"Have I Written Enough Properties?" - A Method of Comparison between Specification and Implementation

Sagi Katz[1], Orna Grumberg[1], and Danny Geist[2]

[1] Computer Science Department, Technion, Haifa, Israel
{ksagi,orna}@cs.technion.ac.il
[2] IBM Haifa Research Lab., Haifa, Israel
geist@haifa.vnet.ibm.com

Abstract. This work presents a novel approach for evaluating the quality of the model checking process. Given a model of a design (or implementation) and a temporal logic formula that describes a specification, model checking determines whether the model satisfies the specification. Assume that all specification formulas were successfully checked for the implementation. Are we sure that the implementation is correct? If the specification is incomplete, we may fail to find an error in the implementation. On the other hand, if the specification is complete, then the model checking process can be stopped without adding more specification formulas. Thus, knowing whether the specification is complete may both avoid missed implementation errors and save precious verification time.
The completeness of a specification with respect to a given implementation is determined as follows. The specification formula is first transformed into a tableau. The simulation preorder is then used to compare the implementation model and the tableau model. We suggest four comparison criteria, each revealing a certain dissimilarity between the implementation and the specification. If all comparison criteria are empty, we conclude that the tableau is bisimilar to the implementation model and that the specification fully describes the implementation. We also conclude that there are no redundant states in the implementation.
The method is exemplified on a small hardware example. We implemented our method symbolically as an extension to SMV. The implementation involves efficient OBDD manipulations that reduce the number of OBDD variables from $4n$ to $2n$.

1 Introduction

This work presents a novel approach for evaluating the quality of the model checking process. Given a model of the design (or implementation) and a temporal logic formula that describes a specification, model checking [8,2] determines whether the model satisfies the specification.

L. Pierre and T. Kropf (Eds.): CHARME'99, LNCS 1703, pp. 280–297, 1999.

Assume that all specification formulas were successfully checked for the implementation. Are we sure that the implementation is correct? If the specification is incomplete, we may fail to find an error in the implementation. On the other hand, if the specification is complete, then the model checking process can be stopped without checking additional specification formulas. Thus, knowing whether the specification is complete may both avoid missed implementation errors and save precious verification time.

Below we describe our method to determine whether a specification is complete with respect to a given implementation. We restrict our attention to safety properties written in the universal branching-time logic ACTL [3]. This logic is relatively restricted, but can still express most of the specifications used in practice. Moreover, it can fully characterizes every deterministic implementation. We consider a single specification formula (the conjunction of all properties).

We first apply model checking to verify that the specification formula is true for the implementation model. The formula is then transformed into a *tableau* [3]. By definition, since the formula is true for the model, the tableau is greater by the *simulation preorder* [9] than the model.

We defined a *reduced tableau* for ACTL safety formulas. Our tableau is based on the *Particle tableau* for LTL, presented in [6]. We further reduce their tableau by removing redundant tableau states.

We next use the simulation preorder to find differences between the implementation and its specification. For example, if we find a reachable tableau state with no corresponding implementation state, then we argue that one of the two holds. Either the specification is not restrictive enough or the implementation fails to implement a meaningful state. Our method will not be able to determine which of the arguments is correct. However, the evidence for the dissimilarity (in this case a tableau state that none of the implementation states are mapped to) will assist the designer to make the decision.

We suggest four *comparison criteria*, each revealing a certain dissimilarity between the implementation and specification. If all comparison criteria are empty, we conclude that the tableau is bisimilar to the implementation model and that the specification fully describes the implementation. We also conclude that there are no redundant states in the implementation.

The practical aspects of this method are straightforward. Model checking activity in industry executes the following methodology: A verification engineer reads the specification, sets up a work environment and then proceeds to present the model checker with a sequence of properties in order to verify the design correctness [1]. The design (or implementation) on which this activity is executed can be quite large nowadays. As a result the set of properties written and verified becomes large as well, to the point that the engineer loses control over it.

A large property set makes it necessary to construct tools to evaluate its overall quality. The basic question to answer is: "Have I written enough properties?". The current solution is to manually review the property set. However, this solution is not scalable and furthermore since it is done manually it makes the description of model checking as "formal verification" imprecise. This in-

adequate solution indicates a growing need for tools that may be able to tell the engineer when the design is "bug-free" and therefore cut down development time.

Quality evaluation of verification activity is not new. Traditional verification methods have developed measurement criteria to measure the quality of test suites [11]. This area of verification is called *coverage*. The notion of coverage in model checking is to have a specification that covers the entire functionality required from the implementation. This can be divided into two questions:

1. Whether the environment is rich enough to provide all possible input sequences.
2. Whether the specification contains a sufficient set of properties.

The method we present addresses both problems as will be later shown by an example.

We compared a small hardware example with the reduced tableau. For the complete specification formula we received a reduced tableau with 20 states. The tableau presented in [3] would have a state space of 2^{15} states for this formula. It is interesting to note that in this example not all the implementation variables are observable.

We implemented our method symbolically as an extension to the symbolic model checker SMV [7]. Given a model with n state variables, a straightforward implementation of this method can create intermediate results that consists of $4n$ OBDD variables. However, our implementation reduces the required number of OBDD variables from $4n$ to $2n$.
The main contributions of our paper can be summarized as follows:

- We suggest for the first time a theoretical framework that provides quality evaluation for model checking. The suggested comparison criteria can assist the designer in finding errors in the design by indicating points in which the design and the specification disagree, and suggest criteria for terminating the verification effort.
- We implemented our method symbolically within SMV. Thus, it can be invoked automatically once the model checking terminates successfully.
  Of a special interest is the symbolic computation of the simulation relation for which no good symbolic algorithm is known.
- We defined a new reduced tableau for ACTL that is often significantly smaller in the number of states and transitions than known tableaux for ACTL.

In these days, another work on coverage of model checking has been independently developed [5]. The work computes the percentage of states in which a change in an observable proposition will not affect the correctness of the specification. Their evidence is closely related to our criterion of *Unimplemented State*. In their paper they list a number of limitations of their work. They are unable to give path evidence, cannot point out functionality missing in the model, and they have no indication that the specification is complete. In the conclusion we explain how our work solves these problems.

The analysis we perform compares the two models and tries to identify dissimilarities. It is therefore related to tautology checking of finite state machines as is done in [10]. However the method in [10] is suggested as an alternative to model checking and not as a complementary method.

The rest of this paper is organized as follows. Section 2 gives the necessary background. Section 3 describes the comparison criteria and the method for their use. Section 4 exemplifies the different criteria by applying the method to a small hardware circuit. Section 5 presents symbolic algorithms that implement our method. In Section 6 we discuss the reduced tableau for ACTL safety formulas. Finally, the last section describes future work and concludes the paper.

