Simplification of NuSMV Model Checking Counter Examples

Jussi Lahtinen

February 14, 2008

Contents

Abstract

The purpose of this study was to find out a method to simplify counter examples of the NuSMV model checking tool. The method creates a visual presentation of the counter example. It also identifies redundant variables, so that only the essential variables are shown. A running example is discussed.

1 Introduction

Model checking [10] is a relatively new method for system verification. It is automatic and can find design flaws that the more traditional methods of simulation and testing easily miss. Model checking is especially applied to systems where a design failure would result in a catastrophic situation such as hardware systems, traffic control software or medical instruments [10].

A model checking tool describes a possible flaw in the system in form of a counter example. The counter example can be used to find out the error in the system. This work concentrates on the NuSMV model checker [8] and the counter examples it produces.

Debugging of the counter example is left for humans, and often it is hard to pinpoint the actual cause of the counter example. The main problem is the amount of redundant information in the counter example. We introduce a method that simplifies the counter example trace and transforms it into a more readable form.

2 Model Checking

Model checking is an automatic technique for verifying finite state concurrent systems[10]. A simple model checking problem is testing whether a formula in propositional logic is satisfied by a given model. This means that the formula must be true in every initial state of the model.

Model checking is used to verify hardware and software designs. More traditional system verification techniques include simulation, testing and deductive reasoning [10]. Deductive reasoning normally means the use of axioms and proof rules to prove the correctness of systems. Deductive reasoning techniques are often difficult and require a lot of manual intervention. Verification techniques based on extensive testing or simulation can easily miss significant errors when the number of possible states of the circuit or protocol is very large. Model checking on the other hand requires no user supervision and always produces a counter example when the design fails to satisfy some examined property.

The process of model checking consists of modeling, specification and verification. First, the design under investigation has to be converted into a formalism understood by the used model checking tool. This means that the behaviour of the system is depicted in a modeling language. The model should capture

important properties of the system and at the same time abstract away uninteresting details that will only complicate the verification process. [10]

Secondly, the system has some properties it must satisfy. These properties or specifications are usually given in some logical formalism. For hardware and software designs, it is typical to use temporal logic [14], which examines the behaviour of the system over time.

After modeling and specification, only the fully automatic verification part remains. If the design meets the desired properties, the verification tool will state that the specifications were true. In case of a design flaw or an incorrect modeling or specification, a counter example will be presented. A counter example presents a legal execution sequence in the model that is not accepted by some specification. The analysis of the counter example is usually impossible to do automatically and thus involves human assistance. The counter example can help the designer find the errors in the design or in the model.

There are several model checking techniques. Many of them suffer from the state explosion problem [9]. State explosion results from the fact that the number of states in a system grows exponentially as the size of the model increases. Although the system is still finite, the verifying might be too complex for even state of the art computers. No total solution to this problem has yet been found, although symbolic representation of the state space using BDDs or reducing the needed state space using abstraction have been found useful [9]. Partial order reduction [12] is also a typical state space reduction method. Nevertheless, model checking is likely to prove an invaluable tool to verify system requirements or design.

2.1 Modeling of Reactive Systems

Often, and especially in our case, the systems to be modeled are such that they interact with their environment frequently and usually will not terminate. Such systems are referred to as reactive systems [5]. Reactive systems can not be adequately modeled only by their input-output behaviour. We also need to be able to model what happens inside such systems.

In order to examine reactive systems we need to capture the state of the system. A state represents a description of the system that captures every variable value at a specific time instant.

Transitions depict the possible ways the system can change its state. A transition is a pair of states. The initial state corresponds to the system configuration before an action, the second state of the pair corresponds to the state after the action. Only some transitions between states are usually allowed.

A computation is a sequence of states. If the model of a reactive system is thought as a directed graph with states as vertices and transitions as edges, a computation would be a path in the graph. These kind of graphs are usually called Kripke structures [10]. Kripke structures are simple, but they can model reactive systems quite effectively. They are expressive when it comes to some

temporal issues that are essential in reactive systems. Kripke structure is the standard representation of models in the model checking literature[4]. It captures adequately the system semantics.

A Kripke structure is defined as a quadruple $M = (S, R, S_0, L)$ where S is the set of states, $S_0 \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is the transition relation and $L : S \to P(A)$ is the labeling function, where A is the set of atomic propositions, and $P(A)$ is the powerset over A. The labeling function describes the relations between the states and the atomic proposition values. Atomic propositions are used to express which variable values are true in the states of the structure. For a state $s \in S$ the set $L(s)$ is the set of atomic propositions that hold in s.

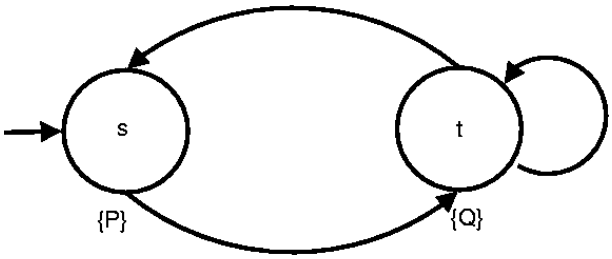

Figure 1: A simple Kripke structure. $S = \{s, t\}, S_0 = s, R = \{(s, t), (t, s), (t, t)\}, L(s) = \{P\}, L(t) = \{Q\}$

Figure 1 shows a graphical representation of a typical Kripke structure. It has two states: s and t. The state s is the initial state since it has an incoming edge without a source. The structure has three transitions: from s to t, from t to s and from t to itself. The labeling is such that $L(s) = \{P\}$ and $L(t) = \{Q\}$ i.e. the atomic proposition P holds in s and the atomic proposition Q holds in t.

2.2 Concurrent Systems

A concurrent system [10] is a set of components that execute together. Concurrent systems are typical targets of model checking. That is why it is important to specify exactly how the components execute with respect to each other. Typically the mode of execution is either synchronous execution or asynchronous execution.

Synchronous execution means that all the components of the concurrent system execute at the same time. The operation of a synchronous system consists of a sequence of steps. In each step, the inputs to the system change. After this the clock pulse occurs, and all the system output values are calculated using these inputs.

In asynchronous execution, only one component makes a step at a time. It is not required that every component gets a turn. Some component can execute repeatedly, without another component ever making a step.

