Handling Inconsistencies in CTL Model-Checking using Belief Revision

Thiago Sousa and Renata Wassermann

[1]Department of Computer Science
Institute of Mathematics and Statistics
University of São Paulo, Brazil

{thiago,renata}@ime.usp.br

Abstract. *Given a formal description of a system, there are computational tools that verify the validity of some properties in the system. These tools usually only tell the user whether the property holds or not, without giving any hint as to how the system could be adapted in order for the desired property to hold.*
In this paper, we present a formal approach for handling inconsistencies in CTL (computation tree logic) model-checking using belief revision. Given a CTL formula inconsistent with a model of the system described in SMV, we revise this model in such a way that the formula becomes true. This revision forces changes in the SMV description of the system. Our implementation enriches the NuSMV model checker with three types of change: addition of lines, elimination of lines and change in the initial state, where the first two cause modifications in the transitions between the states of the model.

1. Introduction

Handling inconsistencies in requirements specifications is a critical activity in the software development process. Inconsistent specifications can lead to system failures, and defects detected late in development can be more expensive to correct than inconsistencies discovered early. Therefore, techniques for the detection and resolution of inconsistencies in requirements specifications can be crucial for the successful development of software systems.

A variety of techniques has been developed for checking specifications for inconsistencies. These include formal techniques such as those based on model checking or theorem proving [Winter 1997, Büessow 2003, Leuschel and Butler 2003, Kolyang et al. 1996, E.M. Clarke et al. 1994]. While many of these approaches provide rigorous, and often automated, analysis of software specifications to reveal inconsistencies, they often also do not support the system developer in solving these inconsistencies after they have been discovered.

To address this issue, we have developed an approach based on belief revision to suggest ways for changing system specifications. Belief revision [Gärdenfors 1988, Hansson 1997] is a sub-area of artificial intelligence whose main focus is to keep the consistency of a set of beliefs when new beliefs are incorporated.

This paper presents an approach based on belief revision for handling inconsistencies focusing in a particular model-checking tool, NuSMV [Cimatti et al. 2002]. The basic principle of model-checking is to analyse whether a model that represents the system

(described in some language) satisfies some property (described in some temporal logic). NuSMV focusses in verifying the validity of Computation Tree Logic (CTL) formulas [Clark and Emerson 1981] in systems described in SMV [McMillan 1992].

The paper describes and tests our approach of using belief revision techniques to suggest changes in a given system description (in SMV) in such way that a desired CTL formula becomes true. In this paper, we only consider simple CTL formulas (without any internal temporal operators) and SMV description with some limitations: only boolean and scalar variables; deterministic transitions; only one initial state; only one system process. An overview of our approach is shown schematically in figure 1: if the CTL formula is valid in the system, OK; otherwise, NuSMV provides a counter-example and our extension provides suggestions of changes in the system.

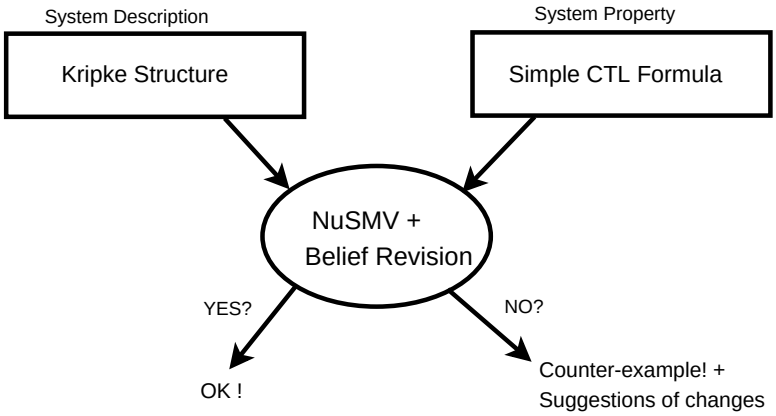

Figure 1. Model-checking with belief revision

Section 2 of the paper quickly reviews belief revision theory, focusing on techniques of comparison of models. Section 3 reviews CTL model-checking and the SMV language of NuSMV. The next section provides a case study to motivate and illustrate our approach. The example shows how to map CTL model-checking onto a belief revision problem, presenting the kind of changes that our technique is able to suggest. The way we have extended the NuSMV tool to support our implementation is also described. We conclude with a discussion of lessons learned from our case study, and a summary of related and future work.

2. Belief Revision

The necessity to model the behaviour of dynamic knowledge bases formed the basis of belief revision theory [Gärdenfors 1988, Hansson 1997]. Most of the literature in the area is based on the seminal work of Alchourrón, Gärdenfors and Makinson [Alchourron et al. 1985], that have proposed some postulates that describe the formal properties that a revision process should obey. They have also proposed some constructions that satisfy the postulates. The theory became known as the AGM paradigm, due to the initial of the authors.

