Techniques for increasing Coverage of Formal Verification

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science
in Computer Science

Sagi Katz

Submitted to the Senate of The Technion - Israel Institute of Technology

TEVET 5762 Haifa December 2001

The research thesis was done under the supervision of Prof. Orna Grumberg, in the Faculty of Computer Science, the Technion - Israel Institute of Technology.

The generous financial help of the Technion is gratefully acknowledged.

I would like to thank Orna Grumberg for her endless patience and willingness. From Orna I learned about scientific disciplines, and about the persistent need for accuracy. In particular I would like to thank Orna for her graceful push and support, which made it possible for this work to complete.

I thank Danny Geist, who assisted me in my research. Danny acted as a second supervisor, and he contributed to this work throughout. In particular, the implementation aspects of this work were inspired by Danny's ideas.

I thank Kirill Grouchnikov. Kirill implemented the ideas in this work. Kirill's questions helped in the definition process, and the excellent outcome of his coding greatly assisted in the research process.

I thank Ilan Beer for suggesting that I look into the problem of coverage in model checking.

I thank Galileo Technology for supporting my work on the subject.

Finally, I thank my parents for all they have gave me.

Contents

List of Figures

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, and no additional specification formulas need be checked. Thus, knowledge of whether the specification is complete may help us avoid missed implementation errors and save precious verification time.

The completeness of a specification for 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. The comparison can be made on different abstraction levels, chosen by the user. When the most strict comparison is applied, empty criteria imply that the tableau is bisimilar to the implementation model and that the specification fully describes the implementation. We then also conclude that there are no redundant states in the implementation.

In order to allow comparison using different abstraction levels we developed a *reduced tableau* for safety ACTL formulas. The states of the reduced tableau consist of the exact requirements imposed by the formula. Everything else is regarded as "don't care."

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 to half of the naive use.

Chapter 1

Introduction

1.1 Why Model Checking is Not Complete

The increasing complexity of hardware components, and of the industrial VLSI logic design in particular, impose new challenges in design verification and compliance. The traditional verification methods, which are mainly *dynamic-simulation* based, are no longer adequate for covering the entire range of functionality to be checked. The effort spent on design verification is huge. This effort tends to increase with the increase in design complexity and in quality requirements. These trends have led to the rapid development and flourishing of the *model checking* field.

Given a model of the design (or implementation) and a temporal logic formula that describes a specification, model checking [8] determines whether the model satisfies the specification.

Model-checking algorithms were first developed about two decades ago for various *temporal languages* [31, 7]. The methods began to appeal to the industry when they became *symbolic* and their memory requirements were reduced [28]. In symbolic model checking, the algorithm deals with sets of states rather than dealing explicitly with each state. This is accomplished with an efficient data structure representation, known as an OBDD [3].

The strength of model checking is that the technique can exhaustively cover the whole state space of a relatively small model. Exhaustive coverage of control logic is a task that is far beyond the ability of traditional verification techniques. However, the fact that model checking can only deal with relatively small models is the major limitation of this method.

Small and critical portions of industrial designs are verified daily using model checking. It is indeed true that a model checking technique, applied on an appropriate model, reaches **100%** of its state space. This fact is the basis of a common but false conception: if model checking is exhaustive, who could ask for more? However, reaching all states is not sufficient: one also has to verify all the relevant requirements. If a requirement is not checked in the model checking verification process, one would like an indication of this.

1.2 Evaluating Completeness

This work presents a novel approach for evaluating the quality of the verification when using model checking.

Assume that all specification formulas were successfully checked for a given 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, knowledge of whether the specification is complete may help us avoid missed implementation errors and save precious verification time. Completeness of a set of specification formulas is achieved when any addition of a valid formula to the set will not change the system behavior as described by the specification.

Below we describe our method for determining whether a specification is complete with respect to a given implementation. We focus only on safety properties written in the universal branching-time logic ACTL [14]. ACTL is the temporal language obtained from the branching temporal language CTL when removing the existential operators. We further remove the liveness operators syntactically. This logic is relatively restricted, but can still express most of the specifications used in practice. Moreover, it can fully characterize every implementation model. The restriction to the safety subset of ACTL is done only from complexity considerations, and does not influence the ability to reach a complete set of specification formulas. 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* [14]. One property of the tableau is that it is greater by the *simulation preorder* [29] than every model that satisfies the formula. A simulation preorder between two models means that all valid behavior of one model can be mapped to the behavior of the other model. Thus, since the formula is true with respect to the model, the tableau is greater than the model by the simulation preorder.

Given a tableau for the specification formula, we use the simulation preorder to find differences between the implementation and its specification. We restrict our attention to the reachable portion of the simulation relation, starting from initial states of the implementation and the tableau. The simulation relation imposes that every implementation state has a corresponding tableau state, but not every tableau state has a corresponding implementation state. For example, if we find a reachable tableau state with no corresponding implementation state, then we argue that one of the following holds: either the specification is not restrictive enough or the implementation fails to implement a meaningful state. In the first case the specification formula should be corrected while in the second case the implementation is missing some behavior. 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 in making 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 [30] to the implementation model and that the specification fully describes the implementation. We also conclude that there are no redundant states in the implementation.

1.3 Using Reduced Tableau for Completeness Evaluation

The method described so far is valid but has drawbacks. If any tableau is used (e.g. [14], false indication of coverage problems may be received. We remove such false indications by defining a new tableau structure. We define a *reduced tableau* for ACTL safety formulas. Our tableau is based on the *particle tableau* for LTL, presented in [26]. We further reduce their construction by using only *elementary formulas* and removing redundant tableau states. Our construction maintains a three-value labeling method.

Although we can use any ACTL tableau structure for determining the completeness of a specification, we define the reduced tableau for four reasons. First, we would like to eliminate false indications of incompleteness. Second, our reduced construction builds states composed of exactly the requirement imposed by the formula. Whatever is not required by the given formula is left as a "don't care." The "don't care" property of the reduced tableau enables us to deduce incompleteness only on conditions stated explicitly in the specification formula. Third, the reduced tableau has a smaller size. Finally, as will be exemplified, the reduced tableau allows us to detect redundancies in the specification formula.

We show an example of a reduced tableau with 20 states (see figure 13.2). The tableau presented in [14] would have a state space of 2^{15} states for the same formula.

1.4 Applicability of the Comparison Method

Our comparison method can deal with various levels of system description. One of the major criticisms of our proposed method is that it is necessary to know all the internal details of the design in order to achieve completeness. In practice, however, all one needs to know in order to describe the model with respect to the given observable variables is their correct behavior. Variables that are not observable are hidden and do not participate in the specification formula. If the observable variables are partial and describe only an abstract system, our method can detail the completeness of the abstracted formula with respect to the system.

Another advantage of our method is its ability to detect that a model for the environment of the implementation does not generate all possible input sequences. If the environment does not generate some input sequence, it may hide an implementation bug, even when the specification for the model under test is complete.

Completeness is determined by reaching empty criteria (which imposes bisimulation). We expand our methodology and suggest a notion of *practical completeness*, where bisimulation need not be preserved. Practical completeness is achieved when the comparison criteria are empty with respect to a three-value reduced tableau. The reduced tableau does not construct states unless they carry meaningful formula elements. When some condition is not explicitly mentioned in the specification formula, it is considered a "don't care" and is not included in the tableau. When the value of a proposition is not specified, we may consider it as a shorthand and check that every possible value of that proposition appears in the implementation. On the other hand, we may consider it as a real "don't care" and allow the implementation to choose only some possible value. The preferred interpretation may be chosen by the user. In order to provide the user with the choice between the two possibilities, we define, in addition to the comparison criteria defined above also a three-value comparison criteria.

The regular (two-value) comparison criteria used with the reduced tableau compares the implementation and specification under the assumption that unspecified propositions are interpreted as “don't cares.” It does require, however, that every condition that explicitly appears in the specification must also be satisfied by the implementation. If bisimulation between the implementation and the reduced tableau is desired, we use the three-value labeling comparison criteria, allowing the interpretation of “don't care” to be “all possible states.”

1.5 Efficient Implementation

We suggest how our method can be implemented symbolically as an extension to a symbolic model checker such as SMV [27]. Given a model with n state variables, a straightforward implementation of this method can create intermediate results that consist of $4n$ OBDD variables. However, our implementation proposal reduces the required number of OBDD variables from $4n$ to $2n$.

1.6 Summary of Contributions

The main contributions of our work can be summarized as follows:

- We suggest a theoretical framework for quality evaluation of model checking. Our comparison criteria can assist the designer in finding errors by showing where the design and the specification disagree, and suggesting when the verification effort should be terminated.

- We show how to implement our method symbolically and in an efficient manner. Of special interest is the symbolic computation of the simulation relation for which no good symbolic algorithm is known.

- We define a new reduced tableau for ACTL that is often significantly smaller in the number of states and transitions than other known tableaux.

1.7 Thesis Organization

The rest of this work is organized as follows. Chapter 2 describes related work. Chapter 3 gives the necessary background. Chapter 4 describes the comparison criteria and the methodology for their use. Chapter 5 exemplifies the different criteria by applying the method to a small hardware circuit. Chapter 6 shows how the comparison method relates to various abstraction levels of the system. Chapter 7 presents symbolic algorithms that implement our method. Chapter 8 explains an innovative implementation of our algorithms. Chapters 9 and 10 define the reduced tableau for ACTL safety formulas. Chapter 11 presents proof of the correctness of our construction. Chapter 12 adapts the comparison criteria to the reduced tableau. Chapter 13 shows applications to the reduced tableau. Chapter 14.1 describes extensions to this work. Finally, the last chapter presents conclusions and describes future work.

Chapter 2

Related Work

2.1 Model-Checking Coverage

Until recently, the issue of coverage in model checking has not been addressed. In addition to identifying the quality problem of model checking, it was not clear how the coverage problem should be addressed. No known work had been published to suggest an approach for checking coverage, nor had an algorithm been suggested.

In parallel to our work [20], another work on coverage of model checking has been independently developed [18]. That work computes the percentage of states in which a change in an observable proposition will not affect the correctness of the specification. This percentage is reported as a *coverage metric*. The authors' *evidence* is closely related to our *unimplemented state* criterion that will later be presented. Hoskote et al. assume a flavor of ACTL, where the only disjunction allowed is of the form $p \rightarrow \varphi$, such that p is propositional. The claim is that this subset of ACTL is acceptable. They use this subset to reduce the complexity of computing the coverage metric to linear time, with performance only a little worse than the model checking task. The appealing complexity is achieved by the language they choose, and by the fact that they can measure the coverage of each formula separately, and combine at the end of the process. The authors also list a number of limitations of their work. They are unable to give path evidence, cannot point out missing functionality aspects in the model, and have no indication that the specification is complete. In the next chapters we explain how our work solves these problems.

A more recent work [5] examines the area of model checking coverage from a wide perspective. Chockler et al. suggest two additional coverage metrics. The first metric is state-based, and is a generalization of the metric suggested by [18]. Again, it checks whether a mutation of the state space (i.e., a value change of an observable proposition in a specific state) satisfies the given specifications. The second metric is circuit-based and assumes mutations to the circuit, rather than the state space. An example of a circuit mutation is to force a value of zero in a specific node, and in any expression that uses this node. Chockler et al. also suggest a principle: to distinguish between the design and the environment, and check only the design. This work examines the complexity of such coverage metrics. Chockler et al. choose not to deal with the transitions of the model (except transitions influenced by circuit mutation), thus claiming their coverage to be effective rather than complete.

In contrast to [18] and to [5], our work is capable of detecting missing states and transitions in the environment. If you divide the observable signals to inputs and outputs, the environment is the part of the model that defines the inputs, as a function of the outputs. Model checking reaches all states of a circuit only if all the combinations of the inputs are generated. An input state or input sequence that cannot be generated by the environment may hide a bug in the circuit. In fact, detecting problems in a model checking environment is a problem that has not been addressed so far. In section 5.5 we exemplify how environment is checked using our method.

2.2 Equivalence Checking and Simulation

The analysis we perform compares the implementation model and the tableau model, and tries to identify dissimilarities. It is therefore related to tautology checking of finite state machines, as is done in [34]. However the method in [34] is suggested as an alternative, not as a complement, to model checking.

Another remotely related area is *equivalence checking*. Boolean and other versions of equivalence checking are very common in the VLSI industry. In equivalence checking, two representations of the same function are compared, and differences, if they exist, are reported. Usually, equivalence checking deals with the equivalence of combinatorial circuits. Some attempts were also made to compare sequential representations of circuits. Since designs are deterministic, trace equivalence is straightforward. When comparing nondeterministic models (which may come from abstracting portions of the design), one will be facing a problem similar to ours: how to compare a nondeterministic model of the design (and its environment) with a tableau, which is a nondeterministic structure.

Simulation and bisimulation relations has been applied in the context of model checking, equivalence checking, refinement checking and minimization [25, 10]. It has never been used for checking coverage. Beyond using simulation, our work describes an efficient implementation of a symbolic algorithm for computing the simulation relation. Previous symbolic algorithm for simulation may be found in [15]. Our contribution is a memory efficient implementation of the same principals.

2.3 Tableau Construction

We build a reduced tableau, to be compared with the given implementation model. We want our tableau, the construction of which will be described later, to be as small as possible.

In the context of model checking, tableaux have been used for the logic of LTL [24], ACTL [14] and μ-calculus [9]. Most initial constructions defined the set of tableau states as the powerset of the subformulas of the given formula, yielding an exponential number of states.

Recently, several works that aim to reduce the tableau size have been published. In [11], Daniele et al. combine a few tableau construction algorithms into a single framework. In an approach similar to ours, Daniele et al. construct the tableau using only elementary formulas. They show significant reduction in the tableau size relative to previous works. Since the main motivation for this tableau is for use *on the fly* in the process of LTL model checking, not all the possible simplifications are utilized. We applied the algorithm reported in [11] to our example. The resulting tableau contained 215 states instead of the 20 states we received using our reduced tableau construction. Using the tableau of [11] would have introduced false evidence to our coverage procedure.

An additional work [33] has recently been published on small tableau construction. Somenzi et al. presented the idea, previously published in [20], that successors can be eliminated when one successor simulates the allowed behavior of the other. Somenzi et al. implement various Boolean reductions, which are aimed at combining semantically identical states. They use a method similar to the one developed in this work, by which a Boolean function is represented by a DNF (Disjunctive Normal Form). The advantage of a normal form is that it is unique for a certain family of equivalent formulas. The tableau presented by Somenzi et al. is not a three-value labeling tableau, but it is certainly appealing from the coverage point of view.

2.4 Vacuity and Sanity Checks

One of the first suggested methods to identify problems in a model checking system is to detect *vacuous satisfaction* of the specification property [1, 22]. A property is vacuously satisfied if part of the specification property does not influence the satisfaction of the property. For example the property $AG(\neg p \rightarrow AX\, p)$ may be trivially satisfied in a system where p is always true. Detecting a vacuous satisfaction may indicate either a badly written specification property or a problem in the design or the environment.

Vacuity is orthogonal to Model-Checking coverage. A specification formula may be complete, but still may contain a vacuous sub-formula. In addition, a non-complete specification formula does not imply weather it is vacuous or not. Vacuity checking is most useful for sanity checking a single formula, while coverage checking deals with the complete set of specification formulas.

Checking the validity of the specification formula is also a possible sanity check. In addition a check of weather the specification formula is satisfied regardless of the implementation model, may be done.

Another form of sanity checks may be found in the model checking tool FormalCheck [23], where properties always have an enabling condition. The tool includes a validation phase that verifies that each enabling condition is satisfied. Again, detecting an enabling condition that is not possible indicates of some problem. As will be seen in section 13.3, we suggest a criterion that enables us to detect redundant portions of the specification formula. This criterion allows the detection of problems not addressed by the initial works in this area.

2.5 Coverage of Dynamic Simulation

Checking how well a verification system covers the interesting cases is not new. It is common in simulation-based hardware verification methods. For dynamic simulation it is clear that exhaustive coverage of all possible cases to be checked is far from being feasible. Many *coverage metrics* are used to monitor and generate test vectors in the dynamic simulation process, and increase the cases it covers. One of the early measures was *toggle coverage*, a metric measuring the percentage of nodes in the design that were toggled while simulating the test vectors. The nodes of a design are typically elements that can be assign logical values, a node was considered toggled if it changed value anywhere in the execution of the test vectors.

An additional common metric is *code coverage* [4], which defines the percentage of HDL statements executed during simulation. While code coverage, toggle coverage and similar methods are useful in detecting large portions of the design not being exercised, these methods are not useful in identifying that all interesting events are present in the test case. A more detailed measure is *Transition coverage* [32, 19] that defines the percentage of code transitions that have been executed. An observability-based approach is suggested in [12]. There is a difference between a test case being executed and an erroneous behavior being observed. The suggested approach is to assign forbidden values to variable, and monitor erroneous behavior of the system.

It was identified that most design bugs are in the control portion. The work in [17] defines and measures *control events*. Defining meaningful control events is not trivial, and in addition may lead to a state explosion problem.

The state explosion problem led to a development of *projection-directed state exploration* techniques [2]. It turns out that exploring just the projected control logic of a design may be a non feasible task.

As may be evident from the area of dynamic simulation, checking the coverage of a verification system is essential. It may also show that reaching a goal of 100% for a certain coverage metric may be far from complete coverage.

Chapter 3

Preliminaries

Our specification language is the universal branching-time temporal logic ACTL [14], 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 $\mathbf{X}$ ("next-state") and $\mathbf{W}$ ("weak until") and the path quantifier $\mathbf{A}$ ("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.

