INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Improving Compositional Verification through Environment Synthesis and Syntactic Model Reduction

Hong Peng

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy at

Concordia University

Montréal, Québec, Canada

September 2002

ABSTRACT

Improving Compositional Verification through Environment Synthesis and Syntactic Model Reduction

Hong Peng, Ph. D.
Concordia University, 2002

With the increasing complexity of large scale Application-Specific Integrated Circuit (ASIC) designs, simulation cannot cover all the corner cases in a reasonable time frame. Formal verification is emerging as a supplementary approach to mainstream simulation. Among various formal verification approaches, model checking is a fully automatic approach to verify a finite state machine against its temporal specifications. It has been successfully used in several industrial verification projects. However, its application is limited by the size of the system to be verified due to state space explosion. There are two main methods to tackle this problem: *compositional verification* and *model reduction.* Compositional verification is to verify each partition in the system separately and then derive the system specification from the partial proofs. Model reduction is to reduce the size of the system such that it can be handled by a model checking tool. In this thesis, we integrate these two approaches to perform model checking.

In a compositional verification, properties are only true under certain environments. One of the problems in the compositional reasoning approach is to generate the *environment assumption*, i.e., stimulus for the module (partition) under verification. In our approach, we provide the environment assumptions as temporal logic formulas and then synthesize an executable hardware description language module

from it. Compared with existing related work, the proposed environment has a smaller state space, which is a key factor in compositional verification. The synthesized modules are composed with the design block under verification and then fed into a model checking tool.

However, in case the size of the composed module is still beyond the capability of model checking, we use a novel syntactic model reduction algorithm, which analyzes the source code and removes the redundant variables and values. The reduction we propose in this thesis is based on the static analysis of control flow diagram of the program. The values of state variables in the program are partitioned into *active values*, and *deactive values* according to their dependency to the properties. The deactive values then can be replaced by an *abstract* value, and thus the value domains of the variables are much smaller than the original ones. Compared with existing related work, the proposed approach is automatic and has better performance in the reduction of datapath intensive designs.

In order to demonstrate the application of the proposed techniques, we verified an industrial size ATM (Asynchronous Transfer Mode) switch fabric from Nortel Networks which complexity is beyond the capability of plain model checking tools.

To Huamin

ACKNOWLEDGEMENTS

I have been very fortunate to have Dr. Sofiène Tahar as my supervisor. I am deeply grateful for his strong support and encouragement through out my Ph.D studies. His expertise and competent advice have shaped the character of my research.

It has been a great opportunity for me to work with Dr. Yassine Mokhtari. I am very indebted to him for his considerable time devoted to me through my research work. I benefited so much from his deep knowledge and insightful thoughts. Without his invaluable guidance and help, I could not have completed this work.

I also wish to express my gratitude to the examination committee members, Dr. Al-Khalili, Dr. Khendek, Dr. Negulescu, and Dr. Seffah for reviewing my thesis and giving me invaluable feedbacks.

I would like to reserve my deepest thanks to my parents for their perpetual love and encouragement, and to my wife for her sacrifice and patience. I can never thank them enough.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

ASIC	Application-Specific Integrated Circuit
ATM	Asynchronous Transfer Mode
CFD	Control Flow Diagram
CNF	Conjunctive Normal Form
COI	Cone Of Influence
CTL	Computational Tree Logic
CTP	Conformance Test Plan
DDD	Data Dependency Diagram
DUT	Device Under Test
FSM	Finite State Machine
HDL	Hardware Description Language
HOL	Higher-Order Logic
IP	Intellectual Property
LHS	Left Hand Side
LTL	Linear Temporal Logic
PVS	Prototype Verification System
RHS	Right Hand Side
ROBDD	Reduced Ordered Binary Decision Diagram
RTL	Register Transfer Lever
SAT	SATisfiability
SMV	Symbolic Model Verifier
SoC	System-on-a-Chip
VIS	Verification Interacting with Synthesis
VLSI	Very Large Scale Integration

Chapter 1

Introduction

1.1 Design Verification

Conventionally, a Very Large Scale Integration (VLSI) design flow goes through several steps from an informal specification to the final physical layout of the chip as illustrated in Figure 1.1 [9]. On the system level, general models are used to do performance evaluation and examine the architectural trade-offs, for example, the decision on which main functions of the chip have to be done in hardware and which can be implemented in software. There are different ways to come to the final physical layout of the chip, and to verify that each of these steps is done correctly against the system model. Usually, the Register Transfer Lever (RTL) implementation is done manually from the system level using Hardware Description Languages (HDLs) such as Verilog or VHDL. Moreover, only a synthesizable subset of these languages can be used in the RTL designs. Then the RTL implementation written in the synthesizable code can be synthesized into gate level.

The verification of a VLSI design is to verify that the implementation of a circuit regardless if it is RTL design or gate-level design, satisfies its high level specifications. This is the so called "functional verification" [9] and simulation is the most often used method to perform the functional verification.

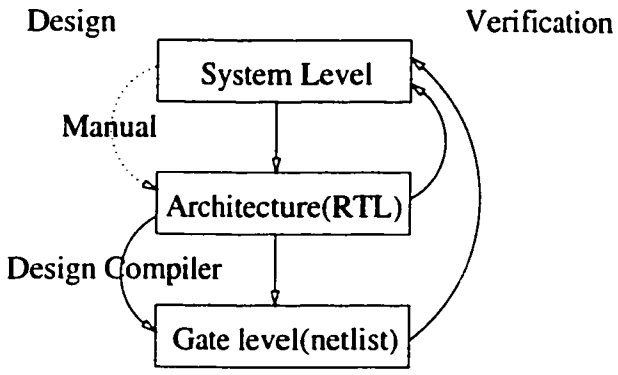


Figure 1.1: General VLSI Design Flow

To verify the proper functions of the RTL design in simulation, test-benches have to be written. They generate test patterns (stimulus) for the Device Under Test (DUT), and compare the output of the DUT (response) with the expected response. The verification challenge is to determine the input patterns to supply to the design, and the expected outputs of a properly working design. In a common verification framework, as shown in Figure 1.2, a design group starts to implement

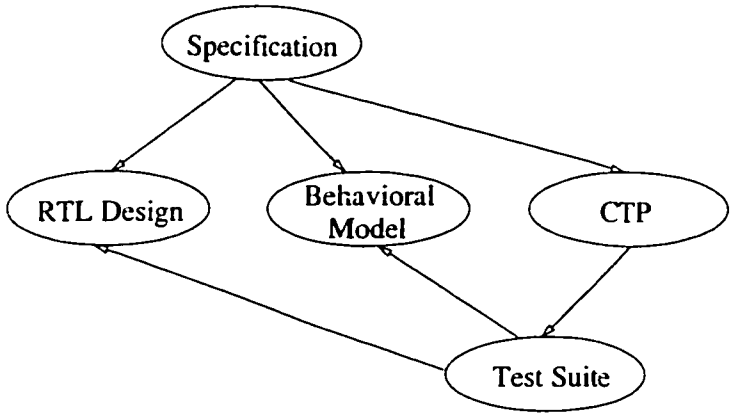


Figure 1.2: General Verification Framework

the RTL design based on the specification, while verification groups start to develop a behavioral model and a test suite. The latter has to cover all test cases given in the Conformance Test Plan (CTP), if possible. The behavioral model is written in a high level language (behavioral level), which can be developed much quicker than the RTL model. The test suite generates test vectors for both models, and their

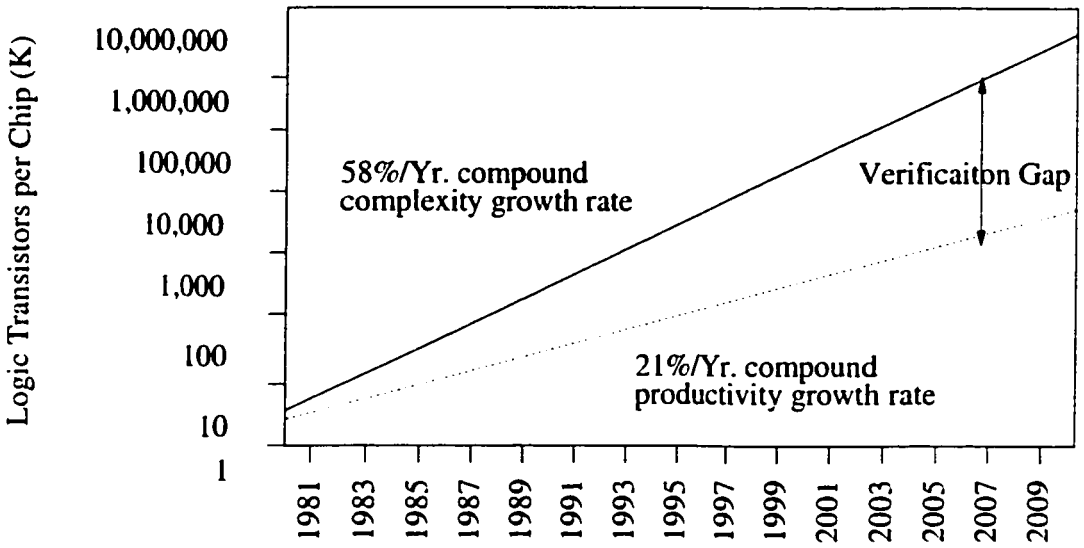
outputs can be compared. The test-bench itself can be tested using the behavioral model. Usually, the test suite as well as the behavioral model is developed using standard HDLs, i.e., Verilog or VHDL.

However, because of the increasing complexity of Application-Specific Integrated Circuits (ASICs) designs, in the above methodology, verification consumes about 70% of the design effort. The number of verification engineers is usually twice the number of RTL designers. When design projects are completed, the code that implements the test-benches makes up to 80% of the total code volume [9]. Table 1.1 shows that in order to cover a reasonable test suite, a large amount of computing resource is needed.

Intel	"Billions of generated vectors take 27 days to run"
Sun	"A test suite with 1500 tests needs 1 billion random simulation cycles"
Cyrix	"170 CPUs running simulations continuously"
Kodak	"Hundreds of 3-4 hours RTL functional simulations"
Xerox	"Simulation runtime occupies 3+ weeks of a design cycle"
Ross	"125 million vector regression tests"
Source: International Technology Roadmap for Semiconductors (1999)	

Table 1.1: Simulation in Industry

Given the amount of effort demanded by verification and the quality and quantity of the code that must be produced, it is not surprising that in most of the projects, verification rests squarely on the critical path of the project management. The more testcases that need to be covered, the more time is going to be invested. Nevertheless, the "verification gap" between the design productivity and the design complexity is becoming larger and larger (Figure 1.3). In fact, this is also the reason why verification is still the target of new tools and methodologies.



Source: Semiconductor Industry Association. International Technology Roadmap for Semiconductors: 1999 edition. Austin, Texas: Interantional SEMATECH, 1999

Figure 1.3: Verification Gap

1.2 Formal Hardware Verification

For the last two decades, formal verification approaches, as supplementary to the simulation, have been proposed in the literature [38, 52, 49]. Formal verification is the process of determining whether the designs of a given phase in the life cycle fulfill a set of established requirements, using formal specification methods.

Formal specification methods include model oriented and algebraic methods. Model oriented methods model the system using mathematical entities (e.g. sets), like Z [26], Finite State Machine [22], Temporal Logic [22], Petri Nets [76], etc. Algebraic methods specify an object class in terms of relationships between the operations defined on that class, like process algebra [10]. Based on the system to be specified, formal specification methods can also be classified into two categories, sequential and concurrent methods. Some formal methods such as Z specify the behavior of sequential system, using mathematical structures like sets, relations, and functions to describe the states. The state transitions are given in terms of pre- and post-conditions. Other methods like Temporal Logic specify the behavior

of concurrent systems. The behavior is defined in terms of sequences, trees, or partial orders of events. Based on the above formal specifications, there are two mainstream approaches, namely theorem proving and Finite State Machine (FSM) based verification. From the latter group, model checking techniques, in particular temporal **CTL** and **LTL** model checking[1], have established themselves as significant means for design verification because of the breakthrough to verify large designs automatically [22].

1.2.1 Theorem Proving

Theorem proving is a powerful verification technique. It is argued that an interactive and general purpose theorem prover could well meet industrial requirements and be the unifying framework for various verification tasks of industrial designs [24, 52].

With theorem proving, an implementation and its specification are usually expressed as higher-order or first-order logic formulas. Their relationship is regarded as a theorem to be proven with the logic system using axioms and inference rules. Designs can thus be represented at different abstraction levels rather than only at the Boolean level. Therefore, it allows a hierarchical verification methodology, which can effectively deal with the overall functionality of designs having complex datapaths. However, theorem proving is basically an interactive process, i.e., users are responsible for coming up with the proof of correctness. The requirement for expertise is the major difficulty for applying theorem proving on industrial designs.

A theorem prover or a proof checker is a tool used to partially automate the verification procedure or to check the manual proof process. There exist many theorem proving systems, some of which are widely used in the hardware verification community. Notable examples are HOL (Higher-Order Logic) [35], developed at Cambridge University; and PVS (Prototype Verification System) [80], developed at

[1]The temporal logics **CTL** (Computation Tree Logic) and **LTL** (Linear Time Temporal Logic) are explained in 2

SRI International Computer Science Laboratory.

1.2.2 FSM based Verification

Synchronous sequential designs can be modeled as Finite state Machines (FSMs). An FSM is a state transition system including initial states and the transitions between states. The basic method of FSM based verification is the exploration of reachable states of the FSM, called reachability analysis. Starting with the initial state as the present state, we compute the states reached in one transition. This process is repeatedly applied until all the reachable states of the FSM have been visited. There are two branches in this category: equivalence checking and model checking.

Equivalence checking verifies that an implementation has the same outputs as the specification has for all input sequences. A classical method for equivalence checking is to model the implementation and the specification as FSMs, and to form their *product machine* by placing the implementation and the specification side by side and feeding them the same inputs, and then to check if the corresponding outputs carry the same value in all reachable states.

In *model checking* [31], the implementation is represented as an FSM and the specification as a set of properties expressed in temporal logic which can concisely capture temporal relationships between states. Then, the validity of the properties is checked by exploring the reachable state space of the implementation FSM. The equivalence checking of two FSMs can be thought of as a special case of model checking where the implementation to be checked is actually the product machine, and the specification is an invariant that states the equality between the corresponding observable signals.

Automation is the major advantage of the FSM-based verification. In general, once an FSM is constructed. The verification algorithm can check its properties by exploring the state space automatically. However, FSM-based methods have a

serious limitation, known as *state space problem* [21]. With the increasing number of state components in a circuit, the state space grows exponentially, which is often the case when the circuit performs substantial data processing. This greatly restricts the application of the approach. One successful technique to reduce the state space explosion problem is known as the *implicit state enumeration* [61]. With this technique, the transition and the output functions as well as the sets of states generated during the reachability analysis are encoded by Reduced Ordered Binary Decision Diagrams (ROBDDs) [12] which is a compact and canonical representation of Boolean functions. The use of ROBDDs significantly enlarged the application of the FSM-based verification [13].

For the moment, the maximum state space size which can be handled in model checking is about two to power several hundreds, namely a circuit less than one thousand latches [22]. How to tackle this problem is still an active research field in recent years.

Two well-known ROBDD-based verification tools are: SMV (Symbolic Model Verifier) [61], a **CTL** model checker developed at Carnegie-Mellon University; and VIS (Verification Interacting with Synthesis) [11], an integrated tool for **CTL** model checking, simulation and synthesis of FSMs, developed at University of California at Berkeley and University of Colorado at Boulder.

1.2.3 Dealing with State Space Explosion

Generally, there are two main methods to tackle the state space explosion problem: *compositional verification* and *model reduction*. With compositional verification we can verify each module in the system separately so as to avoid making the product of the modules in the system, and then compose the local results into a global property. A problem in this approach is that properties are only true under certain environments. In order to make the compositional verification, we have to construct the environment assumptions for the module under verification and guarantee that

the environment covers all possible stimuli. This is the so called *environment problem* [63] in the compositional verification approach.

Furthermore, in today's multi-million-gate ASIC designs, the size of one single module usually is beyond the capability of a plain model checking tool. In this case, the only way to perform model checking is to reduce the size of the module while preserving the correctness of the properties. This is the so called *model reduction* approach. The problem is that existing model reduction approaches need human interaction to guide the reduction. For larger systems, these human-involved reduction methods are infeasible.

In this thesis, we propose two techniques to improve compositional model checking, namely, *environment synthesis* and *syntactic model reduction.*

Using the environment synthesis, in the compositional verification, one can construct a tableau[2] from a set of temporal properties and then synthesize the Verilog HDL environment module from the tableau. Namely in the compositional verification framework, one can specify the environment of the module under verification using temporal properties, then synthesize the formulas into executable Verilog HDL modules. Thus, the synthesized program is guaranteed to cover all the corner cases of the environment. A technique called *reduced tabular construction* is used in the synthesis to make this guaranty sound.

Using the syntactic model reduction, one can reduce the size of the module under verification (RTL implementation) automatically by traversing the syntactic structure of the module while preserving the correctness of the specifications. The whole approach is based on the value dependency analysis of the variables in the module against the variables and values in the properties. Through out the analysis, the redundant variables and values in the module with respect to the specification are reduced. Thus, only the minimum core of the module with respect to the specification is left.

[2]A tableau is a state transition structure.

We provide formal proofs for the soundness of the proposed environment synthesis and syntactic model reduction approaches. The environment synthesis has been implemented in Java. The implementation of syntactic model reduction in C++ is in progress in the Hardware Verification Group of Concordia University.

A comprehensive use of the proposed verification technique is illustrated in Figure 1.4 including following practical steps.

1. Given an RTL design and global properties derived from the specification, if the size of the RTL design, after the reduction, can be handled by the model checking tool, then we will apply model checking directly, else we will do the following compositional verification steps.

2. Partition the RTL design into blocks.

3. Obtain local properties with respect to each RTL block.

4. Derive the environment assumptions (**ACTL** formulas) with respect to each RTL block, then synthesize the formulas into Verilog environment modules, and compose the RTL block and the Verilog environment module.

5. In case the size of the composed module is beyond the capability of the model checking tool, apply model reduction with respect to the local properties, and get the reduced composed module.

6. Verify the reduced composed module against the corresponding local properties using model checking, respectively.

7. Deduce the global properties from these local properties using compositional reasoning rules.

For the verification framework proposed in Figure 1.4, we have chosen the model checker VIS (Verification Interacting with Synthesis) [11] as our model checking tool. This is motivated by the existence of a Verilog front-end such that we can

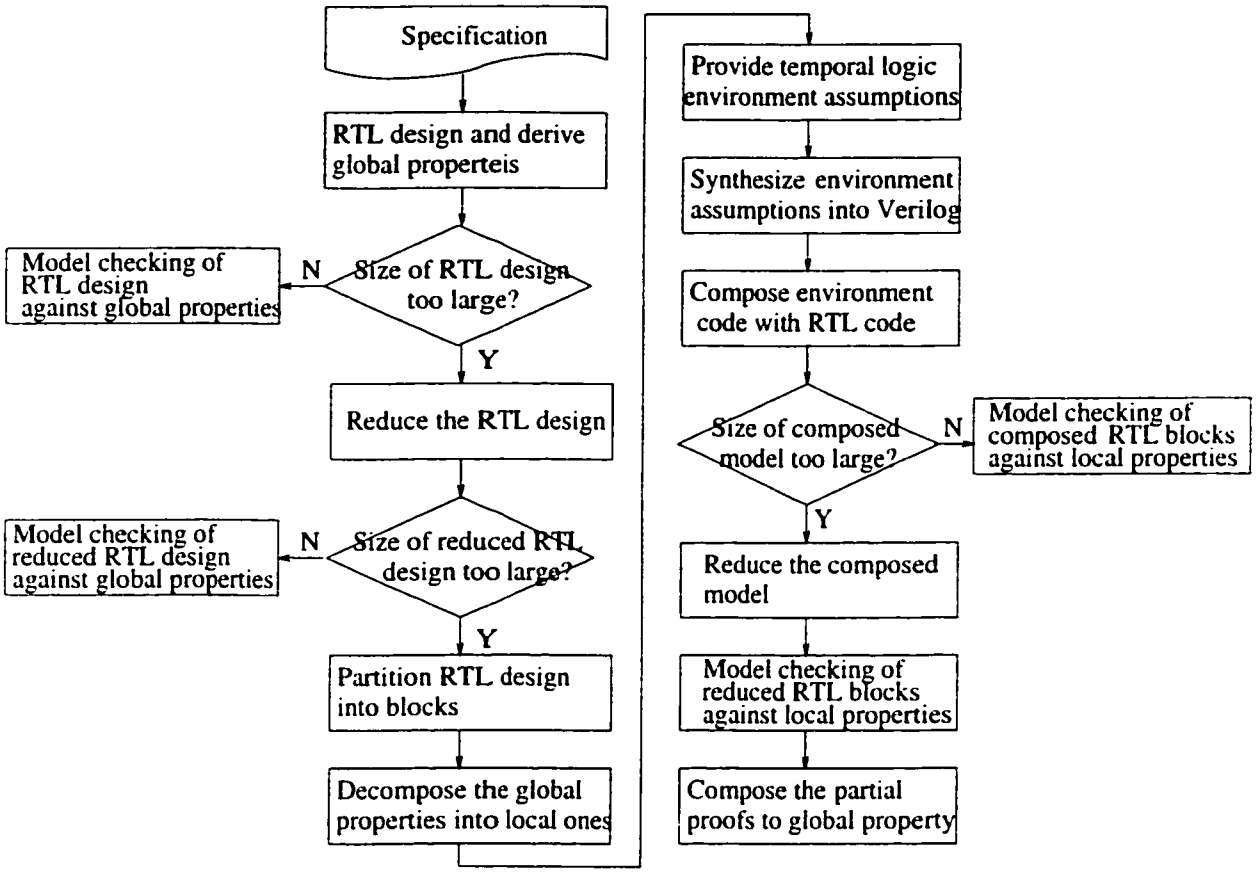


Figure 1.4: Proposed Formal Verification Framework

feed our design into the tool directly. Throughout the compositional verification, the global properties are correct if and only if all the local properties are correct. For now, in terms of verification, partitioning the RTL design, deriving environment assumption formulas and local properties have to be done manually. After we have the local properties and the corresponding environment assumptions, the following verification steps, i.e., the environment synthesis, the model reduction, and model checking are executed *automatically*. Another advantage of this framework is that the compositional reasoning allows us to do block level verification in SoC/IP designs since we can synthesize an executable module from the IP specifications and replace the IP blocks by the synthesized modules.

We have applied the above verification framework on an ATM (Asynchronous

Transfer Mode) switch fabric from Nortel Networks [83] as a case study. Throughout the verification, we discovered some errors in the design. In comparison, we did not succeed in verifying the same switch fabric in FormalCheck [15] because of state space explosion.

1.3 Related Work

1.3.1 Tableau Construction and Environment Synthesis

In [57, 36], D. Long *et al.* proposed a tableau construction to cover all the possible models of an **ACTL**[3] temporal formula. In their approach, states of tableau consist of information about the labelling for the visible state components, plus information about what should be held in successor states. The states of the tableau have the form (f, E), where f is a labelling function that maps each element in the state components to all the possible values, and E is in the power-set of the elementary formulas. The number of states in the tableau are $2^A \times 2^{el}$, where A is the number of visible state components and el is the number of elementary formulas. They proved that this tableau is a maximal model of the formula, which covers all the behaviors of the formula.

When striving for completeness, the size of tableau structures as defined in [57, 36] is usually too large to be practical, and may be much larger than the state space of the given implementation. This is because the state space of such a tableau contains all combinations of subformulas of the specification formula. Such tableau usually contains many redundant states, which can be removed while preserving the tableau properties. If not removed, these states may introduce evidences, which are not of interest.

[3] **ACTL** is a subset of the branching time temporal logic **CTL** restricted to universal path quantification. Fore details, you may refer to Chapter 2.

Our tableau construction work is an extension of [46, 47] by Katz and Grumberg. In their work, they also proposed an idea of reduced tableau construction for safety **ACTL** properties based on three-valued labelling for atomic propositions. During the construction, each state contains exactly the set of formulas required for satisfying the specification formula and the tableau is derived from the particle tableau for **LTL** by replacing the use of **X** temporal operator by **AX**. Since the only difference between **LTL** and **ACTL** is that the universal path quantifier always precedes temporal operators, so this change is sufficient. Since the reduced tableau is based on the three value labelling, the definition of satisfaction and simulation preorder are changed accordingly. The reduced tableau for **ACTL** then has the same properties as the one in [57, 36]. Generally, using the three value labelling, they can derive a smaller tableau compared to the one in [57, 36]. However, they cannot support the construction of liveness **ACTL** properties and they assume the input **ACTL** formulas are concise which is not the case in the practice because engineers may write **ACTL** formulas with much redundancy in it.

In [17, 31], E. M. Clarke *et al.* proposed a method of constructing concurrent programs in which the synchronization skeleton of the program is automatically synthesized from a high level branching time temporal logic specification. Decision procedures were devised which given a formula of temporal logic f, will decide whether f is satisfiable or unsatisfiable. If f is satisfiable, a finite model of f is constructed. A tableau-based decision procedure for satisfiability of **CTL** [17] formulas is used in this approach which has the same flavor as ours. The decision procedure begins by building a tableau $\mathcal{T}$ which is a finite directed AND/OR node network. Each node of the network is either an AND node or an OR node and is labelled by a set of formulas. The AND node successors can be regarded as embodying all of different ways in which the formula in the label can be satisfied. The OR node successors are exactly the successors required to satisfy all the next time formulas in the label. The intended meaning is that a node is considered as

a state in an appropriate structure, which satisfies the formula. However, in this approach, the tableau is not the maximal model.

A. Arora *et al.* [7] used the same tableau construction approach for real-time applications. Let f be a specification, expressed in **CTL**, for a concurrent program. They proceed as follows. First apply the **CTL** decision procedure to f as in [17, 31]. If f is satisfiable, then the decision procedure yields a model M of f. M can be viewed as the global state transition diagram of a fault-intolerant program P which satisfies f, and P can be extracted from M. To extract a faulttolerant program from M, they first transform M by adding to it transitions that model the failure and subsequent recovery of processes. Failure of P_i in (global) state s is modeled by a fault transition from s to a new state. When all of the fault and recovery transitions have been added to M, the fault-tolerant program is extracted from M. This proposed approach is an application of the tableau construction in [17, 31], so the generated program P cannot cover the **CTL** specification.

In [70], C. S. Pasareanu *et al.* describe the adaptation and application of assume-guarantee style model checking to reasoning about correctness properties of software units written in Ada. Units are fundamentally open systems and must be closed with a definition of environment that they will execute in. In particular, they use **LTL** as a means of specifying assumptions about the environment. When both the assumption and the guarantee are specified in **LTL**, one can simply check the property "assumption implies guarantee" with a model checker like SPIN [42]. **LTL** assumptions can also be used to synthesize refined environments, in which case the assumptions can be eliminated from the formula to be checked. The tableau construction approach here is the same as that in SPIN for generating never claims [43, 44]. The result tableau is a maximal model of the environment assumption because every computation, which satisfies the assumption is a path in the tableau, and that every finite path in the tableau is the prefix of some computation that satisfies the assumption. However, throughout the construction, only safety assumptions

are used such that every path in the tableau has to satisfy the formula. Another disadvantage of this approach is that no tableau reduction techniques are considered during the construction. Consequentially, the size of the result tableau is not optimal.

In [1], Y. Abarbanel *et al.* translate a subset of **CTL** properties into VHDL process, which has sets of error states that can never be entered. They use this VHDL process and the design under verification in a simulator to check if these error states can be actually entered. Our approach synthesizes Verilog HDL as an application of the reduced **ACTL** tableau construction, which will cover all the behaviors of the corresponding properties and the tableaus are used in the model checking to guarantee the satisfiability of the properties.

In [58], M. Maidl proposes an **ACTL** tableau construction procedure in order to prove the common fragment of **ACTL** and **LTL**. Since the purpose of this tableau is not to reduce the size and hence rewriting techniques, dummy labelling, which we used, are not addressed.

Besides the above tableau construction approaches, there are other **LTL** tableau construction methods available in open literatures [59, 60, 81]. The tableau nodes contain only temporal formulas and can be exponential in the size of the **LTL** formula. On the contrary, we are working on the **ACTL** tableau construction, which has different semantics. We also introduce three-value labelling for atomic propositions to reduce the tableau size and synthesize equivalent Verilog HDL code.

1.3.2 Model Reduction

Abstract interpretation [23] is a classic static program analysis approach. It has been used intensively in formal verification and model reduction [20, 50, 79]. There are four steps to apply abstract interpretation in model reduction: (1) find abstract domains for the variables in the program with or without the aid of the property, (2) translate the property to an abstract property according to the abstract domains, (3)

translate the operations in the program into abstract operations, and then change the program into an abstract program, (4) perform formal verification based on the abstract program and the abstract property, then deduce whether or not the concrete program satisfies the concrete property.

The key step in the approach is to define the abstract domains. For example, consider a property containing a proposition "$x = 0$", where x is an integer in the implementation. Since an integer variable has infinitely many values, the state space of the implementation potentially is infinite. However, to prove this specific property, we actually do not need to know all the information about the integer x, rather, we only need to know if x is zero or non-zero. In this case, we can put the abstract domain of x as $\{neg, zero, pos\}$ instead of the infinite values, where neg means x is a negative number; $zero$ means x is 0; pos means x is a positive number. Since the value domain of x is changed, the corresponding operations on x has to be changed as well. For example, the abstract operation of '+', '$+_{abs}$', becomes $zero +_{abs} zero = zero$, $pos +_{abs} pos = pos$, $pos +_{abs} neg = \{neg, zero, pos\}$, etc. We can find that non-determinism is introduced into the operation in the latter case. This type of non-determinism over-approximates the concrete model, and is the source of *false-negative* problems in the verification.

How to get the abstract domains is always a big problem in this approach. In the current existing abstract interpretation based approaches [22], the abstract domain is defined by the user. Namely, the user has to define, firstly the abstract values of one variable; secondly, the mapping between the concrete values and the abstract values. Once the abstract domain has been defined, the abstract operations can be generated [8]. However, in some approaches, the abstract operations have to be given by the user [20].

In practice, real systems are much more complicated than the above example. For different data types, and different operator types, we have to define different kinds of abstract interpretations, for example, for integers signs, for integer

even/odd, for integer modulo-k, for sets, and so on [28]. Namely for one variable, there are many abstract interpretation choices. So, it is important that these choices do not conflict. For example, integer y is abstracted as $\{pos, zero, neg\}$, while integer z is abstracted as $\{even, odd\}$. If there is an assignment $y = z$ in the model, it would be a conflicting abstract type, since it is not clear how to covert a $\{pos, zero, neg\}$ type to a $\{even, odd\}$ type. Although tools can help users select different abstract interpretations during the verification [29], a lot of user interaction is involved. Once a correct abstract mapping has been established, translating a concrete program to an abstract program is not difficult. We can replace all the occurrences of the abstracted variable by its abstract type. For example, the definition "$int\ x$" can be replaced by "$\{pos, zero, neg\}\ x$", and "$x + y$" by "$plus\_abs(x, y)$", and so on.

In our approach, although we analyze the value as well, we do not need to define abstract domains. What we will do is to simply find all the redundant values and use one typical value in the redundant value set instead of all the redundant values. Our approach is, in some cases, not as efficient as abstract interpretation, but we do not have the false-negative problem and the approach is automatic.

In [87], K. Yorav proposed ways to use the high level description (program text) of a system in order to improve the model checking process by reduction. The approaches are based on program static analysis, and analyze the control flow graph of a program to reveal runtime information of the program, without actually running it. The idea is to use static analysis to build a reduced Kripke structure for the program. The reduced Kripke is equivalent to the original program behavior with respect to **CTL*** without the next state operator. Two methods are presented and compared which use static analysis to create reduced models for programs. The first method, called "path reduction", reduces according to control, and the second, called "dead-variable reduction", reduces according to data. Both methods automatically create a reduced Kripke structure directly out of the syntax of the program (the control flow graph), thus avoiding the need to create the full Kripke structure.

The path reduction method excludes possible interleaving between processes. In the reduction, all the paths between two states containing variables of interest are examined. All the states in the path that will not affect the variables of interest are compressed in a single step. The dead-variable reduction excludes some of the successors of a state in which the variables of interest are dead. Our proposed approach has advantages over this approach because: (1) our abstract model is less restrictive since in Yorav's approach the program is required to terminate while ours does not, (2) because we use reflexive and transitive closure as the transitions in the Kripke structure of the state space, the state space in our approach is inherently smaller, (3) the reduction in Yorav's approach is simply to compress paths while we do not limit our reduction on paths, so our approach has a better performance in the reduction.

In [48], Miller and Katz proposed a path reduction approach to eliminate invisible states from the model of a program, where invisible states are states for which all the entering transitions cannot influence the specification. Their method constructs the projected visible state space relative to a specification through a data flow traversal that eliminates invisible states. The construction of the visible state space requires a linear traversal of a model that is somewhat reduced from the original model of the system, but is still larger than the reduced model which is produced. The difference from our approach is that we produce the reduced model from the syntactic model of the program and not from the Kripke structure representing it. The syntactic model is significantly smaller since it expands only the program counter and not the program variables, which are the source for the enormous size of the semantic model.

