A Framework for Multi-Valued Reasoning over Inconsistent Viewpoints

Steve Easterbrook Marsha Chechik
Department of Computer Science
University of Toronto
Toronto, Canada M5S 3H5
{sme,chechik}@cs.toronto.edu

Abstract

In requirements elicitation, different stakeholders often hold different views of how a proposed system should behave, resulting in inconsistencies between their descriptions. Consensus may not be needed for every detail, but it can be hard to determine whether a particular disagreement affects the critical properties of the system. In this paper, we describe the χbel framework for merging and reasoning about multiple, inconsistent state machine models. χbel permits the analyst to choose how to combine information from the multiple viewpoints, where each viewpoint is described using an underlying multi-valued logic. The different values of our logics typically represent different levels of agreement. Our multi-valued model checker, χchek, allows us to check the merged model against properties expressed in a temporal logic. The resulting framework can be used as an exploration tool to support requirements negotiation, by determining what properties are preserved for various combinations of inconsistent viewpoints.

1 Introduction

Requirements Engineering (RE) is significantly complicated by the inconsistent and incomplete nature of the information available early in the software lifecycle. Different stakeholders often use different vocabularies, talk about different aspects of the problem, have different ways of structuring their descriptions, and may have conflicting goals. For these reasons, information gathered from different stakeholders can be difficult to consolidate. It can be hard just to distinguish which things the various stakeholders agree about, and which things they disagree about.

Viewpoints-based approaches have been proposed as a way of managing inconsistent and incomplete information gathered from multiple sources [10]. These approaches separate the descriptions provided by different stakeholders, and concentrate on identifying and resolving conflicts between them. A key advantage to the use of viewpoints is that inconsistencies between viewpoints can be tolerated [9]. The work on viewpoints has inspired a number of tools for managing inconsistent descriptions during requirements modeling (e.g. [11, 18, 20]).

Given an inconsistent set of viewpoints, it would be useful to determine how the inconsistencies affect critical system properties. Some inconsistencies may be of little consequence, and do not need to be resolved, while others may be the result of fundamental disagreements about how a system should behave. For example, if the stakeholders' descriptions are modeled as state machines, they may disagree about the details of some states and transitions, without affecting the values of global temporal properties. If the models were consistent and complete, they could be verified using *model-checking* [6]. Unfortunately, existing model checkers are based on classical logic, and so cannot cope with inconsistent (and incomplete) models.

Reasoning based on classical logic cannot solve the problem because the presence of a single contradiction results in trivialization—anything follows from $A \wedge \neg A$, and so all inconsistencies are treated as equally bad. Furthermore, we cannot rely on reasoning about the properties of individual viewpoints and comparing the results, because these properties may change depending on how the viewpoints are combined. Worse still, we may not even be able to *express* global properties over individual viewpoints, because each viewpoint is only a partial model of the system. Hence, faced with an inconsistent set of viewpoints, if we want to perform automated reasoning, we must either remove information until consistency is achieved again, or adopt a non-classical, *paraconsistent* logic. The problem with the former approach is that we may be forced to make premature decisions about which information to discard [12].

Paraconsistent logics permit some contradictions to be true, without the resulting trivialization of classical logic. For example, multi-valued logics use additional truth values to represent different types of contradiction. Multi-valued logics are particularly interesting, as they can handle both inconsistency and incompleteness. For example, Belnap proposed a simple 4-valued logic for incrementally adding information to a database without enforcing consistency [3]. Belnap observed that for any given proposition, A, there are four cases: we have been told nothing about A; we have been told A is TRUE; we have been told A is FALSE; or we have been told both A is TRUE and A is FALSE. His logic therefore adds the values BOTH and NEITHER to the usual TRUE and FALSE.

In this paper, we present the χbel[1] (Multi-Valued **B**elief **E**xploration **L**ogics) framework for merging and reasoning about inconsistent viewpoints. The framework uses a family of multi-valued logics, called *Quasi-Boolean* logics, to support reasoning over inconsistent models. Each logic has a number of different truth values between TRUE and FALSE, representing different types (or levels) of disagreement and uncertainty. The framework includes a multi-valued model checker, χchek, for reasoning about the temporal properties of inconsistent state machine models.

The framework is intended as a way of exploring inconsistencies. Hence, we do not restrict the analyst to any particular way of merging information from multiple viewpoints, or any particular way of handling disagreement. Rather, we provide tools for defining suitable multi-valued logics and for defining different types of interconnection between viewpoints. For example, viewpoints may describe different features of the same system, give different descriptions of the same functionality, or specify individual processes that need to be composed in parallel. We also provide guidance on how to choose suitable analysis strategies for a given set of viewpoints.

