Automated Paraconsistent Reasoning via Model Checking

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

Abstract

Inconsistency is a pervasive problem in software engineering, where different aspects of a system are described in separate models. Resolving all the inconsistencies in a large set of models is often infeasible, in which case automated reasoning tools based on classical logic have limited application. In this paper we describe an automated tool for paraconsistent reasoning, using multi-valued logics. The reasoning engine is an adaptation of classical model checking, and works for any multi-valued logic whose truth values form a quasi-boolean lattice. We describe the design of the model checker, and show how it can be used to reason about models created by merging information from multiple conflicting sources.

1 Introduction

Inconsistency is a pervasive problem in software engineering. Software engineers make use of a large number of different models that are constructed and updated by different developers at different times during development. These models incorporate information from multiple sources, reflecting different points of view. Such models have significant overlaps because they typically describe orthogonal aspects of a system, rather than hierarchical decompositions.

Recent work on inconsistency in software engineering has focused on tools that detect and track inconsistencies in large sets of interrelated models [Easterbrook and Nuseibeh, 1996; Grundy *et al.*, 1998; Robinson and Pawlowski, 1999; van Lamsweerde *et al.*, 1998]. The rationale is that toleration of inconsistency is desirable, to facilitate distributed collaborative working, to prevent premature commitment to design decisions, and to ensure all stakeholder views are taken into account [Schwanke and Kaiser, 1988]. However, there has been relatively little work on automated reasoning with inconsistent models. This means that many of the kinds of analysis routinely performed on software models cannot be applied in the presence of inconsistency. This problem has stimulated an interest in paraconsistent reasoning in software engineering [Hunter and Nuseibeh, 1998; Menzies *et al.*, 1999]

In this paper, we describe an extension of model checking to a family of paraconsistent logics. Model checking has become established as an important analysis tool in software engineering, mainly because of the potential for fully automated reasoning. Unlike theorem proving, a model checker can be used for "push-button verification": given a model M (of a program, a design, or a specification) expressed as a finite state machine, and a property P expressed in a temporal logic, a model checker will automatically determine whether $M \models P$, by exhaustively searching the state space. The current generation of model checkers can efficiently search very large state spaces (e.g. 10^{20} states [Burch *et al.*, 1990]) using compact encodings and pruning techniques. However, the current generation of model checkers are based on classical logic, and hence cannot handle inconsistency.

The logics we use in this work are multi-valued logics whose values form quasi-boolean lattices under the truth ordering. These logics provide most of the properties of classical logic, with the exception of the law of excluded middle ($a \vee \neg a = \top$) and the law of non-contradiction ($a \wedge \neg a = \bot$). We have found these logics to provide a natural formalism for expressing uncertainty and disagreement between different views. By developing a model checker to reason over these logics, we can check which properties of a model are affected by the uncertainty or disagreement, and can even determine how much agreement there is over each property.

The paper is organised as follows. The next section gives a formal introduction to quasi-boolean logics, and briefly contrasts them with other multi-valued logics described in the literature. Section 3 describes the semantics of multi-valued model checking, and outlines the design of the model checker. Section 4 demonstrates how we apply the model checker for reasoning about information merged from multiple sources. Section 5 presents our conclusions.

2 Multi-Valued Reasoning

2.1 Why Multi-Valued Logics?

Multi-valued logics are useful for merging information from inconsistent viewpoints because they allow us to explicitly represent different levels of agreement. For example, if we keep the usual boolean values TRUE and FALSE to mean '*a unanimous yes*' and '*a unanimous no*', we can add any number of intermediate values to represent different kinds of disagreement. Examples include '*a majority said yes*', '*4 yeses and 1 no*', '*nobody knows*', '*the designated expert said no, everyone else said yes*'. By adding these explicitly as values in the logic, and defining a truth order over them, we can reason about the level of agreement for any arbitrary proposition.

A number of specific multi-valued logics have been proposed and studied. For example, Łukasiewicz [Łukasiewicz, 1970] first introduced a three-valued logic to allow for propositions whose truth values are 'unknown', while Belnap [Belnap, 1977] proposed a four-valued logic that also introduces the value 'both' (i.e. "true *and* false"), to handle inconsistent assertions in database systems. Each of these logics can be generalized to allow for different levels of uncertainty or disagreement. For example, Belnap's logic was generalized by Ginsberg to a family of logics represented as bilattices [Ginsberg, 1998].