In [67], K. S. Namjoshi *et al.* proposed a reduction approach which translates a variable with large value domain, for example an integer, into a set of predicates. These predicates are determined by the automated syntactic analysis of the program under verification. The reduction result is a reduced program text, not

an explicit-state graph, namely this reduced program can be reduced further using other explicit-state reduction methods. We propose a different approach from this one by generating abstract domains instead of predicates.

Our reductions are also related to works like cone-of-influence reduction [55, 54]. However, our approach is an extension of these approaches because we analyze the dependency between the *values* of variables in addition to the dependency between variables, thus the dependency relation is more accurate.

1.3.3 Summary

In light of the above related work review and discussions, we believe the contributions of this thesis are twofold as follows:

1. We propose an environment synthesis method for compositional verification, which constructs the environment of a module from temporal formulas via reduced tableaus. Compared with existing related work, the size of this synthesized environment is smaller, which is a key factor in order to make compositional verification. We also formally prove that the synthesized environment covers all the models satisfying the environment temporal properties. Based on this approach, we implemented a tool, which synthesizes the environment assumptions in the compositional verification into Verilog HDL executable programs.

2. We propose a model reduction approach, which is based on the syntactic and semantic analysis of Verilog HDL programs. Compared with previously existing related work, the proposed approach is automatic and has better performance on large datapath designs. On the other hand, since normally the size of the source code is much less than that of its state space or other intermediate forms, this approach uses less resources with respect to the CPU time and memory space throughout the reduction.

To illustrate the above approaches, we used an ATM (Asynchronous Transfer Mode) switch fabric from Nortel Networks as a real case study, which size is beyond the capability of current plain model checking tools.

1.4 Outline of the Thesis

The rest of the thesis is organized in four parts as follows. Part I provides the required theoretical background on model checking and temporal logics; Part II deals with the tableau construction and environment synthesis approach; Part III is dedicated to the proposed static model reduction; Part IV finally covers a detailled case study on both approaches. More specifically:

In Chapter 2, we overview the model checking approach including system models, temporal logics, and model checking procedures. We also highlight the relations between the system models and temporal logics.

In Chapter 3, we review the existing compositional verification approaches, especially assume-guarantee reasoning. Based on these approaches, we define the compositional theorem with respect to systems with different signal dependencies. The compositional theorem gives the conditions where a global property can be decomposed into local properties.

In Chapter 4, we propose the reduced tableau construction method. It provides the theoretical basis for the environment synthesis techniques.

In Chapter 5, we describe the environment synthesis approach where temporal logic formulas are synthesized into executable Verilog modules. These Verilog modules are used as the environment assumptions in the compositional verification.

In Chapter 6, we review the existing model reduction approaches applied in model checking including cone-of-influence, symbolic abstraction, etc.

In Chapter 7, we introduce the abstract program syntax and semantics; this

abstract program is the model of the design under verification. Based on this abstract program model, we propose the syntactic model reduction methods. After the reduction, we get a reduced program, which preserves the specifications. We also apply the compositional verification techniques on a few benchmarks and a forwarding table lookup processor, with emphasis on demonstrating the power of the compositional verification framework.

In Chapter 8, we describe a case study based on an ATM switch fabric from Nortel Networks and illustrate the modeling and verification of the fabric, respectively.

In Chapter 9, we conclude the thesis and outline future research directions.

Chapter 2

Model Checking: Preliminaries

Model checking [31] is to check the correctness of a design (FSM)against a set of general properties. The FSM is formally depicted by a state transition structure, i.e., *Kripke* structure [51], while the general properties are formally described by *temporal logic* [30] formulas. The FSM satisfies the general properties if and only if the Kripke structure satisfies the temporal logic formulas. Usually, in order to improve the model checking efficiency, the Kripke structure is often illustrated by *binary decision diagrams* [61]. This is the so-called symbolic model checking. In the following sections, we will describe the above ideas in detail. Besides, we will review the *fairness* and *simulation relations*. All these concepts provide the background of our research in the sequel.

2.1 System Modeling

Modeling a system formally is the first step in order to verify it. In this thesis, we are primarily concerned with synchronous hardware systems and their behaviors over time. The first feature of such a system we want to capture is its states. Let $V = \{v_1, \ldots, v_n\}$ be the set of system variables which range over a finite set $\mathcal{D}$. A *valuation* for V is a function that associate a value in $\mathcal{D}$ with each variable v in V. A

state is the valuation for V at a particular instant of time or a snapshot. In addition to the state, we also need to know how the states change with respect to some actions occurring in the system, which is given by a *transition* between the state before the action and the state after the action. So, the behaviors of the system can be described by the states and the change of the states, namely transitions. Formally, a state transition graph by the name of *Kripke* structure [51] is used to capture the behaviors.

Definition 2.1.1 [51] *Let AP be a set of atomic propositions. A* Kripke structure *$\mathcal{K}$ over AP is a four tuple $\mathcal{K} = (S, I, \delta, \lambda)$ where*

- *S is the finite set of states.*
- *$I \subseteq S$ is the set of initial states.*
- *$\delta \subseteq S \times S$ is a transition relation that must be total, that is, for every state $s \in S$ there is a state $s' \in S$ such that $\delta(s, s')$. Namely, s' is the next state of s.*
- *$\lambda : S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.*

A *computation* of the above transition system is an infinite sequence of states, where each state is obtained from the previous state by some transitions. This intuition is captured by the *path* as defined below.

Remark 2.1.1 [22] *A* path *in the structure K from a state s_0 is an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ such that for all $i \geq 0$, $\delta(s_i, s_{i+1})$. We will use the notation π^n for the suffix of π which begins at s_n, and π_n for the i^{th} state in the path.*

In order to avoid state space explosion problem, techniques are developed to replace a large structure by a smaller structure, which satisfies the same properties.

More generally, given that the properties are in a logic $\mathcal{L}$ (which will be introduced in the next section), and a transition structure $\mathcal{K}$, we would like to find a smaller structure $\mathcal{K}'$ that satisfies exactly the same set of properties of the logic $\mathcal{L}$ as $\mathcal{K}$. In order to accomplish this goal, we need a notion of equivalence between structures that can be efficiently computed and guarantees that two structures satisfy the same set of properties in $\mathcal{L}$. Following, we review two relations between the structures: *bisimulation* and *simulation.*

Definition 2.1.2 [69] *Given two structures* $\mathcal{K} = (S, I, \delta, \lambda)$ *and* $\mathcal{K}' = (S', I', \delta', \lambda')$ *with the same set of atomic proposition* AP, *a relation* $B \subseteq S \times S'$ *is a* bisimulation relation *between* $\mathcal{K}$ *and* $\mathcal{K}'$ *if and only if for all* s *and and* s', *if* $B(s, s')$ *then the following conditions hold.*

- $\lambda(s) = \lambda'(s)$
- *for every state* s_1 *such that* $\delta(s, s_1)$, *there is a state* s'_1 *with the property that* $\delta'(s', s'_1)$ *and* $B(s_1, s'_1)$
- *for every state* s'_1 *such that* $\delta(s', s'_1)$, *there is a state* s_1 *with the property that* $\delta(s, s_1)$ *and* $B(s_1, s'_1)$

Two transition structure $\mathcal{K}$ and $\mathcal{K}'$ are bisimulation equivalent, denoted $\mathcal{K} \equiv \mathcal{K}'$, if there exists a bisimulation relation B such that for every initial state $s_0 \in I$ in $\mathcal{K}$ there is an initial state $s'_0 \in I'$ in $\mathcal{K}'$ such that $B(s_0, s'_0)$. In addition, for every initial state $s'_0 \in I'$ in $\mathcal{K}'$ there is an initial state $s_0 \in I$ in $\mathcal{K}$ such that $B(s_0, s'_0)$. Bisimulation guarantees that two structures have the same behaviors.

On the other hand, *simulation* relates a structure to an abstraction of another structure. Because the abstraction can hide some of the details of the original structure, it might have a smaller set of atomic propositions. The simulation guarantees that every behavior of a structure is also a behavior of its abstraction. However, the abstraction might have behaviors that are not possible in the original structure.

For example, in an actual implementation, some events always occur within twenty execution steps. But in an abstraction, this event may occur after any number of execution steps.

Definition 2.1.3 [66] *Given two structures* $\mathcal{K} = (S, I, \delta, \lambda)$ *and* $\mathcal{K}' = (S', I', \delta', \lambda')$ *with* $AP' \subseteq AP$, *a relation* $H \subseteq S \times S'$ *is a* simulation relation *between* $\mathcal{K}$ *and* $\mathcal{K}'$, *if and only if for all* s *and* s', *if* $H(s, s')$ *then the following conditions hold.*

- $\lambda(s) \cap AP' = \lambda'(s)$
- *for every state* s_1 *such that* $\delta(s, s_1)$, *there is a state* s'_1 *with the property that* $\delta'(s', s'_1)$ *and* $H(s_1, s'_1)$

Transition structure $\mathcal{K}'$ simulates Transition structure $\mathcal{K}$, denoted by $\mathcal{K} \preceq \mathcal{K}'$, if there exists a simulation relation H such that for every initial state s_0 in $\mathcal{K}$ there is an initial state s'_0 in $\mathcal{K}'$ for which $H(s_0, s'_0)$. The simulation relation can also be proven as a preorder relation[22], namely it is a reflexive and transitive relation.

The simulation is the relation to replace a large structure by a smaller structure, which satisfies the same temporal logic properties [22].

2.2 Temporal Logics

Formally, a *property* of a system can be defined as a set of *behaviors*, i.e., infinite sequences of states, which represent the desired behaviors of the system. The property usually is specified by *temporal logic* [30] formulas interpreted over the Kripke transition structure in Section 2.1. Temporal logic uses atomic propositions and boolean connectives such as conjunction ($\wedge$), disjunction ($\vee$), and negation ($\neg$) to build up complicated expressions describing the states and the transitions.

Generally, temporal logic is the formalism for describing sequences of transitions between states in a system. In the temporal logics that we will consider, time is

not mentioned explicitly; instead, a temporal logic formula might specify that eventually some states are reached, or that an error state is never entered. Properties like *eventually* or *never* are specified using special *temporal operators*. These operators can also be combined with boolean connectives. Here, we will introduce the temporal logics which are used throughout the thesis: **CTL***, **CTL**, and **ACTL**.

2.2.1 CTL*

Conceptually, **CTL*** (Computation Tree Logic *) formulas describe properties of *computation trees*. The tree is formed by designating a state in a Kripke transition structure as the *initial state* and then unwinding the structure into an infinite tree with the designated state at the root, as shown in Figure 2.1. The computation tree shows all of the possible executions starting from the initial state.

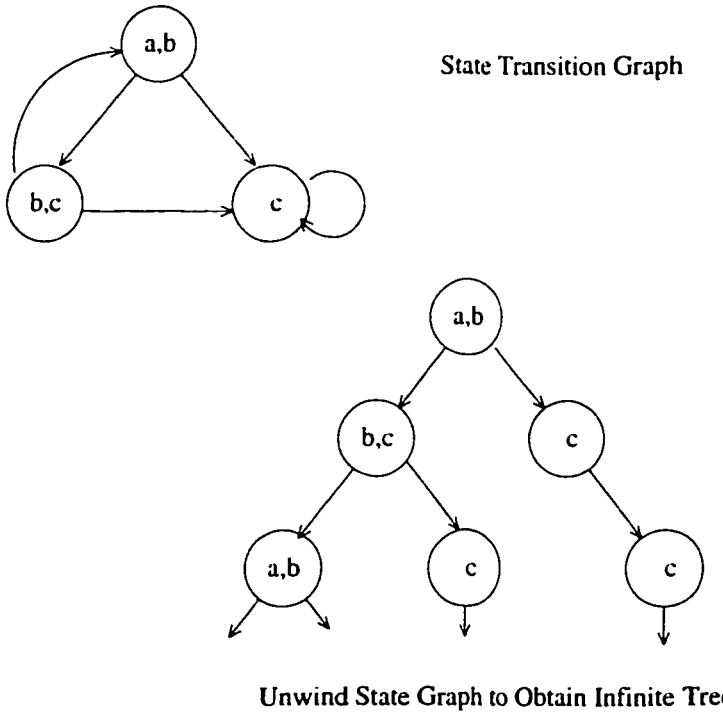


Figure 2.1: Computation Tree

CTL* formulas are composed of *path quantifiers* and *temporal operators*. The

path quantifiers are used to describe the branching structure in the computation tree. There are two such quantifiers **A** ("for all computation paths") and **E** ("for some computation paths"). These quantifiers are used in a particular state to specify that all of the paths or some of the paths starting at the state have some property. The temporal operators describe properties of a path through the tree. There are five basic operators:

- **X** ("next time") requires that a property holds in the second state of the path.
- **F** ("eventually or in the future") asserts that a property will hold at some states on the path.
- **G** ("always or globally") specifies that a property holds at every state on the path.
- **U** ("until") is used to combine two properties. It holds if there is a state on the path where the second property holds, and at every preceding state on the path, the first property holds.
- **V** ("release") is the global dual of **U**. It requires that the second property hold along the path up to and including the first state where the first property holds. However, the first property is not required to hold eventually.

Using the basic operators **A** and **U**, other operators can be defined as abbreviations, e.g.,

- $\mathbf{E}f = \neg\mathbf{A}\neg f$.
- $\mathbf{F}f = \mathbf{true}\mathbf{U}f$.
- $\mathbf{G}f = \neg\mathbf{F}\neg f$.
- $f\mathbf{V}g = \neg(\neg f\mathbf{U}\neg g)$.

Based on the path quantifiers and temporal operators, we can define state formulas and path formulas as follows.

Definition 2.2.1 *Let AP be the set of atomic propositions. There are two types of formulas in* **CTL*** *: state formulas and path formulas, which are defined as follows.*

- *If $p \in AP$, then p is a* state formula.
- *If f and g are* state formulas, *then $\neg f$ and $f \vee g$ are state formulas.*
- *If f is a state formula, then f is also a path formula.*
- *If f and g are path formulas, then $\neg f$, $f \vee g$, $\mathbf{X}f$, and $f\mathbf{U}g$ are path formulas.*
- *If f is a path formula, then $\mathbf{A}(f)$ is a state formula.*

If f is a state formula, the notation $\mathcal{K}, s \models f$ means that f holds at state s in the transition structure $\mathcal{K}$. Similarly, if f is a path formula, $\mathcal{K}, \pi \models f$ means that f holds along path π in transition structure $\mathcal{K}$. When the transition structure $\mathcal{K}$ is clear from context, we will usually omit it. The relation "$\models$" is defined inductively as follows (assuming that p is an atomic proposition, f_1 and f_2 are state formulas and g_1 and g_2 are path formulas):

- $s \models p$ if and only if $p \in L(s)$.
- $s \models \neg f_1$ if and only if $s \not\models f_1$.
- $s \models f_1 \vee f_2$ if and only if $s \models f_1$ or $s \models f_2$.
- $s \models \mathbf{A}(g_1)$ if and only if for all paths π starting with s, $\pi \models g_1$.
- $\pi \models f_1$ if and only if $\pi_1 \models f_1$.
- $\pi \models \neg g_1$ if and only if $\pi \not\models g_1$.

- $\pi \models g_1 \vee g_2$ if and only if $\pi \models g_1$ or $\pi \models g_2$.
- $\pi \models \mathbf{X}(g_1)$ if and only if $\pi^1 \models g_1$.
- $\pi \models (g_1)\mathbf{U}(g_2)$ if and only if there exists a $i \geq 0$ such that $\pi^i \models g_2$ and for all $0 \leq j < i, \pi^j \models g_1$.
- $\pi \models (g_1)\mathbf{V}(g_2)$ if and only if for every $j \geq 0$, if for every $0 \leq i < j, \pi^i \not\models g_1$ then $\pi^j \models g_2$.

Example 2.2.1 $\mathbf{A}(\mathbf{FG}p) \vee \mathbf{AG}(\mathbf{EF}p)$ *is a* **CTL*** *property meaning that p always eventually occurs and then holds, or p always possibly occurs.*

2.2.2 CTL and LTL

There are two sublogics of **CTL***: one is branching time logic (**CTL**) and the other is linear time logic (**LTL**). The distinction between the two is in how they handle branching in the underlying computation tree. In branching time temporal logic the temporal operators quantify over the paths that are possible from a given state. In linear time temporal logic, operators are provided for describing events along a single computation path.

CTL is one of the temporal logics used in model checking [17], and is a restricted subset of **CTL***, where each of the operators **G**, **F**, **X**, **V**, and **U** must be immediately preceded by a path quantifier **A** or **E**. More precisely, **CTL** is the subset of **CTL*** which is obtained if the following rule is used to specify the syntax of path formulas.

- If f and g are state formulas, then $\mathbf{X}f$, $f\mathbf{U}g$ are path formulas.

Linear Temporal Logic (**LTL**) [30], on the other hand, will consist of formulas that have the form $\mathbf{A}f$ where f is a path formula in which the only state subformulas permitted are atomic propositions. More precisely, an **LTL** path formula is either:

- If $p \in AP$, then p is a path formula.
- If f and g are path formulas, then $\neg f$, $f \wedge g$, $f \vee g$, $\mathbf{X}f$, $\mathbf{F}f$, $\mathbf{G}f$, $f\mathbf{U}g$, and $f\mathbf{V}g$ are path formulas.

LTL has different expressive power as **CTL**. For example, the **CTL** formula that is equivalent to **LTL** formula $\mathbf{A}(\mathbf{FG}p)$ does not exist. This formula expresses the property that along every path, there are some states from which p will hold forever. Likewise, there is no **LTL** formula that is equivalent to the **CTL** formula $\mathbf{AG}(\mathbf{EF}p)$ meaning that there always eventually exists a p. The disjunction of these two formulas $\mathbf{A}(\mathbf{FG}p) \vee \mathbf{AG}(\mathbf{EF}p)$ is a **CTL*** formula that is not expressible in either **LTL** or **CTL**.

Here, in this thesis, we will focus on **CTL** and its sublogics, where the semantics of the operators are the same as that in **CTL***. In **CTL**, each of the **CTL** operators can also be expressed in terms of three operators **EX**, **EG**, and **EU**. For example:

- $\mathbf{AX}f = \neg\mathbf{EX}(\neg f)$
- $\mathbf{AG}f = \neg\mathbf{EF}(\neg f)$
- $\mathbf{AF}f = \neg\mathbf{EG}(\neg f)$
- $\mathbf{EF}f = \mathbf{E}(true\mathbf{U}f)$
- $\mathbf{A}(f_1\mathbf{U}f_2) \equiv \neg\mathbf{E}(\neg f_2\mathbf{U}\neg f_1 \wedge \neg f_2) \wedge \neg\mathbf{EG}\neg f_2$
- $\mathbf{A}(f_1\mathbf{V}f_2) \equiv \neg\mathbf{E}(\neg f_1\mathbf{U}\neg f_2)$
- $\mathbf{E}(f_1\mathbf{V}f_2) = \neg\mathbf{A}(\neg f_1\mathbf{U}\neg f_2)$

The four operators that are used most widely are illustrated in Figure 2.2. Each computation tree has the state s_0 as its root.

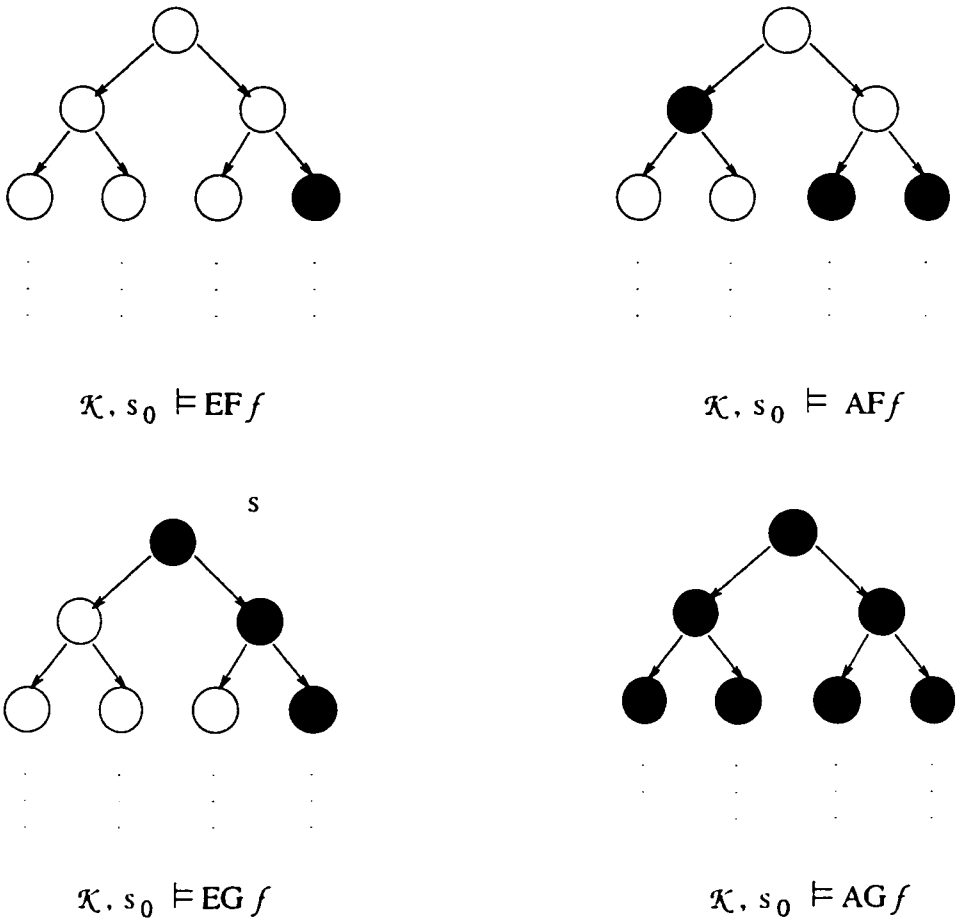


Figure 2.2: Basic **CTL** Operators

2.2.3 ACTL

ACTL [36] is a subset of **CTL**, where only *universal* path quantifier, namely only **A** is allowed.

The set of well-formed universal computation tree logic (**ACTL**) are constructed from a set of atomic propositions AP which represent properties of individual states, the standard boolean operators, the temporal operators **X**, **U** and **V**, and the universal path quantifier **A**.

Similarly, we introduce additional temporal operators as abbreviations: $\mathbf{AG}f_1$ is defined as $\mathbf{A}(\mathbf{false}\ \mathbf{V}f_1)$, which means f_1 always holds, and $\mathbf{AF}f_1$ is defined as $\mathbf{A}(\mathbf{true}\ \mathbf{U}f_1)$, which means f_1 eventually holds. In the following sections, if it

does not lead to confusion, we may use $f_1\mathbf{AU}f_2$ or $f_1\mathbf{AV}f_2$ instead of $\mathbf{A}(f_1\mathbf{U}f_2)$ or $\mathbf{A}(f_1\mathbf{V}f_2)$, respectively.

All **ACTL** formulas belong to **CTL** but the reverse does not hold (syntactically). The syntax of **CTL** requires that the temporal operators **X**, **F**, **G**, **U** and **V** are immediately preceded by a path quantifier **E** or **A**. If this restriction is dropped, then the more expressive branching temporal logic **CTL*** is obtained. It contains for example $\mathbf{A}(\mathbf{F}f_1 \wedge \mathbf{G}f_2)$ formula which is not an **CTL** formula. The relationship among **ACTL**, **CTL**, **LTL** and **CTL*** is depicted in Figure 2.3.

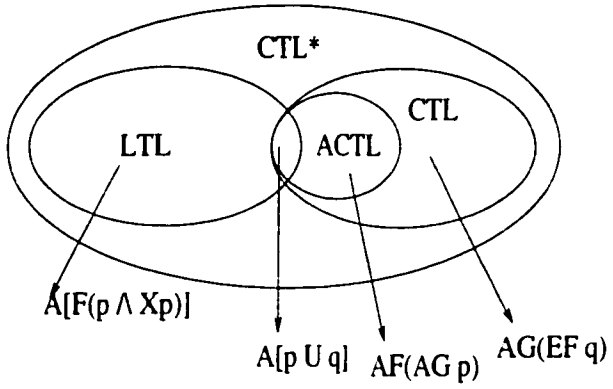


Figure 2.3: Relationship between **ACTL**, **CTL**, **LTL** and **CTL***

2.2.4 Fairness

Another issue we need to consider is *fairness* in the context of temporal logics. In many cases, we are only interested in the correctness along fair computation paths. For example, if we are verifying an arbiter, we may wish to consider only those executions in which the arbiter does not ignore one of its request inputs forever. Alternatively, we may want to consider communication protocols that operate over reliable channels, which have the property that no message is ever continuously transmitted but never received. We call this fairness constraint. A fairness constraint is an arbitrary set of states, which satisfy certain constraints. A fair path in

the transition structure with respect to fairness constraints must contain an element of each fairness constraint infinitely often. Formally,

Definition 2.2.2 *A* fair Kripke structure *is a 5 tuple* $\mathcal{K} = (S, I, \delta, \lambda, \mathcal{F})$, *where*

- S, I, δ, *and* λ *are defined as before;*
- $\mathcal{F} \subseteq 2^S$ *is a set of fairness constraints (often called generalized Büchi acceptance conditions).*

Let $\pi = s_0, s_1, \ldots$ be a path in $\mathcal{K}$. The notation $inf(\pi)$ denotes the set of states that occur infinitely many in π. Namely

$$inf(\pi) = \{s | s = s_i \text{ for infinitely many } i\}$$

We say that π is a fair path if and only if for every $\mathcal{F}_i \in \mathcal{F}$, $inf(\pi) \cap \mathcal{F}_i \neq \emptyset$.

The semantics of the temporal logics with respect to a fair Kripke structure is very similar to their semantics with respect to an ordinary Kripke structure, except that all path quantifiers only range over fair paths.

2.2.5 Temporal Logics and Relations

The relationships between the temporal logics and the relations between transition structures, i.e., bisimulation and simulation, are concluded by the following two theorems.

Theorem 2.2.1 [22] *If* $\mathcal{K} \equiv \mathcal{K}'$ *then for every* **CTL*** *formula* f, $\mathcal{K}' \models f$ *if and only if* $\mathcal{K} \models f$.

Theorem 2.2.2 [22] *Suppose* $\mathcal{K} \preceq \mathcal{K}'$. *Then for every* **ACTL** *formula* f, $\mathcal{K}' \models f$ *implies* $\mathcal{K} \models f$.

In order to tackle state space explosion, a smaller structure, which can preserve temporal logic properties of a larger structure is used. The most often used relation between these two structures is the preorder (simulation) relation ($\preceq$). The preorder relation preserves the **ACTL** properties. Namely, if H is a *preorder* ($\Leftarrow$) relation between $\mathcal{K}$ and $\mathcal{K}'$, and f is an **ACTL** formula, then $\mathcal{K} \models f \Leftarrow \mathcal{K}' \models f$. Another often used relation is the *equivalence* ($\Leftrightarrow$) relation, namely if B is an equivalence relation between $\mathcal{K}$ and $\mathcal{K}'$, and f is a **CTL*** formula, then $\mathcal{K} \models f \Leftrightarrow \mathcal{K}' \models f$.

2.3 CTL Model Checking

We have seen that **CTL** model checking is an automatic formal verification approach to check **CTL** formulas with respect to the Kripke transition structure. Its significance comes from the great ease of use of fully algorithmic methods, as well as from the fact that many synchronization and communication systems can be modeled as finite state machines. Finite state machines then can be modeled by transition structures where each state has a bounded description, and hence can be characterized by a fixed number of Boolean atomic propositions. In the **CTL** model checking [18], the correctness of a finite state machine is verified with respect to a desired behavior by checking whether a labeled transition structure that models the machine satisfies an **CTL** formula that specifies this behavior.

Given a transition structure $M = (S, I, \delta, \lambda)$ that represents a finite-state concurrent system and an **CTL** formula f expressing some desired specification, model checking is to find the set of all states in S that satisfy f: $\{s \in S | M, s \models f\}$. The system satisfies the specification provided that all of the initial states I are in the set.

The model checking process can be described as follows. Assume that we want to determine which states in S satisfy the **CTL** formula f. The algorithm will operate by labeling each state s with the set $label(s)$ of subformulas of f which are

true in s. Initially, $label(s)$ is just $\lambda(s)$. The algorithm then goes through a series of stages. During the i^{th} stage, subformulas with $i-1$ nested **CTL** operators are processed. When a subformula is processed, it is added to the labeling of each state in which it is true. Once the algorithm terminates, we will have that $M, s \models f$ if and only if $f \in label(s)$.

Recall that any **CTL** formula can be expressed in terms of $\neg, \vee, \mathbf{EX}, \mathbf{EU}$ and $\mathbf{EG}$. Thus, for the intermediate stages of the algorithm it is sufficient to be able to handle six cases, depending on whether f is atomic or has one of the following forms: $\neg f, f_1 \vee f_2, \mathbf{EX}f, \mathbf{E}[f_1 \mathbf{U} f_2]$, or $\mathbf{EG}f$.

For formulas of the form $\neg f$, we label those states that are not labelled by f. For $f_1 \vee f_2$, we label any state that is labelled either by f_1 or by f_2. For $\mathbf{EX}f$, we label every state that has some successor labelled by f.

To handle formulas of the form $g = \mathbf{E}[f_1 \mathbf{U} f_2]$, we first find all states that are labelled with f_2. We then work backwards using the converse of the transition relation δ and find all states that can be reached by a path in which each state is labelled with f_1. All such states should be labelled with g.

The case in which $g = \mathbf{EG}f$ is slightly more complicated. It is based on the decomposition of the graph into nontrivial strongly connected components. A strongly connected component C is a maximal subgraph such that every node in C along a directed path entirely contained within C. C is nontrivial if and only if either it has more than one node or it contains one node with a self-loop.

In order to handle an arbitrary **CTL** formula f, the state-labelling algorithm is applied to the subformula of f, starting with the shortest, most deeply nested, and work outward to include all of f. By proceeding in this manner we guarantee that whenever we process a subformula of f, all its subformulas have already been processed. Each pass of the above process takes time $O(|S| + |\delta|)$ and since f has at most $|f|$ different subformulas, the entire algorithm requires time $O(|f| \times (|S| + |\delta|))$

From the complexity, we can find that model checking suffers from the so-called state space explosion problem. In a concurrent setting, the system under consideration is typically the parallel composition of many modules. As a result, the size of the state space of the system ($|S| + |\delta|$) is the product of the sizes of the state spaces of the participating modules. This gives rise to state spaces of exceedingly large sizes, which makes linear time algorithms impractical.

Symbolic model checking is to check **CTL** properties by representing Kripke structure using Reduced Ordered Binary Decision Diagrams (ROBDDs) [12]. Consider the two states structure shown in Figure 2.4. In this case there are two state

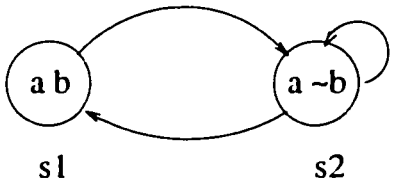


Figure 2.4: Two-state Kripke structure

variables, a and b. We introduce two additional state variables, a' and b' to encode successor states. Thus, we will represent the transition from state s_1 to s_2 by the conjunction

$$(a \wedge b \wedge a' \wedge \neg b')$$

The Boolean formula for the entire transition relation is given by

$$(a \wedge b \wedge a' \wedge \neg b') \vee (a \wedge \neg b \wedge a' \wedge \neg b') \vee (a \wedge \neg b \wedge a' \wedge b')$$

There are three disjuncts in the formula because the Kripke structure has three transitions. This formula is now converted to an ROBDD to obtain a concise representation for the transition relation.

The symbolic model checking procedure *Check* [61] takes the **CTL** formula to be checked as its argument and returns an ROBDD that represents exactly those states of the system that satisfy the formula. Of course, the output of *Check* depends

on the ROBDD representation of the transition relation of the system being checked. The procedure is defined inductively over the structure of **CTL** formulas. For example, if f is an atomic proposition a, then $Check(f)$ is the ROBDD representing the set of states satisfying a; formulas of the form $\mathbf{EX}f$ is handled by the procedure $Check(\mathbf{EX}f) = CheckEX(Check(f))$, where formula $\mathbf{EX}f$ is true in a state if the state has a successor in which f is true, etc.

2.4 Tableau Construction

Tableau construction [17, 59, 57, 36, 46, 22] is a procedure to translate a temporal logic formula f into an FSM $\mathcal{T}_f$. If $\mathcal{T}_f$ covers all the behaviors of formula f, then we call $\mathcal{T}_f$ the maximal model of f, namely for every structure M, the tableau is required to satisfy

$$M \models f \text{ iff } M \preceq \mathcal{T}_f$$

where $\preceq$ is the simulation relation in Section 2.1. This is the key property of the tableau construction.

As we mentioned in Section 1.2, in the compositional verification, properties only can be true under certain environment assumptions. It is our obligation to describe such environment assumptions in the compositional verification such that informally, we can reason as follows:

1. if the real environment satisfies the environment assumptions φ which is temporal logic formulas; and

2. one module of a system under verification satisfies property ψ under environment assumptions φ

3. then the system satisfies property ψ.

This is the so called assume-guarantee reasoning [56]. In step 2, we have to translate environment assumptions φ into an FSM, i.e., $\mathcal{T}_\varphi$, because we cannot directly compose logic formulas φ with an FSM module. In order to make the reasoning sound, it is required that $\mathcal{T}_\varphi$ cannot miss any behavior contained in φ, namely $\mathcal{T}_\varphi$ is the maximal model of φ. Note that, not all the results of tableau construction approaches are maximal models. Here in this thesis, we only consider the maximal model of the tableau.