Definition 3.0.1 ($\models$) *[Satisfaction of ACTL safety formulas]*
Given a Kripke structure M and a formula φ, we define $s \models \varphi$ recursively:

1. *If $\varphi = p$ and $p \in AP$, then $s \models p$ iff $p \in L(s)$.*

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

2. *If $\varphi = \neg p$ and $p \in AP$, then $s \models \neg p$ iff $p \notin L(s)$.*

3. *$s \models \varphi_1 \vee \varphi_2$ iff $s \models \varphi_1$ or $s \models \varphi_2$.*

4. *$s \models \varphi_1 \wedge \varphi_2$ iff $s \models \varphi_1$ and $s \models \varphi_2$.*

5. *$s \models AX\varphi$ iff for every successor s_1 of s $s_1 \models \varphi$.*

6. *$s \models A[\varphi_1 W \varphi_2]$ iff for every path $\pi = s_1 s_2 ...$ from s, one of the following holds: either for all $i > 0$, $s_i \models \varphi_1$, or there exists n such that $s_n \models \varphi_2$ and for all $i < n$, $s_i \models \varphi_1$.*

A structure M satisfies a formula φ, denoted $M \models \varphi$, if every initial state of M satisfies φ.

We now define the maximal simulation preorder. A simulation preorder between two structures suggests that all behavior possible in the "smaller" model is also possible in the "large" one.

Definition 3.0.2 (SIM) *[Maximal Simulation Preorder]*
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 [29] 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*
- *$\forall s'_1[(s', s'_1) \in R' \Longrightarrow \exists s_1[(s, s_1) \in R \wedge (s'_1, s_1) \in SIM']]$.*

We define SIM as the maximal relation of simulation relations SIM'.

If there is a simulation preorder from M' to M, we write $M' \leq M$. In this work we refer to this as *M' simulates M*.

As shown in [14], if M' simulates M then every ACTL formula satisfied in M is also true in M'. This is stated in the following Lemma.

Lemma 3.0.1 *If $M' \leq M$ then $\forall \varphi \in ACTL\, [M \models \varphi \Rightarrow M' \models \varphi]$.*

Lemma 3.0.2 *For every M and M', there is a* greatest *simulation preorder from M' to M.*

Proof:
The empty relation is a simulation preorder. Furthermore, if SIM_1 and SIM_2 are simulation preorders then so is $SIM_1 \cup SIM_2$. Since $S' \times S$ is finite there is a *greatest* simulation preorder. $\Box$

It has been shown in [14] that for every ACTL safety formula ψ it is possible to construct a Kripke structure $\tau(\psi)$, called a *tableau*,[2] for ψ.

Definition 3.0.3 ($\tau(\psi)$) *[Tableau for safety ACTL]*
Given an ACTL safety formula ψ, a tableau *for ψ is a Kripke structure $\tau(\psi)$ with the following properties:*

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

We refer to these properties as the *tableau properties*. In this work we construct a specific reduced tableau that better suits our purpose.

We want to compare a given model M and a formula ψ. We rely on the tableau properties to compare M and $\tau(\psi)$ by analyzing the simulation preorder. Intuitively, the simulation preorder relates two states if the computation tree that starts from the state of the smaller model can be embedded in the computation tree that starts from the state of the greater one. This, however, is not sufficient to determine how similar the two structures are. Instead, we use the *reachable simulation preorder*, which relates two states if they are in the simulation preorder and are also reachable from initial states along corresponding paths.

Definition 3.0.4 (ReachSIM)
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 as follows: $(s', s) \in ReachSIM$ 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.*

[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.

Lemma 3.0.3 *$ReachSIM$ is a simulation preorder from M' to M.*

Proof:
Clearly, for the 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. Assume that $(s', s) \in ReachSIM$. First we note that since $ReachSIM \subseteq SIM$, $(s', s) \in SIM$ and therefore $L'(s') = L(s)$.

Now assume that $(s', s'_1) \in R'$. 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, $(s'_1, s_1) \in ReachSIM$. $\square$

Chapter 4

Comparison Criteria

Let $M = (S_i, S_{0i}, R_i, L_i)$ be an implementation structure and $\tau(\psi) = (S_t, S_{0t}, R_t, L_t)$ 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 $M \leq \tau(\psi)$. We define four criteria, each associated with a set. A criterion is said to hold if the appropriate set is empty. For convenience, we give each criterion the same name as the appropriate set. The following sets define the criteria :

1. $\mathit{UnImplementedStartState} = \{s_t \in S_{0t} \mid \forall s_i \in S_{0i}\, [\, (s_i, s_t) \notin \mathit{ReachSIM}\,]\,\}$
   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.
   In Figure 4.1, state t_2 is an initial state in the tableau model, but it does not have a corresponding initial state in the implementation model.

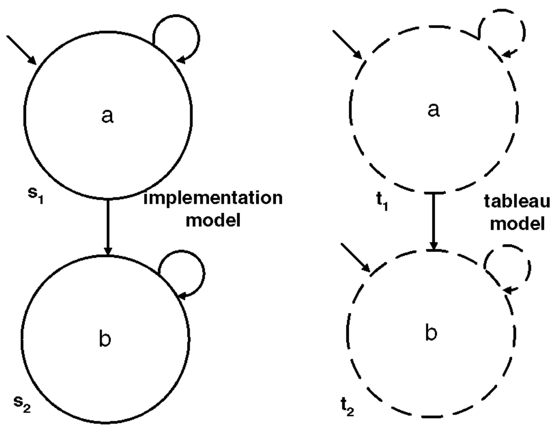

Figure 4.1: Unimplemented Start State Example

2. $\mathit{UnImplementedState} = \{s_t \in S_t \mid \forall s_i \in S_i\,[\,(s_i, s_t) \notin \mathit{ReachSIM}\,]\,\}$
   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.

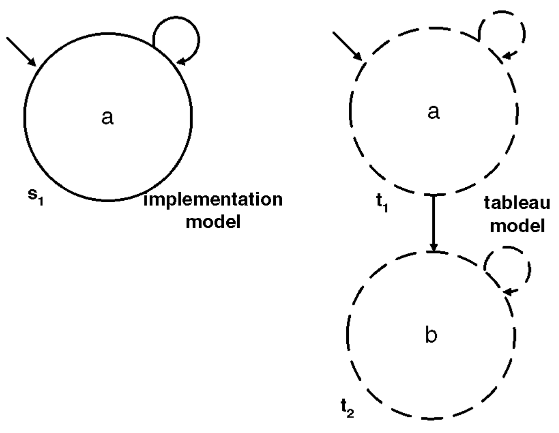

Figure 4.2: Unimplemented State Example

In Figure 4.2, state t_2 is a tableau state that does not have a corresponding state in the implementation model.

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.

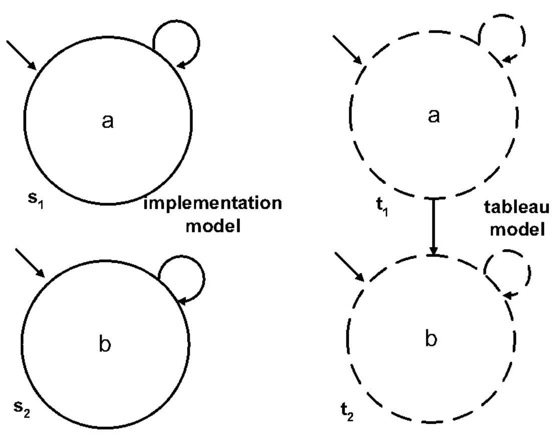

Figure 4.3: Unimplemented Transition Example

In Figure 4.3, the transition (t_1, t_2) is a tableau transition that does not have a corresponding transition in the implementation model.

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 parts of the implementation are redundant.

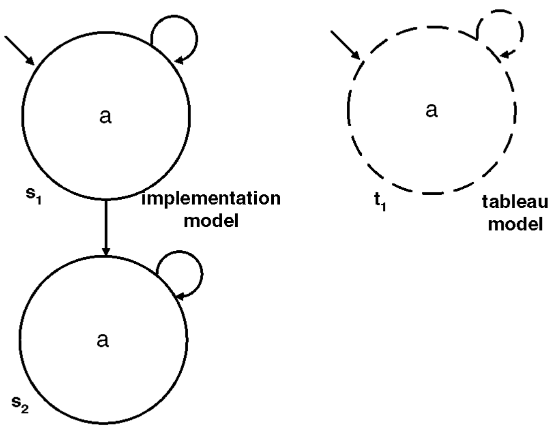

Figure 4.4: Many-to-One Example

In Figure 4.4, both of the implementation states s_1 and s_2 are mapped to the same tableau state t_1.

Our criteria are defined for any tableau that has the tableau properties as defined in Chapter 3. Any dissimilarity between the implementation and the specification will result in a nonempty 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 Chapter 9, redundancies are removed and therefore empty criteria can be obtained.

The algorithm for checking the coverage of a model and its specification is presented below.

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 $\tau(\psi)$ for ψ.
3. Compute SIM of $(M, \tau(\psi))$.
4. Compute *ReachSIM* of $(M, \tau(\psi))$ from SIM of $(M, \tau(\psi))$.
5. For each of the comparison criteria, evaluate if its corresponding set is empty. If not, present evidence for its failure.

Figure 4.5: Coverage Algorithm

Theorem 4.0.4 *Let M be an implementation model and ψ be an ACTL safety formula such that $M \models \psi$. Let $\tau(\psi)$ be a tableau for ψ that has the tableau properties. If the comparison criteria 1-3 hold, then $\tau(\psi) \leq M$.*

Proof:
Since $M \models \psi$, $M \leq \tau(\psi)$. 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 $\tau(\psi)$ to M.

Let s_{0t} be an initial state of $\tau(\psi)$. 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 assume $(s_t, s_i) \in ReachSIM^{-1}$. Since $(s_i, s_t) \in ReachSIM$, $L_t(s_t) = L_i(s_i)$.

Assume $(s_t, s_t') \in R_t$. 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 $\tau(\psi) \leq M$. □

Note that since $\mathit{ReachSIM}$ and $\mathit{ReachSIM}^{-1}$ are both simulation preorders, $\mathit{ReachSIM}$ is actually a bisimulation relation. Thus, if criteria 1-3 hold, then $\tau(\psi)$ and M are in fact *bisimilar*.

The fourth criterion is not necessary for completeness because, whenever there are several non-bisimilar implementation states that are mapped to the same tableau state, there is also an unimplemented state or transition. In the case of two states mapped to the same tableau state and the specification is complete, the two states are bisimilar.

However, this criterion may provide useful information and 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 completeness of the specification. The purpose of the criteria is to assist the designer in the debugging process. Thus, we are looking for meaningful criteria that can distinguish between different types of problems and identify them. In Chapter 9 we define an additional criterion that can reveal redundancy in the specification.

Chapter 5

Example

Consider a synchronous arbiter with two inputs, $req0, req1$, and two acknowledge 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 the case that both request inputs are active. Furthermore, in the case of *simultaneous assertion* (i.e., both requests are asserted and neither was asserted in the previous cycle), request 0 has priority in the first *simultaneous assertion*. In any additional *simultaneous assertion* the priority rotates with respect to the previous one.

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:

```
var
  req0, req1, ack0, ack1, robin : boolean;
assign
  init(ack0) := 0;
  init(ack1) := 0;
  init(robin) := 0;
next(ack0) := case
  !req0              : 0;        – No request results no ack
  !req1              : 1;        – A single request
  !ack0 & !ack1      : !robin;   – Simultaneous requests assertions
  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 chapter, 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 $\mathbf{AG}\varphi \equiv \mathbf{A}[\varphi \mathbf{W} false]$, and $\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_2 \vee \varphi_1$. Using the SMV model checker,we verified that $M \models \psi$. 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 demonstrate the capabilities 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.

The various state spaces of the example and its modification are too detailed to be list here. The reduced tableau state space of the specification formula ψ may be found in figure 13.2.

5.1 Unimplemented Transition Evidence

Consider a modified version of the implementation M, denoted 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, when the comparison method is applied on M_{trans} and ψ_{trans}, an *Unimplemented Transition* is reported. The evidence supplied is 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 the case that variable $robin$ is not asserted.

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

5.2 Unimplemented State Evidence

Consider a modified version of the implementation M, denoted M_{unimp}, obtained by adding the 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, when the comparison method is applied to M_{unimp} and ψ_{unimp}, an *Unimplemented State* is reported. The evidence supplied is 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 see that the evidence violates the mutual exclusion property. Both of the arbiter outputs $ack0$ and $ack1$ are active. This occurs since ψ_{unimp} does not include the mutual exclusion requirement. The original complete specification ψ will detect that M_{unimp} has a bug, since $M_{unimp} \not\models \psi$.

5.3 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 by one of criteria 1-3. The former case, where criteria 1-3 hold but we have *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 fact may give insight into the efficiency of the implementation.

The following implementation, M_{m2o}, uses five implementation variables and two free inputs instead of three variables and two inputs of implementation M. Criteria 1-3 are met for M_{m2o} with respect to ψ.

```
var
  req0, req1, req0q, req1q, ack0q, ack1q, robin : boolean;
assign
  init(req0q) := 0; init(req1q) := 0;
  init(ack0q) := 0; init(ack1q) := 0;
  init(robin) := 1;
define
  ack0 := case
    !req0q              : 0;       – No request results no ack
    !req1q              : 1;       – A single request
    !ack0q & !ack1q     : !robin;  – Simultaneous requests assertions
    1                   : !ack0q;  – Both requesting , toggle ack
  esac;
```

```
$ack1 := case$
  $!req1q$              $: 0;$         – No request results no ack
  $!req0q$              $: 1;$         – A single request
  $!ack0q \,\&\, !ack1q$      $: robin;$     – simultaneous assertion
  $1$                   $: !ack1q;$    – Both requesting , toggle ack
$esac;$
$assign$
$next(robin) := if\ \ req0 \,\&\, req1 \,\&!ack0 \,\&\, !ack1\ \ then\ \ !robin$
  $else\ \ robin\ \ endif;$                    – Two simultaneous request assertions
$next(req0q) := req0;\ next(req1q) := req1;$
$next(ack0q) := ack0;\ next(ack1q) := ack1;$
```

Applying model checking will show that $M_{m2o} \models \psi$.

In the above example we keep information about the current inputs $req0$ and $req1$, as well as their value in the previous cycle (i.e., $req0q$ and $req1q$). Intuitively, we see that each state in M is thus duplicated, so that for each state in M, we have four states in the state space of M_{m2o}.

5.4 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 *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* criterion to hold.

5.5 Comparing the Environments

A given implementation system usually has two components. The first component is the *model under test.* This is the component we would like to verify. The second component is the environment of the model under test. The model under test and its environment communicate via inputs and outputs. The environment is necessary in order to restrict the free inputs of the model under test to include only legal behaviors. The inputs and outputs define a division of the observable variables into those driven by the environment (inputs) and those driven by the model under test (outputs). In our arbiter example, $req0, req1$ are the inputs, and $ack0, ack1$ are the outputs.

Although we are interested in the behavior of the model under test, it is very important to check the completeness of the environment. If some legal input behavior cannot be generated by the environment model, we face the risk of hiding a bug in the model under test. Even if the specification of the model under test is complete (i.e a subset of all specification formulas that fully describe the correct behavior of the model under test), model checking will fail to reveal such a bug.

Our comparison criteria can find problems in the environment as well as in the design. We demonstrate this by an example. This example is somewhat artificial since in our running example $req0$ and $req1$ are free. In more complex cases a degenerated environment is not something easy to detect. Consider a buggy implementation of the arbiter M, denoted M_{robin0}, such that the internal variable $robin$ is stuck at the value of 0. M_{robin0} is obtained by replacing lines 19 and 20 of M with:
$define\ robin := 0;$

We degenerate the free environment by fixing the input $req0$ at the value of 0. We do this by deleting the $req0$ variable from line 2 and adding:
$define\ req0 := 0;$

Now consider the complete specification ψ. Even though the specification is detailed enough, it cannot detect the bug, i.e., $M_{robin0} \models \psi$.

Applying our comparison method will reveal that the comparison criteria are not empty, and some tableau states labeled with $req0$ or $robin$ are unimplemented. The states that could show the bug could not be generated by the environment. Since a restricted environment may hide bugs, this is just as important as finding missing properties.

Chapter 6

Abstraction and Non-observable Implementation Variables

Sometimes we are not interested in the fully detailed specification. We can then restrict our attention to a subset AP' of the atomic propositions AP, called the *observable variables.* This restriction induces an *abstract implementation* which is identical to the original implementation, except that the state labeling is restricted to AP'. Our specification will only describe behaviors over AP'.

If our method is applied to the abstract system with such a specification, and if criteria 1-3 hold, then the abstract system is bisimilar to the specification. Thus, our method is applicable to partially specified systems.

Given a set of observable variables over which the specification can be written, we can distinguish three different relationships between implementations and specifications.

- *Fully observable and fully detailed*, in which all implementation variables are observable.
- *Partially observable and fully detailed*, in which only some of the implementation variables are observable. However, the implementation behavior can be fully described using only the observable variables.
- *Partially observable and partially detailed*, in which only some of the implementation variables are observable and the system can only be partially described with these variables.

To demonstrate the different relationships, consider again our arbiter example. M and ψ exemplify the second case. The implementation variable *robin* is not observable, but the system is fully described using $req0$, $req1$, $ack0$ and $ack1$. This is typical for cases in which the non-observable variables are internal.

We can demonstrate the first case by including *robin* in the set of observable variables. The following is a fully observable specification of M which is also complete.

$$
\begin{array}{llr}
\psi = \neg ack0 \wedge \neg ack1 \wedge \neg robin \quad \wedge & & -\ \varphi_0 \\
\quad \mathbf{AG}(& & \\
\qquad (\neg ack0 \vee \neg ack1) & \wedge & -\ \varphi_1 \\
\qquad (\neg req0 \wedge \neg req1 \rightarrow \mathbf{AX}(\neg ack0 \wedge \neg ack1)) & \wedge & -\ \varphi_2 \\
\qquad (req0 \wedge \neg req1 \rightarrow \mathbf{AX} ack0) & \wedge & -\ \varphi_3 \\
\qquad (\neg req0 \wedge req1 \rightarrow \mathbf{AX} ack1) & \wedge & -\ \varphi_4 \\
\qquad (req1 \wedge ack0 \rightarrow \mathbf{AX} ack1) & \wedge & -\ \varphi_5 \\
\qquad (req0 \wedge ack1 \rightarrow \mathbf{AX} ack0) & \wedge & -\ \varphi_6 \\
\qquad ((\neg req0 \vee \neg req1 \vee ack0 \vee ack1) \wedge robin \rightarrow \mathbf{AX} robin) & \wedge & -\ \varphi_7 \\
\qquad ((\neg req0 \vee \neg req1 \vee ack0 \vee ack1) \wedge \neg robin \rightarrow \mathbf{AX} \neg robin) & \wedge & -\ \varphi_8 \\
\qquad (req0 \wedge req1 \wedge \neg ack0 \wedge \neg ack1 \wedge \neg robin \rightarrow \mathbf{AX}(ack0 \wedge robin)) & \wedge & -\ \varphi_9 \\
\qquad (req0 \wedge req1 \wedge \neg ack0 \wedge \neg ack1 \wedge robin \rightarrow \mathbf{AX}(ack1 \wedge \neg robin)) &) & -\ \varphi_{10}
\end{array}
$$

To demonstrate the third case we consider a restricted description of the system. We choose $AP' = \{req0, ack0\}$. The abstract requirements refer only to the observable variables. Other parts of the arbiter are abstracted away. The verbal description of the abstract system is as follows:

1. Initially $ack0$ is not asserted.
2. If $req0$ is not asserted, then $ack0$ should be de-asserted in the following cycle.
3. Persistent $req0$ should cause $ack0$ in one or two cycles. A request is considered persistent if it remains asserted when not acknowledged.

A formal specification formula would be

$$
\begin{array}{ll}
\psi_{abstract} = \neg ack0 & \wedge \\
\mathbf{AG}((\neg req0 \rightarrow \mathbf{AX} \neg ack0) & \wedge \\
\quad (req0 \rightarrow \mathbf{AX}(ack0 \vee (req0 \rightarrow \mathbf{AX} ack0)))) &
\end{array}
$$

We used the SMV model checker to verify that $M \models \psi_{abstract}$. We then applied our method. We found that comparison criteria 1-3 hold. We therefore concluded that $\psi_{abstract}$ is a complete specification for M, with respect to the given set of observable variables.

An abstract view of a system allows one to check the completeness of one of its critical sections: A full specification for the complete system is not required, nor is it necessary to isolate the critical portion in the implementation. Note that a complete abstract description of a system will always be accompanied by a *many-to-one* indication.

These examples show that we can deal with abstract views of the system; the full implementation details are not always necessary.

Chapter 7

Implementation of the Method

7.1 Symbolic Algorithms

In this section we present the symbolic algorithms that implement various parts of our method. We show how the simulation relation can be computed symbolically. Since we reduce the number of OBDD variables, less memory is required than for the naive implementation. In Chapter 7.2 we show exactly how the number of OBDD variables is reduced. For conciseness, we use $R(s, s')$, $S(s)$, etc., instead of $(s, s') \in R$, $s \in S$.

7.1.1 Computing SIM

The computation of the maximal simulation SIM follows the algorithm presented in [14]. It starts from pairs of states of both models that agree on the labeling values. The algorithm leaves those states that their successors also agree on the labeling values. It is computed by an iterative process that reaches a fixed point.

Let $M = (S_i, S_{0i}, R_i, L_i)$ be the implementation structure and let $\tau(\psi) = (S_t, S_{0t}, R_t, L_t)$ be a tableau structure. The following pseudo-code describes the algorithm for computing SIM: *Init:* $SIM_{next} := \{ (s_i, s_t) \in S_i \times S_t \mid L_i(s_i) = L_t(s_t) \}$;