In AGM theory, the beliefs of an agent are represented by a belief set, a set of formulas closed under logical consequence ($K = Cn(K)$, where Cn is a supraclassical consequence operator). There are three types of change operators for a belief set (K). In *expansion* ($K + \alpha$), a consistent information α, together with its logical consequences, is

added to the belief set K. In *contraction* ($K - \alpha$), the information α is abandoned. Since the set K is logically closed, it could be necessary to abandon other beliefs that would imply α. In *revision* ($K * \alpha$), an information α is added to K and to keep consistency it can be necessary to abandon other beliefs of K.

Besides belief sets, one can use the idea of possible worlds in order to represent the beliefs of the agent. Possible worlds can be thought of as possible states of the world (or, in propositional logics, propositional valuations). Given a belief set K, $[K]$ is the set of the possible worlds where all formulas of K are true. And for a set W_k of possible worlds we can define a corresponding belief set K as the set of formulas that are true in all the worlds of W_k.

Grove has proposed a semantic for belief revision, based on the idea of having "spheres" of possible worlds around the set of possible worlds which satisfy the agent's beliefs [Grove 1988]. When giving up a belief α, one should look at the minimal sphere containing a $\neg\alpha$ world. This is the region of the worlds which are the closest to the ones originally held possible by the agent, and where α is not satisfied.

Let W be the set of all possible worlds. A system of spheres centred in $[K]$ is a collection X of sets that satisfies the following conditions: X is totally ordered; $[K]$ is the least element of this ordering; W is the largest element of the ordering; if any sphere intersects a sentence $[\alpha]$, then there exists a minimal sphere in X that intercepts $[\alpha]$.

The revision of K by a given sentence α in this case is the intersection of $[\alpha]$ with the minimal sphere of X that intersects $[\alpha]$. Contraction of K by α is given by the union of $[K]$ with the revision of K by $\neg\alpha$. This construction, based on the comparison of models (possible worlds), was used as a basis for our implementation.

2.1. Example

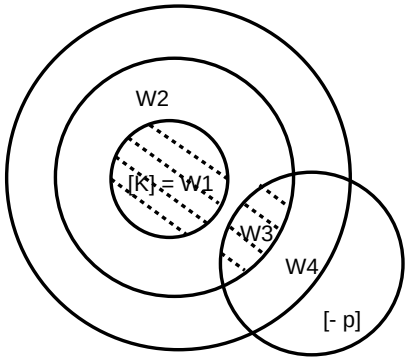

Figure 2. Contraction in a system of spheres

Assume that we are dealing with propositional logic, and let us have K = {p, q} and $\alpha = p$. The set of possible worlds W consists of the possible values that the two propositional variables can take. Thus, we have four possible worlds: W_1, where p and q are both true, W_2, where p is true and q is false, W_3, where p is false and q is true and W_4, where p and q are both false. We have that $[K] = \{W_1\}$. It is necessary to establish an order for the rest of the worlds in order to have a system of spheres centred in $[K]$. In the second level we can put worlds where only one propositional variable differs from W_1 (W_2 and W_3) and in the third level, W_4, which differs from W_1 in two variables.

As the contraction is the union of $[K]$ with the intersection of $[\neg\alpha]$ with the least sphere that intersects it, we have that the contraction $[K - p]$ is given by $\{\{W_1\} \cup \{W_3\})$, which represent the set $\{q, p \vee q, p \rightarrow q\}$ (see Figure 2).

3. CTL Model-Checking

The basic principle of model-checking is: given a property of the system, described in a temporal logic, determine whether the finite state machine that represents the described system satisfies such property. In other words, verify whether a formula f is true in a graph G of states.

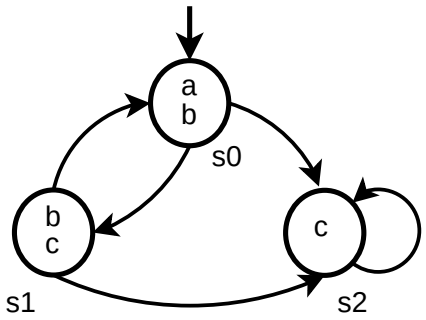

Figure 3. Kripke Structure

The finite state machine that represents the system can be described as a specific Kripke structure, that is defined as: a set S of states, a set R of transitions between states (where each state must have a successor), a set I of initial states and a function L that associates to each state a set of propositions that hold in that state. To model a state in *deadlock* (state without successors) it suffices to create a transition from the state to itself. Figure 3 represents a Kripke structure where $P = \{a, b, c\}$, $S = \{s0, s1, s2\}$, $R = \{(s0, s1), (s0, s2), (s1, s0), (s1, s2), (s2, s2)\}$, $I = \{s0\}$ with $L(s0) = \{a, b\}$, $L(s1) = \{b, c\}$ and $L(s2) = \{c\}$.

3.1. Computation Tree Logic

The properties of the system are described in temporal logic. Temporal logic is a type of modal logic where it is possible to represent and to reason about propositions related to time. Through temporal logic it is possible to express sentences of the type "I am *ALWAYS* hungry" or "I will be hungry *UNTIL* I eat in a all-you-can-eat restaurant".

