Implementing a Multi-valued Symbolic Model Checker

Marsha Chechik, Benet Devereux, and Steve Easterbrook

Department of Computer Science, University of Toronto,
Toronto, ON M5S 3G4, Canada.
{chechik,benet,sme}@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. In this paper, we present an implementation of 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. We describe MDDs and their use in the model checker. We also give its theoretical time complexity and some preliminary empirical performance data.

1 Introduction

Multi-valued logics provide an interesting alternative to classical boolean logic for modeling and reasoning about systems. By allowing additional truth values in the logic, they support the explicit modeling of uncertainty and disagreement. For these reasons, they have been explored for a variety of applications in databases [12], knowledge representation [13], machine learning [17], and circuit design [15].

A number of specific multi-valued logics have been proposed and studied. For example, Łukasiewicz [16] first introduced a three-valued logic to allow for propositions whose truth values are 'unknown', while Belnap [1] proposed a four-valued logic that also introduces the value 'both' (i.e. "true *and* false"), to handle inconsistent assertions in database systems. Each of these logics can be generalized to allow for different levels of uncertainty or disagreement. In practice, it is useful to be able to choose different multi-valued logics for different modeling tasks.

The motivations that led to the development of these logics clearly apply to the modeling of software behaviour, especially the exploratory modeling used in the early stage of requirements engineering and architectural design:

- We need to allow for *uncertainty* – for example, we may not yet know whether some behaviours should be possible;
- We need to allow for *disagreement* – for example, different stakeholders may disagree about how the systems should behave;
- We need to represent *relative importance* – for example, in the case where some behaviours are essential and others may or may not be implemented.

For reasoning about dynamic properties of systems, existing modal logics can be extended to the multi-valued case. Fitting [10] suggests 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 automated verification [6].

Some automated tools for reasoning with multi-valued logics exist. In particular, the work of Hähnle and others [14,19] has led to the development of several theorem-provers for first-order multi-valued logics. However, as yet the question of model checking for multi-valued modal logics has not been addressed.

In this paper we describe our implementation of a multi-valued symbolic CTL model checker, χchek. χchek is generalized for an entire family of multi-valued logics, known as the quasi-boolean logics. It takes as its input a description of a particular quasi-boolean logic, represented as a lattice of truth values, a state machine model, represented as a multi-valued Kripke structure, and a temporal logic property expressed in CTL. It returns the truth value that the property has in the initial state(s).

The paper is structured as follows. Section 2 motivates the work with an example of a multi-valued state machine model. Section 3 describes the family of quasi-boolean multi-valued logics, and shows how these are specified as lattices of truth values. Section 4 explains our approach, describing our multi-valued extension of Kripke structures and our multi-valued extension of CTL. Section 5 presents the design of the model checker and analyses its performance. Section 6 presents our conclusions.

2 Motivation

To motivate the development of our model checker, and to illustrate its application, we present an example state machine model expressed in a multi-valued logic. The model captures an early draft of the requirements for a simple coffee dispenser. We distinguish behaviours that *must* be true (are required), behaviours that *should* be true (are desired, but not required), behaviours that *should not* be true (are undesirable), and behaviours that *must not* be true (are prohibited). We use two types of unknown: *Don't Know* for things that will be controlled by the system, where we do not yet know what behaviours we want; and *Don't Care* for things that are controlled by the environment, where the value does not matter. We represent these six possibilities in a 6-valued logic, arranged as a lattice in Figure 1(a), using the partial order 'more true than'.

Figure 1(b) shows the model. Each variable is assigned a truth value in each state. Each transition between states is also labeled with a truth value. The coffee dispenser starts in state `OFF`. In this state, it is irrelevant whether there is a cup in the machine, so that variable has the value 'DC' ("don't care"). The specification team *have not yet decided* whether they need a power-saving standby mode. They model their uncertainty by including the state `IDLE`, but label the transitions into it 'S', indicating these *may be* desirable. They also use the value 'DK' ("Don't Know") for the state of the power in `IDLE`, and for the transition from `OFF` to `IDLE`. However, the transition from `OFF` to `READY` is labeled 'T', indicating that when the power is switched on, the machine *must* enter the `READY` state. From there, it *must* be able to deliver coffee, and it *should* then

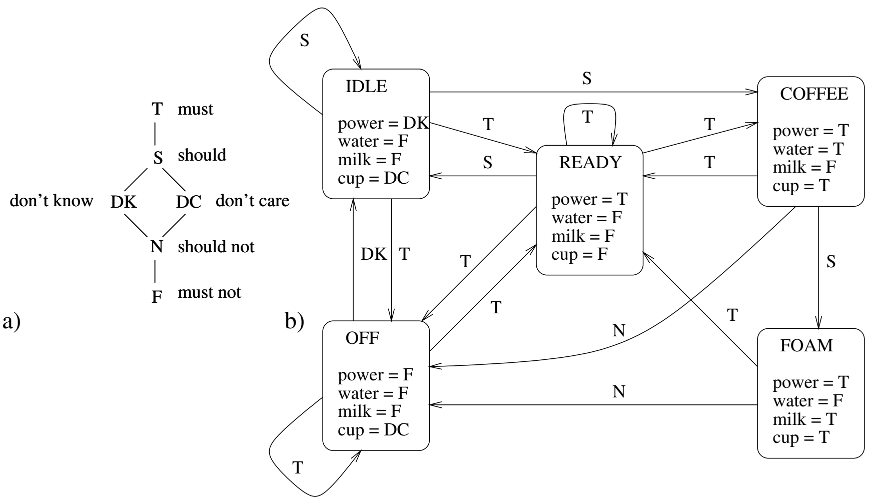

Fig. 1. (a) A lattice of truth values; (b) The coffee dispenser model that uses it.