```
Repeat {
$SIM_{current} := SIM_{next}$;
$SIM_{next} := \{ (s_i, s_t) \mid \forall s_i' [ R_i(s_i, s_i') \rightarrow \exists s_t' [ R_t(s_t, s_t') \wedge SIM_{current}(s_i', s_t')]] \wedge SIM_{current}(s_i, s_t)\}$;
} until $SIM_{current} = SIM_{next}$
$SIM := SIM_{current}$
```

7.1.2 Computing $ReachSIM$:

Computing $ReachSIM$ is done by starting from pairs of initial states in the relation SIM. The algorithm then computes the successors of both states. This mutual reachability process continues until a fixed point.

Given the simulation relation SIM of the pair $(M, \tau(\psi))$, the following pseudo-code describes the algorithm for computing $ReachSIM$:

```
Init: $ReachSIM_{next} := (S_{0i} \times S_{0t}) \cap SIM$;
Repeat {
$ReachSIM_{current} := ReachSIM_{next}$;
$ReachSIM_{next} := ReachSIM_{current} \cup$
$\{ (s_i', s_t') \mid \exists s_i, s_t (ReachSIM_{current}(s_i, s_t) \wedge R_i(s_i, s_i') \wedge R_t(s_t, s_t') \wedge SIM(s_i', s_t')) \}$
} until $ReachSIM_{next} = ReachSIM_{current}$
$ReachSIM := ReachSIM_{current}$
```

7.2 Efficient OBDD Implementation

We now turn our attention to improving the performance of the algorithms described in the previous section. We assume that such an algorithm will be implemented within a symbolic model checker such as SMV [27]. Since formal analysis always suffers from state explosion, it is necessary to find methods to efficiently utilize computer memory. When working with OBDDs, one such method is to try to minimize the number of OBDD variables of any OBDD created during the computation.

In the algorithms presented above, some of the sets constructed during the 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_{next} 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 some of the OBDDs created in the intermediate computations will have $4n$ OBDD variables. However, by breaking the algorithm operations into 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*. *compose* receives two OBDDs $a(\vec{x},\vec{y})$ and $b(\vec{x},\vec{u})$ over a total number of $3n$ variables and returns an OBDD $c(\vec{y},\vec{u})$. Similarly, *compose_odd* receives $a(\vec{y},\vec{x})$ and $b(\vec{u},\vec{x})$ and returns an OBDD $co(\vec{y},\vec{u})$. As will be explained later, the main advantage of these operations is that they can be implemented using only $2n$ OBDD variables. The two operations are defined as follows:

$$\begin{aligned} compose(a(\vec{x},\vec{y}),b(\vec{x},\vec{u})) &\equiv \exists\vec{x}(a(\vec{x},\vec{y})\wedge b(\vec{x},\vec{u})) && (7.1)\\ compose\_odd(a(\vec{y},\vec{x}),b(\vec{u},\vec{x})) &\equiv \exists\vec{x}(a(\vec{y},\vec{x})\wedge b(\vec{u},\vec{x})) && (7.2)\end{aligned}$$

Computing SIM_{next} and $ReachSIM_{next}$ in Chapter 7.1.1 requires $4n$ variables. We show below how we compute them using only $2n$ OBDD variables.

The two operations SIM_{next} and $ReachSIM_{next}$ can be implemented using *compose* and *compose_odd* as follows. Let $\vec{v_i}$, $\vec{v_i'}$ be the encoding of the states s_i, s_i', respectively. Similarly, let $\vec{v_t}$, $\vec{v_t'}$ be the encoding of s_t, s_t', respectively.

The following Lemmas show the implementation of the various expressions using *compose* and *compose_odd.*

Lemma 7.2.1 $SIM_{next}(\vec{v_i},\vec{v_t}) = SIM_{current}(\vec{v_i},\vec{v_t})\wedge$
$\neg compose\_odd(R_i(\vec{v_i},\vec{v_i'}),\neg compose\_odd(R_t(\vec{v_t},\vec{v_t'}),SIM_{current}(\vec{v_i'},\vec{v_t'})))$.

The proof below starts from the definition of SIM_{next} and transforms the format of the expressions to AND Exist, so that the *compose_odd* operation can be used.

Proof:
According to the definition of simulation:
$SIM_{next}(\vec{v_i},\vec{v_t}) =$
$\forall\vec{v_i'}[R_i(\vec{v_i},\vec{v_i'})\rightarrow\exists\vec{v_t'}\,[R_t(\vec{v_t},\vec{v_t'})\wedge SIM_{current}(\vec{v_i'},\vec{v_t'})]]\ \wedge\ SIM_{current}(\vec{v_i},\vec{v_t}) =$

after Boolean manipulation:
$\forall \vec{v_i'}[\neg R_i(\vec{v_i},\vec{v_i'}) \vee \exists \vec{v_t'}\,[R_t(\vec{v_t},\vec{v_t'}) \wedge SIM_{current}(\vec{v_i'},\vec{v_t'})]] \wedge SIM_{current}(\vec{v_i},\vec{v_t}) =$
Using De Morgan manipulation:
$\neg\exists \vec{v_i'}[R_i(\vec{v_i},\vec{v_i'}) \wedge \neg\exists \vec{v_t'}\,[R_t(\vec{v_t},\vec{v_t'}) \wedge SIM_{current}(\vec{v_i'},\vec{v_t'})]] \wedge SIM_{current}(\vec{v_i},\vec{v_t}) =$
According to the *compose_odd* definition:
$\neg compose\_odd(R_i(\vec{v_i},\vec{v_i'}), \neg compose\_odd(R_t(\vec{v_t},\vec{v_t'}), SIM_{current}(\vec{v_i'},\vec{v_t'}))) \wedge$
$SIM_{current}(\vec{v_i},\vec{v_t})$ □

The *ReachSIM* algorithm requires that the implementation and specification "step" together, each along its respective transition relation. We do this by having one "step" first, followed by the other. This is possible because the transitions of the two structures are independent. The following lemma describes how this breakdown is accomplished:

Lemma 7.2.2 $ReachSIM_{next}(\vec{v_i'},\vec{v_t'}) = ReachSIM_{current}(\vec{v_i'},\vec{v_t'}) \vee$
$(compose(compose(ReachSIM_{current}(\vec{v_i},\vec{v_t}), R_i(\vec{v_i},\vec{v_i'})), R_t(\vec{v_t},\vec{v_t'})) \wedge SIM(\vec{v_i'},\vec{v_t'}))$

Proof:
We define two functions f_{next} and g_{next}, based on which *ReachSIM* is defined.
$f_{next}(\vec{v_t},\vec{v_i'}) = \exists \vec{v_i}(ReachSIM_{current}(\vec{v_i},\vec{v_t}) \wedge R_i(\vec{v_i},\vec{v_i'})) =$
We now represent f_{next} using the *compose* operator:
$= compose(ReachSIM_{current}(\vec{v_i},\vec{v_t}), R_i(\vec{v_i},\vec{v_i'}))$
We define g_{next} :
$g_{next}(\vec{v_i'},\vec{v_t'}) = \exists \vec{v_t}(f_{next}(\vec{v_t},\vec{v_i'}) \wedge R_t(\vec{v_t},\vec{v_t'})) =$
Using the *compose* operator:
$= compose(f_{next}(\vec{v_t},\vec{v_i'}), R_t(\vec{v_t},\vec{v_t'}))$
We now use the g_{next} function, but use it over the $\vec{v_i},\vec{v_t}$ variables, instead of the $\vec{v_i'},\vec{v_t'}$ variables we computed. This renaming operation is standard in model checking, and is equivalent in this case to taking one step backwards. We define $g_{next}(\vec{v_i},\vec{v_t})$ as:
$g_{next}(\vec{v_i},\vec{v_t}) \leftrightarrow (g_{next}(\vec{v_i'},\vec{v_t'}) \wedge (\vec{v_i} = \vec{v_i'}) \wedge (\vec{v_t} = \vec{v_t'}))$
We now can express $ReachSIM_{next}$ by:
$ReachSIM_{next}(\vec{v_i},\vec{v_t}) = (g_{next}(\vec{v_i},\vec{v_t}) \wedge SIM(\vec{v_i},\vec{v_t})) \vee ReachSIM_{current}(\vec{v_i},\vec{v_t})$
□

In the above proof, $f_{next}(\vec{v_i},\vec{v_t})$ holds iff there is a predecessor v_i of v_i' such that v_i and v_t are in the $ReachSIM_{next}$ relation. $g_{next}(\vec{v_i'},\vec{v_t'})$ holds iff there is a predecessor v_t of v_t' such that v_t and v_i' are in the f_{next} relation.

However, this means that v'_i and v'_t are in the g_{next} relation iff v'_i has a predecessor v_i and v'_t has a predecessor v_t, such that v_i and v_t are in the $ReachSIM_{current}$ relation.

The *unimplemented transition* and *many-to-one* comparison criteria can also be implemented with these operations. The other two criteria are defined over $2n$ variables and do not require such manipulations.

Lemma 7.2.3 $UnimplementedTransition(\vec{v_t}, \vec{v'_t}) = R_t(\vec{v_t}, \vec{v'_t}) \wedge$
$compose(compose(\neg R_i(\vec{v_i}, \vec{v'_i}), ReachSIM(\vec{v_i}, \vec{v_t})), ReachSIM(\vec{v'_i}, \vec{v'_t}))$

We find here transitions in R_t that are not in R_i, with respect to $ReachSIM$.
Proof:
We define a function f which maps transitions not in R_i, using the $\vec{v_i}$ variable of the $ReachSIM$ relation. The function creates pairs v'_i, v_t. f is defined as follows:
$f(\vec{v'_i}, \vec{v_t}) = \exists \vec{v_i}(\neg R_i(\vec{v_i}, \vec{v'_i}) \wedge ReachSIM(\vec{v_i}, \vec{v_t})) =$
$= compose(\neg R_i(\vec{v_i}, \vec{v'_i}), ReachSIM(\vec{v_i}, \vec{v_t}))$
We now define a function g, that takes the function f and map the pairs using the $\vec{v_t}$ variable of the $ReachSIM$ relation. g is defined as follows:
$g(\vec{v_t}, \vec{v'_t}) = \exists \vec{v'_i}(f(\vec{v'_i}, \vec{v_t}) \wedge ReachSIM(\vec{v'_i}, \vec{v'_t})) =$
According to the definition of *compose*:
$compose(f(\vec{v'_i}, \vec{v_t}), ReachSIM(\vec{v'_i}, \vec{v'_t}))$
We now can use g in the definition of $UnimplementedTransition$:
$UnimplementedTransition(\vec{v_t}, \vec{v'_t}) = g(\vec{v_t}, \vec{v'_t}) \wedge R_t(\vec{v_t}, \vec{v'_t})$ $\Box$

For computing the $many-to-one$ criterion we use a function that takes two vectors of implementation variables $\vec{v_1}$ and $\vec{v_2}$, and returns *True* if $\vec{v_1} \neq \vec{v_2}$. We use the notation $(\vec{v_1} \neq \vec{v_2})$ to refer to the function. The following lemma describes $many - to - one$ in terms of *compose*:

Lemma 7.2.4 $ManyToOne(\vec{v_t}) =$
$\exists \vec{v_1}(ReachSIM(\vec{v_1}, \vec{v_t}) \wedge compose((\vec{v_1} \neq \vec{v_2}), ReachSIM(\vec{v_2}, \vec{v_t})))$

Proof:
$ManyToOne(\vec{v_t}) =$
According to the definition:
$= \exists \vec{v_1}, \vec{v_2}(ReachSIM(\vec{v_1}, \vec{v_t}) \wedge ReachSIM(\vec{v_2}, \vec{v_t}) \wedge (\vec{v_1} \neq \vec{v_2})) =$
Since $\vec{v_1}$ is independent of $\vec{v_2}$, we can write:
$= \exists \vec{v_1}(ReachSIM(\vec{v_1}, \vec{v_t}) \wedge \exists \vec{v_2}((\vec{v_2} \neq \vec{v_1}) \wedge ReachSIM(\vec{v_2}, \vec{v_t}))) =$
From the definition of *compose* we get:

$= \exists \vec{v_1}(ReachSIM(\vec{v_1}, \vec{v_t}) \wedge compose((\vec{v_1} \neq \vec{v_2}), ReachSIM(\vec{v_2}, \vec{v_t}))) \ \Box$

We have shown 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 $\vec{v_i}$ and the specification variables $\vec{v_t}$. These OBDD variables will be referred to as *untagged.* Similarly, we use the same OBDD variables to represent $\vec{v_i'}$ and $\vec{v_t'}$. They will be referred to as *tagged* OBDD variables.

We also specify that whenever we have relations over both implementation variables and specification variables, the implementation variables are represented by untagged OBDD variables while the specification variables are represented by tagged OBDD variables. Note that the relations R_i, R_t, SIM, $ReachSIM$ are now 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., $\vec{y}$ and $\vec{u}$ are represented by the same OBDD variables.

The implementation of *compose* and *compose_odd* uses nonstandard OBDD operations in such a way that the resulting OBDDs are also defined over the same $2n$ variables. This requires that the OBDD variable change semantics in the result. (For example, in Equation 7.1, $\vec{y}$ is represented by tagged OBDD variables in the input parameters and by untagged variables in the result.) OBDD packages can be extended with these operations. In the next chapter we show how to extend the SMV OBDD package to include the *compose* and *compose_odd* operations.

Chapter 8

SMV Implementation of the *compose* Operation

8.1 OBDD Representation in SMV

The SMV model checker is OBDD-based. It has an internal OBDD package. The OBDDs are represented such that each model variable is represented by two OBDD variables. An even number (level) is assigned to the untagged version of the variable, while a successive odd number is assigned to the tagged version of the variable. The tagged version represents the value of the variable in the next state. The tagged and untagged versions of each variable are used for representing the transition relation, and whenever a value of the next state is used.

Given a known number of model variables and a known order, an OBDD is a DAG with a single root, where each node has two sons (left and right), with increasing variable level, while the leaves of the DAG are two predefined nodes representing the constant ZERO and the constant ONE. The leaves have no sons.

The OBDD package consists of an OBDD pool, in which each intermediate OBDD has a single representation. It also consists of OBDD operations. These include Boolean operations between two OBDD structures, universal and existential quantification, and others. The main advantage of OBDDs is that the sub-OBDDs of a single OBDD and of different OBDDs can be shared in the OBDD pool.

In order to further reduce the memory consumption, we define the transition relations of the implementation model and the specification model over the same set of tagged and untagged variables. By representing R_i, R_t, SIM, $ReachSIM$ over the same set of OBDD variables, sharing between OBDDs is increased.

We represent the SIM and $ReachSIM$ relation with the SMV representation for a transition relation, where the even OBDD variables are used to represent the encoding of the implementation model variables and the odd OBDD variables are used to represent the encoding of the tableau variables.

8.2 The *compose* and *compose_odd* Internals

The $compose(a(\vec{x},\vec{y}),b(\vec{x},\vec{u})) \equiv \exists\vec{x}(a(\vec{x},\vec{y}) \wedge b(\vec{x},\vec{u}))$ operation is an *AND Exists* operation between two OBDDs, $a(\vec{x},\vec{y})$ and $b(\vec{x},\vec{u})$, each of which is an OBDD with up to n even variables and up to n odd variables. The existential operation is over the even OBDD variables $\vec{x}$. The operation is recursive, dealing with one level at a time. The recursive function is called from the roots of the two OBDDs. It goes once from the low OBDD level to the higher OBDD levels and terminates when the leaves are reached. Since the *Exists* operation is between two OBDD variables of the same level, a Boolean *OR* operation may be used. The result is an OBDD with $2n$ OBDD variables, which consists of the tagged OBDD variables of $a(\vec{x},\vec{y})$ and $b(\vec{x},\vec{u})$. Instead of representing a resulting OBDD with $3n$ levels, it is packed so that the tagged variables $\vec{y}$ are represented by even OBDD levels, and the tagged variables $\vec{u}$ are represented by odd OBDD levels. This change in the interpretation of OBDD levels is done together with the quantification of the untagged variables of $a(\vec{x},\vec{y})$ and $b(\vec{x},\vec{u})$.

The $compose\_odd(a(\vec{y},\vec{x}),b(\vec{u},\vec{x})) \equiv \exists\vec{x}(a(\vec{y},\vec{x}) \wedge b(\vec{u},\vec{x}))$ is a dual operation where the existential operation is on the tagged OBDD variables of $a(\vec{y},\vec{x})$ and $b(\vec{u},\vec{x})$. Likewise, the resulting OBDD is an OBDD with only untagged variables. This $3n$ level OBDD is packed into a $2n$ OBDD such that the untagged variables of $a(\vec{y},\vec{x})$ are represented by even OBDD levels, and the untagged variables of $b(\vec{u},\vec{x})$ are represented by odd OBDD levels.

The existential operation over even or odd OBDD variables uses the standard SMV $forsome(c,d)$ operation. This operation performs an existential operation on OBDD d, according to the OBDD variables in OBDD c. Thus, for an existential operation over even OBDD variables, we build a thread-shaped OBDD, denoted $current\_vars$, where only the even variables are linked. Each even node has a left son of the next even variable and a right node of the constant ZERO. The last even node points left to the constant ONE. Thus the existential operation over even OBDD variables uses $forsome(current\_vars, d)$.

Likewise, the existential operation over odd OBDD variables uses the $forsome(next\_vars, d)$ operation, where $next\_vars$ is a thread-shaped OBDD with only odd OBDD levels. The C code of the *compose* and the *compose_odd* operations is given in Appendix A and Appendix B respectively.

Chapter 9

Motivation for the Reduced Tableau

9.1 Meaningful Information and Smaller Tableau Structure

The size of tableau structures as defined in [14] is usually too large to be practical. It may even be much larger than the state space of the given implementation. This is because the state space of such tableaux contains all combinations of subformulas of the specification formula. Such tableaux usually contain many redundant states; these can be removed while still preserving the tableau properties. Otherwise, these states may introduce irrelevant evidence.

Much redundancy 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 want a tableau structure in which each state contains only the set of subformulas required to satisfy the formula. Thus, 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. Therefore, 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 [14]. In such examples we may get a tableau whose size is linear rather than exponential.

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

The tableau may be reduced further 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 formula ψ, the definition of the reduced tableau is derived from the Particle tableau for LTL, presented in [26]. This is done by replacing the use of the $\mathbf{X}$ temporal operator with $\mathbf{AX}$. 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 addition, instead of using the closure of ψ, we use a smaller subset of the closure, which is the set of elementary formulas of ψ. Furthermore, we avoid the construction of redundant successors, thus reducing the tableau structure even more.

Since the reduced tableau is based on the three-value labeling, the definitions of satisfaction and of the simulation preorder are changed accordingly. Our reduced tableau $\tau(\psi)$ for ACTL then has the same properties as the one in [14]:

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

Adopting the reduced tableau also requires that we modify our criteria. This is necessitated by the three-value labeling semantics.

Chapter 10

Reduced Tableau for Safety ACTL

In this chapter we define a *reduced tableau* for the subset of ACTL safety formulas. A tableau is a special form of a Kripke structure, consisting of states labeled with atomic propositions and transitions between the states. As is often the case with tableaux for temporal logics (e.g. [14, 6]), a state of the tableau consists of a set of formulas that are supposed to hold along all paths leaving the state. Unlike typical tableaux, however, the propositions in the states of the reduced tableau are interpreted over a three-valued domain. Thus, a state may include a proposition or its negation, or neither one. The latter case reflects a "don't care" situation, i.e., the proposition may be either true or false in the state. The "don't care" on the propositions also induces a "don't care" on Boolean formulas.

As in [14], we wish the reduced tableau for a formula ψ to satisfy ψ. Furthermore, it should be greater by the *simulation preorder* [29] than any Kripke structure that satisfies ψ. In order to achieve these goals we will adapt both the definitions of $\models$ and that of the simulation preorder to the three-valued tableau.

When constructing the tableau, we distinguish between conjunction and disjunction formulas. In the reduced tableau, if a state satisfies a conjunction, then it also satisfies its two conjuncts. On the other hand, if it satisfies a disjunction, it will usually satisfy only one of the disjuncts, leaving the other as a "don't care".

Our tableau construction has two main stages. In the first stage, we define the states of the reduced tableau. The function *cover* receives a set of ACTL formulas and returns a reduced set of states that "covers" all possibilities to satisfy the given formulas. *cover* is then used to produce the set of initial states as well as the set of successors of a given tableau state. In the next stage, the reachable states and transitions of the reduced tableau are generated on-the-fly. We then eliminate unnecessary successors by further reducing the tableau. Finally, pruning is applied to eliminate states with no successors.

Below we present the formal definitions of the tableau and of the three-value satisfaction and simulation.

10.1 Definitions

We present the definitions we need in order to define *cover* and to construct the tableau. We will use these definitions both in the construction and in proving properties of the tableau.

Let AP_ψ be the set of atomic propositions in an ACTL formula ψ .

Definition 10.1.1 (AP_{np})
$AP_{np} = AP_\psi \cup \{\neg p | p \in AP_\psi\}$.

We now define the set of elementary formulas that includes atomic propositions, their negation, and formulas of the form $AX\varphi$. Sets of elementary formulas will be the states of the tableau. We require states to be consistent, meaning that they do not include a proposition and its negation.

Definition 10.1.2 (el) *[elementary formulas set]*
The set of elementary formulas *of ψ is defined recursively as follows :*
$el(\varphi) = \{\varphi\}$ *where* $\varphi \in AP_{np}$
$el(\varphi_1 \vee \varphi_2) = el(\varphi_1) \cup el(\varphi_2)$
$el(\varphi_1 \wedge \varphi_2) = el(\varphi_1) \cup el(\varphi_2)$
$el(AX\varphi) = \{AX\varphi\} \cup el(\varphi)$
$el(A[\varphi_1 W \varphi_2]) = el(\varphi_2) \cup el(\varphi_1) \cup \{AXA[\varphi_1 W \varphi_2]\}$

The states of the reduced tableau are sets of such formulas.

Definition 10.1.3 (locally consistent)
A set B of elementary formulas is called locally consistent *if*
$\forall \varphi_i, \varphi_j \in AP_{np}$, *if* $\varphi_i, \varphi_j \in B$ *then* $\varphi_i \neq \neg\varphi_j$.

A set of states is represented as a set of sets of elementary formulas. To ease manipulation, we sometimes represent this set by an ACTL formula in disjunctive normal form (DNF). In this DNF formula, each disjunct represents a state and is a conjunction of the elementary formulas in the state. We therefore define conversions from sets of states to DNF formulas and back.

The following function converts a set of ACTL formulas to their conjunction:

Definition 10.1.4 (*conj*)
$conj : 2^{ACTL} \rightarrow ACTL$.
Assume $B \in 2^{ACTL}$, $B = \{\varphi_1, \varphi_2, \ldots, \varphi_k\}$. Then $conj(B) = \bigwedge_{i=1}^{k} \varphi_i$.

The function *conj* receives a set of formulas and returns its conjunction. This function is used to turn a set of ACTL formulas into a single ACTL formula.

We now define a transformation that removes the *AW* operator, and leaves an equivalent formula that all its components are elementary.

Definition 10.1.5 ($\alpha\omega$)
$\alpha\omega\ : ACTL \rightarrow ACTL$
Given an ACTL formula φ, we define an equivalent formula, $\alpha\omega(\varphi)$, denoted as follows:
$\alpha\omega(\varphi) = \varphi$ if $\varphi \in el(\psi)$
$\alpha\omega(\varphi_1 \vee \varphi_2) = \alpha\omega(\varphi_1) \vee \alpha\omega(\varphi_2)$
$\alpha\omega(\varphi_1 \wedge \varphi_2) = \alpha\omega(\varphi_1) \wedge \alpha\omega(\varphi_2)$
$\alpha\omega(A[\varphi_1 W \varphi_2]) = \alpha\omega(\varphi_2) \vee (\alpha\omega(\varphi_1) \wedge AXA[\varphi_1 W \varphi_2])$

Note that φ and $\alpha\omega(\varphi)$ are semantically equivalent, i.e. $\varphi \equiv \alpha\omega(\varphi)$. Note also that $\alpha\omega(\varphi)$ includes only elementary formulas and the $\vee$, $\wedge$ operators.

Definition 10.1.6 (BACTL) *[Boolean ACTL]*
The set of BACTL formulas is the set obtained by mapping all ACTL formulas, using the aw operator.

Definition 10.1.7 (minterm)
A minterm is a formula of the form $\bigwedge_j \varphi_j$ such that each $\varphi_j \in el(\varphi)$.
Note that a single elementary formula is also a minterm.

Since the $\vee$ and $\wedge$ operators are associative, we will sometimes represent a formula as $\bigvee_i \alpha_i$ (or $\bigwedge_i \alpha_i$), ignoring the order in which the $\vee$ operators ($\wedge$ operators, respectively) are applied. This abuse of notation will be applied only when there is no chance of confusion.

Definition 10.1.8 ($\otimes(\bigvee_i \alpha_i, \bigvee_j \beta_j)$)
Let α_i, β_j be minterms. Then,
$\otimes(\bigvee_i \alpha_i, \bigvee_j \beta_j) = \bigvee_{i,j}(\alpha_i \wedge \beta_j)$.

Definition 10.1.9 (dnf)
$\mathbf{dnf} : BACTL \to BACTL$
Given an BACTL formula φ, we define the disjunction normal form, $\mathbf{dnf}(\varphi)$, as follows:
$\mathbf{dnf}(\varphi_1 \vee \varphi_2) = \mathbf{dnf}(\varphi_1) \vee \mathbf{dnf}(\varphi_2)$
$\mathbf{dnf}(\varphi_1 \wedge \varphi_2) = \otimes(\mathbf{dnf}(\varphi_1), \mathbf{dnf}(\varphi_2))$
$\mathbf{dnf}(p) = p$ *if* $p \in el(\psi)$

Note that $\varphi \equiv \mathbf{dnf}(\varphi)$.

Definition 10.1.10 (DNF_ACTL)
The set of DNF_ACTL formulas is the set obtained by mapping all BACTL formulas, using the **dnf** *operator.*

Definition 10.1.11 (consistent minterm)
A minterm $\bigwedge_j \varphi_j$ is called consistent *if there are no $j1, j2$ such that $\varphi_{j1} = \neg\varphi_{j2}$.*

Definition 10.1.12 (*Consistent*)
$Consistent(\varphi) : DNF\_ACTL \to \{\mathbf{false}\} \cup DNF\_ACTL$
Given a DNF_ACTL formula φ, $Consistent(\varphi)$ is obtained from φ by removing the minterms that are not consistent*.*
$Consistent(\varphi) = \mathbf{false}$ *if no minterm in φ is consistent.*
Note that $\varphi \equiv Consistent(\varphi)$.

Given a DNF_ACTL formula φ, we define $Split(\varphi)$ to be a set of sets of elementary formulas, as follows : Convert each minterm to a set consisting of its elementary formulas. Now generate a set consisting of all the above sets.

Definition 10.1.13 (*Split*)
$Split(\varphi) : \{\mathbf{false}\} \cup DNF\_ACTL \to 2^{2^{el(\psi)}}$ *is defined as follows:*
$Split(\varphi_1 \vee \varphi_2) = Split(\varphi_1) \cup Split(\varphi_2)$
$Split(\bigwedge_{k=1}^{n} \varphi_k) = \{\varphi_k | k = 1, \ldots, n\}$
If $\varphi \in el(\psi)$ then $Split(\varphi) = \{\varphi\}$
If $\varphi = \mathbf{false}$ then $Split(\varphi) = \{\}$

Now we can define the cover of a set of ACTL formulas. The cover generates a set of sets of elementary formulas. Each set of elementary formulas will define a state of the tableau. Each such state also corresponds to a minterm, which is the conjunction of all the elementary formulas it includes. Only locally consistent states are generated. The cover of a set of formulas represents the various ways to satisfy this set of formulas. As will be seen later, we use *cover* to define the set of initial states and the set of next states.

Definition 10.1.14 (*cover*)
$cover(B)\ : 2^{ACTL} \rightarrow 2^{2^{el(\psi)}}$
Consider a set of ACTL formulas B. We define
$cover(B) = Split(Consistent(\mathbf{dnf}(\alpha\omega(conj(B)))))$.

We now define the successors of a state. The successors of a state are determined according to the formulas of the form AX. The following definitions will define the next set of states as a set of ACTL formulas (a state).

Definition 10.1.15 (*imps*)
$imps(P)\ : 2^{el(\psi)} \rightarrow 2^{ACTL}$.
A formula g is an implied successor *of a state P if $AXg \in P$. We denote by $imps(P)$ the set of implied successors of P, i.e., $imps(P) = \{g | AXg \in P\}$.*

If P does not include any formula of the form AXg, then $imps(P) = \{\}$. The state $\{\}$ is not committed to satisfying any formula. Thus, it may be the start of any possible path. Furthermore, it may simulate any state. We later see that the only son of state $\{\}$ is the state $\{\}$ itself.

When computing the set of next states (or the initial states), we may want to avoid building successor states whose behavior is included in another successor. We refer to such a successor as a "little brother", and avoid constructing this state. If we don't remove "little brothers", we not only get a larger structure, but may also get false evidence when comparing the implementation with the tableau. Since the behavior of a "little brother" is simulated by a brother state, we do not lose any legal behavior by eliminating it. Elimination of little brothers is defined as follows:

Definition 10.1.16 (*elb*) *[Eliminate Little Brothers]*
$elb : 2^{2^{el(\psi)}} \rightarrow 2^{2^{el(\psi)}}$.

Given a set of sets of elementary formulas B,
$elb(B)$ is the set of minimal elements of B with respect to inclusion.
In other words, $elb(B) = \{s \in B | \forall s_i \in B, s_i \not\subset s\}$.

We define *min_cover* by restricting *cover* to its minimal elements:

Definition 10.1.17 (*min_cover*) *[Minimal cover]*
$min\_cover(B) : 2^{ACTL} \to 2^{2^{el(\psi)}}$.
$min\_cover(B) = elb(cover(B))$.

We now can define the set of successor states of a given state.

Definition 10.1.18 (*Successors*)
$Successors(P) : 2^{el(\psi)} \to 2^{2^{el(\psi)}}$.
$Successors(P) = min\_cover(imps(P))$.

10.2 Tableau Construction

We now describe an iterative algorithm, called REDUCED_TAB, that produces the tableau structure.

Algorithm : REDUCED_TAB

```
$S_{\tau 0} := min\_cover(\{\psi\})$
$S_\tau := S_{\tau 0}$
$R_\tau := \emptyset$
Mark all states in $S_\tau$ as unprocessed
For each unprocessed state $s$ in $S_\tau$ do
    Mark $s$ as processed;
    Define $L_\tau(s) = s \cap AP_{np}$
    $S := Successors(s)$;
    For each $q \in S$
        Add $(s, q)$ to $R_\tau$;
        If $q \notin S_\tau$;
            Add $q$ to $S_\tau$;
            Mark $q$ as unprocessed;
        end if
    end for
end for
```

A state is labeled by propositions from AP and by their negations. Since any state is locally consistent, it will never contain both a proposition and its negation, but it may contain neither of them.

We now prune the structure we obtained, so that any state will have at least one successor. This is necessary because ACTL formulas are interpreted over infinite paths.

Algorithm : PRUNE_TAB

