Using Symbolic Model Checking to Verify the Railway Stations of Hoorn-Kersenboogerd and Heerhugowaard

Cindy Eisner

IBM Haifa Research Laboratory
Matam Advanced Technology Center
Haifa, 31905 Israel
eisner@il.ibm.com

Abstract. Stålmarck's proof procedure is a method of tautology checking that has been used to verify railway interlocking software. Recently, it has been proposed [SS98] that the method has potential to increase the capacity of formal verification tools for hardware. In this paper, we examine this potential in light of an experiment in the opposite direction: the application of symbolic model checking to railway interlocking software previously verified with Stålmarck's method. We show that these railway systems share important characteristics which distinguish them from most hardware designs, and that these differences raise some doubts about the applicability of Stålmarck's method to hardware verification.

1 Introduction

Stålmarck's proof procedure is a method of tautology checking that has been used to verify railway interlocking software [GKV94, Fok95]. Based on the observation that these systems are hardware-like, it has been suggested that this method may have the potential to increase the capacity of formal verification tools for hardware [SS98]. Indeed, Biere, Cimatti, Clarke and Zhu [BCCZ98] have built a symbolic model checker in which boolean decision procedures like Stålmarck's method replace BDDs.

In this paper, this potential is examined in light of an experiment in the opposite direction: the application of symbolic model checking to railway interlocking software[1]. It is shown that the two railway stations commonly cited as successful applications of Stålmarck's method are *robust*: most properties required of them hold for all states in the state space rather than holding for the reachable states only. It is also shown that these models exhibit *locality*, in that only a small number of inputs toggle in any one counter-example to a non-valid formula. Finally, we show that the properties checked for these models belong to a subset of CTL formulas we call AGAX formulas. We show how the characteristics of robustness and locality cause these models to be particularly suitable to symbolic model checking of AGAX formulas, and speculate that robustness also aids the application of Stålmarck's method to these types of formulas. Finally, we note that most hardware systems do not exhibit robustness, which raises doubts about the applicability of Stålmarck's method to hardware.

[1] The models of stations Hoorn-Kersenboogerd and Heerhugowaard used in this paper are the property of Holland RailConsult and are used with permission.

The remainder of this paper is structured as follows. Section 2 covers the basics of symbolic model checking. Section 3 describes the structure of railway interlocking software in the language VLC, and the structure of properties required of such software. Section 4 discusses the application of symbolic model checking to the verification of such software, and defines AGAX formulas, which are typical of those used to verify VLC models. Section 5 defines robustness and locality, analyzes why it is "easy" to model check AGAX formula in robust systems, and shows how locality can be used to aid counter-example generation for false formulas. Section 6 presents experimental results of the application of symbolic model checking to the stations at Hoorn-Kersenboogerd and Heerhugowaard. Section 7 concludes with some speculation regarding the application of Stålmarck's method to robust systems, and casts doubt on the applicability of Stålmarck's method to hardware verification.

2 Preliminaries

CTL, or Computation Tree Logic [CE81], is a temporal logic useful for reasoning about the ongoing behavior of reactive systems, and is the logic used by the symbolic model checker SMV [McM93]. In CTL, temporal operators occur in pairs consisting of A or E, followed by F, G, U, or X, as follows:

1. Every atomic proposition is a CTL formula, and
2. If f and g are CTL formulas, then so are $\neg f, (f \wedge g), AXf, EXf, A[fUg], E[fUg]$

The remaining operators are viewed as abbreviations of the above, as follows: $f \vee g = \neg(\neg f \wedge \neg g), AFg = A[trueUg], EFg = E[trueUg], AGf = \neg E[trueU\neg f]$ and $EGf = \neg A[trueU\neg f]$.