3 Temporal Logic

Temporal logic is an extension of classical logic. It uses atomic propositions, boolean connectives and some temporal operators. Some temporal logics like linear temporal logic describe the ordering of events in time without a real valued clock. It might be specified that some property eventually happens or never happens. Temporal logics can be classified according to the assumed structure of time. Some temporal logics assume linear time structure, some assume a branching time structure.

3.1 Linear Temporal Logic

Linear temporal logic(LTL) [13] has linear time structure, so it sees future as a path. The formal semantics of temporal formulas is defined with respect to paths of a Kripke structure[4]. When a set of paths is considered, the LTL formula has to be true on all paths to be true.

Linear temporal logic uses atomic propositions, the usual boolean connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$ and the following temporal modal operators:

- **X** ("next") requires that the property holds at the next state of the path.

- **G** ("globally") requires that the property holds at every state on the path.

- **F** ("eventually" or "in the future") holds when a property is true at some state of the path.

- **U** ("until") is a binary operator. Formula $P\ \mathbf{U}\ Q$ holds when P is true until Q becomes true. Also, the second argument must become true at some point.

- **R** is for release. Formula $P\ \mathbf{R}\ Q$ requires that either Q is always true or it is true until P becomes true.

Consequently, the LTL formulas are as follows:

$$\begin{aligned}\phi \quad ::= \quad & true \mid false \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \phi_1 \rightarrow \phi_2 \\ & \mid \mathbf{G}\phi \mid \mathbf{F}\phi \mid \mathbf{X}\phi \mid [\phi_1 \mathbf{U}\, \phi_2] \mid [\phi_1 \mathbf{R}\, \phi_2]\end{aligned}$$

For example a LTL formula $\mathbf{G}(a\,\mathbf{U}\,(b \vee c))$ would mean that on every path either b or c is true at the current position or a holds until either b or c becomes true.

The properties considered in model checking are usually either safety properties or liveness properties. Safety properties usually state that something bad never happens ($\mathbf{G}\phi \rightarrow \mathbf{G}\neg\sigma$). Liveness properties state that something good keeps happening ($\mathbf{G}\phi \rightarrow \mathbf{F}\sigma$).

4 Bounded Model Checking

Bounded model checking (BMC) [4] is a model checking technique which uses a propositional SAT solver [3] instead of using binary decision diagram (BDD) [4, 6] techniques. In BMC the basic idea is to try to find counter examples whose length is bounded by some integer k. The bound is increased until a counter example is found or the bound exceeds the completeness threshold (CT). Given a model M, a property p and a translation scheme, CT is an integer such that the absence of errors for CT cycles proves that $M \models p$. The completeness threshold exists for finite state systems. Usually the completeness threshold is not known. In these cases the calculations are terminated when they become too intense and complex.

The BMC problem can be effectively reduced to a propositional satisfiability problem that can be solved using a SAT solver. The BMC technique does not ensure better performance or productivity, although in some cases it can outperform the traditional BDD-techniques. This is because SAT solving does not suffer from the state explosion problem unlike BDD techniques.

The reduction to a SAT problem is quite complex. Suppose the model is given as a Kripke structure M, and the specification to be checked is given as a LTL formula f. The integer bound is k. Let $s_0, ..., s_k$ be a finite sequence of states on a path π. We will construct a propositional formula $[[M, \neg f]]_k$ such that it is satisfiable iff π is a trace that contradicts the formula f. The full SAT reduction is represented in [4].

5 NuSMV Model Checker

NuSMV [7] is a symbolic model checker that can be used to analyse temporal logic (LTL or CTL) specifications of various systems. The system can be expressed in the NuSMV modeling language. The system specifications are expressed in temporal logic.

5.1 NuSMV Models

The NuSMV modeling language allows the representation of synchronous and asynchronous finite state systems. The models can be divided into modules that can interact with each other. Every model contains the main module, which can have references to the other modules. The allowed data types in NuSMV are boolean, bounded integer and enumeration types. Arrays and set types are also allowed.

To be able to check LTL properties against the system, NuSMV flattens the model into a Kripke structure. This is possible because a finite state system is very similar to the Kripke structure discussed earlier. Finite state system is a triplet (S, T, I), where S is the set of states, T is the transition relation and I is the initial state. These correspond to the states, transition relation and the initial states of the Kripke structure. Each state in the Kripke structure is essentially a tuple containing one value for each state variable. A transition in

a Kripke structure denotes change in the value of one or more state variables. A given property is actually checked against the Kripke structure obtained by flattening the given model.

The states of a finite state system are described by the state variable values in NuSMV. State variables can be deterministic output variables or non-deterministic input variables. This means that an input variable can be given any value from its domain without any specification of which one will be taken. Output variable values are determined unambiguously based on other variable values. The initial states are determined by giving the state variables some initial values. If an initial value is not given, it means that the variable can have any value at that time point. The transition relation of a finite state system is represented as "next" commands in the NuSMV model. These commands are the rules based on which the value of the output variables are determined after the initial time point.

An example of an output variable is shown below.

```
init(time2) := 0;
next(time2) := case
(relay2buffer = alarm & time2 < 30): time2 +1;
(relay2buffer = alarm & time2 = 30):  30;
1 : 0;
esac;
```

We can see that the initial value of the variable **time2** is set to 0. After the initial time point the value depends on **relay2buffer** and the value of **time2** itself. If **relay2buffer** has value "alarm" and **time2** is less that 30, the value of **time2** will be increased by one. If **relay2buffer** has value "alarm" and **time2** is in its maximum value 30, the value of **time2** will not change. In other cases the value of **time2** is set back to 0.

A model of a possible safety system is presented in Appendix A. The model uses basic boolean logic connectors (AND and OR) and counter variables to calculate the output variable values. It is a description of the rules according to which the system shuts down parts of its power sources in case of an emergency. The model has two parts: the variable definition part (VAR), and the variable assignment part (ASSIGN). The VAR-part contains all the variable declarations. The ASSIGN-part contains the initialisation and the "next"-rules. In the last line of the model is the LTL specification.

The LTL specification to be checked claims that it is not possible for the variable **relay2** to become in the state "alarm". In our presented system this would mean that shutting down some part of the system would never be possible. In other words, we want to know if the relay is unnecessary in the system. We are expecting to find a counter example stating otherwise.