be able to deliver foam. The transitions from `COFFEE` and `FOAM` to `OFF` are labeled 'N'. These are undesirable, but we cannot prohibit them because the machine has no direct control over the power supply. Note that by convention we omit all 'F' transitions. Hence there is an 'F' self-loop for `COFFEE` and `FOAM`, indicating we *must not* stay in either state, and an 'F' transition from `READY` to `FOAM`, indicating this *must not* occur.

We can now write properties that ought to be true of the model, even though it contains some uncertainties. For example:

1. The machine must always be able to make coffee.
2. It is desirable that the machine make foam.
3. Coffee cannot be dispensed if there is no cup.
4. Once coffee is dispensed, we cannot get coffee again until the cup is changed.

We formalize these properties in Section 4 and give results of model checking them on the coffee dispenser model in Table 1 of Section 5.

In this example, the use of a 6-valued logic allows us to distinguish two levels of priority for requirements, and two different types of unknown. We could choose different multi-valued logics if we wanted to distinguish further levels of priority, or different types of 'unknown'. We are also interested in modeling disagreement, and have developed a method for reasoning about whether disagreements between stakeholders' views affect various system properties. In [9] we outline a general framework for combining inconsistent state machine models into a single model using multi-valued logics to capture levels of (dis)agreement. We eventually plan to use the model checker described below as a negotiation tool for constructing and reasoning about such models.

3 Specifying the Logics

Our approach to modeling makes use of an entire family of multi-valued logics. Rather than giving a complete axiomatization for each logic, we simply give a semantics by defining conjunction, disjunction and negation on the truth values of the logic, and restrict ourselves to logics where these operations are well-defined, and satisfy commutativity, associativity etc. Such properties can be easily guaranteed if we require that the truth values of the logic form a lattice. In this section we describe the types of lattices used in our logics.

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. The partial order operation $a \sqsubseteq b$ means "b is at least as true as a". The following properties hold for all lattices:

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

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

$$\bot = \sqcap \mathcal{L} \quad (\bot \text{ characterization}) \qquad \top = \sqcup \mathcal{L} \quad (\top \text{ characterization})$$

For example, in the lattice of Figure 1(a), $\top$ is labeled 'T' and $\bot$ is labeled 'F'. We adopt the convention of labeling $\top$ and $\bot$ in this way in all our lattices. Also, 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] if there exists a unary operator* $\neg$ *defined for it, with the following properties* (a, b *are elements of* $\mathcal{L}$)*:*

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

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 operators, respectively, and we assume that an appropriate negation operation is defined with 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* $H : \mathcal{L} \rightarrow \mathcal{L}$ *such that for every pair* $a, b \in \mathcal{L}$*,*

$$a \sqsubseteq b \Leftrightarrow H(a) \sqsupseteq H(b) \quad (\text{order} - \text{embedding}) \quad H(H(a)) = a \quad (\text{H involution})$$

Theorem 1. *[6] 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]. For example, in Figure 1(a) we have ¬T=F, ¬S=N, ¬DK=DK, ¬DC=DC, etc. Finally, we define an operator $\rightarrow$ as follows:

$$a \rightarrow b \;\equiv\; \neg a \sqcup b \quad (\text{definition of } \rightarrow)$$

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

4 Multi-valued Model Checking

CTL model checking on two-valued logics was introduced by Clarke and his colleagues in [8]. CTL is a branching-time temporal logic that allows quantification over individual paths in a tree of computations exhibited by a model. There are five basic pairs of operators: AX and EX ("next"), AF and EF ("eventually" or "in the future"), AG and EG ("globally"), AU and EU ("until"), AR and ER ("release"). Models are represented as Kripke structures, which are finite-state machines that guarantee that there is a transition out of every state. See [7] for a detailed account of CTL model checking.

In this section we describe our multi-valued extension of Kripke structures, which we call χKripke structures, and we give the semantics of multi-valued CTL [6].

4.1 Defining the Model

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

- L is a quasi-boolean logic represented by a lattice ($\mathcal{L}$, $\sqsubseteq$).
- A is a (finite) set of atomic propositions, otherwise referred to as variables (e.g. `power` or `milk` in the example in Figure 1(b)).
- 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. However, we sometimes use state labels as a shorthand for the respective vector of values, as we did in the coffee dispenser example.
- $S_0 \subseteq S$ is the non-empty set of initial states.
- Each transition (s, t) in M has a logical value in $\mathcal{L}$. Thus, $R : S \times S \to \mathcal{L}$ 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. We also require that each state has at least one non-false transition coming out of it.
- $I : S \times A \to \mathcal{L}$ 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 $I_a : S \to \mathcal{L}$. For symbolic model checking, we compute *partitions* of the state space w.r.t. a variable a using $I_a^{-1} : \mathcal{L} \to 2^S$. A partition has the following properties:

$$\begin{array}{ll} \forall a \in A, \forall \ell_1, \ell_2 \in \mathcal{L} \; : \; \ell_1 \neq \ell_2 \Rightarrow (I_a^{-1}(\ell_1) \cap I_a^{-1}(\ell_2) = \emptyset) & \text{(disjointness)} \\ \forall a \in A, \forall s \in S, \exists \ell \in \mathcal{L} \; : \; s \in I_a^{-1}(\ell) & \text{(cover)} \end{array}$$

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* χ*CTL.* 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 2, are still preserved. Note that the AU fixpoint is somewhat unusual because it includes an additional conjunct,

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

Fig. 2. Properties of CTL operators.