The semantics of a CTL formula is defined with respect to a model M. A model is a quadruple (S, S_0, R, L), where S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is the transition relation, and L is the valuation, a function mapping each state with a set of atomic propositions true in that state. We require that there is at least one transition from every state. A computation path of a model M is an infinite sequence of states $(s_0, s_1, s_2, \cdots)$ such that $R(s_i, s_{i+1})$ is true for every i.

The notation $M, s \models f$ means that the formula f is true in state s of model M. The notation $M \models f$ is equivalent to $\forall s \in S_0 \; M, s \models f$. The semantics of a CTL formula is defined as follows:

$M, s \models p \iff p \in L(s)$, where p is an atomic proposition
$M, s \models \neg f \iff M, s \not\models f$
$M, s \models f \wedge g \iff M, s \models f$ and $M, s \models g$
$M, s_i \models AX\ f \iff$ for all paths $(s_i, s_{i+1}, ...), M, s_{i+1} \models f$
$M, s_i \models EX\ f \iff$ for some path $(s_i, s_{i+1}, ...), M, s_{i+1} \models f$
$M, s_i \models A[fUg] \iff$ for all paths $(s_i, s_{i+1}, ...), \exists k \geq i$ such that $M, s_k \models g$, and $\forall j$ such that $i \leq j < k, M, s_j \models f$
$M, s_i \models E[fUg] \iff$ for some path $(s_i, s_{i+1}, ...), \exists k \geq i$ such that $M, s_k \models g$ and $\forall j$ such that $i \leq j < k, M, s_j \models f$

Emerson and Clarke [CE81b] have shown that the operators of CTL can be characterized as fixed points. This is the basis of CTL model checking. In symbolic CTL model checking, fixed points are frequently expensive to calculate, and their calculation is one of the sources of state space explosion. In the sequel, we will show a way to avoid this explosion for the railway stations of Hoorn-Kersenboogerd and Heerhugowaard.

3 Railway interlocking software in the language VLC

The language VLC is described in full by Groote, Koorn and van Vlijmen [GKV94]. Briefly, a VLC program describes a reactive system which continually executes control cycles. In each control cycle a set of inputs is latched, which means that their value cannot change during the control cycle. Then, the next state value of the internal variables and outputs is calculated. The outputs are transmitted simultaneously to the outside world at the end of the calculation. Finally, the internal variables and outputs are latched.

For the most part, a VLC program consists of a group of boolean equations describing the next state function of the internal variables and outputs, where '+' indicates boolean or, '*' indicates boolean and, and ".N." indicates boolean not. A time delay may be associated with a boolean equation, which means that the assignment is executed only if the right hand side of the equation has been true for the number of control cycles indicated by the time delay.

Following is an example given by [GKV94]:

```
1 DIRECT INPUT SECTION
2 I
3 OUTPUT SECTION
4 U
5 CODE SYSTEM SECTION
6 CURRENT RESULT SECTION
7 R
8 SELF-LATCHED PARAMETER SECTION
9 V
10 TIMER EXPRESSION RESULT SECTION
11 Q
12 BOOLEAN EQUATION SECTION
13 APPLICATION = Example
14 TIME DELAY = 2 SECONDS BOOL Q = I
15 BOOL R = Q + V
16 BOOL V = Q * .N.R
17 BOOL U = V
18 END BOOLEAN EQUATION SECTION
```

Fig. 1. An example VLC program

The boolean assignment to Q on line 14 in Figure 1 is time delayed by 2 seconds (where one second indicates one control cycle). This means that Q will get the value

$TRUE$ only if I is $TRUE$ and was also $TRUE$ in the previous 2 control cycles. The boolean assignment to R on line 15 will use the new value of Q when calculating $Q \vee R$, because it appears after the assignment to Q in the sequential order of the program statements. Similarly, the boolean assignment to V on line 16 will use the new values of Q and R to calculate $Q \wedge \neg R$ and the boolean assignment to U on line 17 will use the new value of V.