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

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

Different multi-valued logics are useful for different purposes. For example, we may wish to have several levels of uncertainty. We may wish to use different multi-valued logics to support different ways of merging information from multiple sources: keeping track of the origin of each piece of information, doing a majority vote, giving priority to one information source, etc. Thus, rather than restricting ourselves to any particular multi-valued logic, we are interested in extending the classical symbolic model-checking procedure to enable reasoning about arbitrary multi-valued logics, as long as conjunction, disjunction and negation of the logical values are specified.

We originally considered the family of logics characterized by Ginsberg's bilattices [Ginsberg, 1998]. However, these do not lend themselves well to modeling the process of merging information from multiple sources. The original intent of Belnap's logic was to model information being added incrementally to a database. Hence these logics represent how much information (certainty) we have about each proposition, as well as it's truth value. However, these logics do not readily model disagreements between multiple authorities for information. We adopted a simpler approach based on the use of logics whose values form quasi-boolean lattices, and use products of these lattices to represent merged information sources. Our logics have the further advantage that classical two-valued logic is a member of the family. We give a formal introduction to these logics in section 2.2

For reasoning about dynamic properties of systems, existing modal logics can be extended to the multi-valued case. Fitting [Fitting, 1991] suggests two different approaches for doing this: the first extends the interpretation of atomic formulae in each world to be multi-valued; the second also allows multi-valued accessibility relations between worlds. The latter approach is more general, and can readily be applied to the temporal logics used in automated verification [Chechik *et al.*, 2001b].

Some automated tools for reasoning with multi-valued logics exist. In particular, the work of Hähnle and others [Hähnle, 1994; Sofronie-Stokkermans, 2000] has led to the development of several theorem-provers for first-order multi-valued logics. However, as yet the question of model checking for multi-valued modal logics has not been addressed.

2.2 Quasi-Boolean Multi-Valued Logics

We wish to use a range of different multi-valued logics, and so we need a way to specify each particular logic, and to ensure that it has the properties needed by our model checking algorithm. We can specify a logic by giving its inference rules or by defining conjunction, disjunction and negation on the truth values of the logic. Since our goal is model-checking as opposed to theorem proving, we chose the latter approach. Further, the logic should be as close to classical as possible; in particular, the defined operations should be idempotent, commutative, etc.

We achieve this by defining each logic using a lattice of truth values, and define the logical operators in terms of lattice operations. We restrict ourselves to those logics whose values form a quasi-boolean lattice, as this ensures that negation in the logic has the expected properties. Finally, we use products of lattices to handle information merged from multiple sources. In the remainder of this section we give a brief overview of the relevant parts of lattice theory.

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

The following are the properties of lattices:

$$\begin{array}{rcll}
a \sqcup a & = & a & \text{(idempotence)} \\
a \sqcap a & = & a & \\
a \sqcup b & = & b \sqcup a & \text{(commutativity)} \\
a \sqcap b & = & b \sqcap a & \\
a \sqcup (b \sqcup c) & = & (a \sqcup b) \sqcup c & \text{(associativity)} \\
a \sqcap (b \sqcap c) & = & (a \sqcap b) \sqcap c & \\
a \sqcup (a \sqcap b) & = & a & \text{(absorption)} \\
a \sqcap (a \sqcup b) & = & a & \\
a \sqsubseteq a' \ \wedge \ b \sqsubseteq b' & \Rightarrow & a \sqcap b \sqsubseteq a' \sqcap b' & \text{(monotonicity)} \\
a \sqsubseteq a' \ \wedge \ b \sqsubseteq b' & \Rightarrow & a \sqcup b \sqsubseteq a' \sqcup b' &
\end{array}$$

$a \sqcap b$ and $a \sqcup b$ are referred to as *meet* and *join*, representing for us conjunction and disjunction operations, respectively. Figure 1 gives examples of a few logic lattices. Our partial order operation $a \sqsubseteq b$ means that "b is more true than a".