Clarke and Emerson [Clark and Emerson 1981] proposed CTL (Computation Tree Logic), a logic capable to consider different possible futures, through the notion of branching time. The idea of this logic is to quantify over the possible runs of a program through the notion of paths that exist in the space of states of the system. The properties can be evaluated with respect to all the runs or some run. This logic is used in some model-checkers, such as NuSMV, which we have used in our implementation. The syntax of CTL is given by the following definition:

$$\phi ::= p|\neg\phi|\phi \wedge \phi|(AX\phi)|(AG\phi)|(AF\phi)|(EX\phi)|(EG\phi)|(EF\phi)|E(\phi\, U\phi)|A(\phi\, U\phi)$$

where p is a propositional atom, $\neg$, $\wedge$ are the usual logical connectives and the other are temporal operators. Each temporal operator is composed of a path quantifier (E, "there exists a path", or A, "for all paths") followed by a state operator (X, next state in the path, U, until, G, globally, or F, finally). CTL has the following semantic definition:

Definition 3.1 *Let M be a Kripke structure and $\pi(i)$ the i-th state of a path. We say that $M, s \models \phi$ if and only if ϕ is true in the state s of M. Thus we have:*

1. $M, s \models p$ *iff* $p \in L(s_0)$

2. $M, s \models \neg\phi$ *iff* $M, s \not\models \phi$
3. $M, s \models \phi_1 \wedge \phi_2$ *iff* $M, s \models \phi_1$ *e* $M, s \models \phi_2$
4. $M, s \models$ *EX* ϕ *iff there is a state* s' *of* M *such that* $(s, s') \in R$ *and* $M, s' \models \phi$
5. $M, s \models$ *EG* ϕ *iff there is a path* π *of* M *such that* $\pi(1) = s$ *and* $\forall i \geq 1 \bullet M$,$\pi(i) \models \phi$
6. $M, s \models$ *E*(ϕ_1 *U* ϕ_2) *iff there is a path* π *of* M *such that* $\pi(1) = s$ *and* $\exists i \geq 1 \bullet (M, \pi(i) \models \phi_2 \wedge \forall j, i > j \geq 1 \bullet M$,$\pi(j) \models \phi_1$)

The other temporal operators can be derived from EX, EG and EU:

AX ϕ = $\neg$ EX $\neg\phi$

AG ϕ = $\neg$ EF $\neg\phi$

AF ϕ = $\neg$ EG $\neg\phi$

EF ϕ = E [true U ϕ]

A[ϕ U β] = $\neg$ E[$\neg\beta$ U $\neg\phi \wedge \neg\beta$] $\wedge \neg$ EG $\neg\beta$

We say that a Kripke structure M satisfies a formula CTL ϕ if $M, s_0 \models \phi$, where s_0 is an initial state.

The main approach for automatic formula verification in CTL is based on the fixed point theory to characterize its operators. For details the reader is referred to [Huth and Ryan 2000].

3.2. NuSMV

The NuSMV language was projected to allow the description of synchronous and asynchronous systems. This language, based on its predecessor SMV, also allows modularization of components, thus facilitating the maintenance. NuSMV uses propositional calculus to describe the transitions of a finite Kripke structure, supplying much flexibility, but at the same time being able to introduce inconsistencies. As NuSMV describes finite state machines, its language only allows types of finite data, as boolean, scalars and vectors of basic data type.

Its first version was developed in the end of the decade of 90 jointly for the italian center of research ITC-IRST and the University of Carnegie-Mellon as a reimplementation of the SMV model-checker created by McMillan during his PhD. The NuSMV has free code (in language C, with license LGPL 2.1) and BDDs representation (thus preventing the explosion of states), besides being used in real projects and having a sufficiently active community.

The main syntatical constructions of NuSMV are: Simple Expressions, Case Expressions, Set Expressions, Next Expressions and CTL Expressions. With Simple Expressions it is possible to describe boolean and numeric variables, as well as logical and numerical operators in the system. The Case Expressions generally are used to relate preconditions with post-conditions. The Set Expressions are used to describe the set theory, as set definition, test of inclusion and union of sets. The Next Expressions, in turn, are used to express the transitions related to one given variable in the finite state machine, being frequently used together with Case Expressions. CTL Expressions are used to define

the property to be verified. We present a fictitious system in order to illustrate the syntax of NuSMV:

```
MODULE main
VAR
state1: {n1, t1}; state2: {n2, t2}; turn: {1, 2};
ASSIGN
init(state1) := n1; init(state2) := n2; init(turn) := 1;
...
next(turn) := case
  (state1 = n1) & (state2 = t2): 2;
  (state2 = n2) & (state1 = t1): 1;
  1: turn;
esac;
...
SPEC
EF((state1 = n1) & (state2 = n2))
```

The space of states of the finite state machine is determined by the declaration of the state variables (in the example above, `state1`, `state2` and `turn`). The variable state1 is declared as being of the type scalar, assuming symbolically the values `n1` and `t1`. The same happens with the variable `state2` and `turn`, that assume the values `n2`, `t2` and 1,2, respectively.