Properties verified by Groote et al [GKV94] and Fokkink [Fok95] are propositional formulas using current state variables and past state variables, as follows: if V is a current state variable, then V_j_1 indicates its value one cycle in the past, V_j_2 indicates its value 2 cycles in the past, etc.

4 The application of symbolic model checking to VLC code

In order to apply symbolic model checking to VLC code, it is necessary to translate VLC to the input language of the symbolic model checker. Here we describe the translation to the language EDL, a dialect of SMV [McM93] accepted by the symbolic model checker RuleBase [BBEL96].

The explanations which follow use the "Little Yard" of [GKV94] in VLC (Figure 2) and EDL (Figure 3).

```
1 DIRECT INPUT SECTION
2 I
3 OUTPUT SECTION
4 Pr Pn A B C
5 CODE SYSTEM SECTION
6 CmdA CmdB CmdC Cmdr
7 CURRENT RESULT SECTION
8 E
9 SELF-LATCHED PARAMETER SECTION
10 TIMER EXPRESSION RESULT SECTION
11 P
12 BOOLEAN EQUATION SECTION
13 APPLICATION = LY
14 BOOL E = A * B * C + A * B + A * C + B * C
15 TIME DELAY = 1 SECONDS BOOL P = I
16 BOOL Pr = .N.A * .N.B * .N.C * Cmdr
17 BOOL Pn = .N.Pr
18 BOOL A = CmdA * .N.CmdB * .N.CmdC * .N.E * P * Pn
19 BOOL B = CmdB * .N.CmdA * .N.CmdC * .N.E * P * Pr
20 BOOL C = CmdC * .N.CmdA * .N.CmdB * .N.E * P * Pn
21 END BOOLEAN EQUATION SECTION
```

Fig. 2. Program "Little Yard" in VLC

```
1 – inputs
2 var I, CmdA, CmdB, CmdC, Cmdr: boolean;
3 – next state variables
4 var Pr_out, Pn_out, A_out, B_out, C_out, E_out, P_out: boolean;
5 assign next(Pr_out) := Pr;
6 assign next(Pn_out) := Pn;
7 assign next(A_out) := A;
8 assign next(B_out) := B;
9 assign next(C_out) := C;
10 assign next(E_out) := E;
11 assign next(P_out) := P;
12 – counters
13 var P_count(0): boolean;
14 assign init(P_count(0)) := 0;
15 next(P_count(0)) := if !(P_temp) then 0 else P_count_inc(0) endif;
16 define P_count_inc(0) := if P_count(0)=1 then 1 else P_count(0)+1 endif;
17 – current state symbols
18 define E := A_out&B_out&C_out | A_out&B_out | A_out&C_out | B_out&C_out;
19 define P := P_temp & (P_count(0)=1);
20 define P_temp := I;
21 define Pr := !A_out & !B_out & !C_out & Cmdr;
22 define Pn := !Pr;
23 define A := CmdA & !CmdB & !CmdC & !E & P & Pn;
24 define B := CmdB & !CmdA & !CmdC & !E & P & Pr;
25 define C := CmdC & !CmdA & !CmdB & !E & P & Pn;
```

Fig. 3. Program "Little Yard" in EDL

4.1 Current vs. latched state variables

There is one important semantic difference between VLC code and EDL that is immediately obvious in the syntax: VLC is sequential in nature, i.e., the order of the statements matters (as in most software languages), while EDL code is parallel - there is no importance to the order of statements (as in most hardware languages). Therefore, the translation must take into account the position of each VLC statement as follows: if the variable V is used in an expression which appears before the assignment of a new value to V, the old, latched value of V should be used in the calculation. If the variable V is used in an expression which appears after the assignment of a new value to V, the new value of V should be used in the calculation.