2 Preliminaries

Our specification language is the universal branching-time temporal logic ACTL [3], restricted to safety properties. Let AP be a set of atomic propositions. The set of ACTL *safety formulas* is defined inductively in negation normal form, where negations are applied only to atomic propositions. It consists of the temporal operators **X** ("next-state") and **W** ("weak until") and the path quantifier **A** ("for all paths").

- If $p \in AP$ then both p and $\neg p$ are ACTL safety formulas.
- If φ_1 and φ_2 are ACTL safety formulas then so are $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\mathbf{AX}\varphi_1$, and $\mathbf{A}[\varphi_1 \mathbf{W} \varphi_2]$[1].

We use Kripke structures to model our implementations. A *Kripke structure* is a tuple $M = (S, S_0, R, L)$ where S is a finite set of states; $S_0 \subseteq S$ is the set of initial states; $R \subseteq S \times S$ is the transition relation that must be total; and $L : S \to 2^{AP}$ is the labeling function that maps each state to the set of atomic propositions true at that state.

A *path* in M from a state s is a sequence $s_0, s_1, \ldots$ such that $s_0 = s$ and for every i, $(s_i, s_{i+1}) \in R$.

The logic ACTL is interpreted over a state s in a Kripke structure M. The formal definition is omitted here. Intuitively, $\mathbf{AX}\varphi_1$ is true in s if all its successors satisfy φ_1. $\mathbf{A}[\varphi_1 \mathbf{W} \varphi_2]$ is true in s if along every path from s, either φ_1 holds forever or φ_2 eventually holds and φ_1 holds up to that point. We say that a structure M satisfies a formula φ, denoted $M \models \varphi$, if every initial state of M satisfies φ.

Let $M = (S, S_0, R, L)$ and $M' = (S', S'_0, R', L')$ be two Kripke structures over the same set of atomic propositions AP. A relation $SIM \subseteq S \times S'$ is a *simulation preorder* from M to M' [9] if for every initial state s_0 of M there is an initial state s'_0 of M' such that $(s_0, s'_0) \in SIM$. Moreover, if $(s, s') \in SIM$ then the following holds:

- $L(s) = L'(s')$, and
- $\forall s_1[(s, s_1) \in R \Longrightarrow \exists s'_1[(s', s'_1) \in R' \wedge (s_1, s'_1) \in SIM]]$.

[1] Full ACTL includes also formulas of the form $\mathbf{A}[\varphi_1 \mathbf{U} \varphi_2]$ ("strong until").

If there is a simulation preorder from M to M', we write $M \leq M'$ and say that M *simulates* M'.

It is well known [3] that if $M \leq M'$ then for every ACTL formula φ, if $M' \models \varphi$ then $M \models \varphi$. Furthermore, for every ACTL safety formula ψ it is possible to construct a Kripke structure $T(\psi)$, called a *tableau*[2] for ψ, that has the following *tableau properties* [3].

- $T(\psi) \models \psi$.
- For every structure M, $M \models \psi \iff M \leq T(\psi)$.

Intuitively, the simulation preorder relates two states if the computation tree starting from the state of the smaller model can be embedded in the computation tree starting from the state of the greater one. This, however, is not sufficient in order to determine how similar the two structures are. Instead, we use the *reachable simulation preorder* that relates two states if they are in the simulation preorder and are also reachable from initial states along corresponding paths.

Formally, let $SIM \subseteq S \times S'$ be the *greatest* simulation preorder from M to M'. The *reachable simulation preorder* for SIM, $ReachSIM \subseteq SIM$, is defined by: $(s, s') \in ReachSIM$ if and only if there is a path $\pi = s_0, s_1, \ldots, s_k$ in M with $s_0 \in S_0$ and $s_k = s$ and a path $\pi' = s'_0, s'_1, \ldots, s'_k$ in M' with $s'_0 \in S'_0$ and $s'_k = s'$ such that for all $0 \leq j \leq k$, $(s_j, s'_j) \in SIM$.
In this case, the paths π and π' are called *corresponding paths* leading to s and s'.

Lemma 1. *$ReachSIM$ is a simulation preorder from M to M'.*

The proof of the lemma is postponed to Appendix A.

3 Comparison Criteria

Let $M = (S_i, S_{0i}, R_i, L_i)$ be an implementation structure and $T(\psi) = (S_t, S_{0t}, R_t, L_t)$ be a tableau structure over a common set of atomic propositions AP. For the two structures we consider only reachable states that are the start of an infinite path.

Assume $M \leq T(\psi)$. We define four criteria, each is associated with a set. A criterion is said to hold if the appropriate set is empty. For convenience we name each criterion the same as the appropriate set. The following sets define the criteria :

1. $UnImplementedStartState = \{s_t \in S_{0t} \,|\, \forall s_i \in S_{0i} \,[\,(s_i, s_t) \notin ReachSIM\,]\}$
   An Unimplemented Start State is an initial tableau state that has no corresponding initial state in the implementation structure. The existence of such a state may indicate that the specification does not properly constrain the set of start states. It may also indicate the lack of a required initial state in the implementation.

[2] The tableau for full ACTL is a *fair* Kripke structure (not defined here). It has the same properties except that $\models$ and $\leq$ are defined for fair structures.

2. $UnImplementedState = \{s_t \in S_t \,|\, \forall s_i \in S_i\, [\,(s_i, s_t) \notin ReachSIM\,]\}$
   An Unimplemented State is a state of the tableau that has no corresponding state in the implementation structure. This difference may suggest that the specification is not tight enough, or that a meaningful state was not implemented.
3. $UnImplementedTransition = \{(s_t, s'_t) \in R_t \,|\, \exists s_i, s'_i \in S_i,$
   $[\,(s_i, s_t) \in ReachSIM$, $(s'_i, s'_t) \in ReachSIM$ and $(s_i, s'_i) \notin R_i\,]\}$
   An Unimplemented Transition is a transition between two states of the tableau, for which a corresponding transition in the implementation does not exist. The existence of such a transition may suggest that the specification is not tight enough, or that a required transition (between reachable implementation states) was not implemented.
4. $ManyToOne = \{s_t \in S_t \,|\, \exists s_{1i}, s_{2i} \in S_i\, [\,(s_{1i}, s_t) \in ReachSIM, (s_{2i}, s_t) \in ReachSIM$ and $s_{1i} \neq s_{2i}\,]\}$
   A Many To One state is a tableau state to which multiple implementation states are mapped. The existence of such a state may indicate that the specification is not detailed enough. It may also suggest that the implementation contains redundancy.