With the declaration `ASSIGN` it is possible to define attributions to variables. The first attribution relates to the initial value of variables (`init`). In the example above the variable `state1` will start with value `n1` and the variable `state2` with `n2`. The variable `turn` initiates with value 1. The attributions defined for the declaration `next` express the transition relations of variables. In the example, we have that, in the next state, the value of the variable `turn` will be 2 if in the current state, the expression `(state1 = n1) & (state2 = t2)` is true. If, on the other side, the expression `(state2 = n2) & (state1 = t1)` is true in the current state, the next value of the variable `turn` will be 1. If none of the two expressions is true, then the variable `turn` continues with the same value in the next state. In this example it is possible to see clearly how the Case Expressions act together with Next Expressions and are used to relate pre-conditions with post-conditions (`if (state1 = n1) & (state2 = t2)` [pre-condition], then `turn: = 2` [post-condition]).

With the SPEC declaration it is possible for the user to define which temporal property should be verified in the system. In the example above, it should be verified whether there exists a path such that in one of its future states, the expression `(state1 = n1) & (state2 = n2)` is true. NuSMV verifies properties in CTL using a fixed point approach.

4. Revising CTL model-checking

We have already briefly introduced the basic concepts of CTL model-checking and belief revision theory. In this section we show our proposal of applying belief revision to model-checking in order to revise system descriptions.

We first go through an example that motivates the proposal.

4.1. Case Study

Suppose that a software engineer desires to model a light control system based on the adaptation of specifications made in [Rodrigues et al. 2004]: - the system has 3 intensities of light: standard (d), chosen by the user (c) and no light (n); - the system also controls the time that the building is vacant and the time that the alarm is ringing; - if sunlight is very intense, then the light is suspended. If the external light is reasonable for the environment and the light was chosen by the user, the standard light is established; - whenever a user enters in the building, the standard light is established. However, if a user enters the building before a minimum time for reoccupation, the light is established in accordance with the choice of the previous user. If the minimum reoccupation times expires, the light is suspended; - when the alarm is ringing and a determined time expires, the light is suspended. Otherwise, the standard light is established.

In accordance with these requirements the software engineer decided to model the system in the following way: - a variable to represent the 3 types of light; - three clock variables ("timeun" to count the unoccupation time, "timeal" to count how long the alarm is ringing and "timer" that it will be used to simulate events); - two variables for events simulation ("user" to detect the presence of people and "alarm" to detect that there is fire); - an input variable ("lux" to verify the external luminosity). This modeling could generate up to 1344 states. But the pre-condition restrictions reduce it to 16 states.

A possible modeling of this system described in the NuSMV language is shown in Table 1.

Suppose that the engineer desires the reoccupation property to hold in all states of the system and therefore he modelled it as an invariant. This invariant can be described in the NuSMV language in the following way: `SPEC AG (((timeun < 3 & to user) - > light = c) & timeun = 3 - > light = n))`

The output verification provided by NuSMV model-checking is a path from the initial state to the inconsistent state. If we run NuSMV in this example, it will prove that the property is not true, showing a state (initial) where the reoccupation time expires and the light is not suspended. If we have a lot of variables in the state and/or the temporal formula is more complex, it will be difficult to identify where the problem lies. Moreover, only one inconsistent state is shown. For example, the modification of our initial state to {light = n, timeun = 3, timeal = 0, to timer = 0, to user = 1, alarm = 0 } is not enough to make the temporal formula true. In our point of view, it would be interesting to have a tool that revises all the inconsistent states with the temporal formula presenting suggestions to modify the system description removing/adding transitions.

4.2. Applying Belief Revision

The first step when modelling a belief revision operation is to define the representation of the belief states. As we are dealing with finite state machines, each state has complete information of the world, that is, each state can be compared with a complete model of the system variables. Thus, a state {light = n, timeun = 3, timeal = 3, timer = 1, user = 1, alarm = 0 } of our example is a possible model of the set {light, timeun, timeal, timer, user, alarm }. Therefore, we will adopt the representation based on models (states, possible worlds). From this point on, whenever we mention a state we are referring to a complete model of the system variables.

```
MODULE main

VAR
light: {n,d,c}; -- state var
timeun : 0..3; -- timer var
timeal : 0..3;
timer : 0..6;
user : boolean; -- event var
alarm : boolean;

IVAR
lux : {1,2}; -- input var

ASSIGN
init(light) := d;
init(timeun) := 3;
init(timeal) := 0;
init(timer) := 0;
init(user) := 1;
init(alarm) := 0;

next(light) := case
 lux = 2 : n;
 lux = 1 & light = c : d;
 timeun < 3 & user : c;
 user : d;
 timeun = 3 : n;
 timeal < 3 & alarm : d;
 timeal = 3 & alarm : n;
 1 : light;
esac;
```

```
next(timeun) := case
 user : 0;
 timeun = 3 : 3;
 !user : timeun + 1;
 1 : timeun;
esac;

next(timeal) := case
 !alarm : 0;
 timeal = 3 : 3;
 alarm : timeal + 1;
 1 : timeal;
esac;

next(timer) := case
 timer = 6 : 0;
 timer < 6 : timer + 1;
 1 : timer;
esac;

next(user) := case
 timer < 2: 1;
 timer <= 6 : 0;
 1 : user;
esac;

next(alarm) := case
 timer < 1 : 0;
 timer <= 6 : 1;
 1 : alarm;
esac;
```