5.2 NuSMV Counter Examples

Model checking procedures will either confirm the specification or produce a counter example. A counter example describes a computation that is allowed

in the model but make the examined system specification false.

NuSMV flattens the model and transforms it into a Kripke structure. The paths in the Kripke structure depict all the possible computations of the model. Counter examples are paths in the Kripke structure, that do not behave according to the given LTL specification.

Since a path is a sequence of states, and states are identified by the variable values of the model, NuSMV can represent counter examples by basically listing the states of the path in traversing order. In the initial state all the variable values are listed. After that, a variable value is printed only in case of a change in its values. An example of such a counter example is shown below.

```
-- specification  G (z = alarm ->  F x)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  x = 1
  n = 1
  z = OK
-> Input: 1.2 <-
-> State: 1.2 <-
  x = 0
  n = 2
  z = alarm
-> Input: 1.3 <-
-- Loop starts here
-> State: 1.3 <-
  n = 1
  z = OK
-> Input: 1.4 <-
-> State: 1.4 <-
  n = 2
-> Input: 1.5 <-
-> State: 1.5 <-
  n = 1
```

First it is stated that the examined specification was false. There exists an execution sequence in the system that breaks the LTL formula:

```
G (z = alarm ->  F x)
```

The specification basically states that if **z** has value "alarm" then **x** has to become true in the future. The counter example shows how this specification could however break. We can see that in the initial time step(State 1.1) variables **x** and **n** have value 1. **z** has an initial value "OK". In the next time step (State 1.2) **z** gets the value "alarm" and **x** is set to 0. Now, according to the specification, **x** should become true in the future.

But there is a possible loop presented in the counter example. This is marked with the "Loop starts here" -line. The line just means that the variable values

are the same in these states and the final state, so the behaviour can occur repeatedly. We can see that **x** never becomes true in the loop. This is why the specification was false.

6 Cone of Influence Reduction

Cone of influence (COI) reduction [10] is an abstraction technique that focuses on the variables in the specification. Cone of influence reduction eliminates variables that do not influence the variables in the specification. In our case, the benefit is seen in the length and complexity of the counter example.

We will show how the cone of influence reduction can be applied to synchronous circuits. Let V be the set of variables in a given circuit. This circuit can be described by a set of equations

$$v_i' = f_i(V),$$

for each $v_i \in V$, where f_i is a boolean function. Suppose we are given a set of variables $V' \subseteq V$ that are of interest with respect to the required specification. We would like to simplify the description of the system by referring only to these variables. However, the values of variables in V' might depend on variables not in V'. We therefore define the cone of influence C for V' and use C in order to reduce the description of the system.

The cone of influence C of V' is the minimal set of variables such that

•$V' \subseteq C$.
• if for some $v_l \in C$ its f_l depends on v_j, then $v_j \in C$.

We will next show that the cone of influence reduction preserves the correctness of specifications used in NuSMV (CTL, LTL) if they are defined over variables in C.

Let $V = \{v_1, ..., v_n\}$ be a set of boolean variables and let $M = (S, R, S_0, L)$ be the model of a synchronous circuit defined over V where,

•$S = \{0,1\}^n$ is the set of all valuations of V.
•$R = \{(s,t) \in S \times S \mid s \models v \in V, t \models v_i', v_i' = f_i(V)\}$
•$L(s) = \{v_i \mid s(v_i) = 1 \text{ for } 1 \leq i \leq n\}$.
•$S_0 \subseteq S$.

Suppose we reduce the circuit with respect to the cone of influence $C = \{v_1, ..., v_k\}$ for some $k \leq n$. The reduced model $\hat{M} = (\hat{S}, \hat{R}, \hat{S_0}, \hat{L})$ is defined by

•$\hat{S} = \{0,1\}^k$ is the set of all valuations of $\{v_1, ..., v_k\}$.
• $\hat{R} = \left\{(s,t) \in \hat{S} \times \hat{S} \mid s \models v \in C, t \models v_i', v_i' = f_i(C)\right\}$
• $\hat{L}(\hat{s}) = \{v_i \mid \hat{s}(v_i) = 1 for 1 \leq i \leq k\}$.
• $\hat{S_0} = \{(\hat{d_1}, ..., \hat{d_k}) \mid$ there is a state $(d_1, ..., d_n) \in S_0$ such that $\left(\hat{d_1} = d_1\right) \wedge ... \wedge$

$\Big(\hat{d_k} = d_k\Big)\}$.

We want to show that the models M and $\hat{M}$ behave similarly. Bisimulation is a very strong indication of similar behaviour. A relation $B \subseteq S \times \hat{S}$ is a bisimulation relation between M and $\hat{M}$ if and only if for all s and $\hat{s}$, if $(s, \hat{s}) \in B$ then the following conditions hold:

$\bullet L(s) = \hat{L}(\hat{s})$.
• For every state s_1 such that $(s, s_1) \in R$ there is $\hat{s_1}$ such that $(\hat{s}, \hat{s_1}) \in \hat{R}$ and $(s_1, \hat{s_1}) \in B$.
• For every state $\hat{s_1}$ such that $(\hat{s}, \hat{s_1})$ there is s_1 such that $(s, s_1) \in R$ and $(s_1, \hat{s_1}) \in B$.

Let $B \subseteq S \times \hat{S}$ be the relation defined as follows:

$$((d_1, ..., d_n), (\hat{d_1}, ..., \hat{d_k})) \in B \iff d_i = \hat{d_i} \text{ for all } 1 \leq i \leq k$$

We show that B is a bisimulation relation between M and $\hat{M}$. For every initial state in S there is a corresponding initial state in $\hat{S}$ and vice versa. Let $s = (d_1, ..., d_n)$ and $\hat{s} = (\hat{d_1}, ..., \hat{d_k})$ such that $(s, \hat{s}) \in B$. Then $d_i = \hat{d_i}$ for every $1 \leq i \leq k$. Thus, their labeling restricted to $C = \{v_1, ..., v_k\}$ agree, that is,

$L(s) \cap C = \hat{L}(\hat{s})$.

Let $s \to t$ be a transition in M. We show that there is a transition $\hat{s} \to \hat{t}$ in M such that $(t, \hat{t}) \in B$. Denote $t = (e_1, ..., e_n)$.