Our criteria are defined for any tableau that has the tableau properties as defined in Section 2. Any dissimilarity between the implementation and the specification will result in a non empty criterion. Empty criteria indicate completeness, but they are hard to obtain on traditional tableaux since such tableaux contain redundancies. In the reduced tableau presented in Section 6, redundancies are removed and therefore empty criteria are more likely to be achieved.
Given a structure M and a property ψ our method consists of the following steps:

1. Apply model checking to verify that $M \models \psi$.
2. Build a (reduced) tableau $T(\psi)$ for ψ.
3. Compute SIM of $(M, T(\psi))$.
4. Compute $ReachSIM$ of $(M, T(\psi))$ from SIM of $(M, T(\psi))$.
5. For each of the comparison criteria, evaluate if its corresponding set is empty and if not present evidence for its failure.

Theorem 2. *Let M be an implementation model and ψ be an ACTL safety formula such that $M \models \psi$. Let $T(\psi)$ be a tableau for ψ that has the tableau properties. If the comparison criteria 1-3 hold then $T(\psi) \leq M$.*

The proof of this theorem is left to Appendix B. The proof implies that if criteria 1-3 hold then $T(\psi)$ and M are in fact *bisimilar*. The fourth criterion is not necessary for completeness since whenever there are several non-bisimilar implementation states that are mapped to the same tableau state, then there is also an unimplemented state or transition. However, this criterion may reveal redundancies in the implementation.

It is important to note that the goal is not to find a smaller set of criteria that guarantees the specification completeness. The purpose of the criteria is to

assist the designer in the debugging process. Thus, we are looking for meaningful criteria that can distinguish among different types of problems and identify them. In Section 6 we define an additional criterion that can reveal redundancy in the specification.

4 Example

Consider a synchronous arbiter with two inputs, $req0, req1$ and two outputs $ack0, ack1$. The assertion of ack_i is a response to the assertion of req_i. Initially, both outputs of the arbiter are inactive. At any time, at most one acknowledge output may be active. The arbiter grants one of the active requests in the next cycle, and uses a round robin algorithm in case both request inputs are active. Furthermore in the case of *simultaneous assertion* (i.e. both requests are asserted and were not asserted in the previous cycle), request 0 has priority in the first *simultaneous assertion* occurrence. In any additional occurrence of *simultaneous assertion* the priority rotates with respect to the previous occurrence.
The implementation and the specification will share a common set of atomic propositions $AP = \{req0, req1, ack0, ack1\}$. An implementation of the arbiter M, written in the SMV language is presented below:

```
1)  var
2)    req0, req1, ack0, ack1, robin : boolean;
3)  assign
4)    init(ack0) := 0;
5)    init(ack1) := 0;
6)    init(robin) := 0;
7)  next(ack0) := case
8)    !req0                 : 0;        – No request results no ack
9)    !req1                 : 1;        – A single request
10)   !ack0 & !ack1         :!robin;    – Simultaneous requests assertions
11)   1                     :!ack0;     – Both requesting , toggle ack
12) esac;
13) next(ack1) := case
14)   !req1                 : 0;        – No request results no ack
15)   !req0                 : 1;        – A single request
16)   !ack0 & !ack1         : robin;    – simultaneous assertion
17)   1                     :!ack1;     – Both requesting , toggle ack
18) esac;
19) next(robin) := if req0 & req1 &!ack0 & !ack1 then !robin
20)                    else robin endif;    – Two simultaneous request
                                              assertions
```

From the verbal description given at the beginning of the section, one may derive a temporal formula that specifies the arbiter :

$$
\begin{array}{lll}
\psi = \neg ack0 \wedge \neg ack1 \wedge & & \\
\quad \mathbf{A}[(\neg req0 \vee \neg req1 \vee ack0 \vee ack1)\mathbf{W} & & \\
\quad\quad (req0 \wedge req1 \wedge \neg ack0 \wedge \neg ack1 \wedge \mathbf{AX} ack0)] & \wedge & -\varphi_0 \\
\quad \mathbf{AG}(& & \\
\quad\quad (\neg ack0 \vee \neg ack1) & \wedge & -\varphi_1 \\
\quad\quad (\neg req0 \wedge \neg req1 \rightarrow \mathbf{AX}(\neg ack0 \wedge \neg ack1)) & \wedge & -\varphi_2 \\
\quad\quad (req0 \wedge \neg req1 \rightarrow \mathbf{AX} ack0) & \wedge & -\varphi_3 \\
\quad\quad (\neg req0 \wedge req1 \rightarrow \mathbf{AX} ack1) & \wedge & -\varphi_4 \\
\quad\quad (req1 \wedge ack0 \rightarrow \mathbf{AX} ack1) & \wedge & -\varphi_5 \\
\quad\quad (req0 \wedge ack1 \rightarrow \mathbf{AX} ack0) & \wedge & -\varphi_6 \\
\quad\quad (req0 \wedge req1 \wedge \neg ack0 \wedge \neg ack1 \rightarrow \mathbf{AX}(ack0 \rightarrow & & \\
\quad\quad \mathbf{A}[(\neg req0 \vee \neg req1 \vee ack0 \vee ack1)\mathbf{W} & & \\
\quad\quad\quad (req0 \wedge req1 \wedge \neg ack0 \wedge \neg ack1 \wedge \mathbf{AX} ack1)])) & \wedge & -\varphi_7 \\
\quad\quad (req0 \wedge req1 \wedge \neg ack0 \wedge \neg ack1 \rightarrow \mathbf{AX}(ack1 \rightarrow & & \\
\quad\quad \mathbf{A}[(\neg req0 \vee \neg req1 \vee ack0 \vee ack1)\mathbf{W} & & \\
\quad\quad\quad (req0 \wedge req1 \wedge \neg ack0 \wedge \neg ack1 \wedge \mathbf{AX} ack0)])) &) & -\varphi_8
\end{array}
$$

where $\mathbf{AG}\varphi \equiv \mathbf{A}[\varphi \mathbf{W} false]$. We verified that $M \models \psi$ using the SMV model checker. We then applied our method. We found that all comparison criteria hold. We therefore concluded that ψ is a complete specification for M.
In order to exemplify the ability of our method we changed the implementation and the specification in different ways. In all cases the modified implementation satisfied the modified specification. However, our method reported the failure of some of the criteria. By examining the evidence supplied by the report, we could detect flaws in either the implementation or the specification.