Table 1. SMV formalization of the light control system

One of basic ideas of belief revision is the Principle of Minimal Change, that states that we should keep as much information as possible from the original set of belief. But how do we define a minimal change? Rodrigues [Rodrigues 1997] defined a function *d* that measures quantitatively how close a model (state) is from another one counting the number of propositional variables that have different truth values in the models (states). Our idea of minimal change will be based on this function, but we will not restrict it to propositional variables. Any difference of values between two variables will influence the calculation. Our distance distance *d* is defined as:

Function *d* Let A and B be states. The distance between A and B is the number of variables p_i such that $A(p_i) \neq B(p_i)$, where $A(p_i)$ is the value of the variable p_i in the state A and $B(p_i)$ is the value of the variable p_i in the state B.

For example, if A = {light = n, timeun = 3, timeal = 3, timer = 1, user = 1,alarm = 0} and B = {light = d, timeun = 0, timeal = 0, timer = 2, user = 1, alarm = 1}, then $d(A, B) = 5$.

We extend the function *d* to calculate the least distance between a state and a set of states:

Function *D* Let C be a state, Θ be set of states and *min* a function that calculates the minimum value of a set of natural numbers. The distance between C and Θ is defined as:

$$D(C, \Theta) = \begin{cases} min\{d(C, E) \text{ — } E \in \Theta\} & \text{if } \Theta \text{ is not empty} \\ \infty & \text{otherwise} \end{cases}$$

We have already defined that the belief representation will be based on possible worlds. To complete the revision structure, we need to define a system of spheres, i.e., the central sphere and the ordering among the possible worlds.

The system of spheres will be centred in a state, that is, it will be centred in a single possible world. The order of the spheres will be based on the value of the function *d* described previously. In the second level they are the worlds that have distance 1 with respect to the central model. In the third level are the worlds with distance 2, in the 4th with distance 3, and so on.

As our knowledge base has a single model, we only need to compare the base model with the models of the input sentence.

But how do we guarantee that the input sentence has at least one model? It is necessary to guarantee that the sentence is consistent and that it is described in some logic that does not suffer interferences with the proposed changes. This excludes temporal logic. On the other hand, propositional logic is invariant with respect to changes in the system space states. Thus, if the input sentence is a consistent propositional formula, then we guarantee that it will have at least one model. In the NuSMV any Simple Expression of its syntax can be adapted to propositional logic. Therefore, the input belief will be a Simple Expression whose adaptation for this logic produces a consistent formula.

Thus, again based in [Rodrigues 1997], our revision can be defined as:

Revision Let F be a model that represents the original beliefs and Ψ a set of models from an input sentence α. The revision $F * \alpha$ will be a set Δ such that:

$$\Delta = \{G \in \Psi \text{ — } d(F, G) = D(F, \Psi)\}$$

If we choose a model of our example, F = {light = d, timeun = 0, timeal = 0, timer = 2, user = 1, alarm = 1}, and revise it with a sentence α = {((timeun $<$ 3) & user $\rightarrow$ light = c) & (timeun = 3 $\rightarrow$ light = n) }, we get:

Δ = {{light = c, timeun = 0, timeal = 0, timer = 2, user = 1, alarm = 1},
{light = d, timeun = 0, timeal = 0, timer = 2, user = 0, alarm = 1}}

The sentence α has 784 models, but only two have the minimum distance (1) from the state represented by F.

4.3. Changes in SMV Description

We have already compared states to models and we have shown how to make the revision based on comparison of models. The next stage is to define the changes in the system description to reach the revised state.

As the system is represented as a graph of state transitions, it is enough to concentrate the efforts in modifying the transitions between a state and its inconsistent successors. We will not modify the temporal property to be verified, nor the amount and or the type of variables.

We have seen previously that the transitions are defined in NuSMV through Next Expressions together Case Expressions. Thus, from now on we will consider as a transition each line (pre-condition: post-condition) of a Case Expression. The idea is to add/remove transitions in the system description in such a way that the revised state is reached from its predecessor.

Figure 4 represents a transition from our example that leads to state inconsistent with the formula α = {((timeun $<$ 3) & user $\rightarrow$ light = c) & (timeun = 3 $\rightarrow$ light = n)}. We also see a possible revised state.

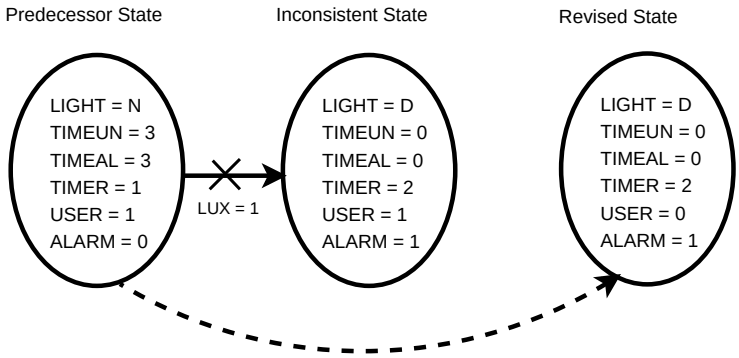

Figure 4. The goal is to force the dotted transition and avoid the other one

To discover which Next Expressions we must modify, we only need to compare the original inconsistent state with the revised state and find the variables that have different values. In figure 4 there is only one variable: user.