$EXA[fUg]$. This additional term preserves a "strong until" semantics for states that have no outgoing T transitions [4].

We first give the semantics of the propositional operators. We extend the domain of the interpretation function I to any CTL formula φ. For a model M, we use $P_\varphi^M(s)$ to denote the truth value that formula φ takes in state s. We omit M if it is clear from context. If $s \in S$ is a state, $a \in A$ is a variable, and φ and ψ are CTL formulae:

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

We proceed by defining the EX operator. In standard CTL, EX is defined using existential quantification over next states. We extend the notion of quantification for multi-valued reasoning by using conjunction and disjunction for universal and existential quantification, respectively. This treatment of quantification is standard [1,18]. The semantics of the EX operator is

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

The definitions of AU, EU and AX are given using the properties in Figure 2:

$$\begin{array}{l}
P_{AX\varphi}(s) \equiv \neg P_{EX\neg\varphi}(s) \\
P_{E[\varphi U\psi]}(s) \equiv P_\psi(s) \vee (P_\varphi(s) \ \wedge \ P_{EXE[\varphi U\psi]}(s)) \\
P_{A[\varphi U\psi]}(s) \equiv P_\psi(s) \vee (P_\varphi(s) \ \wedge \ P_{AXA[\varphi U\psi]}(s) \ \wedge \ P_{EXA[\varphi U\psi]}(s))
\end{array}$$

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

The properties of the coffee dispenser in Figure 1(b), given in Section 2, can be formalized in χCTL as follows [2]:

1. $EF(\texttt{water})$	The expected answer is T.
2. $EF(\texttt{milk})$	The expected answer is S.
3. $AG(\texttt{water} \rightarrow \texttt{cup})$	The expected answer is T.
4. $AG(\texttt{water} \rightarrow AXA[\neg\texttt{water}\ \mathcal{W}\ (\neg\texttt{cup} \wedge \neg\texttt{water})])$	The expected answer is T.

5 Symbolic Multi-valued Model Checker

In this section we describe the implementation of our symbolic multi-valued model checker, χchek. The architecture of χchek is shown in Figure 3. χchek takes as input a model M with variable and transition values from a lattice $\mathcal{L}$, and a χCTL formula φ. It outputs a total mapping from $\mathcal{L}$ to the set S of states, indicating in which states φ takes

[2] We use the operator *while* defined as $A[x\ \mathcal{W}\ y] = \neg E[\neg y\ U\ (\neg x\ \wedge\ \neg y)]$.

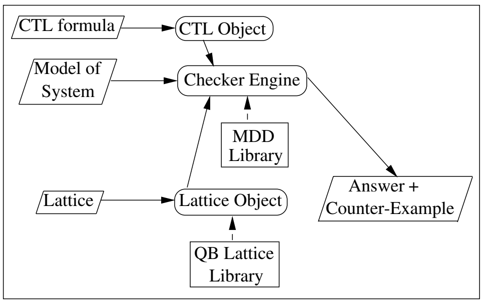

Fig. 3. Architecture of χchek.

each value ℓ. This is simply P_φ^{-1}, the inverse of the valuation function defined above. 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 can be thought of as 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].

As an example, consider the coffee dispenser shown in Figure 1(b). Using the variable ordering (`power`, `water`, `milk`, `cup`), the state labeled `COFFEE` is just the vector $\boldsymbol{s} =$ (T, T, F, T), the one labeled `FOAM` is $\boldsymbol{t} =$ (T, F, T, T), and the existence of a T-valued transition between them is expressed by the fact that R =(T, T, F, T, T, F, T, T)=T or, more compactly, $R(\boldsymbol{s}, \boldsymbol{t}) = \mathrm{T}$.

χchek uses two supplementary libraries: a library for handling quasi-boolean lattices and an MDD library. The former includes functions to compute unions and intersections of sets of logical values, determine whether given lattices have some desired properties, e.g., distributivity, and to support various lattice-based calculations. Our library is based on Freese's Lisp lattice library [11]. The MDD library is described below.

5.1 Data Structures

There is an extensive literature dealing with MDDs [21], mostly in the field of circuit design. To our knowledge, the logics used in that literature are given by total orders (such as the integers modulo n) and not by arbitrary quasi-boolean lattices, but we concede that this is a minor difference. Also, as far as we know, they have not been used in formal verification before, so for the purposes of this paper we will describe them briefly. We will assume a basic knowledge of BDDs [3].

