

# Data Structures for Symbolic Multi-Valued Model-Checking

Marsha Chechik, Arie Gurfinkel, Benet Devereux, Albert Lai and Steve Easterbrook  
Department of Computer Science, University of Toronto,  
Toronto, ON M5S 3G4, Canada.

Email: {`chechik, arie, benet, trebla, sme`}@cs.toronto.edu

Draft of January 7, 2002

## Abstract

Multi-valued logics can be effectively used to reason about incomplete and/or inconsistent systems, e.g. during early software requirements or as the systems evolve. In our earlier work we identified a useful family of multi-valued logics: those specified over finite distributive lattices where negation preserves involution, i.e.,  $\neg\neg a = a$  for every element  $a$  of the logic. Model-checking over this family of logics allows not only to extend the domain of applicability of automated reasoning to new problems, but also to speed up some classical verification problems.

Symbolic model-checking over multi-valued logics can be cast in terms of operations over multi-valued sets: sets whose membership functions are multi-valued. In this paper we propose and empirically evaluate several choices for implementing multi-valued sets with decision diagrams. In particular, we describe two major approaches: (1) representing the multi-valued membership function canonically, using MDDs or ADDs; (2) representing multi-valued sets as a collection of classical sets, using a vector of either MBTDDs or BDDs. The naive implementation of (2) includes having a classical set for each value of the logic. We exploit a result of lattice theory to reduce the number of such sets that need to be represented.

The major contribution of this paper is the evaluation of the different implementations of multi-valued sets, done via a series of experiments and using several case studies.

## 1 Introduction

Multi-valued logics provide an interesting alternative to classical boolean logic for modeling and reasoning about systems. By allowing additional truth values, they support the explicit modeling of uncertainty and disagreement. For these reasons, they have been explored in a variety of applications in databases [27], knowledge representation [28], machine learning [32], and circuit design [29].

In our earlier work [19], we identified a useful family of multi-valued logics, known as the *quasi-boolean* logics [8] (or *De Morgan* logics [24]). These are logics defined over finite quasi-boolean lattices with an additional negation operator  $\neg$  so that  $\neg\neg a = a$  for every element  $a$  of the logic. Clearly, multi-valued logics with a finite number of values do not add expressive power to the modeling. However, in

many cases this approach results in significantly smaller state spaces than the alternative of introducing additional boolean predicates.

We are interested in building a symbolic model-checker over multi-valued logics, both to explore new application areas, such as reasoning about disagreements, and to speed up classical applications. Our model-checker,  $\chi$ Chek, takes as its input a particular quasi-boolean logic, a state machine model, and a correctness property expressed in CTL enriched with multi-valued semantics.  $\chi$ Chek then returns the truth value that the property has in the initial state and a counter-example, if applicable. Algorithms for  $\chi$ Chek were defined and proven correct in a companion paper [16].

Similar approaches have been proposed by other researchers. In particular, Bruns and Godefroid [9] use three-valued logic for model-checking of partially specified systems: the in-between value represents a transition which may become either enabled or disabled in further refinements of the specification. Pazos-Arias et al [34] use a six-valued logic to formalize a more detailed view of partial specification: the additional values are used to distinguish between what refinements are possible, and also to indicate where an inconsistency has arisen. We have previously [18] explored automata-theoretic model-checking for multiple-valued linear-time temporal logic. Finally, Bruns and Godefroid [10] have given algorithms which can be used for automata-theoretic model-checking for branching-time logic as an alternative to the symbolic algorithm.

The goal of this paper is to apply and evaluate the well-tried approach of performing symbolic reasoning using decision diagrams to the new problem of multi-valued model-checking. This paper, which is an extension of our earlier work reported in [15, 17], makes the following major contributions:

1. We define operations of multi-valued symbolic model-checking using one generic data structure called *mv-sets*. Success of symbolic model-checking in a given domain (probabilistic, multi-valued, timed, etc.) depends on efficient algorithms for manipulating the sets of states in which a property holds. We need to calculate union, intersection, complement, and quantify over these sets. For model checking in a given multi-valued logic, we can treat these as operations over multi-valued sets: sets whose membership functions are multi-valued. Mv-sets have multi-valued membership functions, and thus can be naturally encoded using MDDs, the multi-valued extension of binary decision diagrams [12]. MDDs are multi-terminal decision diagrams where each node has multi-valued branching factor.
2. Mv-sets can also be represented and manipulated as a collection of classical sets. The easiest such encoding is to associate one classical set with each element of the logic. Our second contribution is in using the join-irreducible decomposition of lattice elements to encode mv-set manipulations. This decomposition is always better than the one above. The encoding can be represented symbolically using boolean-terminal decision diagrams, and in particular, using Multi-Valued Boolean-Terminal diagrams (MBTDDs). MBTDDs is a special case of MDDs where the input variables are multi-valued but the terminal nodes have only the classical truth values 0 or 1.
3. In order to compare the MDD- and the MBTDD-based implementations of mv-sets, we devised a new method for generating multi-valued benchmarks from classical ones.
4. Using these benchmarks, we have performed experiments to explore the tradeoffs in time and space between the two proposed representations, in the specific domain of model-checking.

The rest of this paper is organized as follows: Section 2 gives background material for this paper, defining quasi-boolean logics and multi-valued sets and relations. Section 3 reviews multi-valued state machine

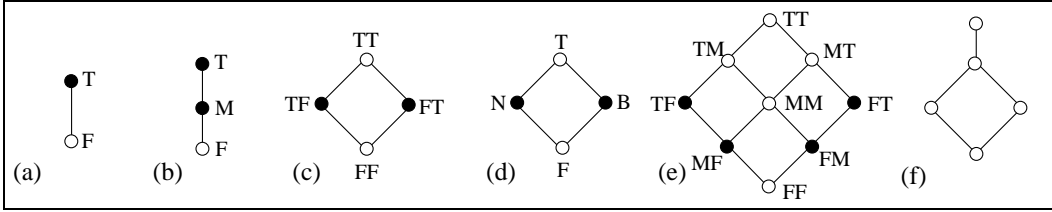


Figure 1: Some lattices: (a)  $\mathbf{2}$ , the classical logic lattice; (b)  $\mathbf{3}$ , a 3-valued lattice representing uncertainty; (c)  $\mathbf{2} \times \mathbf{2}$ , a 4-valued lattice representing disagreement; (d) Belnap’s 4-valued lattice representing inconsistent information; (e) a lattice  $\mathbf{3} \times \mathbf{3}$ ; (f) a non-quasi-boolean lattice. Join-irreducible elements are indicated by filled circles. Lattices in (a)-(e) are quasi-boolean.

models, called  $\chi$ Kripke structures, and the extension of CTL. We discuss the high-level implementation of the multi-valued model-checking algorithm in Section 4. The MDD-based and the MBTDD-based implementations of mv-sets are described in Sections 5 and 6, respectively. The experiments and the theoretical assessment of complexity of mv-set operations under different decision diagram representations are given in Section 7. Section 8 compares these representations on three case studies. We conclude in Section 9 with the summary and an outline for future work.

## 2 Background

In this section we describe the background for multi-valued model-checking. We begin the section with discussion of multi-valued logics and then proceed by discussing mv-sets – data types for storing and manipulating sets with varying membership degrees. A familiar instance of such mv-sets is a probability distribution on a set of independent events. In the case of multi-valued model-checking, the entities are states and transitions.

### 2.1 Quasi-Boolean Logic

Our approach to modeling makes use of an entire family of multi-valued logics. Rather than giving a proof system for each logic, we simply define 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, material implication, etc. Such properties can be easily guaranteed if we require that the truth values of the logic form a lattice. We describe the types of lattices used in our model-checker.

**Definition 1** A lattice is a partially-ordered set  $(\mathcal{L}, \sqsubseteq)$  for which a unique greatest lower bound and least upper bound exist for each finite set of elements.

Given lattice elements  $a$  and  $b$ , their greatest lower bound is referred to as *meet* and denoted by  $a \sqcap b$ , whereas their least upper bound is referred to as *join* and denoted by  $a \sqcup b$ . Meet and join over an empty set are referred to as *top* and *bottom*, respectively:

$$\begin{aligned} \top &\triangleq \sqcap \emptyset && (\top \text{ definition}) \\ \perp &\triangleq \sqcup \emptyset && (\perp \text{ definition}) \end{aligned}$$

We use the equals sign to denote the *identity* relation between different lattice elements:

$$a = b \triangleq a \sqsubseteq b \wedge b \sqsubseteq a \quad (\textit{identity})$$

The following properties hold for all lattices:

$$\begin{aligned} a \sqcup a &= a & a \sqcap a &= a & (\textit{idempotence}) \\ a \sqcup b &= b \sqcup a & a \sqcap b &= b \sqcap a & (\textit{commutativity}) \\ a \sqcup (b \sqcup c) &= (a \sqcup b) \sqcup c & a \sqcap (b \sqcap c) &= (a \sqcap b) \sqcap c & (\textit{associativity}) \end{aligned}$$

Figure 1 gives examples of a few lattices. For example, for the lattice  $\mathbf{2x2}$  in Figure 1(c),  $\top \sqcap \text{FT} = \text{FF}$ , whereas  $\top \sqcup \text{FT} = \text{TT}$ . Note that when the logic is classical, i.e. modeled by the lattice  $\mathbf{2}$ ,  $\sqcap$  and  $\sqcup$  operations are conventionally referred to as  $\wedge$  and  $\vee$ , respectively. We use these notations interchangeably when the interpretation is clear from the context. Further, in the lattices in Figure 1(a)-(e),  $\top$  is labeled ‘T’ (or ‘TT’) and  $\perp$  is labeled ‘F’ (or ‘FF’). We adopt the convention of labeling  $\top$  and  $\perp$  in this way in all our lattices, although in principle  $\top$  and  $\perp$  might be labeled differently.

In defining functions over lattice elements, we distinguish  $\top$  and  $\perp$ , since they are present in every lattice, giving them a special notation:

**Definition 2** Let  $\mathcal{L}$  be a lattice. An element  $x \in \mathcal{L}$  is called *crisp* iff  $x \in \{\top, \perp\}$ . A function  $f : \mathcal{L}^n \rightarrow \mathcal{L}$  is called *crisp-output* iff its only outputs are  $\top$  and  $\perp$ : that is,  $f(\mathcal{L}^n) \subseteq \{\perp, \top\}$ . It is called *crisp-input* iff only inputs in  $\{\perp, \top\}$  can lead to a non- $\perp$  output:  $f(\mathcal{L}^n \setminus \{\perp, \top\}^n) = \{\perp\}$ . A function  $f$  is called *crisp* if it is both *crisp-input* and *crisp-output*.

All lattices used in this paper are assumed to have a finite number of elements. We further assume that these lattices are distributive:

**Definition 3** A lattice is *distributive* if

$$\begin{aligned} a \sqcup (b \sqcap c) &= (a \sqcup b) \sqcap (a \sqcup c) & (\textit{distributivity}) \\ a \sqcap (b \sqcup c) &= (a \sqcap b) \sqcup (a \sqcap c) \end{aligned}$$

All lattices in Figure 1 are distributive.

**Definition 4** A distributive lattice  $(\mathcal{L}, \sqsubseteq)$  is *quasi-boolean* [8] (also called De Morgan [24]) iff there exists a lattice dual automorphism with period 2 for it.

**Theorem 1** Let  $(\mathcal{L}, \sqsubseteq)$  be a quasi-boolean lattice, and  $\neg$  be a lattice dual automorphism with period 2 defined for it. Then:

$$\begin{aligned} \neg(a \sqcap b) &= \neg a \sqcup \neg b & (\textit{De Morgan}) & \quad \neg\neg a = a & (\neg \textit{involution}) \\ \neg(a \sqcup b) &= \neg a \sqcap \neg b & \quad a \sqsubseteq b \Leftrightarrow \neg a \sqsupseteq \neg b & (\neg \textit{antimonotonic}) \end{aligned}$$

where  $a, b$  are elements of  $(\mathcal{L}, \sqsubseteq)$ . 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 distributive lattice.

**Definition 5** A quasi-boolean logic  $L$  is a tuple  $((\mathcal{L}, \sqsubseteq), \neg)$ , where:

- $(\mathcal{L}, \sqsubseteq)$  is a distributive quasi-boolean lattice;
- Conjunction and disjunction operators are  $\sqcap$  and  $\sqcup$  of  $(\mathcal{L}, \sqsubseteq)$ ;
- The partial order operation  $a \sqsubseteq b$  means “ $b$  is more true than  $a$ ”;
- Negation of the logic is the operator  $\neg$ , a lattice dual automorphism with period 2.

The identification of a suitable negation operator is greatly simplified by the observation that all quasi-boolean lattices are symmetric about their horizontal axes. All lattices in Figure 1(a)-(e) exhibit horizontal symmetry and thus are quasi-boolean, whereas the lattice in Figure 1(f) is not. The easiest way to define negation is through horizontal symmetry; however, other ways are also possible. For example, in  $\mathbf{2} \times \mathbf{2}$  shown in Figure 1(c),  $\neg TF = FT$ ,  $\neg FT = TF$ . However, in Belnap’s 4-valued lattice in Figure 1(d), proposed for reasoning about inconsistent databases [6, 2],  $\neg N = N$ ,  $\neg B = B$ . Thus, the two lattices are isomorphic, but the logics defined on them are not.

Implication and equivalence should preserve the law of material implication, and so are defined in a natural way as follows:

$$\begin{aligned} a \rightarrow b &\triangleq \neg a \sqcup b && \text{(material implication)} \\ a \leftrightarrow b &\triangleq (a \rightarrow b) \sqcap (b \rightarrow a) && \text{(equivalence)} \end{aligned}$$

We now define a notion of embedding of a classical logic into a quasi-boolean logic.

**Definition 6** Let  $L = ((\mathcal{L}, \sqsubseteq), \neg)$  and  $L' = ((\mathcal{L}', \leq), -)$  be two quasi-boolean logics. A function  $h : \mathcal{L} \rightarrow \mathcal{L}'$  is a quasi-boolean lattice homomorphism iff  $h$  is a lattice homomorphism between  $(\mathcal{L}, \sqsubseteq)$  and  $(\mathcal{L}', \leq)$ , and  $\forall x \in \mathcal{L} \cdot h(\neg x) = -h(x)$ .

Since any quasi-boolean logic homomorphism is also a lattice homomorphism, it preserves top and bottom elements; therefore, we can define an embedding of a classical logic into an arbitrary quasi-boolean logic:

**Definition 7** Let  $L$  be a quasi-boolean logic. An embedding of a classical logic into  $L$  is a unique homomorphism  $\alpha_L : \mathbf{2} \rightarrow L$ .

Intuitively,  $\alpha_L$  corresponds to an embedding of a classical logic into an arbitrary quasi-boolean logic  $L$ . We extend this notion of embedding to arbitrary boolean functions: if  $f : \mathbf{2}^n \rightarrow \mathbf{2}$  is a boolean function, then  $\alpha_L(f) : L^n \rightarrow L$  is the embedding of  $f$  into a quasi-boolean logic  $L$ .

We now review the concept of join-irreducibility for distributive lattices.

**Definition 8** [23] An element  $j$  in a lattice  $(\mathcal{L}, \sqsubseteq)$  is join-irreducible iff  $j \neq \perp$  and for any  $x$  and  $y$  in  $\mathcal{L}$ ,  $j = x \sqcup y$  implies  $j = x$  or  $j = y$ . Dually, an element  $m$  is meet-irreducible iff  $m \neq \top$  and for any  $x$  and  $y$ ,  $m = x \sqcap y$  implies  $m = x$  or  $m = y$ .

In other words,  $j$  cannot be further decomposed into the join of other elements in the lattice, and  $m$  cannot be further decomposed into the meet of other elements in the lattice, just as prime numbers cannot be further factored into the product of other natural numbers. For example, the join-irreducible elements

of the lattice  $\mathbf{3x3}$  in Figure 1(e), indicated by unfilled circles, are  $\{\text{MF}, \text{TF}, \text{FM}, \text{FT}\}$ . We denote the sets of all join-irreducible and meet-irreducible elements of a lattice  $(\mathcal{L}, \sqsubseteq)$  by  $\mathcal{J}(\mathcal{L}, \sqsubseteq)$  and  $\mathcal{M}(\mathcal{L}, \sqsubseteq)$ , respectively.

Every element of a finite lattice can be defined as a join of all join-irreducible elements below it:

**Theorem 2** [23] *Let  $\ell$  be any element in a lattice  $(\mathcal{L}, \sqsubseteq)$ . Then  $\ell = \bigsqcup\{j \in \mathcal{J}(\mathcal{L}, \sqsubseteq) \mid j \sqsubseteq \ell\}$ .*

For example, in the lattice shown in Figure 1(e),  $\text{TT} = \bigsqcup\{\text{MF}, \text{TF}, \text{FM}, \text{FT}\}$ ,  $\text{TM} = \bigsqcup\{\text{MF}, \text{TF}, \text{FM}\}$ ,  $\text{FF} = \bigsqcup\emptyset$ , and so on. Thus, each element of a lattice can be represented uniquely by a subset of join-irreducibles. For example,  $\text{TT} = \{\text{MF}, \text{TF}, \text{FM}\}$ .

Given a logic  $L = ((\mathcal{L}, \sqsubseteq), \neg)$ , we use notation  $\mathcal{J}(L)$  and  $\mathcal{M}(L)$  to refer to  $\mathcal{J}(\mathcal{L}, \sqsubseteq)$  and  $\mathcal{M}(\mathcal{L}, \sqsubseteq)$ , respectively. Further, we often refer to  $L$  as the *elements comprising the logic*, so  $|L|$  means the size of the logic and  $\forall \ell \in L$  quantifies over all logic elements.

## 2.2 Multi-Valued Sets and Relations

Multi-valued model-checking algorithms can be naturally described using a data structure that encapsulates reasoning about operations on sets of states in which a property holds. Such operations include union, intersection, complement, and backward image for computing predecessors. Given a logic, we can treat these as operations over multi-valued sets: sets whose membership functions are multi-valued. We define the concept of multi-valued sets and relations over quasi-boolean logics here.

In classical set theory, a set is defined by a boolean predicate, also called a *membership function*. Typically, it is written using the *set comprehension notation*: a predicate  $P$  defines the set  $S = \{x \mid P(x)\}$ . For instance, if  $P = \lambda x \in \text{nat} \cdot 0 \leq x \leq 10$ , then  $S$  is the set of all integers between 0 and 10 inclusive. If, instead of using a boolean membership function, we allow the membership function to range over a logic with more than two values, we obtain a *multi-valued set theory* in which it is possible to make statements like “element  $x$  is more in a set  $\mathbb{S}$  than element  $y$ ”. We call the resulting sets *mv-sets* and refer to them using a special font, e.g.,  $\mathbb{S}$ .

Given a logic  $L$ , we define a multi-valued set theory relative to it. For an mv-set  $\mathbb{S}$  and a candidate element  $x$ , we use  $\mathbb{S}(x)$  to denote the membership degree of  $x$  in  $\mathbb{S}$ . In the classical case, this amounts to representing a set by its characteristic function.

We illustrate mv-sets and other concepts in this paper using the following example.

### Example: The Callee system

Figure 2(a)-(b) shows two different versions of a callee’s view of a phone system, referred to as **callee1** and **callee2**. The models are expressed as conventional finite-state machines, where we explicitly mark existing transitions between states with value T. To distinguish between names of states and variables in our examples, the former are capitalized, so DIALTONE is a state, whereas Offhook is a variable. **callee1** describes a phone that allows you to replace the receiver during an incoming call without getting disconnected. **callee2** assumes that replacing the receiver always disconnects the call. Note that they disagree about the transitions out of the state CONNECTED, and about the value of the variable connected in state RINGTONE.