4.1 Unimplemented Transition Evidence

Consider a modified version of the implementation M, named M_{trans} obtained by adding the line : $robin \,\&\, ack1 : \{0, 1\}$;
between line (10) and line (11), and by adding the line :
$ack1 : !next(ack0)$; between line (16) and line (17).
Consider also the modified formula ψ_{trans} obtained from ψ by replacing φ_6 with:
$(req0 \wedge ack1 \rightarrow \mathbf{AX}(ack0 \vee ack1)) \wedge$
SMV shows that $M_{trans} \models \psi_{trans}$. However, applying the comparison method on M_{trans} and ψ_{trans}, reports an *Unimplemented Transition*. It supplies as an evidence a transition between tableau states s_t and s'_t such that $L_t(s_t) = L_t(s'_t) = \{req0, req1, !ack0, ack1\}$. Such a transition is possible by ψ_{trans} but not possible in M_{trans} in case variable *robin* is not asserted.

If we check the reason for the incomplete specification we note that the evidence shows a cycle with $req0$ and $ack1$ asserted followed by a cycle were $ack1$ is asserted. This ill behavior violates the round robin requirement. The complete specification would detect that M_{trans} has a bug, since $M_{trans} \not\models \psi$.

4.2 Unimplemented State Evidence

Consider a modified version of the implementation M, named M_{unimp} obtained by adding line : $ack0 : \{0,1\}$;
between lines (10) and line (11), and replacing line (2) with the following lines:
2.1) $req0\_temp, req1, ack0, ack1, robin : boolean$;
2.2) $define\ req0 := req0\_temp\,\&\,!(ack0\,\&\,ack1)$;
Here $req0\_temp$ is a free variable, and the input $req0$ is a restricted input such that if the state satisfies $ack0\&ack1$ then $req0$ is forced to be inactive.
Consider also the modified formula ψ_{unimp} obtained from ψ by deleting φ_1. SMV shows that $M_{unimp} \models \psi_{unimp}$. However, applying the comparison method on M_{unimp} and ψ_{unimp}, reports an *Unimplemented State*. It supplies as an evidence the state s_t such that $L_t(s_t) = \{req0, !req1, ack0, ack1\}$. This state is possible by ψ_{unimp} but not possible in M_{unimp}.
If we check the source of the incomplete specification we note that the evidence violates the mutual exclusion property. Both of the arbiter outputs $ack0$ and $ack1$ are active. The complete specification would detect that M_{unimp} has a bug, since $M_{unimp} \not\models \psi$.
Note that in this example we can also identify that $req0$ in M_{unimp} is a restricted input relative to the formula ψ_{unimp}. The state space of M_{unimp} does not include the states $\{req0, req1, ack0, ack1\}$ or $\{req0, !req1, ack0, ack1\}$. A restricted environment may hide bugs, so this is just as important as finding missing properties.

4.3 Many To One Evidence

A nonempty *Many To One* criterion may imply one of two cases. Redundant implementation, or incompleteness. The latter case is always accompanied with one of criteria 1-3. The former case where criteria 1-3 hold but we have a *Many To One* evidence implies that the implementation is complete with respect to the specification, but it is not efficient and contains redundancies. There is a smaller implementation that can preserve the completeness. This information may give insight on the efficiency of the implementation.
The following implementation M_{m2o} uses 5 implementation variables and two free inputs instead of 3 variables and two inputs of implementation M. Criteria 1-3 are met for M_{m2o} with respect to ψ.

1) var
2) $\quad req0, req1, req0q, req1q, ack0q, ack1q, robin : boolean$;
3) $assign$
4) $\quad init(req0q) := 0$; $init(req1q) := 0$;
5) $\quad init(ack0q) := 0$; $init(ack1q) := 0$;
6) $\quad init(robin) := 1$;
7) $define$
8) $\quad ack0 := case$

```
9)    !req0q                          : 0;        – No request results no ack
10)   !req1q                          : 1;        – A single request
11)   !ack0q & !ack1q                 : !robin;   – Simultaneous requests assertions
12)   1                               : !ack0q;   – Both requesting , toggle ack
13)  esac;
14)  ack1 := case
15)   !req1q                          : 0;        – No request results no ack
16)   !req0q                          : 1;        – A single request
17)   !ack0q & !ack1q                 : robin;    – simultaneous assertion
18)   1                               : !ack1q;   – Both requesting , toggle ack
19)  esac;
20) assign
21)  next(robin) := if req0 & req1 &!ack0 & !ack1 then !robin
22)    else robin endif;                          – Two simultaneous request assertions
23)  next(req0q) := req0; next(req1q) := req1;
24)  next(ack0q) := ack0; next(ack1q) := ack1;
```

Applying model checking will show that $M_{m2o} \models \psi$.
In the above example we keep information of the current inputs $req0$ and $req1$, as well as their value in the previous cycle (i.e. $req0q$ and $req1q$). Intuitively, this duplicates each state in M to four states in the state space of M_{m2o}.

4.4 Unimplemented Start State Evidence

The *Unimplemented Start State* criterion does not hold when the specification is not restricted to the valid start states. Consider a specification formula obtained from ψ by removing the φ_0 subformula. Applying the comparison method on M and the modified formula would yield a *Unimplemented Start State* evidence of a tableau state s_{0t} such that $\{ack0, !ack1\} \subseteq L_t(s_{0t})$. Restricting the specification to the valid start states would cause the *Unimplemented Start State* criteria to hold.

4.5 Non-observable Implementation Variables

As can be seen in this example, a state of the implementation is not uniquely determined by $req0$, $req1$, $ack0$ and $ack1$. The variable *robin* effects the other variables, but it is a non-observable intermediate variable. This variable is not explicitly described in the specification, and does not appear in the common set of atomic propositions AP, referred to by the simulation preorder. Our criteria are defined with respect to observable variables only, but are not limited to systems where all the variables are observable.

5 Implementation of the Method

5.1 Symbolic Algorithms

In this section we present the symbolic algorithms that implement various parts of our method. In particular, we show how to compute symbolically the simulation relation. Our implementation will require less memory than the naive implementation since we reduce the number of OBDD variables. In Section 5.2 we show how this is achieved.

For conciseness, we use $R(s, s')$, $S(s)$ etc. instead of $(s, s') \in R$, $s \in S$.