```
Changed:=**true**
while Changed=true
    Changed:=false
    Foreach state $s$ in $S_\tau$ do
        If $s$ has no successors
            Remove $s$ from $S_\tau$
            Remove from $R_\tau$ any edge going to $s$
            Changed:=true
        end if
    end for
end while
```

Applying the algorithms REDUCED_TAB and PRUNE_TAB, we obtain the tableau $\tau(\psi) =< S_\tau, S_{\tau 0}, R_\tau, L_\tau >$ for ψ. The tableau we construct is total since any state that has no successors has been removed.

10.3 Definition of Three-Value Satisfaction and Simulation

In this section we define the satisfaction of an ACTL formula with respect to the reduced tableau. We also define the simulation relation between a structure and a reduced tableau. Extending these definitions to any structure labeled with three-value propositions is straightforward.

Definition 10.3.1 ($\models_3$) *[Satisfaction over a reduced tableau]*
Given a tableau $\tau(\psi)$, a state $s \in S_\tau$ and a formula φ such that $AP_\varphi \subseteq AP\psi$, we define $s \models_3 \varphi$ recursively:

1. *$\varphi = p, p \in AP$ then $s \models_3 p$ iff $p \in L_\tau(s)$.*

2. *$\varphi = \neg p, p \in AP$ then $s \models_3 \neg p$ iff $\neg p \in L\tau(s)$.*

3. *$s \models_3 \varphi_1 \vee \varphi_2$ iff $s \models_3 \varphi_1$ or $s \models_3 \varphi_2$.*

4. *$s \models_3 \varphi_1 \wedge \varphi_2$ iff $s \models_3 \varphi_1$ and $s \models_3 \varphi_2$.*

5. *$s \models_3 AX\varphi$ iff for every successor s_1 of s, $s_1 \models_3 \varphi$.*

6. *$s \models_3 A[\varphi_1 W \varphi_2]$ iff for every path $\pi = s_1 s_2 ...$ from s, one of the two holds: either for all $i > 0$, $s_i \models_3 \varphi_1$, or there exists n such that $s_n \models_3 \varphi_2$ and for all $i < n$, $s_i \models_3 \varphi_1$.*

The only difference between $\models_3$ and the usual definition of $\models$ is in case 2. Also, $s \models_3 p \Rightarrow s \not\models_3 \neg p$, whereas $s \not\models_3 p \not\Rightarrow s \models_3 \neg p$.

Definition 10.3.2 ($\tau(\psi) \models_3 \varphi$)
$\tau(\psi) \models_3 \varphi$ if for every initial state $s_0 \in S_{\tau 0}$, $s_0 \models_3 \varphi$.

Definition 10.3.3 ($\leq_3$) *[simulation relation]*
Given a Kripke structure M' with atomic propositions AP' and a reduced tableau $\tau(\psi)$ with atomic propositions $AP_\psi \subseteq AP'$, a relation $H \subseteq S' \times S_\tau$ is a simulation relation *between M' and $\tau(\psi)$ if and only if for all $s'_0 \in S_{M'0}$ there exists $s_0 \in S_{\tau 0}$ such that $H(s'_0, s_0)$ and for all $s' \in S_{M'}$, and $s \in S_\tau$, if $H(s', s)$, then the following conditions hold.*

1. *For every $p \in AP_\psi$, if $p \in L_{M'}(s')$ then $\neg p \notin L_\tau(s)$.*

2. *For every $p \in AP_\psi$, if $p \notin L_{M'}(s')$ then $p \notin L_\tau(s)$.*

3. *For every state s'_1 such that $R_{M'}(s', s'_1)$, there is a state $s_1 \in S_\tau$ such that $R_\tau(s, s_1)$ and $H(s'_1, s_1)$.*

Chapter 11

Properties of the Tableau

To prove that there is a simulation relation between a given structure that satisfies ψ and our tableau construction, we define a relation H. We show the relationship between H and *cover*, and use it to prove that H is a simulation relation. The proof is done using properties of *cover*. The properties of *cover* are based on the properties of its components: *conj*, $\alpha\omega$, **dnf**, *Split*, *Consistent*.

11.1 $\alpha\omega$, **dnf**, *Split*, *Consistent* Properties

A formula is said to be of **dnf** form if it has the form $\bigvee_i \varphi_i$, where each φ_i is a minterm.

Lemma 11.1.1
Let φ be a BACTL formula. Then $\mathbf{dnf}(\varphi) = \bigvee_i \varphi_i$*, where each φ_i is a minterm.*

Proof:
The proof is by induction on the number of $\vee$ and $\wedge$ operators in φ.
Base: Zero $\vee$ and $\wedge$ operators.
Because there are no $\vee$ and $\wedge$ operators, φ consists of a single elementary formula, i.e. $\varphi \in el(\psi)$. Thus, by the **dnf** definition ,$\mathbf{dnf}(\varphi) = \varphi$.

Step: Assume the correctness for φ with k operators, and prove the correctness for $k+1$ operators.
Case 1 : The topmost operator is $\vee$. Assume $\varphi = \varphi_1 \vee \varphi_2$.
In this case $\mathbf{dnf}(\varphi) = \mathbf{dnf}(\varphi_1 \vee \varphi_2) = \mathbf{dnf}(\varphi_1) \vee \mathbf{dnf}(\varphi_2)$, where φ_1, φ_2 have at most k operators. The induction hypothesis holds for φ_1, φ_2. Thus

$\mathbf{dnf}(\varphi_1) = \bigvee_i \varphi_{1i}$, where φ_{1i} are minterms. $\mathbf{dnf}(\varphi_2) = \bigvee_j \varphi_{2j}$, where φ_{2j} are minterms. Thus $\mathbf{dnf}(\varphi) = \bigvee_i \varphi_{1i} \vee \bigvee_j \varphi_{2j}$. By associativity of $\vee$ operators, this can be represented as a single $\bigvee$ over all participating minterms. This $\bigvee$ is of the desired form.

Case 2 : The topmost operator is $\wedge$. Assume $\varphi = \varphi_1 \wedge \varphi_2$.
In this case $\mathbf{dnf}(\varphi) = \mathbf{dnf}(\varphi_1 \wedge \varphi_2) = \otimes(\mathbf{dnf}(\varphi_1), \mathbf{dnf}(\varphi_2))$ where φ_1, φ_2 have at most k operators. The induction hypothesis holds for φ_1, φ_2. Thus $\mathbf{dnf}(\varphi_1) = \bigvee_i \varphi_{1i}$, where φ_{1i} are minterms. $\mathbf{dnf}(\varphi_2) = \bigvee_j \varphi_{2j}$, where φ_{2j} are minterms. By the definition of the $\otimes$ operator, $\mathbf{dnf}(\varphi) = \bigvee_{i,j}(\varphi_{1i} \wedge \varphi_{2j})$. An $\wedge$ of two minterms is also a minterm. Thus the resulting expression is again of the desired form.
$\square$

Lemma 11.1.2 *[***dnf** *of* **dnf** *form]*
If φ is in **dnf** *form, then* $\mathbf{dnf}(\varphi) = \varphi$.

Proof:
The proof is by induction on the number of $\vee$ and $\wedge$ operators.
Base: Zero $\vee$ and $\wedge$ operators.
Because there are no $\vee$ and $\wedge$ operators, φ is elementary, i.e., $\varphi \in el(\psi)$. Thus $\mathbf{dnf}(\varphi) = \varphi$.

Step: Assume the correctness for φ with k operators, and prove the correctness for $k+1$ operators.
Case 1: The topmost operator is $\vee$. $\varphi = \varphi_1 \vee \varphi_2$. Since φ is in **dnf** form, $\varphi_1 = \bigvee_i \varphi_i$ and $\varphi_2 = \bigvee_j \varphi_j$, where φ_i, φ_j are minterms. Thus φ_1 and φ_2 are in **dnf** form. By the induction hypothesis, $\mathbf{dnf}(\varphi_1) = \varphi_1$ and $\mathbf{dnf}(\varphi_2) = \varphi_2$. $\mathbf{dnf}(\varphi_1 \vee \varphi_2) = \mathbf{dnf}(\varphi_1) \vee \mathbf{dnf}(\varphi_2) = \varphi_1 \vee \varphi_2 = \varphi$.

Case 2: The topmost operator is $\wedge$. $\varphi = \varphi_1 \wedge \varphi_2$. Since φ is in **dnf** form, it must consist of a single minterm. Thus, each of φ_1, φ_2 is also a single minterm and therefore in **dnf** form. Therefore, by the induction hypothesis, $\mathbf{dnf}(\varphi_1) = \varphi_1$ and $\mathbf{dnf}(\varphi_2) = \varphi_2$. $\mathbf{dnf}(\varphi_1 \wedge \varphi_2) = \otimes(\mathbf{dnf}(\varphi_1), \mathbf{dnf}(\varphi_2)) = \otimes(\varphi_1, \varphi_2) = \varphi_1 \wedge \varphi_2 = \varphi$. The last equality is a consequence of the definition of $\otimes$, which, when applied to two single minterms, results in a single minterm. $\square$

Corollary 11.1.3 (dnf reactivation property) $\mathbf{dnf}(\varphi) = \mathbf{dnf}(\mathbf{dnf}(\varphi))$

Lemma 11.1.4
Let φ_1, φ_2 be in **dnf** *form, i.e., $\varphi_1 = \bigvee_i \varphi_i$ and $\varphi_2 = \bigvee_j \varphi_j$, where φ_i, φ_j are minterms. Then* $\mathbf{dnf}(\varphi_1 \wedge \varphi_2) = \bigvee_{i,j}(\varphi_i \wedge \varphi_j)$.

Proof:
$\mathbf{dnf}(\varphi_1 \wedge \varphi_2) =$ by **dnf** definition:
$\otimes(\mathbf{dnf}(\varphi_1), \mathbf{dnf}(\varphi_2)) = \otimes(\mathbf{dnf}(\bigvee_i \varphi_i), \mathbf{dnf}(\bigvee_j \varphi_j)) =$
by **dnf** reactivation property Corollary 11.1.3:
$= \otimes(\bigvee_i \varphi_i, \bigvee_j \varphi_j) =$
by $\otimes$ definition:
$= \bigvee_{i,j}(\varphi_i \wedge \varphi_j)$. □

Corollary 11.1.5 $\mathbf{dnf}(\varphi_1 \wedge \varphi_2) = \mathbf{dnf}(\mathbf{dnf}(\varphi_1) \wedge \mathbf{dnf}(\varphi_2))$

Lemma 11.1.6
$Consistent(\alpha \vee \beta) = Consistent(\alpha) \vee Consistent(\beta)$.

Proof:
The proof is by showing that any minterm from the left-hand side exists on the right-hand side, and vice versa. Consider a minterm α_i from $Consistent(\alpha \vee \beta)$. α_i must originate from either α or β, and α_i is consistent. We can conclude that α is a minterm of $Consistent(\alpha)$ or $Consistent(\beta)$ respectively.
On the other hand, consider a minterm α_i from $Consistent(\alpha) \vee Consistent(\beta)$. α_i is consistent and must come from $Consistent(\alpha)$ or from $Consistent(\beta)$ or from both. We thus know that α_i is a minterm of α or β, and also of $\alpha \vee \beta$. Since α_i is consistent, it is also a minterm of $Consistent(\alpha \vee \beta)$. □

Lemma 11.1.7
Let α_i be a minterm, and assume $\alpha = Split(\alpha_i)$. Then α is locally consistent iff $\alpha_i = Consistent(\alpha_i)$.

This property results from the definitions of *consistent* and *locally consistent.*

Lemma 11.1.8
Assume $\bigvee_j \beta_j = Consistent(\bigvee_k \alpha_k)$. Then for every j there is a k such that $\beta_j = \alpha_k$.

Proof:
This property is immediate by the definition of the *Consistent* operation, since the operation only removes some of its minterms. □

Lemma 11.1.9
Assume $\alpha \in Split(Consistent(\bigvee_i \alpha_i))$. *Then there is* m *such that* $\alpha = Split(\alpha_m)$ *and* α *is locally consistent.*

This again is based on the fact that the *Consistent* operation only eliminates minterms. This property can be immediately derived by the definition of *Split* and Lemma 11.1.8.

Lemma 11.1.10
Assume $\alpha = Split(\alpha_m)$,$\beta = Split(\beta_n)$, α_m,β_n *are minterms, and* $\alpha \cup \beta$ *is locally consistent. Then* $\alpha_m \wedge \beta_n$ *is consistent.*

Proof:
The proof is by contradiction. Assume the minterm $\alpha_m \wedge \beta_n$ is not consistent. Then it contains an atomic proposition p and its negation. If p and $\neg p$ both come from either α_m or from β_n, then either α or β are not locally consistent. This contradicts the fact that $\alpha \cup \beta$ is locally consistent.
If p comes from α_m, and $\neg p$ comes from β_n (or vice versa), then $p \in \alpha$ and $\neg p \in \beta$, thus contradicting the fact that $\alpha \cup \beta$ is locally consistent. □

Lemma 11.1.11
If α_m,β_n *are minterms, then* $Split(\alpha_m \wedge \beta_n) = Split(\alpha_m) \cup Split(\beta_n)$.

This results from the definition of *Split*.

Lemma 11.1.12 *[$\alpha\omega$ Reactivation property]*
$\alpha\omega(\alpha\omega(\varphi)) = \alpha\omega(\varphi)$.

Proof:
The proof is by induction on the number of operators in φ.
Base : $\varphi \in el(\psi)$. This is immediate from the definition.
Induction step : Assume the property is valid for formulas with k operators. We show that it is valid for $k+1$.

- $\varphi = \varphi_1 \vee \varphi_2$
  $\alpha\omega(\alpha\omega(\varphi)) = \alpha\omega(\alpha\omega(\varphi_1 \vee \varphi_2)) =$

By $\alpha\omega$ definition
$= \alpha\omega(\alpha\omega(\varphi_1) \vee \alpha\omega(\varphi_2)) =$
By $\alpha\omega$ definition
$= \alpha\omega(\alpha\omega(\varphi_1)) \vee \alpha\omega(\alpha\omega(\varphi_2)) =$
Since φ_1, φ_2 have at most k operators, the induction hypothesis can be used:
$= \alpha\omega(\varphi_1) \vee \alpha\omega(\varphi_2) = \alpha\omega(\varphi_1 \vee \varphi_2) = \alpha\omega(\varphi)$

- $\varphi = \varphi_1 \wedge \varphi_2$
  Similar to the case of $\vee$.

- $\varphi = A[\varphi_1 W \varphi_2]$
  By $\alpha\omega$ definition
  $\alpha\omega(\alpha\omega(\varphi)) = \alpha\omega(\alpha\omega(\varphi_2) \vee (\alpha\omega(\varphi_1) \wedge AXA[\varphi_1 W \varphi_2])) =$
  By definition of $\alpha\omega$ for $\vee$ and $\wedge$:
  $= \alpha\omega(\alpha\omega(\varphi_2)) \vee (\alpha\omega(\alpha\omega(\varphi_1)) \wedge \alpha\omega(AXA[\varphi_1 W \varphi_2])) =$
  By the definition of $\alpha\omega(AX)$:
  $= \alpha\omega(\alpha\omega(\varphi_2)) \vee (\alpha\omega(\alpha\omega(\varphi_1)) \wedge AXA[\varphi_1 W \varphi_2]) =$
  Since φ_1, φ_2 have at most k operators the induction hypothesis can be used :
  $= \alpha\omega(\varphi_2) \vee (\alpha\omega(\varphi_1) \wedge AXA[\varphi_1 W \varphi_2]) =$
  By the definition of $\alpha\omega(AW)$:
  $= \alpha\omega(A[\varphi_1 W \varphi_2])$

□

Lemma 11.1.13
$cover(\alpha\omega(\varphi)) = cover(\varphi)$

This results from the definition of *cover* and the $\alpha\omega$ Reactivation property Lemma 11.1.12.

Lemma 11.1.14
$\alpha\omega(A[\varphi_1 W \varphi_2]) = \alpha\omega(\varphi_2 \vee (\varphi_1 \wedge AXA[\varphi_1 W \varphi_2]))$.

This results from the definition of $\alpha\omega$ and the reactivation of $\alpha\omega$.

11.2 *cover* Properties

For a single formula φ, $cover(\{\varphi\})$ produces a set of states. Recall that each of the states is a set of elementary formulas and corresponds to a minterm.

$cover(\{\varphi\})$ describes the different ways to satisfy φ. φ is satisfied iff some state in $cover(\{\varphi\})$ is satisfied. Each state is locally consistent, meaning that it contains neither an atomic proposition p nor its negation.

Definition 11.2.1 (X_{con})
Let $A, B \in 2^{2^{el(\psi)}}$ be sets of sets of elementary formulas. We define $A\ X_{con}\ B = \{A_i \cup B_j | A_i \in A,\ B_j \in B$, and $A_i \cup B_j$ is locally consistent $\}$.

Lemma 11.2.1
$cover(\{\varphi_1 \vee \varphi_2\}) = cover(\{\varphi_1\}) \cup cover(\{\varphi_2\})$

Proof:
By the definition of $conj$ for a set of size one, we get
$conj(\{\varphi_1 \vee \varphi_2\}) = conj(\{\varphi_1\}) \vee conj(\{\varphi_2\})$.
We apply the $\alpha\omega$ operator, and get
$\alpha\omega(conj(\{\varphi_1 \vee \varphi_2\})) = \alpha\omega(conj(\{\varphi_1\}) \vee conj(\{\varphi_2\}))$.
By the definition of $\alpha\omega$,
$\alpha\omega(conj(\{\varphi_1 \vee \varphi_2\})) = \alpha\omega(conj(\{\varphi_1\})) \vee \alpha\omega(conj(\{\varphi_2\})))$.
Denote $\varphi_1' = \alpha\omega(conj(\{\varphi_1\}))$ and $\varphi_2' = \alpha\omega(conj(\{\varphi_2\}))$.
Recall that $\alpha\omega(conj(\{\varphi_1 \vee \varphi_2\})) = \varphi_1' \vee \varphi_2'$.

Consider $\varphi_1' \vee \varphi_2'$. By the definition of **dnf**,
$\mathbf{dnf}(\varphi_1' \vee \varphi_2') = \mathbf{dnf}(\varphi_1') \vee \mathbf{dnf}(\varphi_2')$.
By applying $Consistent$ to both sides we get
$Consistent(\mathbf{dnf}(\varphi_1' \vee \varphi_2')) = Consistent(\mathbf{dnf}(\varphi_1') \vee \mathbf{dnf}(\varphi_2'))$
By Lemma 11.1.6 we get
$Consistent(\mathbf{dnf}(\varphi_1' \vee \varphi_2')) = Consistent(\mathbf{dnf}(\varphi_1')) \vee Consistent(\mathbf{dnf}(\varphi_2'))$.
We now apply the $Split$ operator on both sides:
$Split(Consistent(\mathbf{dnf}(\varphi_1' \vee \varphi_2'))) =$
$= Split(Consistent(\mathbf{dnf}(\varphi_1')) \vee Consistent(\mathbf{dnf}(\varphi_2')))$
and by $Split$ definition we get :
$Split(Consistent(\mathbf{dnf}(\varphi_1' \vee \varphi_2'))) =$
$= Split(Consistent(\mathbf{dnf}(\varphi_1'))) \cup Split(Consistent(\mathbf{dnf}(\varphi_2')))$.
This implies that $cover(\{\varphi_1 \vee \varphi_2\}) = cover(\{\varphi_1\}) \cup cover(\{\varphi_2\})$ $\Box$

Lemma 11.2.2
$cover(\{\varphi_1 \wedge \varphi_2\}) = cover(\{\varphi_1\})\, X_{con}\, cover(\{\varphi_2\})$

Proof:
By the definition of $conj$ for a set of size one, we get

$conj(\{\varphi_1 \wedge \varphi_2\}) = conj(\{\varphi_1\}) \wedge conj(\{\varphi_2\})$.
We apply the $\alpha\omega$ operator and get
$\alpha\omega(conj(\{\varphi_1 \wedge \varphi_2\})) = \alpha\omega(conj(\{\varphi_1\}) \wedge conj(\{\varphi_2\}))$.
By the definition of the $\alpha\omega$ operator,
$\alpha\omega(conj(\{\varphi_1 \wedge \varphi_2\})) = \alpha\omega(conj(\{\varphi_1\})) \wedge \alpha\omega(conj(\{\varphi_2\})))$.
We apply **dnf** on both sides:
$\mathbf{dnf}(\alpha\omega(conj(\{\varphi_1 \wedge \varphi_2\}))) = \mathbf{dnf}(\alpha\omega(conj(\{\varphi_1\})) \wedge \alpha\omega(conj(\{\varphi_2\})))$.
According to the **dnf** property, (Corollary 11.1.5),
$\mathbf{dnf}(\alpha\omega(conj(\{\varphi_1 \wedge \varphi_2\}))) = \mathbf{dnf}(\mathbf{dnf}(\alpha\omega(conj(\{\varphi_1\}))) \wedge \mathbf{dnf}(\alpha\omega(conj(\{\varphi_2\}))))$.
We now define $\bigvee_i \alpha_i = \mathbf{dnf}(\alpha\omega(conj(\{\varphi_1\})))$ such that each α_i is a minterm.
Similarly, $\bigvee_j \beta_j = \mathbf{dnf}(\alpha\omega(conj(\{\varphi_2\})))$. According to **dnf** property, Lemma 11.1.4, we get
$\mathbf{dnf}(\alpha\omega(conj(\{\varphi_1 \wedge \varphi_2\}))) = \mathbf{dnf}(\bigvee_i \alpha_i \wedge \bigvee_j \beta_j) = \bigvee_{i,j}(\alpha_i \wedge \beta_j)$

We now prove that
$Split(Consistent(\bigvee_{i,j} \alpha_i \wedge \beta_j)) = Split(Consistent(\bigvee_i \alpha_i))\, X_{con}$
$Split(Consistent(\bigvee_j \beta_j))$

The proof is by mutual inclusion.
Consider an element from the right-hand side. According to the definition of X_{con}, this element is a union of two elements,
$\alpha \cup \beta \in Split(Consistent(\bigvee_i \alpha_i))\, X_{con}\, Split(Consistent(\bigvee_j \beta_j))$,
such that $\alpha \in Split(Consistent(\bigvee_i \alpha_i))$ and $\beta \in Split(Consistent(\bigvee_j \beta_j))$.
Furthermore, $\alpha \cup \beta$ is locally consistent. We can conclude that both α and β are locally consistent.

$\alpha \in Split(Consistent(\bigvee_i \alpha_i))$. Thus, there exists m such that $\alpha = Split(\alpha_m)$, and α is locally consistent. Thus α_m is consistent. Similarly, there exists n such that $\beta = Split(\beta_n)$, β is locally consistent, and thus β_n is consistent.

Consider the minterm $\alpha_m \wedge \beta_n$. Since $\alpha \cup \beta$ is locally consistent, then according to Lemma 11.1.10, $\alpha_m \wedge \beta_n$ is consistent. Note also that $Split(Consistent(\alpha_m \wedge \beta_n)) = Split(\alpha_m \wedge \beta_n)$. By the definition of $Split$, we can conclude that $Split(\alpha_m \wedge \beta_n) \in Split(Consistent(\bigvee_{i,j} \alpha_i \wedge \beta_j))$.
On the other hand, $Split(\alpha_m \wedge \beta_n) = Split(\alpha_m) \cup Split(\beta_n) = \alpha \cup \beta$. Thus $\alpha \cup \beta$ is an element of the left-hand side.

We now show inclusion in the other direction.