The basic notion in the construction of binary decision diagrams is the Shannon expansion. A boolean function f of n variables can be expressed relative to a variable a_0, by computing f on $n-1$ variables with a_0 set to $\top$, and the same function with a_0 set to $\bot$. These functions are referred to as $f_\top$ and $f_\bot$, respectively. We write this expansion as $f(a_0, \ldots, a_{n-1}) \to f_\top(a_1, \ldots, \ldots, a_{n-1}), f_\bot(a_1, \ldots, a_{n-1})$ This notion is generalized as follows:

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*, 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.*

We describe constraints on the elements of an MDD below. 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.

Consider the function $f = x_1 \wedge x_2$, with $\ell_0 = \mathrm{F}, \ell_1 = \mathrm{M}, \ell_2 = \mathrm{T}$. The MDD for this expression is shown in Figure 4b. 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.

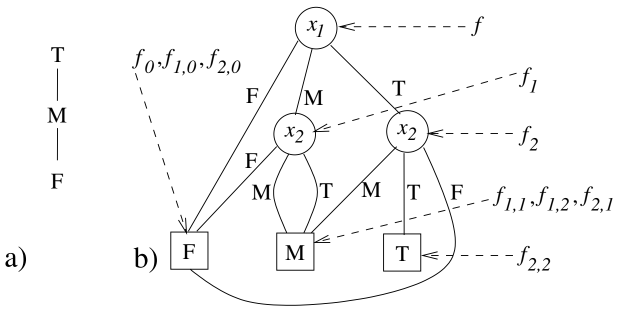

Fig. 4. (a) A three-valued lattice. (b) The MDD for $f = x_1 \wedge x_2$ in this lattice.

The following properties hold for all MDDs:

$$\forall u_0 \in V_n : \mathrm{out}(u_0) = |\mathcal{L}| \;\; \wedge \;\; \forall u_1 \in V_t : \mathrm{out}(u_1) = 0 \qquad \text{(semantics of nodes)}$$
$$\forall u_0, u_1 \in V, \exists \ell \in \mathcal{L} \; : \; (u_0, u_1) \in E \Rightarrow \mathsf{child}_\ell(u_0) = u_1 \qquad \text{(semantics of edges)}$$

where out(u) stands for the number of non-null children of u. Several further properties are required for the data structure to be usable:

$$\forall u_0, u_1 \in V_n : \; (u_0, u_1) \in E \; \wedge \; \mathsf{var}(u_0) = a_i \; \wedge \; \mathsf{var}(u_1) = a_j \Rightarrow i < j \quad \text{(orderedness)}$$
$$\forall u_0, u_1 \in V \; : \; f^{u_0} = f^{u_1} \Rightarrow u_0 = u_1 \quad \text{(reducedness)}$$
$$\forall u_0, u_1 \in V_n, \ell \in \mathcal{L} \; :$$
$$(\mathsf{var}(u_0) = \mathsf{var}(u_1)) \; \wedge \; (\mathsf{child}_\ell(u_0) = \mathsf{child}_\ell(u_1)) \Rightarrow u_0 = u_1 \quad \text{(uniqueness 1)}$$
$$\forall u_0, u_1 \in V_t \; : \; (\mathsf{value}(u_0) = \mathsf{value}(u_1)) \Rightarrow u_0 = u_1 \quad \text{(uniqueness 2)}$$

In general, the efficiency of decision diagrams, binary or multi-valued, comes from the properties of reducedness and orderedness (defined above). Orderedness is also required for termination of many algorithms on the diagrams. Uniqueness implies reducedness [21] – MDDs will be unique by construction, and thus reduced.

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. The function computed by an MDD is denoted $f^u : \mathcal{L}^n \to \mathcal{L}$, and is defined recursively as follows:

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

Consider the MDD in Figure 4. To compute $f = x_1 \wedge x_2$ with $x_1 = \mathrm{T}$ and $x_2 = \mathrm{F}$ using this diagram, we want to find $f(\boldsymbol{s})$ where $\boldsymbol{s} = (\mathrm{T}, \mathrm{M})$. We begin at the root node. Its var is x_1, so we pick out s_1, which is T, and descend to the node $\mathsf{child}_\mathrm{T}(f)$, indicated by the arrow to f_2 (which represents the function $\mathrm{T} \wedge x_2$). Now we compute $f_2(\mathrm{M})$ by choosing $\mathsf{child}_\mathrm{M}(f_2)$, which is a node in V_t, so we stop and return M. Thus, we conclude that $f(\mathrm{T}, \mathrm{F}) = \mathrm{M}$.

We will be calculating equality, conjunction, disjunction, negation, and existential quantification on the functions represented by MDDs. 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 $\bot$ 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:

$$\begin{array}{ll} u \in V_t \Rightarrow \mathsf{image}(u) = \{\mathsf{value}(u)\} & (\mathsf{image} \text{ property } 1) \\ u \in V_n \Rightarrow \mathsf{image}(u) = \bigcup_{\ell \in \mathcal{L}} \mathsf{image}(\mathsf{child}_\ell(u)) & (\mathsf{image} \text{ property } 2) \end{array}$$

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

$$\begin{array}{ll} (f^u)^{-1}(\ell) \neq \emptyset \Leftrightarrow \ell \in \mathsf{image}(u) & (\text{correctness of } \mathsf{image}) \\ (\exists \boldsymbol{s} \in \mathcal{L}^n : f^u(\boldsymbol{s})) = (\bigvee_{\boldsymbol{s} \in \mathcal{L}^n} f^u(\boldsymbol{s})) = (\bigvee_{\ell \in \mathsf{image}(u)} \ell) & \text{(existential quantification)} \end{array}$$

To demonstrate how existential quantification works, we refer again to the example in Figure 4, 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 $\mathrm{F} \vee \mathrm{M} \vee \mathrm{T} = \mathrm{T}$.