The paper is organized as follows. Section 2 surveys recent work on specifying and reasoning in the presence of inconsistency in software engineering. Section 3 give an overview of the χbel framework. Section 4 gives a formal foundation for multi-valued reasoning. Section 5 describes the modeling process, showing how we merge viewpoints, and analyze them using the model checker χchek. Section 6 demonstrates the framework with a telephony example. Section 7 presents our conclusions.

2 Reasoning in the Presence of Inconsistency

In software engineering, it has long been recognized that tolerating inconsistent descriptions can facilitate flexible collaborative working. For example, Schwanke and Kaiser [19] argue that attempting to enforce total consistency during incremental development can be difficult, and it is therefore preferable to allow inconsistencies to occur, and to resolve them periodically. Narayanaswamy and Goldman [16] describe an incremental inconsistency handling solution based on announcing and interleaving "proposed changes", while Balzer [2] introduced the notion of "pollution markers" to flag portions of program code that contain inconsistencies. Inconsistency has also been studied in the context of process modeling. Cugola [7] argues that process improvement can be achieved by allowing deviations from the prescribed process, and by providing support for dealing with the resulting inconsistencies.

Tools that provide explicit support for identifying, tracking and resolving inconsistencies during requirements modeling are emerging [9, 11, 18, 20]. In particular, the ViewPoints framework [10] exploits the natural structure of the modeling process to collect information in coherent, but overlapping chunks. Because viewpoints can overlap, there is the potential for inconsistency. However, inconsistencies between viewpoints can be dealt with separately from the task of describing and elaborating each viewpoint. It is this toleration of inconsistency that distinguishes viewpoints from other problem structuring techniques.

While these tools focus on detecting and managing inconsistency, none support formal reasoning over inconsistent descriptions. This is because formal reasoning systems based on classical logic cannot cope with inconsistency; the presence of a single contradiction results in trivialization. To address this problem, a number of different types of paraconsistent logics have been proposed [17, 12]. For example, relevance logics use an alternative form of entailment that requires there to be a "relevant" connection between the antecedents and the consequents. Non-truth functional logics use a weak form of negation so that proof rules such as disjunctive syllogism fail, i.e. $(A \vee B, \neg B) \nvdash A$. Multi-valued logics use additional truth values to represent intermediate values between TRUE and FALSE.

The development of paraconsistent logics has been driven largely by the need for automated reasoning systems that do not give spurious answers if their databases become inconsistent [3]. They are also of interest in mathematics as a way of addressing paradoxes in semantics and set theory.

In software engineering, there are two obvious applications of paraconsistent logics: to reason about information drawn from multiple sources during requirements elicitation, and to reason about evolving specifications where changes may introduce inconsistencies. However, there have been relatively few attempts to explore these applications. Two notable exceptions are Hunter and Nuseibeh [13], who use a Quasi-Classical (QC) logic to reason about evolving specifications, and Menzies et al. [15], who use a paraconsistent form of abductive inference to reason about information from multiple viewpoints.

In this paper we describe a new application of paraconsistent logic, to reason about the properties of state machine models constructed by combining information from incon-

[1] pronounced "Ki-bel"

sistent viewpoints. We use multi-valued logics to provide paraconsistency, and adapt existing model checking techniques for our automated reasoning system.

3 Framework Overview

The χbel framework provides a flexible approach to merging and reasoning about inconsistent state machine models using multi-valued logics. The framework provides the following components:

- A basic viewpoint model, or χview, which is a state machine model, extended for the multi-valued case.
- Interconnection primitives for defining how to combine χviews into a merged model. These handle differences in vocabulary across the χviews and support a range of different types of interaction between them.
- A set of merge templates, to guide the choice of multi-valued logic and interconnection primitives for different types of analysis.
- Multi-valued logics for reasoning over χviews.
- A multi-valued model checker, χchek [5], for verifying the temporal properties of χviews.

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, we can reason about the level of agreement for any arbitrary proposition.

The choice of values to use in the logic depends on how we wish to combine information from individual viewpoints. There are a number of dimensions to this choice:

- Do we wish to preserve information about how many viewpoints gave each different response (i.e. so '*4 trues, 1 false*' is distinct from '*3 trues, 2 falses*')?
- Do we wish to preserve information about *who* said what (i.e. so that '*A said true, B said false*' is distinct from '*A said false, B said true*')?
- Do we wish to allow viewpoints to say '*don't know*' for some propositions?
- Do we wish to designate some viewpoints as authorities, so that their answers count more?

Because it may be useful to explore different ways of combining information from multiple viewpoints during requirements modeling, we do not commit ourselves to any particular multi-valued logic. Rather, our framework uses a family of logics, and each viewpoint has a specific logic associated with it. When we merge viewpoints, the type of merge determines the logic that will be chosen for the resulting viewpoint. In practice, we expect that a small number of merge types will be useful.