Thus, in the VLC statement on line 14 of Figure 2, the signals A and B and C refer to the values the previous cycle, because no new values have yet been set. The equivalent EDL statement is that on line 18 in Figure 3 in which we must explicitly state (by the use of the $*\_out$ variables) that the assignment to E uses the previous values. On the other hand, in the VLC statement on line 17 of Figure 2 the signal Pr refers to the value the current cycle, because Pr receives a new value in a statement appearing above this statement in the code. This translates into the EDL statement of line 22 in Figure 3 in which we explicitly state (by the use of the current state symbol Pr) that the assignment to Pn uses the current value of Pr.

4.2 Time delay statements

The time delay statements of VLC are translated into temporary variables which count up to the delay, and whose value is tested to have reached the delay before assignment to the signal being assigned. Thus, the time delayed boolean assignment of line 15 in Figure 2 is translated into the EDL counter of lines 13-16 in Figure 3 plus the assignment of line 19. In this small example, a one-bit counter is used, but of course it is usually the case that a wider counter is needed.

4.3 Translation of propositional formulas to CTL

The formulas used by [GKV94] and [Fok95] have the property that they are translatable into a subset of CTL formulas described below. We will see below that this aids the model checking process for the railway stations in question.

Definition 1 (nested-AX formula). *A nested-AX formula is defined as follows:*

1. *Every propositional formula is a nested-AX formula*
2. *If f is a nested-AX formula, then AXf is a nested-AX formula*
3. *If p is a propositional formula and f is a nested-AX formula, then $p \to AXf$ is a nested-AX formula*

Definition 2 (AGAX formula). *An AGAX formula is defined as follows: if f is a nested-AX formula, then AGf is an AGAX formula*

Definition 3 (Depth). *The depth of an AGAX formula is the number of AX operators it contains.*

The propositional formulas used by [GKV94] and [Fok95] as described in section 3 translate into AGAX formulas in CTL as follows. The current state variables and past state variables are "shifted into the future" by as many cycles as are needed to get rid of the past. For instance, the propositional formula

$$\neg(74\_R\_ACO\_J\_1 \vee 74\_R\_ACO) \to 68\_R\_ACO \tag{1}$$

is equivalent to the propsitional formula

$$\neg 74\_R\_ACO\_J\_1 \to (\neg 74\_R\_ACO \to 68\_R\_ACO) \tag{2}$$

We shift the past state variable 74_R_ACO_J_1 to the current state variable 74_R_ACO by dropping the _J_1, and the current state variables to next state variables by adding an AX. The result is the equivalent CTL formula

$$AG(\neg 74\_R\_ACO \to AX(\neg 74\_R\_ACO \to 68\_R\_ACO)) \tag{3}$$

4.4 Motivation for the remainder of this paper

Using the translations described above, the models of stations Hoorn-Kersenboogerd and Heerhugowaard were converted into the input format of RuleBase. The station Hoorn-Kersenboogerd required approximately 200 variables after reduction and was checked relatively easily. However, station Heerhugowaard, which required approximately 600 variables after reduction, showed surprising results. Despite the large size, some formulas were checked easily. However, other formulas suffered from state space explosion during model checking, despite the fact that they induced the exact same model as the easily checked formulas. An investigation into the reasons for this difference led to the work described in this paper.

5 Robustness and locality in symbolic model checking

In this section we will define robustness and locality, and show how these properties aid the model checking of AGAX formulas.

5.1 Robustness

A formula AGf is true in a model if the formula f is true in all reachable states of the model. Informally, a system is robust with respect to a formula AGf if f is true for all states of the model and not only the reachables, or if f is false in the model.

Definition 4 (Robust). *Let $M = (S, S_0, R, L)$ and $M' = (S, S, R, L)$. A model M is robust with respect to a specification f if $M \models f \rightarrow M' \models f$.*

It was observed in the experiments described below that the railway models checked were robust with respect to almost all (47 out of 51) formulas. In the sequel, we will call a model robust when we assume that it is robust with respect to most of the formulas we require to hold.