Part I

Environment Synthesis

Chapter 3

Compositional Reasoning

In this chapter, we will introduce more technical details of the existing compositional verification approaches [73], especially, the assume-guarantee reasoning, which is the foundation of our research. Throughout the illustration, we propose a system partitioning method using arrays. Besides, we propose the reasoning rules for systems with different signal dependencies, such as circular reasoning for systems with circular signal feedbacks. We also formally prove the soundness of our reasoning rules.

3.1 Introduction

Many finite state systems are composed of multiple modules running in parallel. The specifications for such systems can often be decomposed into properties that describe the behavior of small parts of the system. An obvious strategy is to check each of the local properties using only the part of the system that it describes. If we can deduce that the system satisfies each local property, and that the conjunction of the local properties implies the overall specification, then we can conclude that the complete system satisfies this specification as well.

This means that we separate the specification of a system into properties of its components and verify the properties of the components separately. This of course

leaves us the obligation of proving that the component specifications in turn imply the specification of the entire system. This is called *compositional verification* [57].

In the simplest case, imagine we have a system composed of two modules M_1 and M_2, which communicate with each other over a channel or channels, and also communicate with their environment (Figure 3.1)

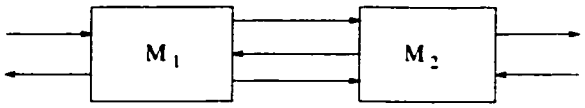


Figure 3.1: Two Reactive Modules

We denote this system $M_1 \| M_2$, for the parallel composition of M_1 and M_2. Given component specifications φ_{M_1} for module M_1 and φ_{M_2} for module M_2, we would like to reason according to the following schema:

$$\frac{\begin{array}{r} M_1 \models \varphi_{M_1} \\ M_2 \models \varphi_{M_2} \\ \varphi_{M_1}, \varphi_{M_2} \models \varphi \end{array}}{M_1 \parallel M_2 \models \varphi}$$

There are a number of difficulties involved in developing a verifier that can support this style of reasoning. First, since it is often the case that the local property is only true under certain environment conditions, we need to be able to make assumptions about the environment of the component when doing the verification. These assumptions, which represent requirements on other components, must also be checked in order to complete the verification. This is called the *environment problem* [63]. In addition, we must provide a method for checking that the conjunction of certain local properties implies a given specification. *Assume-guarantee reasoning* is one of the most important solutions in this regard.

In the assume-guarantee reasoning case [57, 36, 6, 21, 62, 4, 39, 64, 41, 65, 6, 40], the environment is expressed by an assumption that module M_1 makes on module M_2 using a temporal formula ψ_{M_2}. These assumptions can be used when

checking that M_1 satisfies its specification φ_{M_1}, and must, of course, be discharged relative to M_2. To do this, a reasoning rule is used such as the following:

$$\frac{\begin{array}{c} M_2 \models \psi_{M_2} \\ M_1 \parallel \psi_{M_2} \models \varphi_{M_1} \end{array}}{M_1 \parallel M_2 \models \varphi_{M_1}}$$

where $M_1 \parallel \psi_{M_2}$ means $M_1 \parallel M_2'$ with $M_2' \models \psi_{M_2}$. Here, the environment problem is just deferred but not solved because we have to show $M_2 \models \psi_{M_2}$ without any environment.

In the following, we will introduce the assume-guarantee approach, and then we discuss the problematic to deal with.

3.2 Assume-guarantee Reasoning

In [57], D. E. Long and O. Grumberg have shown that given a **CTL** formula φ, there is no efficient algorithm[1] to develop a fully general system of inference rules that will handle arbitrary temporal properties. In this case, the decomposed specifications cannot be proved to imply overall specification. It is a big disadvantage of this approach.

Although no mechanical proof was given, we can image the reason. First, it is generally not possible to decompose a formula into subformulas, check the subformulas, and combine the results. For example, consider checking $\mathbf{EX}(\alpha = 1) \vee \mathbf{EX}(\alpha = 0)$ on a Moore machine where α is an input ranging over $\{0,1\}$. Obviously, the formula as a whole will be true regardless of what the environment does. However, $\mathbf{EX}(\alpha = 1)$ is certainly not true for all environments, nor is $\mathbf{EX}(\alpha = 0)$. Thus, determining whether the two subformulas are true in all environments does not help us solve the overall problem.

[1] Here, efficient algorithm means this algorithm is exponential in input state variables, but polynomial in states and transition relations etc., because we cannot find an algorithm with a run time sub-exponential with respect to input state variables.

Second, because of the complexity of models under verification, we sometimes cannot just look at immediate successors when evaluating temporal formulas. Consider the Moore machine in Figure 3.2 [57], where we try to determine whether **EXEX**b = 1 is true for all systems containing the Moore machine shown in the figure. In standard **CTL** model checking, we would use the truth value for **EX**b = 1 at the two successors of the initial states to determine whether **EXEX**b = 1 was true at the initial state. For this example, there are environments that make **EX**b = 1 false at the left successor and others that make the formula false at the right successor. However, the overall formula is in fact true in all environments. This is because no environment can distinguish between the two successors based on their labelling. Hence, if the environment supplies the input a = 1 to the left successor, b = 1 becomes true in the next state. If it supplies only a = 0, then it must also supply a = 0 to the right successor, and this will again lead to a state where b is true. Thus we cannot just look at immediate successors when evaluating temporal operators.

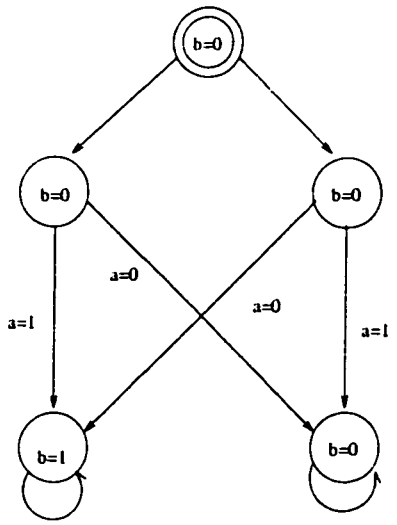


Figure 3.2: Property Decomposition

Because of the above reasons, generally, it is widely accepted that a temporal logic without **E**-path quantifiers, namely, **ACTL**, should be used instead of **CTL**

in the compositional verification.

The assume-guarantee style of the compositional verification was first advocated in the context of temporal logic by Pnueli [77]. In Pnueli's system, triples of the form $\langle\varphi\rangle M\langle\psi\rangle$ are used. The most common reading of such a triple is "if the environment of M satisfies φ, then M in this environment satisfies ψ" A typical chain of reasoning would be as follows:

$$\frac{\begin{array}{c}\langle\rangle M\langle\varphi\rangle \\ \langle\varphi\rangle M'\langle\psi\rangle\end{array}}{\langle\rangle M\|M'\langle\psi\rangle}$$

Here, we are asserting that:

1. If M satisfies φ and
2. If the environment of M' satisfies φ, then M' satisfies ψ;
3. Then the composition of M and M' will satisfy ψ.

The advantage of doing the verification in this manner is that, first, we never have to examine the composite state space of $M\|M'$. Instead, we check φ using just M, and then check ψ using only M' and the (hopefully simple) assumption φ. Second, this kind of chain reasoning gives us the possibility of doing hierarchical verification. The disadvantage is that the user must determine an appropriate φ. Knowledge of how the system should behave plus feedback from an automatic verifier makes this feasible in practice. More generally, we may use multiple levels of assumptions and guarantees when doing verification. That is, once we have proved a guarantee, we may use that guarantee as an assumption in later stages. Because of this, a somewhat more precise reading of $\langle\varphi\rangle M\langle\psi\rangle$ would be "if the system satisfies φ and contains M, then the system also satisfies ψ". This is because φ may in fact be something that is derived based on earlier assumptions about M, and may reflect these assumptions. Also, ψ may describe the combination of M and its environment, instead of just M. Of course, in order to avoid erroneous conclusions, all chains of deduction, i.e, signal dependencies, must be well founded, i.e., the base assumptions must themselves be proved without any assumptions, like $\langle\rangle M\langle\varphi\rangle$. As

we already mentioned in last section, it is hard to prove a formula of a component without specific environment if we use a full set of **CTL**. An efficient solution might be again **ACTL**.

However, the key issue in assume-guarantee reasoning is to establish the truth of a triple $\langle\varphi\rangle M'\langle\psi\rangle$. An elegant way to obtain a system with the above properties is to provide a preorder relation $\preceq$ on the finite state models that captures the notion of "more behaviors" and to use a logic whose semantics relate to the preorder. The preorder should preserve satisfaction of formulas of the logic, i.e., if a formula is true for a model, it should also be true for any model which is smaller in the preorder, namely,

$$(M_1 \preceq M_2) \wedge M_2 \models \varphi \textit{ implies } M_1 \models \varphi$$

Another requirement for the reasoning is that composition should preserve the preorder. That is,

$$(M_1 \preceq M_1') \wedge (M_2 \preceq M_2') \textit{ implies } (M_1 \| M_2) \preceq (M_1' \| M_2')$$

In additional, following condition is needed in the assume-guarantee reasoning. Namely, for all formulas φ with respect to M there exists a process $\mathcal{T}(\varphi)$ called *tableau* of φ, such that $M \preceq \mathcal{T}(\varphi)$ *if and only if* $M \models \varphi$.

In such a framework, the soundness of the assume-guarantee reasoning

$$\frac{\langle\rangle M_1\langle\varphi\rangle \qquad \langle\varphi\rangle M_2\langle\psi\rangle}{\langle\rangle M_1 \| M_2\langle\psi\rangle}$$

can be proved as [57]:

(1) $\langle\rangle M_1\langle\varphi\rangle \rightarrow M_1 \preceq \mathcal{T}(\varphi)$;

(2) $M_2 \preceq M_2$ because the preorder is reflexive;

(3) $M_2 \| M_1 \preceq M_2 \| \mathcal{T}(\varphi)$;

(4) $\langle\varphi\rangle M_2\langle\psi\rangle \rightarrow M_2 \| \mathcal{T}(\varphi) \preceq \mathcal{T}(\psi)$;

(5) $M_1 \| M_2 \preceq \mathcal{T}(\psi)$ because the preorder is transitive;

(6) $\langle\rangle M_1 \| M_2 \langle\psi\rangle$ □

In [36], a notion of "*simulation relation* $\preceq$" between state transition systems is introduced as the preorder. We could in fact view this preorder relation as the basic relationship between an implementation and a specification. $M_1 \preceq M_2$ means "M_1 refines M_2" or "M_2 simulates M_1" or "M_2 abstracts M_1".

Because of the transitivity of $\preceq$, we would get hierarchical verification essentially for free. For example, if $M \preceq M'$ ("M' can simulate M"), and if we want to know whether $M \preceq M''$, then it would be enough to check that $M' \preceq M''$. Here, M' would represent a specification of M that is used to prove a higher level specification M''. The simulation relation will also interact with composition in a nice way: if $M \preceq M'$, then we will have $M \| M'' \preceq M' \| M''$. This type of property allows us to replace an implementation by its specification in a composition.

Another key point in this assume-guarantee reasoning is that any **ACTL** formula ψ has a *maximal* model $\mathcal{T}(\psi)$ which includes all the behaviors of ψ. $\mathcal{T}(\psi)$ can be obtained by a *tableau construction* procedure. This tableau construction maps a formula φ to an associated state transition system $\mathcal{T}(\varphi)$ which is called the tableau of the formula. The standard model checking algorithm is used at one level, then we construct tableaus for the specification formulas and use them as implementations at the next level.

In the above framework, the classic assume-guarantee reasoning can be defined as:

$$\frac{\begin{array}{r} M \preceq \mathcal{T}(\varphi) \\ M' \| \mathcal{T}(\varphi) \preceq \mathcal{T}(\psi) \end{array}}{M \| M' \preceq \mathcal{T}(\psi)}$$

As a final note, this reasoning will actually be working with structures rather than Moore machines. This is mainly because formulas do not have notions of inputs and outputs, so the tableau construction will most naturally produce structures. In

addition, structures can serve as "intermediate language" for representing other, more complex types of models.

Following, we use an example to illustrate the above approach [57]. Let the structures for two circuits be denoted by M and M'. We use $\mathbf{AG}(r = 0 \rightarrow \mathbf{AX}q = 0)$ as an assumption on M', and then prove the desired property $\mathbf{AG}(p = 0 \vee q = 0)\rangle$ by combining this assumption with M:

$$\frac{\langle\rangle M'\langle\mathbf{AG}(r = 0 \rightarrow \mathbf{AX}q = 0)\rangle \quad \langle\mathbf{AG}(r = 0 \rightarrow \mathbf{AX}q = 0)\rangle M\langle\mathbf{AG}(p = 0 \vee q = 0)\rangle}{\langle\rangle M \| M'\langle\mathbf{AG}(p = 0 \vee q = 0)\rangle}$$

Checking $\langle\rangle M'\langle\mathbf{AG}(r = 0 \rightarrow \mathbf{AX}q = 0)\rangle$ will be done with the standard model checking techniques. However, in order to check

$$\langle\mathbf{AG}(r = 0 \rightarrow \mathbf{AX}q = 0)\rangle M\langle\mathbf{AG}(p = 0 \vee q = 0)\rangle$$

we need to construct the tableau for $\mathbf{AG}(r = 0 \rightarrow \mathbf{AX}q = 0)$ and compose it with M. The states of the tableau will have valuations for r and q, plus information about the elementary formulas (For details, see [57]). After constructing the tableau and performing the model checking, we find that

$$\langle\mathbf{AG}(r = 0 \rightarrow \mathbf{AX}q = 0)\rangle M\langle\mathbf{AG}(p = 0 \vee q = 0)\rangle$$

does hold. We hence conclude

$$\langle\rangle M \| M'\langle\mathbf{AG}(p = 0 \vee q = 0)\rangle$$

In above discussion, we illustrated a number of advantages of the assume-guarantee reasoning. However, it does have some disadvantages.

CTL cannot be handled efficiently in current compositional verification including the assume-guarantee reasoning. **ACTL** has to be used instead of **CTL**. In practice, there are actually some cases which cannot be expressed by **ACTL**, such as $\mathbf{AGEF}\neg\varphi$. In addition, as we know before, the reason of this problem is the so-called environment problem, namely we have to prove a tautology of a φ.

Actually, every component will be working under a specific environment. It seems unnecessary to prove a tautology of a temporal logical formula φ. So, can we only prove that φ holds under a specific environment? In the assume-guarantee context, this question can be translated to "can we use assumptions at the very first stage of the reasoning?"

Following, we will describe a circular reasoning, which can avoid the environment problem as in [6, 64, 65, 40]. Briefly, we can express the circular reasoning as:

$$\frac{M_2 \| M_1' \rightarrow M_2' \qquad M_1 \| M_2' \rightarrow M_1'}{M_1 \| M_2 \rightarrow M_1' \| M_2'}$$

Here, we assume that module M_2 satisfies M_2' to prove that M_1 satisfies M_1', and vice versa. Generally, this apparent circularity is not sound. However, this circularity can be broken by induction over time. That is, let the notation $M \uparrow^\tau$ stand for $\forall(t \leq \tau).M$ or "M holds up to time t = τ". We can soundly reason as follows, by induction on τ:

$$\frac{M_1 \uparrow^{\tau-1} \Rightarrow M_2 \uparrow^\tau \qquad M_2 \uparrow^\tau \Rightarrow M_1 \uparrow^\tau}{\forall t.(M_1 \wedge M_2)}$$

In the base case, when $\tau = 0$, note that $M_1 \uparrow^{\tau-1}$ is a tautology. Hence, we have $M_2 \uparrow^0$, and thus $M_1 \uparrow^0$, $M_2 \uparrow^1$, $M_1 \uparrow^1$ and so on. By reasoning inductively, we use each module's specification as the environment of the other and avoid circularity.

3.3 Environment Arrays

Many finite state systems are composed of multiple processes running in parallel. An obvious strategy is to check each of the local properties using only the part of the system that it describes. This means that we divide the system and its specification

into partitions, and sub-specifications (properties) of the partitions and verify the properties of the partitions separately.

Assume $\mathcal{P}_1$ and $\mathcal{P}_2$ are two partitions of a system. According to the assume guarantee rules, we would like to reason according to the following schema:

$$\frac{\begin{array}{c}\mathcal{P}_2 \models \psi_{\mathcal{P}_2} \\ \mathcal{P}_1 \parallel \psi_{\mathcal{P}_2} \models \varphi_{\mathcal{P}_1}\end{array}}{\mathcal{P}_1 \parallel \mathcal{P}_2 \models \varphi_{\mathcal{P}_1}}$$

The environment of partition $\mathcal{P}_1$ is expressed by an assumption that process $\mathcal{P}_1$ makes on process $\mathcal{P}_2$ using a temporal formula $\psi_{\mathcal{P}_2}$. This assumption is used when checking that $\mathcal{P}_1$ satisfies its specification $\varphi_{\mathcal{P}_1}$, and must, of course, be discharged relative to $\mathcal{P}_2$.

A problem is involved in the above reasoning because the above reasoning rule is not sound in a general system. For example, in the above reasoning rule, we prove partition $\mathcal{P}_1$ using partition $\mathcal{P}_2$ as the assumptions. However, it is often the case that we may prove partition $\mathcal{P}_2$ using partition $\mathcal{P}_1$ as the assumptions. This obvious circular reasoning will make the verification fail. In this case, either we cannot separate partition $\mathcal{P}_1$ and $\mathcal{P}_2$, namely if there are circular interface signals between two partitions, then we have to put $\mathcal{P}_1$ and $\mathcal{P}_2$ as one entity in the verification, or, we have to put some constrains on the reasoning rules to handle the circular cases. In this chapter, we will present how we separate the system, and make the compositional verification.

3.3.1 System Partitions

In this section, we develop a model for the system under verification. The global picture of the system is shown in Figure 3.3. We will define the structure of the system under verification in Definition 3.3.1.

Definition 3.3.1 *Let Ω be a system composed of n partitions $\mathcal{P}_i$ ($i \in [1 : n]$). An abstract environment array E_Ω is an $(n+1) \times (n+1)$ array defined as follows,*

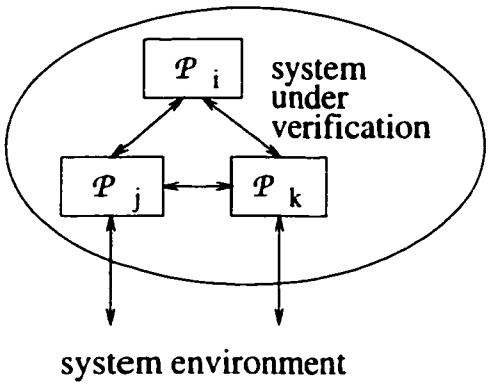


Figure 3.3: System under Verification and System Environment

$$E_\Omega = \begin{pmatrix} \emptyset & E_{e1} & E_{e2} & \dots & E_{en} \\ E_{1e} & E_{11} & E_{12} & \dots & E_{1n} \\ E_{2e} & E_{21} & E_{22} & \dots & E_{2n} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ E_{(n-1)e} & E_{(n-1),1} & E_{(n-1),2} & \dots & E_{(n-1),n} \\ E_{ne} & E_{n1} & E_{n2} & \dots & E_{nn} \end{pmatrix}$$

where E_{ij} $(i, j \in [1 : n])$ *stands for a specification of all interface behaviors from partition* $\mathcal{P}_i$ *to* $\mathcal{P}_j$. E_{ei} *stands for the interface behaviors from the system environment to* $\mathcal{P}_i$. E_{ie} *stands for the interface behaviors from* $\mathcal{P}_i$ *to the system environment. If there are no signals from partition* $\mathcal{P}_i$ *to partition* $\mathcal{P}_j$, E_{ij} *is denoted with* $\emptyset$.

E_{ij} is a set of **ACTL** formulas which capture the behaviors of the input signals of partition $\mathcal{P}_j$, and these signals are from partition $\mathcal{P}_i$. E_{ei} and E_{ie} are sets of **ACTL** formulas which capture the behaviors of the input/output signals of partition $\mathcal{P}_i$, and these signals are from/to the system environment.

Definition 3.3.2 *The inputs of partition* $\mathcal{P}_i$, E_i^I, *is a set of* **ACTL** *formulas which equals to the union of all the entries of one column in the* i^{th} *column in the abstract environment array, namely* $E_{ei} \cup E_{1i} \cup E_{2i} \cup \ldots \cup E_{ni}$.

Definition 3.3.3 *The outputs of partition $\mathcal{P}_i$, E_i^O, is a set of* **ACTL** *formulas which equals to the union of the entries in column 2 to column $n+1$ of i^{th} row in the abstract environment array, namely $E_{i1} \cup E_{i2} \cup \ldots \cup E_{in}$.*

A natural partition of the system is that one module is a partition. In this sense, the compositional verification becomes modular verification where each module is verified separately.

As we know, we have two solutions to tackle the circular reasoning. One is to improve the system partition method; the other is to improve the reasoning rules. In the first solution regard, we require that there are no circular signals between two partitions, which we call it linear system.

Definition 3.3.4 *A system Ω is* linear *if in its abstract environment array, either $E_{ij} = \emptyset$ or $E_{ji} = \emptyset$ for all $i < j$, where $i, j \in [1 : n]$, namely the entries below or above the main diagonal are all $\emptyset$ except E_{ie} and E_{ei}.*

A linear system means that all the interface signals are in the same direction with respect to the index order of the partitions. In such a system, if there exist signals from E_i to E_j, then no signal exists from E_j to E_i via direct or indirect connections.

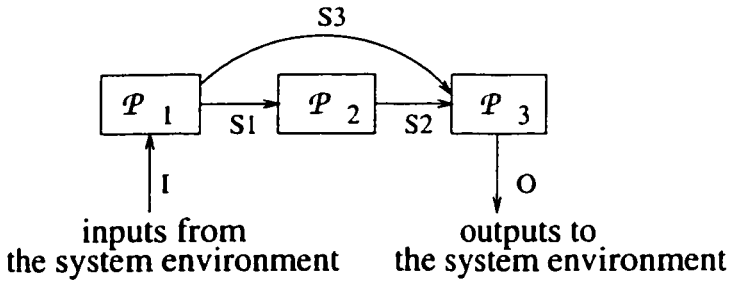


Figure 3.4: A Linear System

Example 3.3.1 *The system Ω shown in Figure 3.4 contains three modules M_1, M_2, and M_3. Each module is corresponding to a partition. Namely M_1 to $\mathcal{P}_1$, M_2 to $\mathcal{P}_2$,*

and M_3 to $\mathcal{P}_3$. This system is a linear system because all the entries E_{ij}, $i, j \in [1:3]$ and $i > j$, equal to $\emptyset$. The corresponding abstract environment array is:

$$E_\Omega = \begin{pmatrix} \emptyset & E_{e1} & \emptyset & \emptyset \\ \emptyset & \emptyset & E_{12} & E_{13} \\ \emptyset & \emptyset & \emptyset & E_{23} \\ E_{3e} & \emptyset & \emptyset & \emptyset \end{pmatrix}$$

where, E_{12}, E_{23}, and E_{13} are the specifications of the interface signals S_1, S_2, and S_3, respectively. E_{e1} and E_{3e} are the specifications of of the input/output signals I and O, respectively.

Usually, when we make a one-to-one mapping between the modules and the partitions in the system, the requirement in Definition 3.3.4 often cannot be satisfied. For example, in a hand-shaking protocol, there always exists a pair of signals, which are circular between two modules, namely, signals "req" from module i to module j and signals "ack" from module j to module i, or vice versa. In this case, both E_{ij} and E_{ji} are not empty which violates Definition 3.3.4.

However, we can always find a mapping between the partitions and the modules in the system so that the abstract environment array satisfies Definition 3.3.4. For example, if a system, which is similar to that in Example 3.3.1, contains five modules where there are circular signals among M_2, M_4, and M_5. Partition $\mathcal{P}_2$ may actually be a set of modules as shown in Figure 3.5. In this case, the mapping between the partitions and the modules in the system becomes:

- M_1 is mapped to $\mathcal{P}_1$.
- M_2, M_4, and M_5 are mapped to $\mathcal{P}_2$.
- M_3 is mapped to $\mathcal{P}_3$.

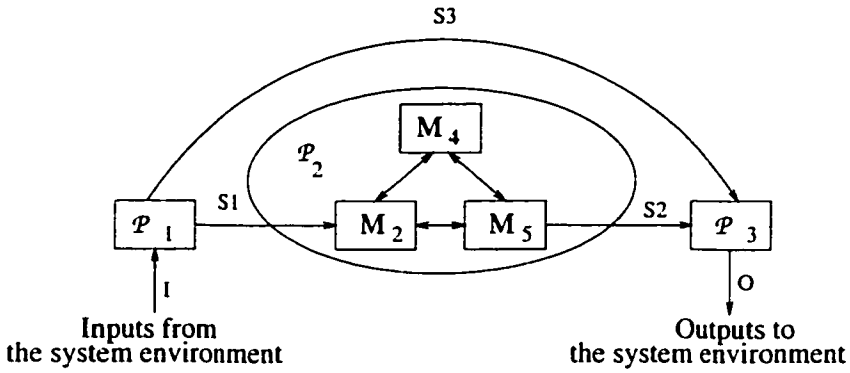


Figure 3.5: New Partitions

Thus, partitions $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$ are linear, so at least we can consider the system as a linear system at the partition level. However, regarding the partition with circular connected modules, we have to develop a circular reasoning approach to tackle this problem.

3.3.2 The Circular Reasoning Rule

In practice, what we want to verify is that the system has proper response with respect to the stimulus of the system environment, namely partition $\mathcal{P}_i$ has expected response E_{ie} if the stimulus of $\mathcal{P}_i$ given by the system environment is E_{ei}. Moreover, via tableau construction, E_{ij} can be transformed into a finite state machine (for detail, see next chapter). In this way, $\mathcal{P}_i$ which is a finite state machine, will be in the same context as that of E_{ij}, so we can use "$\|$" to compose E_{ij} and $\mathcal{P}_i$.

In a linear system, the compositional verification can be done naturally (Theorem 3.3.1).

Theorem 3.3.1 *Let Ω be a linear system composed of n partitions $\mathcal{P}_i$ ($i \in [1:n]$), i.e., $\Omega = \|_{i=1}^{n} \mathcal{P}_i$; if*

1. $\forall j \in [1:n-1], (\mathcal{P}_j \| E_j^I) \Rightarrow E_j^O$;

2. $\mathcal{P}_n \| E_n^I \Rightarrow E_{ne}$;

then, $\Omega \Rightarrow E_{ne}$ *under the system environment stimulus* E_{ei}*, where* $\Rightarrow$ *can be any logic implications.*

PROOF:

Since Ω is a linear system, the entries in the corresponding abstract environment array below or above the main diagonal are all $\emptyset$ except E_{ie} and E_{ei}.

Case 1: If the entries above the main diagonal are all $\emptyset$ except E_{ie} and E_{ei}.
In this case, partition $\mathcal{P}_n$ has no inputs except the inputs from the system environment E_{en}, namely $E_n^I = E_{en}$. Because E_{en} does not need additional assumptions, according to the assume-guarantee reasoning rule and the second condition in the theorem, $\Omega \Rightarrow E_{ne}$ under the stimulus E_{en}.

Case 2: If the entries below the main diagonal are all $\emptyset$ except E_{ie} and E_{ei}.
In this case, partition $\mathcal{P}_1$ has no inputs except the inputs from the system environment E_{e1}, namely $E_1^I = E_{e1}$. The only inputs of $\mathcal{P}_2$ are from $\mathcal{P}_1$ and the system environment, so, according to the assume-guarantee reasoning rule:

$$\frac{E_1^I \| \mathcal{P}_1 \Rightarrow E_1^O \qquad E_2^I \| \mathcal{P}_2 \| E_1^O \Rightarrow E_2^O}{E_2^I \| \mathcal{P}_1 \| \mathcal{P}_2 \Rightarrow E_2^O}$$

Since the only inputs of $\mathcal{P}_3$ are from $\mathcal{P}_1$, $\mathcal{P}_2$, and the system environment, we can prove $E_3^I \| \mathcal{P}_1 \| \mathcal{P}_2 \| \mathcal{P}_3 \Rightarrow E_3^O$. Keep using the similar reason rules, eventually we can prove $E_{n-1}^I \| \mathcal{P}_1 \| \mathcal{P}_2 \| \ldots \mathcal{P}_{n-1} \Rightarrow E_{n-1}^O$. Together with the second condition in the theorem, we can conclude that $\Omega \Rightarrow E_{ne}$ under the system environment stimulus E_{en}. □

According to this theorem, in a linear system, if each partition in the system composed with its environments has correct outputs, then the whole system can guarantee the correct behaviors.

However, as we know it is not always possible that the system under verification is a linear system. Although we can make coarse partitions instead of the module-to-partition one-to-one partitions, in the worst case, there is only one partition in the system, namely the system itself. In this case, the compositional verification of linear systems makes no sense. We therefore have to improve the reasoning rules to handle this situation.

In this case, we use instead induction over time [64] and propose the Theorem 3.3.2 to verify a non-linear system [75, 74]. Since in this case, module M_i has always one-to-one correspondence with partition $\mathcal{P}_i$, we need not distinguish M_i and $\mathcal{P}_i$ any more.

Theorem 3.3.2 *Let Ω be a non-linear system composed of n modules M_i ($i \in [1 : n]$), i.e., $\Omega = \|_{i=1}^{n} M_i$; if*

1. $\forall j \in [1 : n], M_j \uparrow^0 \Rightarrow (E_j^O \| E_{ne}) \uparrow^0$;
2. $\forall j \in [1 : n], (M_j \| E_j^I) \uparrow^{t-1} \Rightarrow E_j^O \uparrow^t$;
3. $(M_n \| E_n^I) \uparrow^t \Rightarrow E_{ne} \uparrow^t$;

then, $\forall_t.(\Omega \Rightarrow E_{ne})$ under the system environment stimulus E_{ei}.

Here, $p \uparrow^\tau$ stands for $\forall(t \leq \tau).p$ or "p holds up to time $t = \tau$".

PROOF:

1. According to the condition (1), every module has correct outputs at time 0, namely the reset values of the modules are the expected values and E_{ne} also has the expected initial values at time 0.
2. According to the condition (2), every module which has correct inputs at time 0 will have expected outputs E_j^O at time 1. At time 1, since the environment E_n^I of module M_n is ready, module M_n has expected E_{ne} at time 1 (the condition (3)).

3. Again, according to the condition (2), since the environments of each module are ready at time 1, those modules will have expected outputs E_j^O at time 2 and module M_n has expected E_{ne} at time 2 as well.
4. Keep using the induction over time, we can know that for anytime from now till future, the system has expected outputs E_{ne}.

□

In this theorem, because there are feedback signals between modules in the system, we request each module must have at least one unit time delay between the its inputs and its circular feedback signals so that we can reason as following. Given correct inputs at time $t-1$, each module in the system has correct outputs at time t and the system is correct at time 0, we can conclude that the system guarantee the correct behaviors.

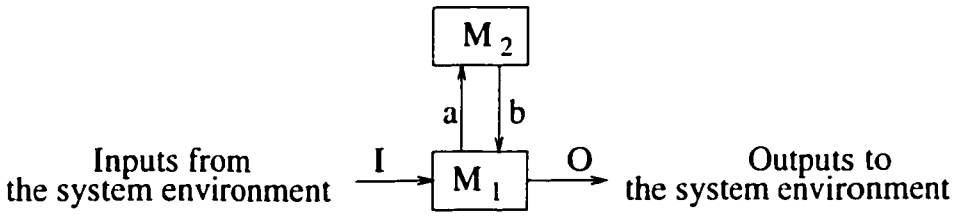


Figure 3.6: A Circular System

Example 3.3.2 *The system Ω shown in Figure 3.6 is a synchronized circuit model which contains two modules M_1 and M_2. There exist circular feedback signals between them, i.e. a and b. I and O are the input/output from/to the system environment. We would like to verify that Ω has an expected output O if the environment stimulus is I. Ω obviously is not a linear system. We have to use Theorem 3.3.2 to verify it.*

The verification is illustrated in Figure 3.7 as follows. At time 0, we check all the reset status to see if the output $O\uparrow^0$ is satisfied.

If module M_1 can guarantee $a\uparrow^1$ under the environment of $I\uparrow^0$ and $b\uparrow^0$, and M_2 can guarantee $b\uparrow^1$ under the environment of $a\uparrow^0$, we can conclude that $O\uparrow^1$ is

satisfied.

If module M_1 can guarantee $a\uparrow^2$ under the environment of $I\uparrow^1$ and $b\uparrow^1$, and M_2 can guarantee $b\uparrow^2$ under the environment of $a\uparrow^1$, we can conclude that $O\uparrow^2$ is satisfied.

So on and so forth, we can conclude that for anytime the output is expected.