Definition 1. *A lattice is* complete *if it includes a meet and a join for for every subset of its elements. Every complete lattice has a top ($\top$) and a bottom ($\bot$).*

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

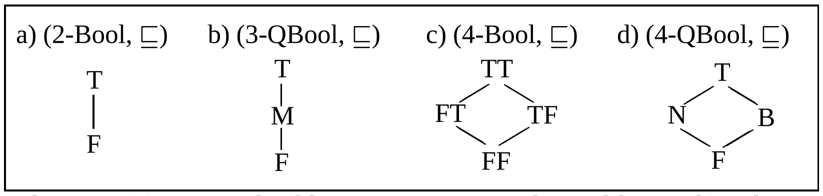


Figure 1: Examples of logic lattices: (a) a two-valued lattice representing classical logic; (b) a three-valued lattice reflecting uncertainty; (c) a four-valued boolean lattice, a product of two (2-Bool, ⊑) lattices; (d) a four-valued quasi-boolean lattice.

We only use lattices that have a finite number of elements. Every finite lattice is complete. We adopt the convention of using T to indicate $\top$ of the lattice, and F to indicate its $\bot$, although in principle $\top$ and $\bot$ might be labelled differently.

Definition 2 *A finite lattice ($\mathcal{L}$, $\sqsubseteq$) is called a* Boolean lattice *if every element $a \in \mathcal{L}$ has a unique* complement $\neg a \in \mathcal{L}$ *satisfying the following conditions:*

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

In fact, $\neg$ involution, de Morgan and antimonotonic laws follow from $\neg$ contradiction and $\neg$ exhaustiveness.

Definition 2. *[Bolc and Borowik, 1992] A finite lattice ($\mathcal{L}$, $\sqsubseteq$) is* quasi-boolean[1] *if there exists a unary operator $\neg$ defined for it, with the following properties (a, b are elements of $\mathcal{L}$):*

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

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

Therefore, all boolean lattices are also quasi-boolean, whereas the converse is not true. Logics represented by quasi-boolean lattices will be referred to as *quasi-boolean logics*.

The identification of a suitable negation operator is greatly simplified by the observation that quasi-boolean lattices are symmetric about their horizontal axes:

Definition 3 *A lattice ($\mathcal{L}$, $\sqsubseteq$) is* horizontally-symmetric *if there exists a bijective function H such that for every pair $a, b \in \mathcal{L}$,*

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

Theorem 1 *[Chechik* et al.*, 2001b] Let ($\mathcal{L}, \sqsubseteq$) be a horizontally-symmetric lattice. Then the following hold for any two elements $a, b \in \mathcal{L}$:*

$$\begin{array}{rcl} H(a \sqcap b) & = & H(a) \sqcup H(b) \\ H(a \sqcup b) & = & H(a) \sqcap H(b) \end{array}$$

[1]also called *De Morgan* [Dunn, 1999]

Definition 4 *A* product *of two lattices $(\mathcal{L}_1, \sqsubseteq)$, $(\mathcal{L}_2, \sqsubseteq)$ is a lattice ($\mathcal{L}_1 \times \mathcal{L}_2$), with the ordering $\sqsubseteq$ holding between two pairs iff it holds for each component separately, i.e.*

$$(a, b) \sqsubseteq (a', b') \quad \Leftrightarrow \quad a \sqsubseteq a' \ \wedge \ b \sqsubseteq b' \qquad (\sqsubseteq \text{ of pairs})$$

Bottom, top, complement, meet and join in the product lattice are component-wise extensions of the corresponding operations of the component lattices. Product of two lattices preserves their distributivity, completeness and boolean properties. For example, out of the four lattices in Figure 1, only (2-Bool, ⊑) and (4-Bool, ⊑) are boolean. The former is boolean because ¬ T = F, ¬ F = T. The latter is a product of two (2-Bool, ⊑) lattices and thus is complete, distributive and boolean. The lattice (3-QBool, ⊑) is not boolean because ¬ M = M, and M ⊓ ¬ M $\neq \bot$.

Finally, we define an operator $\rightarrow$ as follows:

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

3 Model-Checking

In this section we present an overview of our model checker, χchek. χchekis based on classical *Computational Tree Logic* (CTL) model checkers [Clarke *et al.*, 1986]. We first briefly introduce classical CTL model checking. We then extend this to quasi-boolean multi-valued logics by defining multi-valued state machine models, and extending the semantics of CTL.