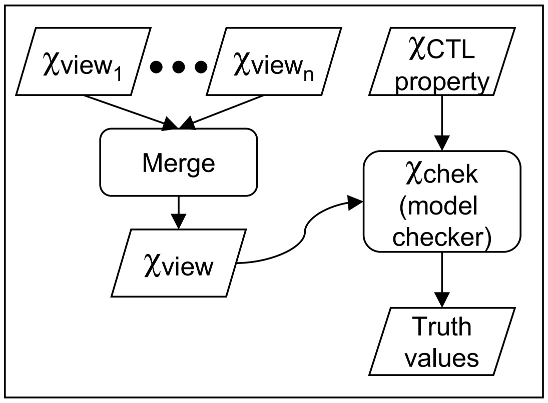


Figure 1. The process model.

Our analysis process is shown in Figure 1. We take a set of source χviews and merge them using a set of interconnection primitives and a merge template chosen by the analyst. The resulting merged χview can then be model checked against a set of properties expressed in χCTL, our multi-valued temporal logic, an extension of 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 χviews and the merged χview. This enables us to run the model checker on individual χviews, as well as the merged ones, and to perform further merges on an already merged χview.

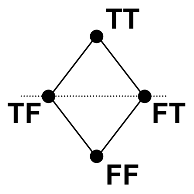
The next section explains our multi-valued logics, while Section 5 describes the merge process.

4 Multi-Valued Reasoning

4.1 Quasi-Boolean Multi-Valued Logics

Our approach to modeling makes use of an entire family of multi-valued logics. In order to ensure that reasoning in our logics makes intuitive sense, we restrict ourselves to logics where the axiomatization is as close to classical logic as possible. For example, we wish to keep useful properties such as associativity, idempotency, distributivity, and De Morgan's laws. However laws relating to incompleteness and inconsistency, such as the law of excluded middle ($a \vee \neg a = \top$) and the law of non-contradiction ($a \wedge \neg a = \bot$), may be discarded.

We achieve this by defining each logic using a lattice of truth values, and define the logical operators in terms of lattice operations. Lattices are useful in this respect because they guarantee most of the properties we need. For example, a lattice by definition includes a unique least upper bound (a *join*) and a unique greatest lower bound (a *meet*) for each



$\vee$	TT	TF	FT	FF
TT	TT	TT	TT	TT
TF	TT	TF	TT	TF
FT	TT	TT	FT	FT
FF	TT	TF	FT	FF
$\neg$	FF	FT	TF	TT

Figure 2. The logic 4val as a lattice, together with the truth tables for disjunction, $\vee$, and negation, $\neg$.

pair of elements. By defining disjunction as join, and conjunction as meet in the lattice, we ensure that a disjunction and a conjunction of each pair of values exists and is unique in the logic. Furthermore, these operations are commutative, associative and idempotent in a lattice. To see how the corresponding truth tables can be constructed from the lattice, consider the 4-valued logic **4val** shown in Figure 2. The truth table for disjunction was constructed by taking the join for each pair of values. The table for conjunction is formed similarly using the meet. By specifying the lattice, we avoid having to give the tables.

As well as conjunction and disjunction, we need to specify negation so that most of its expected properties hold: each element must have a negation which is a value in the lattice, such that $\neg\neg a = a$, and negation satisfies De Morgan's laws. The family of multi-valued logics that have these properties (but not necessarily the laws of excluded middle and non-contradiction) are called *quasi-boolean* logics [4]. Rather than providing a formal definition here[2], we note that symmetry of the lattice diagram across its horizontal axis is a sufficient condition for a quasi-boolean logic, where the negation of each element is defined to be its image through horizontal symmetry. Finally, we still have to choose a negation for values that fall on the axis of symmetry. For example, in Figure 2 we chose to make TF and FT negations of each other; we could equally have chosen to make them their own negations, which would have given us a different logic.

In summary, we restrict ourselves to quasi-boolean logics, and specify them as horizontally-symmetric lattices of truth values, together with a definition of negation. Note that classical 2-valued boolean logic is included in this family as a 2-valued lattice — we refer to it as **2val**.

4.2 Some Example Multi-Valued Logics

When merging two viewpoints, each with its own logic, we could simply take the *product* of their lattices as the logic for the merged viewpoint. A product of two lattices is a lattice where each element is a pair (a, b) made up of an element, a, from the first lattice and and element b from the second lattice[3]. The lattices of figures 3(a), 3(c), and 3(e) are products. Products are interesting because these are exactly the lattices that preserve *all* information about the individual logics. Products can be taken of more than two lattices (e.g. Figure 3(e) is the product of three lattices), but the number of values in the resulting logic grows exponentially. For this reason, an abstraction of the product is often more useful. The lattices on the right of Figure 3 are abstractions of those on the left. Each discards some information—for example, 3(f) only retains information about majority votes.