The definition of R implies that for every $1 \leq i \leq n, v_i' = f_i(V)$. However, for $1 \leq i \leq k$, v_i depends only on variables in C, hence $v_i' = f_i(C)$. Furthermore, $(s, \hat{s}) \in B$ implies $\bigwedge_{i=1}^{k}(d_i = \hat{d_i})$. Thus, for every $1 \leq i \leq k$,

$e_i = f_i(d_1, ..., d_k) = f_i(\hat{d_1}, ..., \hat{d_k})$.

If we choose $\hat{t} = (e_1, ..., e_k)$ then $\hat{s} \to \hat{t}$ and $(t, \hat{t}) \in B$ as required. Now let $\hat{s} \to \hat{t}$ be a transition in $\hat{R}$ where $\hat{t} = (\hat{e_1}, ..., \hat{e_k})$. Then, for every $1 \leq i \leq k$, $\hat{e_i} = f_i(\hat{d_1}, ..., \hat{d_k})$. Consider the transition $s \to t$ in R for some $t = (e_1, ..., e_n)$. Since $\bigwedge_{i=1}^{k}(d_i = \hat{d_i})$ and since the value of $v_i \in C$ depends only on values of variables in C, we have:

$\hat{e_i} = f_i(\hat{d_1}, ..., \hat{d_k}) = f_i(d_1, ..., d_k) = f_i(d_1, ..., d_k, d_{k+1}, ...d_n) = e_i$.

Hence, $(t, \hat{t}) \in B$. This completes the proof that B is a bisimulation between M and $\hat{M}$. Thus, $M \equiv \hat{M}$.

If two structures are bisimulation equivalent, then every initial state of one is bisimilar to some initial state of the other [10]. Because a structure satisfies a formula if and only if each of its initial states satisfies the formula, both structures will satisfy the same set of LTL formulas. Now, because $M \equiv \hat{M}$, it directly follows:

$M \models f \Leftrightarrow \hat{M} \models f$, where f is a LTL formula with atomic propositions in C.

In other words, the cone of influence reduction preserves correctness.

In practice, NuSMV can calculate the cone of influence structure directly from the NuSMV model. Variable relationships can be determined from the variable assignments. In our example, the amount of variables of the counter example in Appendix B was decreased from 25 to 5. The counter example was produced with NuSMV from the model in Appendix A.

If the model in case is examined, it becomes clear that only these five variables have had influence on the value of the specification. The LTL formula there was:

```
LTLSPEC G !(relay2 = alarm);
```

The goal was to find out which variables of the model really create the counter model. It is quite clear that the output of **relay2** = alarm depends on the value of the variable **relay2**. No other variables have a direct connection or influence to the value of the clause. It is, however, possible for the other variables to have some indirect influence to the output of the LTL formula. That is because the value of **relay2** depends on other variables on the previous time steps. In the model we can see how the value of **relay2** is determined:

```
next (relay2) := case
     ((time2 = 30) & (ch1 & ch4)) : alarm;
     1   : OK;
esac;
```

On the grounds of this we can deduce that since **relay2** is directly influenced by variables **time2**, **ch1** and **ch4**, also the LTL formula is somehow influenced by these variables i.e. the variables belong to the cone of influence of the formula. Now we can continue this kind of iterative examination of the variables with **time2**, **ch1** and **ch4**. We notice that **ch1** and **ch4** are non-determined input variables of the model. This means that they can have any values and no other variables have any influence on them. This leaves only variable **time2**:

```
next (time2) := case
     (relay2buffer = alarm & time2 < 30): time2 +1;
     (relay2buffer = alarm & time2 = 30):  30;
     1 : 0;
esac;
```

The value of **time2** depends on the value of itself and the variable **relay2buffer**. This is also added to the cone of influence of the formula since it has influence on **time2** which has influence on **relay2** which solely determines the output of the formula. We still have to check the dependencies of **relay2buffer**:

```
next(relay2buffer) := case
(ch1 & ch4) : alarm;
1   : OK;
esac;
```

It is easy to see that no new variables are introduced. Variable **relay2buffer** depends only on **ch1** and **ch4**, but both of these variables are already in the cone of influence. Now the iterative work is done because no new variables were found. No other variables can belong to the cone of influence. It now contains all the variables that have any influence on the formula, and all variables that have any influence on other variables that have some direct or indirect influence on the formula.

7 Value Change Dump

Value Change Dump (VCD) [11] is an industry standard file format that is widely supported. VCD files are used to describe a computation of a computer program. The changes in variable values according to time are recorded.

A simple VCD file could be the following:

```
$timescale 1 ns $end
$var integer 8 cycle cycle $end

$enddefinitions     $end

#1
b1 cycle

#3
b10 cycle

#5
b11 cycle
```

First, the time scale of the computation is defined. In this case one time step corresponds to one nanosecond. Next, a single integer variable **cycle** is defined. After this the definition part ends, and the actual computation part begins. The lines starting with the "#"-character depict the changes in time. The value of time starts from 1 and is only supposed to increase. Some time values can be skipped. Here the only time values shown are 1, 3 and 5.

After the current time value is mentioned, the changes in variable values can be stated. The new variable value is always written in binary, regardless of the actual variable type. The line "b10 cycle" states that variable "cycle" has a new value 10 in binary, which translates into 2 in decimal value.

The computation depicted in this VCD example is such that in the beginning the variable **cycle** has value 1. After that the value increases by 1 every two time steps. The computation ends after five time steps.

8 Simplifying Counter Examples

When a model checking method detects a contradicting execution path, it produces a counter example. It is left for humans to find out the real cause of the counter example i.e. what is the actual flaw in the system. Sometimes counter examples make this examination really difficult. Counter examples might have a lot of redundant information. Usually only some proportion of the variables are necessary to understand the flaw. Counter examples can also be quite long. A lengthy example of a NuSMV counter example is in Appendix B. It is a path containing 37 states and 25 different variables. The counter example was produced from the fairly simple NuSMV model in Appendix A. Already here it is quite time consuming to try to find out what really happens in the counter example. When the modeled system becomes more complex, the amount of shown variables naturally increases. Also, longer counter examples are possible. Apparently, it is of interest to try to reduce the amount of redundant information. A more readable format would also be convenient.

8.1 Implementation

The program created for this task is called NuSMVToVCD. It can do both: translate the counter example into a more compact graphical representation, and reduce the amount of shown variables in the counter example.