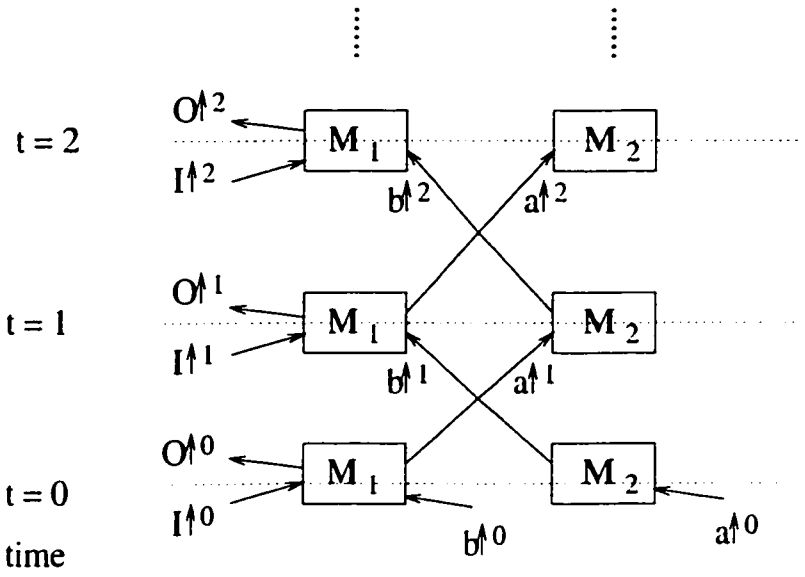


Figure 3.7: Verification of the Circular System

Actually, in practice, it is the most significant portion of the overall design cycle: starting with the definition of high-level specifications during design capture; moving on to verify those specifications via verification; and continuing by partitioning a design, decomposing it into functional blocks and smaller subsystems, and verifying each of them. Further partitioning is carried out until the descriptive level is suitable for synthesis. Some design management tools, such as SUMMIT [82], can help us achieve this. Using this tool, engineers can capture the design graphically in a hierarchical way and provide synthesis inputs partition by partition.

3.4 Summary

In this chapter, we reviewed the assume-guarantee compositional reasoning approach, as reported in the literature, and introduced the notion of environment array to capture the reasoning rules. Next, let's consider the following classical assume-guarantee reasoning [36] (where $M_1 \| M_2$ means the parallel composition of module M_1 and M_2; $M_1 \models \varphi_{M_1}$ means that the module M_1 satisfies the **ACTL** specification φ_{M_1}; $(\varphi_{M_1}, M_2) \models \varphi$ means model M_2 satisfies formula φ under the environment described by φ_{M_1}):

$$\frac{\begin{array}{c} M_1 \models \varphi_{M_1} \\ (\varphi_{M_1}, M_2) \models \varphi \end{array}}{M_1 \parallel M_2 \models \varphi}$$

We propose to replace (φ_{M_1}, M_2) by the composition of the synthesized Verilog module of the tableau of φ_{M_1} and module M_2, and the composed system then can be fed into a model checker like VIS.

Since the compositional verification of full **CTL** is an NP-hard problem [57], a widely accepted way to automate the assume-guarantee compositional verification had been proposed by [36] using a *simulation preorder* and **ACTL** whose semantics is consistent with the simulation preorder. The simulation preorder on finite state machines captures the notion of *more behaviors* and preserves satisfaction of **ACTL** formulas. Based on the close relation between the satisfaction and the preorder, we can verify $M \models \varphi$ by checking the relation $M \preceq \varphi$. In practice, we use classical model checking for verifying $M \models \varphi$ if φ is given by a formula, and $M \preceq \varphi$ if φ is a tableau. Consequently, the efficiency of checking the preorder depends on the size of the tableau.

Based on these compositional verification approaches, we discussed how to make the compositional verification using the abstract environment array. There are two ways to do the verification. One is to partition the system under verification into a linear system so that we soundly verify the system according to the partitions

step by step. However, if the partition is still too large for verification, and we cannot divide this partition further more because of some circular feedback signals, we can turn to the induction over time reasoning to verify this partition. Using the induction, we verify the reset status of the model and the induction steps.

A problem we intentionally avoid in this chapter is that the system under verification is a finite state machine, but the entries in the abstract environment array are a set of formulas. When we use the "||" or "⇒" operators, we have to guarantee that the operands must be in the same context. We will discuss this problem in the next chapter.

Chapter 4

Reduced Tableau Construction

4.1 Introduction

The complexities for different assume-guarantee reasoning styles with **LTL** (Linear Time Logic) or **CTL** (Computational Tree Logic) as the environment assumptions are explored in [53] [84]. Generally speaking, using **LTL** and **CTL** as assumptions are both computation hard. The selection is a trade-off depending on the applications. In our approach, we use **ACTL** (A-Computational Tree Logic), a subset of **CTL**, to capture the assumptions because (1) the environment assumptions can be used as guarantees in the next verification stage so that hierarchical verification is possible, since both assumptions and guarantees are in **ACTL**, (2) the expressiveness of **ACTL** is different from **LTL**, as in the case of the well known expression **AFAG***p*. The theoretical analysis described in [53] and [84] shows that the complexity of using **ACTL** or **LTL** as the assumptions in the compositional verification is PSPACE-complete and EXPSPACE-complete, respectively. The tableau size is a key factor affecting the verification efficiency. This motivated us to look into the construction of tableau in an optimized way.

Another practical problem we have in the compositional verification of the previous chapter is that the system under verification is a finite state machine, but

the entries in the abstract environment array are a set of **ACTL** formulas. When we use "||" to compose M with E_{ij} (Chapter 3), we have to guarantee that the operands must be in the same format, namely E_{ij} has to be in the form of transition structures.

Here, in this chapter, we will show how to transform **ACTL** formulas into transition structures using reduced tableau construction techniques.

The tableau construction aims (if possible) to construct the model of a given formula φ. Namely, given a formula φ, the tableau construction of φ builds a *Kripke structure* $\mathcal{K}$ consisting of states labelled by atomic propositions derived from φ and transitions between states, such that every model of φ is represented as an infinite path in $\mathcal{K}$.

Consider the formula **AF***p*, i.e., for all computation paths, p eventually holds. Intuitively, the tableau is going to represent all those behaviors that are consistent with the formula. Thus, a first guess of the tableau might be the structure shown in Figure 4.1 where the initial states are represented as double circles and each state is labelled with its label (first line), and the generalized Büchi fair sets to which it belongs (second line). The idea is that starting from one initial state, we should eventually reach a state having p true. The transitions from there are completely unconstrained.

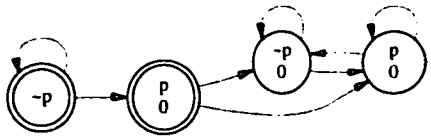


Figure 4.1: Proposed Tableau for **AF***p*

As is often the case with tableaus for temporal logics, e.g., [36, 60], a state of the tableau consists of a set of formulas that are supposed to hold along all paths leaving the state. Therefore we propose to define a reduced tableau of **ACTL**

formulas consisting of less states and transitions but accepting precisely the models of the formulas. Here, the formulas in the states are interpreted over a *three-valued domain.* Thus, a state may include a formula or its negation, or none of them. If the latter occurs, it reflects *a don't care* situation, and we call this state a *dummy state.*

Consider the previous example where we collapsed two states to get one state with a don't care value as shown in Figure 4.2. Thus, we have a reduced tableau. Once we have the reduced tableau, we then can synthesize it into Verilog code [1].

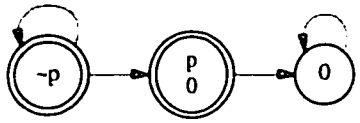


Figure 4.2: Reduced Tableau for **AF***p*

Next, we give a tableau construction for **ACTL** formulas (for a similar construction for **LTL** see [60]). We have proved that the tableau of an **ACTL** formula is a maximal model for the formula under the simulation relation ($\preceq$). Thus, the structure of the tableau generated by the construction can be used as an assumption, by composing the structure with a desired system before applying model checking. Discharging the assumption is simply done by either model checking or by $\preceq$ relation.

4.2 ACTL in Three-Valued Domain

ACTL formulas here are interpreted over fair Kripke structures called *tableau* with fairness constraints over the three value domains. Below, we give the semantics of **ACTL** formulas relative to a Kripke structure $\mathcal{K}$ (Definition 2.2.2). In addition, we use π to denote a sequence of states $s_0 s_1 \ldots$ and use π^i to denote the i-th state of π (counting from 0). A Kripke state will be interpreted over three-valued domain,

[1] A subset of Verilog HDL acceptable by VIS [11]

i.e., a formula, the negation of the formula, and a *don't care value* $\emptyset$. Consequently, we will adapt the definition of the satisfaction relation $\models$ defined as follows:

Definition 4.2.1 *The relation $s \models \varphi$ (read: φ* holds at *s) is inductively defined as follows:*

- $s \models p$ *if and only if* $p \in \lambda(s)$ *or* $\lambda(s) = \emptyset$
- $s \models \neg p$ *if and only if* $\neg p \in \lambda(s)$ *or* $\lambda(s) = \emptyset$
- $s \models \varphi \wedge \psi$ *if and only if* $s \models \varphi$ *and* $s \models \psi$.
- $s \models \varphi \vee \psi$ *if and only if* $s \models \varphi$ *or* $s \models \psi$.
- $s \models \mathbf{AX}\varphi$ *if and only if for every fair path* π *starting at the state* s, $\pi^1 \models \varphi$.
- $s \models \mathbf{A}(\varphi\ \mathbf{U}\ \psi)$ *if and only if for every fair path* π *starting at the state* s, *there exists* $i \geq 0$ *such that* $\pi^i \models \psi$ *and* $\pi^j \models \varphi$ *holds for* $j < i$.
- $s \models \mathbf{A}(\varphi\ \mathbf{V}\ \psi)$ *if and only if for every path* π *starting at the state* s, *and every* $j \geq 0$, *if for every* $0 \leq i < j, \pi^i \not\models \varphi$ *then* $\pi^j \models \psi$.

We say that $\mathcal{K}$ satisfies a formula φ, written $\mathcal{K} \models \varphi$ if for every $s \in I$, $s \models \varphi$.

Additional temporal operators are introduced as abbreviations: $\mathbf{AG}\varphi$ is defined as $\mathbf{A}(\mathbf{false}\mathbf{V}\varphi)$ and $\mathbf{AF}\varphi$ is defined as $\mathbf{A}(\mathbf{true}\mathbf{U}\varphi)$.

Definition 4.2.2 *Let $\mathcal{K} = \{S, I, \delta, \lambda, \mathcal{F}\}$ and $\mathcal{K}' = \{S', I', \delta', \lambda', \mathcal{F}'\}$ be two structures. The composition of $\mathcal{K}$ and $\mathcal{K}'$, denoted $\mathcal{K} \| \mathcal{K}'$, is the structure $\mathcal{K}'' = \{S'', I'', \delta'', \lambda'', \mathcal{F}''\}$ given as follows*

- *S'' is the set of pairs $(s, s') \in S \times S'$ for which $\lambda(s, p) = \lambda'(s', p')$ with respect to all p and p' in $AP \cup AP'$ or one of $\lambda(s)$ and $\lambda(s')$ is $\emptyset$.*
- *I'' is $(I \times I') \cap S''$.*

- $\delta''((s_i, s'_i), (s_j, s'_j))$ *if and only if* $\delta(s_i, s_j)$ *and* $\delta'(s'_i, s'_j)$.
- $\lambda''((s, s'), p) = \lambda(s, p)$ *for all* p *in* AP. $\lambda''((s, s'), p') = \lambda'(s', p')$ *for all* p' *in* AP'. $\lambda''(s, s') = \emptyset$ *if* $\lambda(s)$ *or* $\lambda'(s')$ *is* $\emptyset$.
- $\mathcal{F}''$ *is* $\{((f \times S') \cap S'') | f \in \mathcal{F}\} \cup \{((f' \times S) \cap S'') | f' \in \mathcal{F}'\}$.

Finally, we will adapt the simulation relation to the three-value domain.

Definition 4.2.3 *Let* $\mathcal{K}$ *and* $\mathcal{K}'$ *be two fair Kripke structures over the same set of atomic propositions* AP. *A relation* $\mathcal{R} \subseteq S \times S'$ *is a simulation relation from* $\mathcal{K}$ *to* $\mathcal{K}'$ *if and only if the following conditions hold for all* $(s, s') \in \mathcal{R}$*:*

- $\lambda(s) = \lambda(s')$ *where eventually* $\lambda(s') = \emptyset$.
- *for every initial state* $s_0 \in I$, *there is an initial state* $s'_0 \in I'$ *such that* $(s_0, s'_0) \in \mathcal{R}$.
- *for all* $(s, s') \in \mathcal{R}$, $\lambda(s) = \lambda'(s')$ *or* $\lambda'(s') = \emptyset$ *and*

$$\forall t[(s, t) \in \delta \implies \exists t'[(s', t') \in \delta' \wedge (t, t') \in \mathcal{R}]$$

We say that $\mathcal{K}'$ simulates $\mathcal{K}$ *(denoted by* $\mathcal{K} \preceq \mathcal{K}'$*) if there exists a simulation relation* $\mathcal{R}$ *from* $\mathcal{K}$ *to* $\mathcal{K}'$.

Moreover, if there is a simulation preorder from $\mathcal{K}$ to $\mathcal{K}'$, then $\mathcal{K} \| \mathcal{K}'' \preceq \mathcal{K}' \| \mathcal{K}''$ and $\mathcal{K} \preceq \mathcal{K} \| \mathcal{K}$. Finally, it is well known that if $\mathcal{K} \preceq \mathcal{K}'$ then for every **ACTL** formula φ, $\mathcal{K}' \models \varphi$ implies $\mathcal{K} \models \varphi$.

4.3 Rewriting Formulas

Rewriting is the first step towards an efficient tableau construction. It is considered an easy and effective way to minimize the tableau [32, 81]. Rewriting consists of a set of rules applied recursively to an **ACTL** formula in positive normal form (i.e., the

negation is applied only on atomic propositions), reducing the number of temporal operators and/or connectives. Although there are no dedicated **ACTL** rewriting rules available in the open literature, we can still derive the rules based on well-known identities. One possible way is to deploy similar **LTL** rewriting rules, which can be found, e.g., in [60]. However, we cannot refer to the **LTL** rewriting rules by simply padding the path quantifier by for-all-path quantifier **A**. For example, in **ACTL**, $\mathbf{AGAF}p \vee \mathbf{AGAF}q$ is different from $\mathbf{AGAF}(p \vee q)$, while this is not the case in **LTL**. The application of a rewrite rule consists of replacing in the (sub)formula the left hand side with the right hand side which has in principle less temporal operators and/or connectives.

A set of rewriting rules that we have proved sound and implemented in our tool is given below. Our implementation also provides the ability to renew or add new rewriting rules in its internal rewriting database.

Sample Rewriting Rules

$\mathbf{AGAG}\ p \equiv \mathbf{AG}\ p$
$\mathbf{AFAF}\ p \equiv \mathbf{AF}\ p$
$p\ \mathbf{AU}\ (p\ \mathbf{AU}\ q) \equiv (p\ \mathbf{AU}\ q)$
$p\ \mathbf{AV}\ (p\ \mathbf{AV}\ q) \equiv (p\ \mathbf{AV}\ q)$
$(p\ \mathbf{AU}\ q)\ \mathbf{AU}\ q \equiv (p\ \mathbf{AU}\ q)$
$\mathbf{AX}\ p \wedge \mathbf{AX}\ q \equiv \mathbf{AX}\ (p \wedge q)$
$\mathbf{AG}\ p \wedge \mathbf{AG}\ q \equiv \mathbf{AG}\ (p \wedge q)$
$(p\ \mathbf{AU}\ r) \wedge (q\ \mathbf{AU}\ r) \equiv (p \wedge q)\ \mathbf{AU}\ r$

The application of the rewriting rules can reduce the size of the formula if only the presentation form the formula is not the positive normal form. In some cases when the formula is written by a non-experienced person, the rewriting rules are especially useful since there may be lots of redundant information in the original formula.

4.4 Tableau Construction Procedure

The translation of **ACTL** formulas into tableaus is accomplished by the application of *tableau rules*, which decompose a formula φ into *particles*. A particle of a formula φ is a set of formulas which represents the information about what needs to hold in the current and the next states. It identifies a (possible) state of a Kripke structure. The atomic propositions in the set define the *label* of the state and the formulas starting with **AX** operator, determine the transitions out of the state as well as the acceptance conditions. The expansion process is applied to the **AX** part of each state until no new obligations are produced. The *initial states* are identified with particles that are derived immediately from the original formula.

The process of expansion does not guarantee that *eventuality* is fulfilled. This happens because it is possible to propagate forever the fulfilling of **U**-formulas. For example, the formula $\varphi \equiv \mathbf{A}\ (\mu \mathbf{U} \nu)$ can be fulfilled by fulfilling μ and by promising to fulfill φ later by fulfilling $\mathbf{AXA}\ (\mu \mathbf{U} \nu)$. Consequently, an additional condition is necessary to identify those paths along which φ holds. In order to identify these paths, we define generalized Büchi acceptance conditions, namely a set of states which will be visited infinitely often.

Next, we will introduce some useful definitions for the algorithm, and one example is used through out the illustration.

Definition 4.4.1 *The set of sub-formulas of* φ, *denoted by* SUB (φ), *is defined inductively as follows:*

- *If* $\varphi = \mathbf{true}$, *then* SUB $(\mathbf{true})$ *is* $\{\emptyset\}$.
- *If* $\varphi = p$ *or* $\varphi = \neg p$, *where* $p \in AP$, *then* SUB $(p) = \{p\}$ *and* SUB $(\neg p) = \{\neg p\}$.
- *If* $\varphi = \varphi_1 \wedge \varphi_2$ *or* $\varphi = \varphi_1 \vee \varphi_2$ *then* SUB $(\varphi) = \{\varphi\} \cup$ SUB $(\varphi_1) \cup$ SUB (φ_2).
- *If* $\varphi = \mathbf{AX}\varphi_1$ *then* SUB $(\varphi) = \{\ \mathbf{AX}\varphi_1\} \cup$ SUB (φ_1).

- *If* $\varphi = \mathbf{A}(\varphi_1 \mathbf{U} \varphi_2)$ *then* SUB $(\varphi) = \{\mathbf{A}(\varphi_1 \mathbf{U} \varphi_2), \mathbf{AXA}(\varphi_1 \mathbf{U} \varphi_2)\} \cup$ SUB $(\varphi_1) \cup$ SUB (φ_2).

- *If* $\varphi = \mathbf{A}(\varphi_1 \mathbf{V} \varphi_2)$ *then* SUB $(\varphi) = \{\mathbf{A}(\varphi_1 \mathbf{V} \varphi_2), \mathbf{AXA}(\varphi_1 \mathbf{V} \varphi_2)\} \cup$ SUB $(\varphi_1) \cup$ SUB (φ_2).

We classify some of the temporal formulas into α-formula and β-formula. A formula is an α formula if it can be viewed as a conjunctive formula $\alpha = \alpha_1 \wedge \alpha_2$; A formula is a β formula if it can be viewed as a disconjunctive formula $\beta = \beta_1 \vee \beta_2$. If a formula $\psi \in$SUB (φ) is an α-formula, we define $k(\psi) = \{\psi_1, \psi_2\}$ where $k(\psi)$ is the set of the conjunctive part of this formula and $\psi_1 = \alpha_1$, $\psi_2 = \alpha_2$. If a formula $\psi \in$SUB (φ) is a β-formula, we define $k_1(\psi)$ and $k_2(\psi)$, where $k_1(\psi)$ and $k_2(\psi)$ are the left and right operands of the disconjunction, as follows:

1. if $\psi = \psi_1 \vee \psi_2$, then $k_1(\psi) = \{\psi_1\}$ and $k_2(\psi) = \{\psi_2\}$.

2. if $\psi = \mathbf{A}(\psi_1 \mathbf{U} \psi_2)$, then $k_1(\psi) = \{\psi_2\}$ and $k_2(\psi) = \{\psi_1, \mathbf{AXA}(\psi_1 \mathbf{U} \psi_2)\}$

3. if $\psi = \mathbf{A}(\psi_1 \mathbf{V} \psi_2)$, then $k_1(\psi) = \{\psi_1, \psi_2\}$ and $k_2(\psi) = \{\psi_2, \mathbf{AXA}(\psi_1 \mathbf{V} \psi_2)\}$

The intended meaning of this classification is semantics. An α-formula φ, which is a conjunction, holds at the tableau state s if and only if α_1 and α_2, denoted by $k(\varphi)$, also hold at s. A β-formula φ, which is a disjunction, holds at the tableau state s if and only if either $k_1(\varphi)$ or $k_2(\varphi)$ hold at s. A formula that involves no modalities or has main connective **AX** is both α and β and is called an *elementary* formula. All other formulas are *non-elementary* formulas.

Definition 4.4.2 *A set of formulas P is a* particle *over φ if:*

1. $P \subseteq$SUB (φ).

2. *If* $p_1 \wedge p_2 =$ **false**, *then* $\{p_1, p_2\} \not\subseteq P$.

3. *For every α-formula $\psi \in SUB(\varphi)$, $k(\psi) \subseteq P$ holds.*

4. For every β-formula $\psi \in$SUB (φ), we distinguish two cases:

(a) If $\psi = \psi_1 \vee \psi_2$ then either $k_1(\psi) \subseteq P$ or $k_2(\psi) \subseteq P$.

(b) If $\psi = \mathbf{A}(\psi_1 \mathbf{U} \psi_2)$ or $\psi = \mathbf{A}(\psi_1 \mathbf{V} \psi_2)$, then $\psi \in P$ if and only if either $k_1(\psi) \subseteq P$ or $k_2(\psi) \subseteq P$.

In the calculation of the particles, since we decompose α or β formulas till no obligation is generated, namely there is at least one elementary formula in each particle. Either it is the formula without modalities, or it is the formula beginning with **AX**. The formulas without modalities need to hold in the current state, and the formulas beginning with **AX** modality illustrate what needs to hold in the next state.

Example 4.4.1 *P_1 in the following is a particle of the formula φ :* $\mathbf{A}(\mathbf{true}\mathbf{U}p) \vee \mathbf{A}(\mathbf{false}\mathbf{V}\neg p)$. *$P_1$ actually illustrates one possibility to satisfy φ, namely the second part of the φ, φ_1 :* $\mathbf{A}(\mathbf{false}\mathbf{V}\neg p)$ *is true. φ_1 is true if and only if p is not true at the current state and φ_1 is true at the next state along any paths.*

$P_1 \quad = \quad \{\mathbf{A}(\mathbf{false}\mathbf{V}\neg p), \mathbf{AXA}(\mathbf{false}\mathbf{V}\neg p), \neg p\}$

Till now, we have obtained all the information in a formula. Therefore, we can define a *cover* of a formula meaning that it will cover all the behaviors of a formula either in the current or in the next state. A cover is a set of particles. Each particle is considered as a state label in the corresponding Kripke structure of the formula. For example, COVER $(p) = \{\{p\}\}$, where p is a propositional formula in the atomic propositions (APs), meaning that in order to satisfy formula p, the current state label (particle) has to be $\{p\}$.

During the formula decomposition, we merge two covers by the union operator "$\cup$". As a matter of fact, it is not always the case that the union on two particles makes sense because they are not compatible[2], for example, $\{p\} \cup \{\neg p\}$ has no

[2]Two particles ρ and τ are compatible if $\forall_{p \in \rho, q \in \tau} p \wedge q \neq \mathbf{false}$

semantic meaning since a state satisfying both p and $\neg p$ does not exist. In this case, we use a "JOIN" operator to exclude this case. Namely,

$$S \text{ JOIN } T = \{\rho \cup \tau : \rho \in S, \tau \in T, \rho \text{ and } \tau \text{ are compatible}\}$$

for any two elements of a particle ρ and τ in S and T.

Definition 4.4.3 *A cover of the formula φ, denoted by* COVER (φ), *returns a set of particles and is defined as follows:*

- COVER $(p) = \{\{p\}\}$.
- COVER $(\neg p) = \{\{\neg p\}\}$.
- COVER $(\mu \wedge \nu)$ =COVER (μ) JOIN COVER (ν).
- COVER $(\mu \vee \nu)$ =COVER $(\mu) \cup$ COVER (ν).
- COVER $(\mathbf{AX}\mu) = \{\{\mathbf{AX}\mu\}\}$
- COVER $(\mathbf{A}(\mu \mathbf{U} \nu))$ = [{{$\mathbf{A}(\mu \mathbf{U} \nu)$}} JOIN COVER (φ)] $\cup$ [{{$\mathbf{A}(\mu \mathbf{U} \nu)$}} JOIN COVER (ψ)], *where φ and ψ are formulas in $k_1(\mathbf{A}(\mu \mathbf{U} \nu))$ and $k_2(\mathbf{A}(\mu \mathbf{U} \nu))$, respectively.*
- COVER $(\mathbf{A}(\mu \mathbf{V} \nu))$ = [{{$\mathbf{A}(\mu \mathbf{V} \nu)$}} JOIN COVER (φ)] $\cup$ [{{$\mathbf{A}(\mu \mathbf{V} \nu)$}} JOIN COVER (ψ)], *where φ and ψ are formulas in $k_1(\mathbf{A}(\mu \mathbf{V} \nu))$ and $k_2(\mathbf{A}(\mu \mathbf{V} \nu))$, respectively.*

The generalization of a cover of set A of formulas is the Join of each cover of the formula, i.e., COVER (A) =JOIN$_{\varphi \in A}$COVER (φ)

Example 4.4.2 *The cover of the formula φ : $\mathbf{A}(\mathbf{true}\mathbf{U}p) \vee \mathbf{A}(\mathbf{false}\mathbf{V}\neg p)$ are shown as follows:*

$$P_1 = \{\mathbf{A}(\mathbf{true}\mathbf{U}p), p\}$$
$$P_2 = \{\mathbf{A}(\mathbf{true}\mathbf{U}p), \mathbf{AXA}(\mathbf{true}\mathbf{U}p)\}$$
$$P_3 = \{\mathbf{A}(\mathbf{false}\mathbf{V}\neg p), \mathbf{AXA}(\mathbf{false}\mathbf{V}\neg p), \neg p\}$$

P_1, P_2, and P_3 have enclosed all the possibilities to satisfy φ, namely formula φ is covered by P_1, P_2, and P_3.

The cover of a formula is a set of particles, and each element in the cover is considered as a state in the corresponding Kripke structure of the formula. The cover itself is enough to present the behavior of the formula. However, because the transitions in the Kripke structure should be total, namely every state has to have a successor, next, we consider the process of computing the particles that are successors of the given particle P.

Definition 4.4.4 *A formula φ is an implied successor of a particle P if $\mathbf{AX}(\varphi) \in P$. The set of implied successors of P denoted by* IMPS (P) *is* IMPS $(P) = \{\varphi : \mathbf{AX}(\varphi) \in P\}$.

Definition 4.4.5 *Let P be a particle, and the successors P be* SUCC (P)

- *if a particle P is not realizable or does not include any formula of the form $\mathbf{AX}(\varphi)$ then* SUCC (P) *is a* dummy *state.*
- *else* SUCC (P) *is equal to* COVER (IMPS (P)).

The reason why we calculate the cover of the implied successors again is that IMPS (P) is a new set of formulas which may not contain any elementary formulas. For example, IMPS $(\mathbf{AX}\varphi) = \varphi$, where φ is not an elementary formula. In this case, we have to calculate the particles of φ to see that regarding this specific formula φ, what has to be hold in the current state and what has to be hold in the next state.

In fact not all particles have successors. For example, the particle $\{\mathbf{AX}(p), \mathbf{AX}(\neg p)\}$ has no successors because any successor of this particle must contain both p and

$\neg p$ which is not a particle. A particle P is called *realizable* if there exits a Kripke structure $\mathcal{K}$ and a state s such that for each formula $\varphi \in P$, we have $\mathcal{K}, s \models \varphi$.

Consequently, only realizable particles can participate in fulfilling paths and can have meaningful successors.

Remark 4.4.1 *If a particle P is not realizable or does not include any formula of the form* $\mathbf{AX}(\varphi)$ *then the successor of a state with label P is a* dummy *state. The latter means the state reached has no commitments to satisfy any of the formulas. Thus, it may be the start of any possible paths. The dummy state is the successor of itself.*

Acceptance conditions have to be added to the tableau for each formula $\psi \in$SUB (φ) that promise to fulfill another formula. Namely, we cannot have an eventuality $\mathbf{AXA}(p\mathbf{U}r)$ where r is never fulfilled, where a formula $\psi \in$SUB (φ) is said to promise the formula r if ψ has the form $\mathbf{A}(p\mathbf{U}r)$, formally,

Definition 4.4.6 *Let P be a particle over $\psi = A(p\mathbf{U}r)$. P* fulfills *ψ that promise r if:*

- *P doesn't imply ψ or*
- *P imply r.*

Remark 4.4.2 *Let S be the states of final tableau, the set of acceptance states can be expressed as*

$$(S - \text{COVER}\ (\mathbf{AXA}(p\mathbf{U}r)))\cup \text{COVER}\ (r)$$

The above definition presents a fact that once a particle constrains a formula ψ promising r, it must contain a r to fulfill the eventuality, namely $\psi \rightarrow r$. The states in the Kripke structure satisfies such a condition are called *acceptance states.* Regarding each eventuality in the formula, there is a set of corresponding acceptance states, and we use an integer number to identify these sets of acceptance states.

Example 4.4.3 *Regarding formula* φ : $\mathbf{A}(\mathbf{true}\mathbf{U}p) \vee \mathbf{A}(\mathbf{false}\mathbf{V}\neg p)$, *the eventuality is* $\mathbf{A}(\mathbf{true}\mathbf{U}p)$. *The acceptance states of the corresponding Kripke structure should be either the states contain label p, or the states do not contain the eventuality* $\mathbf{A}(\mathbf{true}\mathbf{U}p)$ *at all.*

Given the above definitions, we describe an iterative algorithm that produces the reduced tableau, which is a Kripke structure $\mathcal{K} = (S, I, \delta, \lambda, \mathcal{F})$ of an **ACTL** formula in Algorithm 1.

Algorithm 1 tableau(ϕ)

```
I :=COVER (φ);
S := I;
δ := ∅;
Mark all particles in S as unprocessed
for each unprocessed particle P such that P ∈ S do
  S' :=SUCC (P).
  for each unprocessed particle Q such that Q ∈ S' do
    if Q is not a subset of any particle in S then
      Add Q to S.
      Mark Q as unprocessed.
    end if
    Add (P, Q) to δ.
    λ(P) := P ∩ {p : p ∈ AP} ∪ P ∩ {¬p : p ∈ AP}.
  end for
  Mark P as processed.
  if P promise r as A(pUr) then
    Add (S− COVER (AXA(pUr)))∪ COVER (p) to accept state F_i.
  end if
end for
```

In order to illustrate the size of the reduced tableau, we implemented a reduced tableau construction tool in Java. Using this tool, we generated the tableaus of the following formulas (see below) [81]. For comparison, we also generated tableaus used in the compositional verification approach proposed in [57]. The results are demonstrated in Figure 4.3 based on the number of tableau nodes against that of elementary formulas. From the figure, we can find that our the reduced tableau size

is always smaller.

$$\mathbf{AF}p, \mathbf{AX}p$$

$$\mathbf{AG}p, \mathbf{AF}(p \wedge q)$$

$$\mathbf{A}(p\mathbf{U}\ q), \mathbf{A}(p\mathbf{W}\ q)$$

$$\mathbf{A}(p\mathbf{U}\ \mathbf{A}(q\mathbf{U}\ r), \mathbf{AG}\ \mathbf{A}(p\mathbf{U}\ q)$$

$$\mathbf{AF}\ \mathbf{A}(p\mathbf{U}\ \mathbf{AG}\ q)$$

$$\mathbf{AG}\ (\mathbf{AF}\ p \wedge \mathbf{AF}\ q)$$

$$\mathbf{AF}\ p \wedge \mathbf{AF}\ \neg p$$

$$\mathbf{AFAG}\ p \vee \mathbf{AFAG}\ q$$

$$\mathbf{A}(p\mathbf{U}\ (q \wedge \mathbf{AG}\ r))$$

$$\mathbf{AG}\ (q \vee \mathbf{AGAF}p)$$

$$\mathbf{AG}\ (r \vee \mathbf{AGAF}\neg p) \vee \mathbf{AG}p \vee \mathbf{AG}r$$

$$\mathbf{AX}(\mathbf{A}(\mathbf{A}(\mathbf{A}(p\mathbf{U}q)\mathbf{W}r))\mathbf{U}\mathbf{A}(p\mathbf{W}r))$$

$$\mathbf{A}((\mathbf{AX}q \wedge r)\mathbf{WAX}(\mathbf{A}(\mathbf{A}(s\mathbf{U}p)\mathbf{W}r)\mathbf{U}\mathbf{A}(s\mathbf{W}r)))$$

$$\mathbf{AG}(q \vee \mathbf{AGAF}p) \wedge \mathbf{AG}(r \vee \mathbf{AGAF}\neg p) \vee \mathbf{AG}q \vee \mathbf{AG}r$$

$$\mathbf{AG}(q \vee \mathbf{AFAG}p) \wedge \mathbf{AG}(r \vee \mathbf{AFAG}\neg p) \vee \mathbf{AG}q \vee \mathbf{AG}r$$

$$\mathbf{AG}(q\mathbf{AFAG}p) \wedge \mathbf{AG}(r \vee \mathbf{AFAG}\neg p)) \vee \mathbf{AG}p \vee \mathbf{AG}r$$

$$\mathbf{AG}(q\mathbf{AGAF}p) \wedge \mathbf{AG}(r \vee \mathbf{AGAF}\neg p)) \vee \mathbf{AG}p \vee \mathbf{AG}r$$

Example 4.4.4 *After applying Algorithm 1, the tableau of formula* $\mathbf{A}(\mathbf{true}\mathbf{U}p) \vee \mathbf{A}(\mathbf{false}\mathbf{V}\neg p)$ *is shown in Figure 4.4. The states marked by* 0 *are* $\mathcal{F}_0$. *The state without any label except the acceptance state mark is the dummy state. The double circle states are the initial states. The* **AX***-formula labels in a state means in order to satisfy the formula what has to hold in the next state. The no-modalities-formula labels in a state means what needs to hold in the current state in order to satisfy the formula .*

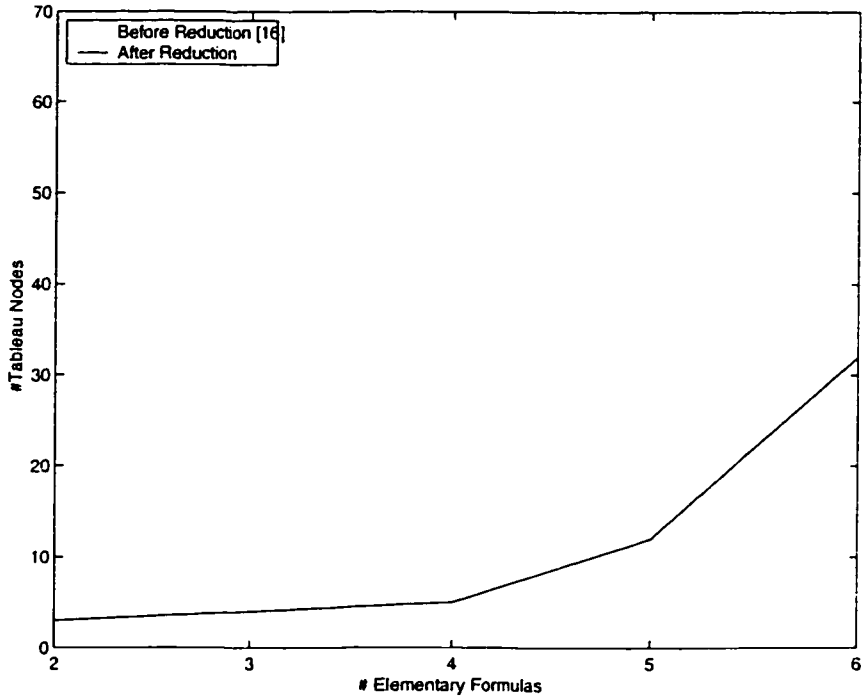


Figure 4.3: Comparison Results of Sample Formulas

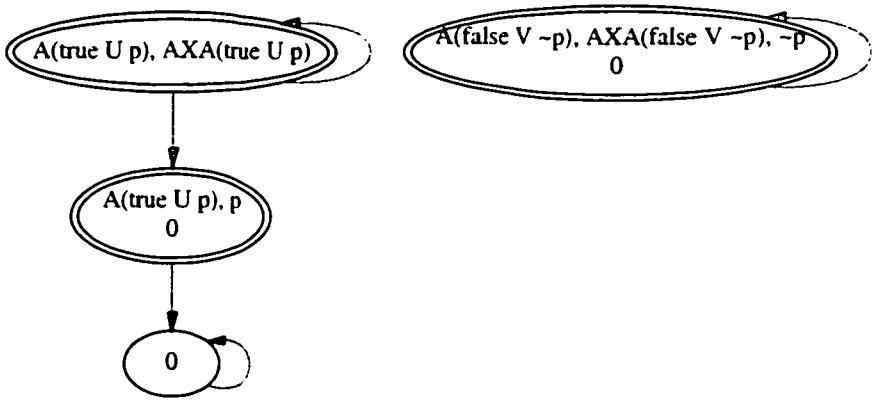


Figure 4.4: Tableau of $\mathbf{A}(\mathbf{true}\mathbf{U}p) \vee \mathbf{A}(\mathbf{false}\mathbf{V}\neg p)$

4.5 Reduced Tableau Properties

Our goal is to build a reduced fair Kripke structure $\mathcal{K}(\varphi)$ (tableau) that satisfies the formula φ, and has the following *tableau properties*:

- $\mathcal{K}(\varphi) \models \varphi$.
- for a structure $\mathcal{M}$, $\mathcal{M} \models \varphi$ iff $\mathcal{M} \preceq \mathcal{K}(\varphi)$.