5.2 Model checking AGAX formulas in robust models

Consider the process of model checking the following AGAX formula in a robust model:

$$AG(a \rightarrow AXb) \tag{4}$$

First, we negate b to get $\neg b$. Then, we take a backward symbolic step to find the set of states which model $EX\neg b$. Finally, we intersect the result with the set of states which model a to get the set of states BAD which models $a \wedge EX\neg b$. There are two cases. Either the intersection of BAD with the reachables is non-empty, and then the original formula is false in the model, or the intersection is empty, and the original formula is true in the model. In order to decide, the model checking algorithm will take backward symbolic steps until either an initial state is seen, or a fixed point is reached.

However, if the model is robust with respect to the formula and the original formula is valid, the set BAD is the empty set and the fixed point calculation will be trivial. In fact, the process of model checking any model which is robust with respect to a valid AGAX formula will consist of a number of backward symbolic steps equal to the depth of the AGAX formula, plus one trivial fixed point calculation.

For a false AGAX formula, the set BAD will not be empty, and the backward symbolic steps will not be trivial. However, the fact that BAD is not empty is itself indicative that the formula is false.

At this point we have an explanation for the behavior observed in Section 4.4. The question was why some formulas model checked easily while others, which induced the same reduction, suffered from state space explosion. The answer is that the formulas which model checked easily were true formulas, with respect to which the model was robust. Thus, the set BAD was empty, and the fixed point calculation was trivial. The formulas which suffered from state space explosion were false formulas, for which the set BAD was not empty.

For a model which is designed to be robust, we have discovered a decision procedure which is faster than full symbolic model checking: simply compare set BAD with the empty set. However, we have still not solved the problem of generating a counter-example for false formulas. We would like to be able to generate a counter-example while avoiding state space explosion. This is the subject of the next subsection.

5.3 Generating counter-examples

One way to generate a counter-example is to notice that in a model which is designed to be robust, it is not necessary to show a counter-example in the original model M. Rather, it is enough for a counter-example to show that the model is not robust, that is, that the formula is false in model M' of definition 4. This can be done without any changes to the symbolic model checker itself, as follows.

Definition 5 (Non-deterministic inputs method). *Code the initial states of the model to be the set of all states.*

In EDL, this is accomplished by coding $assign\ init(v) := \{0,1\}$; for every state variable v in the model.

Now, if the formula is false, the set BAD will be non-empty as before. However, the path from an initial state to BAD will be trivial. If the formula is true, the set BAD will be empty as before, and the fixed point calculation will be trivial.

Possibly (indeed, most probably) the counter-example generated according to the non-deterministic inputs method will not start in a true initial state of model M. However, if the model is intended to be robust, the counter-example generated in this manner is sufficient to show that it is not so, and is therefore useful. In the sequel, we use the following definitions to distinguish between a counter-example generated from model M, and one generated from model M'.

Definition 6 (True counter-example). *A true counter-example is a counter-example which starts in some initial state.*

Definition 7 (Bogus counter-example). *A bogus counter-example is a counter-example which starts in some state which is not an initial state.*

Until we have generated it, there is no useful bound on the length of a true counter-example for most CTL formulas. However, it is easy to see that there exists a bogus counter-example to an AGAX formula with length exactly equal to the depth of the formula plus one. For instance, the formula

$$AGp \tag{5}$$

where p is a propositional formula, has depth 0. A true counter-example will consist of a path from an initial state to some state in which $\neg p$, while a bogus counter-example can consist simply of a state in which $\neg p$, and thus will have length 1. The formula

$$AG(p \rightarrow AXq) \tag{6}$$

where p and q are propositional formulas, has depth 1. A true counter-example will consist of a path from an initial state to some state in which p holds, followed by a state in which $\neg q$, while a bogus counter-example can consist simply of a state in which p holds followed by a state in which $\neg q$, and thus will have length 2.

In the case that a true counter-example in model M is desired, we can make use of locality to generate it as described below.