We can merge the two individual descriptions, referred to as *viewpoints*, and reason about the properties they share and those they disagree about. The merged model is given in Figure 2(c) and referred to as **callee-m**. **callee1** and **callee2** are classical models defined over the same vocabulary. Thus, in order to preserve the exact source of all disagreements between the two individual viewpoints, we use the logic

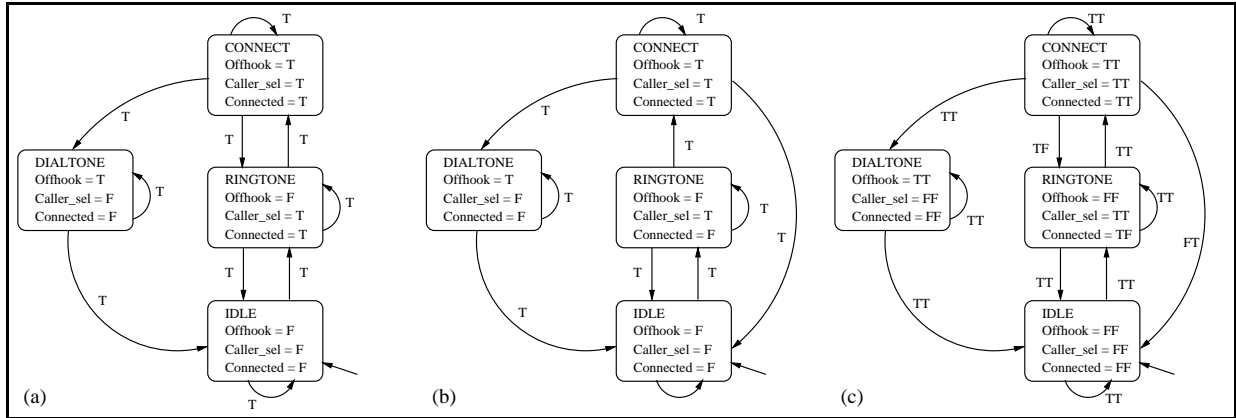


Figure 2: Three models of a callee’s view of a phone system (from [25]): (a) **callee1**; (b) **callee2**; (c) **callee-m** – a merge between (a) and (b).

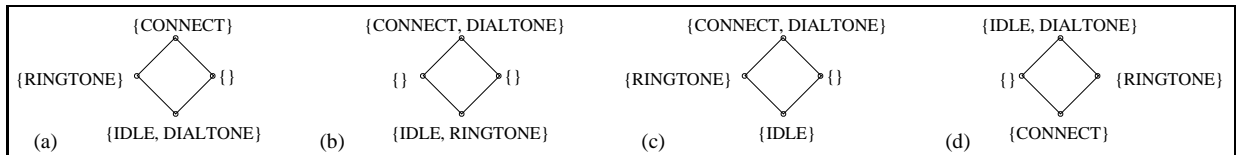


Figure 3: (a) The 4-valued set of states reflecting values of **Connected** in **callee-m**; (b) The 4-valued set of states reflecting values of **Offhook**; (c) A union of mv-sets in (a) and (b); (d) A multi-valued complement of the mv-set in (a).

**2x2** of Figure 1(c). Values of transitions between states or variables in states are computed by taking a *tuple* of corresponding values from **callee1** and **callee2**. For example, the value of a transition between **RINGTONE** and **CONNECT** is (T,T), or TT in **2x2**, since this transition is specified in either viewpoint. On the other hand, the value of a transition between states **CONNECT** and **IDLE** is FT, indicating that this transition was specified in the second viewpoint but not in the first. The viewpoints also disagree on the value of variable **Connected** in state **RINGTONE**, which is reflected in **callee-m** by assigning it value TF. By convention of classical state machines, we do not show transitions with value FF.  $\square$

We can use the logic **2x2** to encode information about values of variable **Connected** in different states of **callee-m**. We refer to this mv-set as  $\llbracket \text{Connected} \rrbracket$ . Each value  $\ell$  in **2x2** is associated with a set of states where **Connected** has value  $\ell$ . For example, **Connected** is TT in {**CONNECT**}, TF in {**RINGTONE**}, FF in {**IDLE**, **DIALTONE**} and FT nowhere. This mv-set can be graphically represented as shown in Figure 3(a), where the structure corresponds to that of the underlying lattice.

We extend some standard set operations to the multi-valued case by lifting the lattice meet and join operations, as follows:

$$\begin{aligned}
 (\mathbb{S} \cap_L \mathbb{S}') (x) &\triangleq (\mathbb{S}(x) \sqcap \mathbb{S}'(x)) && \text{(multi-valued intersection)} \\
 (\mathbb{S} \cup_L \mathbb{S}') (x) &\triangleq (\mathbb{S}(x) \sqcup \mathbb{S}'(x)) && \text{(multi-valued union)}
 \end{aligned}$$

For example, in computing the union of mv-sets  $\llbracket \text{Connected} \rrbracket$  and  $\llbracket \text{Offhook} \rrbracket$  given in Figure 3(a) and

(b), respectively, we note that in state DIALTONE, `Connected` is FF and `Offhook` is TT. Thus,

$$(\llbracket \text{Connected} \rrbracket \cup_L \llbracket \text{Offhook} \rrbracket)(\text{DIALTONE}) = \text{TT} \sqcup \text{FF} = \text{TT}$$

The set representing this union is given in Figure 3(c).

We can also extend the notion of *set complement* to the multi-valued case, by defining it in terms of the quasi-complement of  $L$ , and denoting it with a bar:

$$\overline{\mathbb{S}}(x) \triangleq \neg(\mathbb{S}(x)) \quad (\text{multi-valued complement})$$

Mv-set  $\overline{\llbracket \text{Connected} \rrbracket}$  is given in Figure 3(d).

We then obtain the expected properties:

$$\begin{aligned} \overline{\mathbb{S} \cup_L \mathbb{S}'} &= \overline{\mathbb{S}} \cap_L \overline{\mathbb{S}'} && (\text{De Morgan 1}) \\ \overline{\mathbb{S} \cap_L \mathbb{S}'} &= \overline{\mathbb{S}} \cup_L \overline{\mathbb{S}'} && (\text{De Morgan 2}) \\ \mathbb{S} \subseteq_L \mathbb{S}' &= \overline{\mathbb{S}'} \subseteq_L \overline{\mathbb{S}} && (\text{antimonotonicity}) \end{aligned}$$

We now define some additional concepts for mv-sets that are not needed in the classical set theory but used later in this paper.

**Definition 9** *Support, Core,  $\ell$ -cut, and  $\ell$ -clip of a multi-valued set  $\mathbb{S}$ :*

$$\begin{aligned} \sigma(\mathbb{S}) &\triangleq \{x \mid \mathbb{S}(x) \neq \perp\} && (\text{Support}) \\ \mathcal{C}(\mathbb{S}) &\triangleq \{x \mid \mathbb{S}(x) = \top\} && (\text{Core}) \\ \uparrow_\ell(\mathbb{S}) &\triangleq \{x \mid \ell \sqsubseteq \mathbb{S}(x)\} && (\ell\text{-cut}) \\ \downarrow_\ell(\mathbb{S}) &\triangleq \{x \mid \mathbb{S}(x) \sqsubseteq \ell\} && (\ell\text{-clip}) \end{aligned}$$

All of these operations create classical sets. Support, cut, and core are standard concepts from fuzzy set theory [43]; clip is a new operation, which we need to prove a later result. For example,

$$\begin{aligned} \sigma(\llbracket \text{Connected} \rrbracket) &= \{\text{CONNECT}, \text{RINGTONE}\} \\ \mathcal{C}(\llbracket \text{Connected} \rrbracket) &= \{\text{CONNECT}\} \\ \uparrow_{\text{TF}}(\llbracket \text{Connected} \rrbracket) &= \{\text{CONNECT}, \text{RINGTONE}\} \\ \uparrow_{\text{FT}}(\llbracket \text{Connected} \rrbracket) &= \{\text{CONNECT}\} \\ \downarrow_{\text{TF}}(\llbracket \text{Connected} \rrbracket) &= \{\text{IDLE}, \text{RINGTONE}, \text{DIALTONE}\} \\ \downarrow_{\text{FT}}(\llbracket \text{Connected} \rrbracket) &= \{\text{IDLE}, \text{DIALTONE}\} \end{aligned}$$

Following the conventions of fuzzy set theory, we identify an explicit universe of discourse  $U$ , rather than use an undefinable set of all possible entities. Mv-sets can be thought of as functions from elements of  $U$  to the underlying logic; therefore,  $U = \uparrow_\perp(\mathbb{S})$ .

Now we extend the concept of degrees of membership in an mv-set to degrees of relatedness of two entities. This concept, formalized by *multi-valued relations*, allows us to define multi-valued transitions in state machine models.

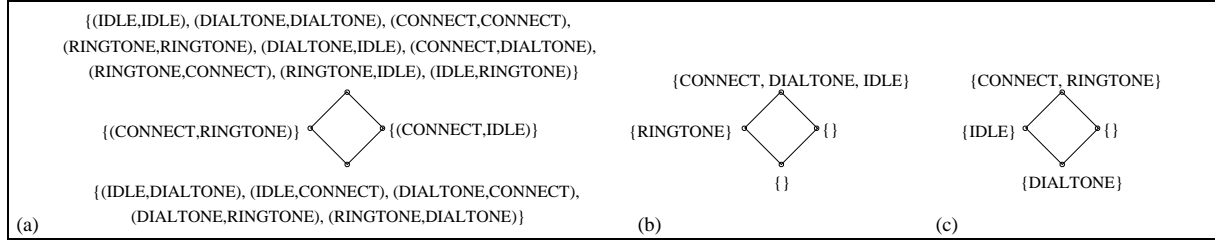


Figure 4: (a) The multi-valued relation between pairs of states of **callee-m**; (b) Forward image of  $\llbracket \text{Connected} \rrbracket$  over the relation in (a); (c) Backward image of  $\llbracket \text{Connected} \rrbracket$  over the relation in (a).

**Definition 10** A multi-valued relation  $\mathbb{R}$  on two sets  $S$  and  $T$  is an mv-set over  $S \times T$ .  $S$  and  $T$  are referred to as  $\mathbb{R}$ 's source and destination, respectively.

For example, the multi-valued relation representing values of transitions between pairs of states in **callee-m** is given in Figure 4(a). This relation, referred to as  $\mathbb{T}$ , is over  $S \times S$ , where  $S$  is the set of all states of **callee-m**.

**Definition 11** The forward image  $\vec{\mathbb{R}}(\mathbb{Q})$  of an mv-set  $\mathbb{Q}$  over  $S$  under relation  $\mathbb{R}$  is

$$\vec{\mathbb{R}}(\mathbb{Q}) \triangleq \lambda t \cdot \bigsqcup_{s \in S} (\mathbb{Q}(s) \sqcap \mathbb{R}(s, t))$$

The backward image  $\overleftarrow{\mathbb{R}}(\mathbb{Q})$  of an mv-set  $\mathbb{Q}$  over  $T$  under relation  $\mathbb{R}$  is

$$\overleftarrow{\mathbb{R}}(\mathbb{Q}) \triangleq \lambda s \cdot \bigsqcup_{t \in T} (\mathbb{Q}(t) \sqcap \mathbb{R}(s, t))$$

Intuitively, a forward image of an mv-set  $\mathbb{Q}$  over the relation  $\mathbb{R}$  means computing all elements reachable from  $\mathbb{Q}$  by  $\mathbb{R}$ , where multi-valued memberships of  $\mathbb{R}$  and  $\mathbb{Q}$  are taken into consideration. Similarly, a backward image of an mv-set  $\mathbb{Q}$  over  $\mathbb{R}$  is computing all elements that can reach  $\mathbb{Q}$  by  $\mathbb{R}$ . Given an mv-set over  $S$ , its forward image under the relation  $\mathbb{R}$  is an mv-set over  $T$ ; likewise, an mv-set over  $T$  has an mv-set over  $S$  as its backward image.

The forward and the backward images of  $\llbracket \text{Connected} \rrbracket$  (see Figure 3(a)) under the multi-valued relation between states of **callee-m** (Figure 4(a)) are shown in Figure 4(b) and (c), respectively. For example, in computing forward image of **IDLE**, we get

$$\bigsqcup_{s \in S} \llbracket \text{Connected} \rrbracket(s) \sqcap \mathbb{T}(s, \text{IDLE}) = (\text{FF} \sqcap \text{TT}) \sqcup (\text{FF} \sqcap \text{TT}) \sqcup (\text{TT} \sqcap \text{FT}) \sqcup (\text{FT} \sqcap \text{TT}) = \text{FT} \sqcup \text{TF} = \text{TT}$$

When we compute backward image of **IDLE**, we get

$$\bigsqcup_{t \in S} \llbracket \text{Connected} \rrbracket(t) \sqcap \mathbb{T}(\text{IDLE}, t) = (\text{FF} \sqcap \text{TT}) \sqcup (\text{FF} \sqcap \text{FF}) \sqcup (\text{TT} \sqcap \text{FF}) \sqcup (\text{TF} \sqcap \text{TT}) = \text{TF}$$

**Theorem 3 ([16])** If  $L = (\{T, F\}, \sqsubseteq, \neg)$ , i.e.,  $L$  is the classical logic, then  $\mathbb{R}$  becomes a boolean relation, and  $\mathbb{Q}$  becomes a classical set.

In this paper we do not use forward image. It is defined here for symmetry.

### 3 Multi-Valued Model-Checking

In this section we briefly review the multi-valued extensions of boolean CTL model-checking. We describe multi-valued Kripke structures, which we call  $\chi$ Kripke structures, and multi-valued CTL ( $\chi$ CTL). For more information about these, please refer to [16].

#### 3.1 $\chi$ Kripke Structures

$M$  is a  $\chi$ Kripke structure if  $M = (S, s_0, R, I, A, L)$ , where:

- $L = ((\mathcal{L}, \sqsubseteq), \neg)$  is a quasi-boolean logic.
- $A$  is a (finite) set of atomic propositions, otherwise referred to as variables. We assume that all variables are of the same type, with values ranging over the values of the logic  $L$ .
- $S$  is a (finite) set of states; each state is identified by a unique (within  $M$ ) label  $s$ .
- $s_0 \in S$  is the initial state.
- $\mathbb{R} : S \times S \rightarrow \mathcal{L}$  is the multi-valued transition relation.
- $I : S \rightarrow \mathcal{L}^A$  is a (total) labeling function that maps states in  $S$  to mv-sets over  $A$ . Intuitively, for any atomic proposition  $a \in A$ ,  $(I(s))(a) = \ell$  means that the variable  $a$  has value  $\ell$  in state  $s$ . Given an atomic proposition  $a \in A$ ,  $I'_a : S \rightarrow \mathcal{L}$  is a (total) multi-valued characteristic function for an mv-set of  $S$ .  $I'_a$  is defined as follows

$$I'_a \triangleq \lambda s \cdot (I(s))(a)$$

Thus, for each proposition  $a$ ,  $I'_a$  partitions the statespace with respect to it, i.e.,

$$\forall s \in S, \forall a \in A \cdot \exists! \ell \cdot I'_a(s) = \ell \quad (\text{partition})$$

Note that a  $\chi$ Kripke structure is a completely connected weighted graph. We also ensure that there is at least one non-false transition out of each state, extending the classical notion of Kripke structures. Formally,

$$\forall s \in S \cdot \exists t \in S \cdot \mathbb{R}(s, t) \neq \perp$$

To avoid clutter, we follow the convention of finite-state machines of not drawing  $\perp$  transitions. Our three models of the Callee given in Figure 2 are examples of  $\chi$ Kripke structures.

#### 3.2 Multi-Valued CTL

Here we give semantics of CTL operators on a  $\chi$ Kripke structure  $M$  over a quasi-boolean logic  $L$ . We refer to this language as *multi-valued CTL*, or  $\chi$ CTL.

We start defining  $\chi$ CTL by giving the semantics of propositional operators. We use the double-brace notation, adopted from denotational semantics, and write  $\llbracket \varphi \rrbracket$  to denote the mv-set of states representing the degree to which  $\varphi$  holds in a given state. Note that we have already used this notation when illustrating mv-sets in Section 2.2.

The semantics is as follows:

$$\begin{aligned}
[[a]] &\triangleq I'_a \\
[[\neg\varphi]] &\triangleq \overline{[[\varphi]]} \\
[[\varphi \wedge \psi]] &\triangleq [[\varphi]] \cap_L [[\psi]] \\
[[\varphi \vee \psi]] &\triangleq [[\varphi]] \cup_L [[\psi]]
\end{aligned}$$

We proceed by defining the  $EX$  operator. In classical CTL, this operator is defined using existential quantification over next states. We extend the notion of existential quantification for multi-valued reasoning through the use of disjunction. This treatment of quantification is standard [6, 37]. The semantics of  $EX$  is:

$$[[EX\varphi]] \triangleq \overleftarrow{\mathbb{R}} ([[ \varphi ]]) \quad (\text{def. of } EX)$$

$AX$  is then defined as a dual to  $EX$ :

$$[[AX\varphi]] \triangleq \overline{[[EX\neg\varphi]]} \quad (\text{def. of } AX)$$

Expanding this definition,  $[[AX\varphi]] = \overline{\overleftarrow{\mathbb{R}} ([[ \varphi ]])} = \lambda s \cdot \prod_{t \in S} ([[ \varphi ]](t) \sqcup \neg \mathbb{R}(s, t))$ , we see that universal quantification in the  $AX$  operator is replaced by conjunction.

We further define  $EG$  and  $EU$  using the  $EG$  and  $EU$  fixpoint properties of classical CTL:

$$\begin{aligned}
[[EG\varphi]] &\triangleq \nu Z. [[\varphi]] \cap_L [[EXZ]] && (\text{def. of } EG) \\
[[E[\varphi U \psi]]] &\triangleq \mu Z. [[\psi]] \cup_L ([[ \varphi ]] \cap_L [[EXZ]]) && (\text{def. of } EU)
\end{aligned}$$

Then  $A[\varphi U \psi]$  becomes

$$[[A[\varphi U \psi]]] \triangleq \overline{[[E[\neg\psi U \neg\varphi \wedge \neg\psi]]]} \cap_L \overline{[[EG\neg\psi]]} \quad (\text{def. of } AU)$$

and the remaining  $\chi$ CTL operators are defined as their classical counter-parts:

$$\begin{aligned}
[[AF\varphi]] &\triangleq [[A[\top U \varphi]]] && (\text{def. of } AF) \\
[[EF\varphi]] &\triangleq [[E[\top U \varphi]]] && (\text{def. of } EF) \\
[[AG\varphi]] &\triangleq \overline{[[EF\neg\varphi]]} && (\text{def. of } AG)
\end{aligned}$$

Properties of  $\chi$ CTL have been studied in detail in [16].

## 4 Implementing a Symbolic $\chi$ CTL Model-Checker

Symbolic model checkers for boolean logic (e.g. [21, 13]) are naturally extended to the multi-valued case. Figure 5 shows the high-level implementation of our multi-valued model-checker  $\chi$ Chek.