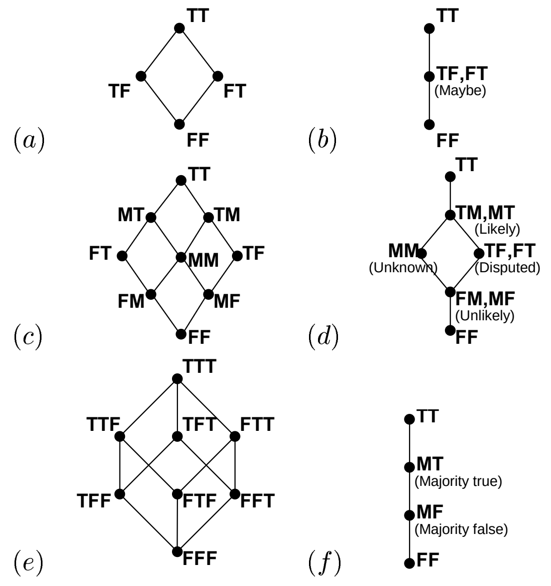


Figure 3. Some logics for combining viewpoints: (a) 2val$\times$2val; (b) 3val; (c) 3val$\times$3val; (d) an abstraction of c; (e) 2val$\times$2val$\times$2val; (f) an abstraction of e

The choice of whether to use the product, or an abstraction of the product depends on the reasoning we wish to perform. Consider the logic **4val** in Figure 3(a), which can be used to merge two viewpoints based on **2val**. Because it is the product, **4val** preserves all information about who said what, allowing us to make precise inferences about what the two viewpoints would agree on. For example, if the first viewpoint said "x is TRUE" and "y is FALSE", while our second viewpoint said the opposite, then in the merge we obtain $x=$TF and $y=$FT. We can then infer that $x \vee y =$TT. In other words, the viewpoints still agree that at least one of x or y is true, even though they disagree about the actual values of x and y.

In contrast, in **3val** (Figure 3(b)), all disagreements are treated as MAYBE — we discard the distinction between TF

[2]A complete treatment can be found in [5].

[3]Formally, the product of two lattices $\mathcal{L}_1$, $\mathcal{L}_2$, is a lattice $\mathcal{L}_1 \times \mathcal{L}_2$ with elements (a, b), such that the lattice ordering $\sqsubseteq$ holds between two pairs iff it holds for each component separately, i.e.

$$(a, b) \sqsubseteq (a', b') \;\Leftrightarrow\; a \sqsubseteq a' \;\wedge\; b \sqsubseteq b'$$

and FT. We can still reason about what is agreed and disagreed, except that we will obtain more MAYBE answers than we should. For example, if we have the same values for x and y as above, then in the merged model, $y = x =$ MAYBE. In this logic, $x \vee y =$MAYBE, whereas we saw above that the viewpoints actually agree that $x \vee y =$TRUE. Despite this information loss, this type of merge is useful when product lattices become too large.

4.3 χviews

We formalize our viewpoints as χviews. χviews are state machine models, extended with a multi-valued logic. 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 χview 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 stakeholders disagree over which transitions can occur. To avoid clutter, we adopt the convention that FALSE transitions[4] 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 χview to have a *set* of initial states.

Formally, a χview is a Kripke structure [6], extended for the multi-valued case. It can be defined as a tuple (L, S, S_0, R, I, A) where:

- L is a quasi-boolean logic defined by a lattice with elements $\mathcal{L}$ and a negation operator $\neg$;
- S is a set of states, each with a unique label;
- $S_0 \subseteq S$ is a (non-empty) set of initial states;
- $R : S \times S \to \mathcal{L}$ is a total function assigning a truth value from the logic, L to each possible transition between states (including the transition from each state to itself). Each state must have at least one non-FALSE transition out of it;
- A is a set of atomic propositions, or variables;
- $I : A \times S \to \mathcal{L}$ is a total function giving a truth value to each variable in each state.

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 standard state-machine specification languages can be translated into Kripke structures (e.g. SCR [1]), and we plan to eventually adopt a richer specification language as a front end to our framework. Also, χviews do not have an explicit representation of time, although we plan to add this in the future.

[4]In this paper we use TRUE and FALSE to refer to the top of the lattice, $\top$, and the bottom of the lattice, $\bot$, respectively.

5 Merging and Analyzing χviews

5.1 Signature and Value Maps

Given a set of χviews, we can imagine a number of different relationships between the behaviours they describe:

- 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.

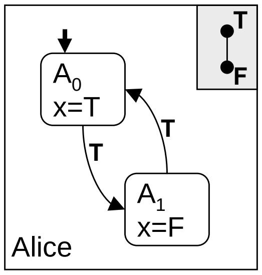
All of these are supported in our framework. Combinations of these are also possible: versions of parallel devices; projections of a feature; and so on.