5.4 Locality

The definition of locality is informal, and leads to a heuristic method of searching for a true counter-example for models which exhibit it.

Definition 8 (Locality). *A model has locality if for every false formula, there exists a counter-example in which most inputs have constant value.*

It was observed in the experiments described below that all counter-examples generated for false formulas exhibited locality. Intuitively, this makes sense for railway models, because the properties which prevent trains from crashing or derailing must be local properties: dependent on only the behavior of "close" tracks or signals. In other words, to push it to the extreme: the fact that two trains do not crash in Amsterdam should not be dependent on the state of a signal in Istanbul. Thus, if a counter-example exists, there should also exist a counter-example in which signals on "far" tracks are quiet.

We have seen previously that if a system is robust, no fixed point calculations are needed to decide on the validity of an AGAX formula. Further, if we are willing to accept a counter-example which is bogus, we can also generate a counter-example with less symbolic steps than needed for a true counter-example. Now we will show how to make use of locality to generate a true counter-example for a robust system when the symbolic steps cause state space explosion.

The method heuristically searches for the inputs which are not needed in the counter-example. It then sets these to 0 and uses pre-model checking reductions [BBEL96] to reduce the size of the model. The method is as follows:

Set half of the inputs to 0, check the formula. Probably the result finds that the formula is true. This is inconclusive, so free up some of the inputs and rerun. If the run does not terminate quickly (i.e., starts a fixed point calculation), then the formula is false even with the inputs chosen set to 0. Choose some more inputs to set to 0 from the half not previously chosen, and so on, until the model is small enough to complete a true counter-example generation.

6 Experimental Results

The experimental results presented below support the observations made above regarding model checking robust models, and demonstrate the effectiveness of the proposed heuristic for counter-example generation of robust models which exhibit locality.

The railway stations were verified in four modes:

1. the properties are checked for all states of the system
2. the properties are checked only for the reachable states of the system
3. the properties are checked for all states of the system with the aid of some simple invariants which are separately checked
4. the properties are checked only for the reachable states of the system with the aid of some simple invariants which are separately checked

The first mode assumes that the systems are robust, while the second conforms more closely to the way the VLC code is used in practice. The third and fourth modes were used as a way to deal with the size problems of the station at Heerhugowaard [Fok99].

6.1 Station Hoorn-Kersenboogerd

Four invariants used by mode 3 were checked in mode 1 with a run time of 9 seconds and memory usage of 33 MB[2]. Thus, the invariants themselves were found to be robust.

Forty-seven formulas were checked for station Hoorn-Kersenboogerd, including all the formulas described in [Fok95], as well as some additional formulas [Gro98]. The formulas included known true and known false formulas which were checked in modes 1, 2 and 3 as described above using the model checker RuleBase [BBEL96]. Results are shown per rule in Table 1 below, where a rule is a group of formulas.

Notice that there is little if any difference in run time between modes 1 and 2 for rules containing only true formulas. This is because the additional fixed point calculation needed by mode 2 is trivial for formulas which are true in mode 1. The additional fixed point calculation can be significant in the case of a false formula. This is especially evident in rules spoor1 and page11. Finally, in all cases, the use of invariants significantly speeded the model checking.

The model is robust with respect to all formulas with the exception of 4 formulas of rule page11. All results agreed with the expected results of [Gro98, Fok95], with one exception. RuleBase found one formula of rule page7 to be false which was found to be true by [Fok95]. After some additional investigation, this formula was confirmed as false by the current version of the tool used in the original work [Gro99]. This formula is now under investigation by Holland RailConsult. This leads to an important conclusion

[2] All statistics shown are for an IBM RS/6000 workstation model 140.

