- Each transition (s, t) in M has a logical value in $\mathcal{L}$. Thus, $R : S \times S \rightarrow \mathcal{L}$ is a total function assigning a truth value from the logic L to each possible transition between states, including self-loops. Note that a χKripke structure is a completely connected graph. Furthermore, each state must have at least one non-false transition coming out of it.
- $I : S \times A \rightarrow \mathcal{L}$ is a total function that maps a state s and an atomic proposition (variable) a to a truth value ℓ of the logic. For simplicity we assume that all our variables are of the same type, ranging over the values of the logic. For a given variable a, we will write I as $I_a : S \rightarrow \mathcal{L}$.

$$
\begin{array}{rcll}
\neg AX\varphi & = & EX(\neg\varphi) & (\neg \text{ "}\texttt{next}\text{"}) \\
A[\bot U \varphi] & = & E[\bot U \varphi] = \varphi & (\bot \text{ "}\texttt{until}\text{"}) \\
A[\varphi U \psi] & = & \psi \vee (\varphi \wedge AXA[\varphi U \psi] & \\
 & & \quad \wedge EXA[\varphi U \psi]) & (AU \text{ fixpoint}) \\
E[\varphi U \psi] & = & \psi \vee (\varphi \wedge EXE[\varphi U \psi]) & (EU \text{ fixpoint})
\end{array}
$$

Figure 1. Properties of CTL operators.

For symbolic model checking, we compute *partitions* of the state space w.r.t. a variable a using $I_a^{-1} : \mathcal{L} \rightarrow 2^S$. A partition has the following properties:

$$
\begin{array}{ll}
\forall a \in A, \forall \ell_i, \ell_j \in \mathcal{L} \ : & \\
\quad i \neq j \rightarrow (I_a^{-1}(\ell_i) \cap I_a^{-1}(\ell_j) = \emptyset) & \text{(disjointness)} \\
\forall a \in A, \forall s \in S, \exists \ell \in \mathcal{L} \ : \ s \in I_a^{-1}(\ell) & \text{(cover)}
\end{array}
$$

4.2. Multi-Valued CTL

Here we give semantics of CTL operators on a χKripke structure M over a quasi-boolean logic L. We will refer to this language as *multi-valued CTL, or χCTL*. L is described by a finite, quasi-boolean lattice $(\mathcal{L}, \sqsubseteq)$, and thus the conjunction $\sqcap$, disjunction $\sqcup$ and negation $\neg$ operations are available. In extending the CTL operators, we want to ensure that the expected CTL properties, given in Figure 1, are preserved. Note that the AU fixpoint includes an additional conjunct, $EXA[fUg]$. This preserves a "strong until" semantics for states that have no outgoing $\top$ transitions [4].

We first extend the domain of the interpretation function I to any CTL formula φ, using $P_\varphi(s)$ to denote the truth value that formula φ takes in state s. If $s \in S$ is a state, $a \in A$ is a variable, and φ and ψ are CTL formulae:

$$
\begin{array}{rclcrcl}
P_a(s) & \equiv & I(s, a) & & P_{\varphi \wedge \psi}(s) & \equiv & P_\varphi(s) \wedge P_\psi(s) \\
P_{\neg\varphi}(s) & \equiv & \neg P_\varphi(s) & & P_{\varphi \vee \psi}(s) & \equiv & P_\varphi(s) \vee P_\psi(s)
\end{array}
$$

We proceed by defining EX. In standard CTL, this operator is defined using the existential quantification over next states. We define quantification for our multi-valued logics using conjunction and disjunction for universal and existential quantification, respectively. This treatment of quantification is standard [1, 19]. EX is defined by:

$$
P_{EX\varphi}(s) \quad \equiv \quad \bigvee_{t \in S} (R(s, t) \wedge P_\varphi(t))
$$

The definitions of AU, EU and AX are given using the properties in Figure 1:

$$
\begin{array}{rcl}
P_{AX\varphi}(s) & \equiv & \neg P_{EX\neg\varphi}(s) \\
 & = & \bigwedge_{t \in S} (R(s, t) \rightarrow P_\varphi(t)) \\
P_{E[\varphi U \psi]}(s) & \equiv & P_\psi(s) \vee (P_\varphi(s) \ \wedge \ P_{EXE[\varphi U \psi]}(s)) \\
P_{A[\varphi U \psi]}(s) & \equiv & P_\psi(s) \vee (P_\varphi(s) \ \wedge \ P_{AXA[\varphi U \psi]}(s) \\
 & & \wedge\, P_{EXA[\varphi U \psi]}(s))
\end{array}
$$

The remaining CTL operators, $AF(\varphi)$, $EF(\varphi)$, $AG(\varphi)$, $EG(\varphi)$ are the abbreviations for $A[\top U\varphi]$, $E[\top U\varphi]$, $\neg EF(\neg\varphi)$, $\neg AF(\neg\varphi)$, respectively.