In general, algorithms for manipulating BDDs are easily extensible to the multi-valued case, provided they do not use any optimizations that depend on a two-valued boolean logic (e.g. complemented edges [20]). The differences are discussed below.

```
function MakeUnique(name, children)
    find (create if not found) a node $u$ s.t.
        $\mathsf{var}(u) = \mathsf{name} \wedge$
        $\forall \ell,\ \mathsf{child}_\ell(u) = \mathsf{children}(\ell)$
    return $u$

function Quantify($u$, $i$)
// existentially quantifies over all variables $a_j$ with $j \geq i$.
    if $\mathsf{var}(u) < i$
        then foreach $\ell \in \mathcal{L}$
            $\mathsf{children}(\ell) :=$ Quantify($\mathsf{child}_\ell(u), i$)
        return MakeUnique($\mathsf{var}(u)$, children)
    else return Lattice.bigOR($\mathsf{image}(u)$)

function Apply(op, $u_1$, $u_2$)
// applies the lattice operation op to the MDDs $u_1$ and $u_2$
    global $G = |u_1| \times |u_2|$ array of int
    Apply'(op, $u_1$, $u_2$)

function Apply'(op, $u_1$, $u_2$)
// helper function for Apply which actually does the work
    if $G[u_1][u_2]$ non-empty
        then return $G[u_1][u_2]$
    else
        if $u_1 \in \mathcal{L} \wedge u_2 \in \mathcal{L}$
            then $u :=$ Lattice.doOp($u_1$, $u_2$, op)
        else if $\mathsf{var}(u_1) = \mathsf{var}(u_2)$
            then foreach $\ell \in \mathcal{L}$
                $\mathsf{children}(\ell) :=$ Apply'(op, $\mathsf{child}_\ell(u_1)$, $\mathsf{child}_\ell(u_2)$)
            $u :=$ MakeUnique($\mathsf{var}(u_1)$, children)
        else if $\mathsf{var}(u_1) < \mathsf{var}(u_2)$
            then foreach $\ell \in \mathcal{L}$
                $\mathsf{children}(\ell) :=$ Apply'(op, $\mathsf{child}_\ell(u_1)$, $u_2$)
            $u :=$ MakeUnique($\mathsf{var}(u_1)$, children)
        else
            foreach $\ell \in \mathcal{L}$
                $\mathsf{children}(\ell) :=$ Apply'(op, $u_1$, $\mathsf{child}_\ell(u_2)$)
            $u :=$ MakeUnique($\mathsf{var}(u_1)$, children)
        $G[u_1][u_2] := u$
        return u
```

Fig. 5. The MDD algorithms `MakeUnique`, `Quantify` and `Apply` for binary operators. `Apply` for unary operators is defined similarly.

The most-used method in an MDD (or BDD) library is `MakeUnique`, defined in Figure 5. This guarantees uniqueness and thus reducedness [21]. `MakeUnique` is not a public method, but it is used by most of the public methods.