Table 1. Run time and memory usage for Station Hoorn-Kersenboogerd

Rule	Vars	mode 1			mode 2			mode 3		
		run time	memory	false	run time	memory	false	run time	memory	false
spoor1	209	64 s	35 MB	x	766 s	51 MB	x	5 s	36 MB	x
spoor2	209	1 s	31 MB		1 s	31 MB		1 s	32 MB	
spoor3	209	1 s	31 MB		1 s	31 MB		1 s	32 MB	
spoor4	209	1 s	31 MB		1 s	31 MB		1 s	32 MB	
page6	209	28 s	29 MB		37 s	30 MB		1 s	32 MB	
page7	215	468 s	80 MB	x	574 s	198 MB	x	152 s	80 MB	x
page8	215	114 s	42 MB		112 s	47 MB		28 s	43 MB	
page9	215	121 s	43 MB		88 s	45 MB		28 s	44 MB	
page10	215	115 s	43 MB		70 s	51 MB		29 s	46 MB	
page11	215	423 s	43 MB	x	42316 s	216 MB		37 s	36 MB	
page12	215	74 s	33 MB		33 s	31 MB		22 s	34 MB	
page13	215	75 s	33 MB		33 s	31 MB		21 s	34 MB	
page14	209	28 s	29 MB		37 s	31 MB		1 s	32 MB	

of this experiment: for safety critical systems, it is necessary to apply two independent methods of formal verification.

The longest true counter-example was of length 126 cycles.

6.2 Station Heerhugowaard

Forty-five invariants were checked in mode 1 with a run time of 1-2 minutes per invariant. Thus, the invariants for station Heerhugowaard were found to be robust.

Four formulas from [Gro98] were checked for this station. The formulas were checked in modes 1, 2, 3 and 4 as described above using the model checker RuleBase [BBEL96]. Results are shown per rule in Table 2 below, where a rule is a group of formulas.

Table 2. Run time and memory usage for Station Heerhugowaard

Rule	Vars	mode 1			mode 2			mode 3			mode 4		
		run time	memory	false	run time	memory	false	run time	memory	false	run time	memory	false
hh2	594	522 s	162 MB	x	*space out*			778 s	166 MB	x	*space out*		
hh3	594	337 s	157 MB		280 s	144 MB		263 s	149 MB		263 s	151 MB	
hh4	594	416 s	247 MB		287 s	149 MB		268 s	149 MB		277 s	149 MB	
hh5	588	*space out*			*space out*			*space out*			*space out*		

For this station, as for station Hoorn-Kersenboogerd of the previous section, it is apparent that for rules containing only true formulas (hh3 and hh4), there is little difference in run times between modes 1 and 2, because the fixed point calculations are trivial for true formulas in a robust model. In station Heerhugowaard the difficulty of the fixed point calculations for false formulas is evident. Neither of the rules containing false formulas could complete when a fixed point calculation was needed.

The heuristic method described in Section 5.4 was used to create true counter-examples for rules hh2 and hh5 (note that bogus counter-examples were created easily for them, and should be enough to debug a robust model in the usual case). Results are shown in Table 3 below.

Table 3. Run time and memory usage for heuristic generation of counter-example

Rule	Vars (original)	Vars (after heur.)	heur. iterations	run time	memory
hh2	594	189	7	2370 s	89 MB
hh5	588	165	10	51 s	55 MB

For these rules, results agreed with those described in [Gro98].

The length of the longest true counter-example was 124 cycles.

7 Conclusions and future directions

We have seen that symbolic model checking of AGAX formulas can avoid fixed point calculations by making use of the robustness of a model. This includes true formulas and bogus counter-examples for false formulas. In addition, we have seen how locality can be used to heuristically reduce models, thus enabling true counter-example generation even for very large (almost 600 state variables) models.

The use of robustness is not limited to model checking, however. It is also applicable to the use of Stålmarck's method, and can be used to limit the unfolding of the model to the depth of the formula. Indeed, for the work described in [GKV94] and [Fok95], this is exactly what was done [Fok99]. The following calculation shows that it was done out of necessity: The station Heerhugowaard has 4789 triples per cycle. If robustness had not been used, 4789 * 124 = 593836 triples would have been needed to generate the counter-example of length 124. This is above the size of the largest formula reported by [SS98], which was 350000.