To reach the revised state from the predecessor we have two options: adding or eliminating transitions. To define a transition to be added it is necessary to guarantee that it is applied only in the state predecessor in order to prevent interferences in other transitions of the system. The only way to guarantee this is that such transition has as pre-condition the conjunction of the common variables and the input variables of the predecessor state. The post-condition is the value of the variable in the revised state.

In the example of figure 4, we would add the transition “light = n & timeun = 3 & timeal = 3 & timer = 1 & user = 1 & alarm = 0 & (lux = 1) : 0;” in the Case Expression that is inside of the Next Expression of variable “user”. This is a modification transition because it is changing the value of variable in the predecessor state for the desired one. It is necessary to define in which “position” of the Case Expressions we will add the transitions. As pre-conditions are analysed from top to bottom and we desire that these

transitions are obligatorily accomplished, we will add them at the beginning of the Case Expressions.

Another option is to eliminate transitions. First it is necessary to verify from top to bottom if there are some transition in the Case Expression of the variable that modifies or keeps its value in accordance with the revised state. If this transition exists, then we can eliminate the other transitions (above it in the Case Expression) that can be accomplished in the predecessor state. In our example, we can simply eliminate the transition "timer $<$ 2: 1;" in the Case Expression of variable "user".

We should note that it is not always possible to reach the revised state eliminating transitions, because sometimes the description does not have a transition that modifies the variable to the value in the revised state. In our example, if we did not have the transition "timer $<=$ 6: 0;" we would never reach a state where "user = 0" using only the option to eliminate transitions.

A good point to observe is that this approach is possible because the transitions depend only on the values of the variable and there are no duplicate states in the model of the system.

Thus, we can have two possible variations for the new description of the Next Expression of the variable user in such a way that the predecessor state reach the new revised state instead of the original inconsistent state.

```
Option 1 :   /* Add a transition */
next(user) := case
  light = n & timeun = 3 & timeal = 3 & timer = 1
  & user = 1 & alarm = 0 & (lux = 1) : 0;
  timer < 2 : 1;
  timer <= 6 : 0;
  1 : user;
esac;
Option 2 :   /* Delete a transition */
next(user) := case
  timer <= 6 : 0;
  1 : user;
esac;
```

But there is a problem to deal with when eliminating a transition. When we are adding, we do not modify the system space states previously generated, what guarantees that the revised state is reached. When we eliminate a transition, we can modify the space of states previously generated and analysed, without any guarantee to reach the revised state. To solve this problem it is necessary to store the transitions that have already been used in the space of states generated so far. If a given transition was never used, then we can eliminate it. Thus, we guarantee that such elimination does not affect the predecessors states and the revised state will be reached. Otherwise, we cannot eliminate it.

There are still some cases where we will need to modify the initial state. As it has no predecessor state, modifying the "init" declarations with the variable values of the new revised state is enough.

4.4. Algorithm Sketch

The next step is to define an algorithm that finds all the changes that have to be done in the system description such that the temporal formula becomes true.

Given a finite state machine and a temporal formula $T\alpha$, where T is one of the eight temporal operators and α a consistent Simple Expression, the main idea is to consider each state as the original belief set and α as the input belief. As NuSMV begins the verification of a temporal formula with the initial state, our algorithm initiates considering the initial state as the current belief set.

After determining the top-level temporal operator, the specific algorithm for this operator is traversing the graph of the system model from the initial state, using depth-first search. As soon as the algorithm finds an inconsistency with the temporal formula, a state is revised. Amongst the revision options, we choose one of them randomly. First, we try to remove transitions to reach this revised state. When it is not possible, transitions are added in the description of the system to guarantee that this new state is reached from its state predecessor.

The halting process of the algorithm varies depending of the type of temporal operator we are dealing with. Only one criterion is used: once a path is chosen, we delay as long as we can the revision process, guaranteeing, for this path, the largest number of identical states in relation to the original model. When the algorithm finishes we have all the transitions that must be removed and/or be added in the system description, as well as whether it is necessary to modify the initial state, so that the temporal formula becomes true.

It is important to emphasise that the modification of the initial state will only occur when strictly necessary, that is, only with operators AG, EG, AU and EU.

In the algorithm, there is also an intensive use of a global stack **P** and a adjacencies lists **AdjList** that store for each state its successive states. This list and this stack are important because the algorithm uses depth-first search. The stack is manipulated with the usual functions: **Push**, **Top** and **Pop**. The function **Remove** removes randomly a state of a given adjacencies list. The function **Recreate_AdjList** remakes the adjacencies lists and is useful because in the revision process there are some changes in the transitions of states. The function **Used_Trans** stores the used transitions. The function **Inconsistent** verifies if a model (state) is consistent or not with a formula. Due to the space limit, here we only outline the main idea of implementing function Revise_AG.