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 par-

```
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)
            AXTerm$_{i+1}$ := Apply($\neg$, EX(Apply($\neg, QU_i$)))        // AX($QU_i$)
            EXTerm$_{i+1}$ := EX($QU_i$))
        else
            AXTerm$_{i+1}$ := $P_\varphi$
            EXTerm$_{i+1}$ := EX(Apply($\neg, QU_i$)))
        $QU_{i+1}$ := Apply($\vee, P_\psi$, (Apply($\wedge, P_\varphi$, Apply($\wedge$, EXTerm$_{i+1}$, AXTerm$_{i+1}$))))
    until $QU_{i+1} = QU_i$
    return $QU_n$

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

Fig. 6. The multi-valued symbolic model checking algorithm.

tition $P_\varphi^{-1}(\mathcal{L})$. Build ensures orderedness of MDDs while they are being constructed, and Apply preserves it. Apply and Quantify are shown in Figure 5. Note how each interfaces with the lattice library: Apply calls the method Lattice.doOp to compute $\wedge$ or $\vee$ of two terminal nodes, while Quantify requires Lattice.bigOR to compute the disjunction of an MDD's image-set.

An additional function, Prime, primes all of the variables in an MDD. In general, primed and unprimed variables may be mixed in the variable ordering, but for the purposes of this presentation, primed variables are always higher in the ordering. For instance, $(\mathsf{a}, \mathsf{c}, \mathsf{b}, \mathsf{a}', \mathsf{c}', \mathsf{b}')$ is an acceptable variable ordering, but $(\mathsf{a}, \mathsf{a}', \mathsf{b}, \mathsf{b}', \mathsf{c}, \mathsf{c}')$ is not. Quantify will still work in the more general case, but some preliminary variable reordering will be needed.

5.2 The Model Checker

Symbolic model checkers for boolean logic [7,4] are naturally extended to the multi-valued case. The model checker presented here is a symbolic version of the multi-valued model checker described in [6].

The full model checking algorithm is described in Figure 6. The function EX(P_φ) 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, AF, ER and AR are not

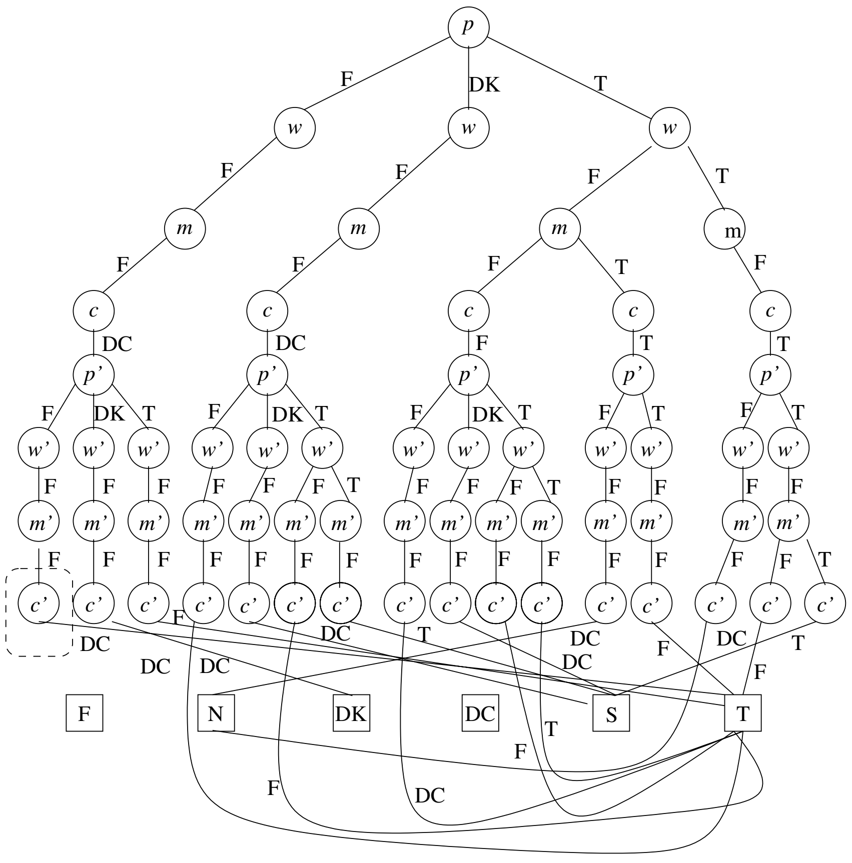

Fig. 7. MDD for R, representing the transition relation for the coffee dispenser.

shown in this Figure, but could be added as cases and defined in terms of calls to `EX`, `QUntil`, and `Apply`.

Proposition 1 *The function* $\texttt{EX}(P_\varphi)$ *computes* $EX\varphi$. *That is, a state vector* $\boldsymbol{s}$ ℓ*-satisfies* $\texttt{Quantify}(R \wedge P_{\varphi'}, n)$ *if and only if* $(\bigvee_{\boldsymbol{t} \in S} R(\boldsymbol{s}, \boldsymbol{t}) \wedge P_\varphi(\boldsymbol{t})) = \ell$.

To illustrate the algorithm, we compute the partition given by the χCTL formula $EX(\texttt{cup})$ in the coffee dispenser example in Figure 1(b). This computation is equivalent, in symbolic terms, to computing $\exists \boldsymbol{v'}$, s.t. $R \wedge P_{\texttt{cup}'}$, where $\boldsymbol{v'}$ is a *primed state vector*; the intuition here is the quantification over all possible next states. This is implemented in the model checker by the expression

$$\texttt{Quantify}(\texttt{Apply}(\wedge, R, \texttt{Prime}(P_\varphi)), n)$$

Not every n-ary function over an arbitrary quasi-boolean lattice can be written in the relatively economical form of a propositional formula, and so we need to show either an MDD or a function-table representation. We will use MDDs. The MDD for the transition relation of the model (R) is shown in Figure 7. For clarity and to save space, we used the following conventions: (a) all state variables are abbreviated to their initial

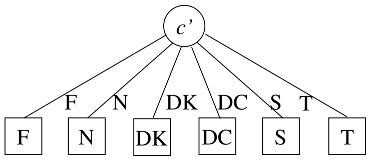

Fig. 8. The MDD for $P_{\mathtt{cup}'}$.

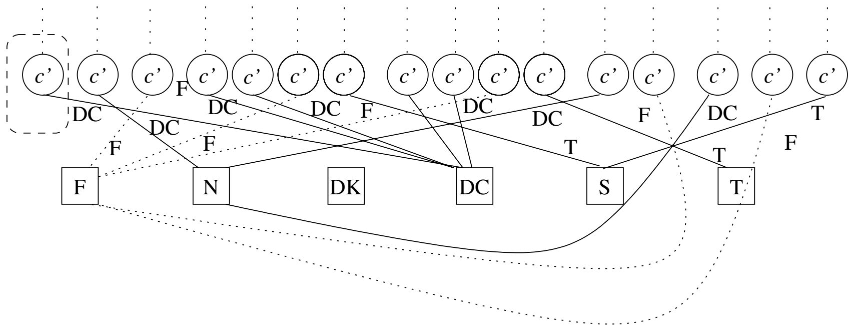

Fig. 9. The lowest level of the MDD for $R \wedge P_{\mathtt{cup}'}$. Dotted lines indicate the transitions which existed in Figure 7 and are now set to F.

letters; e.g., m stands for `milk`, w for `water`, etc; (b) the MDD is not reduced; (c) the transitions to the terminal node F are not shown. Note that there are no transitions to node DC in this diagram. The MDD for $P_{\mathtt{cup}'}$ is given in Figure 8.