Assume $\gamma \in \mathit{Split}(\mathit{Consistent}(\bigvee_{i,j} \alpha_i \wedge \beta_j))$. Since $\mathit{Consistent}$ only eliminates minterms, and by the definition of Split, there exists some m and some n such that $\gamma = \mathit{Split}(\alpha_m \wedge \beta_n)$. Furthermore, since $\alpha_m \wedge \beta_n$ is consistent, then γ is locally consistent.

We define $\alpha = \mathit{Split}(\alpha_m)$ and $\beta = \mathit{Split}(\beta_n)$. Since α_m and β_n are minterms of elementary formulas, we have $\gamma = \alpha \cup \beta$.
Since $\alpha \cup \beta$ is locally consistent, we conclude that α and β are also locally consistent.

We show that $\alpha \in \mathit{Split}(\mathit{Consistent}(\bigvee_i \alpha_i))$ and
$\beta \in \mathit{Split}(\mathit{Consistent}(\bigvee_j \beta_j))$.

Recall that α_m is a minterm in $\bigvee_i \alpha_i$ and that β_n is a minterm in $\bigvee_j \beta_j$. We again apply the fact that α_m is consistent, the definition of Split, and the fact that α_m is a minterm in $\bigvee_i \alpha_i$. We can then claim that $\alpha \in \mathit{Split}(\mathit{Consistent}(\bigvee_i \alpha_i))$, and similarly, $\beta \in \mathit{Split}(\mathit{Consistent}(\bigvee_j \beta_j))$.
We add the fact that $\alpha \cup \beta$ is locally consistent, and conclude that
$\gamma \in \mathit{Split}(\mathit{Consistent}(\bigvee_i \alpha_i))\, X_{con}\, \mathit{Split}(\mathit{Consistent}(\bigvee_j \beta_j))$.

From the two inclusions we conclude that
$\mathit{cover}(\{\varphi_1 \wedge \varphi_2\}) = \mathit{cover}(\{\varphi_1\})\, X_{con}\, \mathit{cover}(\{\varphi_2\})$. $\Box$

Lemma 11.2.3
$\mathit{cover}(\{A[\varphi_1 W \varphi_2]\}) =$
$\mathit{cover}(\{\varphi_2\}) \cup (\mathit{cover}(\{\varphi_1\})\, X_{con}\, \mathit{cover}(\{AXA[\varphi_1 W \varphi_2]\}))$

Proof:
$\mathit{cover}(\{A[\varphi_1 W \varphi_2]\}) =$
$= \mathit{Split}(\mathit{Consistent}(\mathbf{dnf}(\alpha\omega(\mathit{conj}(\{A[\varphi_1 W \varphi_2]\}))))) =$
A set of size one:
$= \mathit{Split}(\mathit{Consistent}(\mathbf{dnf}(\alpha\omega(A[\varphi_1 W \varphi_2])))) =$
From $\alpha\omega$ definition :
$= \mathit{Split}(\mathit{Consistent}(\mathbf{dnf}(\alpha\omega(\varphi_2) \vee (\alpha\omega(\varphi_1) \wedge AXA[\varphi_1 W \varphi_2])))) =$
Again, a set of size one :
$= \mathit{Split}(\mathit{Consistent}(\mathbf{dnf}(\alpha\omega(\mathit{conj}(\{\alpha\omega(\varphi_2) \vee (\alpha\omega(\varphi_1) \wedge AXA[\varphi_1 W \varphi_2])\}))))) =$

Which is the definition of :
$= \mathit{cover}(\{\alpha\omega(\varphi_2) \vee (\alpha\omega(\varphi_1) \wedge AXA[\varphi_1 W \varphi_2])\}) =$
According to $\vee$ property, Lemma 11.2.1:
$= \mathit{cover}(\{\alpha\omega(\varphi_2)\}) \cup \mathit{cover}(\{\alpha\omega(\varphi_1) \wedge AXA[\varphi_1 W \varphi_2]\}) =$
According to $\wedge$ property, Lemma 11.2.2:

$= cover(\{\alpha\omega(\varphi_2)\}) \cup (cover(\{\alpha\omega(\varphi_1)\})\, X_{con}\, cover(\{AXA[\varphi_1 W \varphi_2]\})) \ \Box$

Lemma 11.2.4
$cover(\{\varphi_1 \wedge \varphi_2 \wedge ... \wedge \varphi_k\}) = cover(\{\varphi_1, \varphi_2, ..., \varphi_k\})$.

Proof:
Lemma 11.2.4 results from the definition of *conj* :
$cover(\{\varphi_1, \varphi_2, ..., \varphi_k\}) =$
$Split(Consistent(\mathbf{dnf}(\alpha\omega(conj(\{\varphi_1, \varphi_2, ..., \varphi_k\}))))) =$
By *conj* definition :
$= Split(Consistent(\mathbf{dnf}(\alpha\omega(conj(\{\varphi_1 \wedge \varphi_2 \wedge ... \wedge \varphi_k\}))))) =$
$= cover(\{\varphi_1 \wedge \varphi_2 \wedge ... \wedge \varphi_k\}) \ \Box$

Lemma 11.2.5
For any formula φ, and every $s \in cover(\{\varphi\})$, s is locally consistent.

Proof:
This is straightforward from the definition of cover and the fact that every element of $Split(Consistent(B))$ is locally consistent. $\Box$
This Lemma implies that any tableau state is locally consistent.

11.3 Properties of the Reduced Tableau

Fix a structure $M' =< S', S'_0, R', L' >$, and an ACTL formula ψ.

Definition 11.3.1 (H)
We define a relation $H \subseteq S' \times 2^{el(\psi)}$ by :
$H = \{(s', s) | \forall \varphi \in s \, : \, s' \models \varphi\}$

Lemma 11.3.1 *[cover lemma]*
Given a state s' in a model M', and an ACTL property φ such that $s' \models \varphi$, then there is $s \in cover(\{\varphi\})$ and $H(s', s)$.

Proof:
The proof is by induction on the structure of φ.
Base: $\varphi = \varphi_i$, an elementary formula.
By the definition of *cover*, $cover(\{\varphi_i\}) = \{\{\varphi_i\}\}$. We choose s to be the only member in $cover(\{\varphi_i\})$. We have $\varphi_i \in s$, which is the only elementary formula in s. We are also given that $s' \models \varphi_i$. By the definition of H we see

that $H(s', s)$.

Induction step: Assume correctness for a formula with k operators, and show correctness for a formula with $k+1$ operators.

1. $\varphi = \varphi_1 \vee \varphi_2$.
   For φ_1 and φ_2, the induction hypothesis holds. $s' \models \varphi$ implies that either $s' \models \varphi_1$ or $s' \models \varphi_2$.
   Assume without loss of generality that the first case is true, and let s be the state from $cover(\{\varphi_1\})$ such that $H(s', s)$. By the induction hypothesis, such a state exists. Lemma 11.2.5 implies that any tableau state is locally consistent. In both cases there exists a state $s \in cover(\{\varphi_1\}) \cup cover(\{\varphi_2\})$. By Lemma 11.2.1, $cover(\{\varphi\}) = cover(\{\varphi_1\}) \cup cover(\{\varphi_2\})$.
   Therefore there exists $s \in cover(\{\varphi_1 \vee \varphi_2\})$ and $H(s', s)$.

2. $\varphi = \varphi_1 \wedge \varphi_2$.
   $s' \models \varphi$ implies that $s' \models \varphi_1$ and $s' \models \varphi_2$. Let s_1 be a state from $cover(\{\varphi_1\})$ such that $H(s', s_1)$. Such a state exists by the induction hypothesis. Similarly, let $s_2 \in cover(\{\varphi_2\})$ and $H(s', s_2)$. We choose s to be $s_1 \cup s_2$.

   We now show that $H(s', s)$. Consider $\varphi_j \in s$. If $\varphi_j \in s_1$, then by $H(s', s_1)$ we have $s' \models \varphi_j$. If $\varphi_j \in s_2$, then by $H(s', s_2)$ we have $s' \models \varphi_j$. Thus, for every $\varphi_j \in s : s' \models \varphi_j$, and so $H(s', s)$.

   We now show that s is locally consistent. From Lemma 11.2.5 we know that both s_1 and s_2 are locally consistent. Assume by way of contradiction that s is not locally consistent. In this case there exists some atomic proposition φ_j (or its negation), $\varphi_j \in s_1$ and $\neg\varphi_j \in s_2$. Since $H(s', s_1)$ and $H(s', s_2)$, we have $s' \models \varphi_j$ and $s' \models \neg\varphi_j$, a contradiction.

   By Lemma 11.2.2 $cover(\{\varphi\}) = cover(\{\varphi_1\})\, X_{con}\, cover(\{\varphi_2\})$.
   Since s_1 is in $cover(\{\varphi_1\})$, s_2 is in $cover(\{\varphi_2\})$, and s is locally consistent, $s \in cover(\{\varphi_1 \wedge \varphi_2\})$.

3. $\varphi = A[\varphi_1 W \varphi_2]$.
   For φ_1 and φ_2, the induction hypothesis holds. $s' \models A[\varphi_1 W \varphi_2]$ implies that $s' \models \varphi_2$ (first case) or $s' \models \varphi_1$ and $s' \models AXA[\varphi_1 W \varphi_2]$ (second case).

In the first case, let s be a state from $cover(\{\varphi_2\})$ such that $H(s', s)$. By Lemma 11.2.3, $cover(\{A[\varphi_1 W \varphi_2]\}) =$
$cover(\{\varphi_2\}) \cup (cover(\{\varphi_1\})\, X_{con}\, cover(\{AXA[\varphi_1 W \varphi_2]\}))$. Therefore, $s \in cover(\{A[\varphi_1 W \varphi_2]\})$ and $H(s', s)$.
In the second case, let s_1 be a state from $cover(\{\varphi_1\})$ such that $H(s', s_1)$. Now let $s = s_1 \cup \{AXA[\varphi_1 W \varphi_2]\}$.
Consider $\varphi_j \in s$. By $H(s', s_1)$ we know that $\varphi_j \in s_1$ implies $s' \models \varphi_j$. In addition, $s' \models \{AXA[\varphi_1 W \varphi_2]\}$. Thus, $H(s', s)$. Moreover, since s_1 is locally consistent, so is s, and adding $AXA[\varphi_1 W \varphi_2]$ does not change this.

We note that $cover(\{AXA[\varphi_1 W \varphi_2]\}) = \{AXA[\varphi_1 W \varphi_2]\}$.
Since $s = s_1 \cup \{AXA[\varphi_1 W \varphi_2]\}$, and s is locally consistent, we conclude that $s \in cover(\{\varphi_1\})\, X_{con}\, cover(\{AXA[\varphi_1 W \varphi_2]\})$ and $H(s', s)$.
In both cases

$s \in cover(\{\varphi_2\}) \cup (cover(\{\varphi_1\})\, X_{con}\, cover(\{AXA[\varphi_1 W \varphi_2]\}))$, which results in $s \in cover(\{A[\varphi_1 W \varphi_2]\})$ and $H(s', s)$.

□

Lemma 11.3.2 *[min_cover lemma]*
Given a state s' in a model M', and an ACTL property φ such that $s' \models \varphi$, then there is a state $s \in min\_cover(\{\varphi\})$ and $H(s', s)$.

Proof:
By Lemma 11.3.1 there is $s_c \in cover(\{\varphi\})$ and $H(s', s_c)$. We need to show that there is a state $s_{mc} \in elb(cover(\{\varphi\})$ and $H(s', s_{mc})$. By the definition of H, $\forall \varphi \in s_c : s' \models \varphi$. By the definition of elb, there is a state $s_{mc} \in elb(cover(\{\varphi\})$ such that $s_{mc} \subseteq s_c$. By the above, $H(s', s_{mc})$. □

The following two theorems imply one direction of the first property of a tableau for ACTL formulas, as defined in definition 3.0.3:
$M' \models \psi \Rightarrow M' \leq_3 \tau(\psi)$.

Theorem 11.3.3 *[Simulation of Tableau]*
For every structure M', $M' \models \psi$, H is a simulation relation (according to definition 10.3.3) between M' and $\tau(\psi)$.

Proof:
$M' \models \psi$ implies that for every initial state s'_0 of M', $s'_0 \models \psi$. Therefore, by Lemma 11.3.2, we have that for every initial state $s'_0 \in S'$ there exists a state $s_0 \in min\_cover(\{\psi\})$ such that $H(s'_0, s_0)$. By the definition of $\tau(\psi)$, state s_0 is an initial state of $\tau(\psi)$.
We show that s'_0 and s_0 agree with regard to their labels. First we show that $\neg p \in L_\tau(s_0) \;\rightarrow\; p \notin L_{M'}(s'_0)$. Since $H(s'_0, s_0)$, then by definition 11.3.1, $\neg p \in L_\tau(s_0)$ results $s'_0 \models \neg p$. Thus, by the definition of satisfaction, $p \notin L_{M'}(s'_0)$. Similarly, $p \in L_\tau(s_0) \;\rightarrow\; \neg p \notin L_{M'}(s'_0)$.

We now show that if $H(s', s)$ and $R_{M'}(s', s'_1)$, then there exists s_1 such that $H(s'_1, s_1)$ and $R_\tau(s, s_1)$.
Let $\{AX\varphi_1, AX\varphi_2, ..., AX\varphi_k\}$ be the set of all elementary formulas in s of the form $AX\varphi$. By the definition of H we know that $s' \models AX\varphi_i$ for $i = 1, 2, .., k$.
Let s'_1 be some successor of state s'. Then $s'_1 \models \varphi$, where $\varphi = \varphi_1 \wedge \varphi_2 \wedge ... \wedge \varphi_k$. Lemma 11.3.2 implies that there exists a state $s_1 \in min\_cover(\{\varphi\})$ such that $H(s'_1, s_1)$.

By the construction of $\tau(\psi)$, $imps(s) = \{\varphi_1, \varphi_2, ..., \varphi_k\}$. Thus, the set of successors of s is the set $elb(cover(\{\varphi_1, \varphi_2 ... \varphi_k\}))$ which, by Lemma 11.2.4, is equal to $elb(cover(\{\varphi_1 \wedge \varphi_2 \wedge ... \wedge \varphi_k\}))$. This means that s_1 is a successor of s, or, in other words, $R_\tau(s, s_1)$. $\Box$

Theorem 11.3.4 *[Pruning theorem]*
Let $H \subseteq S' \times S$ be a simulation relation, and let $R' \subseteq S' \times S'$ be a total relation. We define $\hat{S} = S \backslash \{s|$ no infinite path is leaving s $\}$. Then $H|_{S' \times \hat{S}} = H$.

Proof:
Let S_k be the set of states in S for which the longest path leaving them is of length k. Let $\hat{S}_k = S \backslash S_k$. We prove by induction over k that $H|_{S' \times \hat{S}_k} = H$
Base: k=0.
Consider the set of states $\hat{S}_0 = S \backslash \{s|$no edges leaving s$\}$. Let $s' \in S'$. Since R' is total, every s' has some successor s'_1. Let s be a tableau state such that $H(s', s)$. Then by the simulation relation, s has a successor as well. Therefore $s \in \hat{S}_0$. Thus $H|_{S' \times \hat{S}_0} = H$

Induction step: Assume $H|_{S' \times \hat{S}_k} = H$ for any value up to k, and prove $H|_{S' \times \hat{S}_{k+1}} = H$.
Consider a state $t \in S_{k+1}$, and consider an outgoing edge (t, t_1). Clearly $t_1 \in S_k$. Otherwise we would have a path longer than $k+1$. From the induction hypothesis we know that there is no state $t' \in S'$ such that $(t', t_1) \in H$, thus no successor of s in H. By the definition of simulation relation we know that for every state $s' \in S'$ there is a state s such that $H(s', s)$. We also know that there is an edge $(s', s'_1) \in R'$, since R' is total, and so, s must also have a successor s_1, such that $H(s'_1, s_1)$. We therefore conclude that $s \notin S_{k+1}$. Thus $H|_{S' \times \hat{S}_{k+1}} = H$. $\square$

Next, we prove the other tableau property, namely $\tau(\psi) \models_3 \psi$.

Lemma 11.3.5 *[tableau state satisfaction]*
Given a tableau structure $\tau(\psi)$, let φ be an ACTL formula such that $el(\varphi) \subseteq el(\psi)$. For every tableau state $t \in S_\tau$, if there is $s \in cover(\{\varphi\})$ such that $s \subseteq t$, then $\tau(\psi), t \models_3 \varphi$.

Proof:
The proof is by induction on the structure of φ.
Base: $\varphi = g$, $g \in AP_{np}$.
By the definition of *cover*, $cover(\{g\}) = \{\{g\}\}$. Thus $s = \{g\}$. For every tableau state t such that $s \subseteq t$, we know that $g \in t$. By the definition of satisfaction we know that $t \models_3 g$.
Induction step: Assume the correctness for formulas with k operators and prove the correctness for formulas with $k+1$ operators.

1. $\varphi = \varphi_1 \vee \varphi_2$. Consider $s \in cover(\{\varphi_1 \vee \varphi_2\})$. By Lemma 11.2.1, $s \in cover(\{\varphi_1\}) \cup cover(\{\varphi_2\})$. Assume first that $s \in cover(\{\varphi_1\})$. Since φ_1 has at most k operators, the induction hypothesis holds. Thus for every $t \in S_\tau$, $s \subseteq t$ implies $t \models_3 \varphi_1$. Similarly, if $s \in cover(\{\varphi_2\})$, then $t \models_3 \varphi_2$. By the definition of satisfaction, $t \models_3 \varphi_1 \vee \varphi_2$.

2. $\varphi = \varphi_1 \wedge \varphi_2$. Consider $s \in cover(\{\varphi_1 \wedge \varphi_2\})$. By Lemma 11.2.2, $s \in cover(\{\varphi_1\}) X_{con} cover(\{\varphi_2\})$, which means that $s = s_1 \cup s_2$, for $s_1 \in cover(\{\varphi_1\})$ and $s_2 \in cover(\{\varphi_2\})$. Since φ_1, φ_2 have at most k operators, the induction hypothesis holds. Thus, for every $t \in S_\tau$, $s_1 \subseteq t$ results in $t \models_3 \varphi_1$, and $s_2 \subseteq t$ results in $t \models_3 \varphi_2$. For tableau state t such that $s \subseteq t$, we have both $s_1 \subseteq t$ and $s_2 \subseteq t$, and by the definition of satisfaction, $t \models_3 \varphi_1 \wedge \varphi_2$.

3. $\varphi = AX\varphi_1$
   Consider $s \in cover(\{AX\varphi_1\})$. By the definition of *cover* we get $s = \{AX\varphi_1\}$. Consider any tableau state t, such that $s \subseteq t$. Assume that t includes, in addition to $AX\varphi_1$, the formulas $AX\psi_1, \ldots, AX\psi_k$. Let ψ be $\psi_1 \wedge \ldots \wedge \psi_k$. By the tableau construction, the set of successors of t is $Successors(t) = min\_cover(imps(t)) \subseteq cover(imps(t)) = cover(\{\varphi_1, \psi_1, \ldots, \psi_k\})$. By Lemmas 11.2.2 and 11.2.4, $Successors(t) \subseteq cover(\{\varphi_1 \wedge \psi\}) = cover(\{\varphi_1\})X_{con}cover(\{\psi\})$. Thus, for every successor t_1 of t, $t_1 = s_1 \cup s_2$, where $s_1 \in cover(\{\varphi_1\})$ and $s_2 \in cover(\{\psi\})$. By the induction hypothesis, $s_1 \subseteq t_1$ implies $t_1 \models_3 \varphi_1$. Consequently, $t \models_3 AX\varphi_1$.

4. $\varphi = A[\varphi_1 W \varphi_2]$
   Consider $s \in cover(\{A[\varphi_1 W \varphi_2]\})$ and a tableau state t such that $s \subseteq t$. According to Lemma 11.2.3,
   $s \in cover(\{\varphi_2\}) \cup (cover(\{\varphi_1\})\, X_{con}\, cover(\{AXA[\varphi_1 W \varphi_2]\}))$.
   We prove this case by negation. Assume that $t \not\models_3 A[\varphi_1 W \varphi_2]$. Then there exists a path $\pi = s_0 s_1 ...$, starting at t, such that
   $\exists i[s_i \not\models_3 \varphi_1 \wedge \forall j \leq i \,:\, s_j \not\models_3 \varphi_2]$. Let i be the smallest such index of all paths leaving t. Since φ_1, φ_2 have at most k operators, the induction hypothesis holds. Therefore, since $s_i \not\models_3 \varphi_1$, then there is no $s'_i \in cover(\{\varphi_1\})$ such that $s'_i \subseteq s_i$. Similarly, $\forall j \leq i \,:\, s_j \not\models_3 \varphi_2$ and therefore there is no $s'_j \in cover(\{\varphi_2\})$ such that $s'_j \subseteq s_j$.
   From the above we can conclude that $\forall s'_i \subseteq s_i\; s'_i \notin cover(\{A[\varphi_1 W \varphi_2]\})$.

   Given the above i, we show by induction on j that for each state s_j along π, if $j < i$ then the following holds for s_j:
   1. $AXA[\varphi_1 W \varphi_2] \in s_j$.
   2. There exists $s'_j \subseteq s_j$ such that $s'_j \in cover(\{\varphi_1\})$.

   Proof: In the case that $i = 0$ the conditions hold in an empty manner. Assume $i > 0$.
   Base: $j = 0$. Let t be the first state in the path, and recall that $s \subseteq t$. By assumption, $t \not\models_3 A[\varphi_1 W \varphi_2]$, thus $t \not\models_3 \varphi_2$. For any $s' \subseteq t$ $s' \notin cover(\{\varphi_2\})$. We therefore conclude that
   $s \in cover(\{\varphi_1\})\, X_{con}\, cover(\{AXA[\varphi_1 W \varphi_2]\})$, which implies that $AXA[\varphi_1 W \varphi_2] \in t$ and there exists $s' \subseteq s$ such that $s' \in cover(\{\varphi_1\})$. Because $s' \subseteq s \Rightarrow s' \subseteq t$, the second condition holds as well.
   Induction step: Assume the conditions hold for s_j. We show that

they hold for s_{j+1}.
If $j+1 \geq i$ then the conditions hold trivially. Assume $j+1 < i$.
According to the induction hypothesis, $AXA[\varphi_1 W \varphi_2] \in s_j$. Since s_{j+1} is a successor of s_j, then from the tableau construction there is $s'_{j+1} \subseteq s_{j+1}$ such that $s'_{j+1} \in cover(\{A[\varphi_1 W \varphi_2]\})$. Since $j+1 < i$, we have $s_{j+1} \notin cover(\{\varphi_2\})$. Thus $s_{j+1} \in cover(\{\varphi_1\})\, X_{con}\, cover(\{AXA[\varphi_1 W \varphi_2]\})$, which implies $AXA[\varphi_1 W \varphi_2] \in s_{j+1}$ and there exists $s'_{j+1} \subseteq s_{j+1}$ such that $s'_{j+1} \in cover(\{\varphi_1\})$.
This completes the inductive proof.

We have shown that for all $0 \leq j < i$ $AXA[\varphi_1 W \varphi_2] \in s_j$. This is true in particular for s_{i-1}. Therefore, by the construction of the tableau for the successor s_i, there exists $s'_i \subseteq s_i$, such that $s'_i \in cover(\{A[\varphi_1 W \varphi_2]\})$, a contradiction.