The graphical representation is produced by changing the NuSMV output into an equivalent VCD file. VCD files can then be opened with a graphical software such as GtkWave [1]. GtkWave's graphical view has time as the x-axis. The variable values are plotted in parallel with respect to time. Boolean values are shown as waveform signals. In case of integer or ASCII -variables, the value of the variable is printed. A picture of GtkWave interface with the NuSMV counter example presented earlier is in Figure 2. The list of variables is in the middle titled Signals. On the right are the variable values with time as the x-axis.

NuSMVToVCD can also perform some variable reductions. This is done by examining the cone of influence structure of the variables mentioned in the specification. Variables not in this structure are left out.

NuSMVToVCD consists of perl scripts and a shell script nusmv2vcd which will handle the use of the perl scripts and all the other files created during execution of the program. The program uses NuSMV [7] to create a counter example and then parses it. The counter example length is then used as an upper bound for bounded model checking of the model. The bounded model checking result can be different from the original output so the counter example needs to be parsed again. After this it can be transformed into a VCD -file. During

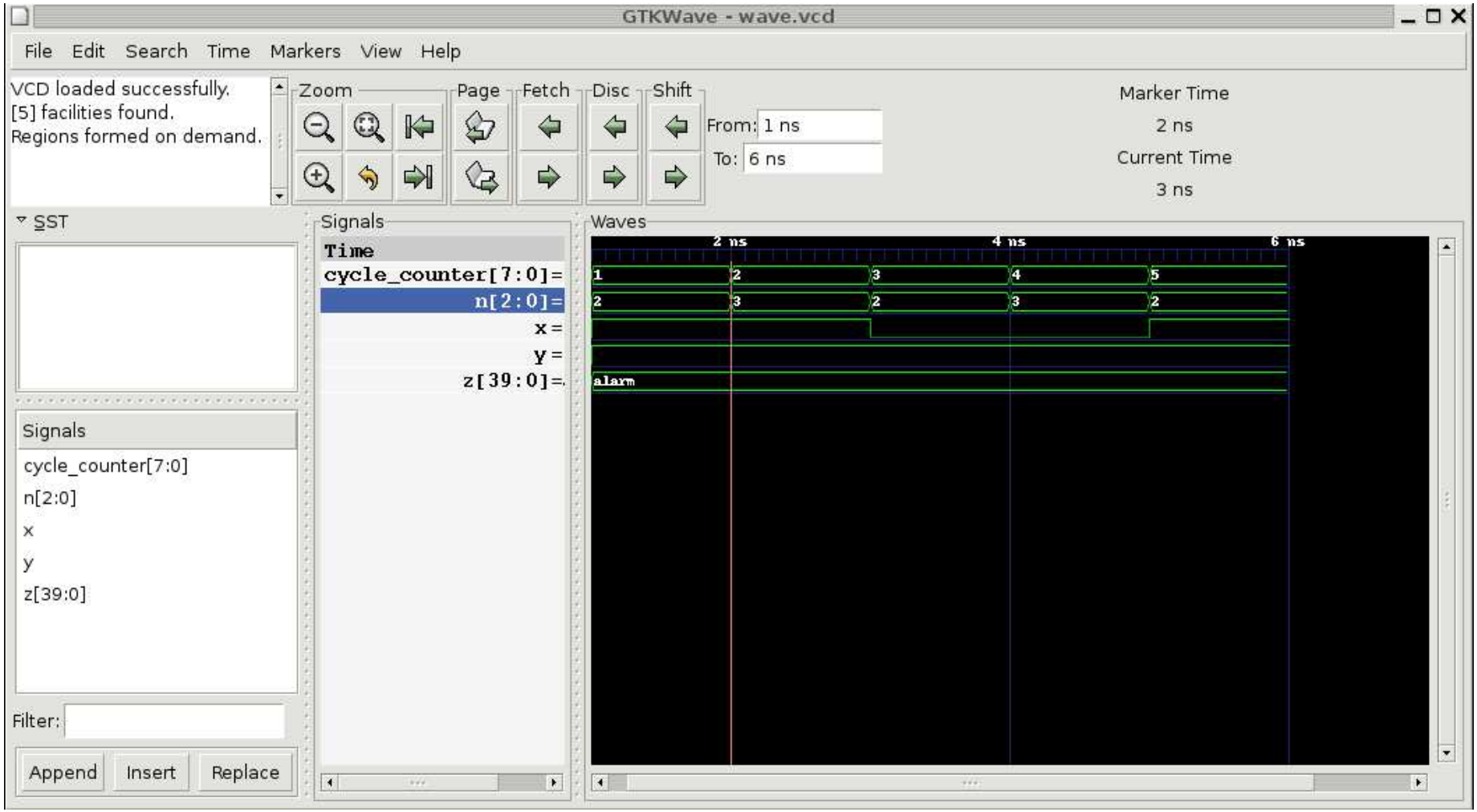

Figure 2: GtkWave Interface

the bounded model checking a cone of influence for every variable is calculated. Using this information, the necessary variables are isolated from the others by putting them to a separate variable scope. The redundant variables are also removed from the graphical view.

A description of the program with Cone of Influence -reduction in pseudocode:

```
if (parameter = -coi)
   Create all tempfiles
   Write variable names to a tempfile
   Write examined property to a tempfile
   Use NuSMV to find a counter example, write to a file
   if (specification is true)
      print "The specification was true"
      exit

   Check counter example length
   Use the length as an upper bound of BMC with NuSMV
   Use NuSMV to write shortest counter example and the
      Cone of Influence structure to a file

   parse the counter example -file
   extract Cone of Influence -variables
   extract the counter example -part and filter it to a basic SMV format
   change variable names with dots(.)

   Use smv2vcd to create a .vcd file and a .sav file
   Modify the files to create variable scopes(Relevant / Other)
   Modify the .sav -file so that only coi-variables are shown
   Create a script file to assist gtkwave appearance
   remove temporary files
   launch gtkwave
   remove .vcd .sav and the gtkwave-script -files
end
```

The transformation from NuSMV counter examples to a vcd-file is possible because the NuSMV [7] counter example format is very similar to the VCD-format [11]. Both have some measurement of time, and in both the changes of variable values are somehow recorded. This is why the change between these formats is quite easy and can be done line by line of the NuSMV output.