3.1 Classical Model Checking

CTL model-checking is an automatic technique for verifying properties expressed in a propositional branching-time temporal logic called *Computational Tree Logic* (CTL) [Clarke *et al.*, 1986]. The system is defined by a Kripke structure, and properties are evaluated on a tree of infinite computations produced by the model of the system. The standard notation $M, s \models P$ indicates that a formula P holds in a state s of a model M. If a formula holds in the initial state, it is considered to hold in the model.

A Kripke structure consists of a set of states S, a transition relation $R \subseteq S \times S$, an initial state $s_0 \in S$, a set of atomic propositions A, and a labeling function $L : S \rightarrow \mathcal{P}(A)$. To ensure that all paths are infinite length, every state must have at least one transition out of it, i.e. $\forall s \in S,\ \exists t \in S : (s, t) \in R$. If a state s_n has no successors, we add a self-loop to it, so that $(s_n, s_n) \in R$. For each $s \in S$, the labeling function provides a list of atomic propositions which are *True* in this state.
CTL is defined as follows:

1. Every atomic proposition $a \in A$ is a CTL formula.

2. If φ and ψ are CTL formulae, then so are $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $EX\varphi$, $AX\varphi$, $EF\varphi$, $AF\varphi$, $EG\varphi$, $AG\varphi$, $E[\varphi U \psi]$, $A[\varphi U \psi]$.

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

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

where the remaining operators are defined as follows:

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

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

3.2 Multi-valued state machines

Conventionally, a state machine model consists of a set of states, a set of transitions between states, and a set of variables whose values vary from state to state. We extend this by associating each model with a specific Quasi-Boolean logic. 'Boolean' variables now range over the values of the logic, rather than just being TRUE or FALSE.

Transitions between states also range over the values of the logic. In a conventional state machine model, all transitions are implicitly TRUE, because FALSE transitions (i.e. transitions that cannot occur) are simply omitted from the notation. If we extend this to the multi-valued case, we can assign any value of the logic to each transition. This allows us to model cases where information sources disagree over which transitions can occur. To avoid clutter, we adopt the convention that FALSE transitions (i.e. transitions taking the value $\perp$) are omitted from our diagrams.

For model checking purposes, we also need to define an initial state. In a conventional state machine model, there is one initial state. Because stakeholders may disagree on the initial state, we allow a model to have a *set* of initial states.

We call our multi-valued models χviews. Formally, a χview is defined as a tuple (L, S, S_0, R, I, A) where:

- L is a quasi-boolean logic given as a lattice $(\mathcal{L}, \sqsubseteq)$.
- A is a (finite) set of atomic propositions
- S is a (finite) set of states, each with a unique label;

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

Figure 2: Properties of CTL operators.

- $S_0 \subseteq S$ is the non-empty set of initial states.
- $R : S \times S \rightarrow \mathcal{L}$ is a total function assigning a truth value from the logic, L to each possible transition between states (including the transition from each state to itself). Each state must have at least one non-FALSE transition out of it;
- $I : S \times A \rightarrow \mathcal{L}$ is a total function giving a truth value to each variable in each state. For simplicity we assume that all our variables are of the same type, ranging over the values of the logic. For a given variable a, we will write I as $I_a : S \rightarrow \mathcal{L}$.

Note that if the logic L is a two-valued boolean logic, then a χview reduces to a standard Kripke structure. By adopting Kripke structures as our underlying formalism, we gain generality and analytical power but lose some expressive power. However, many modeling languages can be translated into Kripke structures (e.g. SCR [Atlee and Gannon, 1993]), and we plan to eventually adopt a richer specification language as a front end to our framework.

Central to a symbolic model checking algorithm is the computation of *partitions* of the state space w.r.t. a variable a using $I_a^{-1} : \mathcal{L} \rightarrow 2^S$. A partition has the following properties:

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

3.3 Multi-Valued CTL

We extend the semantics of CTL operators to a χKripke structure M over a quasi-boolean logic L. We will refer to this language as *multi-valued CTL, or χCTL*. L is described by a finite, quasi-boolean lattice $(\mathcal{L}, \sqsubseteq)$, and thus the conjunction $\sqcap$, disjunction $\sqcup$ and negation $\neg$ operations are available. In extending the CTL operators, we want to ensure that the expected CTL properties, given in Figure 2, are preserved. Note that the AU fixpoint includes an additional conjunct, $EXA[fUg]$. This preserves a "strong until" semantics for states that have no outgoing $\top$ transitions [Bultan *et al.*, 2000].