□

Theorem 11.3.6 *[Satisfaction Theorem]*
For every ACTL formula ψ, $\tau(\psi) \models_3 \psi$.

Proof:
By the construction of the tableau, every initial state s_0 is in $min\_cover(\{\psi\}) \subseteq cover(\{\psi\})$. According to Lemma 11.3.5 for every tableau state $t \in S_\tau$, if there is $s \in cover(\{\psi\})$ such that $s \subseteq t$, then $\tau(\psi), t \models_3 \psi$. Thus for every initial state s_0, $s_0 \models_3 \psi$, which means that $\tau(\psi) \models_3 \psi$. □

We can now complete the proof of the first tableau property.

Lemma 11.3.7
Given $M' \leq_3 \tau(\psi)$. If $s' \leq_3 s$, then $\forall \varphi \in ACTL\ [\tau(\psi), s \models_3 \varphi \Rightarrow M', s' \models \varphi]$

Proof:
The proof is by induction on the structure of φ. It follows the proof of a similar Lemma for the case of two-value simulation in [14]. The only difference is the base of the induction, which relates to the labeling of states.

Induction base: $\varphi \in AP_{np}$
Consider $\varphi = p$ and $s \models_3 p$. According to the definition of satisfaction 10.3.1,

$p \in L_\mathcal{T}(s)$. According to definition of three-value simulation 10.3.3, $p \in L_{M'}(s')$ and therefore $M', s' \models p$.
Similarly, if $\varphi = \neg p$ and $s \models_3 \neg p$. According to the definition of satisfaction, $\neg p \in L_\mathcal{T}(s)$. According to the definition of three-value simulation, $p \notin L_{M'}(s')$ and therefore $M', s' \models \neg p$.

The induction step is similar to a corresponding Lemma in [14]. □

Theorem 11.3.8 *[First tableau property]*
For every structure M', $M' \models \psi \Leftrightarrow M' \leq_3 \tau(\psi)$.

Proof:
By Theorem 11.3.3 and 11.3.4 we conclude that $M' \models \psi \Rightarrow M' \leq_3 \tau(\psi)$. To see the other direction, note that by Theorem 11.3.6 $\tau(\psi) \models_3 \psi$. Thus, according to Lemma 11.3.7, if $M' \leq_3 \tau(\psi)$ then $M' \models \psi$. □

Chapter 12

Three-Value Comparison Criteria Over the Reduced Tableau

The comparison criteria defined in Chapter 4 assume a two-value labeling tableau. We can apply these criteria to the three-value labeling reduced tableau and still get meaningful evidence. However, each state in the reduced tableau may represent many combinations of atomic propositions. In this case, the two-value comparison criteria refer to propositions that do not appear in a state as "don't care." The criteria will be empty if all but the "don't cares" are implemented.

The completeness obtained when the comparison criteria are empty for the reduced tableau is called *practical completeness.*

It is possible to achieve full completeness in the presence of the three-value reduced tableau. In order to do so we define new comparison criteria over the reduced tableau. These criteria are called *three-value comparison criteria.* They guarantee that for any tableau state in which some propositions are not specified there are corresponding implementation states with all possible values for these propositions.

We start by defining the *Induced labeling* function of the reduced tableau, which is the set of all possible labeling combinations associated with a state.

Given a three-value reduced tableau $\tau =< S_\tau, S_{0\tau}, R_\tau, L_\tau >$ over AP, we first define all the maximal consistent sets over AP_{np}, and then intersect them with the labeling function $L_\tau(s)$.

Definition 12.0.2 ($Consistent\_AP$)
$Consistent\_AP =$
$\{C \subset AP_{np} | \forall p \in AP \ holds \ (p \in C \wedge \neg p \notin C) \vee (p \notin C \wedge \neg p \in C) \ \}.$

Definition 12.0.3 ($L_{ind}(s)$) *[Induced Labeling function]*
$L_{ind}(s) = \{l \in Consistent\_AP | l \cap L_\tau(s) = L_\tau(s)\}.$

We can define now the comparison criteria over a three-value reduced tableau:

1. $UnImplementedState_3 =$
   $\{s_t \in S_t \,|$ either $\forall s_i \in S_i,\ (s_i, s_t) \notin ReachSIM$ or
   $\exists l \in L_{ind}(s_t)\, \forall s_i \in S_i, (s_i, s_t) \in ReachSIM \rightarrow l \neq L_i(s_i)\}.$

   This criterion defines either a reduced tableau state that is not implemented or a tableau state that does not have a corresponding implementation state for each induced labeling combination.

2. $UnImplementedTransition_3 = \{(s_t, s_t') \in R_t \,|\, \exists s_i, s_i' \in S_i,$
   $[\,(s_i, s_t) \in ReachSIM,\ (s_i', s_t') \in ReachSIM$ and $(s_i, s_i') \notin R_i\,]\}$

   This definition is identical for the two-value tableau and for the three-value reduced tableau. It defines a tableau transition that does not have a corresponding implementation transition.

3. $UnImplementedStartState_3 =$
   $\{s_t \in S_{0t} \,|$ either $\forall s_i \in S_{0i}\,,\ (s_i, s_t) \notin ReachSIM$ or
   $\exists l \in L_{ind}(s_t)\, \forall s_i \in S_{0i}, (s_i, s_t) \in ReachSIM \rightarrow l \neq L_i(s_i)\}$

   This criterion is similar to $UnImplementedState_3$, except that it relates only to initial states.

4. $ManyToOne_3 = \{s_t \in S_t \,|\, \exists s_{1i}, s_{2i} \in S_i\, [\,(s_{1i}, s_t) \in ReachSIM, (s_{2i}, s_t) \in ReachSIM$ and $s_{1i} \neq s_{2i} \wedge L_i(s_{1i}) = L_i(s_{2i})\,]\,\}.$

   This criterion defines two implementation states mapped to the same state in the reduced tableau. Since we don't want to get false evidence for any "don't care" in a state, we also require that the labeling of the two states s_{1i}, s_{2i} be identical.

Chapter 13

Reduced Tableau Applications

As mentioned before the reasons for using a reduced tableau are:

1. Smaller structure than traditional tableau.
   This will exemplified be by an example in section 13.2.

2. Removing false evidences.
   When constructing extra tableau states or transitions we may get false indications of unimplemented states or transitions. Extra tableau states may be generated if the power set of all elementary states in the given formula is constracted, for instance, as done in [14]. Another example is the existance of redundant states in the reduced tableau (referred to as *Little Brothers*), where the behavior of the extra state is included in the behavior of another state, both having a common predecessor.

3. Comparing only to meaningful information (practical completeness).
   Following are the steps needed to determine practical completeness.

4. Identifying redundancies in the specification formula.
   This issue will be defined and examplified in section 13.3.

13.1 Practical Completeness Methodology

We now present the algorithm for checking the coverage of a model and its specification, with respect to practical completeness and completeness.

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 the reduced tableau $\tau(\psi)$ for ψ.
3. Compute SIM of $(M, \tau(\psi))$ according to the definition of $\leq_3$.
4. Compute $ReachSIM$ of $(M, \tau(\psi))$ from SIM of $(M, \tau(\psi))$.
5. For each of the two value comparison criteria, evaluate if its corresponding set is empty. If not, present an evidence for its failure.
6. Otherwise, if all two value comparison sets are empty we have **practical completeness**.
7. For each of the three value comparison criteria, evaluate if its corresponding set is empty. If not, present an evidence for its failure.
8. If all three value comparison criteria are empty our implementation is **complete** with respect to the specification.

Figure 13.1: Determining Completeness and Practical Completeness

13.2 Reduced Tableau Example

We have constructed the reduced tableau for the specification formula ψ from the arbiter example of Chapter 5. We received a structure with 20 states. A traditional tableau structure, as in [14] would have a state space of 2^{15} states for ψ. Figure 13.2 depicts the tableau received for ψ.

The semantics of the tableau drawing is taken from the StateChart specification semantics. The large round cornered boxes are meta-states. Each meta-state includes several states. An edge going to a meta-state should be interpreted as an edge to each state in the meta-state. Similary, labeling of a meta-state (the state values not in an internal box) should be applied to each state in the meta-state.

13.3 Identifying Redundancies in the Specification

Chapter 4 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., sub formulas that can be removed or be rewritten without destroying the completeness of the specification. This additional checking is a method for detecting sub formulas which are vacous.

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:

Definition 13.3.1 ($OneToMany$)
$OneToMany = \{s_i \in S_i \mid \exists s_{1t}, s_{2t} \in S_t[(s_i, s_{1t}) \in ReachSIM \wedge$
$(s_i, s_{2t}) \in ReachSIM \wedge s_{1t} \neq s_{2t}]\}$.

13.4 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 :

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

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 agree with φ_6 for states labeled with $\{req0, req1\}$, and 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.

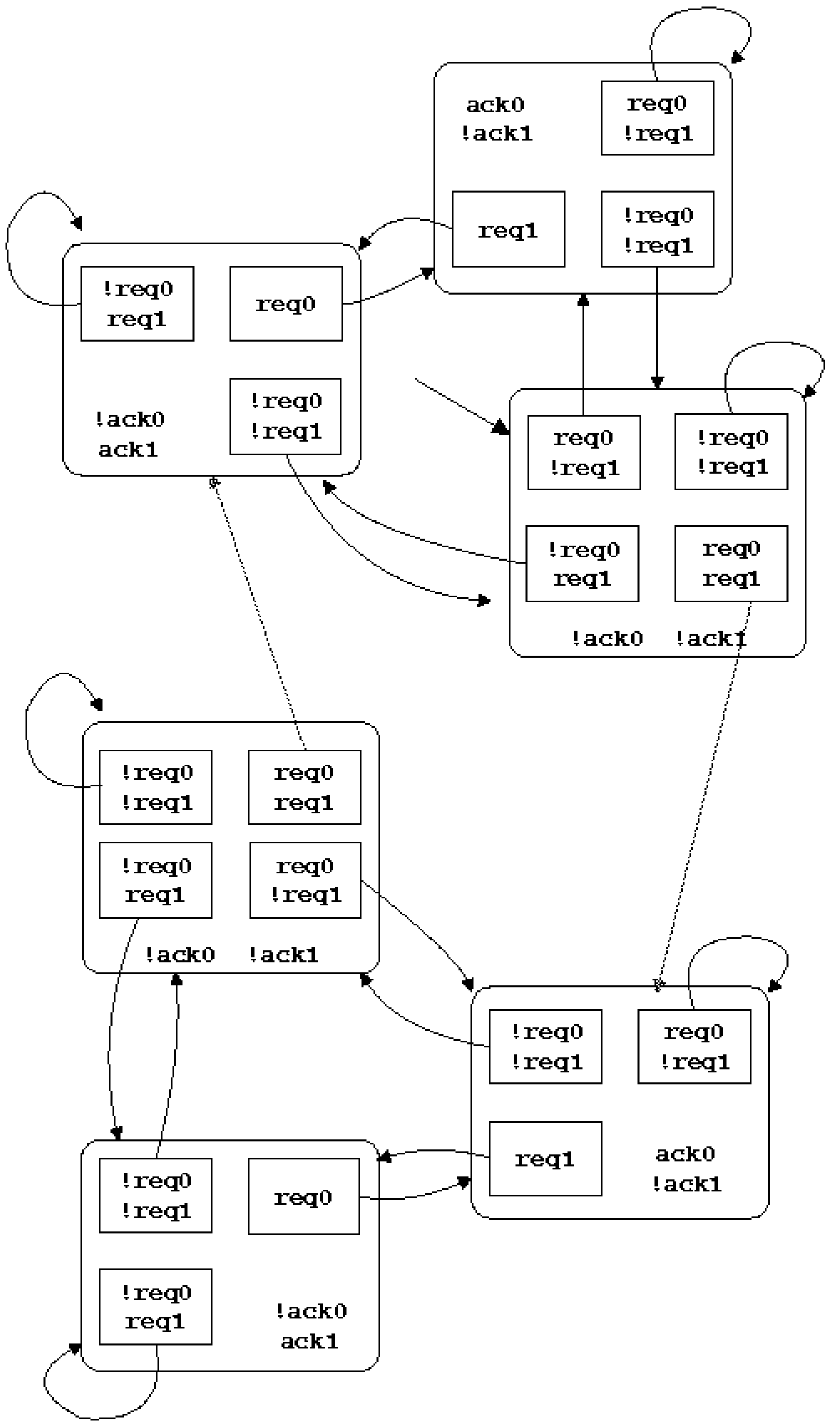

Figure 13.2: Reduced Tableau of Full Specification

Chapter 14

Concluding Remarks and Future work

14.1 Extensions to Our Work

The major part of this work has been presented in [20]. In a related work, [21], there is a simple approach for checking completeness, using a model checker. The use of a model checker replaces the computation of the *ReachSIM* preorder. The model checker compares the given model with a tableau model such that when both models receive the same inputs, they should produce the same outputs. This approach, however, is useful only when the tableau is deterministic. The ACTL language can be restricted to make the tableau deterministic by limiting the disjunctions in the specification language. Examples are the language RCTL in [1] and the ACTL subset used in [18]. Chockler et al. [5] characterizes the requirements for a deterministic tableau.

The framework reported here, including the reduced tableau construction, the symbolic algorithms and the criteria analysis, has been implemented and coded as an extension to the symbolic model checker SMV [27]. The details of the additional code may be found as a separate report [13] describing the code outline and the data structure used. Grouchnikov [13] detected various equivalences of propositional formulas, combining states that are syntactically different but semantically identical, into a single state. Grouchnikov's prototype [13] has been used to test the various aspects of this coverage work, including the arbiter example from section 5.

14.2 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 still 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 $\tau(\psi)$ are both in memory. This implies that the circuits to which we can apply the method at present are smaller than those that we can model check. Therefore, we cannot provide a solution for large models at this time. However, we believe that optimizations in this area will eventually be introduced, as they were for model checking. We are investigating the possibility of running our 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 [18] as a preliminary step, and try to expand it to fully support our methodology. The definition of an Unimplemented State is closely related to the notion of evidence in [18]. On the other hand, our Unimplemented Transition criterion provides path evidences, while path coverage is not addressed by the methodology of [18]. Furthermore, our method can indicate complete agreement between the specification and the implementation. This may indicate that the verification process can be stopped.

2. **Irrelevant information:** As in the area of traditional simulation coverage, measurement of quality produces a lot of irrelevant information. A major problem is that specifications tend to be incomplete by nature. Therefore, we do not necessarily want to obtain a bisimulation relation between the specification and the implementation. Adopting practical completeness together with abstraction gives a partial solution to this problem. There is a need for methodology on how to write specification properties that effectively utilize the notion of

"don't care" and the abstraction feature. In general, it will eventually be necessary to devise techniques to filter the results so that only the interesting evidence is reported.

In addition, we are also investigating whether the reduced tableau described in Section 9 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 straightforward to extend our method to full ACTL. This will require, however, the addition of fairness constraints to the tableau structure and to use the *fair simulation preorder* [14]. Unfortunately, there is no efficient algorithm to implement fair simulation [16]. Thus, it is currently impractical to use full ACTL. It is necessary to find logics that are both reasonable in terms of their expressivity and practical in terms of tableau construction and comparison criteria.

Bibliography

[1] I. Beer, S. Ben David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. In *Proceedings of the Conference on Computer Aided Verification (CAV 97)*, volume 1254, pages 279–290, 1997.

[2] J.P. Bergmann and M.A. Horowitz. Improving coverage analysis and test generation for large designs. In *Proceedings of the Conference on Computer Aided Verification (CAV 99)*, volume 1742 of *Lecture Notes in Computer Science*, pages 580–584, November 1999.

[3] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24:298–318, September 1992.

[4] K.T. Cheng and A. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proc. of Design Automation Conference*, pages 86–91, 1993.

[5] H. Chockler, O. Kupferman, and M.Y. Vardi. Coverage Metrics for Temporal Logic Model Checking. In *TACAS proceedings*, Lecture Notes in Computer Science. Springer-Verlag, 2001.

[6] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. L. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ Cache Coherence Protocol. In *Proceedings of the 11th International Conference on Computer Hardware Description Languages*, pages 15–30, 1993.

[7] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop of Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, 1981.

[8] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT press, 1999.

[9] R. Cleaveland. Tableau-Based Model Checking in the Propositional Mu-Calculus. *Acta Informatica*, 27:725–747, 1990.

[10] R. Cleaveland, J. Parrow, and B. Steffen. A semantics-based verification tool for finite-state systems. In Brinksma, Scollo, and Vissers, editors, *Proc. 9^{th} IFIP WG 6.1 International Symposium on Protocol Specification, Testing, and*

Verification, pages 287–302, Univ. of Twente, The Netherlands, 1989. North-Holland.

[11] M. Daniele, F. Giunchiglia, and M. Vardi. Improved Automata Generation for Linear Time Temporal Logic. In *Proceedings of the Conference on Computer Aided Verification (CAV 99)*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[12] S. Devadas, A. Ghosh, and K. Keutzer. An Observability-based code coverage metric for functional simulation. In *Proceedings of the International Conference on Computer-Aided Design*, pages 418–425, November 1996.

[13] K. Grouchnikov. Building tableau in SMV environment. *Project Report, Computer Science Technion, Israel*, March 1999.

[14] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. on Programming Languages and Systems*, 16(3):843–871, 1994.

[15] M.R. Henzinger, T.A. Henzinger, and P.W. Kopke. Computing simulation on finite and infinite graphs. In *Proc. Symp. Foundations of Computer Science*, pages 453–462, 1995.

[16] 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.

[17] R.C. Ho and M.A. Horowitz. Validation coverage analysis for complex digital designs. In *Proceedings of the International Conference on Computer-Aided Design*, pages 146–151, November 1996.

[18] 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.

[19] Y. Hoskote, D. Moundanos, and J. Abraham. Automatic extraction of the control flow machine and application to evaluating coverage of verification vectors. In *Proceedings of ICDD*, pages 532–537, October 1995.

[20] S. Katz, D. Geist, and O. Grumberg. Have I written enough properties? A method of comparison between specification and implementation. In *10th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science.* Springer-Verlag, 1999.

[21] S. Katz, D. Geist, and O. Grumberg. Identification of missing properties in model checking. *US patent application IS899-0018*, 2000.

[22] O. Kupferman and M.Y. Vardi. Vacuity detection in temporal model checking. In *10th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science.* Springer-Verlag, 1999.

[23] R.P. Kurshan. FormalCheck User's Manual. *Cadence Design Inc.*, 1998.

[24] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specifications. In *Preceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, January 1985.

[25] LOTOS - A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. In *ISO. Information Processing Systems - Open Systems Interconnection - ISO/IEC 8807, International Organisation for Standardization*, Geneva,Switzerland, 1989.

[26] Z. Manna and A. Pnueli. *Temporal verifications of Reactive Systems - Safety.* Springer-Verlag, 1995.

[27] K. L. McMillan. *The SMV System DRAFT.* Carnegie Mellon University, Pittsburgh, PA, 1992.

[28] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Press, Norwell, MA, 1993.

[29] R. Milner. An Algebraic Definition of Simulation between Programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pages 481–489, September 1971.

[30] D. Park. Concurrency and Automata on Infinite Sequences. In *5th GI-Conference on Theoretical Computer Science*, pages 167–183. Springer-Verlag, 1981. LNCS 104.

[31] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th International Symp. on Programming*, volume 137, 1981.

[32] M. Horowitz R. Ho, C. Yang and D. Dill. Architecture validation for processors. In *Proceeedings of the 22nd Annual Symposium on Computer Architecture*, pages 404–413, June 1995.

[33] F. Somenzi and R. Bloem. Efficient Buchi Automata from LTL Formulae. In *Proceedings of the Conference on Computer Aided Verification (CAV 2000)*, volume 1855 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[34] 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.

Appendix A

The *compose* C code