This is demonstrated by the following lemmas and theorem [72].

Lemma 4.5.1 *If $\mathcal{M} \preceq \mathcal{M}'$ and φ is a formula on $\mathcal{M}'$, then $\mathcal{M}' \models \varphi$ implies $\mathcal{M} \models \varphi$.*

First, a dummy state simulates any states. When s is a non-dummy state, it is enough to show that if $s' \models \varphi$ and $s \preceq s'$, then $s \models \varphi$. We proceed by induction on the structure of formulas.

- For **true**, the proof is trivial since s and s' share the same propositional value.
- For atomic formula p and its negation, the result is obvious because when $s' \models p$ and $s \preceq s'$, $s \models p$. For conjunctions and disjunctions, it follows immediately from the induction hypothesis.
- Consider a formula of the form $\mathbf{A}(\varphi \mathbf{U} \psi)$. Let $\pi = s_0 s_1 s_2 \ldots$ be a path from $s = s_0$. we want to show that this path satisfies $\varphi \mathbf{U} \psi$. Since $s \preceq s'$, there is a path $\pi' = s'_0 s'_1 s'_2 \ldots$ from $s' = s'_0$ that corresponds to π. For each i, $s_i \preceq s'_i$. Hence by the induction hypothesis, $s'_i \models \varphi$ implies $s_i \models \varphi$, and similarly for π. If π does not satisfy $\varphi \mathbf{U} \psi$, then this implies that π' does not satisfy $\varphi \mathbf{U} \psi$ either. Hence $s' \not\models \mathbf{A}(\varphi \mathbf{U} \psi)$, a contradiction. Thus we conclude that $s \models \mathbf{A}(\varphi \mathbf{U} \psi)$. The same deduction applies to $\mathbf{A}(\varphi \mathbf{V} \psi)$.
- Consider the cases for $\mathbf{AX}\varphi$. Let s_0 be a successor of s, we want to show this state satisfies φ. Since there is a state s', such that $s \preceq s'$, and by the assumption $s' \models \varphi$, Hence, $s \models \varphi$. □

Lemma 4.5.2 *Let s be a state of $\mathcal{K}(\varphi)$. For all subformulas ψ of φ, if $\lambda(s)$ is in* COVER (ψ), *then $s \models \psi$.*

We proceed by induction on the structure of the subformula.

- For **true**, we have COVER (**true**) = $\{\emptyset\}$, which generates a dummy state and dummy state satisfies any formula, so $s \in$COVER (**true**) iff $s \models$ **true**.

- For a subformula of the form p, we have COVER $(p) = \{\{p\}\}$. According to the assumption, $\lambda(s)$ is in $\{\{p\}\}$. By choosing $\lambda(s) = \{p\}$. According to the definition of $\models$, $s \models p$.

- For the negation of an atomic formula, just note that the above two cases are iff, and each state has to be compatible.

- For formula $\chi \wedge \psi$, COVER $(\chi \wedge \psi)$ =COVER (χ) JOIN COVER $(\psi) = \{\rho \cup \tau :$ $\rho \in$COVER $(\chi), \tau \in$COVER $(\psi), \rho$ and τ are compatible $\}$. Since ρ and τ are compatible, let s be a state and $\lambda(s) = \rho \cup \tau$. Let s_1 be another state and $\lambda(s_1) = \rho$. By induction hypothesis, $s_1 \models \chi$. Because $\lambda(s_1) \subseteq \lambda(s)$, so $s \models \chi$. Same reason, $s \models \psi$. So, $s \models \chi \wedge \psi$.

- For formula $\chi \vee \psi$, COVER $(\chi \vee \psi)$ =COVER $(\chi) \cup$ COVER (ψ). if $\lambda(s) \in$COVER $(\chi) \cup$ COVER (ψ), then $\lambda(s)$ either in COVER (χ) or in COVER (ψ). By induction hypothesis, $s \models \chi \vee \psi$.

- For $\mathbf{AX}\psi$, we have COVER $(\mathbf{AX}\psi) = \{\{\mathbf{AX}\psi\}\}$. Let s be a state and $\lambda(s) = \{\mathbf{AX}\psi\}$. The label of the successor of s, $\lambda(s')$, can be calculated by $\lambda(s') \in$SUCC $(\{\mathbf{AX}\psi\})$ =COVER (IMPS $(\{\mathbf{AX}\psi\})$ =COVER $(\{\psi\})$. By induction hypothesis, $s' \models \psi$. According to the definition of $\models$, $s \models \mathbf{AX}\psi$.

- For a subformula of the form $\mathbf{A}(\chi \mathbf{V} \psi)$, COVER $(\mathbf{A}(\chi \mathbf{V} \psi)) = [\{\{\mathbf{A}(\chi \mathbf{V} \psi)\}\}$ JOIN COVER $(\{\chi, \psi\})] \cup [\{\{\mathbf{A}(\chi \mathbf{V} \psi)\}\}$ JOIN COVER $(\{\psi, \mathbf{AXA}(\chi \mathbf{V} \psi))]$. Assuming s is any state, we distinguish two cases. Case 1: if $\lambda(s) \in \{\{\mathbf{A}(\chi \mathbf{V} \psi)\}\}$ JOIN

COVER $(\{\chi, \psi\})$, then for any path π starting from s, by induction hypothesis, $\pi^1 \models \psi$. So, $s \models \mathbf{A}(\chi \mathbf{V} \psi)$; Case 2: if $\lambda(s) \in \{\{\mathbf{A}(\chi \mathbf{V} \psi)\}\}$ JOIN COVER $(\{\psi, \mathbf{AXA}(\chi \mathbf{V} \psi))$, since SUCC $(\mathbf{AXA}(\chi \mathbf{V} \psi)) = \{\mathbf{A}(\chi \mathbf{V} \psi)\}$, s has two successors, i.e., state of case 1 and 2. In the case of case 1, we have already proved $s \models \mathbf{A}(\chi \mathbf{V} \psi)$. In the case of case 2, because COVER $(\{\psi\})$ in $\lambda(s)$, by induction hypothesis, $s \models \psi$. So, for any path π starting from s, $\pi^1 \models \psi$. Namely, $s \models \mathbf{A}(\chi \mathbf{V} \psi)$ as well. □

Corollary 4.5.1 *Let s be a state of $\mathcal{K}(\varphi)$. For all subformulas ψ of φ, if $\lambda(s)$ are in* COVER (ψ), *then $\mathcal{K}(\varphi) \models \varphi$.*

This can be derived easily from Lemma 4.5.2. If s is an initial state of the tableau, then $\lambda(s) \in$COVER (φ). Hence $s \models \varphi$, and so $\mathcal{K}(\varphi) \models \varphi$. □

Lemma 4.5.3 *If $\mathcal{M} \preceq \mathcal{K}(\varphi)$, then $\mathcal{M} \models \varphi$, where $\mathcal{K}(\varphi)$ is the tableau of φ.*

Since by assumption, $\mathcal{M} \preceq \mathcal{K}(\varphi)$, then according to Lemma 4.5.1, if $\mathcal{K}(\varphi)$ satisfies a formula φ, then $\mathcal{M}$ satisfies the same formula as well. So, in order to prove $\mathcal{M} \models \varphi$, we need to prove $\mathcal{K}(\varphi) \models \varphi$. As a matter of fact, this is the conclusion of Corollary 4.5.1. Hence, we can deduce that $\mathcal{M} \models \varphi$ as well. □

Lemma 4.5.3 concludes one direction of our goal. Now we want to prove that if $\mathcal{M}' \models \varphi$, then $\mathcal{M}' \preceq \mathcal{K}(\varphi)$. This will be done by constructing an explicit simulation relation between $\mathcal{M}'$ and $\mathcal{K}(\varphi)$. The idea will be to take a state s' of $\mathcal{M}'$, look at its labelling and use this to construct a unique state of $\mathcal{K}(\varphi)$ that can simulate s'. First, we define what will be the simulation relation.

Lemma 4.5.4 *Let $\mathcal{M} = \mathcal{K}(\varphi)$, and let $\mathcal{M}'$ be another structure. Define $\sqsubseteq$ on $S' \times S$ iff the following conditions hold.*

- *For subformula p, $s' \models p$ iff $p \in \lambda(s)$ or $\lambda(s) = \emptyset$.*
- *For every $\mathbf{AX}\psi$ in the particle of φ, $\mathbf{AX}\psi \in \lambda(s)$ iff $s' \models \mathbf{AX}\psi$.*

Then $s' \sqsubseteq s$ implies that for every subformula or elementary formula ψ of φ, $s' \models \psi$ implies $\lambda(s) \in$COVER (ψ).

By induction on the structure of formulas, in this proof, the base cases are the atomic subformulas and the elementary subformulas.

- For **true**, $s' \models$ **true** iff $s \in$COVER (**true**).
- For a subformula p, in the case of dummy state, the proof is trivial since $\emptyset$ is in any set and the dummy state satisfies any formula. Otherwise, we get that $s' \models p$ iff p in $\lambda(s')$ iff p in $\lambda(s)$ iff $\lambda(s) \in$COVER (ψ).
- For a negated atomic subformula, the result follows from the facts that each state has to be compatible and that the above two cases are iffs.
- if s' satisfies a formula $\mathbf{AX}\psi$, then by definition of $\sqsubseteq$, $\mathbf{AX}\psi \in \lambda(s)$, and $\lambda(s) \in$COVER $(\mathbf{AX}\psi)$ iff $\mathbf{AX}\psi \in \lambda(s)$.
- For a subformula such as $\chi \wedge \psi$, we get that s' must satisfy χ and ψ. By the induction hypothesis, $\lambda(s) \in$COVER (χ) and $\lambda(s) \in$COVER (ψ). Hence $\lambda(s) \in$COVER $(\chi) \cap$ COVER (ψ), and $s \in$COVER $(\chi \wedge \psi)$.. Subformulas of the form $\chi \vee \psi$ are handled in a similar manner.
- Consider a subformula of the form $\mathbf{A}(\chi \mathbf{V} \psi)$. If s' is not the start of some path, then $s' \models$ **AXfalse**, so **AXfalse** $\in \lambda(s)$, and hence $\lambda(s) \in$COVER $(\mathbf{A}(\chi \mathbf{V} \psi))$. Assume s' is the start of a path. If $s' \models \mathbf{A}(\chi \mathbf{V} \psi)$, then we first have that $s' \models \psi$. By induction hypothesis, $\lambda(s) \in$COVER (ψ). Also, either $s' \models \chi$, or every successor of s' must satisfy $\mathbf{A}(\chi \mathbf{V} \psi)$. In the former case, $\lambda(s) \in$COVER (χ). In the latter, s' must satisfy $\mathbf{AXA}(\chi \mathbf{V} \psi)$. This is an elementary formula, and hence by the induction hypothesis, $\lambda(s) \in$COVER $(\mathbf{AXA}(\chi \mathbf{V} \psi))$. Similarity in the case of $\mathbf{A}(\chi \mathbf{U} \psi)$. All together, we have $\lambda(s) \in$COVER $(\mathbf{A}(\chi \mathbf{V} \psi))$ □

Lemma 4.5.5 *The relation $\sqsubseteq$ given above is a simulation relation.*

Assume $s' \sqsubseteq s$. By definition, $s' \models p$ iff $p \in \lambda(s)$ or $\lambda(s) = \emptyset$. in the case of $\emptyset$, according to the tableau construction procedure, since the self-loop on the dummy state can simulate any path and satisfy any formula, the conclusion is trivial, Now, considering the non-dummy case, suppose that $\pi' = s'_0 s'_1 s'_2 \ldots$ from $s' = s'_0$ in $\mathcal{M}'$. We will construct a path π from $s = s_0$ in $\mathcal{M}$ that corresponds to π'. Assume that we have constructed states up to s_i so far, and that we know $s'_i \sqsubseteq s_i$. Let $\mathbf{AX}\psi_0, \ldots, \mathbf{AX}\psi_{m-1}$ be the formulas that s'_i satisfies. Then s'_{i+1} must satisfy $\psi_0, \ldots, \psi_{m-1}$. Now, observe that each state of $\mathcal{M}'$ is related to a unique state of $\mathcal{M}$ by $\sqsubseteq$. Let s_{i+1} be the state related to s'_{i+1} in this manner. By the previous lemma, $\lambda(s_{i+1}) \in$COVER $(\psi_0), \ldots, \lambda(s_{i+1}) \in$COVER (ψ_{m-1}). Since $s'_i \sqsubseteq s_i$, we know that formulas $\mathbf{AX}\psi_j$ are the only formulas for which $\mathbf{AX}\psi_j \in s_i$. Thus, we found s_{i+1}, which is a successor of s_i, extends the sequence and $s'_{i+1} \sqsubseteq s_{i+1}$. Now we just have to show that this sequence satisfies the acceptance conditions.

Assume that it does not. Then looking at the acceptance conditions for the tableau, for formula $\mathbf{AXA}(\chi \mathbf{U} \psi)$, we see that there must be some states s in the acceptance conditions which are not in $(S -$ COVER $(\mathbf{AXA}(\chi \mathbf{U} \psi)))\cup$ COVER (ψ). So, $s \notin (S -$ COVER $(\mathbf{AXA}(\chi \mathbf{U} \psi))$ implies that $\mathbf{AXA}(\chi \mathbf{U} \psi) \in \lambda(s)$. By definition of $\sqsubseteq$, $s' \models \mathbf{AXA}(\chi \mathbf{U} \psi)$. However, since $s \notin$COVER (ψ), then by the previous lemma, we have $s' \not\models \psi$. But we have $s' \models \mathbf{AXA}(\chi \mathbf{U} \psi)$. This implies that π' must not be a path, a contradiction. □

Lemma 4.5.6 *if $\mathcal{M}' \models \varphi$, then $\mathcal{M}' \preceq \mathcal{K}(\varphi)$, where $\mathcal{K}(\varphi)$ is the tableau of φ.*

If $\mathcal{M}' \models \varphi$, then every initial state s' of $\mathcal{M}'$ satisfies φ. Recall the simulation relation $\sqsubseteq$ defined above pairs every such s' with a unique state s of the tableau. Now, lemma 4.5.4 implies that s is in COVER (φ), and s is an initial state. Since $\sqsubseteq$ is a simulation relation, we conclude that s' can be simulated by an initial state of the tableau. Hence $\mathcal{M} \preceq \mathcal{K}(\varphi)$. □

Theorem 4.5.1 *for a structure $\mathcal{M}$, $\mathcal{M} \models \varphi$ iff $\mathcal{M} \preceq \mathcal{K}(\varphi)$, where $\mathcal{K}(\varphi)$ is the*

tableau of φ.

The theorem includes two directions, which are proved as follows.

- By assumption $\mathcal{M} \preceq \mathcal{K}(\varphi)$, according to Lemma 4.5.3, we can deduce $\mathcal{M} \models \varphi$
- By assumption $\mathcal{M}' \models \varphi$, according to Lemma 4.5.6, we can deduce $\mathcal{M}' \preceq \mathcal{K}(\varphi)$

From the above two points, the theorem is sound. □

4.6 Summary

In this chapter. We proposed a *reduced tableau* of **ACTL** formulas, which consists of less states and transitions but accepts precisely the models of the formulas. A formula φ is first simplified by rewriting, then the tableau is constructed by decomposing φ into particles which cover all the models satisfying the models of φ using atomic propositions and the **AX** formulas. The fairness constraints are also defined by sets of states in the tableau, which will be visited infinitely often. As is often the case with tableaus for temporal logics, e.g., [36, 60], a state of the tableau consists of a set of formulas that are supposed to hold along all paths leaving the state. Unlike typical tableaus, however, the formulas in the states of the reduced tableau are interpreted over a *three-valued domain*, i.e., positive, negative, and don't care. Thus, a state may include a formula or its negation, or none of them. If the latter occurs, it reflects *a don't-care* situation, i.e., the formula may be either true or false in the state. We also prove that the reduced tableau is the maximal model of the corresponding formula, namely it covers all the models satisfying the formula. In the next chapter, we will show how to synthesize the reduced tableau into Verilog HDL modules.

Chapter 5

Verilog Synthesis

Based on the reduced tableau, we implemented an assume-guarantee reasoning approach based on the VIS model checker [11]. The implementation is made possible by synthesizing the tableau into Verilog code *automatically*. An overview of the proposed approach is depicted in Figure 5.1. The flow is implemented in Java HotSpot(TM) Client VM 1.3.0 on SUN Solaris OS. In this chapter, we will illustrate how to synthesize the reduced tableau into Verilog HDL programs.

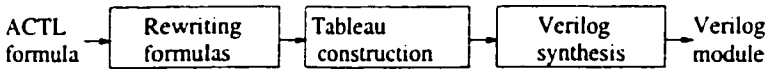


Figure 5.1: Tableau Construction and Verilog Synthesis

5.1 VIS and Verilog

5.1.1 VIS

VIS [11] is a formal verification and synthesis tool for finite-state hardware systems, developed at University of California at Berkeley and University of Colorado at Boulder. It is based on the temporal logic model checking approach, where the

properties to be checked are expressed as formulas in **CTL**, and the system is expressed as a finite state system. The formulas are checked against the finite state system to see if they are satisfied.

VIS has two design-input languages: Verilog HDL and BLIF-MV. Verilog is a widely used digital design language. The Verilog used in VIS is a subset of the commercial Verilog, which we will give a brief introduction in Section 5.1.2. BLIF-MV is a logic intermediate format designed at UC Berkeley. The relationship between a behavioral description language like Verilog and a machine description language like BLIF-MV is similar to that between a high-level programming language and an assembly language. VIS has a stand-alone compiler from Verilog to BLIF-MV called VL2MV, which extracts a set of interacting FSMs that preserves the behavior of the Verilog program. Through the interacting FSMs, VIS performs **CTL** model checking. If model checking fails, VIS reports the failure with a counter-example.

The fairness constrains are handled by a modification of **CTL**, called fair **CTL**. Fair **CTL** is characterized by the introduction of fairness constraints, which are sets of states expressed by means of **CTL** formulas, each giving a Büchi fairness condition.

5.1.2 Verilog in VIS

A Verilog specification [68] consists of one or more *modules*. The *top level module* specifies a closed system containing both test data and hardware models. Component modules normally have input and output *ports*. Events on the input ports cause changes on the outputs. Events can be either changes in the values of *wire* variables (i.e., combinational variables) or in the values of *reg* variables (i.e., register variables), or can be explicitly generated abstract events (i.e., non-deterministic values). Modules can represent pieces of hardware ranging from simple gates to complete systems (e.g., microprocessors), and they can be specified either *behaviorally* or *structurally*, or by a combination of the two. A behavioral specification defines

the behavior of a module using programming language constructs. A structural specification expresses a module as a hierarchical interconnection of submodules. The components at the bottom of the hierarchy are either primitives or are specified behaviorally.

The internal form of Verilog in VIS is BLIF-MV. Basic constructs of BLIF-MV are module declarations/instantiations, input-output relational tables which allow descriptions of non-determinism, symbolic wires, and latches. In BLIF-MV, symbolic latches are implicitly controlled by a global clock. This clock does not need to be a real wire in the hardware sense. All symbolic latches transit instantaneously to the next state indicated by the relevant transition tables. At each clock cycle, each table continuously updates its outputs according to the inputs it sees until convergence is reached. In the very beginning of the next cycle, all latches simultaneously update their present state outputs according to their next state inputs. Then again tables update their outputs accordingly.

VL2MV extracts a set of interacting FSMs that preserve the behavior of the source Verilog program defined in terms of simulated results. Allocation of hardware gates to operators in Verilog (resource binding) is based on the assumption of unlimited resources, where resources are all possible gates expressible in one table in BLIF-MV. No scheduling and optimization are performed, so the extracted FSMs are not guaranteed to be optimal (for area, speed, and so on).

A design in a synthesizable subset of Verilog consists of a set of modules (either hardware or software). The first module encountered is regarded as the root module. All modules run in parallel and communicate with each other through a set of channels (set of wire variables declared in the modules to which these channels belong). It is assumed that communication through channels is instantaneous. Within each module, values on channels can be accessed through a set of ports, which can be either wires or registers. Through wire ports, a module can input and output from and to channels instantaneously, while through register ports it takes one time unit.

A wire port has no storage element associated with it, while a register port has one storage element associated with it.

A Verilog module contains declarations, module instantiations, continuous assignments and procedural blocks. Continuous assignments begin with the keyword *assign* and are always active; they can be thought of as combinational blocks. Procedural blocks are referred to as *always* statements; statements within a procedural block are executed sequentially.

Module instances, continuous assignments, and procedural blocks within a module run concurrently. Execution of each continuous assignment, basic block in a procedural block and module instance is assumed to be atomic within each instant. If there is more than one procedural block in the same module, and outputs of one are inputs to another, the simulated result may depend on how expressions from different blocks are interleaved by the simulator.

5.2 Verilog Synthesis Procedures

In the tableau construction, we obtain a tableau from an **ACTL** formula, which covers all the behaviors that satisfy the formula. In this section, we will present how to synthesis the tableau into a Verilog module.

According to the syntax [68] of Verilog, we use the procedures in Figure 5.2.

In order to distinguish the states in the tableau, for each state in the tableau, we allocate an ID to each of them. For example, allocate S_0 to state $\{\mathbf{A}(\mathbf{false}\mathbf{V}\neg p)$, $\mathbf{AXA}(\mathbf{false}\mathbf{V}\neg p), \neg p\}$ as shown in Figure 5.3. For the transitions in the tableau, we also allocate an ID to it. For example, allocate S_2_NEXT to the transition going out of S_2. However, there exist many transitions going out of one state. In this case, these transitions share one ID. For example, the IDs of the transitions from S_1 to S_1 and form S_1 to S_2 are both S_1_NEXT (Figure 5.3).

In order to obtain the proposition labels in each state in the tableau, we define

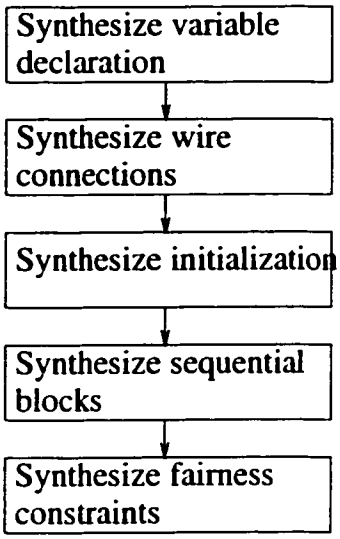


Figure 5.2: Verilog Synthesis Procedures

$comp(\varphi)$ as the set of atomic components in the formula. For example, if φ is $\mathbf{A}(\mathbf{true}\mathbf{U}p) \vee \mathbf{A}(\mathbf{false}\mathbf{V}\neg p)$, then $comp(\varphi) = \{p\}$.

When constructing the Verilog module, since the **ACTL** formulas with modalities are not acceptable in Verilog, we must label each state in the tableau explicitly by the components in $comp(\varphi)$. In the following, we distinguish three cases.

1. In a state, if there are propositions without modalities, we put the propositions as the label of state S_i and remove the other propositions with modalities.

2. In a state, if every proposition has modalities, we put the negation of all the atomic propositions as the label of the state, and remove all the propositions with modalities.

3. If a state is the dummy state, we set all the components in $comp(\varphi)$ as the non-deterministic true or false.

Example 5.2.1 *The label of the states in the tableau of* $\mathbf{A}(\mathbf{true}\mathbf{U}p) \vee \mathbf{A}(\mathbf{false}\mathbf{V}\neg p)$ *shown in Figure 5.3 are as follows.*

- $comp(\mathbf{A}(\mathbf{true}\mathbf{U}p) \vee \mathbf{A}(\mathbf{false}\mathbf{V}\neg p)) = \{p\}$.

- *In state S_0, every sub-formula has modalities. So, $\neg p$ is the label of this state.*
- *The label of state S_1 is set to $\neg p$, since $\neg p$ is in the label of this state.*
- *The label of state S_2 is set to p, since p is in the label of this state.*
- *Since state S_3 is a dummy state, we put all the components in comp(φ) as non-deterministic variables, namely p is set as a non-deterministic true or false.*

Figure 5.3: Tableau of $\mathbf{A}(\mathbf{true}\mathbf{U}p) \vee \mathbf{A}(\mathbf{false}\mathbf{V}\neg p)$ with IDs

Till now, we have has enough information to construct the Verilog module of the tableau of an **ACTL** formula φ.

Synthesize Variable Declaration Since an **ACTL** formula φ is to describe the behaviors of the components in *comp*(φ), in the simplest case, we set the outputs of the synthesized Verilog module as the components in *comp*(φ), namely the synthesized program is to emulate the behaviors of φ.

comp(φ) is obtained from the **ACTL** formula φ iteratively. The procedure can be outlined as follows. (1) if the formula is bi-nary, namely has two operands, both the operands are scheduled to be variables in the Verilog behavior module; (2)

if the formula is uni-nary, namely has one operand, the operand is scheduled to be a variable. (3) if the formula is an atomic proposition, then this atomic proposition is a variable. The above procedure is iterated until all the atomic propositions are obtained.

Example 5.2.2 *Given a formula* φ : $\mathbf{AG}(p \wedge \mathbf{AF}(q))$, *the procedure to obtain* $comp(\varphi)$ *is as follows.*

- *Since* $\mathbf{AG}(p \wedge \mathbf{AF}(q))$ *is an uni-nary formula,* $(p \wedge \mathbf{AF}(q))$ *is scheduled to be in* $comp(\varphi)$.
- *Since* $(p \wedge \mathbf{AF}(q))$ *is a bi-nary formula, both* p *and* $\mathbf{AF}(q)$ *are scheduled to be in* $comp(\varphi)$.
- *Since* p *is an atomic proposition,* p *is in* $comp(\varphi)$.
- *Since* $\mathbf{AF}(q)$ *is an uni-nary formula,* q *is scheduled to be* $comp(\varphi)$.
- *Since* q *is an atomic proposition,* q *is in* $comp(\varphi)$.

From the above procedure, we know that $comp(\varphi) = \{p, q\}$. *The synthesized program is to be claimed as*

```
module <module_name> (p, q);
output p,q;
```

All the states in the tableau are coded according to their IDs. For example, S_0 is coded as '0', S_1 is coded as '1', and so on. The encoded states are declared as a register vector, *STATE*. The size of the register vector is determined by the number of the states in the tableau. Namely, the size of the register vector equals to $\lceil \log_2(\textit{the number of the states}) \rceil$. For example, if there are 7 states in the tableau, we need 3-bit to contain the IDs of the states, so the *STATE* is declared as

```
reg [2:0] STATE;
```

Synthesize the Wire Connection Transitions in the tableau are declared as wire connections according to their IDs. For example, S_1_NEXT is declared as $S_1\_NEXT\_W$, S_2_NEXT is declared as $S_2\_NEXT\_W$, and so on. The size of the wire connections is the same as that of the *STATE* because the next transition can be anyone of the states. So, if the size of the *STATE* is n, then the wire connections are declared as n as well, i.e.,

```
wire [n-1:0] S_i_NEXT_W;
```

In a similar way, the wire connection of the initial states is declared as

```
wire [n-1:0] S_INIT_W;
```

S_INIT_W may be connected to a random-number-generator, so that we obtain the non-deterministic initial states according to the tableau.

However, VIS does not allow multi-output non-deterministic constants. Namely it allows a random statement like x = *$ND (0,1,2,3)* where x is a 2-bit boolean variable and $\$ND$ is the random-generator instruction, but does not allow the random statement like x = *$ND(0,1,2)* because internally VIS split the 2-bit variable into two 1-bit non-deterministic variables which will lead to the possibility of $x = 3$. However, such a situation comes up naturally when we want the initial states to be state 0, 1, and 2 but not 3. A solution around this case is to declare a temporary variable and then merge the unused non-deterministic numbers with another non-deterministic value like the following.

```
assign S_INIT_W_TMP = $ND(0,1,2,3);//$
assign S_INIT_W = (S_INIT_W_TMP == 3) ? 2 : S_INIT_W_TMP;
```

where $S\_INIT\_W\_TMP$ is the temporary variable. The value 3 is merged with 2 and the new non-deterministic value set $\{0, 1, 2\}$ is assigned to non-deterministic initial state variable S_INIT_W which indicates what the initial states are. However, when there is only one initial state in the tableau, we do not need $S\_INIT\_W\_TMP$, but declare S_INIT_W directly as the initial state.

Since the initial states may be any state in the state space, in order to improve the efficiency of the generation and to make the above temporary variable setting easy, we sort the initial state IDs. The temporary variable setting procedure is updated as follows: (1) the IDs of initial states are bubble-sorted; (2) set $initMark$ as one of the initial states; (3) set a counter i which starts from 0 and ends with the minimum number which is the power of 2 and greater than maximum ID among the initial states; (4) scan the sorted initial-state-ID-list, if an integer number i is not in the IDs of the initial states, then i is merged on-the-fly with respect to $initMark$.