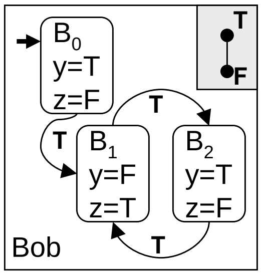
Furthermore, an analyst may want to explore different ways of combining the same set of χviews. We support this flexibility by allowing the analyst to choose which logic to use for the merged view, how to unify the vocabularies of the source views, and how to map truth values of the source χviews onto truth values of the merged χview.

The first step in merging a set of χviews is to define a *signature map* that unifies their vocabularies. Rather than assuming that the χviews share the same vocabulary, we allow each χview to preserve its local namespace, and allow the analyst to determine which states and variables should be unified across the source χviews. The analyst may choose to rename states and variables in the merged χview, or may keep some of the names from the source χviews. Figure 4(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 χviews must map to a state in the merged χview. However, not all variables need to be mapped—variables can be 'private' to the source χviews, and not appear in the merged χview.
3. A name in a source χview may map to more than one name in the merged χview. This allows us to duplicate

(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

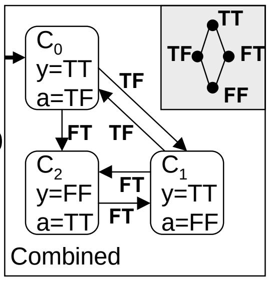
(e)


Figure 4. (a)-(b) Sample χviews; (c) a signature map used to unify the vocabularies; (d) a value map; (e) result of the merge.

a variable (or state) and treat the instances differently. This is useful for χviews that are at different levels 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. For example, each lattice in Figure 3 was designed with a specific value map in mind, and the elements of the lattice were labeled accordingly. 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. Value maps are defined over tuples of values from the source χviews. We may get undefined entries in a tuple in six different cases as follows:

When querying the value of variable x in state s:
UNDEF$_1$: The χview knows about s, but not x.
UNDEF$_2$: The χview knows about x, but not s.
UNDEF$_3$: The χview knows about neither s nor x.
For the value of a transition from state s_1 to state s_2:
UNDEF$_4$: The χview knows about s_1 but not s_2.
UNDEF$_5$: The χview knows about s_2 but not s_1.
UNDEF$_6$: The χview knows about neither s_1 nor s_2.
These are *not* values from the logics—they represent cases where we do not know which value applies. We allow the analyst to choose how to handle each of these cases, by extending the value map appropriately, and provide guidance for each choice. For example, the value map in Figure 4(d) treats all six cases in the same way. Alternatively, we could map *UNDEF*$_4$ to FALSE and *UNDEF*$_5$ and *UNDEF*$_6$ to MAYBE, to indicate a viewpoint that insists there can be no additional transitions from the states it describes, but does not care about transitions from states that other people describe.

5.2 Model Checking Merged Viewpoints

We have developed a symbolic multi-valued χCTL model checker, χchek [5], for analysing χviews. The model checker takes as input a χview, including the definition of its quasi-boolean logic, and a temporal logic property expressed in χCTL, and returns a tuple of values from the logic representing the values that the χCTL property takes in each initial state of the χview.

The language χCTL for expressing properties is based on *Computational Tree Logic* (CTL), a branching-time temporal logic used in model checkers such as SMV [14]. Properties expressed in CTL are evaluated on a tree of infinite computations produced from a finite-state machine expressed as a Kripke structure. CTL is defined as follows:

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

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.

χCTL is a multi-valued extension of CTL that gives a semantics to CTL operators over a χview. χCTL has the same operators as CTL, with the quantifiers redefined for the multi-valued case. [5] gives the full semantics of χCTL.

The model checker χchek allows us to verify properties of our χviews. For example, given the χview of Figure 4(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 4(d) represents a specific choice about the relationship between the viewpoints: 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 4(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.

5.3 Method Guidance

The χbel framework provides a great deal of flexibility for merging and analyzing viewpoints, because there are many possible relationships between viewpoints that an analyst may wish to explore. Choosing a good merge strategy is not easy—the analyst must decide what roles the individual viewpoints play in the overall system, how their vocabularies overlap, and what logics to use. The choices affect the kind of analysis that is possible and the interpretation of the results. We provide guidance in three ways.

First, we maintain a library of useful logics. In most situations, an appropriate logic can be selected from the library for each merge operation.

Second, we define a set of merge templates based on the roles that the individual viewpoints play in the overall system description. For example, a *feature* adds new behaviours and/or modifies existing behaviours of a system. In the χbel framework, we model features as χviews, but add information about how they are to be merged. We declare that some of the states of the χview belong to the system, and some belong only to the feature. The latter are handled differently during merges—we do not unify them with other states in a signature map, and do not permit them to have *external transitions* (transitions to states described in other χviews). This treatment of external transitions is achieved by giving appropriate values to $UNDEF_4$, $UNDEF_5$, $UNDEF_6$, described earlier.