Computing *SIM*: Let $M = (S_i, S_{0i}, R_i, L_i)$ be the implementation structure and let $T(\psi) = (S_t, S_{0t}, R_t, L_t)$ be a tableau structure. The following pseudo-code depicts the algorithm for computing SIM:
Init: $SIM_0(s_i, s_t) := \{ (s_i, s_t) \in S_i \times S_t \mid L_i(s_i) = L_t(s_t) \}$; $j := 0$
Repeat {
$SIM_{j+1} := \{ (s_i, s_t) \mid \forall s'_i [R_i(s_i, s'_i) \rightarrow \exists s'_t [R_t(s_t, s'_t) \wedge SIM_j(s'_i, s'_t)]]$
$\wedge\ SIM_j(s_i, s_t)\}$ $j := j + 1$ } *until* $SIM_j = SIM_{j-1}$
$SIM := SIM_j$

Computing *ReachSIM*: Given the simulation relation SIM of the pair $(M, T(\psi))$ the following pseudo-code depicts the algorithm for computing $ReachSIM$:
Init: $ReachSIM_0 := (S_{0i} \times S_{0t}) \cap SIM$; $j := 0$
Repeat {
$ReachSIM_{j+1} := ReachSIM_j \cup$
$\{ (s'_i, s'_t) \mid \exists s_i, s_t (ReachSIM_j(s_i, s_t) \wedge R_i(s_i, s'_i) \wedge R_t(s_t, s'_t) \wedge SIM(s'_i, s'_t)) \}$
$j := j + 1$ } *until* $ReachSIM_j = ReachSIM_{j-1}$
$ReachSIM := ReachSIM_j$

5.2 Efficient OBDD Implementation

We now turn our attention to improving the performance of the algorithms described in the previous section. We assume that an implementation of such an algorithm will be done within a symbolic model checker such as SMV [7]. Since formal analysis always suffers from state explosion it is necessary to find methods to efficiently utilize computer memory. When working with OBDDs one possible way to do so is to try to minimize the number of OBDD variables that any OBDD created during the computation will have.

We can see from the algorithms presented before that some of the sets, constructed in intermediate computation steps, are defined over four sets of states: implementation states, specification states, tagged (next) implementation states, and tagged (next) specification states. For example, the computation of SIM_{j+1} is defined by means of the implementation states s_i, specification states s_t, tagged

implementation states s'_i (representing implementation next states), and tagged specification states s'_t (representing specification next states).

Assume that we need at most n bits to encode each set of states. Then potentially some of the OBDDs created in the intermediate computations will have $4n$ OBDD variables. However, by breaking the algorithm operations to smaller ones and manipulating OBDDs in a nonstandard way we managed to bound the number of variables of the OBDDs created in intermediate computations by $2n$.

We define two operations, *compose* and *compose_odd*, that operate on two OBDDs a and b over a total number of $3n$ variables. As explained later, the main advantage of these operations is that they can be implemented using only $2n$ OBDD variables.

$$compose(\boldsymbol{y}, \boldsymbol{u}) \equiv \exists \boldsymbol{x}(a(\boldsymbol{x}, \boldsymbol{y}) \wedge b(\boldsymbol{x}, \boldsymbol{u})) \quad (1)$$

$$compose\_odd(\boldsymbol{y}, \boldsymbol{u}) \equiv \exists \boldsymbol{x}(a(\boldsymbol{y}, \boldsymbol{x}) \wedge b(\boldsymbol{u}, \boldsymbol{x})). \quad (2)$$

SIM and *ReachSIM* can be implemented using *compose* and *compose_odd* as follows. Let $\boldsymbol{v_i}$, $\boldsymbol{v'_i}$ be the encoding of the states s_i, s'_i respectively. Similarly, let $\boldsymbol{v_t}$, $\boldsymbol{v'_t}$ be the encoding of s_t, s'_t respectively.

$$SIM_{j+1}(\boldsymbol{v_i}, \boldsymbol{v_t}) := SIM_j(\boldsymbol{v_i}, \boldsymbol{v_t}) \wedge \neg compose\_odd(R_i(\boldsymbol{v_i}, \boldsymbol{v'_i}), \neg compose\_odd(R_t(\boldsymbol{v_t}, \boldsymbol{v'_t}), SIM_j(\boldsymbol{v'_i}, \boldsymbol{v'_t})))$$

$$ReachSIM_{j+1}(\boldsymbol{v'_i}, \boldsymbol{v'_t}) := ReachSIM_j(\boldsymbol{v'_i}, \boldsymbol{v'_t}) \vee (compose(compose(ReachSIM_j(\boldsymbol{v_i}, \boldsymbol{v_t}), R_i(\boldsymbol{v_i}, \boldsymbol{v'_i})), R_t(\boldsymbol{v_t}, \boldsymbol{v'_t})) \wedge SIM(\boldsymbol{v'_i}, \boldsymbol{v'_t}))$$

The derivation of these expressions can be found in Appendix C. The algorithms above require that the implementation and specification "step" together along the transition relation. We break this to stepping along one, followed by stepping along the other. This is possible since transitions of the two structures are independent.

The comparison criteria *Unimplemented Transition* and *Many To One* can also be implemented with these operations. The two other criteria are defined over $2n$ variables and do not require such manipulation.

$$UnimplementedTransition(\boldsymbol{v_t}, \boldsymbol{v'_t}) := R_t(\boldsymbol{v_t}, \boldsymbol{v'_t}) \wedge compose(compose(\neg R_i(\boldsymbol{v_i}, \boldsymbol{v'_i}), ReachSIM(\boldsymbol{v_i}, \boldsymbol{v_t})), ReachSIM(\boldsymbol{v'_i}, \boldsymbol{v'_t}))$$

$$ManyToOne(\boldsymbol{v_t}) := \exists \boldsymbol{v_1}(ReachSIM(\boldsymbol{v_1}, \boldsymbol{v_t}) \wedge compose((\boldsymbol{v_1} \neq \boldsymbol{v_2}), ReachSIM(\boldsymbol{v_2}, \boldsymbol{v_t})))$$

The details of these derivations can be found in Appendix C.

Up to now we showed how to reduce the number of OBDD variables from $4n$ to $3n$. We now show how to further reduce this number to $2n$. Our first step is to use the same OBDD variables to represent the implementation variables $\boldsymbol{v_i}$ and the specification variables $\boldsymbol{v_t}$. These OBDD variables will be referred to as *untagged*. Similarly, we use the same OBDD variables to represent $\boldsymbol{v'_i}$ and $\boldsymbol{v'_t}$. They will be referred to as *tagged* OBDD variables.

We also specify that whenever we have relations over both implementation variables and specification variables then the implementation variables are represented by untagged OBDD variables while the specification variables are represented by tagged OBDD variables. Note that now the relations R_i, R_t, SIM, $ReachSIM$ are all defined over the same sets of OBDD variables. Consequently, in all the derived expressions we apply *compose* and *compose_odd* to OBDDs that share variables, i.e. $\boldsymbol{y}$ and $\boldsymbol{u}$ are represented by the same OBDD variables. The implementation of *compose* and *compose_odd* uses non-standard OBDD operations in such a way that the resulting OBDDs are also defined over the same $2n$ variables.

Notice that this requires that the OBDD variable change semantics in the result (e.g., in Equation 1 $\boldsymbol{y}$ is represented by tagged OBDD variables in the input parameters and by untagged variables in the result). OBDD packages can easily be extended with these operations.