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

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

We proceed by defining EX. In standard CTL, this operator is defined using the existential quantification over next states. We define quantification for our multi-valued logics using conjunction and disjunction for universal and existential quantification, respectively. This treatment of quantification in multi-valued logics is standard [Belnap, 1977; Rasiowa, 1978]. EX is defined by:

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

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

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

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

3.4 Model-Checking Algorithm

In this section we give an overview of the design of our symbolic multi-valued model checker, χchek. χchek takes as input a model M taking its variable and transition values from a lattice $\mathcal{L}$, and a χCTL formula φ. It produces as output a total mapping from $\mathcal{L}$ to the set S of states, indicating in which states φ takes each value ℓ. This is simply P_φ^{-1}, the inverse of the valuation function defined in section 3.3; and thus, the task of the model checker is to compute P_φ given the transition function R.

Since states are assignments of values to the variables, an arbitrary ordering imposed on A allows us to consider a state as a vector in $\mathcal{L}^n$, where $n = |A|$. Hence P_φ and R are functions of type $\mathcal{L}^n \rightarrow \mathcal{L}$ and $\mathcal{L}^{2n} \rightarrow \mathcal{L}$ respectively. Such functions are represented within the model checker by multi-valued decision diagrams (MDDs), a multi-valued extension of the binary decision diagrams (BDDs) [Bryant, 1992].

We give a detailed treatment of MDDs in [Chechik *et al.*, 2001a]. Here we illustrate them by means of a brief example.

Definition 3. *[Srinivasan* et al.*, 1990] 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*

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

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

Definition 4. *Assuming a finite set of logic values, L, and an ordered set of variables A, multi-valued decision diagram (MDD) is a tuple $(V, E, \mathsf{var}, \mathsf{child}, \mathsf{image}, \mathsf{value})$ where*

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

For example, consider the function $f = x_1 \wedge x_2$, with $\ell_0 = \mathrm{F}, \ell_1 = \mathrm{M}, \ell_2 = \mathrm{T}$. The MDD built from this expression, and its lattice, are shown in Figure 3. The diagram is constructed by Shannon expansion, first with respect to x_1, and then (for each cofactor of f) with respect to x_2. The dashed arrows indicate f and its cofactors, and also the cofactors of the cofactors.

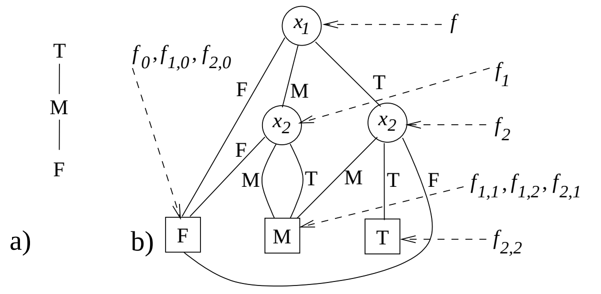


Figure 3: (a) A three-valued lattice. (b) The MDD for $f = x_1 \wedge x_2$ in this lattice.

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

Algorithms for manipulating BDDs are extensible to the multi-valued case, provided they do not use optimizations that depend on a two-valued boolean logic (e.g. complemented edges [Somenzi, 1999]). The differences are discussed in [Chechik *et al.*, 2001a]. The public methods required for model checking are: `Build`, to construct an MDD based on a function table; `Apply`, to compute $\wedge, \vee$ and $\neg$ of MDDs; `Quantify`, to existentially quantify over the primed variables; and `AllSat` to retrieve the computed partition $P_\varphi^{-1}(\mathcal{L})$. `Build` ensures orderedness of MDDs while they are being constructed, and `Apply` preserves it. An additional function, `Prime`, primes all of the variables in an MDD.