Table 1 lists types used in the algorithm and its underlying mv-set library. Type `Op` represents binary or unary logic operator, and `CTL` represents a CTL expression. `Var` represents an atomic proposition (a variable), where we adopt the following conventions: (a) for every variable  $v$  in the source state there is a corresponding variable  $v'$  in the destination state, and (b) there exists a variable ordering, such that each variable has a unique index in it. Function `variableMap(Var[] source)` receives names of variables in the source state and returns names of variables in the destination state. For the purposes of this presentation,

```

function checkCTL(CTL  $p$ ) : MvSet
  case  $p \in A$ :      return projection( $p$ )
  case  $p = \neg\varphi$ :   return ptwiseApply( $\neg$ , checkCTL( $\varphi$ ))
  case  $p = \varphi \wedge \psi$ : return ptwiseApply( $\wedge$ , checkCTL( $\varphi$ ), checkCTL( $\psi$ ))
  case  $p = \varphi \vee \psi$ : return ptwiseApply( $\vee$ , checkCTL( $\varphi$ ), checkCTL( $\psi$ ))
  case  $p = EX \varphi$ :   return backwardImg(checkCTL( $\varphi$ ),  $Rel$ )
  case  $p = E[\varphi U \psi]$ : return checkEU(checkCTL( $\varphi$ ), checkCTL( $\psi$ ))
  case  $p = EG \varphi$ :  return checkEG(checkCTL( $\varphi$ ))

function checkEG(MvSet  $\varphi$ ) : MvSet
  result0 = constant( $\top$ )
  do
    result $i+1$  = ptwiseApply( $\wedge$ ,  $\varphi$ , backwardImg(result $i$ ,  $Rel$ ))
  while result $i$   $\neq$  result $i+1$ 
  return result $i$ 

function checkUntil(MvSet  $\varphi$ , MvSet  $\psi$ ) : MvSet
  result0 = constant( $\perp$ )
  do
    result $i+1$  = ptwiseApply( $\vee$ ,  $\psi$ , ptwiseApply( $\wedge$ ,  $\varphi$ , backwardImg(result $i$ ,  $Rel$ )))
  while result $i$   $\neq$  result $i+1$ 
  return result $i$ 

function buildTransitionRelation(Relation  $\mathbb{R}$ ) : MvRelation
  MvSet trans = constant( $\perp$ )
  foreach edge  $e = (s, t, \ell) \in \mathbb{R}$ 
    newEdge = point( $s, t, \ell$ )
    trans = ptwiseApply( $\vee$ , trans, newEdge)
  return trans

function ModelCheck(CTL  $p$ , Vector  $s_0$ , Relation  $\mathbb{R}$ ) : Val
  Global Var[]  $SourceVars$ 
  Global Var[]  $DestVars = \text{variableMap}(SourceVars)$ 
  Global MvRelation  $Rel = \text{buildTransitionRelation}(\mathbb{R})$ 
  return cofactor(checkCTL( $p$ ),  $s_0$ )

```

Figure 5: The model-checking algorithm.

we do not distinguish between a variable name and its index in the variable ordering. Type `Val` represents logic values, with a special value `null` used to denote a “don’t care” value. `Vector` is a vector of elements of type `Val`, such that for any `Vector`  $\vec{s}$  and any atomic proposition  $v$ , both  $\vec{s}[v]$  and  $\vec{s}[v']$  are defined. In this and all other examples in this section, we assume a variable ordering `Offhook < Caller_sel < Connected < Offhook' < Caller_sel' < Connected'`. For example, state `DIALTONE` in `callee-m`

Name	Description	Example
MvSet	an multi-valued set	see Figure 3
MvRelation	same as MvSet	see Figure 4(a)
Var	an atomic proposition	Connected, 1, Offhook
Val	a logic value	TF, TT, <i>null</i>
Vector	a vector of type Val	(TF, FT, FF), ( <i>null</i> , TT, <i>null</i> )
Oper	a logic operator	$\wedge, \vee, \neg$
CTL	a CTL expression	<i>AG</i> Connected

Table 1: Types used by the mv-set library.

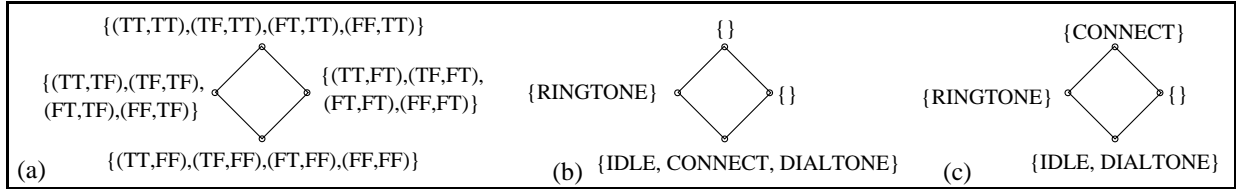


Figure 6: Illustrations of functions of the mv-set library: (a)  $\text{projection}(\text{Caller\_sel})$  for  $\text{Offhook} < \text{Caller\_sel}$ ; (b)  $\text{cofactor}(\llbracket \text{Connected} \rrbracket, \text{RINGTONE})$ ; (c)  $\text{cofactor}(\llbracket \text{Connected} \rrbracket, (\text{null}, \text{TT}, \text{null}, \text{null}, \text{null}, \text{null}))$ .

(see Figure 2(c)) under our variable ordering is represented as  $(\text{TT}, \text{FF}, \text{FF}, \text{null}, \text{null}, \text{null})$ . That is,  $\text{DIALTONE}[\text{Caller\_sel}] = \text{FF}$ , whereas  $\text{DIALTONE}[\text{Caller\_sel}'] = \text{null}$ . We use the name of a state interchangeably with a vector of values of its variables.

**MvSet** is a type representing an mv-set, i.e., a mapping between vectors and their values. Type **MvRelation** represents an mv-relation between two mv-sets. Its representation is the same as that of **MvSet**, but we use a different name for clarity of the presentation.

We further assume that the transition relation  $\mathbb{R}$  is represented as a set of *edges*  $(\vec{s}, \vec{t}, \ell)$ , indicating that an edge between states  $\vec{s}$  and  $\vec{t}$  has value  $\ell$ . For example, a TT transition between DIALTONE and IDLE under our variable ordering is represented as  $((\text{TT}, \text{FF}, \text{FF}, \text{null}, \text{null}, \text{null}), (\text{FF}, \text{FF}, \text{FF}, \text{null}, \text{null}, \text{null}), \text{TT})$ , or  $\mathbb{R}(\text{DIALTONE}, \text{IDLE}) = \text{TT}$ .

The algorithm **ModelCheck** receives a  $\chi$ CTL formula  $p$  to verify, the transition relation  $\mathbb{R}$  of the  $\chi$ Kripke structure, and its initial state  $s_0$ . **ModelCheck** assumes that the list of variables *SourceVars* is available and computes, for each variable  $v \in \text{SourceVars}$ , its associated  $v' \in \text{DestVars}$  using a function **variableMap**. It proceeds by computing *Rel* – an **MvRelation** representation of the transition relation. Then it calls the function **checkCTL** which computes the mv-set representing the partition of the state-space with respect to  $p$ . **checkCTL** uses *Rel* for the computation of **backwardImage**. Finally, **ModelCheck** constraints the result to return the value that  $p$  has in the initial state  $s_0$  of the system.

Function **checkCTL** recursively goes through the structure of the  $\chi$ CTL property  $p$ , associating each subformula of  $p$  with an mv-set representing the value that  $p$  has in each state of the  $\chi$ Kripke structure. **checkEG** and **checkEU** carry out the fix-point computations of the corresponding  $\chi$ CTL operators, as defined in Section 3.2.

Functions of the mv-set library are discussed below.

- Function `constant(Val  $\ell$ )` returns an mv-set  $\mathbb{S}$  where membership of each element is  $\ell$ :  $\forall \vec{x} \cdot \mathbb{S}(\vec{x}) = \ell$ . For example, `constant(TF)` returns an mv-set where  $\forall \vec{x} \cdot \mathbb{S}(\vec{x}) = \text{TF}$ .
- Function `projection(Var  $v$ )` creates an mv-set  $\mathbb{S}$  where membership of each element is determined by the value of variable  $v$ . Suppose for tractability that the universe of discourse for mv-sets is represented using two variables, `Offhook` and `Caller_sel`, ranging over the logic **2x2**. The size of this universe is  $4^2$  (two variables, 4 values each). Then `projection(Caller_sel)` returns an mv-set given in Figure 6(a). Note that membership of all vectors in this figure is determined by the value of their second variable.
- Function `point(Vector  $\vec{x}$ , Val  $\ell$ )` returns an mv-set  $\mathbb{S}$  in which  $\vec{x}$  has membership  $\ell$ , and all other elements have membership  $\perp$ :  $\forall \vec{y} \cdot \text{if } \mathbb{S}(\vec{y}) = \vec{x} \text{ then } \ell \text{ else } \perp$ . For example, `point((TT,TF,TF), FT)`, restricted to the universe of variables `Offhook`, `Caller_sel` and `Connected` is an mv-set where (TT,TF,TF) has value TF, while the remaining  $4^3 - 1$  states have value FF.
- Function `point(Vector  $\vec{x}$ , Vector  $\vec{y}$ , Val  $\ell$ )` computes a representation for one edge. For example, to compute a representation of a TT transition between DIALTONE and IDLE, we call `point((TT,FF,FF, null,null,null), (FF,FF,FF,null,null,null), TT)` which produces a vector  $\vec{x}=(\text{TT,FF,FF,FF,FF,FF,FF})$  representing values of source variables in DIALTONE together with values of source variables in IDLE moved to the destination position. `point( $\vec{x}$ , TT)` then computes the desired mv-set.
- Function `ptwiseApply(Oper  $op$ , MvSet  $\mathbb{S}$ )` returns an mv-set  $\mathbb{T}$  s.t.  $\forall \vec{x} \cdot \mathbb{T}(\vec{x}) = op \ \mathbb{S}(\vec{x})$ . For example, result `ptwiseApply( $\neg$ , [[Connected]])` is shown in Figure 3(d). Note that we abuse notation somewhat, using  $[[\varphi]]$  as an mv-set and as its `MvSet` representation. Membership of each element in the mv-set returned by `ptwiseApply(Oper  $op$ , MvSet  $\mathbb{S}$ , MvSet  $\mathbb{T}$ )` is determined by a pairwise application of  $op$  to each element of  $\mathbb{S}$  and  $\mathbb{T}$ . See Figure 3(c) for a result of `ptwiseApply( $\wedge$ , [[Connected]], [[Offhook]])`.
- Function `cofactor(MvSet  $\mathbb{S}$ , Vector  $\vec{x}$ )` returns an mv-set  $\mathbb{T}$  in which an element  $\vec{y}$  has membership  $\mathbb{S}(\vec{y})$  if it matches  $\vec{x}$  and  $\perp$  otherwise:  $\forall \vec{y} \cdot \text{if } \vec{y} = \vec{x} \text{ then } \mathbb{S}(\vec{y}) \text{ else } \perp$ . For example, mv-set for `cofactor([[Connected]], RINGTONE)`, or, equivalently, `cofactor([[Connected]], (FF,TT,TF,null,null,null))`, appears in Figure 6(b). In this set RINGTONE has value TF, since it has this value in `[[Connected]]`. `cofactor([[Connected]], (null, TT, null, null, null, null))` returns an mv-set shown in Figure 6(c)<sup>1</sup>: the only two non- $\perp$  states are RINGTONE and CONNECT, and in both `Caller_sel` has value TT.
- Function `backwardImg(MvSet  $\mathbb{S}$ , MvRelation  $\mathbb{R}$ )` returns an mv-set corresponding to  $\overleftarrow{\mathbb{R}}(\mathbb{S})$  in Definition 11.

Function `buildTransitionRelation` of the model-checker uses the mv-set library to construct the transition relation  $Rel$  in an `MvRelation` form.  $Rel$  is a disjunction of edges, where the mv-set for each

---

<sup>1</sup>This mv-set is the same as `[[Connected]]`.

edge is a conjunction of mv-sets representing each state and the value of the edge. For example, computation of `buildTransitionRelation` on `callee-m` shown in Figure 4(a). Suppose we are checking a property  $EX \text{ Connected}$  on `callee-m`. `checkCTL` returns  $\overleftarrow{\mathbb{R}} (\llbracket \text{Connected} \rrbracket)$ , shown in Figure 4(c). `cofactor`, called on this result and `IDLE`, the initial state of the `callee-m`, yields `TF`, which is returned to the user.

	<b>Boolean-terminal</b>	<b>Multi-terminal</b>
Boolean branching factor	BDD-vector	ADD
Multi-valued branching factor	MBTDD-vector	MDD

Table 2: Choices of Decision Diagram Packages for Implementing mv-sets..

In the remainder of the paper, we discuss alternatives for implementing the mv-set library. The library is built on top of several decision diagram packages. The choices we face with these is whether each node has two successors (boolean) or many successors (multi-valued), and whether the mv-set membership function is represented by a single multi-valued functions or by a collection of boolean functions. The first choice is referred to as *the branching factor*, and the second – deciding on the number of terminal nodes: *boolean-terminal* or *multi-terminal*. These choices are summarized in Table 2. The entries of the tables are the names of decision diagram packages supporting this implementation. For example, MDDs are multi-valued multi-terminal decision diagrams. When *boolean-terminal* diagrams are used, the mv-set membership function must be encoded by a collection of diagrams. We refer to this collection as a *decision diagram vector*. For example, BDD-vector refers to a representation where a collection of *binary decision diagrams* is used to encode a single mv-set membership function.

All four varieties of decision diagrams listed in Table 2 have been proposed in the literature: MDDs were first described by Srinivasan et al [40]. They included MBTDDs as a special case, but these are discussed in more detail by Sasao and Butler [38]. ADDs were proposed by Bahar et al [4] (these are also known under the name MTBDDs [26]), and BDDs were introduced by Akers [1] and later by Bryant [11], who suggested the added properties of reducedness and orderedness to guarantee canonicity. Technology for the diagrams with a branching factor of 2, i.e., ADDs and BDDs, is very mature and is supported by standard libraries such as CUDD [39]. It has also been shown in [33] that the difference between a decision diagram vector and a multi-terminal decision diagram is reducible to the variable ordering problem for them. It has also been shown in [41] that the trade-offs between the use of boolean versus multi-valued nodes in decision diagrams depend on the application at hand. We therefore set out to define and evaluate performance of multi-valued model-checking over all four data structures to identify trade-offs between them in the model-checking domain. The rest of the paper discusses multi-terminal decision diagram representation in Section 5 and boolean-terminal representation in Section 6.

## 5 Mv-set Implementation Using MDDs

Multi-Valued Decision Diagrams (MDDs) provide a viable option for implementing mv-sets and thus multi-valued model-checking. In this section we review definitions and properties of MDDs and discuss the implementation of mv-sets using these.

## 5.1 Multi-Valued Decision Diagrams

There is an extensive literature dealing with MDDs [40], 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 we briefly describe them below. We assume a basic knowledge of binary decision diagrams (BDDs) [12].

The central notion in the construction of BDDs 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  $\perp$ . These functions are defined as  $f|_{a_0=\top}$  and  $f|_{a_0=\perp}$ , respectively. We write this expansion as  $f(a_0, \dots, a_{n-1}) \rightarrow f|_{a_0=\top}(a_1, \dots, a_{n-1}), f|_{a_0=\perp}(a_1, \dots, a_{n-1})$ , and generalize it as follows:

**Definition 12** [40] *Given a finite domain  $D$ , the generalized Shannon expansion of a function  $f : D^n \rightarrow D$ , with respect to the first variable in the ordering, is:*

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

where  $\forall d_i \in D \cdot f_i = f|_{a_0=d_i}$  is the function obtained by substituting a literal  $d_i$  for  $a_0$  in  $f$ . These functions are called cofactors.

**Definition 13** *Assuming a finite set  $D$  and an ordered set of variables  $A$ , a multi-valued decision diagram (MDD) is a tuple  $(V, E, \text{var}, \text{child}, \text{value})$  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;
- $E \subseteq V_n \times V$  is a set of directed edges;
- $\text{var} : V_n \rightarrow A$  is a variable labeling function.
- $\text{child} : V_n \rightarrow D \rightarrow V$  is an indexed successor function for nonterminal nodes;
- $\text{value} : V_t \rightarrow D$  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, we are interested only in lattices; so instead of  $D$ , we refer to the elements of a quasi-boolean logic  $L$ . Note that in general we do not distinguish between a single node in an MDD and the subgraph rooted from it, referring to both indiscriminately as  $u$ .

Consider the function  $f = x \wedge y$ , with  $\ell_0 = \text{F}$ ,  $\ell_1 = \text{M}$ ,  $\ell_2 = \text{T}$ . The MDD for this expression is shown in Figure 7(a). The diagram is constructed by Shannon expansion, first with respect to  $x$ , and then (for each cofactor of  $f$ ) with respect to  $y$ . The dashed arrows point to initial states of MDDs representing  $f$  and its cofactors. For example, the MDD rooted at a node with label  $x$  represents  $f$ . When  $x = \text{F}$ ,  $f$  evaluates to  $\text{F}$ . Thus,  $f_0 = \text{F}$ .  $f_1$  corresponds to setting  $x = \text{M}$ , and is indicated in Figure 7(a) by a dashed arrow that points to the left subtree rooted at  $y$ . This means that the value of  $f$  is further determined by  $y$ : when  $y = \text{F}$ , corresponding to  $f_{1,0}$  cofactor,  $f$  has value  $\text{F}$ ; otherwise, i.e., for cofactors  $f_{1,1}$  and  $f_{1,2}$ ,  $f$  has value  $\text{M}$ .

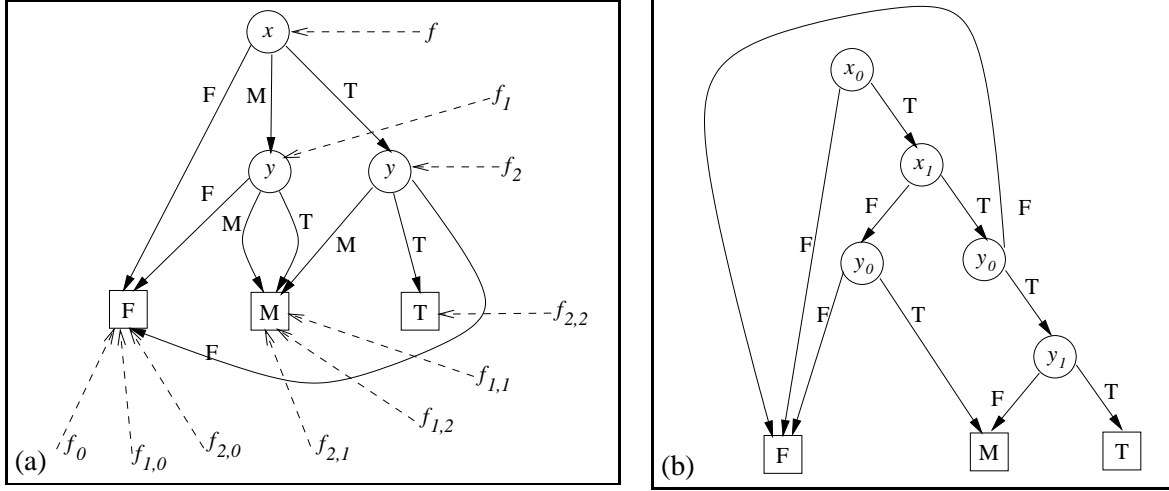


Figure 7: Representation of function  $f = x \wedge y$  in logic **3**: (a) using MDD; (b) using ADD.

**Definition 14** The function computed by an MDD  $u$  is denoted  $f^u : L^n \rightarrow L$ , and is defined recursively as follows:

$$u \in V_t \Rightarrow f^u(s_0, \dots, s_{n-1}) = \text{value}(u) \quad (\text{terminal constants})$$

$$u \in V_n \Rightarrow f^u(s_0, \dots, s_{n-1}) = f^{\text{child}_{s_i}(u)}(s_0, \dots, s_{n-1}),$$

where  $a_i = \text{var}(u)$  and  $\vec{s} \in L^n$  (cofactor expansion)

Consider the MDD in Figure 7(a). To compute  $f = x \wedge y$  with  $x = T$  and  $y = M$  using this diagram, we want to find  $f(\vec{s})$  where  $\vec{s} = (T, M)$ . We begin at the root node. Its  $\text{var}$  is  $x$ , so we choose the first argument, which is  $T$ , and descend to the node  $\text{child}_T(f)$ , indicated by the arrow to  $f_2$  (which represents the function  $T \wedge y$ ). Now we compute  $f_2(M)$  by choosing  $\text{child}_M(f_2)$ , which is a node in  $V_t$ , so we stop and return  $M$ . Thus, we conclude that  $f(T, M) = M$ . We further define size of an MDD:

$$\text{size}(u) \triangleq |\{v \mid v \text{ is reachable from } u\}|$$

For the function  $f$  in Figure 7(a),  $\text{size}(f) = 6$ .

The following properties hold for all MDDs:

$$\forall u_0 \in V_n \cdot \text{out}(u_0) = |L| \quad \wedge \quad \forall u_1 \in V_t \cdot \text{out}(u_1) = 0 \quad (\text{semantics of nodes})$$

$$\forall u_0, u_1 \in V \cdot \exists \ell \in L \cdot (u_0, u_1) \in E \Rightarrow \text{child}_\ell(u_0) = u_1 \quad (\text{semantics of edges})$$

where  $\text{out}(u)$  stands for the number of children of  $u$ . Several further properties are required for the data structure to be usable:

$$\forall u_0, u_1 \in V_n \cdot (u_0, u_1) \in E \wedge \text{var}(u_0) = a_i \wedge \text{var}(u_1) = a_j \Rightarrow i < j \quad (\text{orderedness})$$

$$\forall u_0, u_1 \in V \cdot f^{u_0} = f^{u_1} \Rightarrow u_0 = u_1 \quad (\text{reducedness})$$

$$\forall u_0, u_1 \in V_n \cdot ((\text{var}(u_0) = \text{var}(u_1)) \wedge (\forall \ell \in L \cdot \text{child}_\ell(u_0) = \text{child}_\ell(u_1))) \Rightarrow u_0 = u_1 \quad (\text{uniqueness 1})$$

$$\forall u_0, u_1 \in V_t \cdot (\text{value}(u_0) = \text{value}(u_1)) \Rightarrow u_0 = u_1 \quad (\text{uniqueness 2})$$

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 [40] – MDDs are unique by construction, and thus reduced.