Note that this algorithm (as well as the others) represents only one of the possibilities of revising so that the temporal formula becomes true. If the formula is E [α U β], AF β or A[α U β], it would simply be enough to revise the initial state with some consistent state where β holds to make the formula become true. But as the initial state is so clear for the user, it does not seem reasonable suppose that the user made a mistake. Another possibility would be revise the initial state with α and its successors with β. But, the closer a state is from the initial state, more certain the user is of its validity. Therefore, when a path is chosen, we try to postpone the revision process as much as possible.

FUNCTION Revise_AG
Input: Initial state and formula F
Output: Changes to make the formula AG F true

```
 1: Initial ⇐ Revise_Initial(Inicial,F);
 2: Recreate_AdjList();
 3: Push(Initial,P);
 4: while P != ∅ do
 5:    S ⇐ Top(P);
 6:    if AdjList(S) != Nil then
 7:       W ⇐ Remove(AdjList(S));
 8:       if Inconsistent(W,F) then
 9:          W ⇐ Revise(S,W,F);
10:          Recreate_AdjList();
11:       end if
12:       if W ∉ P then
13:          Push(W,P);
14:       end if
15:       Used_Trans(S,W);
16:    else
17:       Pop(P);
18:    end if
19: end while
20: return ;
```

4.5. Results

We have implemented our proposal on top of NuSMV and tested it on two SMV examples, each one with eight CTL formulas (one for each operator) and have obtained the expected results. Following the example in section 4.1, we have the following output where NuSMV would present a counter-example:

```
-- but this specification can be true if you make the following
changes:

1) Change your initial state to (light = n) & (timeun = 3) & (timeal
= 0) & (timer = 0) & (user = 1) & (alarm = 0)

2) Add the transition (light = n) & (timeun = 3) & (timeal = 0) &
(timer = 0) & (user = 1) & (alarm = 0) & (((lux = 2))) : c; into
case expression's first position of variable light

3) Remove the transition lux = 2 : n; into case expression's of
variable light

4) Add the transition (light = n) & (timeun = 3) & (timeal = 3) &
(timer = 1) & (user = 1) & (alarm = 0) & (((lux = 1)) | ((lux = 2)))
: c; into case expression's first position of variable light

5) Add the transition (light = c) & (timeun = 0) & (timeal = 0) &
(timer = 1) & (user = 1) & (alarm = 0) & (((lux = 1))) : c; into
case expression's first position of variable light
```

```
6) Add the transition (light = n) & (timeun = 3) & (timeal = 0) &
(timer = 0) & (user = 1) & (alarm = 0) & (((lux = 1))) : c; into
case expression's first position of variable light
```

The change of the initial state and the removal of transitions have little impact on the legibility of the code. The main problem occurs when there is additionLauracampos.html of transitions. If a system has lot of variables, the pre-condition (formed as a conjunction of these variables) can become very long, making the new code unreadable. But, most of the time, this pre-condition can be reduced because some values of the variable occur only in one state. Moreover, we can join transitions (lines) in the code, improving its legibility.

If we manually reduce the new code proposed by the algorithm, cutting these redundancies, we can get the following reasonable revised code:

```
...
ASSIGN
init(light) := n; init(timeun) := 3; init(timeal) := 0;
init(timer) := 0; init(user) := 1; init(alarm) := 0;
next(light) := case
  (light = n) & (timeal = 0) & (((lux = 1)) |
  ((lux = 2))) : c;
  (light = c) & (timer = 1) & (((lux = 1))) : c;
  (light = n) & (timer = 1) & (((lux = 1)) |
  ((lux = 2))) : c;
  lux = 1 & light = c : d;
  timeun < 3 & user : c;
  user : d;
  timeun = 3 : n;
  timeal < 3 & alarm : d;
  timeal = 3 & alarm : n;
  1 : light;
esac;
...
```

5. Conclusions

We have shown that through an implementation of belief revision into NuSMV model-checker it is possible to eliminate the inconsistencies of software models. Our algorithm, implemented in C on top of release 2.3.1 of NuSMV is available at `http://www.ime.usp.br/~thiago/mestrado/Br-NuSMV-2.3.1.tgz`. It provides a suggestion of changes that could be made so that a given temporal formula would hold in the system model. We do not claim that the suggestion is the best nor most simple change possible.

The belief revision area has few implemented practical applications. Our contribution in this area increases this number. It seems that this is the first implementation of belief revision in a tool used for formal modelling.

The contribution of our work in the area of formal methods for modelling finite state systems is also practical. We believe that there is no model-checking tool with support for automatic handling of inconsistencies so far.

Related Work In his Phd thesis, Gorogiannis [Gorogiannis 2003] proposed an algorithm that generates an automaton from ACTL formulas (fragment of CTL), transforms

it into NuSMV code and joins it with the original system description. The synchronous composition provided by NuSMV works like a automata product operation, making the formula true. This approach is similar to ours, also considering alterations in the SMV code. The difference is that this approach focusses exclusively in the temporal formula, ignoring completely the system description made by the user. Moreover, the logic ACTL is more restricted, not allowing for example to express that in the next state a variable p is true (EX p). Unfortunately we did not have access to the implementation to make comparative tests. But in the example presented by Gorogiannis with formula AG (p $\rightarrow$ AF q), we can verify that 3 new variables were created, 44 new transitions and 7 new initial states, what makes the new code unreadable.