We start by computing $R \wedge P_{\mathtt{cup}'}$. The top part of the MDD is the same as shown in Figure 7. The bottom row is given in Figure 9. The conjunction of two MDDs with the same variable in the root node is carried out by pairwise conjunction of their children; for instance, consider the leftmost node in Figure 7 labeled with `cup`$'$, indicated by the dashed box; its $\mathsf{child}_{\mathrm{DC}}$ is T, and its other children are F. The MDD for `cup`$'$ has $\mathsf{child}_\ell = \ell$ for all $\ell \in \mathcal{L}$. Their conjunction, then, has $\mathsf{child}_\ell = \ell$ for any $\ell \in \mathcal{L}$ except for DC; $\mathrm{DC} \wedge \mathrm{T} = \mathrm{DC}$, so the child is DC, as shown by the dashed box in Figure 9.

To complete the computation, the model checker needs to existentially quantify over the primed variables. `Quantify` replaces all primed-variable nodes u which are immediate children of unprimed-variable nodes, with the constant node $\bigvee \mathsf{image}(u)$. For instance, we can see by inspection that the leftmost subgraph with `power`$'$ at the root (which corresponds to the successor states of `OFF`) has DC, N, F in its image-set, so it is replaced by the terminal node $\bigvee\{\mathrm{DC}, \mathrm{N}, \mathrm{F}\} = $ DC; from this we conclude that EX`cup` has the value DC in state `OFF`, the model's initial state.

The properties of the coffee dispenser example given in Section 2 and formalized in Section 4 can be model checked with the results given in Table 1.

Num	*Property*	*Result*	*Comment*
1.	$EF(\mathtt{water})$	T	as expected
2.	$EF(\mathtt{milk})$	S	as expected
3.	$AG(\mathtt{water} \rightarrow \mathrm{cup})$	T	as expected
4.	$AG(\mathtt{water} \rightarrow AXA[\neg\mathtt{water}\ \mathcal{W}\ (\neg\mathrm{cup} \wedge \neg\mathtt{water})])$	S	as $R(\mathtt{COFFEE}, \mathtt{FOAM})$ = S

Table 1. Results of model checking the coffee dispenser.

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

Table 2. MDD methods used for model checking, and their running times.

5.3 Analysis

Table 2 shows the running times of MDD operations used by the model checker in terms of $|u|$, the size of the MDD. In the worst case, this size is $O(|\mathcal{L}|^n)$ [21].

The running times of MDD methods Build and Apply critically depend on the sizes of MDD structures. In order to form a rough estimate of these sizes in the average case, we ran the MDD library on several test sets, with results shown in Figure 10. Each data point in the graph stands for a set of 200 MDDs, each representing a function generated by filling in a random value (chosen from a uniform distribution) from $\mathcal{L}$ for each possible input. We generated one such set for 3, 4, and 5 variables for lattices ranging in size from 3 to 8 (the x-axis of the graph), and took the average size of the MDDs representing the functions. The figure shows the *worst-case* size $|\mathcal{L}|^n$, and our *experimental* results, for $n = 4$ and $n = 5$; $n = 3$ is similar.

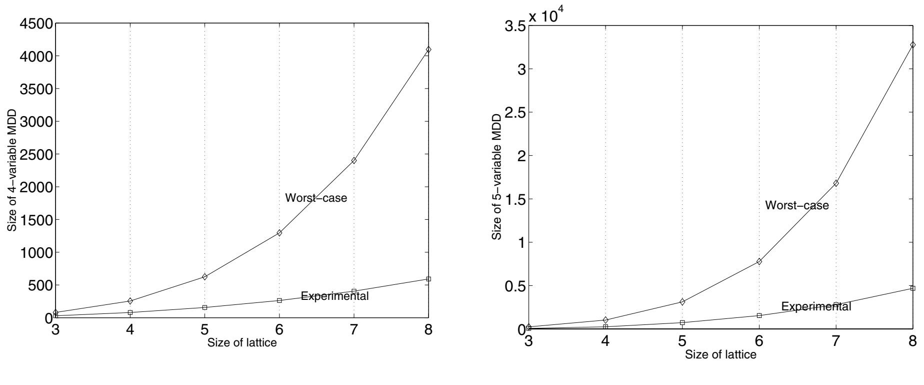

Fig. 10. Worst-case and experimental average-case sizes of MDDs plotted against lattice size. (a) $n = 4$, (b) $n = 5$.

The results show the average size of the generated MDDs to be roughly $|\mathcal{L}|^{n-1}$, a linear improvement over the worst-case $O(|\mathcal{L}|^n)$. We recognize the weakness of this methodology: that it does not give a good idea of how the structure of the problem affects the size of MDDs. We suspect that the structure of the model checking problem results in a somewhat better improvement, but do not yet have adequate benchmarks with which to test this hypothesis. In the future, we would like to perform the same test for an appropriate test suite of multi-valued models, to check whether the structure of the model checking task has an impact on the size of MDDs.

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; a detailed proof can be found in [6]. 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. The results of Figure 10 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 be performing significantly worse than BDDs ($O(|\mathcal{L}|^n)$ versus $O(2^n)$ in the worst case). 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 ($p << n$) 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. This computation did not take into an account the presence of "unknown" transitions; these could also be encoded into the binary representation, but would lead to a further state-space increase. Thus, we expect that often our model checker would perform as well as the classical one, and on some problems even better.