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 4 Canonicity** [40] *For any quasi-boolean logic  $L$ , any nonnegative integer  $n$ , and any function  $f : L^n \rightarrow L$ , there is exactly one reduced ordered MDD  $u$  such that  $f^u = f(a_0, \dots, a_{n-1})$ .*

We now formalize a notion of *partial assignment* which is used to express partial evaluation of a function or a decision diagram.

**Definition 15** *Let  $A$  be a set of variables, and  $L$  be a logic. Then a partial assignment  $\rho \in 2^{A \times L}$  is a set of pairs  $\rho_i = (a_i, \ell_i)$  where  $i \neq j \Rightarrow a_i \neq a_j$ . A function  $f$  may be partially evaluated, given a partial assignment:*

$$f|_{\rho} = f|_{a_1=\ell_1, \dots, a_{|\rho|}=\ell_{|\rho|}}$$

*This is a cofactor of  $f$ , with more than a single variable assigned.*

For example, given a function  $f(x, y) = x \wedge y$  in Figure 7(a) and a partial assignment  $\rho = \{(x, \top)\}$ , the partial evaluation of  $f$  is  $f|_{\rho} = f|_{x=\top} = f(\top, y) = \top \wedge y = y$ .

A partial assignment  $\rho'$  is *more complete than*  $\rho$  if  $\rho \subseteq \rho'$ ; the most complete such assignment is called *total*. The total assignment for a function provides a value for every variable in this function. Thus, an assignment  $\rho = \{(x, \perp), (y, \top)\}$  is more complete than  $\rho' = \{(x, \perp)\}$ , and is a total assignment for any function on two variables. A partial assignment  $\rho$  is said to be *unambiguous* if  $f|_{\rho}$  is constant, and is said to be *minimally unambiguous* if, for any  $\rho' \subset \rho$ ,  $f|_{\rho'}$  is non-constant. For example, for a function  $f(x, y) = x \wedge y$  in Figure 7(a), a partial assignment  $\rho = \{(x, \perp), (y, \top)\}$  is unambiguous, since  $f|_{\rho} = f|_{x=\perp, y=\top} = \perp$ ; and a partial assignment  $\rho' = \{(x, \perp)\}$  is minimally unambiguous, since  $f|_{\rho'} = f|_{x=\perp} = \perp$ . An unambiguous assignment for a function does not need to provide values for each variable of this function; thus, every total assignment is an unambiguous assignment, whereas the converse is not true. Note that in a decision diagram, a minimally unambiguous partial assignment corresponds to a path from the root of the diagram to a terminal node.

## 5.2 Implementation of mv-sets

Here we discuss the implementation of the mv-set library on top of MDDs. We begin by defining the mv-set layer, and then describe the direct implementation of the MDD package in Section 5.2.2 and the implementation of MDDs on top of the standard ADD library [4] in Section 5.2.3.

### 5.2.1 The mv-set library on top of MDDs

Some of the functions of the mv-set library implementation on top of MDDs are given in Figure 8. We assume that each `MvSet`  $\mathbb{S}$  has a field `\mathbb{S}.dd`, for the decision diagram representation of this mv-set, and a function `createMvSetFromMDD` that allows to convert between the two representations. We use function names prefixed with “MDD” to indicate MDD library functions. As seen in Figure 8, there is a one-to-one correspondence between MDD and mv-set library functions `constant` and `ptwiseApply`. The

```

function constant(Val  $\ell$ ) : MvSet
    return createMvSetFromMDD(MDDconstant( $\ell$ ))

function ptwiseApply (Oper  $op$ , MvSet  $\mathbb{S}$ , MvSet  $\mathbb{T}$ ) : MvSet
    return createMvSetFromMDD(MDDptwiseApply( $op$ ,  $\mathbb{S}$ .dd,  $\mathbb{T}$ .dd))

function point (Vector  $\vec{s}$ , Vector  $\vec{t}$ , Val  $\ell$ ) : MvSet
    return ptwiseApply( $\wedge$ , point( $\vec{s}$ ,  $\top$ ), exchange(point( $\vec{t}$ ,  $\ell$ ), SourceVars, DestVars))

function backwardImg (MvSet  $\mathbb{S}$ , MvRelation  $\mathbb{R}$ ) : MvSet
    return existAbstract (ptwiseApply ( $\wedge$ ,  $\mathbb{R}$ , exchange ( $\mathbb{S}$ , SourceVars, DestVars), DestVars))

Global Var[] SourceVars, DestVars

```

Figure 8: Partial implementation of the mv-set library using MDD operations.

same correspondence also holds for other mv-set functions, such as `projection` and `cofactor`. Two functions of the mv-set library, a 3-parameter `point` and `backwardImg`, are defined using low-level routines `exchange` and `existAbstract`:

- Function `exchange(MvSet  $\mathbb{S}$ , Var[] oldVars, Var[] newVars)` takes a decision diagram representation of an mv-set  $\mathbb{S}$  and replaces labels of variables from `oldVars` into those of `newVars`.
- Function `existAbstract(MvSet  $\mathbb{S}$ , Var[] vars)` computes  $\exists \text{vars}[0], \text{vars}[1], \dots \cdot \mathbb{S}$  – the existential quantification used in the computation of backward and forward image.

`exchange` and `existAbstract` correspond one-to-one to their MDD library counter-parts.

The function `cofactor` performs partial evaluation of an mv-set  $\mathbb{S}$ . A partial assignment  $\rho$  is represented by a Vector  $\vec{s}$ : for any variable  $v$ , if  $(v, \ell) \in \rho$  then  $\vec{s}[v] = \ell$ ; if  $v$  is not assigned any value by  $\rho$ , then  $\vec{s}[v] = \text{null}$ .

### 5.2.2 MDD library

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 [39]). The public methods required for implementation of the mv-set library are: `MDDconstant`, `MDDprojection` and `MDDpoint`, to construct an MDD from an mv-set; `MDDptwiseApply`, to compute  $\wedge$ ,  $\vee$  and  $\neg$  of MDDs; `MDDexistAbstract`, to existentially quantify over variables; and `MDDcofactor` to compute a cofactor. These functions use a private method `makeUnique`, defined in Figure 9, to guarantee uniqueness, and thus reducedness [40], of MDD diagrams.

`MDDconstant`, `MDDprojection` and `MDDpoint` ensure orderedness of MDDs while they are being constructed, and the other functions preserve it. `MDDptwiseApply` is shown in Figure 9. Note that it interfaces with the lattice library by calling the method `Lattice.doOp` to compute  $\wedge$  or  $\vee$  of two terminal nodes.

```

function makeUnique(Var name, MDD[] children) : MDD
    find (create if not found) a node  $u$  s.t.
         $\text{var}(u) = \text{name} \wedge \forall \ell \cdot \text{child}_\ell(u) = \text{children}(\ell)$ 
    return  $u$ 

function MDDptwiseApply(Oper op, MDD  $u_1$ , MDD  $u_2$ ) : MDD
// applies the logic operation op to the MDDs  $u_1$  and  $u_2$ 
    Global int[ $u_1$ ][ $u_2$ ]  $G$ 
    return apply'(op,  $u_1$ ,  $u_2$ )

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

```

Figure 9: The MDD algorithm for MDDptwiseApply for binary operators, together with its helper functions. MDDptwiseApply for unary operators is defined similarly.

### 5.2.3 MDDs on Top of ADDs

Algebraic Decision Diagrams (ADDs) [4] are multi-terminal decision diagrams where each node has a boolean branching factor. In model-checking, ADDs are typically used for representing enumerated types, and are part of the CUDD decision diagram library [39].

Encoding of an MDD into an ADD follows the same algorithm as encoding of enumerated types and integer subranges into BDDs [31]. An ADD for a multi-valued logic  $L$  has up to  $|L|$  terminal nodes, one for each element of the logic. To turn a multi-valued variable into a boolean variable, we can use the

concept of join-irreducibility. Recall from Section 2.1 that every element of a logic can be represented as a set of join-irreducibles of this logic. For example, the join-irreducibles of the logic **3** are M and T, so each element of the logic can be uniquely represented as follows:  $F = \emptyset$ ,  $M = \{M\}$  and  $T = \{M, T\}$ . The characteristic function for this representation allows to encode each value  $\ell \in L$  using  $|\mathcal{J}(L)|$  boolean variables, one for each join-irreducible. For example, each 3-valued variable  $x$  can be encoded using a pair of variables  $x_0$  and  $x_1$ :  $x = F$  is represented as  $(\perp, \perp)$  and  $(\perp, \top)$ ,  $x = M$  as  $(\top, \perp)$ , and  $x = T$  as  $(\top, \top)$ . With this encoding, the ADD corresponding to the MDD in Figure 7(a) is shown in Figure 7(b).

The number of variables in an ADD is  $\mathcal{J}(L)$  times larger than that of the corresponding MDD. Clearly, other encodings of MDDs into ADDs are also possible, with the ratio between the number of variables in an ADD and the one in the corresponding MDD ranging from  $\lceil \log_2(|L|) \rceil$  to  $|L|$ .

## 6 mv-set Implementation Using MBTDDs

In this section we discuss alternatives for encoding operations on multi-valued sets using multi-valued branching boolean-terminal diagrams (MBTDDs). The major advantages of this approach over the one described in Section 5 is that for a fixed logic  $L$ , all operations where one operand is a constant take  $O(1)$  time, and that the model-checking algorithm can potentially be distributed over  $|\mathcal{J}(L)|$  machines.

The naive approach is to represent a multi-valued set as a collection of classical sets, one for each value of the logic. For example,  $\llbracket \text{Connected} \rrbracket$  (see Figure 3(a), Section 2.2) can be encoded using four classical sets as follows:

$\{\text{CONNECT}\}$	// for value TT
$\{\text{RINGTONE}\}$	// for value TF
$\emptyset$	// for value FT
$\{\text{IDLE}, \text{DIALTONE}\}$	// for value FF

Suppose we are interested in computing a union of this set and  $\llbracket \text{Offhook} \rrbracket$ , shown in Figure 3(b). From the logic we know that  $a \vee b = \text{TT}$  iff  $a = \text{TT}$ , or  $b = \text{TT}$ , or both  $a = \text{TF}$  and  $b = \text{FT}$ ; thus the following is the computation of the TT set in  $\llbracket \text{Connected} \rrbracket \vee \llbracket \text{Offhook} \rrbracket$ :

$\{\text{CONNECT}\} \cup$	// TT set of $\llbracket \text{Connected} \rrbracket$
$\{\text{CONNECT}, \text{DIALTONE}\} \cup$	// TT set of $\llbracket \text{Offhook} \rrbracket$
$(\{\text{RINGTONE}\} \cap \emptyset) \cup$	// TF set of $\llbracket \text{Connected} \rrbracket$ and FT set of $\llbracket \text{Offhook} \rrbracket$
$(\emptyset \cap \emptyset)$	// FT set of $\llbracket \text{Connected} \rrbracket$ and TF set of $\llbracket \text{Offhook} \rrbracket$

which results in  $\{\text{CONNECT}, \text{DIALTONE}\}$ . Computations of the remaining sets are done similarly, driven by *inverse tables*, i.e., sets of values of operands that yield the desired value in the result. Our first implementation of the model-checker [19] was based on this encoding.

In Section 5.2.3 we used the fact that each element of the lattice can be uniquely represented as a subset of join-irreducibles to efficiently implement MDDs on top of ADDs. The same idea is used in the rest of this section: we first show how to use this encoding to perform the required mv-set functions and recover the complete mv-set, and then proceed with the implementation of the mv-set library.

## 6.1 Encoding mv-sets Using $j$ -cuts

As indicated in Section 2.1, each element of a lattice can be encoded using the join-irreducible elements below it. We aim to use join-irreducibles to provide a “prime factorization” for mv-sets.

**Lemma 1** [23] *Let  $j$ ,  $x$ , and  $y$  be elements of a distributive lattice  $(\mathcal{L}, \sqsubseteq)$ , with  $j$  being a join-irreducible. Then  $j \sqsubseteq x \sqcup y$  iff  $j \sqsubseteq x$  or  $j \sqsubseteq y$ .*

**Lemma 2** *Let  $\ell$  be any element of a quasi-boolean lattice  $(\mathcal{L}, \sqsubseteq)$ . Then*

$$\{\neg x \mid x \in \mathcal{L}, \ell \sqsubseteq x\} = \{x \mid x \in \mathcal{L}, x \sqsubseteq \neg\ell\}$$

**Proof:**

$$\begin{aligned} & y \in \{\neg x \mid x \in \mathcal{L}, \ell \sqsubseteq x\} \\ \Leftrightarrow & \text{double negation} \\ & \neg(\neg y) \in \{\neg x \mid x \in \mathcal{L}, \ell \sqsubseteq x\} \\ \Leftrightarrow & \text{set specification} \\ & \ell \sqsubseteq \neg y \\ \Leftrightarrow & \text{negation is order-reversing; double negation} \\ & y \sqsubseteq \neg\ell \\ \Leftrightarrow & \text{set specification} \\ & y \in \{x \mid x \in \mathcal{L}, x \sqsubseteq \neg\ell\} \end{aligned}$$

□

The above two lemmas describe properties of join-irreducible elements that we exploit in our representation of mv-sets.

**Definition 16** *Up-sets and down-sets of lattice elements are defined as follows:*

$$\begin{aligned} \uparrow \ell &\triangleq \{x \mid x \in \mathcal{L} \cdot \ell \sqsubseteq x\} && \text{(Up-set of } \ell) \\ \downarrow \ell &\triangleq \{x \mid x \in \mathcal{L} \cdot \sqsubseteq \ell\} && \text{(Down-set of } \ell) \end{aligned}$$

Given a logic  $L$ , we aim to encode mv-sets defined over  $L$  using a collection of  $j$ -cuts for  $j \in \mathcal{J}(L)$ . We then show that the  $j$ -cuts of an mv-set contain sufficient information to rebuild the original mv-set, and prove that all the necessary operations for symbolic model-checking can be easily carried out in this encoding.

Our first result, which follows trivially from Theorem 2, states that the membership degree of an element in an mv-set is the join of all join-irreducibles whose cuts contain the element.

**Theorem 5** *For any mv-set  $\mathbb{S}$  over the logic  $L$ ,*

$$\mathbb{S}(x) = \bigsqcup \{j \mid j \in \mathcal{J}(L), x \in \uparrow_j(\mathbb{S})\}$$

We now define the operations over mv-sets in terms of operations over their  $j$ -cuts.

**Theorem 6** *The  $\ell$ -cut of a multi-valued intersection is the intersection of the  $\ell$ -cuts of the individual mv-sets. For  $j \in \mathcal{J}(L)$ , the  $j$ -cut of a multi-valued union is the union of the  $j$ -cuts.*

$$\begin{aligned} \forall \ell \in L \cdot \uparrow_\ell(\mathbb{S} \cap_L \mathbb{T}) &= \uparrow_\ell(\mathbb{S}) \cap \uparrow_\ell(\mathbb{T}) && \text{(Cut-intersection)} \\ \forall j \in \mathcal{J}(L) \cdot \uparrow_j(\mathbb{S} \cup_L \mathbb{T}) &= \uparrow_j(\mathbb{S}) \cup \uparrow_j(\mathbb{T}) && \text{(Cut-union)} \end{aligned}$$

**Proof:**

We show both equations by set extensionality:

$$\begin{array}{ll} x \in \uparrow_\ell(\mathbb{S} \cap_L \mathbb{T}) & x \in \uparrow_j(\mathbb{S} \cup_L \mathbb{T}) \\ \Leftrightarrow \text{Definition 9} & \Leftrightarrow \text{Definition 9} \\ \ell \sqsubseteq (\mathbb{S} \cap_L \mathbb{T})(x) & j \sqsubseteq (\mathbb{S} \cup_L \mathbb{T})(x) \\ \Leftrightarrow \text{multi-valued intersection} & \Leftrightarrow \text{multi-valued union} \\ \ell \sqsubseteq \mathbb{S}(x) \cap \mathbb{T}(x) & j \sqsubseteq \mathbb{S}(x) \sqcup \mathbb{T}(x) \\ \Leftrightarrow \text{lattice properties} & \Leftrightarrow \text{Lemma 1} \\ \ell \sqsubseteq \mathbb{S}(x) \text{ and } \ell \sqsubseteq \mathbb{T}(x) & j \sqsubseteq \mathbb{S}(x) \text{ or } j \sqsubseteq \mathbb{T}(x) \\ \Leftrightarrow \text{Definition 9} & \Leftrightarrow \text{Definition 9} \\ x \in \uparrow_\ell(\mathbb{S}) \text{ and } x \in \uparrow_\ell(\mathbb{T}) & x \in \uparrow_j(\mathbb{S}) \text{ or } x \in \uparrow_j(\mathbb{T}) \\ \Leftrightarrow \text{set theory} & \Leftrightarrow \text{set theory} \\ x \in \uparrow_\ell(\mathbb{S}) \cap \uparrow_\ell(\mathbb{T}) & x \in \uparrow_j(\mathbb{S}) \cup \uparrow_j(\mathbb{T}) \end{array}$$

□

Formulation of complementation uses two observations. First is a trivial corollary of Lemma 2, indicating that multi-valued complement turns each  $j$ -cut into a  $\neg j$ -clip:

**Lemma 3** *Each  $j$ -cut of an mv-set  $\mathbb{S}$  is the  $\neg j$ -clip of the complement of  $\mathbb{S}$ :*

$$\uparrow_j(\mathbb{S}) = \downarrow_{\neg j}(\overline{\mathbb{S}})$$

Second, every up-set of a join-irreducible  $j$  is the complement (in the elements of the logic) of the down-set of some meet-irreducible  $m$ :

**Lemma 4** *Let  $L$  be a quasi-boolean logic. Then, there exists a bijection  $f : \mathcal{J}(L) \rightarrow \mathcal{M}(L)$  such that  $\forall j \in \mathcal{J}(L) \cdot \uparrow j = L \setminus \downarrow f(j)$  and  $\forall m \in \mathcal{M}(L) \cdot \downarrow m = L \setminus \uparrow f^{-1}(m)$ .*

**Proof:**

Since  $L = ((\mathcal{L}, \sqsubseteq), \neg)$  is a quasi-boolean logic,  $(\mathcal{L}, \sqsubseteq)$  is a finite distributive lattice and thus, by Proposition 9.4 of [23], there exists a bijection  $g : \mathcal{J}(L) \rightarrow \{\downarrow m \mid m \in \mathcal{M}(L)\}$  given by  $g(j) = \mathcal{L} \setminus \uparrow j$ . Further, since  $\sqcup : \{\downarrow m \mid m \in \mathcal{M}(L)\} \rightarrow \mathcal{M}(L)$  is a bijection, we obtain a bijection  $f : \mathcal{J}(L) \rightarrow \mathcal{M}(L)$  where  $f(j) = \sqcup(\mathcal{L} \setminus \uparrow j)$ . It is easy to show that its inverse  $f^{-1}$  is given by  $f^{-1}(m) = \cap(\mathcal{L} \setminus \downarrow m)$ , where  $m \in \mathcal{M}(L)$ . Converting this back into logic terms, we get  $f^{-1}(m) = \cap(L \setminus \downarrow m)$ , where  $m \in \mathcal{M}(L)$ .