Example 5.2.3 *Given the initial states are* S_3 *by the ID of* 3, S_6 *by the ID of* 6, S_2 *by the ID of* 2, *and* S_5 *by the ID of* 5, *the temporary variable setting procedure is shown as follows.*

- *Sort the IDs of the initial states from* $3, 6, 2, 5$ *to* $2, 3, 5, 6$.
- *Set* $initMark$ *as one initial state, for example* 3.
- *Since the minimum number which is the power of* 2 *and greater than* 6 *is* 8, *we set* i *as an integer from* 0 *to* 8.
- *Scanning the sorted next state IDs,* 0 *and* 1 *are unused random numbers, so we put* $((S\_INIT\_W\_TMP = 0) \vee (S\_INIT\_W\_TMP = 1))$. *Then* 2 *and* 3 *are used random numbers. However,* 4 *is an unused random number, and so are* 7 *and* 8. *We increase the unused number list on-the-fly and get* $((S\_INIT\_W\_TMP = 0) \vee (S\_INIT\_W\_TMP = 1) \vee (S\_INIT\_W\_TMP = 4) \vee (S\_INIT\_W\_TMP = 7) \vee (S\_INIT2\_W\_TMP = 8))$. *This list is to be merged with respect to the* $initMark$, *3.*

The initial state structure is then declared as follows.

```
assign S_INIT_W_TMP = $ND(0,1,2,3,5,6,7,8);//$
assigm S_INIT_W = ((S_INIT_W_TMP = 0) || (S_INIT_W_TMP = 1) ||
```

```
(S_INIT_W_TMP = 4) || (S_INIT_W_TMP = 7) ||
(S_INIT_W_TMP = 8)) ? 3 : S_INIT_W_TMP;
```

The transitions in the tableau are handled in a similar way. If the next state of state S_2 is state S_3, then the wire connection $S\_2\_NEXT\_W$ is assigned value '3' as

```
assign S_2_NEXT_W = 3;
```

Moreover, if transition $S_i\_NEXT\_W$ has non-deterministic next states, we will declare non-deterministic values by a temporary variable $S_i\_NEXT\_W\_TMP$ and then merge the unused values in the declaration of $S_i\_NEXT\_W$. For example, if the next state of state S_i is S_0, S_1 and S_2, then the next state transitions are declared as follows.

```
assign S_i_NEXT_W_TMP = $ND(0,1,2,3);//$
assigm S_i_NEXT_W = (S_i_NEXT_W_TMP == 3) ? 2 : S_i_NEXT_W_TMP;
```

Synthesize the Initialization Block The next step is to generate the initial blocks of the program. In Verilog, actually there is no initialization phase. Everything is initialized to 'x', and the *initial* blocks are identical to *always* blocks, except that they execute only once, whereas always blocks execute forever, as if they were stuck in an infinite loop. Whether or not the initial blocks will be executed first depends on the simulation or verification tool. Some tools choose to execute the initial blocks first. However, most of the tools first execute blocks in the same order as they are specified in the file. But subsequent execution order is not so deterministic. In VIS, the initial blocks are alway the first one to be executed, thus the statements in the initial blocks are used to construct the initial conditions in the state space.

When we specify the initial states, we need to specify first the initial states among the states in the tableau, second the positive atomic propositions and negative atomic propositions in the initial states, namely which atomic proposition equals to true and which equals to false.

Since we have specified the non-deterministic initial wire connections S_INIT_W, so in the initial block we can simply declare the initial states as

```
initial STATE =  S_INIT_W;
```

Thus, the initial states are determined by the wire connection S_INIT_W. If there are more than one value in S_INIT_W, the initial states are non-deterministic. However, if there is only one initial state in the tableau, we need not bother ourselves to use S_INIT_W and $S\_INIT\_W\_TMP$. We can directly assign the initial state to the $STATE$ variable in the initial block.

Since there are three kinds of states in the tableau, i.e., the state with atomic proposition labels (normal state), the state without any atomic proposition labels (null state), and the state with dummy labels (dummy state), in the generation of the positive/negative atomic propositions, we have to distinguish these cases. In the null state, the label is implicitly the negation of all the atomic propositions, namely if the atomic proposition is p in the formula, the label of the null state is $\neg p$. In the dummy state, the label is non-deterministically either the atomic propositions or the negation of the propositions. In the above example, the label of the dummy state is non-deterministic p or $\neg p$.

The procedure to obtain the labels of the initial states has two steps. The first is to get the atomic propositions, which equal to true in the initial states; the second step is to get the atomic propositions, which equal to false in the initial states.

The procedure to get the positive atomic propositions of the initial block is as follows. (1) if there is a null set in the initial states of the tableau, the positive atomic proposition set is empty since all the negative atomic propositions will be in the initial states, so it is impossible for the initial states contain any positive atomic propositions; (2) all the labels in the initial states perform intersection with $comp(\varphi)$, and the result is the positive atomic propositions in the initial block in the program.

Example 5.2.4 *Given* $comp(\varphi) = \{p, q, r, s, t\}$ *, where* φ *is the* **ACTL** *formula and*

the labels of the initial states are states $S_0 = \{p, r\}$ *and* $S_1 = \{p, s, t\}$*, the procedure to get the positive atomic propositions of the initial block is as follows.*

- *The label of* $S_0 = \{p, r\}$ *intersects with* $comp(\varphi) = \{p, q, r, s, t\}$*. The result is* $\{p, r\}$*.*
- *The label of* $S_1 = \{p, s, t\}$ *intersects with the result of the previous step* $\{p, r\}$*. The result is* $\{p\}$*.*
- *Since all the initial states are processed, the positive atomic propositions of the initial block is* $\{p\}$*.*

Consequently, the proposition is declared in the initial block as follows.

```
initial p =  1;
```

The procedure to get the negative atomic propositions of the initial block is as follows. (1) establish a set $\neg comp(\varphi)$ which contains the negations of the variables in $comp(\varphi)$, where φ is the **ACTL** formula; (2) all the labels in the initial states perform intersection with $\neg comp(\varphi)$, and the result is the negative atomic propositions in the initial block in the program.

Example 5.2.5 *Given* $comp(\varphi) = \{p, q, r, s, t\}$ *, where* φ *is the* **ACTL** *formula and the labels of the initial states are states* $S_0 = \{\neg p, r\}$ *and* $S_1 = \{\neg p, \neg s, t\}$*, the procedure to get the positive atomic propositions of the initial block is as follows.*

- $\neg comp(\varphi)$ *equals to* $\{\neg p, \neg q, \neg r, \neg s, \neg t\}$*.*
- *The label of* $S_0 = \{\neg p, r\}$ *intersects with* $\{\neg p, \neg q, \neg r, \neg s, \neg t\}$*. The result is* $\{\neg p\}$*.*
- *The label of* $S_1 = \{\neg p, \neg s, t\}$ *intersects with the result of the previous step* $\{\neg p\}$*. The result is* $\{\neg p\}$*.*

- *Since all the initial states are processed, the negative atomic propositions of the initial block is* $\{\neg p\}$.

Consequently, the proposition is declared in the initial block as follows.

```
initial p = 0;
```

Synthesize the Sequential Block The sequential behavior of the tableau is described by a sequential block in Verilog, which is an *always* block and uses the *case* statement to branch the transitions with respect to the variable *STATE*. Every state in the tableau is a case branch in the block. The value of the variables in $comp(\varphi)$ are specified in each corresponding state (case branch). There are three kinds of states in the tableau, the normal state, the null state, and the dummy state. In the normal state, the value of the components in $comp(\varphi)$ are set according to the labels of the state. In the null state, they are set to 0. In the dummy state, they are set to non-deterministically 0 or 1. Because the always block in VIS does not allow non-deterministic declarations, we put the non-deterministic declarations of the components in $comp(\varphi)$ in the combinational part as a wire variable and declare as follows.

```
assign p_NDW = $ND(0, 1);//$
```

where p is one of the components in $comp(\varphi)$ and p_NDW is its corresponding wire variable in the combinational part.

Since we have set the next state transitions in the combination part, we can use the results directly. For example, we can set the next state transition of state S_2 as

```
assign STATE = S_2_NEXT_W;
```

Example 5.2.6 *In a tableau,* $comp(\varphi) = \{p, q\}$ *the next state of* S_0 *is* $S_1, \ldots, S_n$. *The label of* S_0 *is* $\{p, \neg q\}$. S_1 *is a null state.* S_5 *is a dummy state. These states are captured by the following always block.*

```
always  begin
   case (STATE)
     0: begin // state S0
        p = 1; // p is true in this state.
        q = 0; // q is false in this state.
        STATE = S0_NEXT_W; //the next state is S0_NEXT_W which
                           //is preset to 1,...n.
     end
     1: begin // state S1
        p = 0; // this is a null state.
        q = 0;
        STATE = S1_NEXT_W; //the next state is S1_NEXT_W which
                           //is preset in the combinational part
     end
     ......
     5: begin // state S5
        p = p_NDW; // this is a dummy state
        q = q_NDW;
        STATE = S5_NEXT_W; //the next state is S5_NEXT_W which
                           //is preset in the combinational part
     end
     ......
end
```

Synthesize the Fairness Constrains It is often necessary to introduce some notion of fairness. For example, if the system allocates a shared resource among several users, only those paths along which no user keeps the resource forever should be considered. In VIS, fairness constrains are sets of states expressed by means of **CTL**

formulas, each giving a fairness condition which is satisfied infinitely often along a fair path, namely Büchi fairness constrains. An example of a fairness condition is p, that restricts the system to only those paths where p is asserted infinitely often.

Here, we will generate the fairness constrain file which is going to be used as the input of VIS during the verification. The file contains a set of formulas, where each of them has to be asserted infinitely often.

The fairness constraints are specified by the acceptance-state-marks in the tableau. A state in the tableau marked by i means that this state is an acceptance state of the i^{th} generalized Büchi fairness set. At least one acceptance state in each fairness set has to be asserted infinitely often.

The procedure to generate the fairness constraint is as follows. (1) scan the tableau and get the ID of the state which is in the i^{th} fairness set; (2) repeat the first step until all the IDs of the states in each fairness set are generated; (3) all the IDs in each fairness set are or-ed as a formula and this formula is put in the fairness constraint file of VIS. So, if there are n fairness sets, there will be n formulas in the fairness constraint file.

Example 5.2.7 *In a tableau with states S_0, S_1, and S_2, the acceptance marks in S_0 are $\{0,2\}$; the acceptance marks in S_1 are $\{0\}$; the acceptance marks in S_2 are $\{0,1,2\}$. The formulas in the fairness constrain file are obtained as follows.*

- *The states in the 0th fairness set are S_0, S_1, and S_2. So, the formula is $((STATE = S\_0) \vee (STATE = S\_1) \vee (STATE = S\_2))$.*
- *The state in the 1th fairness set is S_2. So, the formula is $(STATE = S\_2)$.*
- *The states in the 2th fairness set are S_0, and S_2. So, the formula is $((STATE = S\_0) \vee (STATE = S\_2))$.*

The fairness constrain file in VIS thus is specified as

```
((STATE = S_0) || (STATE = S_1) || (STATE = S_2));
```

```
 (STATE = S_2);
((STATE = S_0)|| (STATE = S_2));
```

5.3 Applications

In this section, we will demonstrate the applications of the environment synthesis approaches. Assuming we want to prove properties of a module M under a certain environment *env*. Module *env* captures the input signals of M. A property φ usually describes the corresponding behaviors of the output signals under the stimulus of the input signals as shown in Figure 5.4.

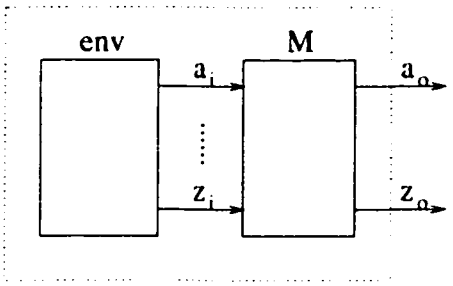


Figure 5.4: A Closed System

Generally, module M is written in some hardware description language such as Verilog or VHDL. In order to verify M, there are at least two requirements upon module *env*. First, *env* should be in the same language as that of M. Secondly, *env* has to be an accurate and complete representation of the real environment, not a simple case.

In our verification approach, the stimulus is captured by **ACTL** formulas and the formulas are translated into a finite state machine (tableau) through the tableau construction. In this approach, the second requirement is implicitly satisfied since the tableau includes all the behaviors of the formulas.

In order to satisfy the first requirement, we have to translate the tableau into the same language of the module under verification. Here, we will synthesis the

tableau into a Verilog module. We know that this Verilog module is only used in the verification of module M, but not a part of the final implementation. Therefore, we can take advantages of the full range of the Verilog language to construct the module, but need not care whether or not this module can be synthesized into a lower level, for example gate level. Namely, even if module M is an RTL level implementation, we still can construct a behavior level Verilog environment module for verification purposes.

Example 5.3.1 *Given the tableau of* $\varphi = \mathbf{A}(\mathbf{true}\mathbf{U}p) \vee \mathbf{A}(\mathbf{false}\mathbf{V}\neg p)$ *as shown in Figure 5.3, mechanically we procedure below the synthesized Verilog code, where states* $0, 1, 2, 3$ *are the states* S_0, S_1, S_2, S_3*, respectively. The line numbers are added manually for illustration purposes in this example. The synthesis including the tableau construction takes 2 seconds CPU time on a SUN Ultra 5. In the Verilog code, the Lines 0 to 5 are comments. The initial states are declared under "Initialization", which are state* $0, 1, 2$ *non-deterministically (Line 15). The next state transitions of the states in the tableau are set under "Combinational part" (Lines 20 to 25). Lines 27 to 47 represent the "Sequential part". Each state in the tableau is a case branch in the always block. In each state, the component in p is set a value according to the label in the tableau.*

```
L0: //`define TRUE 1
L1: //`define FALSE 0
L2: //`define S0 0
L3: //`define S1 1
L4: //`define S2 2
L5: //`define S3 3
L6: module tableau(p);
L7: output p;
L8: //Variable declaration
```

```
L9: reg p;
L10: wire pND_W;
L11: reg [1:0] STATE;
L12: wire [1:0] S_INIT_W_TMP, S_INIT_W, S0_NEXT_W,
                S1_NEXT_W, S2_NEXT_W, S3_NEXT_W;
L13: //Initialiazation
L14: assign S_INIT_W_TMP = $ND(0, 1, 2, 3);//$
L15: assign S_INIT_W = ((S_INIT_W_TMP == 3)) ? 2 : S_INIT_W_TMP;
L16: initial begin
L17:     STATE = S_INIT_W;
L18: end // Initial
L19: //Combinational part
L20: assign S2_NEXT_W = 3;
L21: assign S3_NEXT_W = 3;
L22: assign S1_NEXT_W = 1;
L23: wire [1:0] S0_NEXT_W_TMP;
L24: assign S0_NEXT_W_TMP = $ND (0,1,2,3);//$
L25: assign S0_NEXT_W = ((S0_NEXT_W_TMP == 1) || (S0_NEXT_W_TMP == 3))
                        ? 2 : S0_NEXT_W_TMP;
L26: assign pND_W = $ND(0, 1);//$
L27: //Sequential part
L28: always begin
L29:    case (STATE)
L30:    0: begin
L31:        p = 0;
L32:        STATE = S0_NEXT_W;
L33:       end
L34:    1: begin
```

```
L35:          p= 0;
L36:          STATE = S1_NEXT_W;
L37:         end
L38:     2: begin
L39:          p= 1;
L40:          STATE = S2_NEXT_W;
L41:         end
L42:     3: begin
L43:          p = pND_W;
L44:          STATE = S3_NEXT_W;
L45:         end
L46:     endcase // case (STATE)
L47: end // always begin
L48: endmodule // tableau
```

The synthesized fairness constraint is shown as follows.

```
(tableau.STATE = 1
 || tableau.STATE = 2
 || tableau.STATE = 3
);
```

There is only one set of fairness constraint in the tableau which includes three acceptance states, S_1, S_2, and S_3. So, there is only one formula in the file which includes three or-ed states.

Example 5.3.2 *We use here a benchmark example of an arbiter from the VIS package. There are three entities in the system: clients, controllers, and an arbiter as shown in Figure 5.5. The client sends a req signal to the controller to request services. The arbiter will decide if the controller should give services to the client. If*

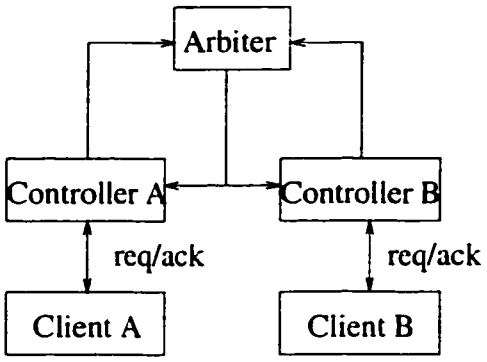


Figure 5.5: The Arbiter

the controller gets the permission, it will return an ack signal to the client. Now, the property is if the client sends the *req* signal and the arbiter gives the permission, the controller will acknowledge the *req* signal of the client. *We will prove the property in our compositional verification framework by the following steps:*

1. *Assume that the client sends the req signal and the arbiter gives the permission.*
2. *Construct the tableau of the above assumption.*
3. *Synthesize the tableau into the Verilog behavior module.*
4. *Compose the environment module and the controller module, and verify the acknowledge signal will eventually come true.*
5. *Discharge the assumptions on the client and the arbiter, respectively.*

Regarding the first step, we use two **ACTL** *formulas instead of the informal assumptions. One is* $\mathbf{AFAG}(req = 1)$*; the other is* $\mathbf{AF}(select = 1)$*, where* $select = 1$ *means that the arbiter gives the permission.*

The above two formulas can be synthesized into Verilog modules. These modules can be composed with the controller module and construct a verification system as follows:

```
module verif_system();
```

```
    environment1 environment1_instance (req);
    environment2 environment2_instance (select);
    controller  controller_instance  (req, select, ack);
endmodule // verif_system
```

*where, req and select are the outputs of environment*1 *and environment*2*, respectively.*

Then, this verif_system module can be put into a model checking tool to verify that ack will eventually come true, namely $\mathbf{AF}(ack = 1)$*. The property is verified in VIS. The verification CPU time is 0.7 second on a SUN ultra 5 workstation, and the memory in use is 383 KB.*

The synthesized Verilog modules of the two assumptions are as follows.

```
module environment1(req);
output req;

//Variable declaration
reg req;
wire reqND_W;
reg [0:0] STATE;
wire [0:0] S_INIT_W_TMP, S_INIT_W, S0_NEXT_W, S1_NEXT_W;

//Initialiazation
assign S_INIT_W = $ND(0, 1);

initial begin
    STATE = S_INIT_W;
end // Initial
```

```
//Combinational part
assign S1_NEXT_W = $ND(0, 1);//$
assign S0_NEXT_W = 0;
assign reqND_W = $ND(0, 1);//

//Sequential part
always begin
   case (STATE)
   1: begin
       req = 0;
       STATE = S1_NEXT_W;
      end
   0: begin
       req= 1;
       STATE = S0_NEXT_W;
      end
   endcase // case (STATE)
end // always begin
endmodule // environment



module environment2(is_selected);
output is_selected;

//Variable declaration
reg is_selected;
wire is_selectedND_W;
reg [1:0] STATE;
```

```
wire [1:0] S_INIT_W_TMP, S_INIT_W, S0_NEXT_W, S1_NEXT_W, S2_NEXT_W;

//Initialiazation
assign S_INIT_W_TMP = $ND(0, 1, 2, 3);//$
assign S_INIT_W = ((S_INIT_W_TMP == 2) || (S_INIT_W_TMP == 3))
                  ? 1 : S_INIT_W_TMP;

initial begin
    STATE = S_INIT_W;
end // Initial

//Combinational part
wire [1:0] S1_NEXT_W_TMP;
assign S1_NEXT_W_TMP = $ND (0, 1, 2, 3);//$
assign S1_NEXT_W = ((S1_NEXT_W_TMP == 2) || (S1_NEXT_W_TMP == 3))
                   ? 1 : S1_NEXT_W_TMP;
assign S0_NEXT_W = 2;
assign S2_NEXT_W = 2;
assign is_selectedND_W = $ND(0, 1);//$

//Sequential part
always begin
   case (STATE)
   1: begin
       is_selected = 0;
       STATE = S1_NEXT_W;
      end
   0: begin
```

```
        is_selected= 1;
        STATE = S0_NEXT_W;
       end
   2: begin
        is_selected = is_selectedND_W;
        STATE = S2_NEXT_W;
       end
   endcase // case (STATE)
end // always begin
endmodule // environment
```

The verification script of the controller module is as follows:

```
read_verilog verif_system.v
init
read_fairness environment.fair
time
mc -i -d 1 -f a.out arbiter.ctl
time
print_bdd_stats
```

where verif_system.v is the source file; environment.fair is the fairness constrain file for the environment since there is an eventual subformula in the environment; arbiter.ctl is the file containing the above three properties.

5.4 Summary

In this chapter, we illustrated how to synthesize the reduced tableau into Verilog behavioral level HDL module through a conventional encoding of finite state machine and by taking the advantages from VIS-Verilog specific features such as non-deterministic assignments. The synthesized Verilog module then can be composed

with a design under verification as the environment. The whole procedure is implemented in Java 1.3.0 on SUN Solaris 5.7.

Part II

Syntactic Model Reduction

Chapter 6

Model Reduction

In spite of the impressive progress in the development of model checking, state space explosion is still a major problem. It is generally recognized that the only way to scale up model checking to industry designs is the modularization and the model reduction. In the sense of modularization, a system is decomposed into many modules. The properties of the system are deduced by verifying the modules separately as we mentioned in the previous chapters. However, in the million-gate era, even the size of one module can easily break the model checking limit.

Beyond compositional approaches, model reduction is the most important technique for relieving the state explosion problem. Model reduction is a general approach [23, 20] which reduces a concrete system (M) under verification to a more abstract and smaller one (M'). Both systems M and M' are connected by an abstraction relation which is *safe* with respect to a given property φ, namely it preserves the property. This means if the property holds for the abstract system, it holds for the concrete one as well. More formally, the property φ is either *weakly* preserved if $M' \models \varphi \Rightarrow M \models \varphi$, or *strongly* preserved if $M' \models \varphi \equiv M \models \varphi$. It should be intuitively clear that the more weakly the property is preserved, the more reduction can be achieved.

Generally, in industry, one module may contain both control and a big datapath [50]. Hopefully, in practice, the size of the control structure is not large so that model checking can handle it using the ROBDD presentation. However, in the verification of the datapath, because of the large quantity of memory elements, model checking leads to state space explosion in the verification. In this case, we have to reduce the model to avoid the construction of the whole state space that might be too large to be fit into memory.

There are two ideas on reducing the model. One is to separate the datapath from the control logic. Generally, using the ROBDD based representation, the control unit can be verified [86, 16, 34, 2, 14, 85]. The concrete datapath elements are verified separately. In this framework, we have to deal with the feedbacks between the control units and the datapaths. The above approaches are very efficient but ad-hoc because they cannot handle general properties.

The other idea is to derive a reduced model from the concrete module with the control unit and the datapath part being together, and guarantee that the small reduced model preserves the properties of the original module [23, 57, 20, 55, 21, 54, 5, 3, 87, 67, 22, 19]. In this framework, obviously, we do not need to consider the feedbacks between the control unit and the datapath. Thus, using model checking, we can possibly verify a real industry design with complex control logic and big datapaths.

In the following, we illustrate the model reduction methods with combined control and datapaths, which include the *cone of influence reduction* [55] (or property localization [54]) and *symbolic data abstraction* [21, 20, 57, 19]. These methods are widely used in the model checking tools, such as FormalCheck, SMV, etc.

6.1 Cone of Influence Reduction

We have defined the Kripke structure in Section 2.1 which is used as a formal model of FSMs. Here, we refine the definition to adapt the circuit designs in practice.

Let $V = \{v_1, \ldots, v_n\}$ be the set of variables of a given circuit. This circuit can be described by a Kripke structure $M = (S, I, \delta, \lambda)$, where

- $S = \{0, 1\}^n$ is the set of all valuations of V.
- $I \subseteq S$ is the initial states.
- $\delta = \bigwedge_{i=1}^{n} [v_i' = f_i(V)]$, for each $v_i \in V$, where f_i is a boolean function.
- $\lambda(s) = \{v_i | s(v_i) = 1 \text{ for } 1 \leq i \leq n\}$ are the label functions.

Suppose we want to verify a specification φ, where $V' \subseteq V$ is the set of variables that are of interest with φ. We would like to reduce the original model M with respect to the specification φ, namely we will abstract away all the variables and transitions which are not of interest with the specification φ (property localization). However, because of the internal transitions in M, the values of variables in V' might depend on values of variables not in V'. Therefore, we define a cone of influence C of V' and use C in order to reduce the original model M.

The *cone of influence* $C = \{v_1, \ldots, v_k\}$ of V' is the minimal set of variables such that

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

C is created by removing all the transitions whose left hand side variables do not appear in C. Since C is the subset of the variables of structure M (implementation) and includes all the variables of interest with the variables in specification φ, it is an abstraction of the variables in M. We thus can construct a new reduced structure $\widehat{M} = (\widehat{S}, \widehat{I}, \widehat{\delta}, \widehat{\lambda})$ base on the cone of influence $C = \{v_1, \ldots, v_k\}$, where

- $\widehat{S} = \{0,1\}^k$ is the set of all valuations of $\{v_1, \ldots, v_k\}$.
- $\widehat{I} = \{(\widehat{d_1}, \ldots, \widehat{d_k}|$ there is a state $(d_1, \ldots, d_n) \in S_0$ such that $\widehat{d_1} = \widehat{d_1} \wedge \ldots \wedge \widehat{d_k} = d_k\}$.
- $\widehat{\delta} = \bigwedge_{i=1}^{k} [v_i' = f_i(V)]$.
- $\widehat{\lambda}(\widehat{s}) = \{v_i | \widehat{s}(v_i) = 1 \text{ for } 1 \leq i \leq k\}$.

Let B is a relation between M and $\widehat{M}$, one can prove that B is a equivalence relation [22]. Hence, a direct consequence is that B preserves the **CTL*** properties [22]. Namely, let f be a **CTL*** property with atomic propositions in C, $M \models f$ if and only if $\widehat{M} \models f$.

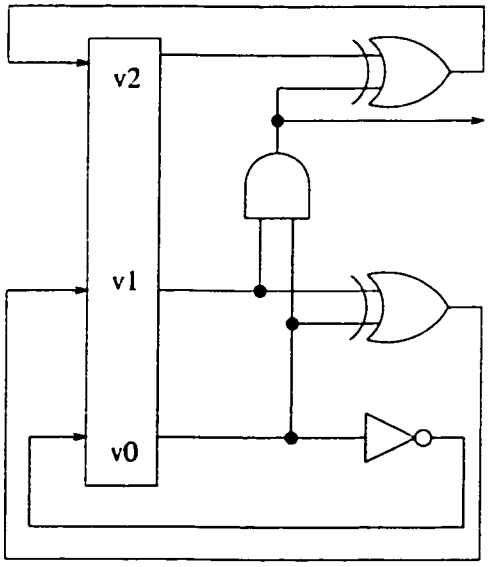


Figure 6.1: Synchronous Modulo 8 Counter

Consider an synchronous modulo 8 counter example as shown in Figure 6.1, where

$v_0' = \neg v_0$

$v_1' = v_0 \oplus v_1$

$v_2' = (v_0 \wedge v_1) \oplus v_2$

Clearly, if $V' = \{v_0\}$ then $C = \{v_0\}$, since f_0 does not depend on any variable other than v_0. If $V' = \{v_1\}$ then $C = \{v_0, v_1\}$, since f_1 depends on both of the variables, but $v_2 \notin C$ because no variable in C depends on v_2. Finally, if $V' = \{v_2\}$ then C is the set of all the variables. In this case, the reduction is not working.

6.2 Symbolic Abstraction

Symbolic abstraction is based on the observation that the specification of systems that includes datapaths usually involves fairly simple relationships among the data values in the system. The abstraction is usually specified by giving a mapping between the actual data values in the system and a small set of abstract data values. By extending the mapping to states and transitions, it is possible to produce an abstract version of the system under consideration as shown in Figure 6.2. We

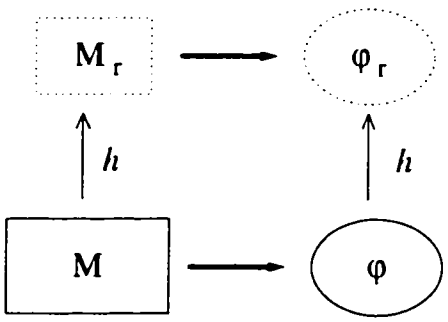


Figure 6.2: Symbolic Abstraction

would like to verify model M satisfying specification φ. Since M is too large to fit into a model checking tool, we can design a mapping function h and apply h on both M and φ to get M_r and φ_r. M_r is an abstract version of M and often much smaller than the actual model, and as a result, it is usually much simpler to verify properties at the abstract level.

The idea of the abstraction is to merge together all states in the Kripke structure that have the same labelling of abstract level atomic propositions. Suppose that we have a structure M whose definition is the same as the one given in last

section. In order to construct the reduced structure, we first change the labelling of the original structure. This is done by choosing an abstraction domain A and a mapping h from D to A. This determines a set of abstract atomic propositions AP. We now obtain a new structure M' that is identical to M except that the label function labels each state with a set of abstract atomic propositions from AP. The structure M' can be collapsed into a reduced structure $M_r = (S_r, I_r, \delta_r, \lambda_r)$ defined as follows:

- $S_r = \{L(s) | s \in S\}$. Thus, the set of states in the reduced structure is the set of all labelling of states of M.
- $s_r \in I_r$ if and only if there exists s such that $s_r = \lambda(s)$ and $s \in I$.
- $AP_r = AP$.
- $\delta_r(s_r, t_r)$ if and only if there exist s and t such that $s_r = \lambda(s), t_r = \lambda(t)$, and $\delta(s, t)$.
- Each s_r is just a set of atomic propositions, so $\lambda_r(s_r) = s_r$.

M_r is an abstract version of the Kripke structure of the original model. Each abstract state represents a set of concrete states that are merged together during the collapsing process. Note that by using this technique it is possible to determine whether formulas over the abstract atomic propositions AP are true in M. In practice, AP is chosen by the user so that it is possible to express the properties of M that needs to be checked. The main difficulty is that building M_r requires constructing M. When it is impossible to build M directly, one uses an implicit representation of M in terms of the formulas I and δ. Instead of constructing M, M_r is derived from these formulas [20]. One can also prove $M \preceq M_r$ [57]. Since the **ACTL** property is preserved in this relation, so we can verify **ACTL** properties under this framework.

For example, Figure 6.3 illustrates the abstraction procedure for a simple traffic light controller. The original model has one variable *color* that can take on values from the set $D = \{red, yellow, green\}$. Its states are labelled with atomic propo-

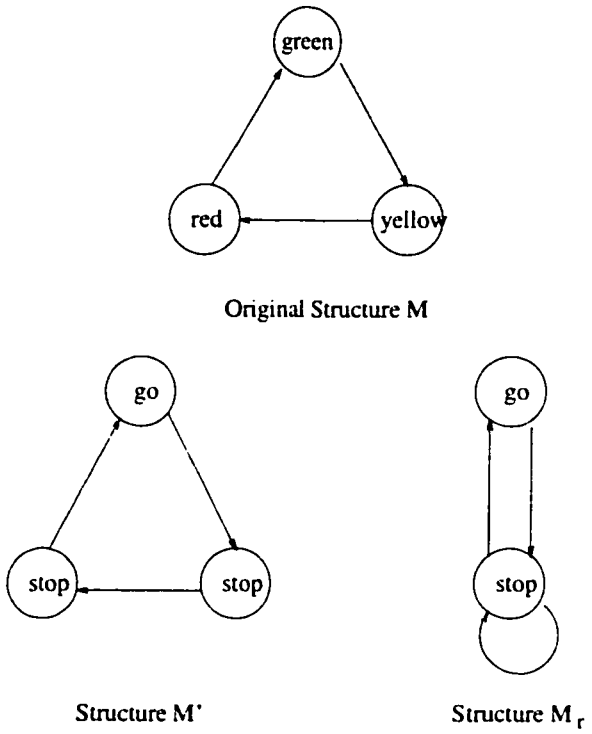


Figure 6.3: Traffic Light Example

sitions $'color = red'$,$'color = yellow'$, and $'color = green'$, which we abbreviate in the figure by *red*, *yellow*, and *'green'*, respectively. The structure M is obtained by choosing an abstract set of values $A = \{stop, go\}$ and a mapping function h defined by:

$$h(red) = stop, h(yellow) = stop, h(green) = go$$

The set of abstract atomic propositions is given by

$AP = \{'\widehat{color} = stop','\widehat{color} = go'\}$, where $\widehat{color}$ is the abstract variable of "*color*"

In the figure we use *stop* and *go* to abbreviate these atomic propositions. The reduced structure M_r results from merging together those states of M with the same labelling of abstract atomic propositions.

Generally, when we make such an abstraction, we have to find a smaller model, which can contain the behaviors of the concrete one. In this case, *false negative* becomes a key problem in the verification. Suppose we are given a module M and a property φ, the abstracted version of M is $\widehat{M}$. Usually, $\widehat{M} \models \varphi$ implies $M \models \varphi$, but $\widehat{M} \not\models \varphi$ does not implies $M \not\models \varphi$, namely the counter-example from the abstraction module $\widehat{M}$ is spurious. This is the so-called false negative problem. How to solve this problem is now an active research topic [25, 78, 19]. Here, we will introduce the counter-example guided abstraction refinement in [19].

In this approach, an algorithm is provided to determine whether an abstract counter-example is spurious. If the counter-example is not spurious, it is reported to the user and the verification is stopped. If the counter-example is spurious, the abstraction function must be refined to eliminate it. In the approach, one can identify the shortest prefix of the abstract counter-example that does not correspond to an actual trace in the concrete model. The last abstract state in this prefix is split into less abstract states so that the spurious counter-example is eliminated. Thus, a more refined abstraction function is obtained. Note that there may be many ways of splitting the abstract state; each determines a different refinement of the abstraction function. It is desirable to obtain the coarsest refinement, which eliminates the counter-example because this corresponds to the smallest abstract model that is suitable for verification. The approach is complete for the fragment of **ACTL*** which has counter-examples that are either paths or loops, i.e., it is guaranteed to either find a valid counter-example or prove that the system satisfies the desired property. In principle, the framework can be extended to all of **ACTL***.