The full model checking algorithm is given in Figure 4. The function `EX`(P_φ) computes $P_{EX\varphi}$ symbolically; `QUntil` carries out the fixed-point computation of both AU and EU. $AX\varphi$ is computed as $\neg EX\neg\varphi$. EG, AG, EF, and AF are not shown in this Figure, but could be added as cases and defined in terms of calls to `EX`, `QUntil`, and `Apply`.

Correctness and efficiency of this algorithm are analysed in [Chechik *et al.*, 2001a]. The worst-case running time for χchek is $O(|\mathcal{L}|^{3n} \times h)$, where n is the number of variables and h is the height of the lattice. Experimental results suggest that the average case is $O(|\mathcal{L}|^{3n-2} \times h \times |p|)$, where $|p|$ is the

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

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

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

Figure 4: The multi-valued symbolic model checking algorithm.

size of the $\mathcal{X}$CTL formula. This compares favourably with classical model checking.

4 Merging and Analyzing $\mathcal{X}$views

We use $\mathcal{X}$chek for automated reasoning about models created my merging information from multiple sources. Our analysis process is shown in Figure 5. We take a set of source $\mathcal{X}$views and merge them using a set of interconnection primitives and a merge template chosen by the analyst. The resulting merged $\mathcal{X}$view can then be model checked against a set of properties expressed in $\mathcal{X}$CTL. The model checker returns the value(s) from the logic that each property takes in the initial state(s). We use the same multi-valued state machine notation for both the source $\mathcal{X}$views and the merged $\mathcal{X}$view. This enables us to run the model checker on individual $\mathcal{X}$views, as well as the merged ones, and to perform further merges on an already merged $\mathcal{X}$view. Note that as classical two-valued logic is also a quasi-boolean logic, any $\mathcal{X}$view may just be a classical state machine model.

4.1 Signature and Value Maps

For software engineering applications, we have identified a number of different ways of composing models. For example, given a set of $\mathcal{X}$views, there are a number of possible relationships between the behaviours they describe:

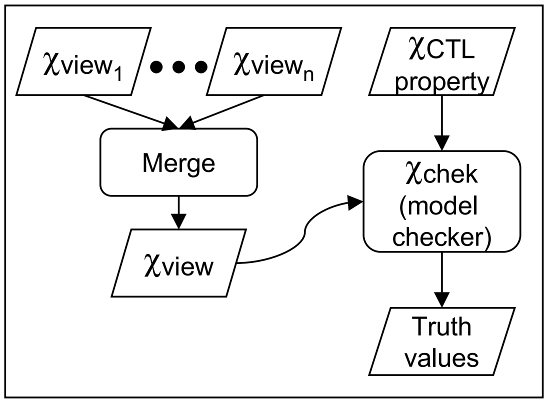


Figure 5: The analysis process.

- They may be *parallel* devices that interact through shared data or shared events.
- They may be *projections* of the overall state space of a system—each view describes some of the states and some of the transitions, leaving other parts undefined.
- They may be competing *versions* of a system, differing over some of the variables or transitions, where each view claims to describe all the possible behaviours of the system.
- They may be *features* that add new behaviours and/or modify existing behaviours of a system.

All of these are supported. Combinations of these are also possible: versions of parallel devices; projections of a feature; and so on.

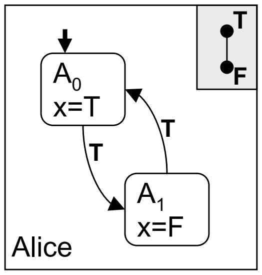
We support exploratory approaches to composing models by allowing the analyst to choose which logic to use for the merged view, how to unify the vocabularies of the source views, and how to map truth values of the source $\mathcal{X}$views onto truth values of the merged $\mathcal{X}$view. Figure 6 presents a simple example.

The first step in merging a set of $\mathcal{X}$views is to define a *signature map* that unifies their vocabularies. Rather than assuming that the $\mathcal{X}$views share the same vocabulary, we allow each $\mathcal{X}$view to preserve its local namespace, and allow the analyst to determine which states and variables should be unified across the source $\mathcal{X}$views. The analyst may choose to rename states and variables in the merged $\mathcal{X}$view, or may keep some of the names from the source $\mathcal{X}$views. Figure 6(c) shows an example signature map.