Now we show that  $\uparrow j = L \setminus \downarrow f(j)$  and  $\downarrow m = L \setminus \uparrow f^{-1}(m)$  by set extensionality:

$x \in L \downarrow f(j)$	$x \in L \uparrow f^{-1}(m)$
$\Leftrightarrow$ Definition 16	$\Leftrightarrow$ Definition 16
$x \not\sqsubseteq f(j)$	$f^{-1}(m) \not\sqsubseteq x$
$\Leftrightarrow$ definition of $f$	$\Leftrightarrow$ definition of $f^{-1}$
$x \not\sqsubseteq \sqcup(L \uparrow j)$	$\sqcap(L \downarrow m) \not\sqsubseteq x$
$\Leftrightarrow$ lattice properties and contraposition	$\Leftrightarrow$ lattice properties and contraposition
$x \notin L \uparrow j$	$x \notin L \downarrow m$
$\Leftrightarrow$ double negation	$\Leftrightarrow$ double negation
$x \in \uparrow j$	$x \in \downarrow m$

□

We can generalize the above result to mv-sets:

**Corollary of Lemma 4** For an mv-set  $\mathbb{S}$  of some universe  $U$ ,

$$\begin{aligned} \uparrow_j(\mathbb{S}) &= U \setminus \downarrow_{f(j)}(\mathbb{S}) && \text{(Cut of join-irreducible)} \\ \downarrow_m(\mathbb{S}) &= U \setminus \uparrow_{f^{-1}(\neg m)}(\mathbb{S}) && \text{(Clip of meet-irreducible)} \end{aligned}$$

We use the above Corollary to construct the  $j$ -cut of a multi-valued complement of an mv-set:

**Theorem 7** Let  $\mathbb{S}$  be an mv-set of some  $U$ , based on a logic  $L$ , and let  $j \in \mathcal{J}(L)$ . The  $j$ -cut of the multi-valued complement of  $\mathbb{S}$  is the classical complement (in  $U$ ) of the  $f^{-1}(\neg j)$  cut of  $\mathbb{S}$ :  $\uparrow_j(\overline{\mathbb{S}}) = U \setminus \uparrow_{f^{-1}(\neg j)}(\mathbb{S})$ .

**Proof:**

$$\begin{aligned} &\uparrow_j(\overline{\mathbb{S}}) \\ &= \text{Lemma 3} \\ &\downarrow_{\neg j}(\overline{\mathbb{S}}) \\ &= \text{Involution} \\ &\downarrow_{\neg j}(\mathbb{S}) \\ &= \text{By Corollary of Lemma 4, since } \neg j \in \mathcal{M}(L), \text{ and by clip of meet-irreducible} \\ &U \setminus \uparrow_{f^{-1}(\neg j)}(\mathbb{S}) \end{aligned}$$

□

These results allow us to represent each mv-set compactly as a family of its  $j$ -cuts. Theorem 5 guarantees that this representation of mv-sets preserves all information; it also tells us how to recover mv-set membership from the  $j$ -cuts.

The advantage of this representation is in the computation of multi-valued intersections and unions. If mv-sets were represented by families of pieces, i.e., indexed by all logic values, then intersection and union would both take as many as  $O(|L|^2)$  classical set operations. In contrast, the cut-intersection and the cut-union theorems imply that our representation requires just  $O(|\mathcal{J}(L)|)$  classical set operations.

```

function constant(Val  $\ell$ ) : MvSet
  foreach  $j \in \mathcal{J}(L)$ 
    if ( $j \sqsubseteq \ell$ ) then result[ $j$ ] = MBTconstant( $\top$ )
    else result[ $j$ ] = MBTconstant( $\perp$ )
  return createMvSetFromMBTDD(result)

function projection(Var  $v$ ) : MvSet
  foreach  $j \in \mathcal{J}(L)$ 
    result[ $j$ ] = projectionElement( $v$ , UpSet[ $j$ ])
  return createMvSetFromMBTDD(result)

function projectionElement(Var  $v$ , Val[] UpSet) : MBTDD
  result0 =  $\perp$ 
  foreach  $u_j \in \text{UpSet}$ 
    result $i+1$  = MTBptwiseApply( $\vee$ , MBTpoint( $v := u_j, \top$ ), result $i$ )
  return result $i+1$ 

function point(Vector  $\vec{x}$ , Val  $\ell$ ) : MvSet
  foreach  $j \in \mathcal{J}(L)$ 
    if ( $j \sqsubseteq \ell$ ) then result[ $j$ ] = MBTpoint( $x, \top$ )
    else result[ $j$ ] = MBTconstant( $\perp$ )
  return createMvSetFromMBTDD(result)

function ptwiseApply (Oper  $op$ , MvSet  $\mathbb{S}$ , MvSet  $\mathbb{T}$ ) : MvSet
  foreach  $j \in \mathcal{J}(L)$ 
    result[ $j$ ] = MBTpointwiseApply( $op$ ,  $\mathbb{S}$ .dd[ $j$ ],  $\mathbb{T}$ .dd[ $j$ ])
  return createMvSetFromMBTDD(result)

function ptwiseApply (Oper  $\neg$ , MvSet  $\mathbb{S}$ ) : MvSet
  foreach  $j \in \mathcal{J}(L)$ 
    result[ $j$ ] = MBTpointwiseApply( $\neg$ ,  $\mathbb{S}$ .dd[neg[ $j$ ]])
  return createMvSetFromMBTDD(result)

Global Val[] UpSet = computeUp( $\mathcal{J}(L)$ )
Global Val[] neg

```

Figure 10: Implementation of the mv-set library using MBTDD operations.

## 6.2 Implementation of mv-sets

A single MBTDD computes a boolean-output function. By creating an MBTDD for each  $j$ -cut of a multi-valued function, we can represent that function as a *vector* of MBTDDs. These vectors and their semantics are defined formally as follows, using Theorem 5:

**Definition 17** Let  $\hat{u}$  be an MBTDD-vector indexed by the join-irreducibles of a logic  $L$ . Each  $\hat{u}[j]$  defines a boolean-output function by the semantics of Definition 14. The function defined by the vector is computed as

$$f^{\hat{u}} = \bigsqcup_{j \in \mathcal{J}(L)} f^{\hat{u}[j]} \sqcap j$$

and conversely, for all  $j$ ,

$$f^{\hat{u}[j]} = \uparrow_j (f^{\hat{u}})$$

The size of an MBTDD-vector is defined as follows:

$$\text{size}(\hat{u}) = |\{v \mid v \text{ is reachable from } \hat{u}[j] \text{ for some } j \in \mathcal{J}(L)\}|$$

The mv-set library is implemented on top of MBTDDs as described in Figure 10. As in the MDD implementation, we assume that each `MvSet`  $\mathbb{S}$  has a field `S.dd`, for a decision diagram representation of this mv-set. However, here `dd` is an *array* of decision diagrams, one for each  $j$ -cut of  $\mathbb{S}$ : it corresponds to  $\hat{u}$  from Definition 17. Function `createMvSetFromMBTDD` allows the construction of an mv-set from its decision diagram representation. In this figure, all function names prefixed with “MBT” are part of the MBTDD package. Functions `backwardImg` and `point` on three parameters are the same as shown in Figure 8 and thus are omitted here. In Figure 10,  $\mathcal{J}(L)$  is an ordered list of join-irreducible elements of the logic, and `UpSet` is the list of up-sets of join-irreducibles, indexed by  $\mathcal{J}(L)$ :

$$\text{UpSet}[j] \triangleq \uparrow j = \{\ell \in L \mid j \sqsubseteq \ell\}$$

`UpSet` is computed by a function `computeUp`. We use the notation  $v \vec{=} \ell$  to denote an element of type `Vector` corresponding to a partial assignment  $\rho$  consisting of a single tuple  $(v, \ell)$ .

Additional functions, such as `cofactor`, `exchange` and `existAbstract` follow the same pattern as that of `ptwiseApply` in Figure 10. Note that all operations are performed w.r.t.  $j$ -cuts of mv-sets. Handling of  $\wedge$  and  $\vee$  follows from Theorem 6. The same theorem also serves as proof of correctness for `existAbstract`, since it is a disjunction over values of existentially-quantified variables. Handling of the  $\neg$  operator uses a precomputed `neg` table for a quick lookup of  $f^{-1}(\neg j)$ , for any join-irreducible element  $j$ , as required by Theorem 7. It is precomputed from the constructive proof of Lemma 4 where  $f$  is specified: first find  $f$  using  $f(j) = \sqcup(L \setminus \uparrow j)$ , then fill in the `neg` table using  $\text{neg}[\neg f(j)] = j$ .

Figure 11(a) shows a T-cut and an M-cut for the representation of a function  $x \wedge y$  over a logic  $\mathbf{3}$  using an MBTDD-vector. The direct implementation of the MBTDD package is very similar to that of the MDD package described in Section 5.2.2, and is omitted here for brevity. MBTDD-vectors can also be easily implemented on top of a standard BDD library, such as CUDD [39]. Each MBTDD in the vector is represented using a BDD following the same process as described in the MDD to ADD conversion (see Section 5.2.3). For example, Figure 11(b) shows a BDD-vector representation for the function  $f = x \wedge y$ . Note that if had we used a different variable ordering, i.e.,  $x_0 > y_0 > x_1 > y_1$ , then we would have been able to achieve an even more compact representation via the structure sharing between the two  $j$ -cuts of this function.

## 7 Experimental Comparisons

In the first part of this paper we defined mv-sets and discussed how to implement them using four decision diagram encodings: as MDDs and ADDs (Section 5) and as MBTDD-vectors and BDD-vectors

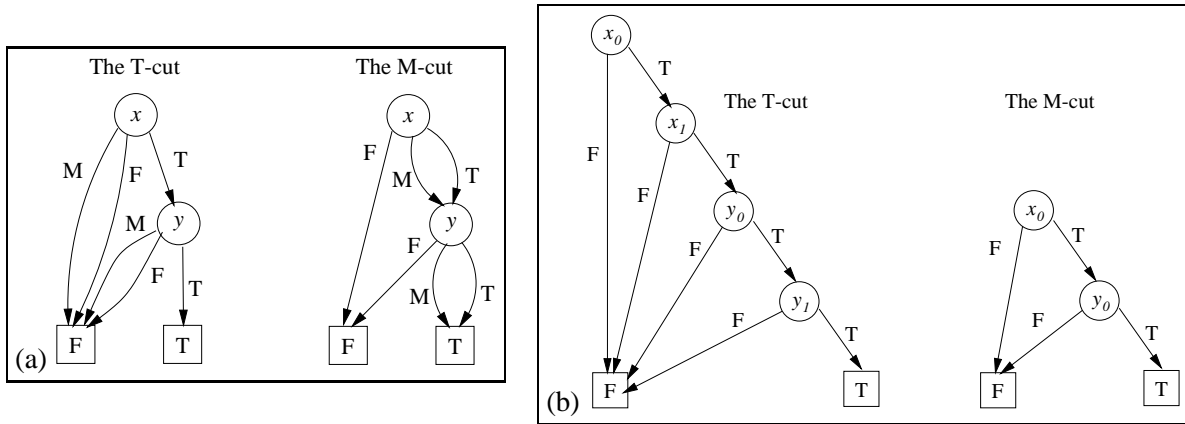


Figure 11: Representation of function  $f = x \wedge y$  in logic **3**: (a) using an MBTDD-vector; (b) using a BDD-vector.

Name	Description
<b>abp4</b>	alternating bit protocol
<b>dartes</b>	communication protocol of an Ada program
<b>dme2-16</b>	distributed mutual exclusion protocol
<b>dpd75</b>	dining philosophers
<b>ftp3</b>	file transfer protocol
<b>furnace17</b>	remote furnace program
<b>key10</b>	keyboard/screen interaction in a window manager
<b>mmgt20</b>	distributed memory manager
<b>over12</b>	automated highway system overtake protocol

Table 3: List of NuSMV models used in the experiments.

(Section 6). In this section we describe experiments we conducted to study trade-offs between the four implementations of mv-sets, in terms of space requirements and the running time. In Section 7.1, we outline the types of experiments we perform. We propose to study the impact of two factors on the size of underlying decision diagram representation: (a) encoding a characteristic function of an mv-set as a multi-valued function or as a vector of boolean functions (we refer to this as “the number of terminal nodes”, meaning that if the underlying decision diagram has two terminal nodes,  $\top$  and  $\perp$ , then the characteristic function is boolean; otherwise, it is multi-valued); and (b) the branching factor of the decision diagrams. Each factor is analyzed in detail in Sections 7.2 and 7.3, respectively. In Section 7.4, we relate the decision diagram size and the running time of mv-set operations. This relationship is further illustrated on case studies in Section 8.

## 7.1 Methodology

Theoretical estimates of space and time complexity for decision diagrams are very loose. Further, it has been shown [41] that the performance of decision diagrams for most *randomly chosen* functions is very poor. Yet, it is generally known that functions used in formal verification have performance that is considerably better than the worst-case. To study multi-valued decision diagrams, we propose to use benchmarks that represent functions occurring in model checking, i.e., those similar in structure to classical models but with some non-boolean inputs and outputs. We anticipate two principal ways in which multi-valued models arise in model-checking. The first is from *inconsistency*: we may want to combine several slightly different classical models while preserving all of their differences. The system **callee-m** in Figure 2(c) is an example of a model with inconsistencies. The second cause of multi-valued functions is *partial* systems. In a fully-specified system, each transition is either present or absent, whereas in a partial system, some transitions can be marked as *unknown*, indicated by the value M of logic **3**. Finite total orders with more values may be used for different levels of partiality [9, 34].

As a source of classical models, we adopt benchmarks used in a previous performance study of symbolic model-checking [42]; these represent *transition relations* of several NuSMV [20] models and are expressed in the modeling language of NuSMV. The list of the models we use is given in Table 3. For brevity, in this paper we present partial experimental results for models **abp4** and **dme2-16**. Results for all other models are identical, except for the absolute size of the decision diagram representation, and are available on our web site, <http://www.cs.toronto.edu/fm/experiments.html>.

After constructing multi-valued models, we represent them using decision diagrams and then conduct experiments, measuring the size of the representations in terms of the *number of nodes*. This metric can be easily obtained; further, storage requirements can be computed by simply multiplying the branching factor of a given decision diagram by its number of nodes. Finally, the number of nodes allows us to compare running times for different mv-set operations, as described in Section 7.4.

As shown in Table 2 in Section 4, the four representations of mv-sets vary along two axes: the branching factor and the number of terminal nodes. We are interested in exploring the independent effects of these two axes of variation. We therefore want to design experiments that allow us to measure the effect of varying the number of terminal nodes, i.e. one row of Table 2 (see Section 7.2) and the branching factor, i.e., one column of Table 2 (see Section 7.3).

Finally, we perform each experiment on a number of multi-valued logics. We restrict ourselves to logics with at most 16 elements (to ensure tractability of the experiments). The logics we use are listed in Table 4. The minimum number of join-irreducible elements for a logic  $L$  is  $\lceil \log_2(|L|) \rceil$ . For example, logic **2x2x2x2** has 16 elements and 4 join-irreducibles. The maximum number of join-irreducibles for a logic  $L$  is  $|L| - 1$ . This number is attained when the elements of the logic form an FTO – a finite total order<sup>2</sup>, such as in logic **16** which has 15 join-irreducibles. For our experiments, we pick logics from each side of the spectrum and some from the middle, making sure that we have several logics of different sizes with the same number of join-irreducible elements, e.g. **5**, **6C** and **3x3**, where each logic has 4 join-irreducibles.

---

<sup>2</sup>An FTO is a lattice  $(\mathcal{L}, \sqsubseteq)$ , where  $\sqsubseteq$  is a *total order*. Logic **5**, given in Figure 12(b), is an FTO.

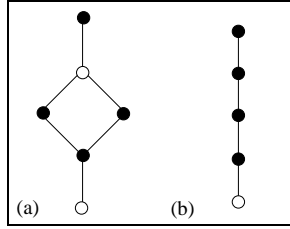


Figure 12: Some logics used in the experiments: (a) Logic **6C**; (b) Logic **5**. Join-irreducible elements are indicated by filled circles.

Name	Size	No. of join-irreducibles	Notes
<b>2</b>	2	1	Classical logic, see Figure 1(a)
<b>3</b>	3	2	Logic of uncertainty, see Figure 1(b)
<b>2x2</b>	4	2	Logic of disagreement, see Figure 1(c)
<b>4</b>	4	3	Finite total order with 4 elements
<b>5</b>	5	4	Finite total order with 5 elements, see Figure 12(b)
<b>2x3</b>	6	3	Product of <b>2</b> and <b>3</b>
<b>6C</b>	6	4	See Figure 12(a), used in [15]
<b>6</b>	6	5	Finite total order with 6 elements
<b>2x2x2</b>	8	3	Product of three classical logics ( <b>2</b> )
<b>3x3</b>	9	4	Product of two three-valued logics ( <b>3</b> ), see Figure 1(e)
<b>2x2x2x2</b>	16	4	Product of four classical logics ( <b>2</b> )
<b>16</b>	16	15	Finite total order with 16 elements

Table 4: Logics used in our experiments.

## 7.2 Experiments 1 & 2: Number of Terminal Nodes

The goal of these experiments is to study the effect of the number of terminal nodes on the size of the resulting decision diagram. Thus, we hold the branching factor of our diagrams constant and vary the number of terminal nodes. In particular, we let the branching factor be equal to the size of the logic, and thus study the trade-offs between the MDD and MBTDD implementations of mv-sets. Alternatively, we could have chosen to branching factor to be equal to 2 and studied the trade-offs between ADDs and BDDs.

### 7.2.1 Experiment Design

All model-checking benchmarks are based on boolean, i.e., crisp, functions; however, MDDs and MBTDD-vectors yield exactly the same sizes for such functions; further, the size does not change even when we allow the input to be multi-valued. This indicates that the number of terminal nodes does not affect the representation of boolean-output functions. However, if we go back to boolean benchmarks and instead make the output multi-valued, the MDD and the MBTDD-vector implementations exhibit different behaviour, which we study in this section. Functions with crisp input and multi-valued output arise from models where disagreements or partialities occur *only on transitions*, and likely encompass the majority

```

function DIS (Logic  $L$ , int  $n$ )
  //  $f$  is a crisp function over  $L$ , defined globally.
   $g_0 = f$ 
  foreach  $1 \leq i < n$ 
    compute  $\rho$  – a minimal unambiguous partial assignment s.t.  $g_i|_{\rho} \in \{\top, \perp\}$ 
    pick some  $\ell \in (L \setminus \{\top, \perp\})$ 
    compute  $g_{i+1}$  such that for each total assignment  $\rho'$ ,
      
$$g_{i+1}|_{\rho'} = \begin{cases} \ell & \text{if } \rho \subseteq \rho' \\ g_i|_{\rho'} & \text{otherwise} \end{cases}$$

  return  $g_n$ 

```

Figure 13: Adding disagreements to a crisp function  $f$ .

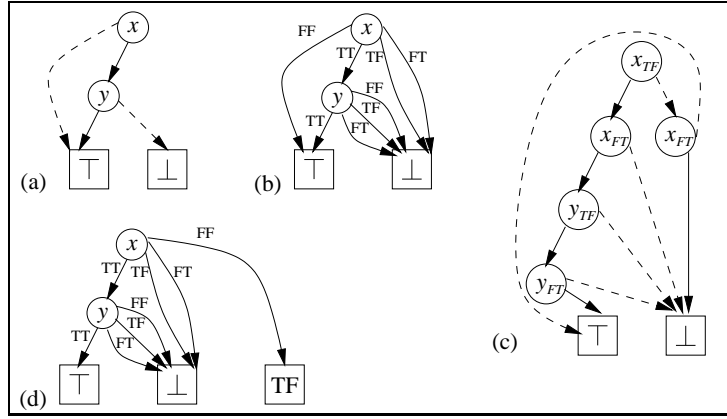


Figure 14: A boolean function  $\neg x \vee y$ : (a) as a BDD; (b) as an MBTDD-vector in  $\mathbf{2x2}$ ; (c) as a BDD-vector in  $\mathbf{2x2}$ ; (d) result of introducing a TF disagreement to (b).

of multi-valued model-checking problems.

In order to construct models with multi-valued transition relation, we start with a classical model and then change some of the transition values. Depending on the logic, this change may represent either a disagreement (when the logic is a product of several classical ones), an underspecification (in the case of an FTO), or a combination of the two. A global transition relation is typically computed via synchronous or asynchronous parallel compositions of transition relations of smaller models. Thus, a disagreement on a single local transition may result in a large number of changes to the global transition relation. With this in mind, we propose the following method for creating multi-valued models.