A tool for this action already exists (smv2vcd)[2] , but it does not fully support this version of NuSMV. That is why the NuSMV output needs to be formatted: the indentation has to be removed as well as the "Input" lines and arrow signs (-> and <-).

The relevant variables can be found by examining the output of the bounded model checking [10] of the model, with high verbose level and the cone of influence reduction enabled. From this output a cone of influence of every variable can be found. By combining the COIs of the variables in the examined specifi-

cation, only the relevant variables remain.

When the relevant variables are known, and a VCD-file is made, the visible variables can be selected by modifying the GtkWave save-file where a list of visible variables is.

8.2 Limitations of the Used Method

The program will expect to find only one LTL-specification that has to be false. More than one specification will probably cause unexpected behaviour.

We will be checking only LTL(Linear Temporal Logic) specifications using NuSMV. The use of CTL-clauses is not supported although the program will probably manage to visualise a counter model when the "-coi" flag is not used.

The NuSMV model checker does not recognise the macro-variables as real variables. This is why the COI-reduction sometimes fails to represent explanatory variables, and only some core variables are shown. The macro variables in NuSMV are not part of the cone of influence -structure.

8.3 Known Flaws of the Implementation

In GtkWave save file -format the use of dot(.)-character is reserved for variable grouping purposes. This causes problems with variable names containing the dot-character. This is resolved so that all the dots in variable names will be automatically changed to underscores(_) after the counter model has been created.

LTL-specifications that are outside the main module are troublesome. This is because the variables outside the main module are normally referred to as "module.variable-name". In a LTL-specification outside the main module the module part is left out. Thus the variables can not be recognised.

9 Results

Using the program, the counter example (273 lines) in Appendix B can be fit on one page. The graphical representation is in Figure 3. After the variable reductions are applied, the amount of shown variables decreased from 25 to 5. This result can be seen in Figure 4.

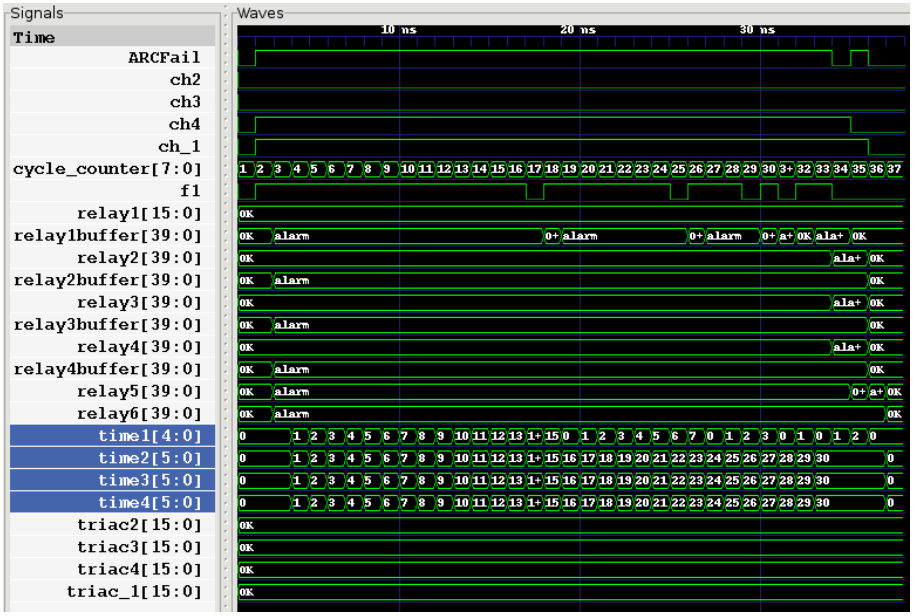

Figure 3: Counter model in graphical form, without variable reductions.

10 Conclusions

In our result, the amount of variables of the counter example in Appendix B decreased from 25 to 5. The counter example was produced with NuSMV from the model in Appendix A.

We have come to the conclusion that only these five variables cause the specification to fail. These are exactly the variables presented by our program. In this case the reduction has thus been successful.

In general, the effectiveness of this method depends greatly on the model itself. In some models all the variables depend on each other, and the cone of influence reduction does not help at all. Sometimes the method can come up with only a small set of explanatory variables.

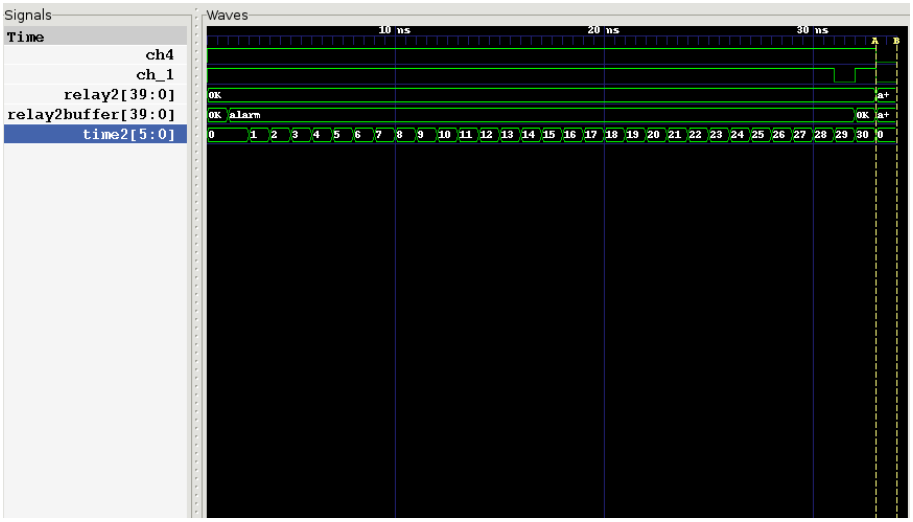

Figure 4: Counter model in graphical form with cone of influence reductions.

11 Further Development

The cone of influence reduction is not perfect. It only shows the variables that can, but not necessarily have influence on the specification. It is also possible that some variable forms a tautology (a clause that is always true). In this case the variable is redundant, unless it additionally exists somewhere else.

The LTL-clause:

```
LTLSPEC G((x & !x) | y);
```

does not really depend on x, because (x & !x) is always false. Basically the clause reduces to:

```
LTLSPEC G(y);
```

This reduction is out of our reach with cone of influence examination.