5. Symbolic Model-Checker

In this section we describe the design of our symbolic multi-valued model checker, χchek. χchek takes as input a model M taking its variable and transition values from a lattice $\mathcal{L}$, and a χCTL formula φ. It produces as output a total mapping from $\mathcal{L}$ to the set S of states, indicating in which states φ takes each value ℓ. This is simply P_φ^{-1}, the inverse of the valuation function defined in section 4.2; and thus, the task of the model checker is to compute P_φ given the transition function R.

Since states are assignments of values to the variables, an arbitrary ordering imposed on A allows us to consider a state as a vector in $\mathcal{L}^n$, where $n = |A|$. Hence P_φ and R are functions of type $\mathcal{L}^n \to \mathcal{L}$ and $\mathcal{L}^{2n} \to \mathcal{L}$ respectively. Such functions are represented within the model checker by multi-valued decision diagrams (MDDs), a multi-valued extension of the binary decision diagrams (BDDs) [3].

5.1. MDDs

MDDs [21] have been extensively studied, mostly in the field of circuit design. The logics used in that literature are given by total orders (such as the integers modulo n) rather than quasi-boolean lattices, but this is a minor difference. Also, as far as we know, they have not been used in formal verification before. For brevity, we illustrate MDDs with an example rather than giving a complete theoretical treatment.

Definition 5. *[21] Given a finite domain D, the generalized Shannon expansion of a function $f : D^n \to D$, with respect to the first variable in the ordering, is*

$$f(a_0, a_1, \ldots, a_{n-1}) \to f_0(a_1, \ldots, a_{n-1}), \ldots, f_{|D|-1}(a_1, \ldots, a_{n-1})$$

where $f_i = f[a_0/d_i]$, the function obtained by substituting the literal $d_i \in D$ for a_0 in f. These functions are called cofactors.

Definition 6. *Assuming a finite set D, and an ordered set of variables A, multi-valued decision diagram (MDD) is a tuple $(V, E, \mathsf{var}, \mathsf{child}, \mathsf{image}, \mathsf{value})$ where*

- *$V = V_t \cup V_n$ is a set of nodes, where V_t and V_n indicate a set of terminal and non-terminal nodes, respectively;*
- *$E \subseteq V \times V$ is a set of directed edges;*
- *$\mathsf{var} : V_n \to A$ is a variable labeling function.*
- *$\mathsf{child} : V_n \to D \to V$ is an indexed successor function for nonterminal nodes;*
- *$\mathsf{image} : V \to 2^D$ is a total function that maps a node to a set of values reachable from it;*
- *$\mathsf{value} : V_t \to D$ is a total function that maps each terminal node to a logical value.*

Although D may be any finite set, for our purposes we are interested only in lattices; so instead of D we will refer to elements of the finite lattice $(\mathcal{L}, \sqsubseteq)$ modeling a logic.

For example, consider the function $f = x_1 \wedge x_2$, with $\ell_0 = \mathrm{F}, \ell_1 = \mathrm{M}, \ell_2 = \mathrm{T}$. The MDD built from this expression, and its lattice, are shown in Figure 2. The diagram is constructed by Shannon expansion, first with respect to x_1, and then (for each cofactor of f) with respect to x_2. The dashed arrows indicate f and its cofactors, and also the cofactors of the cofactors.

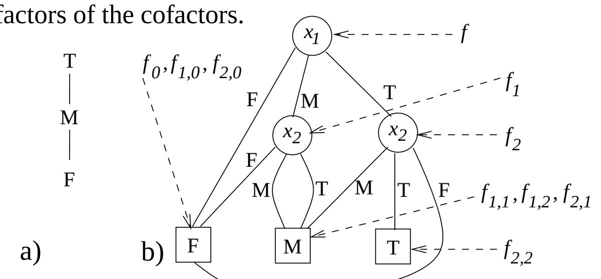

Figure 2. (a) A three-valued lattice. (b) The MDD for $f = x_1 \wedge x_2$ in this lattice.