Let  $\delta$  be a function representing a transition relation of a classical model, and  $f = \alpha_L(\delta)$  be the embedding of  $\delta$  into  $L$ , defined via Definition 7. Multi-valued models are created by adding *disagreements* to  $f$  using a function DIS defined in Figure 13, parameterized by the logic  $L$  and the number of disagreements  $n$ . The minimal unambiguous partial assignment, used in this algorithm, is defined in Section 5.1. The algorithm is declarative – the exact computation of  $g_{i+}$  is technical and is omitted from this presentation. For example, Figure 14(b) shows a representation of a boolean function  $f$  as an MBTDD-vector over logic  $\mathbf{2x2}$ . The result of adding one disagreement where  $\rho = \{(x, \perp)\}$  and  $\ell = \text{TF}$  is a function  $g$ ,

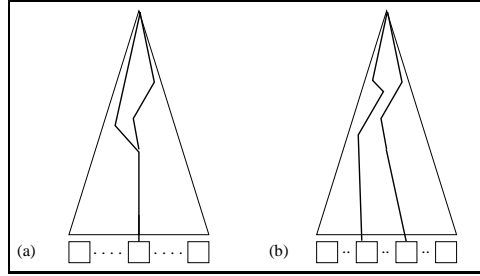


Figure 15: Illustration of Hypothesis 1 with two disagreements: (a) the same output; (b) different outputs.

shown in Figure 14(d). This function is identical to  $f$  except that the FF child of node  $x$  now points to TF.

For our experiments, we range the number of disagreements between 5 and 100, corresponding to an intuition that a model with *too* many disagreements is not yet ready for formal analysis. Given a logic  $L$  and a number of disagreements  $n$ , let  $\text{SAMPLE}(n, L)$  be a set of 100 different functions obtained by calling  $\text{DIS}(n, L)$  on  $\alpha_L(\delta)$ . For each function  $f_i \in \text{SAMPLE}(n, L)$ , we generate an MDD  $u_i$  and an MBTDD-vector  $\hat{u}_i$  such that  $f_i = f^{u_i} = f^{\hat{u}_i}$ , thus creating two new sets of functions:

$$\begin{aligned} \text{MDDSAMPLE}(n, L) &\triangleq \{u_i \mid f^{u_i} = f_i\} \\ \text{MBTDDSAMPLE}(n, L) &\triangleq \{\hat{u}_i \mid f^{\hat{u}_i} = f_i\} \end{aligned}$$

We use these sets in the experiments discussed below.

## 7.2.2 Experimental Results

Before performing the experiments, we make an initial hypothesis, formulated below, that, all other things being equal, an MDD based on a larger logic should contain more nodes than an MDD based on a smaller one:

**Hypothesis 1** *Let  $u \in \text{MDDSAMPLE}(n, L)$ ,  $v \in \text{MDDSAMPLE}(n, L')$ , and  $|L| > |L'|$ . Then we expect that  $\text{size}(u) > \text{size}(v)$ .*

For example, consider adding two disagreements  $\rho$  and  $\rho'$  with values  $\ell$  and  $\ell'$ , respectively. If  $\ell = \ell'$ , then the two paths corresponding to the disagreements must share a suffix (see Figure 15(a)). In the worst case, the shared suffix consists of just one terminal node corresponding to  $\ell$ . On the other hand, if  $\ell \neq \ell'$ , the two paths cannot have a common suffix at all (see Figure 15(b)). Thus, the higher the chance that any two disagreements have the same value, the smaller the size of the resulting diagram. Clearly, the possibility of assigning the same value to disagreements depends on the number of possible values, which in our case is  $|L| - 2$ ; therefore, the larger the logic, the larger the size of the resulting MDD.

To validate this hypothesis, we conduct the following experiment:

### Experiment 1(a):

For each logic and each possible number of disagreements, we create 100 MDDs and measure their size. The sizes of MDDs are clustered within a narrow range, so we feel justified in extracting their mean value, denoted  $\|\text{MDDSAMPLE}(n, L)\|$ , and plotting it as a single point. Henceforward we shall simply call this value “the MDD size for  $n$  and  $L$ ”. Results of the experiment are shown in Figure 16(a). For each logic,

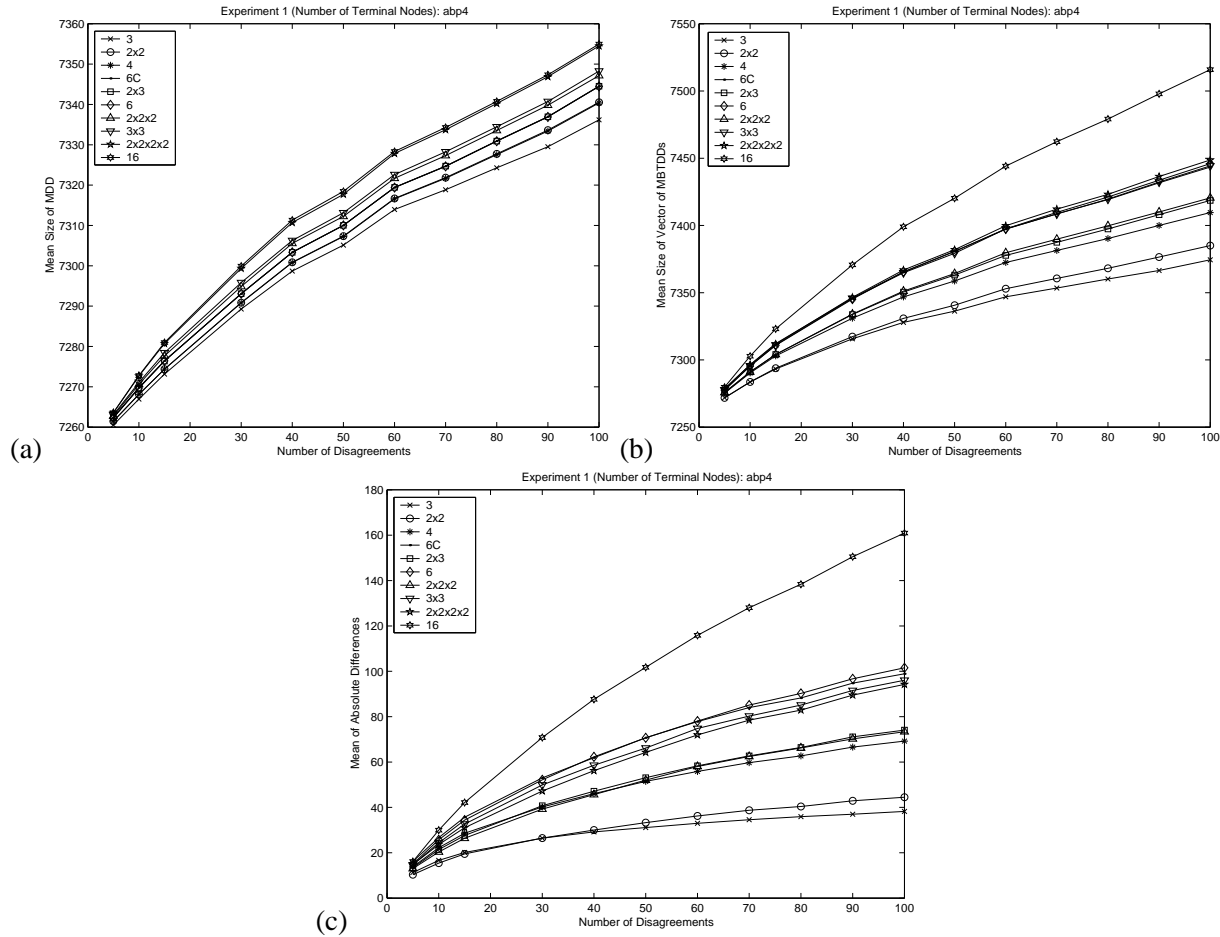


Figure 16: Graphs for Experiment 1: (a) MDD; (b) MBTDD-vector (note that  $y$ -axis is labeled differently here than in (a)); (c) The difference between the MDD and MBTDD-vector representations.

we plot the increase of the mean size of MDDs as the number of disagreements grows, for model **abp4**. Comparing the MDD size at a given number of disagreements for any two logics validates Hypothesis 1. For example, MDDs over **3** have the smallest size, whereas those over **16** and **2x2x2x2** have the largest; further, curves corresponding to those two logics (both with 16 elements) coincide.

From the shape of the curves in the graph in Figure 16(a) we conclude that, as more disagreements are added, the marginal impact of each subsequent disagreement is lessened. That is, for any  $i, j, k$  with  $i < j < k \leq 100$  and  $k - j = j - i$ ,

$$\| \text{MDDSAMPLE}(j, L) \| - \| \text{MDDSAMPLE}(i, L) \| \geq \| \text{MDDSAMPLE}(k, L) \| - \| \text{MDDSAMPLE}(j, L) \|$$

We have not explored the space of parameters beyond 100, where there may be some unexpected phenomena.  $\square$

We now turn our attention to measuring the size of MBTDD-vectors. Note that the observation about mean sizes of MDDs, made above, holds for MBTDD-vectors as well; this justifies computing

$\|\text{MBTDDSAMPLE}(n, L)\|$  and referring to it as “MBTDD-vector size”. Since each MBTDD is also an MDD, we also expect subsequent changes to have a diminishing impact on the individual MBTDDs of a vector, just as they do for single MDDs, formulated in the following hypothesis:

**Hypothesis 2** *The diminishing impact of disagreements observed for MDDs also holds for MBTDD-vectors: for any  $i, j, k$  with  $i < j < k \leq 100$  and  $k - j = j - i$ ,*

$$\begin{aligned} & \|\text{MBTDDSAMPLE}(j, L)\| - \|\text{MBTDDSAMPLE}(i, L)\| \\ & \geq \|\text{MBTDDSAMPLE}(k, L)\| - \|\text{MBTDDSAMPLE}(j, L)\| \end{aligned}$$

The number of terminal nodes in MBTDD-vectors remains set at 2, regardless of the number of logic values. Therefore, we do not anticipate the same dependence on the number of possible outputs as we observed with MDDs. However, we do predict dependence on the number of join-irreducibles, since it corresponds to the number of diagrams in the vector. For example, consider an MBTDD-vector  $\hat{u} \in \text{MBTDDSAMPLE}(n, L)$  with  $|\mathcal{J}(L)| = 2$ . The two individual MBTDDs comprising this vector share some nodes, but each contains some nodes not appearing in the other. If we compare this case with the one of  $\hat{v} \in \text{MBTDDSAMPLE}(n, L')$ , with  $|\mathcal{J}(L')| = 3$ , we note that the shared size of any *two* individual MBTDDs in this vector is close to  $\text{size}(\hat{u})$ . However, the third MBTDD is likely to have some nodes not shared with the first two and thus contributes a non-zero number of unshared nodes to the overall size of the MBTDD-vector. We formulate this intuition in the following hypothesis:

**Hypothesis 3** *Let  $u \in \text{MBTDDSAMPLE}(n, L)$ ,  $v \in \text{MBTDDSAMPLE}(n, L')$ , and  $|\mathcal{J}(L)| > |\mathcal{J}(L')|$ . Then we expect that  $\text{size}(\hat{u}) > \text{size}(\hat{v})$ .*

**Hypothesis 4** *Introducing disagreements destroys most of the sharing which holds within an MBTDD-vector.*

### Experiment 1(b):

For each logic and each possible number of disagreements, we create 100 MBTDD-vectors and measure their size. Results are given in Figure 16(b). These results allow us to make the following observations: (a) we see the same diminishing impact of subsequent disagreements in Figure 16(b), confirming Hypothesis 2; (b) comparing results of running experiments on two logics with different numbers of join-irreducibles, we see that Hypothesis 3 holds in general. For example, the curve for **16**, with 15 join-irreducibles, is above the one for **2x2x2x2**, with 4 join-irreducibles. However, when two logics have the same number of join-irreducibles, e.g., **3** and **2x2**, we see that their curves are clustered together, but do not coincide.

We now compare results in Figure 16(a) and 16(b). First, we note that for the same logic  $L$  and the same number of disagreements  $n$ , the MDD size is less than the MBTDD-vector size. We can show that this is not *necessarily* the case. Consider an example in Figure 17. Figure 17(a) shows an MDD for some crisp-input function  $h$  with outputs from logic **2x2**. This function requires 7 nodes (4 terminal and 3 non-terminal) to be represented as an MDD under the variable ordering  $x < y$ . In contrast, the same function represented as a vector of two MBTDDs (see Figure 17(b)) requires just 4 nodes: 2 terminal and 2 non-terminal. As can be seen from this simple example, the relationship we observe in the experimental data between the MDD size and the MBTDD-vector size is not universal, but only particular to our domain of interest.

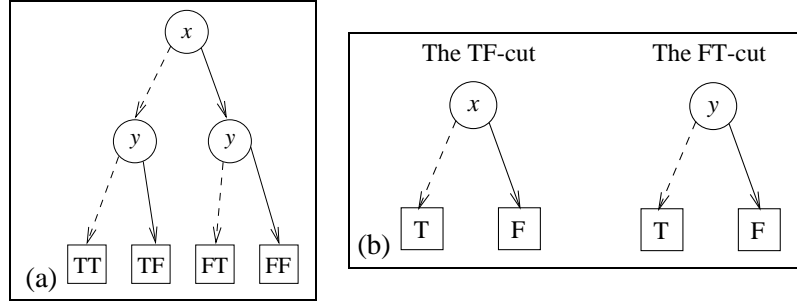


Figure 17: Representation of a crisp-input function  $h$  in  $x$  and  $y$ : (a) as an MDD (to reduce clutter, only outgoing edges for TT (solid) and FF (dashed) are shown; all others lead directly to FF); (b) as an MBTDD-vector.

Second, even though the MBTDD-vector size is greater than the MDD size, their ratio never exceeds 1.1. Without disagreements, each MBTDD is the same size as the original MDD. Were Hypothesis 4 to hold, we would expect to see the size of the MBTDD-vector to be close to  $|\mathcal{J}(L)|$  times the size of the MDD: with minimal sharing, each pair of MBTDDs is mostly disjoint, and thus the size of the MBTDD-vector is roughly the sum of the sizes of its elements. We therefore conclude that this Hypothesis does *not* hold, and the addition of disagreements leaves most sharing between elements of the MBTDD-vector intact.

Third, to get a clearer picture of the differences in size, we also graph the absolute difference between mean MDD size and mean MBTDD-vector size on a single grid. This graph, shown in Figure 16(c), allows us to observe that (a) the impact of disagreements on the *difference* in size also diminishes; and (b) for any two logics  $L, L'$  with the same number of join-irreducibles and any number of disagreements  $n$ , if  $|L'| > |L|$ , then the difference between MDD size and MBTDD-vector size is *smaller* for  $L'$  than for  $L$ : that is, the *larger* the logic, the *less* is the difference in the representation size. For example, comparing results for two logics with 4 join-irreducible elements,  $\mathbf{3x3}$  with size 9, and  $\mathbf{2x2x2x2}$  with size 16, we note that the curve corresponding to the former is above the one corresponding to the latter.

□

Results of Experiment 1(b) allow us to make a yet another conjecture: given two logics with the same number of join-irreducibles, the size of the MBTDD-vector also depends on the *size of the logic*. We illustrate this point on a concrete example. Consider two logics with the same number of join-irreducibles,  $\mathbf{3}$  and  $\mathbf{2x2}$ . However, logic  $\mathbf{3}$  has only one “non-crisp” join-irreducible (M) whereas  $\mathbf{2x2}$  has two (TF and FT). Thus, each disagreement, i.e., a change of an output from  $\perp$  to some other value of the logic, requires a change to exactly one MBTDD, the M-cut, leaving the T-cut intact. The same disagreement added to a function in  $\mathbf{2x2}$  causes a change *either* to the FT-cut or to the TF-cut; thus, as disagreements accumulate, both decision diagrams are changed in different ways, diminishing the sharing between them. Further, we observe the diminishing effect of disagreements: changing the M-cut twice has less effect than changing each of TF- and FT-cuts once. Thus, we formulate the following hypothesis:

**Hypothesis 5** *If the same value  $\ell$  is used for adding disagreements, then the size of an MBTDD-vector after adding disagreements is the same in logic  $\mathbf{2x2}$  as it is in logic  $\mathbf{3}$ , despite the differences in the structure of the two logics.*

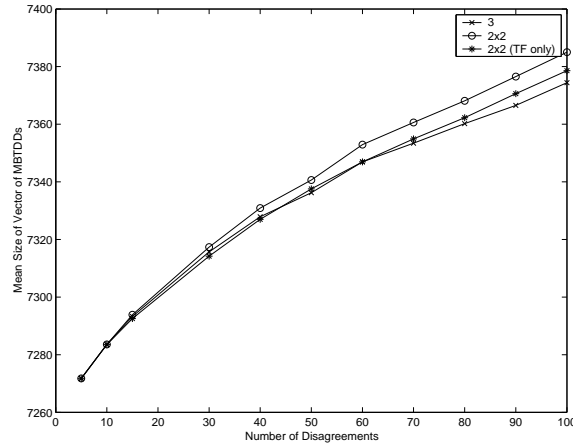


Figure 18: Graph for Experiment 2 with logics **3** and **2x2**.

In order to test our hypothesis about this dependency on structure, we perform an additional experiment:

### Experiment 2:

In this experiment, we compare sizes of MBTDDs for logics **3** and **2x2**. The method for adding disagreements with **3** remains unchanged, but in the case of **2x2**, we choose  $\ell$  to be just TF, instead of varying it randomly between TF and FT. We now expect to see no dependence on the size of the logic, apart from random noise, since our sample size may not be sufficiently large to even it out. Figure 18 gives the results of performing this experiment, validating Hypothesis 5 and showing the reduction in size for logic **2x2** with the new treatment of disagreements over the original one.

□

## 7.3 Experiment 3: Branching Factor

The goal of this experiment is to study the effect of the branching factor on the different mv-set representations. Thus, we hold the number of terminal nodes in our diagrams constant and vary the branching factor. In particular, we fix the number of terminal nodes at 2 and study the trade-offs between the representation of mv-sets as MBTDD-vectors and as BDD-vectors. Equivalently, it is possible to keep the number of terminal nodes equal to the size of the logic and study the trade-offs between MDDs and ADDs.

### 7.3.1 Experiment Design

At a first glance, a representation of any given function as an MBTDD-vector requires fewer nodes than its representation as a BDD-vector. Intuitively, this follows from the fact that each multi-valued node in an MBTDD is simulated by several boolean nodes in a BDD. However, replacing each multi-valued variable by a collection of boolean variables enables additional freedom for reordering; therefore, it is possible for a BDD-vector of a given function to be smaller than its MBTDD-vector representation.

Every multi-valued node in an MBTDD can be simulated by a BDD with at most  $2^{d+1} - 1$  nodes, where  $d$  is the maximum number of bits required to encode the branching factor of the MBTDD. Thus,

in the optimal case  $d = \lceil \log_2(|L|) \rceil$ , and if the join-irreducible encoding, as proposed in Section 5.2.3, is used, then  $d = |\mathcal{J}(L)|$ . However, this only provides an upper bound on the representation. A better estimate depends on the actual encoding used, and the variable orderings enabled by it.

Instead, let us consider an embedding of a boolean function into some logic  $L$ . For simplicity, assume that the boolean function is given by a BDD (see Figure 14(a) where  $\perp$ -edges are indicated by dashed lines). Clearly, the number of nodes in an MBTDD-vector for this function is independent of the logic used. Every BDD node is simply replaced by a corresponding multi-valued MBTDD node with additional children pointing to  $\perp$  (see Figure 14(b)). Thus, the result is an MBTDD-vector where (a) all MBTDDs are equivalent; (b) every node has at least two and at most three distinct children; and (c)  $\forall c \in L \setminus \{\top, \perp\}. \text{child}_c(u) = \perp$ . The BDD-vector is then obtained by replacing each MBTDD node that has three distinct children by  $2 \times |\mathcal{J}(L)| - 1$  BDD nodes, and replacing each node that has two distinct children by  $|\mathcal{J}(L)|$  BDD nodes. The result is shown in Figure 14(c). Comparing this figure with Figure 14(b), we note that node  $x$  is replaced by 3 nodes, whereas node  $y$  is replaced by 2 nodes. Thus, an embedding of a boolean function given by an MBTDD-vector with  $n$  nodes can be encoded by a BDD-vector with  $k \times n$  nodes, where  $k$  ranges between  $|\mathcal{J}(L)|$  and  $2 \times |\mathcal{J}(L)| - 1$ . We measure  $k$  empirically in the experiments described below.