It could also be the case that y was always true and the specification is:

```
LTLSPEC G !(x | y);
```

Here we don't really need to include x and the variables in its cone of influence, since the fact that y is always true, already explains the supposed counter example.

To further reduce the amount of necessary variables we could simulate the counter example with some of the variables left non-deterministic and see if the counter example would still unavoidably be the same.

A Appendix A: The NuSMV model

```
MODULE main
VAR
ch1 : boolean;
ch2 : boolean;
ch3 : boolean;
ch4 : boolean;

f1 : boolean;
ARCFail : boolean;
triac1 : {OK, alarm};
triac2 : {OK, alarm};
triac3 : {OK, alarm};
triac4  : {OK, alarm};

relay1 : {OK, alarm};
relay2 : {OK, alarm};
relay3 : {OK, alarm};
relay4 : {OK, alarm};
relay5 : {OK, alarm};
relay6 : {OK, alarm};

relay1buffer : {OK, alarm};
relay2buffer : {OK, alarm};
relay3buffer : {OK, alarm};
relay4buffer : {OK, alarm};

time1 : 0..30;
time2 : 0..30;
time3 : 0..30;
time4 : 0..30;

ASSIGN
init(triac1) := OK;
init(triac2) := OK;
init(triac3) := OK;
init(triac4) := OK;
init(relay1) := OK;
init(relay2) := OK;
init(relay3) := OK;
init(relay4) := OK;
init(relay5) := OK;
init(relay6) := OK;
init(relay1buffer) := OK;
init(relay2buffer) := OK;
init(relay3buffer) := OK;
init(relay4buffer) := OK;
init(time1) := 0;

next(time1) := case
(relay1buffer = alarm & time1 < 30): time1 +1;
(relay1buffer = alarm & time1 = 30):  30;
1 : 0;
esac;
init(time2) := 0;

next(time2) := case
(relay2buffer = alarm & time2 < 30): time2 +1;
(relay2buffer = alarm & time2 = 30):  30;
1 : 0;
esac;
init(time3) := 0;

next(time3) := case
(relay3buffer = alarm & time3 < 30): time3 +1;
(relay3buffer = alarm & time3 = 30): 30;
1 : 0;
esac;
init(time4) := 0;

next(time4) := case
(relay4buffer = alarm & time4 < 30): time4 +1;
(relay4buffer = alarm & time4 = 30): 30;
1 : 0;
esac;


next(triac1) := case
(ch2 & ch4) : alarm;
1   : OK;
esac;
next(triac2) := case
(ch3 & ch4) : alarm;
1   : OK;
esac;
next(triac3) := case
((ch2 & ch4) | (ch3& ch4)) : alarm;
1 : OK;
esac;
next(triac4) := case
((ch2 & ch4)| (ch3 & ch4)) : alarm;
1 : OK;
esac;
```

```
next(relay1buffer) := case
(ch4 & f1) : alarm;
1   : OK;
esac;
next(relay2buffer) := case
(ch1 & ch4) : alarm;
1   : OK;
esac;
next(relay3buffer) := case
((ch4 & f1) | (ch1 & ch4)) : alarm;
1   : OK;
esac;

next(relay4buffer) := case
((ch4 & f1 ) | (ch1 & ch4)) : alarm;
1   : OK;
esac;


next(relay1) := case
((time1 = 30) & (ch4 & f1)): alarm;
1   : OK;
esac;
next(relay2) := case
((time2 = 30) & (ch1 & ch4)) : alarm;
1   : OK;
esac;
next(relay3) := case
((time3 = 30) & ((ch4 & f1) | (ch1 & ch4))) : alarm;
1   : OK;
esac;
next(relay4) := case
((time4 = 30) & ((ch4 & f1)| (ch1 & ch4))) : alarm;
1   : OK;
esac;


next(relay5) := case
(ARCFail) : alarm;
1   : OK;
esac;
next(relay6) := case
(ch1|ch2|ch3|ch4|f1) : alarm;
1   : OK;
esac;

LTLSPEC G !(relay2 = alarm);
```