Note that in general we do not distinguish between a single node in an MDD and the subgraph rooted there, referring to both indiscriminately as u. Let $\vec{s} = (s_0, \ldots, s_{n-1})$. The function computed by an MDD is denoted $f^u : \mathcal{L}^n \to \mathcal{L}$, and is defined recursively as follows:

$$\begin{array}{ll} \text{if } u \in V_t,\ f^u(\vec{s}) = \mathsf{value}(u) & \text{(terminal constants)} \\ \text{if } u \in V_n,\ f^u(\vec{s}) = f^{\mathsf{child}_{s_i}(u)}(s_0, \ldots, s_{i-1}, s_{i+1}, \ldots, s_{n-1}) & \\ \quad \text{where } a_i = \mathsf{var}(u),\ \vec{s} \in \mathcal{L}^n & \text{(cofactor expansion)} \end{array}$$

Efficiency of decision diagrams, binary or multi-valued, comes from the properties of reducedness and orderedness. Orderedness is also required for termination of many algorithms on the diagrams. We perform various logical operations on the functions represented by MDDs: equality, conjunction, disjunction, negation, and existential quantification. MDDs have the same useful property as BDDs: given a variable ordering, there is precisely one MDD representation of a function. This allows for constant-time checking of function equality.

Theorem 2. Canonicity *[21] For any finite lattice (or finite set) $\mathcal{L}$, any nonnegative integer n, and any function $f : \mathcal{L}^n \to \mathcal{L}$, there is exactly one reduced ordered MDD u such that $f^u = f(a_0, \ldots, a_{n-1})$.*

In the boolean case, BDDs allow for constant-time existential quantification, since any node which is not a constant $\perp$ is satisfiable. In order to implement multi-valued quantification efficiently, we introduce the image attribute of MDD nodes, which stores the possible outputs of functions. The following properties hold for image:

$$u \in V_t \quad \Rightarrow \quad \mathsf{image}(u) = \{\mathsf{value}(u)\}$$
$$u \in V_n \quad \Rightarrow \quad \mathsf{image}(u) = \bigcup_{\ell \in \mathcal{L}} \mathsf{image}(\mathsf{child}_\ell(u))$$

Definition 7. *A function f is ℓ-*satisfiable *if some input yields ℓ as an output, or, equivalently, $f^{-1}(\ell) \neq \emptyset$:*

$$(f^u)^{-1}(\ell) \neq \emptyset \;\Leftrightarrow\; \ell \in \mathsf{image}(u)$$
$$(\exists \vec{s} \in \mathcal{L}^n : f^u(\vec{s})) \;=\; (\bigvee_{\vec{s} \in \mathcal{L}^n} f^u(\vec{s})) \;=\; (\bigvee_{\ell \in \mathsf{image}(u)} \ell)$$

The second property is called existential quantification.

To demonstrate how existential quantification works, we refer to the example in Figure 2, and compute $\exists x_2 : x_1 \wedge x_2$. There are two nodes labeled with x_2 to be dealt with. By inspection we see that $\mathsf{image}(f_1) = \{\mathrm{F}, \mathrm{M}\}$ and $\mathsf{image}(f_2) = \{\mathrm{F}, \mathrm{M}, \mathrm{T}\}$. So f_1 is replaced with the terminal node $\mathrm{F} \vee \mathrm{M} = \mathrm{M}$, and f_2 with the terminal node $\mathrm{F} \vee \mathrm{M} \vee \mathrm{T} = \mathrm{T}$.

Algorithms for manipulating BDDs are extensible to the multi-valued case, provided they do not use optimizations that depend on a two-valued boolean logic (e.g. complemented edges [20]). The differences are discussed in [6]. The public methods required for model checking are: `Build`, to construct an MDD based on a function table; `Apply`, to compute $\wedge$, $\vee$ and $\neg$ of MDDs; `Quantify`, to existentially quantify over the primed variables; and `AllSat` to retrieve the computed partition $P_\varphi^{-1}(\mathcal{L})$. `Build` ensures orderedness of MDDs while they are being constructed, and `Apply` preserves it. An additional function, `Prime`, primes all of the variables in an MDD. Table 1 shows the running times of MDD operations used by the model checker in terms of the size of the MDD.

5.2. The Model Checker

The model checking algorithm is given in Figure 3. The function $\mathtt{EX}(P_\varphi)$ computes $P_{EX\varphi}$ symbolically; `QUntil` carries out the fixed-point computation of both AU and EU. $AX\varphi$ is computed as $\neg EX \neg \varphi$. EG, AG, EF, and AF are not shown in this Figure, but could be added as cases and defined in terms of calls to `EX`, `QUntil`, and `Apply`.