### 7.3.2 Experimental Results

As shown in the above discussion, we expect the growth factor  $k$  to be at least  $|\mathcal{J}(L)|$ . However, we know that it depends on the number of nodes that do not have  $\perp$  children in the original BDD representation of the transition relation. Since the transition relation of a typical model is rather sparse, the reachable state space is much smaller than the total state space, so we expect  $k$  to be closer to its lower bound.

**Hypothesis 6** *Let  $L$  be a logic,  $f$  be a transition relation of some model,  $\hat{u}$  be an MBTDD-vector representation of  $\alpha_L(f)$ , and  $\hat{v}$  be a BDD-vector representation of  $\alpha_L(f)$ . Then  $k = \text{size}(\hat{u})/\text{size}(\hat{v})$  is close to  $|\mathcal{J}(L)|$ .*

#### Experiment 3(a):

The goal of this experiment is to empirically measure the growth factor  $k$ . In this experiment we embed the transition relation of each model of Table 3 into a quasi-boolean logic and then measure its size as an MBTDD-vector and as a BDD-vector.

Table 5 summarizes the results of our experiments for the **dme2-16** model. As expected, the size of the MBTDD-vector remains constant, and the size of the BDD-vector increases dramatically. Notice that the size of the BDD-vector does not depend on the size of the logic, but rather on the number of join-irreducible elements in it. For example, the size of the BDD-vector for logics **2x2x2x2** and **3x3** is exactly the same, although the logics have different number of elements. The actual growth factor  $k$  is given by the last column of the table. As suggested by Hypothesis 6, the growth factor is very close to the number of join-irreducible elements in the logic. For example, for **2x2x2x2** that has 4 join-irreducible elements, the growth factor is just 4.65. □

#### Experiment 3(b):

In this experiment, we repeat Experiment 3(a) while trying to achieve the best ordering for the BDD-vector representation, using the *sifting* reordering heuristic of CUDD [39]. The results of this experiment

Logics	Number of nodes			Growth factor
	MBTDD-vector	BDD-vector	BDD-vector (reordered)	
<b>2</b>	141840	141840	141839	1.00
<b>3</b>	141840	314495	314495	2.22
<b>4</b>	141840	487150	467704	3.43
<b>2x2</b>	141840	314495	314495	2.22
<b>2x3</b>	141840	487150	467704	3.43
<b>6C</b>	141840	659805	613704	4.65
<b>6</b>	141840	832460	801850	5.87
<b>2x2x2</b>	141840	487150	467704	3.43
<b>3x3</b>	141840	659805	613704	4.65
<b>2x2x2x2</b>	141840	659805	613704	4.65

Table 5: Results for Experiment 3.

Operation	Complexity	Complexity relative to number of nodes
Unary ( $\neg$ )	$O( G )$	$O(b \times n)$
Binary ( $\sqcup, \sqcap$ )	$O( G  \times  F )$	$O(b^2 \times n_G \times n_F)$

Table 6: Complexity of operations on decision diagrams.

are summarized in the Table 5 under the heading *BDD-vector (reordered)*. As can be seen from the table, the reordering heuristic reduced the size of the BDD-vector only marginally and did not affect the size of the BDD-vector at all in many of our experiments on smaller models. Since it is hard to predict an effect of a given reordering heuristic on an arbitrary decision diagram, we have not been able to explain this fact. We conjecture that a more aggressive heuristic should be able to significantly reduce the BDD-vector size.  $\square$

## 7.4 Complexity

In this section we analyze complexity of different operations on decision diagrams. It is well known that using a different branching factor, or representing a function by a vector of diagrams, has no effect on the complexity of the operations used during model-checking [41]. The middle column of Table 6 summarizes these complexities of the operations from the left column with respect to the size of the graph representing the diagram. Note that even though we can think of representing an mv-set using a vector of diagrams, the underlying implementation constructs a single directed acyclic graph. Moreover, since the underlying graph is connected, we can express the complexity of operations relative to the number of nodes in this graph. These complexities are given in the right column of Table 6, where  $n$  is the number of nodes and  $b$  is the branching factor of the decision diagram. Using this representation of complexity, we infer the expected running time based on the empirical evidence on the sizes of different decision diagrams.

### 7.4.1 Worst-case Complexity

As can be seen from the experiments, representing an mv-set using a vector of diagrams incurs only a small overhead in the number of nodes. Thus, since the same branching factor is used to represent a given function as a single MDD or as an MBTDD-vector, the expected worst-case time complexity of an MBTDD-vector is only slightly worse than that of a single MDD. On the other hand, representing the same function using decision diagrams with a fixed branching factor of 2 (either as a BDD-vector, or as a single ADD), incurs a considerable overhead, since each multi-valued node is replaced by several boolean nodes. Let us assume that this size increase is given by  $k$ , that is, if a function represented by an MBTDD-vector has  $n$  nodes, then the same function represented by a BDD-vector has  $k \times n$  nodes. Since BDDs enjoy a constant branching factor of 2, it follows that for a fixed logic  $L$ , the BDD-vector representation has lower worst-case time complexity than an MBTDD-vector representation if and only if  $2k < |L|$ .

### 7.4.2 Special Cases

Unfortunately, as outlined in Section 7.3, it is hard to obtain a good bound on the growth factor  $k$ , given an arbitrary function  $f$ , without constructing its decision diagram representation first. Thus, in general, this analysis is not helpful in deciding when a given representation is best suited for the task. However, there are some special cases. For example, the growth factor for a boolean function embedded into  $2 \times 2 \times 2 \times 2$  ranges between 4 and 7, whereas the branching factor of an MBTDD in this logic is 16. Moreover, our empirical evidence suggests that for functions occurring in model-checking, the growth factor is actually close to its lower bound. Therefore, we expect that, in this case, a BDD-vector representation has a significantly lower time complexity than that of an MBTDD-vector (and similarly a single MDD) representation.

The above analysis suggests that for a fixed branching factor there is no big distinction, in terms of the time complexity, between representing an mv-set using a single decision diagram, or a vector of decision diagrams. However, it hides some of the advantages of the vector representation. Let  $f$  be a function and let  $u$  and  $\hat{u}$  be its representations as an MDD and as an MBTDD-vector, respectively. Suppose further that we are trying to perform a meet (or join) of  $f$  and a crisp constant  $c \in \{\top, \perp\}$ , referred to as  $f^u \sqcap c$  or  $f^{\hat{u}} \sqcap c$ . The complexity of this operation is *constant* for an MDD operation and is linear in the *number of join-irreducible elements* of the logic, i.e.  $O(|\mathcal{J}(\mathcal{L})|)$ , for an MBTDD-vector operation. On the other hand, if  $c$  is a non-crisp value, the complexity of the MDD operation is linear in the *size of the graph representing  $u$* , whereas it remains linear in the *number of join-irreducibles* in the case of the MBTDD-vector operation. Thus, if we expect that a large number of operations to be performed involve non-crisp values from the logic, then we should represent mv-sets by a vector of decision diagrams. Notice that the exact same argument illustrates the differences between an ADD and a BDD-vector representations of mv-sets.

Another hidden benefit of using a vector of decision diagrams to represent an mv-set is seen during the implementation of the merge procedure for several viewpoints (see Section 2.2). Assume that the difference between the viewpoints is restricted to a transition relation, that is, all viewpoints are defined on a common state space  $S$ . Under this assumption, the merge problem is formulated as follows: given a collection of boolean functions  $\{f_i\}$  representing a transition relation of each viewpoint ( $f_i : S \times S \rightarrow \{\top, \perp\}$ ), construct a crisp-input function  $f : S \times S \rightarrow L$  such that  $f(y, y') = (x_1, x_2, \dots) \Rightarrow \forall f_i \cdot f_i(y, y') = x_i$ , where  $L$  is the corresponding product logic. It is easy to show that if the result of the merge operation is to be represented as a single decision diagram (either an MDD or an ADD), the time

complexity is linear in the *size* of each viewpoint. At the same time, if the result is to be represented as a vector of decision diagrams (either MBTDD-vector, or BDD-vector), the complexity is linear in the *number* of viewpoints being merged.

The above analysis shows the difference in the worst-case time complexity between different implementations of mv-sets. However, it is not clear what it says about the running time of a model-checker that uses these implementations. The problem lies in the fact that different implementations of mv-sets dictate different encodings of *model-checking properties*, and thus different sizes of decision diagrams used during the analysis. We study the observed effects of different representations using several complete model-checking case studies in the next section.

## 8 Case Studies

In this section we study the effects of the different representations of mv-sets on some realistic problems using our multi-valued model-checker  $\chi$ Chek [16]. The problems come with a series of CTL questions, thus providing us with a workload and letting us explore the relationship between the size of decision diagrams, discussed in detail in the previous section, and the running times of the verification.

This section includes three case studies of the use of multi-valued model-checking: checking abstractions of systems [16], checking compositions of features to determine which of these interact, and checking systems containing disagreements [25]. Results of the case studies are given in Table 7. For each analyzed model (“3-floor”, “5-floor”, “feature interaction”, “phone-system”), Table 7 shows verification times for several properties and the size of the transition relation. Since the latter does not change from property to property, it is shown only once for each model. All experiments were done on a Pentium III with 850 MHz processor and 256 MB RAM, running Sun JDK 1.3 under Linux 2.2.19.

### 8.1 Checking Abstractions

This case study uses the SMV elevator model of Plath & Ryan [36]. This model consists of a single elevator which accepts requests made by users pressing buttons on the floor landings and from inside the elevator. The elevator moves up and down between floors and opens and closes its doors in response to these requests according to Single Button Collective Control (SBCC) [7]. The model is implemented as several SMV modules. The `Main` module declares several instances of the module `Button` (one per floor, called `landingButi`), parameterized by the condition under which the request is considered fulfilled (`reset`), and one instance of the module `Lift`, called `lift`. The `Lift` module declares the variables `floor`, `door` and `direction` as well as further instances of `Button` to indicate requests from within the elevator (also one per floor, called `liftButi`).

In this case study we abstracted the original module `Button`, shown in Figure 19(a). In the original module, once a button is pressed, it latches and remains pressed until the elevator fulfills the request. First, we specified the latching explicitly: each variable `pressed` is decomposed into two variables, with `button` representing the actual button that users can press, and `pressed` representing the latching. The modified button is shown in Figure 19(b), and its abstracted version – in Figure 19(c). Note that this version uses a three-valued variable `button`. This gives us the three-valued model of the elevator which we can verify. More information about this case study is given in [16].

The properties of the elevator system that we verified appear in Figure 20. These are taken directly from [36], with the following exceptions: (a) we replaced `pressed` by `button` in all terms involving

<pre> <b>MODULE</b> Button(reset) <b>VAR</b>   pressed : <b>boolean</b>; <b>ASSIGN</b>   <b>init</b> (pressed) := 0;   <b>next</b> (pressed) :=   <b>case</b>     reset : 0;     pressed : 1;     1 : {0, 1};   <b>esac</b> </pre>	<pre> <b>MODULE</b> Button(reset) <b>VAR</b>   pressed : <b>boolean</b>;   button : <b>boolean</b>; <b>ASSIGN</b>   <b>init</b> (button) := 0;   <b>next</b> (button) := {0, 1};   <b>init</b> (pressed) := 0;   <b>next</b> (pressed) :=   <b>case</b>     button : 1;     reset : 0;     1 : pressed;   <b>esac</b> </pre>	<pre> <b>MODULE</b> Button(reset) <b>VAR</b>   pressed : <b>boolean</b>;   button : {T, M, F}; <b>ASSIGN</b>   button :=   <b>case</b>     reset   pressed : M     1 : {0, 1}   <b>esac</b>   <b>init</b> (pressed) := 0;   <b>next</b> (pressed) :=   <b>case</b>     button = T : 1;     reset : 0;     1 : pressed;   <b>esac</b> </pre>
--	--	---

Figure 19: Three models of the elevator button in SMV: (a) the original module Button of Plath & Ryan; (b) a modified module Button; (c) an abstracted module Button.

1. “If the door closes, it will eventually open”:  
 $AG(\text{lift.door} = \text{closed} \rightarrow AF \text{ lift.door} = \text{open})$
2. “If a button inside the lift is pressed, the lift will eventually arrive at the corresponding floor”:  
 $AG(\text{lift.liftBut2.button} \rightarrow AF(\text{lift.floor} = 2 \wedge \text{lift.door} = \text{open}))$
3. “Pressing a landing button guarantees that the lift will arrive at that landing and open its doors”:  
 $AG(\text{landingBut2.button} \rightarrow AF(\text{lift.floor} = 2 \wedge \text{lift.door} = \text{open}))$
4. “The lift may stop at floor 2 for landing calls when traveling downwards”:  
 $\neg AG((\neg \text{lift.floor} = 2 \wedge \neg \text{lift.liftBut2.button} \wedge \text{lift.direction} = \text{down}) \rightarrow \text{lift.door} = \text{closed})$
5. “Whenever a button indicator is on, a button is being pressed”:  
 $AG(\text{lift.liftBut2.pressed} \rightarrow \text{lift.liftBut2.button})$

Figure 20: Properties of the elevator system.

landing or elevator buttons because of our change in the module Button; (b) we selected only a subset of properties from [36]. Similar properties can be formulated for all other floors and all other landing and elevator buttons. We further parameterized the model by the number of floors and ran our experiments with 3 and 5 floors. The results are summarized in Table 7 under “3-floor” and “5-floor” models. For example, verification of property 1 of the 5-floor elevator can range from 19.009 s when mv-sets are implemented using MDDs, to 66.713 s, when they are implemented via a BDD-vector. Note that the MDD and the MBTDD-vector implementations yield the same transition relation sizes in the 3- and 5-floor elevator models. MBTDDs perform better in the 3-floor model, whereas MDDs perform better in

Model	CTL Property Number	Result	$\chi$ Chek			
			MDD	MBTDD	ADD	BDD
3-floor	1.	F	0.505 s	0.423 s	1.222 s	1.007 s
	2.	T	0.194 s	0.125 s	1.306 s	0.23 s
	3.	T	0.197 s	0.122 s	1.171 s	0.233 s
	4.	T	0.497 s	0.591 s	1.406 s	0.1.196 s
	5.	M	0.202 s	0.125 s	0.596 s	0.23 s
Size of trans. relation			954	954	2467	1991
5-floor	1.	F	19.009 s	21.251 s	30.89 s	66.713 s
	2.	T	2.254 s	2.407 s	6.978 s	5.156 s
	3.	T	2.331 s	2.406 s	5.21 s	4.929 s
	4.	T	14.853 s	19.394 s	79.753 s	63.978 s
	5.	M	1.017 s	0.862 s	4.263 s	1.725 s
Size of trans. relation			4870	4870	12615	10114
Feature-interaction (3 floors)	1.	TT	1.081 s	1.001 s	2.854 s	0.716 s
	2.	FF	1.318 s	1.484 s	3.415 s	1.119 s
	3.	TF	2.449 s	2.445 s	6.169 s	1.95 s
	4.	FF	0.95s s	0.957 s	2.463 s	0.695 s
	5.	TT	2.257 s	2.413 s	5.776 s	2.362 s
Size of trans. relation			6596	7099	14868	9401
Phone system (9-valued)	1.	MM	0.048 s	0.066 s	2.3 s	0.048 s
	2.	MF	0.044 s	0.074 s	2.135 s	0.052 s
	3.	MF	0.046 s	0.068 s	2.613 s	0.037 s
Size of trans. relation			108	142	453	599
Phone system (4-valued)	1.	TT	0.031 s	0.023 s	0.024s	0.029 s
	2.	FF	0.031 s	0.027 s	0.042 s	0.036 s
	3.	FF	0.031 s	0.027 s	0.041 s	0.031 s
Size of trans. relation			84	99	178	207

Table 7: Case studies: sizes of transition relation and running times of the four implementations of  $\chi$ Chek.

the 5-floor case.

## 8.2 Finding Feature Interactions

This case study is based on the work of Plath and Ryan [36, 35] in the feature interaction domain.

Let  $P$  be a system, and  $\delta_1$  and  $\delta_2$  be two features. Then  $P + \delta_1$  is a system composed with the first

1. “If the door closes, it may stay closed forever”:  

$$\neg AG(\text{lift.door} = \text{closed} \rightarrow AF \text{lift.door} = \text{open})$$
2. “If a button inside the lift is pressed, the lift will eventually arrive at the corresponding floor”:  

$$AG(\text{lift.liftBut2.button} \rightarrow AF(\text{lift.floor} = 2 \wedge \text{lift.door} = \text{open}))$$
3. “Lift calls have precedence when the lift is 2/3 full (indicated by `tt_full`)”:  

$$AG((\text{lift.tt\_full} \wedge \text{lift.liftBut2.button} \wedge \neg \text{lift.liftBut3.button})$$

$$\rightarrow A[(\text{lift.floor} \neq 2 \vee \text{lift.door} = \text{closed}) U$$

$$(\text{lift.floor} = 2 \wedge \text{lift.door} = \text{open})$$

$$\vee \neg \text{lift.tt\_full} \vee \text{lift.liftBut3.button}])$$
4. “Pressing a landing button guarantees that the lift will arrive at that landing and open its doors”:  

$$AG(\text{landingBut2.button} \rightarrow AF(\text{lift.floor} = 2 \wedge \text{lift.door} = \text{open}))$$
5. “The lift may stop at floor 2 for landing calls when traveling downwards”:  

$$\neg AG((\neg \text{lift.floor} = 2 \wedge \neg \text{lift.liftBut2.button} \wedge \text{lift.direction} = \text{down})$$

$$\rightarrow \text{lift.door} = \text{closed})$$

Figure 21: Properties of the combined elevator system.

feature, and  $P + \delta_1 + \delta_2$  is the system composed with two features, where  $\delta_1$  is applied first. We assume that the feature descriptions as well as the actual procedure to combine a feature and the system are the ones outlined in the work of Plath and Ryan. We are interested in different interactions between features. For example, for a property of interest  $\varphi$ , we want to know if the features commute relative to  $\varphi$ . That is, we want to know whether  $([P + \delta_1 + \delta_2] \models \varphi) \wedge ([P + \delta_2 + \delta_1] \models \varphi)$ . A naive approach is to break this problem into two independent model-checking problems. However, with  $\chi\text{Chek}$ , a different solution is possible. First, we notice that the state space of  $[P + \delta_1 + \delta_2]$  and  $[P + \delta_2 + \delta_1]$ , referred to as  $S$ , is exactly the same. This allows us to combine the two systems into one 4-valued (over  $\mathbf{2x2}$  logic) model  $M = (\hat{S}, \hat{R})$ , where  $\hat{S}$  is the result of embedding  $S$  into logic  $\mathbf{2x2}$  ( $\hat{S} = \alpha_{\mathbf{2x2}}(S)$ ), and  $\hat{R}$  is constructed by merging transition relations of  $[P + \delta_1 + \delta_2]$  and  $[P + \delta_2 + \delta_1]$  using the process described in Section 2.2 and illustrated in Figure 2.

In the case where the two systems share the same state space, it can be easily shown that our model-checker  $\chi\text{Chek}$ , run on an arbitrary property  $\varphi$ , returns TT if and only if the result of model-checking  $\varphi$  on each individual model is T;  $\chi\text{Chek}$  returns TF if and only if the result of model-checking  $\varphi$  on the first model is T and on the second model is F, etc. Thus, instead of solving two boolean model-checking problems, we can solve one multi-valued problem.

In this case study, we composed the base elevator system (SBCC) with two features:

**Lift-2/3-full.** When the elevator detects that it is more than two-thirds full, it does not stop in response to external calls, since it is unlikely to be able to accept more passengers. Instead, it gives priority to passengers already inside the elevator, as serving them will help reduce its load.

**Executive Floor.** The elevator gives priority to calls from the executive floor.

and ran the model-checker over the properties that appear in Figure 21. Note that most of these properties are the same as in Figure 20. We expect these properties to hold in the composed system. However, property 3 is likely to reveal feature interaction when the third floor is designated as the executive floor: when the elevator is 2/3 full and there is a call from the executive floor and from within the elevator, which call should get priority?