Finally, the scope of the applicability of an MDD-based model checker includes reasoning about inconsistent models.

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 multi-valued model checker χchek and illustrated it using a simple coffee dispenser example.

We plan to extend the work presented here in a number of directions to ensure that χchek can effectively reason about non-trivial systems. We will start by addressing some of the limitations of our χKripke structures. In particular, so far we have assumed that our variables are of the same type, with elements described by values of the lattice associated with that machine. We need to generalize this approach to variables of different types. We are also working on generalizing our algorithm to verification of properties expressed in CTL*.

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. This model checker has the same running time as the MDD-based one. However, lattice-theoretic results can be used to significantly optimize the table-driven model checker. Our report on this work is forthcoming [5].

Acknowledgments

We thank the members of the University of Toronto formal methods reading group for many interesting discussions and for comments on earlier drafts of this paper. Special thanks go to Albert Lai, Victor Petrovykh, Mark Pichora, Arie Gurfinkel, Christopher Thompson-Walsh and Ric Hehner. We are also grateful to Ralph Freese of the University of Hawaii for the lattice library. This work was funded by NSERC and CITO.

References

1. N.D. 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 Trans. on Software Engineering and Methodology*, 9(1):3–50, Jan 2000.
5. M. Chechik, B. Devereux, S. Easterbrook, A. Lai, & V. Petrovykh. "Efficient Multiple-Valued Model-Checking Using Lattice Representations". submitted for publ. Jan 2001.
6. M. Chechik, S. Easterbrook, and V. Petrovykh. "Model-Checking Over Multi-Valued Logics". In *Proc. Formal Methods Europe (FME'01)*, March 2001.
7. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
8. E. M. Clarke, D. E. Long, and K. L. McMillan. "Compositional Model Checking". In *Proc. 4th Ann. Symp. on Logic in Computer Science*, pages 464–475, June 1989.
9. S. Easterbrook and M. Chechik. "A Framework for Multi-Valued Reasoning over Inconsistent Viewpoints". In *Proc. 23rd Int. Conf. on Software Engineering (ICSE'01)*, May 2001.
10. M. Fitting. Many-Valued Modal Logics. *Fundamenta Informaticae* 15(3-4):335–350, 1991.
11. R. Freese, J. Ježek, and J. B. Nation. *Free Lattices*. Amer. Math. Soc., Providence, 1995. Mathematical Surveys and Monographs, vol. 42.
12. B. R. Gaines. "Logical Foundations for Database Systems". *Int. J. of Man-Machine Studies*, 11(4):481–500, 1979.
13. M. Ginsberg. "Multi-valued logic". In M. Ginsberg, editor, *Readings in Nonmonotonic Reasoning*, pages 251–255. Morgan-Kaufmann, 1987.
14. R. Hähnle. *Automated Deduction in Multiple-Valued Logics*, volume 10 of *International Series of Monographs on Computer Science*. Oxford U. Press, 1994.
15. S. Hazelhurst. *Compositional Model Checking of Partially Ordered State Spaces*. PhD thesis, Dept of Computer Science, U. of British Columbia, 1996.
16. J. Łukasiewicz. *Selected Works*. North-Holland, Amsterdam, Holland, 1970.
17. R. S. Michalski. "Variable-Valued Logic and its Applications to Pattern Recognition and Machine Learning". In D. C. Rine, editor, *Computer Science and Multiple-Valued Logic: Theory and Applications*, pages 506–534. North-Holland, Amsterdam, 1977.
18. H. Rasiowa. *An Algebraic Approach to Non-Classical Logics. Studies in Logic and the Foundations of Mathematics*. Amsterdam: North-Holland, 1978.
19. V. Sofronie-Stokkermans. Automated theorem proving by resolution for finitely-valued logics based on distributive lattices with operators. *Multiple-Valued Logic: An International Journal* 5(2), 2000.
20. F. Somenzi. "Binary Decision Diagrams". In Manfred Broy and Ralf Steinbrüggen, editors, *Calculational System Design*, vol 173 of *NATO Science Series F: Computer and Systems Sciences*, pp 303–366. IOS Press, 1999.
21. A. Srinivasan, T. Kam, S. Malik, and R.E. Brayton. "Algorithms for Discrete Function Manipulation". In *IEEE Int. Conf. on Computer-Aided Design*, pages 92–95, 1990.