Ding [Ding and Zhang 2006] considers a model updater, where the variable are updated with a minimum criteria to satisfy a CTL formula. This approach is similar to ours in the direction to consider changes in the model defined by the user. But, besides working only with boolean variables, this approach has not been implemented in any known formal language. Ding developed a prototype with a self-designed language to describe the initial state, transitions between states and the temporal formula. We did not have access to this prototype either, but in the sample case study presented, the transitions do not use the relation pre-condition with post-condition, which is very common in finite state systems.

Future Work The dissemination of the belief revision idea in other products and communities of software development must produce good opportunities. In special, it would be interesting and challenging an implementation of this type on tools for B method and Z language, which are more widely used. Another future work is to extend our implementation for full CTL formulas. Another possible improvement would be the definition of criteria for minimal changes in the model. It would be necessary to define how to measure the changes: counting the number of different states between the models or the number of changes in the description of the system. Our distance function *d* can be improved to calculate the best state, instead of a random choice amongst two or more states with the same distance. Our algorithm is concentrated in the lines changes of the Case Expressions and in the change of the initial state. It would be interesting to analyse the possibility of other types of changes, as changing only the pre-conditions or the post-condition of a given line. The automatic reduction of the revised NuSMV code would be also useful for the process because it would leave the code more legible.

Acknowledgments: The first author was partially supported by CAPES. The second author is partially supported by CNPq grant 304486/2004-3. The work is part of FAPESP project 2004/14107-2.

References

Alchourron, C., Gärdenfors, P., and Makinson, D. (1985). On the logic of theory change: Partial meet contraction and revision functions. *Journal of Symbolic Logic*, 50(2):510–530.

Büessow, R. (2003). *Model Checking Combined Z and Statechart Specifications*. PhD thesis, Technical University of Berlin, Faculty of Computer Science. `http://edocs.tu-berlin.de/diss/2003/buessow_robert.pdf`.

Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tacchella, A. (2002). Nusmv 2: An Opensource Tool for Symbolic Model Checking. In Brinksma, E. and Larsen, K., editors, *Proceedings of Computer Aided Verification (CAV 02)*, LNCS 2404, pages 359–364. Springer-Verlag.

Clark, E. M. and Emerson, E. A. (1981). Design and synthesis of synchronization skeletons using branching time temporal logic. In Kozen, D., editor, *Logics of Programs*, LNCS 131, pages 52–71. Springer-Verlag.

Ding, Y. and Zhang, Y. (2006). CTL Model Update: Semantics, Computations and Implementation. In *Proceedings of the 17th Europen Conference on Artificial Intelligence (ECAI 2006)*, pages 362–366. IOS Press.

E.M. Clarke, O. Grumberg, and K. Hamaguchi (1994). Another look at LTL model checking. In D. Dill, editor, *Proceedings of the 6th International Conference on Computer-Aided Verification CAV*, LNCS 818, pages 415–427. Springer-Verlag.

Gorogiannis, N. (2003). *Computing Minimal Changes of Models of Systems*. PhD thesis, University of Birmingham, School of Computer Science. `http://www.cems.uwe.ac.uk/~ngkorogi/pubs/thesis.ps.gz`.

Grove, A. (1988). Two modellings for theory change. *Journal of Philosophical Logic*, 17(2):157–170.

Gärdenfors, P. (1988). *Knowledge in Flux: Modeling the Dynamics of Epistemic States*. MIT Press.

Hansson, S.-O. (1997). *A Textbook of Belief Dynamics*. Kluwer Academic Publishers.

Huth, M. and Ryan, M. (2000). *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press.

Kolyang, Santen, T., and Wolff, B. (1996). A structure preserving encoding of Z in Isabelle/HOL. In von Wright, J., Grundy, J., and Harrison, J., editors, *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, pages 283–298, Turku, Finland. Springer-Verlag LNCS 1125.

Leuschel, M. and Butler, M. (2003). ProB: A model checker for B. In Araki, K., Gnesi, S., and Mandrioli, D., editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag.

McMillan, K. (1992). *Symbolic Model Checking - an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, School of Computer Science. `http://www.kenmcmil.com/pubs/thesis.pdf`.

Rodrigues, O. (1997). *A Methodology for Iterated Information Change*. PhD thesis, Imperial College, Department of Computing. `http://www.dcs.kcl.ac.uk/staff/odinaldo/pdf/phd_thesis.pdf`.

Rodrigues, O., Garcez, A., and Russo, A. (2004). Reasoning about requirements evolution using cluster belief revision. In Bazzan, A. and Labidi, S., editors, *Proceedings of 17th Brazilian Symposium on Artificial Intelligence (SBIA 2004)*, LNCS 3171, pages 41–51. Spring-Verlag.

Winter, K. (1997). Model checking for abstract state machines. *Journal of Universal Computer Science*, 3(5):689–701.