```
#include "bdd.h"
extern bdd_ptr current_vars;
bdd_ptr compose(a,b)
bdd_ptr a,b;
{
int alevel,blevel;
    /* Check the trivial cases (the base of the recursion) : */
if (a==ONE) return forsome(current_vars,b);
    /* if a is the OBDD ONE, we have to return exist x variables in OBDD b
    This is done by forsome with an OBDD that has all the even variables */
if (b==ONE) return f_shift(forsome(current_vars,a));
    /* if b is the OBDD ONE, we have to return exist out the x
    variables in OBDD a This is done by forsome with an OBDD that has
    all the even variables
    After exist out we have to f_shift the y variables to have even
    variables */
if ((a==ZERO) | (b==ZERO)) return ZERO;
    /* If any one of the "and" operation is 0 the result is 0 */
    /* Compose it for two non trivial OBDD's : */
    /* Get the levels of the OBDDs */
alevel = GETLEVEL(a) ;
blevel = GETLEVEL(b) ;
    /* Compare the levels of the two OBDD's */
if (alevel==blevel) {
    if (IS_CURRENT_VAR(alevel)) {
        /* x from a(x,y) and from b(x,u)
        If a level is equal to b level we have the same variable
        If a is CURRENT, we have to exist it out of a and b
        we can compose them directly as belonging to the same level
        recursively
        or_bdd() is used as replacement of forsome()
        */
        return or_bdd(compose(a->left,b->left),
            compose(a->right,b->right));
    }
    else {
        /* y from a(x,y) and u from b(x,u) */
        /* If a level is equal to b level but the variable is of kind NEXT,
        we don't have the same variable. The variable on a is from group y,
        the variable on b is from u
        We have to compose them recursively by splitting a (which has to sit
        in the even place in the result) , but without existing it out
        we should find if we have this OBDD already, if not, compose it. */
        return find_bdd(NEXT_TO_CURRENT(alevel),
            compose(a->left,b),
            compose(a->right,b));
        /* The result OBDD has to be a current variable. (u variable is NEXT
        in the result OBDD)
        and the a portion of the result OBDD is a CURRENT (y variable).
        We do it by changing the level from blevel to NEXT_TO_CURRENT(alevel)
        and going "down" on a only, resulting in an OBDD of level CURRENT,
        and the recursion will fix everything magically. */
    } /* a=b, NEXT */
} /* alevel == blevel */
if (alevel>blevel) {
    /* alevel value is higher, split on values of b variable */
    if (IS_CURRENT_VAR(blevel)) {
        /* We have x variable (even) from b(x,u),
        exist it out and go down one level */
        return or_bdd(compose(a,b->left),compose(a,b->right));
    }
    else {
        /* We have u variable (odd) from b(x,u), no need to exist it out.
        Calculate by going down one level and returning OBDD of odd level */
        return find_bdd(blevel,compose(a,b->left),compose(a,b->right));
        }  /* a > b, NEXT(b) */
    }      /* a > b */
else { /* a < b */
    /* blevel value is higher, split on values of a variable */
    if (IS_CURRENT_VAR(alevel)) {
        /* We have x variable (even) from a(x,y) ,
        exist it out and go down one level */
        return or_bdd(compose(a->left,b),compose(a->right,b));
    }
    else { /* NEXT_VAR */
        /* We have y variable (odd) from a(x,y), no need to exist it out.
        Calculate by going down one level and returning OBDD of even level */
        return find_bdd(NEXT_TO_CURRENT(alevel),compose(a->left,b),
            compose(a->right,b));
    }  /* a<b, NEXT(a) */
}  /* a<b, */
}
```

Appendix B

The *compose_odd* C code