6 Reduced Tableau and Redundancies in Specification

6.1 Smaller Tableau Structure

When striving for completeness, the size of tableau structures as defined in [3] is usually too large to be practical, and may be much larger than the state space of the given implementation. This is because the state space of such tableaux contain all combinations of subformulas of the specification formula. Such tableaux usually contain many redundant states, that can be removed while preserving the tableau properties. If not removed, these states may introduce evidences which are not of interest.

Much of the redundancies can be eliminated if each state contains *exactly* the set of formulas required for satisfying the specification formula. Consider for example the ACTL formula $\mathbf{AXAX}p$. Its set of subformulas is $\{\mathbf{AXAX}p, \mathbf{AX}p, p\}$. We desire a tableau structure in which each state contains only the set of subformulas required to satisfy the formula. In this case, the initial state should satisfy $\mathbf{AXAX}p$, its successor should satisfy $\mathbf{AX}p$ and its successor should satisfy p. In each of these states all unmentioned subformulas have a "don't care" value. Thus, one state of the reduced tableau represents many states. For instance, the initial state $\{\mathbf{AXAX}p\}$ represents four initial states in the traditional tableau [3]. In such examples we may get a linear size tableau instead of an exponential one.

Following the above motivation, the reduced tableau will be defined over a *3-value labeling* for atomic propositions, i.e., for an atomic proposition p, a state may be labeled by either p, $\neg p$ or neither of them. Also, only the reachable portion of the structure will be constructed.

Further reduction may be obtained if the set of successors for each state is constructed more carefully. If a state s has two successors s' and s'', such that the set of formulas of s'' is contained in the set of formulas of s', then s' is not constructed. Any tableau behavior starting at s' has a corresponding behavior from s''. Thus, it is unnecessary to include both.

Given an ACTL safety formulas, the definition of the reduced tableau is derived from the Particle tableau for LTL, presented in [6] by replacing the use of the **X** temporal operator by **AX**. Since the only difference between LTL and ACTL is that temporal operators are always preceded by the universal path quantifier, this change is sufficient. In general, we will obtain a smaller tableau since we also avoid the construction of redundant successors.

Since the reduced tableau is based on the 3-value labeling, the definition of satisfaction and simulation preorder are changed accordingly. Our reduced tableau $T(\psi)$ for ACTL then has the same properties as the one in [3]:

- $T(\psi) \models \psi$.
- For every Kripke structure M, $M \leq T(\psi)$ if and only if $M \models \psi$.

Note that adopting the reduced tableau also requires modifications to our criteria due to the 3-value labeling semantics.

6.2 Reduced Tableau Results

We have defined the reduced tableau and proved its tableau properties. In addition we have adapted the comparison criteria to comply with the *3-value labeling*. We also coded the reduced tableau construction and the comparison criteria into the SMV model checker, performing the structure comparison in a symbolic manner.

We have run the arbiter example of Section 4 with the reduced tableau. For the complete specification formula ψ presented there we received a structure with 20 states. A traditional tableau structure would have a state space of 2^{15} states for ψ.

6.3 Identifying Redundancies in the Specification

Section 3 defines criteria that characterize when a specification is rich enough (i.e., complete). We would like also to determine whether a complete specification contains redundancies, i.e., subformulas that can be removed or be rewritten without destroying the completeness of the specification.

Given the reduced tableau, we suggest a new criterion, called *One To Many*, that identifies implementation states that are mapped (by $ReachSIM$) to multiple tableau states. Finding such states means that there is a smaller structure that corresponds to an equivalent specification formula. The criterion $OneToMany$ is defined by:
$OneToMany = \{s_i \in S_i \mid \exists s_{1t}, s_{2t} \in S_t[(s_i, s_{1t}) \in ReachSIM \wedge (s_i, s_{2t}) \in ReachSIM \wedge s_{1t} \neq s_{2t}]\}$.

6.4 One To Many Example

The following example demonstrates the One to Many criterion. It identifies a redundant sub formula, which does not add to the completeness of the specification formula. Consider the following specification formula :

$$
\begin{array}{lll}
\psi_{One2Many} = & & \\
\quad \neg ack0 \wedge \neg ack1 & \wedge & \\
\quad \mathbf{A}[(\neg req0 \vee \neg req1 \vee ack0 \vee ack1)\mathbf{W} & & \\
\qquad (req0 \wedge req1 \wedge \neg ack0 \wedge \neg ack1 \wedge \mathbf{AX}ack0)] \wedge & & -\ \varphi_0 \\
\quad \mathbf{AG}(& & \\
\qquad (\neg ack0 \vee \neg ack1) & \wedge & -\ \varphi_1 \\
\qquad (\neg req0 \wedge \neg req1 \wedge \mathbf{AX}(\neg ack0 \wedge \neg ack1) & \vee & -\ \varphi_2 \\
\qquad req0 \wedge \neg req1 \wedge \mathbf{AX}ack0 & \vee & -\ \varphi_3 \\
\qquad \neg req0 \wedge req1 \wedge ack1 \wedge \mathbf{AX}ack1 & \vee & -\ \varphi_4 \\
\qquad req0 \wedge req1 \wedge ack0 \wedge \mathbf{AX}ack1 & \vee & -\ \varphi_5 \\
\qquad req0 \wedge req1 \wedge ack1 \wedge \mathbf{AX}ack0 & \vee & -\ \varphi_6 \\
\qquad req1 \wedge ack1 \wedge \mathbf{AX}(ack0 \wedge \neg ack1) & \vee & -\ \varphi_{redundant} \\
\qquad req0 \wedge req1 \wedge \neg ack0 \wedge \neg ack1 \wedge \mathbf{AX}(& & \\
\qquad ack0 \wedge \mathbf{A}[(\neg req0 \vee \neg req1 \vee ack0 \vee ack1)\mathbf{W} & & \\
\qquad (req0 \wedge req1 \wedge \neg ack0 \wedge \neg ack1 \wedge \mathbf{AX}ack1)] \vee & & \\
\qquad ack1 \wedge \mathbf{A}[(\neg req0 \vee \neg req1 \vee ack0 \vee ack1)\mathbf{W} & & \\
\qquad (req0 \wedge req1 \wedge \neg ack0 \wedge \neg ack1 \wedge \mathbf{AX}ack0)])))) & & -\ \varphi_7
\end{array}
$$

Our method reported that for $\psi_{One2Many}$ criteria 1-4 are met. In addition, it reported that the *One To Many* criterion is not met. As an evidence it provides the implementation state s_i such that $L_i(s_i) = \{req0, req1, \neg ack0, ack1\}$. This state is mapped to s_{1t} and s_{2t} of the reduced tableau for which $L_t(s_{1t}) = \{req0, req1, \neg ack0, ack1\}$ and $L_t(s_{2t}) = \{req1, \neg ack0, ack1\}$.
We may note that $\varphi_{redundant}$ sub formulas agrees with φ_6 for states labeled with $\{req0, req1\}$, and does not agree with φ_4 for states labeled with $\{\neg req0, req1\}$. Since it comes as a disjunct, it does not limit the reachable simulation, and does not add allowed behavior. Deleting sub formula $\varphi_{redundant}$ leaves a specification formula such that criteria 1-4 are met and the *One to Many* criterion is also met.