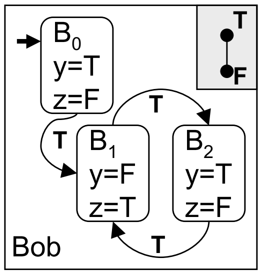
A signature map must have the following properties:

1. Type information is preserved—state names can only be mapped to state names, and variable names can only be mapped to variable names.
2. Every state in the source $\mathcal{X}$views must map to a state in the merged $\mathcal{X}$view. However, not all variables need to be mapped—variables can be 'private' to the source $\mathcal{X}$views, and not appear in the merged $\mathcal{X}$view.
3. A name in a source $\mathcal{X}$view may map to more than one name in the merged $\mathcal{X}$view. This allows us to duplicate a variable (or state) and treat the instances differently. This is useful for $\mathcal{X}$views that are at different levels

(a)



(b)



(c) Signature Map

Alice	Bob	Comb.
A$_0$	B$_0$	C$_0$
A$_1$	B$_2$	C$_1$
–	B$_1$	C$_2$
x	z	a
–	y	y

(d) Value Map

Alice	Bob	Comb.
T	T	TT
T	F	TF
F	T	FT
F	F	FF
T	*UNDEF*	TT
UNDEF	T	TT
F	*UNDEF*	FF
UNDEF	F	FF

(e)

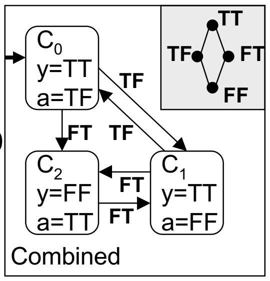


Figure 6: (a)-(b) Sample χviews; (c) a signature map used to unify the vocabularies; (d) a value map; (e) result of the merge.

of granularity, where a single state in one χview corresponds to several states in another χview.

4. Two different names from the same source χview cannot be mapped to the same name in the merged χview.

The next step is to define how the truth values of the source χviews are combined. A *value map* is a total function mapping each tuple of truth values in the source χviews to a truth value of the merged χview. In many cases the chosen logic for the merged χview has a 'natural' value map. In general, we expect there to be a small number of commonly-used logics and value maps.

The final step is to extend the value map to handle gaps in the available information during a merge. For example, if we have a state, s, from one source χview, and a variable, a, from another, we may not be able to determine what value a should take in s.

4.2 Model Checking Merged Models

The model checker χchek allows us to verify properties of our χviews. For example, given the χview of Figure 6(e), we can check the value of $AX(a = \text{FF})$ (roughly: "a is FALSE in the next state on all paths (from the initial state)"). χchek returns the value TF, indicating that this property is TRUE in Alice's χview and FALSE in Bob's. Similarly, for $EF(a = \text{TT} \vee a = \text{FF})$ (roughly: "you can reach a state where they agree on the value of a"), χchek returns TT. Note that this question cannot be expressed in Alice or Bob's individual χviews.

Interpreting the results returned by the model checker on a merged χview requires some knowledge of the type of merge that was used. For example, the value map in Figure 6(d) represents a specific choice about the relationship between Alice's and Bob's models: if only one person can answer the question, we take that person's answer as undisputed. Thus, the property $AG(y = \text{TT})$ is FF for the χview in Figure 6(e), but the value map does not allow us to distinguish whether this is because the property is FALSE in each individual χview, or because it is FALSE in one and *UNDEF*in the other. If we really need to know which is the case, then a different type of merge that distinguishes these possibilities would be needed.

5 Conclusions

In this paper we have presented an extension of classical CTL model checking for a family of multi-valued logics. The model checker works for any logic whose values form a quasi-boolean lattice. We use the model checker as an automated reasoning tool for analyzing disagreements when merging information from multiple sources.

Because the analysis is fully automated, we can support an exploratory approach to merging information: an analyst can analyze whether critical properties still hold if information is composed in different ways. Furthermore, the tool can be used to support negotiation of conflicting viewpoints. The model checker will determine whether disagreements in the details of each model affect various key properties of the combined model. Proposed changes to the individual models can then be rapidly assessed for their effect on the level of agreement about these key properties.