Results of the verification of the feature interaction of the three-floor elevator are shown in Table 7 under “feature interaction”. Note that each property except 3 yields TT or FF, indicating that the two features commute w.r.t. this property. Value TF for property 3 indicates that it holds if Lift-2/3-full is the last feature added, and does not if Executive Floor is the last feature added.

In this example, the implementation based on a BDD-vector performs the best. The implementations that use MDDs and MBTDD-vectors exhibit similar performance.

### 8.3 Reasoning about Disagreement

In this case study we use multi-valued logics to model disagreement between descriptions of two aspects of a system. Model-checking is then used to determine whether these disagreements matter [25].

Figure 22(a)-(b) shows the individual (partial) descriptions of the caller’s and the callee’s perspectives on the phone system. The combined model is given in Figure 22(c). The combined model can be described either using a 9-valued logic  $3 \times 3$  or the four-valued logic  $2 \times 2$ , with values other than TT and FF representing disagreements between the two viewpoints, either on the values of variables in the states, or the values of the transitions. We can determine whether these disagreements matter by checking the combined system against the correctness criteria of the phone system which appear in Figure 23.

We give verification results of the phone system in the 9-valued and the 4-valued logic in Table 7 under “Phone system”. Note that the transition relation has a fairly small size in all four representations when the four-valued logic is used. Further, the implementation that uses ADDs performs better than the others. However, in the 9-valued logic case, the implementations based on ADD and a BDD-vector exhibit a roughly 250% size increase, as opposed to a 50% increase for the MDDs and the MBTDD-vectors. Further, in this case the ADD representation performs the worst, whereas the BDD representation gives the best performance.

## 9 Conclusion

In this section we summarize the contribution of this paper and outline venues for future research.

### 9.1 Summary

Multi-valued logics can be effectively used to reason about incomplete and/or inconsistent systems. In this paper we cast the problem of model-checking a class of multi-valued logics, called quasi-boolean logics, in terms of operations on mv-sets. We also study two major choices for representing a membership function in an mv-set and its implementation using decision diagrams: either as one multi-valued characteristic function, implemented using MDDs or ADDs, or as a set of boolean characteristic functions, implemented

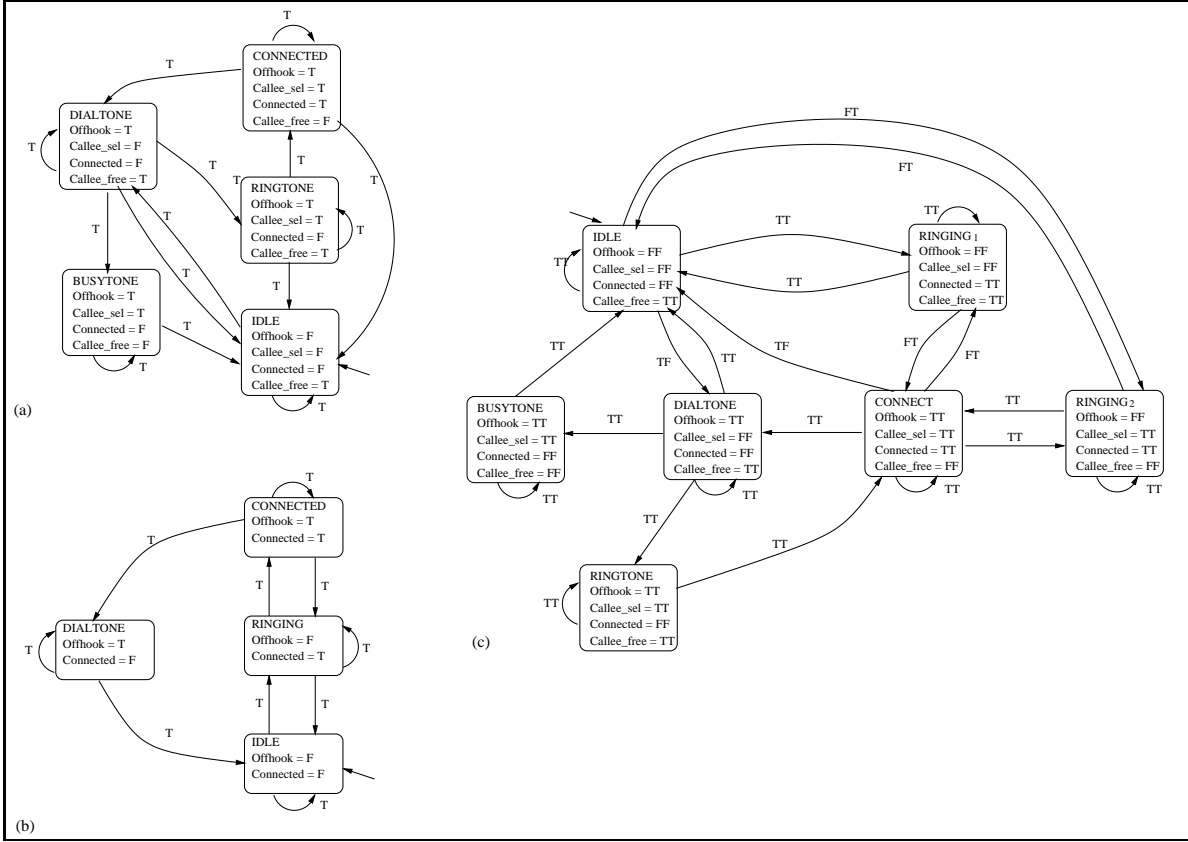


Figure 22: Partial descriptions of a phone system: (a) Caller's perspective; (b) Callee's perspective; (c) combined system.

1. "If you are connected, you can hang up":  

$$AG(\text{connected} \rightarrow EX\neg\text{offhook})$$
2. "A connection doesn't start until you pick up the phone":  

$$AG(\neg\text{connected} \rightarrow \neg E[\neg\text{offhook} U (\text{connected} \wedge \neg\text{offhook})])$$
3. "If you are offhook and connected, hanging up always disconnects you":  

$$AG((\text{offhook} \wedge \text{connected}) \rightarrow AX(\neg\text{offhook} \rightarrow \neg\text{connected}))$$

Figure 23: Properties of the phone system.

using an MBTDD-vector or a BDD-vector. The naive implementation of the latter choice can be improved if we represent mv-sets as operations on join-irreducible elements of the lattices underlying the logic.

The main result of the paper is a series of experiments with the four decision diagram representations. As is usual with decision diagrams, no one representation is a clear winner in all situations. However, we did characterize, both analytically and using empirical data, some situations where one representation performs better than the others. We further confirmed these results on a number of real case studies.

We currently have a prototype implementation of  $\chi$ Chек – a symbolic multi-valued model-checker that uses any of the four decision diagram varieties as the representation of mv-sets.  $\chi$ Chек is available upon request, and its redesign is currently underway.

## 9.2 Related Work

Work on multi-valued model-checking was surveyed in Section 1. Here we look at the work related to decision diagrams: representation of multi-valued functions and their use in non-classical model-checking.

Some of the questions relating to decision-diagram representations of finite functions, discussed in this paper, have also arisen in different domains of application. For instance, incompletely-specified boolean functions [41] have been represented as single diagrams with three outputs (true ( $T$ ), false ( $F$ ), don't care ( $DC$ )), i.e., via 3-valued MDD diagrams. Alternatively, each incompletely-specified function  $f$  can be represented by three diagrams,  $f_T, f_F, f_{DC}$ , with each diagram representing a characteristic function of the respective output. For example,  $f_{DC}(x) = 1$  if and only if  $f(x)$  is undefined. Since the value of the function  $f$  is exactly one of  $T, F$  or  $DC$ , storing two characteristic functions is sufficient. This is a specific case of the general problem which we have examined in our domain of interest: that is, given a function range of size greater than 2, what are the tradeoffs between using a single diagram with multiple terminal nodes, and a vector of diagrams with only 2 terminal nodes. Sasao et al [38, 3] have explored these alternatives experimentally in the domain of arithmetic circuits. Orthogonal to this issue is that of the arity of *input* variables. Sasao's experiments of [38] also deal with this question, but it has been explored considerably less. The above experiments typically concluded that there is no clear winner, and the best solution depends on the problem at hand. In this paper we characterized, in the domain of model-checking, some of the circumstances under which different implementations are more preferable.

Decision diagrams have been used as underlying data structures in several other fixpoint-base approaches over labeled graphs. For example, Kwiatkowska and her group [30] use MTBDDs (multi-terminal boolean diagrams) for model-checking probabilistic systems. These diagrams are roughly equivalent to ADDs. MTBDDs are also used by Baier and Clarke [5], for solving fixed-point problems such as shortest-path computations. These problems are defined via a numeric-valued generalization of  $\mu$ -calculus, called *algebraic  $\mu$ -calculus*. In both cases, the algorithms are quite similar to ours. However, these problems do not enjoy Knaster-Tarski convergence conditions; therefore, exact solutions often cannot be obtained, and approximations are used instead.

## 9.3 Future Work

In the future, we plan to concentrate on implementing and evaluating a number of optimizations of the approach presented in this paper.

Here we have assumed that the transition relation of a system is given by a single monolithic decision diagram. However, in the classical symbolic model-checking it has been shown that in many cases the transition relation is too large to be represented by a single diagram. An alternative solution is to partition the transition relation into several decision diagrams [14]. Such partitioning may reduce the overall size of the representation of the transition relation, and thus result in a significant performance increase. We are interested in extending the idea of partitioning to multi-valued domains.

In our current implementation we utilize only a fraction of all possible optimizations that are currently available in the state-of-the-art BDD libraries such as CUDD [39]. Many of these optimizations,

such as reordering heuristics, complement (shift) edges, etc., have a natural extension to multi-valued domains. We are interested in exploring their effect and the resulting tradeoffs between different mv-set representations.

Throughout this paper we maintain a very strong assumption about the branching factor and the number of terminal nodes in the diagrams. These parameters are always fixed either at 2 or the size of the logic. However, it is interesting to explore the middle ground. For example, instead of representing functions in a given logic  $L$  by  $\mathcal{J}(L)$  diagrams with 2 terminal nodes each, we can reduce the number of diagrams by allowing each diagram to have more terminal nodes. Similarly, it is possible to use any branching factor between 2 and the size of the logic. However, a more interesting approach is to allow for the branching factor to vary throughout the diagram. The problem with using a fixed branching factor becomes evident when we consider functions that are input-crisp in almost all inputs. For example, if the branching factor is fixed at the size of the logic, each crisp input variable must be represented by a node with a large branching factor, which results in performance penalty since the worst-time complexity is linear in the branching factor. Alternatively, with varying branching factor diagrams each crisp input can be represented by a node with a branching factor of 2, resulting in no significant loss in performance. Essentially, this is equivalent to adding types to decision diagrams, where a node in the diagram has a branching factor corresponding to the *type* of the variable it represents (i.e. boolean variables are represented by nodes with branching factor of 2, ternary with branching factor of 3, etc.). Diagrams with varying branching factor are also good candidates for representing enumerated types in both classical and multi-valued model-checking. Since, the branching factor is not fixed, there is an additional freedom in deciding how to encode a variable of an enumerated type in the diagram. For example, a variable on an enumerated type with 3 values can be encoded by a single node with a branching factor of 3, or potentially a ternary decomposition can be used for all enumerated types in the model independently of their size. We would like to further study the effects of the number of terminal nodes and the various branching factors on the performance of model-checking, and hope to develop different heuristics for finding the optimal parameters to use.

The success of multi-valued, as well as classical, model-checking depends on the ability to construct abstractions of a model without constructing the full model first. Currently, there are two possible approaches: a) the abstraction is performed at the level of the modeling language; b) the abstraction is performed at the level of decision diagrams, before the final transition relation is constructed. The second approach is possible through the use of *abstract binary decision diagrams (aBDD)* [22]. We are interested in exploring the applicability of abstract decision diagrams to multi-valued domains to allow the combination of mergers of different viewpoints and logic abstraction in a single step.

Finally, we are interested in carrying out more case studies of the use of  $\chi$ Chek. We hope to form a better understanding of its domains of applicability and factors determining its performance.

## References

- [1] S. Akers. “Binary Decision Diagrams”. *IEEE Trans. Computers*, C-27:509–516, 1978.
- [2] A.R. Anderson and N.D. Belnap. *Entailment. Vol. 1*. Princeton University Press, 1975.
- [3] Hafiz Md. Hasan Babu and Tsutomu Sasao. “Representations of Multiple-Output Functions Using Binary Decision Diagrams for Characteristic Functions”. *IEICE Transactions on Fundamentals*, E82-A(11):2398–2406, 1999.

- [4] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. “Algebraic Decision Diagrams and Their Applications”. In *IEEE /ACM International Conference on CAD*, pages 188–191, Santa Clara, California, 1993. IEEE Computer Society Press.
- [5] C. Baier and E. M. Clarke. “The Algebraic Mu-Calculus and MTBDDs”. In *Proceedings of the 5th Workshop on Logic, Language, Information and Computation, (WoLLIC’98)*, 1998.
- [6] N.D. Belnap. “A Useful Four-Valued Logic”. In Dunn and Epstein, editors, *Modern Uses of Multiple-Valued Logic*, pages 30–56. Reidel, 1977.
- [7] G.C. Berney and S.M. dos Santos. *Elevator Analysis, Design and Control*. IEE Control Engineering Series 2. Peter Peregrinus Ltd., 1985.
- [8] L. Bolc and P. Borowik. *Many-Valued Logics*. Springer-Verlag, 1992.
- [9] G. Bruns and P. Godefroid. “Generalized Model Checking: Reasoning about Partial State Spaces”. In *Proceedings of CONCUR’00*, volume 877 of *LNCS*, pages 168–182, August 2000.
- [10] G. Bruns and P. Godefroid. “Temporal Logic Query Checking”. In *Proceedings of LICS 2001: the 16th IEEE Symposium on Logic in Computer Science*, pages 409–417, Boston, 2001.
- [11] R. E. Bryant. “Graph-based algorithms for boolean function manipulation.”. *Transactions on Computers*, 8(C-35):677–691, 1986.
- [12] R. E. Bryant. “Symbolic Boolean Manipulation with Ordered Binary-Decision diagrams”. *Computing Surveys*, 24(3):293–318, September 1992.
- [13] 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.
- [14] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. “Symbolic Model Checking for Sequential Circuit Verification”. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 13(4):401–424, June 1992.
- [15] M. Chechik, B. Devereux, and S. Easterbrook. “Implementing a Multi-Valued Symbolic Model-Checker”. In *Proceedings of TACAS’01*, volume 2031 of *LNCS*, pages 404–419. Springer, April 2001.
- [16] M. Chechik, B. Devereux, S. Easterbrook, and A. Gurfinkel. “Multi-Valued Symbolic Model-Checking”. Submitted for publication, October 2001.
- [17] M. Chechik, B. Devereux, S. Easterbrook, A. Lai, and V. Petrovykh. “Efficient Multiple-Valued Model-Checking Using Lattice Representations”. In *Proceedings of CONCUR’01*, volume 2154 of *LNCS*, pages 451–465. Springer, August 2001.
- [18] M. Chechik, B. Devereux, and A. Gurfinkel. “Model-Checking Infinite State-Space Systems with Fine-Grained Abstractions Using SPIN”. In *Proceedings of the 8th SPIN Workshop on Model Checking Software*, volume 2057 of *LNCS*, pages 16–36. Springer, May 2001.
- [19] M. Chechik, S. Easterbrook, and V. Petrovykh. “Model-Checking Over Multi-Valued Logics”. In *Proceedings of FME’01*, volume 2021 of *LNCS*, pages 72–98. Springer, March 2001.
- [20] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings Eleventh Conference on Computer-Aided Verification (CAV’99)*, number 1633 in *Lecture Notes in Computer Science*, pages 495–499, Trento, Italy, July 1999. Springer.
- [21] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
- [22] E. Clarke, Y. Lu, and H. Veith. “A Survey of Abstract BDDs”. In *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics (SCI’00)*, July 2000.

- [23] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.
- [24] J.M. Dunn. “A Comparative Study of Various Model-Theoretic Treatments of Negation: A History of Formal Negation”. In Dov Gabbay and Heinrich Wansing, editors, *What is Negation*. Kluwer Academic Publishers, 1999.
- [25] S. Easterbrook and M. Chechik. “A Framework for Multi-Valued Reasoning over Inconsistent Viewpoints”. In *Proceedings of the International Conference on Software Engineering (ICSE’01)*, pages 411–420, May 2001.
- [26] M. Fujita, P. C. McGeer, and J. C.-Y. Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design: An International Journal*, 10(2/3):149–169, April 1997.
- [27] B. R. Gaines. “Logical Foundations for Database Systems”. *International Journal of Man-Machine Studies*, 11(4):481–500, 1979.
- [28] M. Ginsberg. “Multi-valued logic”. In M. Ginsberg, editor, *Readings in Nonmonotonic Reasoning*, pages 251–255. Morgan-Kaufmann Pub., 1987.
- [29] S. Hazelhurst. *Compositional Model Checking of Partially Ordered State Spaces*. PhD thesis, Department of Computer Science, University of British Columbia, 1996.
- [30] M.Z. Kwiatkowska, G. Norman, D.A. Parker, and R. Segala. “Symbolic Model Checking of Probabilistic Processes Using MTBDDs and the Kronecker Representation”. In *Proceedings of TACAS 2000*, number 1587 in Lecture Notes in Computer Science. Springer-Verlag, 2000.
- [31] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.
- [32] 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.
- [33] S. Minato. “Arithmetic Boolean Expression Manipulator using BDDs”. *Formal Methods in System Design*, 10:221–242, 1997.
- [34] J.J. Pazos-Arias and J.G. Duque. “SCTL-MUS: A Formal Methodology for Software Development of Distributed Systems. A Case Study”. *Formal Aspects of Computing*, 13:50–91, 2001.
- [35] M.C. Plath and M.D. Ryan. A semantics of a feature construct for SMV: A case study in non-monotonic composition. Technical report, School of Computer Science, University of Birmingham, 1999. Available as <ftp://ftp.cs.bham.ac.uk/pub/tech-reports/1999/CSR-99-10.ps.gz>.
- [36] M.C. Plath and M.D. Ryan. “SFI: A Feature Integration Tool”. In R. Berghammer and Y. Lakhnech, editors, *Tool Support for System Specification, Development and Verification*, Advances in Computer Science, pages 201–216. Springer, 1999.
- [37] H. Rasiowa. *An Algebraic Approach to Non-Classical Logics. Studies in Logic and the Foundations of Mathematics*. Amsterdam: North-Holland, 1978.
- [38] T. Sasao and J.T. Butler. “A Method to Represent Multiple-Output Switching Functions Using Multi-Valued Decision Diagrams”. In *Proceedings of the IEEE International Symposium on Multiple-Valued Logic*, pages 248–254, Santiago de Compostela, Spain, 1996.
- [39] F. Somenzi. “Binary Decision Diagrams”. In Manfred Broy and Ralf Steinbrüggen, editors, *Calculational System Design*, volume 173 of *NATO Science Series F: Computer and Systems Sciences*, pages 303–366. IOS Press, 1999.
- [40] A. Srinivasan, T. Kam, S. Malik, and R.E. Brayton. “Algorithms for Discrete Function Manipulation”. In *IEEE International Conference on Computer-Aided Design*, pages 92–95, 1990.

- [41] I. Wegener. “*Branching Programs and Binary Decision Diagrams: Theory and Applications*”. Monographs on Discrete Mathematics and Applications. SIAM, 2000.
- [42] B. Yang, R.E. Bryant, D. R. O’Hallaron, A. Biere, O. Coudert, G. Janssen, R. K. Ranjan, and F. Somenzi. “A Performance Study of BDD-Based Model Checking”. In *Formal Methods in Computer-Aided Design*, pages 255–289, 1998.
- [43] L.A. Zadeh. “Fuzzy Sets”. In R. R. Yager, S. Ovchinnikov, R. M. Tong, and H. T. Nguyen, editors, *Fuzzy Sets and Applications: Selected Papers by L.A. Zadeh*, pages 29–44, New York, 1987. John Wiley & Sons, Inc.