Consider a program with only one variable with domain $D = \{1, \ldots, 12\}$.

Assume that the abstraction function h maps $x \in D$ to $\lfloor (x-1)/3 \rfloor + 1$. There are four abstract states corresponding to the equivalence classes $\{1,2,3\}$, $\{4,5,6\}$, $\{7,8,9\}$, and $\{10,11,12\}$. We call these abstract states $\hat{1}$, $\hat{2}$, $\hat{3}$, $\hat{4}$. The transitions between states in the concrete model are indicated by the arrows in Figure 6.4, where black dots denote non-reachable states. Suppose that we obtain an abstract

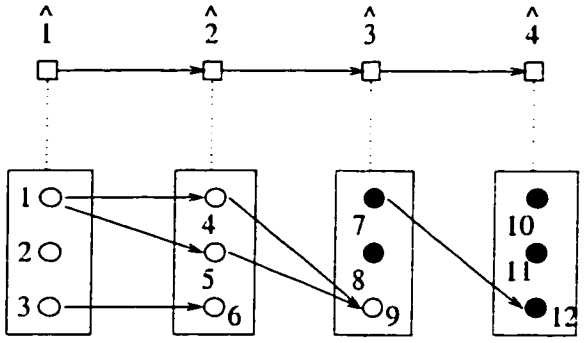


Figure 6.4: Abstracted Counter

counter-example $\hat{T} =< \hat{1}, \hat{2}, \hat{3}, \hat{4} >$. It is easy to see that $\hat{T}$ is spurious, and state $\hat{3}$ is the failure state because state 7 is unreachable in the concrete model, but in the abstract model, it is reachable. In order to eliminate this spurious counter-example, we can partition the concrete states into three types.

- The *dead-end state* 9 is reachable, but there are no outgoing transitions to the next state in the counter-example.
- The *bad state* 7 is not reachable but outgoing transitions cause the spurious counter-example. The spurious counter-example is caused by the bad state.
- The *irrelevant state* 8 is neither reachable nor bad.

The goal of the refinement framework is to refine the abstract states so that the dead-end states and bad states do not belong to the same abstract state. Then the spurious counter-example will be eliminated. Obviously, throughout the refinement, the size of the abstract model is becoming larger and the granularity is finer.

6.3 Summary

In this chapter, we overviewed two existing model reduction techniques, which are the foundation of our proposed syntactic model reduction approach, namely cone of influence and symbolic data abstraction. These techniques are based on different formalisms and each has its own advantages. For example, cone of influence can be done automatically but suffers from state space explosion, while symbolic data abstraction is more efficient but not automatic. Our goal in the next chapter is to propose a reduction method, which is efficient in typical applications and can be made automatic.

Chapter 7

Syntactic Model Reduction

7.1 Introduction

Cone of influence (COI) reduction decreases the size of the concrete system by focusing on the variables of the concrete system that are referred to in the property and eliminating variables that do not influence the variables of interest against the properties [55]. In this way, the property satisfaction is preserved, but the size of the model that needs to be verified is smaller. However, sometimes, there are still lots of redundant information in the COI reduced model. We can easily find a case in practice where a variable A depends on a variable B, but the *value* of variable B does not affect the *value* of variable A. For example in a two-input AND gate, if one of the inputs is set to zero, then no matter what value the other input takes, the output of the gate is always at zero.

Based on the above observation, we will give a refined dependency definition by examining the values of the variables that influence the truth of the property.

In the proposed approach, a system under verification is considered as a program for which abstract syntax and semantics are defined. The approach analyzes its syntactic structure, i.e., the *control flow diagram* of the programs [33]. Throughout the analysis, the dependencies of the state variables in the program are extracted.

Then, using a SAT solver, SATO [88] (Appendix B), we can partition the values of state variables in the program into *active values*, and *deactive values* according to their dependency to the properties. The deactive values then can be replaced by a typical *abstract* value, and thus the value domains of the variables are much smaller than the original ones. Accordingly, we can have a reduced program with respect to the abstracted variables. As we will see, after the above procedures are done, the state space of the reduced program is smaller than that of the original one, while the correctness of the properties is preserved. The proposed approach has the same flavor as the static analysis approaches such as symbolic data abstraction [20] or abstract interpretation [23]. However, we do not have to define the abstract domains. Throughout the analysis, we show that the approach is efficient in reducing the size of the datapath of a system with finite value domains. Because the reduction is based on the analysis of the control flow diagram, which is a canonical form of the program, it can be automated.

In order to achieve our goal, i.e., determining the active and deactive values, we will use two semantic functions adapted from [33]. Namely, the reachability condition RC_τ , associated with every path τ of the *Control Flow Diagram* (CFD), is a boolean condition under which this path is traversed. The state transformation function ST_τ computes the values of the program variables at the end of the path, provided that this path is traversed. These functions are obtained by backward induction over paths of the control flow diagram of the program. A problem to deploy this approach in practice is on how to solve RC_τ and get the true value assignment. Computing all the combinations of variables and their domains would be exponential. However, SAT [37] solvers are good at finding such variable assignments. Namely, we can use a SAT solver to compute RC_τ [71].

7.2 System Models

In this chapter, we model the various designs under verification by an abstract program including the syntax and the semantics. Generally, what we want to model is a reactive system, which reacts to the stimulus by changing the value of the variables in the program. The reaction of the system is *continuous* starting from an initial condition. For simplicity, we only define two types of statements of the program. One is assignment; the other is test. Using these two statements, we can simulate most of the cases in the computation.

The semantics of the program is defined by the Kripke structure [51] (Section 2.1), where each state is an execution result of the statements in the program. The transition between two states is the reflexive and transitive closure of two consecutive statements in the execution. Intuitively, assuming there is a clock signal in the program and all the statements are executed in each clock cycle, one can consider that a state is the valuations of all the variables in the program, which are sampled at one edge of the clock.

In the following, we define the syntax and the semantics respectively.

7.2.1 Abstract Program Syntax

Definition 7.2.1 *An* abstract program P *is specified by a tuple* (V, I, S), *where*

- $V = \{v_1, \ldots, v_n\}$ *is a finite, non-empty set of variables. Each variable* v_i, $i \in [1:n]$, *has an associated finite domain of values,* $dom(v_i)$.
- $I(V)$ *is the initial condition, specified as assignments on* V.
- S *is a set of statements which are partitioned into the assignment statements* S_a *and the test statements* S_t.

Given a program $P = (V, I, S)$, the *control flow diagram* (CFD) of this program is defined as follows.

Definition 7.2.2 *A Control flow diagram (CFD) of a program $P = (V, I, S)$ is a directed diagram (N, E, L_e) where*

- *N is a finite set of nodes labelled by the program counter locations.*
- *$E \subseteq N \times N$ is a finite set of edges.*
- *L_e is the edge labelling function $L_e : (E \rightarrow S)$ such that for every node n in the diagram either n is of out-degree 1 and the edge leaving n is labelled with assignment statement(s) or n is of out-degree 2 and the edges leaving n are labelled with test statement S_t and $\neg S_t$.*

There are three special nodes in a CFD, i.e., *Init*, ϵ, and ω. These nodes are part of the program syntax, where

- $Init \in N$ is the entry point of the program and marks the initialization block of the program, which has no predecessor.
- $\epsilon \in N$ is the beginning of the code. The edge to ϵ is marked by $I(V)$.
- $\omega \in N$, marks the end of the program text, where the code between ϵ and ω is in a forever loop.

In the test statement, "*if B then* $proc_1$ *else* $proc_2$" or "*while B then* $proc_1$", the *true action domain* of the test is the set of statements in $proc_1$, which will be executed when the test is **true**, and denoted by $AD_{S_t^+}$. The *false action domain* of the test is the set of statements in $proc_2$, which will be executed when the test is **false**, and denoted by $AD_{S_t^-}$.

The CFD of program P is actually its graphic presentation. So, P can be translated into $CFD(P)$ according to the definitions or vice versa.

Example 7.2.1 *To illustrate our purpose, we have chosen a small program P. In Figure 7.1, we give P and its CFD. In program P, the value domains of the variables,*

i, j, and k, are $\{0,1,2,3\}$, respectively. The initial condition is "$i = j = k = 0$", which is the starting point of program P. There are two kinds of statements in P, assignment and test. The statements between ϵ and ω are in a forever loop, namely when statement L_2 is finished, statement ϵ will start again automatically. In the CFD, nodes are marked by the program counter locations ($\epsilon, L_0, L_1, L_2, \omega$) and edges are marked by the statements in the program. Since the code between ϵ and ω is in a forever loop, the edge from ω to ϵ is a "forever" transition.

```
// Variable declaration
i, j, k : {0,1,2,3};
Init:     i = 0;
          j = 0;
          k = 0;
epsilon:  i = (i + 1) mod 4;
L0:       j = (j + 1) mod 4;
L1:       if (i == 3)
L2:         k = 1;
omega:
```

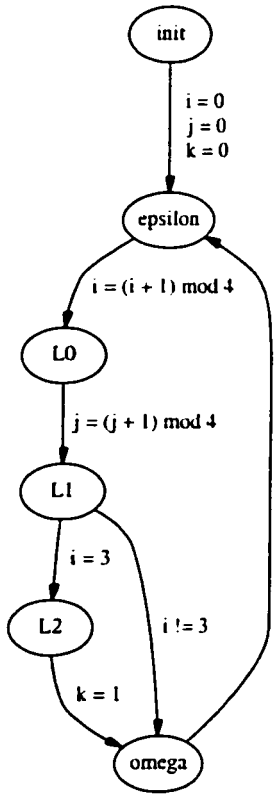


Figure 7.1: Program P and its CFD

7.2.2 Abstract Program Semantics

Let $\vec{v} = \{v_1, \ldots, v_k\}$ be a vector of variables taking their values in a *universe* $U_{\vec{v}} = \{dom\{v_1\}, \ldots, dom\{v_k\}\}$. Semantically, we refer to the valuation of the variables $v_1 = e_1, \ldots, v_k = e_k$, where $e_i \in dom(v_i)$, as a *state* σ, and these variables v_i

are often referred to as *state variables*. We use the notation $\sigma[\![v_i]\!]$ to denote the value of state variable v_i in state σ. Similarly, for an expression x (over state variables in the domain of state σ), we use $\sigma[\![x]\!]$ to denote the value of x in the state σ. The special brackets '$[\![$' and '$]\!]$' emphasize the fact that the entity enclosed between them (v_i or x in the above cases) is of a syntactic nature.

The *initial condition* $I(V)$ serves as the entry point of the program just like the reset value of a circuit, where the computation starts. In the initial condition, all the variables have to be initialized, however, this is not an essential restriction.

The *assignment* statement is denoted by S_a, where an assignment $\overrightarrow{v_1} = T(\overrightarrow{v_2})$ is represented as a function T from U_{v_2} into U_{v_1}. In the simple case, $v_i = T(\overrightarrow{v_2})$. In the above assignment statement, $\overrightarrow{v_1} = T(\overrightarrow{v_2})$ can be multiple assignment, where the current value of the expression $T(\overrightarrow{v_2})$ can be assigned to the variable v_i simultaneously for all i. For example, $(x_1, x_2) = (x_1 + 1, x_1 + x_2)$ assigns to x_1 the successor of its value and to x_2 the sum of the old value of x_1 and x_2. However, two assignment statements, "$\overrightarrow{v_1} = T(\overrightarrow{v_2})$; $\overrightarrow{v_3} = T'(\overrightarrow{v_4})$", are sequential, namely statement $\overrightarrow{v_3} = T'(\overrightarrow{v_4})$ can be executed only after statement $\overrightarrow{v_1} = T(\overrightarrow{v_2})$. For example, two assignment statements, "$x_1 = x_1 + 1; x_2 = x_1 + x_2$", x_1 is assigned the successor of its value and x_2 the sum of the *new* value of x_1 and x_2.

The *test* statement is denoted by S_t, where a test statement, "$if(B(\overrightarrow{v}))$" or "$while(B(\overrightarrow{v}))$", is represented as a function B from $U_{\overrightarrow{v}}$ into $\{\mathbf{true}, \mathbf{false}\}$. The test statement is a positive test in a state σ when $\sigma[\![B(\overrightarrow{v})]\!] = \mathbf{true}$, or a negative test when $\sigma[\![B(\overrightarrow{v})]\!] = \mathbf{false}$. The computation branches according to the value of the test function.

The semantics of program P is defined in terms of *configurations*. A configuration of a program is a pair $c = \langle \sigma, \lambda \rangle$. The state σ is a visible component and assigns values to program variables. The *counter location* λ is a hidden component and its value corresponds to a current statement to be executed.

Next, we define the particular transition relation "$\rightarrow$" for configurations.

Definition 7.2.3 *Let $c = \langle \sigma, \lambda \rangle$ and $c' = \langle \sigma', \lambda' \rangle$ be two configurations. We write $c \rightarrow c'$ if and only if the following hold.*

- *If λ is the Init node, then $\lambda' = \epsilon$ and $\sigma' = \sigma$.*
- *If λ is the ω node, then $\lambda' = \epsilon$ and $\sigma' = \sigma$.*
- *If λ is an assignment $S_a : \overrightarrow{v_1} = T(\overrightarrow{v_2})$, where $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$ are two vectors and T is a function from $U_{\overrightarrow{v_2}}$ into $U_{\overrightarrow{v_1}}$, then $\lambda' = suc(\lambda)$ and $\sigma'[\![\overrightarrow{v_1}]\!] = \sigma[\![T(\overrightarrow{v_2})]\!]$, where $suc(\lambda)$ means the successor of λ in a CFD.*
- *If λ is a test statement $S_t : if(B(\overrightarrow{v}))$ or $while(B(\overrightarrow{v}))$, then $\sigma' = \sigma$, and either $\lambda' = suc^+(\lambda)$, in the case $\sigma[\![B(\overrightarrow{v})]\!]$ is true or $\lambda' = suc^-(\lambda)$ in the case $\sigma[\![\neg B(\overrightarrow{v})]\!]$ is true, where $suc^+(\lambda)$ and $suc^-(\lambda)$ means the true or false successor of λ in a CFD, respectively.*

  We will use "$\rightarrow^*$" to denote the reflexive transitive closure of "$\rightarrow$".

Generally, "$\rightarrow$" is the transition from one statement to the next statement in the execution. However, "$\rightarrow^*$" is the transition after n continuous statements. During these n steps, the execution may either include arbitrary number of steps of $\rightarrow$, or do nothing ($n = 0$).

Definition 7.2.4 *The semantics of the program $P = (V, I, S)$ is a Kripke structure $\mathcal{K}(P) = (\Sigma, \langle \sigma_0, Init \rangle, \rightarrow, L)$, where*

- *Σ is a set of configurations, where $\Sigma = \{\langle \sigma', \epsilon \rangle : \langle \sigma, \epsilon \rangle \rightarrow^* \langle \sigma', \epsilon \rangle\} \cup \{\langle \sigma_0, \epsilon \rangle : \langle \sigma_0, Init \rangle \rightarrow \langle \sigma_0, \epsilon \rangle\}$.*
- *$\langle \sigma_0, Init \rangle \in \Sigma$ is an initial configuration such that $\sigma[\![I(V)]\!]$ is true.*
- *$\rightarrow$ is a transition relation defined in Definition 7.2.3.*
- *a labelling function $L : (\Sigma, 2^{AP})$ associates a non-empty set of atomic propositions (AP) with a configuration $c \in \Sigma$.*

Example 7.2.2 *The Kripke structure of the program in Example 7.2.1 is shown in Figure 7.2. The double circle is the initial state where the labels of the initial states are decided by the initial assignments. Here, since in the Kripke structure, the value of* λ *is always* ϵ*, we omit* ϵ *in the label of the configurations, namely* $< (0,0,0), \lambda >$ *becomes* $(0,0,0)$*. In this example, the initial condition is "*$i = j = k = 0$*", so the initial state is "*$i = j = k = 0$*". The labels of the other states are decided by the valuation of the variables in the program when program counter is* ϵ*. For example, if the current state is "*$i = j = 1, k = 0$*", according to the program, statement* L_2 *will not be executed because the test condition does not hold, so* k *keeps at* 0*. However, in this case, statement* ϵ *and* L_0 *are executed, the consequential results are "*$i = j = 2$*". When the execution gets to* ω*, the valuation of the variables are "*$i = j = 2, k = 0$*". So, the next state of "*$i = j = 1, k = 0$*" is "*$i = j = 2, k = 0$*".*

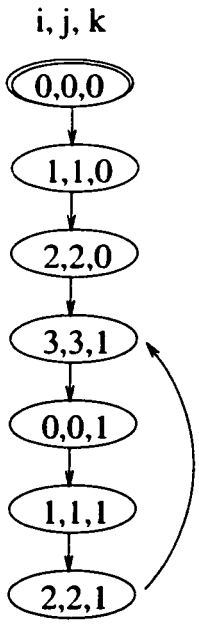


Figure 7.2: Kripke Structure of Example 7.2.1

7.3 Data Dependency Reduction

Data dependency defines the relationship between two state variables. If one variable affects the values of another variable during the computation, we call these two variables *dependent*. Generally, in the statements of a program, there are two types of dependency. One is the statement with direct data dependency; the other is that with indirect data dependency. Direct data dependency means that the value of one variable will directly affect the value of the other variable.

Definition 7.3.1 *Direct data dependency is the data dependency in the assignment statement* $Sa : \overrightarrow{v_1} = T(\overrightarrow{v_2})$, *where* $\overrightarrow{v_1}$ *directly depends on* $\overrightarrow{v_2}$.

For example, in the assignment statement "$x = y + 1$", the value of the *RHS* (Right Hand Side) variable y will affect the value of *LHS* (Left Hand Side) variable x directly.

Indirect data dependency means that the value of one variable will affect the value of the other variable via another variable.

Definition 7.3.2 *Indirect data dependency is the data dependency in the test statements, "*$if(B(\overrightarrow{v}))$*" or "*$while(B(\overrightarrow{v}))$*". Variables in* $\overrightarrow{v}$ *indirectly affects the LHS variables in* $AD_{S_t^+}$ *and* $AD_{S_t^-}$.

For example, in the "$if(z)$ *then* $w = y+1$ *else* $w = y-1$" statement, the value of z will affect the value of the LHS variable w indirectly via variable y. Another example from hardware, such as a D flip-flop as shown in Figure 7.3, the relation between D and Q is the direct dependency, and the relation between Clr and Q is the indirect dependency.

A Data Dependency Diagram (DDD) of program P is a directed diagram which records the dependency between the variables in the program. The data dependency diagram is defined as follows:

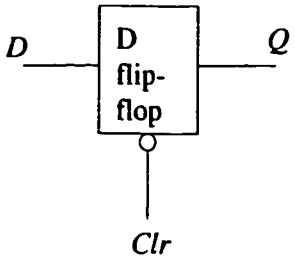


Figure 7.3: A D Flip-flop

Definition 7.3.3 *The data dependency diagram of a program* (V, I, S) *is represented as a directed graph* (D, L_d, X)*, where*

- D *is the set of nodes.*
- L_d *is a one-to-one function* $L_d : (D, V)$ *which labels each node by a variable.*
- $X \subseteq D \times D$ *is the set of transitions between the nodes to mark the dependency via directions,* $(d_i, d_j) \in X$ *if and only if there is a direct dependency between* v_i *and* v_j*, i.e.,* $v_j = T(v_i)$*, or there is an indirect dependency between* v_i *and* v_j*, i.e.,* v_i *in* $B(\vec{v})$*) and* v_j *in the LHS of* $AD_{S_t^+}$ *or* $AD_{S_t^-}$*. Namely* v_i *is either RHS of* S_a *or in* $B(\vec{v})$*;* v_j *is either LHS of* S_a *or LHS in* $AD_{S_t^+}$ *or* $AD_{S_t^-}$ *with respect to* $B(\vec{v})$*.*

Example 7.3.1 (DDD of the search program) *In Example 7.2.1, the DDD of the program is shown in Figure 7.4, where only* i *affects* k*. Because* i *is in the test condition of statement* L_1 *and* k *is the LHS of* L_2*, this is an indirect dependency.*

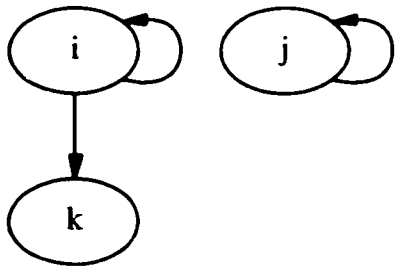


Figure 7.4: DDD of Example 7.2.1

Since in each verification run, a property is always a partial specification, and not all the state variables in the implementation are involved in the verification, we can prune those unnecessary state variables in verification. Thus, the system to be verified will be smaller in terms of the number of state variables. This is exactly the idea of *cone of influence* reduction [55, 22]. Here, the cone C is created by removing all the variables where the nodes marked by these variables cannot reach the nodes marked by the variables of interest in DDD. Since C is the subset of the variables V of a model and includes all the variables of interest against specification φ, it is an abstraction of the variables in the model.

Theorem 7.3.1 [20] *Let f be a* **CTL*** *formula with atomic proposition in C, and M is the concrete model, $\widehat{M}$ is the COI reduced model. Then*

$$M \models f \Leftrightarrow \widehat{M} \models f$$

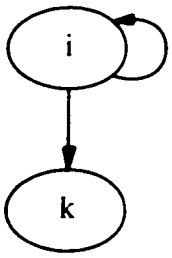


Figure 7.5: COI Reduced DDD

Example 7.3.2 (COI reduced DDD and CFD) *In Example 7.2.1, assuming we want to prove a property of variable k, the DDD of the COI reduced program is shown in Figure 7.5, where only i affects k and j is removed because the node marked by j cannot reach the node marked by k via a path in the diagram.*

The reduced program is shown in the left of Figure 7.6 and the corresponding CFD is shown in the middle. The state space of the program after the COI reduction is shown in the right.

```
// Variable declaration
i, k : {0,1,2,3};
Init:     i = 0;
          k = 0;
epsilon:  i = (i + 1) mod 4;
L1:       if (i == 3)
L2:         k = 1;
omega:
```

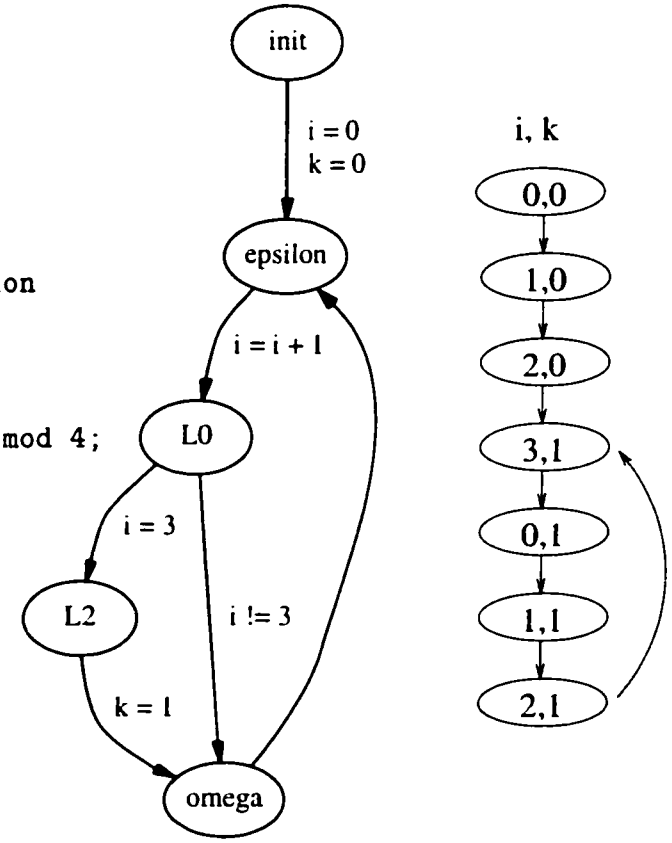


Figure 7.6: COI Reduced Program and its CFD and State Space

7.4 Reachability Condition and State Transform

In this section, we introduce two notations used in the following reduction approach, i.e., *reachability condition* and *state transformation*. These two notations are obtained from the CFD.

For a sequence of nodes in the CFD, $N_i \stackrel{E_{i+1}}{\rightarrow} N_{i+1} \stackrel{E_{i+2}}{\rightarrow} \ldots \stackrel{E_{i+k}}{\rightarrow} N_{i+k}$, where $k \geq 1$, we say this is a *finite path* in the diagram. We use the notation $expr[v \leftarrow e]$ for the expression that every occurrence of v in $expr$ is replaced by the value e. For example, "$x + y[x \leftarrow 1, y \leftarrow 1] = 2$" means that x and y are replaced by constant 1 respectively, and the result of the expression is 2. By convention, **true**$[v \leftarrow e]$ = **true** and **false**$[v \leftarrow e]$ = **false**.

Definition 7.4.1 [33] *Let τ be a finite path in the CFD.*

- Reachability condition *guarantees that the control will traverse τ. It is denoted by $RC_\tau(\vec{v})$, where $\vec{v}$ is the variable vector of interest.*
- State transformation *is the final state of $\vec{v}$ obtained if control indeed traverses τ with $\vec{v}$. It is denoted by $ST_\tau(\vec{v})$.*

$RC_\tau(\vec{v})$ and $ST_\tau(\vec{v})$ are obtained by backward induction over τ as follows.

Let $\tau = N_0 \stackrel{E_1}{\rightarrow} \ldots \stackrel{E_{k-1}}{\rightarrow} N_k$ be a finite path in the CFD of length $k+1$, for some natural number k. We first define $RC^m_\tau(\vec{v})$ and $ST^m_\tau(\vec{v})$, the corresponding characteristics of the suffix $N_m \stackrel{E_{m+1}}{\rightarrow} \ldots \stackrel{E_{k-1}}{\rightarrow} N_k$ of τ, by induction on m going down from k to 0.

- *Induction basis*

  $RC^k_\tau(\vec{v}) = true$, $ST^k_\tau(\vec{v}) = \vec{v}$

  In other words, being at the end of τ to traverse τ implies an identically true reachability condition and an identity state transformation.

- *Induction assumptions*

  Suppose that $RC_\tau^{m+1}(\overrightarrow{v})$ and $ST_\tau^{m+1}(\overrightarrow{v})$ have already been defined for $0 \leq m < k$

- *Induction step*

  We define $RC_\tau^m(\overrightarrow{v})$ and $ST_\tau^m(\overrightarrow{v})$ according to the statements at edge E_m.

  - *Initialization*: $RC_\tau^m(\overrightarrow{v}) = (RC_\tau^{m+1}(\overrightarrow{v}))[\overrightarrow{v} \leftarrow I(V)]$, $ST_\tau^m(\overrightarrow{v}) = ST_\tau^{m+1}(\overrightarrow{v})[\overrightarrow{v} \leftarrow I(V)]$. Thus, the initial statement is captured by an assignment of new value to the variables.
  - *Assignment*: $RC_\tau^m(\overrightarrow{v}) = (RC_\tau^{m+1}(\overrightarrow{v}))[\overrightarrow{v} \leftarrow T(V)]$, $ST_\tau^m(\overrightarrow{v}) = ST_\tau^{m+1}(\overrightarrow{v})[\overrightarrow{v} \leftarrow T(V)]$. Thus, the effect of an assignment is captured by substitution.
  - *Positive test*: $RC_\tau^m(\overrightarrow{v}) = (RC_\tau^{m+1}(\overrightarrow{v})) \wedge B(\overrightarrow{v})$, $ST_\tau^m(\overrightarrow{v}) = ST_\tau^{m+1}(\overrightarrow{v})$. Thus a test does not change the state and remembers the condition.
  - *Negative test*: $RC_\tau^m(\overrightarrow{v}) = (RC_\tau^{m+1}(\overrightarrow{v})) \wedge \neg B(\overrightarrow{v})$, $ST_\tau^m(\overrightarrow{v}) = ST_\tau^{m+1}(\overrightarrow{v})$. This is similar to the positive test case.

Finally, we define $RC_\tau(\overrightarrow{v}) = RC_\tau^0(\overrightarrow{v})$, and $ST_\tau(\overrightarrow{v}) = ST_\tau^0(\overrightarrow{v})$.

Example 7.4.1 *Consider the program in Example 7.2.1. If we set τ as a path $Init \rightarrow \epsilon \rightarrow L_0 \rightarrow L_1 \rightarrow \omega$, the reachability condition and the state transformation are as follows.*

- $RC_\tau^\omega(i,j,k) = \mathbf{true}$, $ST_\tau^\omega(i,j,k) = (i,j,k)$.
- $RC_\tau^{L_1}(i,j,k) = \mathbf{true} \wedge (i \neq 3) = (i \neq 3)$, $ST_\tau^{L_1}(i,j,k) = (i,j,k)$.
- $RC_\tau^{L_0}(i,j,k) = (i \neq 3)[j \leftarrow (j+1)] = (i \neq 3)$, $ST_\tau^{L_0}(i,j,k) = (i,j,k)[j \leftarrow (j+1)] = (i, j+1, k)$.
- $RC_\tau^{\epsilon}(i,j,k) = (i \neq 3)[i \leftarrow (i+1)] = (i \neq 3)$, $ST_\tau^{\epsilon}(i,j,k) = (i, j+1, k)[(i \leftarrow (i+1))] = (i+1, j+1, k)$.

- $RC_\tau^{Init}(i,j,k) = (i \neq 3)[i \leftarrow 0, j \leftarrow 0, k \leftarrow 0] = \textbf{true}$, $ST_\tau^{Init}(i,j,k) = (i, j+1, k)[i \leftarrow 0, j \leftarrow 0, k \leftarrow 0] = (1,1,0)$.

From the above induction, we know $RC_\tau(i,j,k) = \textbf{true}$ *and* $ST_\tau(i,j,k) = (1,1,0)$ *which means that node* ω *can be reached from node Init by path* $Init \rightarrow \epsilon \rightarrow L_0 \rightarrow L_1 \rightarrow \omega$ *and after the traverse the valuations of the variables are* $(1,1,0)$, *respectively.*

```
// Variable declaration
int x;
Bool y,z;

// Initialization
Init:     x = 1;
          y = 1;
          z = 1;

epsilon: if (y <= x)
L0:        y = y + z;
          else
L1:        x = x + z;
         endif
omega:
```

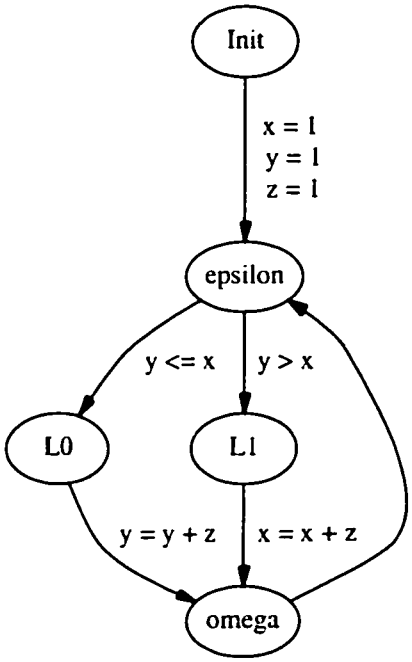


Figure 7.7: An RC/ST Example

Example 7.4.2 *Consider the program in the left of Figure 7.7. The CFD of this program is shown in the right.*

If we set τ *as a path* $\epsilon \rightarrow L_0 \rightarrow \omega$, *the reachability condition and the state transformation are as follows.*

- $RC_\tau^{\omega}(x,y,z) = \textbf{true}$, $ST_\tau^{\omega}(x,y,z) = (x,y,z)$
- $RC_\tau^{l_0}(x,y,z) = \textbf{true}[y \leftarrow (y+z)] = \textbf{true}$, $ST_\tau^{l_0}(x,y,z) = (x,y,z)[y \leftarrow (y+z)] = (x, y+z, z)$

- $RC^{\epsilon}_{\tau}(x, y, z) = \textbf{true} \wedge (y \leq x) = (y \leq x)$, $ST^{\epsilon}_{\tau}(x, y, z) = (x, y + z, z)$

From the above induction, we know $RC_{\tau}(x, y, z) = (y \leq x)$ *and* $ST_{\tau}(x, y, z) = (x, y+z, z)$. *The actual values of* $RC_{\tau}(x, y, z)$ *and* $ST_{\tau}(x, y, z)$ *depend on the present values of* x, y, *and* z *at node* ϵ. *In the computation right after the initialization* $x = 1$, $y = 1$ *and* $z = 1$, $RC_{\tau}(x, y, z) = \textbf{true}$ *and* $ST_{\tau}(x, y, z) = (1, 2, 1)$.

7.5 Deactive Variables Reduction

As we mentioned, even after COI reduction, the reduced model still has some redundant information, which will create a redundant state space. In the context of model checking, this redundancy means much more CPU time and memory usage in the verification. In this section, we give a refined dependency definition of the COI approach. In the following, we consider whether or not the change of the value of a variable v_i will lead to a change of the value of another variable v_j. If so, then variable v_j is indeed depending on variable v_i. For example, v_j is the variable in a property. If the value change of v_i will alter the value of v_j, then variable v_i will definitely affect the truth of the property, else even v_i and v_j are dependent with respect to the COI dependency, v_i is a redundant variable according to the property.