B Appendix B: The NuSMV counter example

```
-- specification  G !(relay2 = alarm)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-- Loop starts here
-> State: 1.1 <-
  ch1 = 0
  ch2 = 0
  ch3 = 0
  ch4 = 0
  f1 = 0
  ARCFail = 0
  triac1 = OK
  triac2 = OK
  triac3 = OK
  triac4 = OK
  relay1 = OK
  relay2 = OK
  relay3 = OK
  relay4 = OK
  relay5 = OK
  relay6 = OK
  relay1buffer = OK
  relay2buffer = OK
  relay3buffer = OK
  relay4buffer = OK
  time1 = 0
  time2 = 0
  time3 = 0
  time4 = 0
-> Input: 1.2 <-
-> State: 1.2 <-
  ch1 = 1
  ch4 = 1
  f1 = 1
  ARCFail = 1
-> Input: 1.3 <-
-> State: 1.3 <-
  relay5 = alarm
  relay6 = alarm
  relay1buffer = alarm
  relay2buffer = alarm
  relay3buffer = alarm
  relay4buffer = alarm
-> Input: 1.4 <-
-> State: 1.4 <-
  time1 = 1
  time2 = 1
  time3 = 1
  time4 = 1
-> Input: 1.5 <-
-> State: 1.5 <-
  time1 = 2
  time2 = 2
  time3 = 2
  time4 = 2
-> Input: 1.6 <-
-> State: 1.6 <-
  time1 = 3
  time2 = 3
  time3 = 3
  time4 = 3
-> Input: 1.7 <-
-> State: 1.7 <-
  time1 = 4
  time2 = 4
  time3 = 4
  time4 = 4
-> Input: 1.8 <-
-> State: 1.8 <-
  time1 = 5
  time2 = 5
  time3 = 5
  time4 = 5
-> Input: 1.9 <-
-> State: 1.9 <-
  time1 = 6
  time2 = 6
  time3 = 6
  time4 = 6
-> Input: 1.10 <-
-> State: 1.10 <-
  time1 = 7
  time2 = 7
  time3 = 7
  time4 = 7
-> Input: 1.11 <-
-> State: 1.11 <-
  time1 = 8
  time2 = 8
  time3 = 8
  time4 = 8
-> Input: 1.12 <-
-> State: 1.12 <-
  time1 = 9
  time2 = 9
  time3 = 9
  time4 = 9
-> Input: 1.13 <-
-> State: 1.13 <-
  time1 = 10
  time2 = 10
  time3 = 10
  time4 = 10
-> Input: 1.14 <-
-> State: 1.14 <-
  time1 = 11
  time2 = 11
  time3 = 11
  time4 = 11
-> Input: 1.15 <-
-> State: 1.15 <-
  time1 = 12
  time2 = 12
  time3 = 12
  time4 = 12
-> Input: 1.16 <-
-> State: 1.16 <-
  time1 = 13
  time2 = 13
  time3 = 13
  time4 = 13
-> Input: 1.17 <-
-> State: 1.17 <-
  f1 = 0
  time1 = 14
  time2 = 14
  time3 = 14
  time4 = 14
```

```
-> Input: 1.18 <-
-> State: 1.18 <-
  f1 = 1
  relay1buffer = OK
  time1 = 15
  time2 = 15
  time3 = 15
  time4 = 15
-> Input: 1.19 <-
-> State: 1.19 <-
  relay1buffer = alarm
  time1 = 0
  time2 = 16
  time3 = 16
  time4 = 16
-> Input: 1.20 <-
-> State: 1.20 <-
  time1 = 1
  time2 = 17
  time3 = 17
  time4 = 17
-> Input: 1.21 <-
-> State: 1.21 <-
  time1 = 2
  time2 = 18
  time3 = 18
  time4 = 18
-> Input: 1.22 <-
-> State: 1.22 <-
  time1 = 3
  time2 = 19
  time3 = 19
  time4 = 19
-> Input: 1.23 <-
-> State: 1.23 <-
  time1 = 4
  time2 = 20
  time3 = 20
  time4 = 20
-> Input: 1.24 <-
-> State: 1.24 <-
  time1 = 5
  time2 = 21
  time3 = 21
  time4 = 21
-> Input: 1.25 <-
-> State: 1.25 <-
  f1 = 0
  time1 = 6
  time2 = 22
  time3 = 22
  time4 = 22
-> Input: 1.26 <-
-> State: 1.26 <-
  f1 = 1
  relay1buffer = OK
  time1 = 7
  time2 = 23
  time3 = 23
  time4 = 23
-> Input: 1.27 <-
-> State: 1.27 <-
  relay1buffer = alarm
  time1 = 0
  time2 = 24
  time3 = 24
  time4 = 24
-> Input: 1.28 <-
-> State: 1.28 <-
  time1 = 1
  time2 = 25
  time3 = 25
  time4 = 25
-> Input: 1.29 <-
-> State: 1.29 <-
  f1 = 0
  time1 = 2
  time2 = 26
  time3 = 26
  time4 = 26
-> Input: 1.30 <-
-> State: 1.30 <-
  f1 = 1
  relay1buffer = OK
  time1 = 3
  time2 = 27
  time3 = 27
  time4 = 27
-> Input: 1.31 <-
-> State: 1.31 <-
  f1 = 0
  relay1buffer = alarm
  time1 = 0
  time2 = 28
  time3 = 28
  time4 = 28
-> Input: 1.32 <-
-> State: 1.32 <-
  f1 = 1
  relay1buffer = OK
  time1 = 1
  time2 = 29
  time3 = 29
  time4 = 29
-> Input: 1.33 <-
-> State: 1.33 <-
  relay1buffer = alarm
  time1 = 0
  time2 = 30
  time3 = 30
  time4 = 30
-> Input: 1.34 <-
-> State: 1.34 <-
  f1 = 0
  ARCFail = 0
  relay2 = alarm
  relay3 = alarm
  relay4 = alarm
  time1 = 1
-> Input: 1.35 <-
-> State: 1.35 <-
  ch4 = 0
  ARCFail = 1
  relay5 = OK
  relay1buffer = OK
  time1 = 2
-> Input: 1.36 <-
-> State: 1.36 <-
  ch1 = 0
  ARCFail = 0
  relay2 = OK
  relay3 = OK
  relay4 = OK
  relay5 = alarm
  relay2buffer = OK
  relay3buffer = OK
  relay4buffer = OK
  time1 = 0
-> Input: 1.37 <-
-> State: 1.37 <-
  relay5 = OK
  relay6 = OK
  time2 = 0
  time3 = 0
  time4 = 0
```

References

[1] *GTKWave 3.0 Wave Analyzer User's Guide*, April 2006.

[2] Model Checking at Carnegie Mellon University. A perl script to convert an smv/nusmv counterexample to industry standard value change dump (vcd) format. Available in `http://www.cs.cmu.edu/~modelcheck/`.

[3] Boolean Satisfiability Research Group at Princeton. SAT research at princeton. `http://www.princeton.edu/~chaff/`.

[4] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. In Advances in Computers (volume 58). Academic Press, 2003.

[5] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.

[6] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[7] Roberto Cavada, Alessandro Cimatti, Charles Arthur Jochim, Gavin Keighren, Emanuele Olivetti, Marco Pistore, Marco Roveri, and Andrei Tchaltsev. *NuSMV 2.4 User Manual*. ITC-irst, `http://nusmv.irst.itc.it/`.

[8] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model verifier. In *Computer Aided Verification*, pages 495–499, 1999.

[9] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. *Lecture Notes in Computer Science*, 2000:176–194, 2001.

[10] Orna Grumberg Doron A. Peled Edmund M. Clarke, Jr. *Model Checking*. The MIT Press, 1999.

[11] IEEE. *IEEE Standard Hardware Description Language Based on the Verilog Language*. The Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1996.

[12] David K. Probst and Hon F. Li. Using partial-order semantics to avoid the state explosion problem in asynchronous systems. In Edmund M. Clarke and Robert P. Kurshan, editors, *CAV*, volume 531 of *Lecture Notes in Computer Science*, pages 146–155. Springer, 1990.

[13] P. S. Thiagarajan and Jesper G. Henriksen. Distributed versions of linear time temporal logic: A trace perspective. In *Petri Nets*, pages 643–681, 1996.

[14] William G. Wood. Temporal logic case study. In *Automatic Verification Methods for Finite State Systems*, pages 257–263, 1989.