```
#include "bdd.h"
extern bdd_ptr next_vars;
bdd_ptr compose_odd(a,b)
bdd_ptr a,b;
{
int alevel,blevel;
if (a==ONE) return r_shift(forsome(next_vars,b));
if (b==ONE) return forsome(next_vars,a);
/* If any one of the "and" operation is 0 the result is 0 */
if ((a==ZERO) | (b==ZERO)) return ZERO;
/* Compose it for two non trivial BDD's : */
/* Get the levels of the bdds */
alevel = GETLEVEL(a) ;
blevel = GETLEVEL(b) ;
/* Compare the levels of the two bdd's */
if (alevel==blevel) {
    if (IS_NEXT_VAR(alevel)) {
    return or_bdd(compose_odd(a->left,b->left),
        compose_odd(a->right,b->right));
}
else {
    return find_bdd(alevel,
        compose_odd(a->left,b),
        compose_odd(a->right,b));
    } /* CURRENT */
} /* alevel == blevel */
if (alevel>blevel) {
    if (IS_NEXT_VAR(blevel)) {
    return or_bdd(compose_odd(a,b->left),compose_odd(a,b->right));
    }
    else {
    return find_bdd(CURRENT_TO_NEXT(blevel),
        compose_odd(a,b->left),
        compose_odd(a,b->right));
    }
}
else {
    if (IS_NEXT_VAR(alevel)) {
        return or_bdd(compose_odd(a->left,b),
        compose_odd(a->right,b));
    }
    else { /* CURRENT */
        return find_bdd(alevel,
            compose_odd(a->left,b),
            compose_odd(a->right,b));
        }
    }
}
```