7 Future Work

In this paper we presented a novel approach for evaluating the quality of the model checking process. The method we described can give an engineer the confidence that the model is indeed "bug-free" and reduce the development time.

We are aware that the work we have done is not complete. There are a few technical issues that will have to be addressed:

1. **State explosion:** The state explosion problem is even more acute than with model checking because we have to perform symbolic computations while M and $T(\psi)$ are both in memory. This implies that at present the circuits that we can apply this method to are smaller than those that we can model check. Therefore we currently cannot provide a solution for large models. However we believe that over time optimizations in this area will be introduced as

was done for model checking. We are investigating the possibility of running this method separately on small properties and then combining the results. Another solution to the state explosion is to compute the criteria "on-the-fly" together with the computation of $ReachSIM$ and to discover violations before $ReachSIM$ is fully computed.
A third solution is to use the algorithm in [5] as a preliminary step, and try to expand it to fully support our methodology. The definition of Unimplemented State is closely related to the evidences in [5]. On the other hand, our Unimplemented Transition criterion provides path evidences, while path coverage is not addressed by the methodology of [5]. Furthermore, our method can indicate that the specification and the implementation totally agree. This may serve as an indication that the verification process can be stopped.

2. **Irrelevant information:** Similar to the area of traditional simulation coverage, measurement of quality produces a lot of information which is often irrelevant. A major problem is that specifications tend to be incomplete by nature and therefore we do not necessarily want to achieve a bisimulation relation between the specification and implementation. Therefore, it will eventually be necessary to devise techniques to filter the results such that only the interesting evidences are reported.
We are also investigating whether the reduced tableau described in Section 6 is optimal in the sense that it does not contain any redundancies.

3. **Expressivity:** Our specification language is currently restricted to ACTL safety formulas. It is straight forward to extend our method to full ACTL. This will require, however, to add fairness constraints to the tableau structure and to use the *fair simulation preorder* [3]. Unfortunately, there is no efficient algorithm to implement fair simulation [4]. Thus, it is currently impractical to use full ACTL. There is a need to find logics that are both reasonable in terms of their expressivity and practical in terms of tableau construction and comparison criteria.

Acknowledgment

We thank Ilan Beer for suggesting to look into the problem of coverage in model checking. The first author thanks Galileo Technology for the opportunity to work on the subject.

References

1. I. Beer, S. Ben-David, C. Eisner, and A. Landver. Rulebase - an industry oriented formal verification tool. In *33th Design Automation Conference*, 1996. DAC. 281
2. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT press, 1999. To appear. 280
3. O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. on Programming Languages and Systems*, 16(3):843–871, 1994. 281, 282, 283, 284, 292, 293, 295

4. T. A. Henzinger, O. Kupferman, and S. K. Rajamani. Fair simulation. In *Proc. of the 7th Conference on Concurrency Theory (CONCUR'97)*, volume 1243 of *LNCS*, Warsaw, July 1997. 295
5. Hoskote, Kam, Ho, and Zhao. Coverage estimation for symbolic model checking. In *proceedings of the 36rd Design Automation Conference (DAC'99)*. IEEE Computer Society Press, June 1999. 282, 295
6. Z. Manna and A. Pnueli. *Temporal verifications of Reactive Systems - Safety.* Springer-Verlag, 1995. 281, 293
7. K. L. McMillan. *The SMV System DRAFT.* Carnegie Mellon University, Pittsburgh, PA, 1992. 282, 290
8. K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Press, Norwell, MA, 1993. 280
9. R. Milner. An algebraic definition of simulation between programs. In *In proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pages 481–489, September 1971. 281, 283
10. T. Filkorn. A method for symbolic verification of synchronous circuits. In D. Borrione and R. Waxman, editors, *Proceedings of The Tenth International Symposium on Computer Hardware Description Languages and their Applications*, IFIP WG 10.2, pages 249–259, Marseille, April 1991. North-Holland. 283
11. Elaine J. Weyuker and Bingchiang Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 2(17), July 1991. 282

A Proof of Lemma 1

Lemma 1 *ReachSIM is a simulation preorder from M to M'.*

Proof. Clearly, for initial states, $(s_0, s'_0) \in SIM$ if and only if $(s_0, s'_0) \in ReachSIM$. Thus, for every initial state of M there is a $ReachSIM$-related initial state of M'. Let $(s, s') \in ReachSIM$. First we note that since $ReachSIM \subseteq SIM$, $(s, s') \in SIM$ and therefore $L(s) = L'(s')$.

Now let $(s, s_1) \in R$. Then there is s'_1 such that $(s', s'_1) \in R'$ and $(s_1, s'_1) \in SIM$. Since $(s, s') \in ReachSIM$, there are corresponding paths π and π' leading to s and s'. These paths can be extended to corresponding paths leading to s_1 and s'_1. Thus, $(s_1, s'_1) \in ReachSIM$. □

B Proof of Theorem 2

Theorem 2 *Let M be an implementation model and ψ be an ACTL safety formula such that $M \models \psi$. Let $T(\psi)$ be a tableau for ψ that satisfy the tableau properties. If the comparison criteria 1-3 hold then $T(\psi) \leq M$.*

Proof. Since $M \models \psi$, $M \leq T(\psi)$. Thus, there is a simulation preorder $SIM \subseteq S_i \times S_t$. Let $ReachSIM$ be the reachable simulation preorder for SIM. Then $ReachSIM^{-1} \subseteq S_t \times S_i$ is defined by $(s_t, s_i) \in ReachSIM^{-1}$ if and only if $(s_i, s_t) \in ReachSIM$. We show that $ReachSIM^{-1}$ is a simulation preorder from $T(\psi)$ to M.