The running time of χchek is dominated by the fixpoint computation of `QUntil`. The proof of termination of this algorithm is based on each step of `QUntil` being a monotonic lattice operator (for the proof, see [9]). The total number of steps is bounded above by $|\mathcal{L}|^n \times h$ (h is the height of the lattice $\mathcal{L}$), and the time of each step is dominated by the time to compute the EXTerm and AXTerm, which is $O(|\mathcal{L}|^{2n})$; so the worst-case running time for χchek is $O(|\mathcal{L}|^{3n} \times h)$, where h is the height of the lattice. Experimental results [6] suggest that in the average case, each step's running time is $O(|\mathcal{L}|^{2n-2})$, for an average termination time of $O(|\mathcal{L}|^{3n-2} \times h \times |p|)$, where $|p|$ is the size of the χCTL formula.

At first glance, MDDs appear to perform significantly worse than BDDs ($O(|\mathcal{L}|^n)$ versus $O(2^n)$ in the worst case).

```
function EX($P_\varphi$)
  return Quantify(Apply($\wedge$, $R$, Prime($P_\varphi$)), $n$)

function QUntil(quantifier, $P_\varphi$, $P_\psi$)
  $QU_0 = P_\psi$
  repeat
    if (quantifier is A)
      $\mathrm{AXTerm}_{i+1}$ := Apply($\neg$, EX(Apply($\neg$, $QU_i$)))
      $\mathrm{EXTerm}_{i+1}$ := EX($QU_i$))
    else
      $\mathrm{AXTerm}_{i+1}$ := $P_\varphi$
      $\mathrm{EXTerm}_{i+1}$ := EX(Apply($\neg$, $QU_i$)))
    $QU_{i+1}$ := Apply($\vee$, $P_\psi$, (Apply($\wedge$, $P_\varphi$,
                  Apply($\wedge$, $\mathrm{EXTerm}_{i+1}$, $\mathrm{AXTerm}_{i+1}$))))
  until $QU_{i+1} = QU_i$
  return $QU_n$

procedure MC($p$, $M$)
Case
  $p \in A$:            return Build($p$)
  $p = \neg\varphi$:           return Apply($\neg$, MC($\varphi$, $M$))
  $p = \varphi \wedge \psi$:        return Apply($\wedge$, MC($\varphi$, $M$), MC($\psi$, $M$))
  $p = \varphi \vee \psi$:        return Apply($\vee$, MC($\varphi$, $M$), MC($\psi$, $M$));
  $p = EX\varphi$:         return EX(MC($\varphi$, $M$))
  $p = AX\varphi$:         return Apply($\neg$, EX(Apply($\neg$, MC($M$, $\varphi$))))
  $p = E[\varphi U \psi]$:    return QUntil(E, MC($\varphi$, $M$), MC($\psi$, $M$))
  $p = A[\varphi U \psi]$:    return QUntil(A, MC($\varphi$, $M$), MC($\psi$, $M$))
```

Figure 3. The multi-valued symbolic model checking algorithm.

However, our multi-valued logics compactly represent incompleteness in a model. For example, suppose we have a model with n states and wish to differentiate between p of those states ($p \ll n$) by introducing an extra variable a. In classical model checking this uncertainty can only be handled by duplicating each of $n - p$ states (one for each value of a). In fact, most of these states are likely to be reachable; thus, the size of the state space nearly doubles. In the multi-valued case, the reachable state-space will increase at most by p states. Thus, we expect that often our model checker would perform as well as the classical one, and on some problems even better. Further, our use of multi-valued logics allow us to elegantly model "unknown" transitions and inconsistencies.

6. Conclusion and Future Work

Multi-valued logics can be useful for describing models that contain incomplete information or inconsistency. In this paper we presented an extension of classical CTL model checking to reasoning about arbitrary quasi-boolean logics. We also described an implementation of a symbolic

MDD Method	*Running Time*	*Notes*
$\texttt{MakeUnique}(\mathsf{var}, \mathsf{child})$	$O(1)$	Hash-table lookup.
$\texttt{Build}(f)$	$O(\lvert\mathcal{L}\rvert^n)$	O(size of the function table to convert to MDD).
$\texttt{Apply}(op, u_1, u_2)$	$O(\lvert u_1\rvert\lvert u_2\rvert)$	Implemented using dynamic programming. The worst-case is pairwise conjunction of every node in u_1 with every node in u_2.
$\texttt{Quantify}(u, i)$	$O(\lvert u\rvert)$	Depth-first traversal of the graph.
$\texttt{Prime}(u)$	$O(\lvert u\rvert)$	Same as above.