Finally, we identify and prove properties for different merge templates. For example, if we merge two logics onto **3val** such that only TRUE values of individual viewpoints are mapped into T, only FALSE values of individual viewpoints are mapped into F, and all others are mapped into M, then, if the model-checker returns T (respectively, F) for a property on the merged model, this property is TRUE (respectively, FALSE) on the individual models.

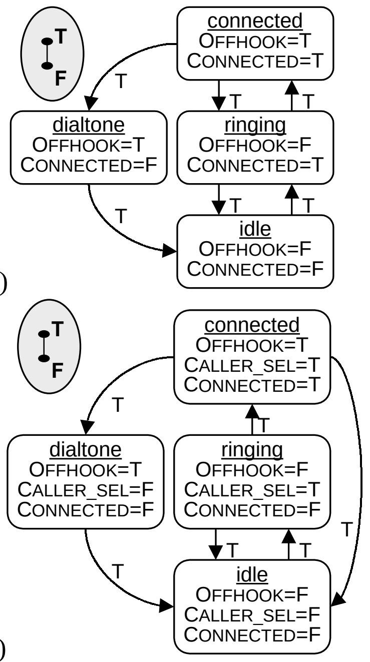


Figure 5. (a) version 1 of callee; (b) version 2 of callee.

6 Example

We now demonstrate how the χbel framework can be used to reason about inconsistencies in a model of a simple telephone system[5]. We separately specify two features, and two different versions of the same feature, and merge these specifications to reason about which properties they agree on, and which are disputed.

6.1 Different Versions of a Feature

Figure 5 shows two different versions of the feature for receiving a call (the "callee feature"). The two models are expressed as χviews, each using the classical two-valued logic, **2val**. In this example, we assume that each state has a TRUE transition back to itself, and don't explicitly draw these. **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

[5]This example is adapted from [9].

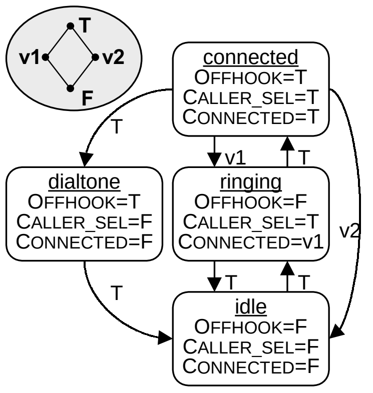


Figure 6. A possible merge of the χviews of Figure 5.

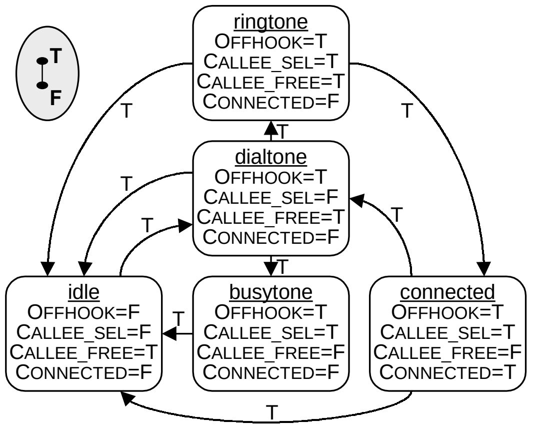


Figure 7. The χview for caller.

disconnects the call. Note that they disagree about the transitions out of the connected state, and they disagree about the value of the variable CONNECTED. Without some analysis, it is hard to tell what these disagreements affect, and hence whether they matter in the overall design.

Having described a feature, we may wish to:

- reason about it in isolation;
- merge it with one or more other features to reason about feature interaction;
- merge it with the base system model to reason about the system behaviour;
- merge it with a model of the environment to check that it captures the intended requirements;

Or some combination of the above. The χbel framework supports all of these; we will illustrate the first two here.

Obviously, we can reason about each feature separately using χchek. More interestingly, we can merge the two versions of the feature (even though they are inconsistent) to reason about the properties they share and the properties they disagree about. We do this using a template for merging multiple version χviews as follows:

1. Choose a logic for the merged viewpoint. Because we wish to maintain traceability between versions and properties, we select the product lattice **4val**. However, we re-label the truth values TF as v1 and FT as v2, as they correspond to '*only* TRUE *in* ***callee1***', and '*only* TRUE *in* ***callee2***', respectively.
2. Choose a signature map. The owners of the χviews *appear* to have used the same vocabulary, so we will try simply mapping together items that share the same name, and preserve these names in the merged χview.
3. Choose a value map. We use the value map for the product **2val**×**2val**, shown in Figure 4(d). The template for merging versions treats all external transitions as FALSE—each version denies the existence of transitions other than those it describes.

The resulting merged χview is shown in Figure 6. Note that the value of CONNECTED is disputed in the ringing state, and that the versions disagree about two of the transitions.