Compositionality is an important problem for formal modeling in software engineering. The principle of *separation of concerns* allows software engineers to manage the complexity of large systems by modeling separately different aspects. In formal modeling languages, this has been supported by including various composition primitives within the modeling language. However, the available set of composition primitives in each modeling language has been limited. By treating composition at the logical level rather than embedding it in the language, our approach provides greater flexibility. We plan to extend our work by mapping various common modeling languages into our χview formalism.

Finally, we anticipate a wider set of applications for this approach beyond its application in software engineering. The model checker can be adapted to work for different types of modal logic, and the underlying logics are general enough to support general modeling tasks where information from multiple conflicting sources must be combined.

Acknowledgments

We thank members of the University of Toronto formal methods reading group, and in particular Ric Hehner, Albert Lai, Benet Devereux, Arie Gurfinkel, Victor Petrovykh and Christopher Thompson-Walsh for numerous interesting and *useful discussions about this work. Financial support was provided by NSERC and CITO.*

References

[Atlee and Gannon, 1993] J.M. Atlee and J. Gannon. "State-Based Model Checking of Event-Driven System Requirements". *IEEE Transactions on Software Engineering*, pages 22–40, January 1993.

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

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

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

[Bultan *et al.*, 2000] 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.

[Burch *et al.*, 1990] J.R. Burch, E.M. Clarke, K.L. McMillan, D.J. Dill, and L.J. Hwang. "Symbolic Model Checking: 10^{20} States and Beyond". In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pages 428–439, June 1990.

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

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

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

[Dunn, 1999] 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.

[Easterbrook and Nuseibeh, 1996] S.M. Easterbrook and B.A. Nuseibeh. "Using Viewpoints for Inconsistency Management". *BCS/IEE Software Engineering Journal*, pages 31–43, January 1996.

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

[Ginsberg, 1998] Matthew L. Ginsberg. "Multivalued Logics: A Uniform Approach to Reasoning in Artificial Intelligence". *Computational Intelligence*, 4(3):265–316, 1998.

[Grundy *et al.*, 1998] J. Grundy, J. Hosking, and W. B. Mugridge. "Inconsistency Management for Multiple-View Software Development Environments". *IEEE Transactions on Software Engineering*, 24(11):960–981, 1998.

[Hähnle, 1994] Reiner Hähnle. *Automated Deduction in Multiple-Valued Logics*, volume 10 of *International Series of Monographs on Computer Science*. Oxford University Press, 1994.

[Hunter and Nuseibeh, 1998] A. Hunter and B. Nuseibeh. "Managing Inconsistent Specifications: Reasoning, Analysis and Action". *ACM Transactions on Software Engineering and Methodology*, 7(4):335–367, October 1998.

[Łukasiewicz, 1970] J. Łukasiewicz. *Selected Works*. North-Holland, Amsterdam, Holland, 1970.

[Menzies *et al.*, 1999] T.J. Menzies, S. Easterbrook, B. Nuseibeh, and S. Waugh. "An Empirical Investigation of Multiple Viewpoint Reasoning in Requirements Engineering". In *Proceedings of the Fourth International Symposium on Requirements Engineering (RE'99)*, Limerick, Ireland, June 7-11 1999. IEEE Computer Society Press.

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

[Robinson and Pawlowski, 1999] W.N. Robinson and S.D. Pawlowski. "Managing Requirements Inconsistency with Development Goal Monitors". *IEEE Transactions on Software Engineering*, 25(6):816–835, 1999.

[Schwanke and Kaiser, 1988] R. W. Schwanke and G. E. Kaiser. "Living With Inconsistency in Large Systems". In *Proceedings of the International Workshop on Software Version and Configuration Control*, pages 98–118, Grassau, Germany, January 27-29 1988. B. G. Teubner, Stuttgart.

[Sofronie-Stokkermans, 2000] Viorica Sofronie-Stokkermans. Automated theorem proving by resolution for finitely-valued logics based on distributive lattices with operators. *Multiple-Valued Logic*, 2000.

[Somenzi, 1999] Fabio 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.

[Srinivasan *et al.*, 1990] 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.

[van Lamsweerde *et al.*, 1998] A. van Lamsweerde, R. Darimont, and E. Letier. "Managing Conflicts in Goal-Driven Requirements Engineering". *IEEE Transactions on Software Engineering*, 24(11):908–926, 1998.