Table 1. MDD methods used for model checking, and their running times.

multi-valued model checker χchek.

We plan to extend the work presented here in a number of directions to ensure that χchek can be effectively applied to reasoning about non-trivial systems. For example, in this paper we concentrated our attention on a purely symbolic model checker. The union, intersection, and quantification were computed using MDD operations. Alternatively, one can build a table-driven model checker, where such operations are table lookups, with similar running times. However, lattice-theoretic results can be used to significantly optimize the table-driven model checker [7]. We are also working on generalizing our symbolic algorithm to verification of properties expressed in CTL*, and on automata-theoretic multi-valued model-checking [8].

Acknowledgements: *We thank members of the University of Toronto formal methods reading group, and in particular Ric Hehner, Albert Lai, Arie Gurfinkel, Victor Petrovykh and Christopher Thompson-Walsh. Financial support was provided by NSERC and CITO.*

References

[1] N. Belnap. "A Useful Four-Valued Logic". In Dunn and Epstein, editors, *Modern Uses of Multiple-Valued Logic*, pages 30–56. Reidel, 1977.

[2] L. Bolc and P. Borowik. *Many-Valued Logics*. Springer-Verlag, 1992.

[3] R. E. Bryant. "Symbolic Boolean manipulation with ordered binary-decision diagrams". *Computing Surveys*, 24(3):293–318, September 1992.

[4] T. Bultan, R. Gerber, and C. League. "Composite Model Checking: Verification with Type-Specific Symbolic Representations". *ACM Transactions on Software Engineering and Methodology*, 9(1):3–50, January 2000.

[5] M. Chechik. "On Interpreting Results of Model-Checking with Abstraction". CSRG Technical Report 417, University of Toronto, Department of Computer Science, September 2000.

[6] M. Chechik, B. Devereux, and S. Easterbrook. "Implementing a Multi-Valued Symbolic Model-Checker". In *Proceedings of TACAS'01*, April 2001.

[7] M. Chechik, B. Devereux, S. Easterbrook, A. Lai, and V. Petrovykh. "Efficient Multiple-Valued Model-Checking Using Lattice Representations". Submitted for publication, January 2001.

[8] M. Chechik, B. Devereux, and A. Gurfinkel. "Model-Checking Infinite State-Space Systems with Fine-Grained Abstractions Using SPIN". Submitted for publication, February 2001.

[9] M. Chechik, S. Easterbrook, and V. Petrovykh. "Model-Checking Over Multi-Valued Logics". In *Proceedings of FME'01*, March 2001.

[10] E. Clarke, E. Emerson, and A. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[11] D. Dill. "The Murϕ Verification System". In R. Alur and T. Henzinger, editors, *Computer-Aided Verification Computer*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393, New York, N.Y., 1996. Springer-Verlag.

[12] J. Dunn. "A Comparative Study of Various Model-Theoretic Treatments of Negation: A History of Formal Negation". In D. Gabbay and H. Wansing, editors, *What is Negation*. Kluwer Academic Publishers, 1999.

[13] S. Easterbrook and M. Chechik. "A Framework for Multi-Valued Reasoning over Inconsistent Viewpoints". In *Proceedings of International Conference on Software Engineering (ICSE'01)*, May 2001.

[14] M. Fitting. "Many-Valued Modal Logics". *Fundamenta Informaticae*, 15(3-4):335–350, 1991.

[15] S. Hazelhurst. *Compositional Model Checking of Partially Ordered State Spaces*. PhD thesis, Department of Computer Science, University of British Columbia, 1996.

[16] G. Holzmann. "The Model Checker SPIN". *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[17] A. Hunter. "Paraconsistent Logics". In D. Gabbay and P. Smets, editors, *Handbook of Defeasible Reasoning and Uncertain Information*, volume 2. Kluwer, 1998.

[18] K. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.

[19] H. Rasiowa. *An Algebraic Approach to Non-Classical Logics. Studies in Logic and the Foundations of Mathematics*. Amsterdam: North-Holland, 1978.

[20] F. Somenzi. "Binary Decision Diagrams". In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*, volume 173 of *NATO Science Series F: Computer and Systems Sciences*, pages 303–366. IOS Press, 1999.

[21] A. Srinivasan, T. Kam, S. Malik, and R. Brayton. "Algorithms for Discrete Function Manipulation". In *IEEE International Conference on Computer-Aided Design*, pages 92–95, 1990.