We can now model check this χview. Example properties include: "If you are connected, you can hang up":

$$AG(\text{CONNECTED} \rightarrow EX(\neg\text{OFFHOOK})) \qquad (1)$$

"If connected, hanging up always disconnects you":

$$AG(\text{CONNECTED} \rightarrow AX(\neg\text{OFFHOOK} \rightarrow \neg\text{CONNECTED})) \qquad (2)$$

"A connection does not start until you pick up the phone"[6]:

$$AG(\neg\text{CONNECTED} \rightarrow A[\neg\text{CONNECTED}\, W\, \text{OFFHOOK}]) \qquad (3)$$

"If no caller is selected, you cannot be connected":

$$AG(\neg\text{CALLER\_SEL} \rightarrow \neg\text{CONNECTED}) \qquad (4)$$

χchek returns T for property (1), meaning that both versions agree on its value. It returns v2 for (2) and (3) meaning that only **callee2** has these properties. The disagreement over (3) in particular indicates that the two versions use a different meaning for CONNECTED—the definition in **callee1** would not work with typical billing systems! Finally, it returns T for (4). This is interesting because property (4) is not expressible in **callee1**. That is, **callee1** can have this property, as long as it accepts the definitions given in **callee2** for the missing variables.

6.2 Merging Features

Figure 7 shows an χview for making a call (the "**caller**" feature). It shares some states with the **callee** model and introduces some new states. We can merge this with each of our versions of **callee** to study the *feature interaction* [21]. Alternatively, we could combine **caller** with the merged model of both **callee**s, to combine elements of all three. For example, we may wish to extend **callee1** with the extra variable defined by **callee2**, and then combine this with **caller**. We can achieve this by merging the χviews of Figures 6 and 7 as follows:

[6]We used the "weak until" operator $A[xWy]$ for this property, defined as $A[xWy] = \neg E[\neg yU(\neg x \wedge \neg y)]$ [8].

Caller-Callee Signature Map

Caller	**Callee**	**Merged**
idle	idle	idle
dialtone	dialtone	dialtone
connected	connected	connected
–	ringing	ringing
busytone	–	busytone
ringtone	–	ringtone
OFFHOOK	OFFHOOK	OFFHOOK
CALLEE_SEL	CALLER_SEL	LINE_SEL
CALLEE_FREE	–	CALLEE_FREE
CONNECTED	CONNECTED	CONNECTED

Caller-Callee Value Map

Caller	**Callee**	**Merged**
T	T	T
T	V1	T
T	V2	TF
T	F	TF
F	T	FT
F	V1	FT
F	V2	F
F	F	F

Figure 8. Vocabulary map and value map for merging callee with caller.

1. We choose a logic for the merged viewpoint. Because we are only interested in feature interactions between **caller** and **callee1**, we can adopt **4val**.
2. We choose a signature map. As before, we choose to map together the names that are common to both χviews. We also map CALLEE_SEL and CALLER_SEL together, as they both refer to the selection of the remote party for a call, and use the new name LINE_SEL for this variable. The full signature map is given in Figure 8.
3. We use the value map shown in Figure 8. To select only the behaviours of **callee1** from the combined model, we treat V1 as though it were T, and V2 as F in defining the map. The template for merging features determines how we treat external transitions: *UNDEF* transitions concerning the shared states idle, dialtone and connected are treated as TRUE, while *UNDEF* transitions concerning the private states ringing, busytone and ringtone are treated as FALSE.

The resulting merged χview is shown in Figure 9.

We can verify the merged model against the same properties as before, although we need to translate them according to the vocabulary changes we made in the merge. χchek returns T, F, F and T for properties (1), (2), (3) and (4), respectively. The values of F are interesting, because they indicate feature interaction. Properties (2) and (3) hold in the **caller** χview but not in the **callee1** χview. We would

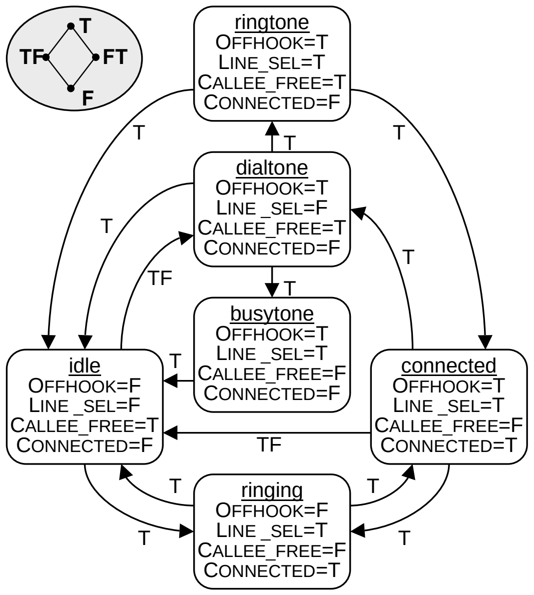


Figure 9. Caller merged with the combined callee χview.

therefore expect them to take the value TF in the merged model. The fact that they take the value F indicates that we have 'broken' **caller** by adding **callee1** to it—some of its properties are no longer TRUE when the features interact.