Let s_{0t} be an initial state of $T(\psi)$. Since $UnImplementedStartState$ is empty, there must be an initial state s_{0i} of M such that $(s_{0i}, s_{0t}) \in ReachSIM$. Thus, $(s_{0t}, s_{0i}) \in ReachSIM^{-1}$.
Now let $(s_t, s_i) \in ReachSIM^{-1}$. Since $(s_i, s_t) \in ReachSIM$, $L_t(s_t) = L_i(s_i)$.
Let $(s_t, s'_t) \in R_t$. Since $UnimplementedState$ is empty, there must be a state $s'_i \in S_i$ such that $(s'_i, s'_t) \in ReachSIM$. Since $UnImplementedTransition$ is empty we get $(s_i, s'_i) \in R_i$. Thus, s'_i is a successor of s_i and $(s'_t, s'_i) \in ReachSIM^{-1}$.
We conclude that $ReachSIM^{-1}$ is a simulation preorder and therefore $T(\psi) \leq M$. □

Note that, since $ReachSIM$ and $ReachSIM^{-1}$ are both simulation preorders, $ReachSIM$ is actually a bisimulation relation.

C Derivation of the *Compose* Formulas

Following is the algebraic derivation that enables the use of the *compose* and *compose_odd* operations described in Section 5.

- Derivation of SIM_j:
  $SIM_{j+1}(\boldsymbol{v_i}, \boldsymbol{v_t}) :=$
  $\forall \boldsymbol{v'_i}[R_i(\boldsymbol{v_i}, \boldsymbol{v'_i}) \rightarrow \exists \boldsymbol{v'_t}\,[\,R_t(\boldsymbol{v_t}, \boldsymbol{v'_t}) \wedge SIM_j(\boldsymbol{v'_i}, \boldsymbol{v'_t})]] \wedge SIM_j(\boldsymbol{v_i}, \boldsymbol{v_t}) =$
  $\forall \boldsymbol{v'_i}[\neg R_i(\boldsymbol{v_i}, \boldsymbol{v'_i}) \vee \exists \boldsymbol{v'_t}\,[\,R_t(\boldsymbol{v_t}, \boldsymbol{v'_t}) \wedge SIM_j(\boldsymbol{v'_i}, \boldsymbol{v'_t})]] \wedge SIM_j(\boldsymbol{v_i}, \boldsymbol{v_t}) =$
  $\neg\exists \boldsymbol{v'_i}[R_i(\boldsymbol{v_i}, \boldsymbol{v'_i}) \wedge \neg\exists \boldsymbol{v'_t}\,[\,R_t(\boldsymbol{v_t}, \boldsymbol{v'_t}) \wedge SIM_j(\boldsymbol{v'_i}, \boldsymbol{v'_t})]] \wedge SIM_j(\boldsymbol{v_i}, \boldsymbol{v_t}) =$
  $\neg compose\_odd(R_i(\boldsymbol{v_i}, \boldsymbol{v'_i}), \neg compose\_odd(R_t(\boldsymbol{v_t}, \boldsymbol{v'_t}), SIM_j(\boldsymbol{v'_i}, \boldsymbol{v'_t}))) \wedge$
  $SIM_j(\boldsymbol{v_i}, \boldsymbol{v_t})$

- Derivation of $ReachSIM_j$:
  $f_{j+1}(\boldsymbol{v_t}, \boldsymbol{v'_i}) :=$
  $\exists \boldsymbol{v_i}(ReachSIM_j(\boldsymbol{v_i}, \boldsymbol{v_t}) \wedge R_i(\boldsymbol{v_i}, \boldsymbol{v'_i}))$
  $= compose(ReachSIM_j(\boldsymbol{v_i}, \boldsymbol{v_t}), R_i(\boldsymbol{v_i}, \boldsymbol{v'_i}))$ $g_{j+1}(\boldsymbol{v'_i}, \boldsymbol{v'_t}) :=$
  $\exists \boldsymbol{v_t}(f_{j+1}(\boldsymbol{v_t}, \boldsymbol{v'_i}) \wedge R_t(\boldsymbol{v_t}, \boldsymbol{v'_t})) = compose(f_{j+1}(\boldsymbol{v_t}, \boldsymbol{v'_i}), R_t(\boldsymbol{v_t}, \boldsymbol{v'_t}))$
  $g_{j+1}(\boldsymbol{v_i}, \boldsymbol{v_t}) := g_{j+1}(\boldsymbol{v'_i}, \boldsymbol{v'_t})$
  $ReachSIM_{j+1}(\boldsymbol{v_i}, \boldsymbol{v_t}) := (g_{j+1}(\boldsymbol{v_i}, \boldsymbol{v_t}) \wedge SIM(\boldsymbol{v_i}, \boldsymbol{v_t})) \vee ReachSIM_j(\boldsymbol{v_i}, \boldsymbol{v_t})$

- Derivation of $ManyToOne$:
  $ManyToOne(\boldsymbol{v_t}) :=$
  $\exists \boldsymbol{v_1}, \boldsymbol{v_2}(ReachSIM(\boldsymbol{v_1}, \boldsymbol{v_t}) \wedge ReachSIM(\boldsymbol{v_2}, \boldsymbol{v_t}) \wedge (\boldsymbol{v_1} \neq \boldsymbol{v_2})) =$
  $\exists \boldsymbol{v_1}(ReachSIM(\boldsymbol{v_1}, \boldsymbol{v_t}) \wedge \exists \boldsymbol{v_2}((\boldsymbol{v_2} \neq \boldsymbol{v_1}) \wedge ReachSIM(\boldsymbol{v_2}, \boldsymbol{v_t}))) =$
  $\exists \boldsymbol{v_1}(ReachSIM(\boldsymbol{v_1}, \boldsymbol{v_t}) \wedge compose((\boldsymbol{v_1} \neq \boldsymbol{v_2}), ReachSIM(\boldsymbol{v_2}, \boldsymbol{v_t})))$

- Derivation of $UnimplementedTransition$:
  $f(\boldsymbol{v'_i}, \boldsymbol{v_t}) :=$
  $\exists \boldsymbol{v_i}(\neg R_i(\boldsymbol{v_i}, \boldsymbol{v'_i}) \wedge ReachSIM(\boldsymbol{v_i}, \boldsymbol{v_t})) =$
  $compose(\neg R_i(\boldsymbol{v_i}, \boldsymbol{v'_i}), ReachSIM(\boldsymbol{v_i}, \boldsymbol{v_t}))$ $g(\boldsymbol{v_t}, \boldsymbol{v'_t}) :=$
  $\exists \boldsymbol{v'_i}(f(\boldsymbol{v'_i}, \boldsymbol{v_t}) \wedge ReachSIM(\boldsymbol{v'_i}, \boldsymbol{v'_t})) = compose(f(\boldsymbol{v'_i}, \boldsymbol{v_t}), ReachSIM(\boldsymbol{v'_i}, \boldsymbol{v'_t}))$
  $UnimplementedTransition(\boldsymbol{v_t}, \boldsymbol{v'_t}) := g(\boldsymbol{v_t}, \boldsymbol{v'_t}) \wedge R_t(\boldsymbol{v_t}, \boldsymbol{v'_t})$