Although [BCCZ98] report good results for some hardware models typically difficult for BDD-based methods, for instance multipliers, the above calculations cast doubt on the applicability of Stålmarck's method for verification of typical control intensive hardware. This is because control intensive hardware does not usually exhibit robustness, and a quick calculation indicates that well in excess of 350000 triples would be needed to create the counter-examples we see in our hardware. The intuition that Stålmarck's method performs best for robust models is strengthened by the authors of [BCCZ98] themselves, who, in a new paper [BCRZ99] propose a methodology based on first weeding out the easy cases: those formulas for which the system is robust.

These results are not conclusive, of course. Future work involves experiments in applying Stålmarck's method to large hardware models. Finally, we have seen that for safety critical systems, (at least!) two methods of formal verification should be used.

Acknowledgements

Thanks are due to Jan Friso Groote for supplying the railway models used in this paper, and to Holland RailConsult for agreeing to such. Thank you to Wan Fokkink for supplying the invariants which enabled the verification of station Heerhugowaard. The help of both Jan Friso Groote and Wan Fokkink in investigating the discrepancy of section 6.1 was greatly appreciated. Thank you to Sharon Keidar for implementation of optimizations to RuleBase which solved the size problems for these models. Thank you to Shoham Ben-David for careful review and important comments.

References

[BBEL96] I. Beer, S. Ben-David, C. Eisner, A. Landver, "RuleBase: an Industry-Oriented Formal Verification Tool", in Proc. 33^{rd} Design Automation Conference 1996, pp. 655-660.

[BCCZ98] A. Biere, A. Cimatti, E. Clarke, Y. Zhu, "Symbolic Model Checking without BDDs", in TACAS '99, to appear.

[BCRZ99] A. Biere, E. Clarke, R. Raimi, Y. Zhu, "Verifying Safety Properties of a PowerPC Microprocessor Using Symbolic Model Checking without BDDs", submitted, CAV '99.

[CE81] E.M. Clarke and E.A. Emerson, "Design and synthesis of synchronization skeletons using Branching Time Temporal Logic", in Proc. Workshop on Logics of Programs, Lecture Notes in Computer Science, Vol. 131 (Springer, Berlin, 1981) pp. 52-71.

[CE81b] E.M. Clarke and E.A. Emerson, "Characterizing Properties of Parallel Programs as Fixed-point", in Seventh International Colloquium on Automata, Languages, and Programming, Volume 85 of LNCS, 1981.

[CG+95] E. Clarke, O. Grumberg, K. McMillan, X. Zhao, "Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking", Design Automation Conference 1995, pp. 427-432.

[Fok95] W.J. Fokkink, "Safety criteria for Hoorn-Kersenboogerd Railway Station", Logic Group Preprint Series 135, Utrecht University 1995.

[Fok99] W.J. Fokkink, personal communication to C. Eisner.

[GKV94] J.F. Groote, J.W.C. Koorn and S.F.M. van Vlijmen: "The Safety Guaranteeing System at Station Hoorn-Kersenboogerd." Technical Report 121, Logic Group Preprint Series, Utrecht Univ., 1994.

[Gro98] J.F. Groote, personal communication to C. Eisner.

[Gro99] J.F. Groote, personal communication to C. Eisner.

[McM93] K.L. McMillan, "Symbolic Model Checking", Kluwer Academic Publishers, 1993.

[SS98] M. Sheeran and G. Stålmarck, "A Tutorial on Stålmarck's Proof Procedure for Propositional Logic", in Second International Conference on Formal Methods in Computer-Aided Design, FMCAD '98, Volume 1522 of LNCS, 1998, pp. 82-99.