We can also verify new properties that are not expressible in either **caller** or **callee1** individually. For example: "If you make a call, the phone cannot ring without returning to idle first":

$$AG((\text{ringtone} \vee \text{busytone}) \rightarrow A[\neg\text{ringing}\, W\, \text{idle}]) \qquad (5)$$

χchek returns F for this property. This confirms that we have found an undesirable feature interaction. It looks likely that we could resolve the problem by keeping the two connected states separate in the merged model, to distinguish between being connected as a **caller** and connected as a **callee**. The χbel framework supports exploration of such possibilities by allowing us to construct such alternative merges and to verify whether the desired properties hold for them.

7 Conclusions

The ability to reason about inconsistent and incomplete models is often needed when modeling requirements. However, this cannot be done effectively using classical logic. Multi-valued logics allow such reasoning, although no single multi-valued logic is sufficient to represent the many different ways in which multiple descriptions of the system may be inter-related. Any framework for combining inconsistent models of a system to reason about them needs to be flexible enough to support different types of merge and different forms of analysis.

In this paper we described the χbel framework for reasoning about a set of inconsistent state machines using multi-valued logic. The framework includes support for specifying appropriate multi-valued logics as quasi-boolean lattices, vocabulary maps for explicitly unifying the namespaces of the individual descriptions, and value maps for mapping tuples of values of individual lattices onto values of a combined lattice. System properties can then be specified in χCTL and analyzed using our multi-valued model-checker χchek.

The framework provides flexibility in the choice of the logic and the type of the merge, and thus can serve as an effective exploration tool for reasoning about different combinations of information from multiple viewpoints. Although the framework is very flexible, we have found that most of the interesting merges of viewpoints fall into one of a small number of categories, and we have defined merge templates to provide guidance in using these merge types.

In order to make the framework truly useful, we plan to continue our work in defining the library of templates and proving properties about them. We also plan to explore the implications of different approaches for treating undefined values during merges, and believe that viewpoint invariants may provide a powerful means of expressing these. Further, we plan to explore the issues related to merging models at different levels of abstraction.

Further work on the model-checker is also required. At present, it only handles synchronous parallelism. We are planning to extend it to reasoning about asynchronous parallelism and real-time.

Acknowledgements

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

References

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

[2] R. Balzer. “Tolerating Inconsistency”. In *Proc. 13th Int. Conf. on Software Engineering (ICSE-13)*, pages 158–165, Austin, Texas, USA, 1991. IEEE CS Press.

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

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

[5] M. Chechik, S. Easterbrook, and V. Petrovykh. “Model-Checking Over Multi-Valued Logics”. In *Proc. Formal Methods Europe (FME’01)*, March 2001.

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

[7] G. Cugola, E. D. Nitto, A. Fuggetta, and C. Ghezzi. “A Framework for Formalizing Inconsistencies and Deviations in Human-Centered Systems”. *ACM Trans. on Software Engineering and Methodology*, 5(3):191–230, July 1996.

[8] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. “Patterns in Property Specifications for Finite-State Verification”. In *Proc. 21st Int. Conf. on Software Engineering (ICSE-21)*, Los Angeles, May 1999.

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

[10] A. C. W. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. “Inconsistency Handling in Multi-Perspective Specifications”. *IEEE Trans. on Software Engineering*, 20(8):569–578, August 1994.

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

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

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

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

[15] T. Menzies, S. Easterbrook, B. Nuseibeh, and S. Waugh. “An Empirical Investigation of Multiple Viewpoint Reasoning in Requirements Engineering”. In *Proc. 4th Int. Symp. on Requirements Engineering (RE’99)*, Limerick, Ireland, June 7-11 1999. IEEE CS Press.

[16] K. Narayanaswamy and N. Goldman. “*Lazy* Consistency: A Basis for Cooperative Software Development”. In *Proc. 4th Int. Conf. on Computer Supported Cooperative Work (CSCW’92)*, pages 257–264, Toronto, Canada, 1992.

[17] G. Priest and K. Tanaka. “Paraconsistent Logic”. In *The Stanford Encyclopedia of Philosophy*. Stanford University, 1996.

[18] W. Robinson and S. Pawlowski. “Managing Requirements Inconsistency with Development Goal Monitors”. *IEEE Trans. on Software Engineering*, 25(6):816–835, 1999.

[19] R. W. Schwanke and G. E. Kaiser. “Living With Inconsistency in Large Systems”. In *Proc. Int. Workshop on Software Version and Configuration Control*, pages 98–118, Grassau, Germany, January 27-29 1988. B. G. Teubner, Stuttgart.

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

[21] P. Zave. “Feature Interactions and Formal Specifications in Telecommunications”. *IEEE Computer*, 26(8):20–30, August 1993.