The basic approach to deploy the above idea is to partition the value domain of the variables in the program. In the following, we will partition the value domain into *active values* and *deactive values*. The active values are those values, which will affect the variables of interest, while the deactive values do not affect the variables. The partition is based on the analysis of the two notations of control flow diagram (CFD), i.e, reachability condition RC and state transform ST.

Now, we are able to define our partition of value domains. We consider first the *key nodes*, which are nodes of the CFD that directly influence the specification. Next, we compute the reachability condition and state transformation function of the path starting from a node in the CFD and ending at a key node.

Definition 7.5.1 *Let v be the variable of interest in the property, n be a node in the CFD and e be the outgoing edge of n. We say that n is a* key node *if and only if* $L_e(e) \triangleq v = T(\overrightarrow{v})$.

For example, if i is the variable of interest, then a node n in the CFD with the outgoing edge labelled by $i = T(\overrightarrow{v})$ is the key node. In the keynode, the value of i is to be changed, so the correctness of the properties of i will be affected. However, i is not the only variable in the program which can affect the correctness of the properties, because the variables may affect the correctness of the properties indirectly by affecting the value of i.

In the program, a variable j may affect variables i if there is a path such that the CFD node marked by the label of i is reachable from the node marked by the label of j. For example, there are two continues assignment statements in program P as shown in the left of Figure 7.8. The CFD of the program is on the right of

```
L1:  j = j +1;
L2:  i = j +1;
...
```

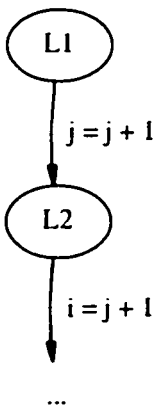


Figure 7.8: Dependency Example 1

the figure. The value change of j will affect the value of i since the CFD node L_2 is reachable from L_1 via the path "$L_1 \rightarrow L_2$". On the other hand, consider the piece of program in the left of Figure 7.9, where the corresponding CFD is shown in the right of the figure. The CFD node L_3 is only reachable from L_1 when $j \geq 10$, which means that all the values of j less than 10 cannot affect the value of i in this piece

```
L1:  j = j +1;
L2:  if (j > 10)
L3:     i = j +1;
...
```

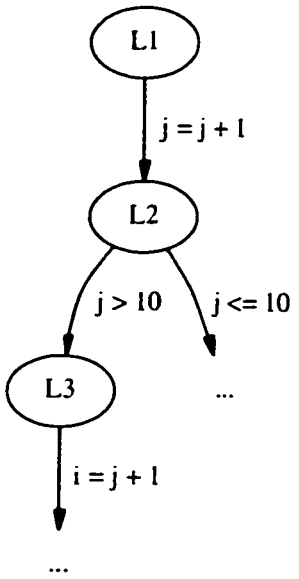


Figure 7.9: Dependency Example 2

of code. Moreover, we can consider that only the values of j greater than 10 will affect the properties where i is the variable of interest, so we will keep this partition of value in the program, i.e., active values, and abstract the value of j less than 10, i.e., deactive values.

Definition 7.5.2 *Let v be a variable among the dependency list of interest variables, e the value of v and τ be a path leading to a key node.*
We say that the value e is active *if and only if $RC_\tau[v \leftarrow e]$ is true or $ST_\tau(v) = e$. Similarity, we say that the value e is* deactive *if and only if $RC_\tau[v \leftarrow e]$ is false.*

To use above definition in the practice, a problem is on how to find all the true value assignments for the reachability condition. Since SAT solvers are good at finding variable assignments which make the CNF (Canonical Normal Form) propositional formula true, we can use a SAT solver, here SATO [88], to get all the values satisfying formula $RC_\tau[v \leftarrow e] = \mathbf{true}$, which are the *active* values with respect to the RC. However, since SAT-solvers expect their input to be a propositional formula in the CNF form, we still need to convert the reachability condition into CNF. Besides,

when we do the conversion, since SATO does pure literal simplification [89], we have to keep a literal and its opposite in the CNF. Generally and during computations, a variable v may have both active and deactive values with respect to the key nodes. Hence, the domain values of v, i.e., $dom(v)$, is partitioned into disjoint subsets ACTIVE (v), eventually empty, of active values and DEACTIVE (v), eventually empty, of deactive values, where

- if ACTIVE $(v) = \emptyset$, then v is a dead variable and DEACTIVE $(v) = \{e\}$, where $e \in dom(v)$ and e is a typical value of $dom(v)$.
- if ACTIVE $(v) \neq \emptyset$ and DEACTIVE $(v) \neq \emptyset$, then v is a partial dead variable and DEACTIVE $(v) = \{e\}$, where e is a typical value of DEACTIVE (v).
- if DEACTIVE $(v) = \emptyset$, then v is a live variable and ACTIVE $(v) = dom(v)$.

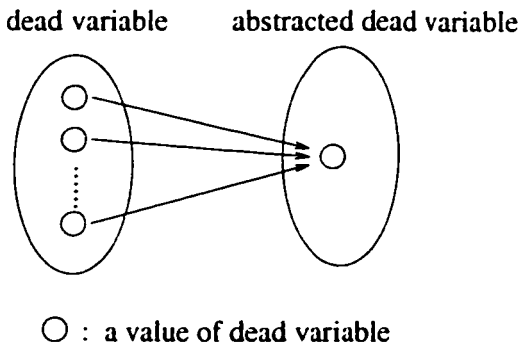


Figure 7.10: Abstraction of Dead Variables

From the above analysis, the variables of a program is partitioned into subsets with respect to the variable of interest, i.e., a set of live variables, that of dead variables, and that of partial dead variables. Every value of the active variables will affect the value of the variable of interest while that of the deactive variables cannot. In model checking, the variable of interest is the variable in the property. The value of dead variables cannot affect the correctness of the property. Consequently we can abstract these values and preserve the correctness of the property. This abstraction

is shown in Figure 7.10, where all the values in a dead variable are mapped into a single value. This single value can be any value in the domain of the dead variable.

In the case of partial dead variables, the active values are kept while the deactive values are abstracted as shown in Figure 7.11, where the active values are kept while the deactive values are mapped into a single value. This single value can be any value among the deactive values.

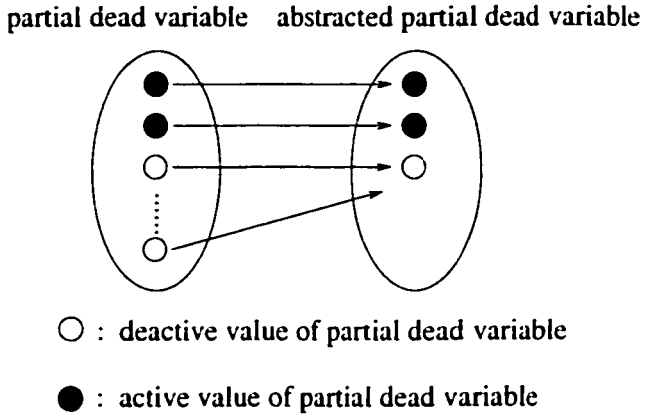


Figure 7.11: Abstraction of Partial Deactive Variables

In the case of live variables, all the values are kept as shown in Figure 7.12. In this case, no reduction is achieved.

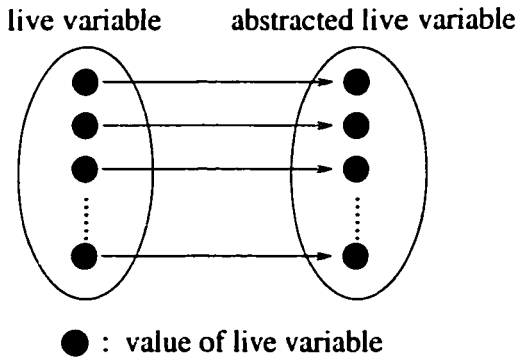


Figure 7.12: Abstraction of Active Variables

Given the above reduction rules, we can obtain a reduced program from the original program as follows.

Definition 7.5.3 *$P = (V, I, S)$ is a program. The reduced program $\widehat{P} = (\widehat{V}, \widehat{I}, \widehat{S})$ is:*

- *$\widehat{V} = V$ is a finite, non-empty set of variables. For each $v_i \in \widehat{V}$, $dom(v_i)$ =*ACTIVE (v_i)∪ DEACTIVE (v_i).
- *$\widehat{I}(\widehat{V})$ is equal to $I(V)[e_i \leftarrow$*DEACTIVE $(v_i)]$ *for each v_i such that $e_i \notin$*ACTIVE (v_i).
- *$\widehat{S}$ is obtained as follows: (1) if $B(\overrightarrow{v})$ or while $B(\overrightarrow{v})$ statements are obtained like the initial condition. (2) An abstract assignment statement $\widehat{S_a}$ is defined non-deterministically by $v_i =$*ACTIVE (v_i)∪ DEACTIVE (v_i).

In the above reduced program, the variables in the program have new domains, i.e., ACTIVE (v_i)∪ DEACTIVE (v_i). Since DEACTIVE (v_i) is actually the reduced set of deactive values, the abstract domain of v_i obviously is smaller than its original domain. Accordingly, because some values are missed from the program, the initial condition, assignment and test statement have to be revised as well. For the assignment statement, the new value of LHS variables is non-deterministically its active values and the typical deactive value. For test statement, if there is a deactive value in the test condition, then this value is replaced by the typical deactive value.

In the following, we will prove that the Kripke structure of the reduced program $\mathcal{K}(\widehat{P})$ simulates the original Kripke structure $\mathcal{K}(P)$ so that the reduction preserves **ACTL** properties.

Definition 7.5.4 *Let $B \subseteq \Sigma \times \widehat{\Sigma}$ be the relation defined as follows:*

$$((e_1, \ldots, e_n), (\widehat{e}_1, \ldots, \widehat{e}_n)) \in B$$

$$\Leftrightarrow$$

$$((\widehat{e}_i = e_i) \wedge (\widehat{e}_i \in \text{ACTIVE } (v_i))) \vee ((\widehat{e}_i \neq e_i) \wedge (\widehat{e}_i \in \text{DEACTIVE } (v_i)))$$

Theorem 7.5.1 *B is a simulation relation between the models $\mathcal{K}(P)$ and $\mathcal{K}(\widehat{P})$, namely $\mathcal{K}(P) \preceq \mathcal{K}(\widehat{P})$.*

PROOF:

1. For the initial configuration $c_0 \in \Sigma$ there exist an initial $\widehat{c}_0 \in \widehat{\Sigma}$. Since the abstract initial condition may differ only on deactive values, it follows trivially that $(c_0, \widehat{c}_0) \in B$

2. Let $c_1 \rightarrow c_2$ be a transition in $\mathcal{K}(P)$. We show that there is a transition $\widehat{c}_1 \rightarrow \widehat{c}_2$ such that $(c_2, \widehat{c}_2) \in B$, where $(c_1, \widehat{c}_1) \in B$. If the corresponding statement of λ_1 is an assignment, since the abstract assignment statement is nondeterministic, there is always a transition from $\widehat{c}_1 \rightarrow \widehat{c}_2$ and $(c_2, \widehat{c}_2) \in B$ as required. For the *if* and *while* abstract statements, we update only inside their body the deactive values. Hence there is also a transition from $\widehat{c}_1 \rightarrow \widehat{c}_2$ and $(c_2, \widehat{c}_2) \in B$ as required.

3. By using steps 1 and 2, we can conclude that B is a simulation relation between $\mathcal{K}(P)$ and $\mathcal{K}(\widehat{P})$. □

Finally, since the reduced program simulates the original program, according to Theorem 7.5.2, the **ACTL** properties are preserved.

Theorem 7.5.2 [20] *Suppose $M \preceq M'$, where M and M' are two models. Then for every* **ACTL** *formula φ, $M' \models \varphi \Rightarrow M \models \varphi$*

Example 7.5.1 (How the reduction works) *Considering the program P of Example 7.2.1, we want to verify a property about variable k. We can first apply COI reduction to find that variable j is irrelevant and thus can be removed. We will apply our approach based on the COI reduced program.*

In the CFD, node L_2 is the key node since the statement $k = 1$ changes the value of k. Consider the value 0 of i. i is 0 at node ϵ. Through the backward reachability analysis from key node L_2 to ϵ, we find that $RC_{\epsilon \rightarrow L_2} =$ "$i = 3$". According to Definition 3 and using SATO, $RC_{\epsilon \rightarrow L_2}[i \leftarrow 0] =$ "$0 = 3$" which is false. So, the value 0 of i is a deactive value. In a similar way, we find that values 1 and 2 do not

satisfy RC, so 1 *and* 2 *are deactive values* [1]. *The value* 3 *of* i *is active since* L_2 *is reachable in path* ϵ, L_1, L_2 ($RC_{\epsilon \to L_2}[i \leftarrow 3] = true$) *while* $ST_{\epsilon \to L_2}(i) = 3$. *Hence, we can use abstract value domain* $\{2, 3\}$ *instead of its concrete value domain* $\{0, 1, 2, 3\}$. *Using the same approach, the abstract value domain of variable* k *is* $\{0, 1\}$.

In Figure 7.13, the reduced program $\widehat{P}$ *of* P *is shown on the left side, where "*ND*" means nondeterministic assignment. In the middle, the state space of the program* P *after COI reduction has 7 states. On the right side, the reduced state space of* $\widehat{P}$ *has three states.*

```
// Variable declaration
i : {2,3};
k : {0,1};
// Initialization
Init:     i = 2;
          k = 0;
epsilon:  i = ND(2,3);
L1:       if (i == 3)
L2:         k = 1;
omega:
```

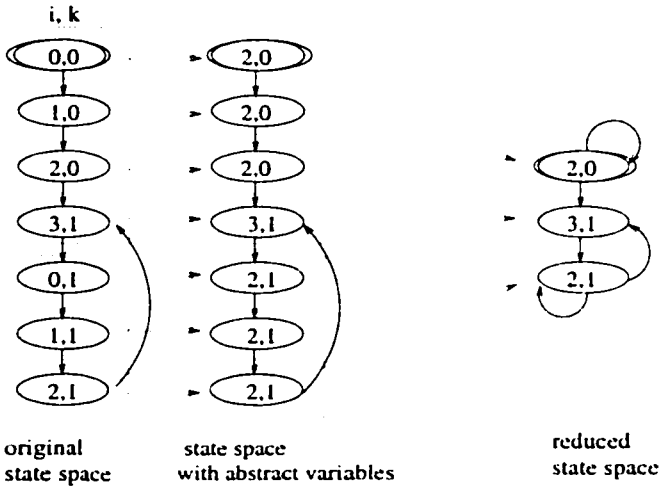


Figure 7.13: Reduced Program $\widehat{P}$ and its Kripke structure

Example 7.5.2 (Reduction of the counter) *Consider the program in the left of Figure 7.14. The property to verify is* $y \geq 0$. *The CFD of the program is shown in the right of the Figure.*

In the CFD, the key node is L_1 *because the outgoing edge is labelled by* $j = i$. *Since the RC to the key node must contain a condition "*$i \geq 1000$*", the key node is unreachable from* $\{\epsilon, L_0\}$ *with respect to the value* $\{1, \ldots, 999\}$ *of* i, *So,* $\{1, \ldots, 999\}$ *are the deactive values of* i, *yet the value* 1000 *of* i *is an active value since from path*

[1] *In practice, we input the corresponding two bits propositional formula of* $i = 3$ *to SATO, SATO outputs that in order to satisfy the formula, both bits of* i *have to be* 1.

$\{\epsilon, L_1, \omega\}$, *the key node is reachable with respect to the value* 1000 *of* i. *Hence, the abstract domain of* i *is* $\{0, 1000\}$. *However, in this example, the domain of variable* j *cannot be reduced since* L_1 *is reachable with respect to any value of* j.

```
// Variable declaration
{0..10000} i, j;

// Initialization
Init:    i = 0;
         j = 0;

epsilon: while (i < 1000) begin
L0:         i = i+1;
         endwhile
L1:      j = i;
omega:
```

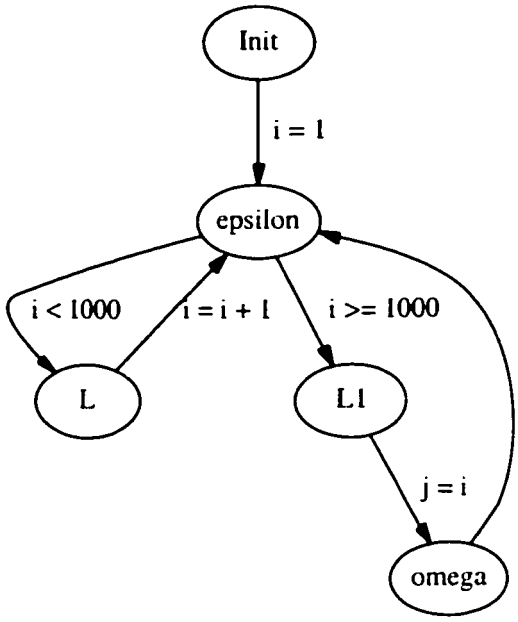


Figure 7.14: The Counter Program

7.6 Applications

7.6.1 A Forwarding Table Lookup Processor

With the expansion of communication networks, fast and intelligent routers are required. Usually, these routers perform multiple lookups per packet into potentially large tables. For instance, the router of an ATM switch fabric [83] translates the incoming identifier (8-bit $Item\_In1$) into the destination identifier according to the routing table $Lookup\_Table$ [83]. Only those $Item\_In1$ inputs which can find a match ($Match\_Found = 1$) in the routing table can be switched. Here, we consider a simplified version of the original program and illustrate the application of our method.

```
//module name
module search;

 //Variables declaration
 reg [7 : 0] Lookup_Table [9 : 0];
 reg Match_Found;
 integer i, Error_Count;
 reg [7:0] Item_in1;

 //Initialization
Init:
 initialization  i=0; Match_Found = 0; Error_Count = 0;

epsilon: while (i <= 9) begin
L0:      if (Lookup_Table[i][7:0] == Item_In1) begin
L1:          Match_Found = 1;
L2:          i=9;
          endif
L3:       i = i+1;
          endwhile
L4:      if (!Match_Found)
L5:          Error_Count = Error_Count + 1;
         else
L6:          Error_Count = Error_Count - 1;
         endif
omega:
```

In the above program, if a match is found, the flag *Match_Found* is set. If after searching the whole table, no match can be found, the error counter (*Error_Count*)

increments by one, else *Error_Count* decreases by one. The table totally has 10 items $\{00, 01, \ldots, 09\}$, and the length of each item is 16-bit. Each statement in the program is labelled by a program counter such as l_0, l_1 and so on, where *Init* labels the initial statements, ϵ labels the beginning of the code, and ω labels the end of the code. The CFD of the search program is shown in Figure 7.15.

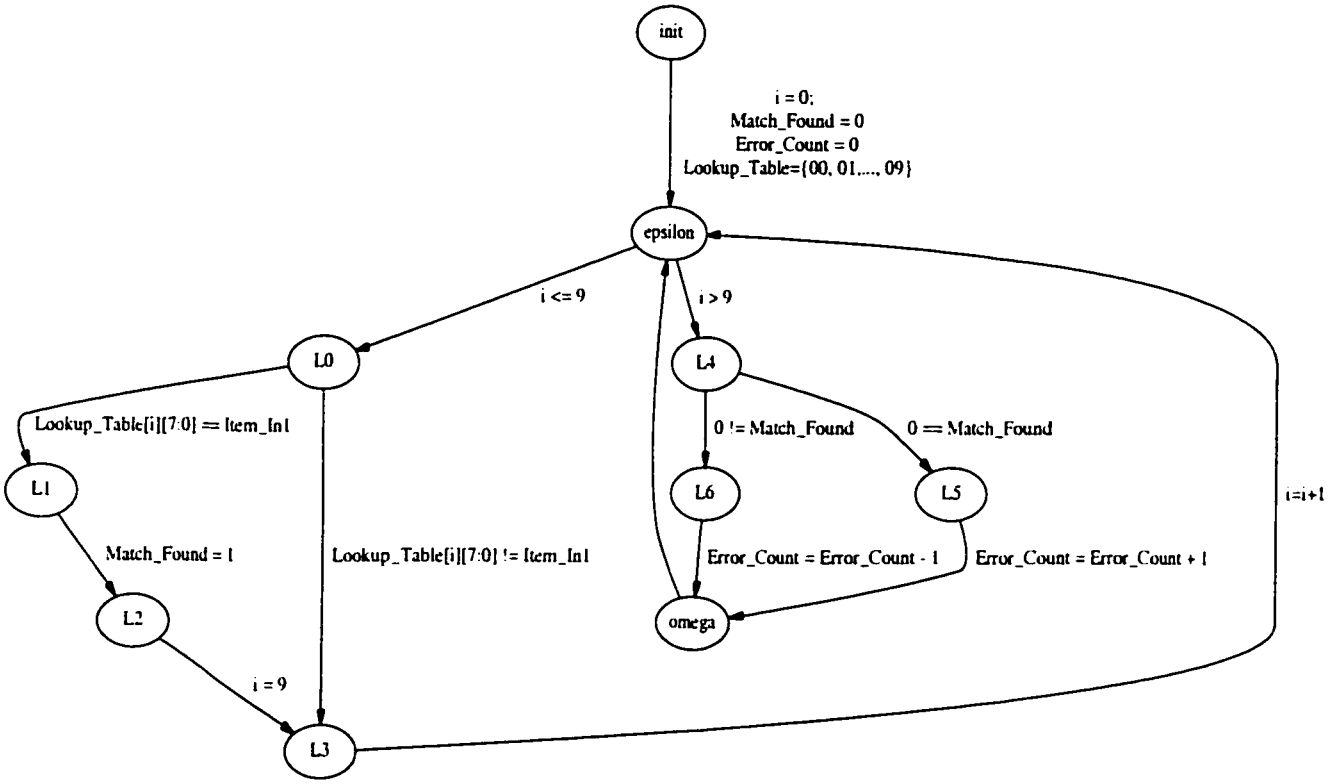


Figure 7.15: CFD of the Search Program

We want to prove that "the input $Item\_in = 02$ can find a match in the table such that $Match\_Found = 1$". The correspondent properties are φ_1 : $\mathbf{AF}((Item\_In1 = 2) \wedge (Match\_Found = 1))$ and φ_2 : $\mathbf{AG}((\neg(Item\_In1 = 2)) \rightarrow (Match\_Found = 0))$ meaning that $(Item\_In1 = 2)$ eventually can find the match and the others cannot. These properties could not be verified on the original model due to state space explosion during the verification in VIS [11] on a SUN Enterprise server with 6GB memory after more than 10 hours running. Hence, we have to reduce the model

before we can make model checking. First, using COI, we want to remove any variables which are not affecting the variable of interest, i.e., *Match_Found*. We do not need to consider *Item_In*1 since it is an input.

The DDD of the search program is shown in Figure 7.16. Through the analysis

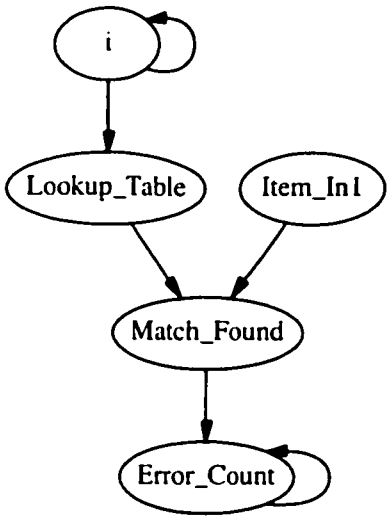


Figure 7.16: Dependency Diagram of the Search Program

of the data dependency diagram of the search program, we can find that variable *Match_Found* depends on variables *Lookup_Table*, *Item_In*1, and *i*, but not on *Error_Count*. Consequently, we can simply remove variables *Error_Count* in the verification of the property, and preserve the soundness and completeness of the property. The corresponding reduced data dependency diagram of the reduced program is shown in Figure 7.17. The control flow diagram of the search program after the COI reduction is shown in Figure 7.18.

Although we can remove *Error_Count* in the above, however, we cannot remove *Lookup_Table* which *Match_Found* depends on. In this case, we will do the following reduction.

The key node in the CFD is node L_2 because the variable of interest *Match_Found* is changed in the incoming edge of this node. The value $\{00, 01, 03, \ldots, 09\}$ of *Lookup_Table*$[7 : 0]$ are deactive because we cannot find a path which leads to

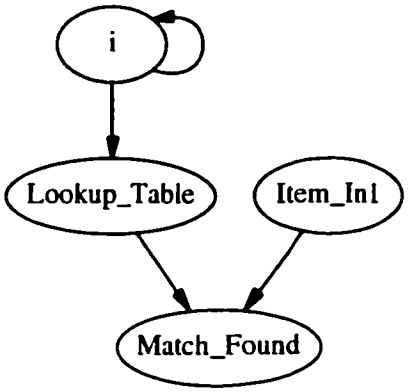


Figure 7.17: COI Reduced Dependency Diagram of the Search Program

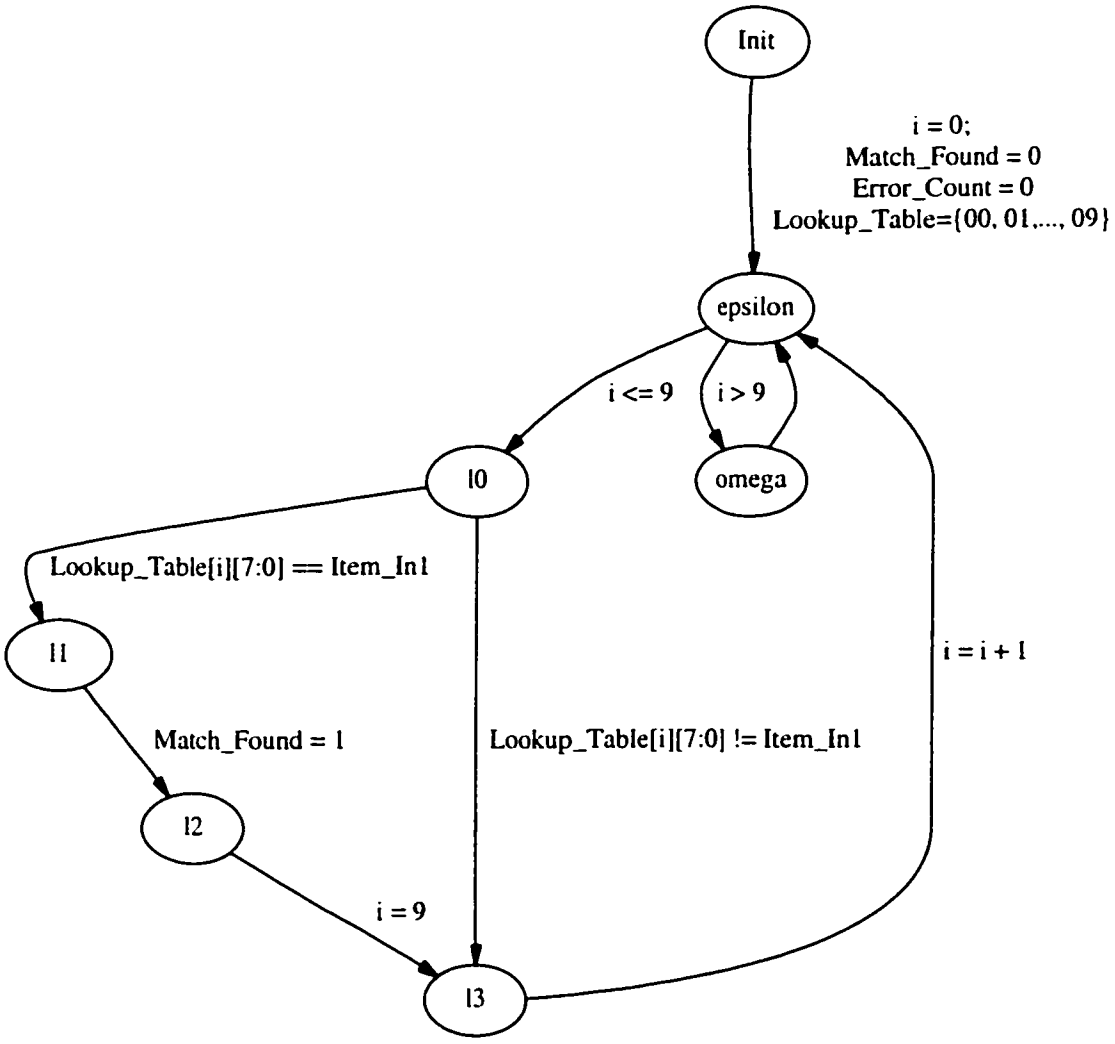


Figure 7.18: COI Reduced CFD of the Search Program

L_2 and $ST_{n \to L_2}(Lookup\_Table[7:0]) = 00, 01, 03, \ldots, 09$ for at least one node n in the path. The value $\{02\}$ of $Lookup\_Table[7:0]$ is active since L_2 is reachable in path ϵ, L_0, L_1, L_2 with $ST_{\epsilon \to L_2}(Lookup\_Table[7:0]) = 02$. Consequently, variable $Lookup\_Table[7:0]$ is an abstract variable, where the active value set ACTIVE $(Lookup\_Table[7:0]) = \{02\}$ and the typical deactive value DEACTIVE $(Lookup\_Table[7:0]) = 03$. So, we can use $Lookup\_Table[7:0] = \{02, 03\}$ instead of $\{00, 01, 03, \ldots, 09\}$. After such a reduction, we removed 8 items from the table, and get a reduced program. This reduced program can finish model checking in VIS on the same machine (SUN Enterprise with 6GB memory). The experimental results are shown in Table 7.1, where the verification could not be completed on the concrete model (actually, the concrete model has 1024 items), but succeeded on the reduced model. The table shows the statistic results of CPU time (in seconds) and memory usage (in bytes) of VIS verification on the reduced model, and that of SAT solver on RC_τ, respectively.

	Reduced Model		RC_τ Computation	
Prop.	Verif. CPU	Verif. Mem	SAT CPU	SAT Mem
φ_1	11.3	8.54M	0.02	83.53K
φ_2	11.6	88.51M	0.02	83.53K

Table 7.1: Verification Results of Sample Properties in VIS

7.6.2 The Bakery Controller

Following is the Verilog code of the bakery controller [11] [2].

```
module bakery_process ;
//Variable declaration
```

[2]The program has been revised to fit our syntax defined in Section 7.2.1.

```
{ L1, L2, L3, L4, L5, L6, L7, L8, L9 } loc;
{0, 1, 2, 3} i, j;
{0, 1, 2, 3} numberi, numberj;
{0, 1} choosingj;

//Initialization
Init:
      pc=L1; j=0;
      numberi = 0;
      numberj = 0;
      choosingj = 0;

epsilon:

    if (permit) begin
        case (pc)
            L1: begin pc=L2; end
            L2: begin pc=L3; end
            L3: begin pc=L4;end
            L4: begin j=0; pc=L5; end
            L5: begin if (j<2) pc=L6; else pc=L9; end
            L6: begin if (choosingj==1) pc=L6; else pc=L7; end
            L7: begin
                if ((numberj!=0) &&
                    (numberj<numberi || (numberj==numberi && j<i)))
                    pc=L7;
                else
                    pc=L8;
```

```
            end
        L8: begin j=j+1; pc=L5; end
        L9: begin pc = $ND(L1, L9); end//$
    endcase
end
omega:
```

The reachable states of this program are as follows.

```
Reachability analysis results:
FSM depth =                  14
reachable states =          448
```

The property we want to verify is $\mathbf{AG}((pc = L1) \rightarrow (pc = L9))$. In this case, among the values of *pc*, only $L1$, and $L9$ are active. The other values are deactive, so we will use $L2$ as the typical value of the deactive values. Now, the abstract domain of *pc* becomes $\{L1, L2, L9\}$. Using the same approach, the abstract domain of j is $\{1, 2, 3\}$, where $\{1\}$ is the typical deactive value and $\{2, 3\}$ are the active values. Since there are no assignment statements of $numberi, numberj$, and $choosingj$, these variables can be treated as wired variables in order not to generate states. Because there are 6 *case* branches with deactive values in the condition, we use non-deterministic variable (*NDTran*) to make the branch, namely the branches from $L2$ to $L3$, $L4$, $L5$, $L6$, $L7$, and $L8$, respectively, are nondeterministic decided by *NDTran*. The new reduced program is as follows.

```
module bakery_process;
//Variable declaration
{ L1, L2, L9 } loc;
{ 1, 2, 3} j;
{0, 1, 2, 3} numberi, numberj;
{0, 1} choosingj;
```

```
//Initialization
Init:
      pc=L1; j=0;
      numberi = 0;
      numberj = 0;
      choosingj = 0;

   wire [1:0]   NDTran;
   assign  NDTran = $ND(0,1,2,3,4,5);//$

epsilon:
    if (permit) begin
        case (pc)
          L1: begin pc=L2; end
          L2: begin
              case (NDTran)
                0: pc = L2;
                1: j = 0; pc = L2;
                2: if (j<2) pc=L2; else pc=L9;
                   end
                3: if (choosingj==1) pc=L2;
                4: if ((numberj!=0) &&
                       (numberj<numberi || (numberj==numberi && j<i)))
                            pc=L2;
                5: j=$ND(1,2,3); pc=L2; //$
              endcase
           L9: begin pc = $ND(L1, L9); end//$
```

```
        endcase
    end
omega:
```

The reachable states of the reduced program are as follows, which is only half the size of the original one.

```
Reachability analysis results:
FSM depth =                  6
reachable states =         224
```

Through out the reduction, we can find that we have to examine the values of the variables one by one. Hence, it is required that the domain of the variables be finite. Moreover, to calculate whether a key node is reachable, we need to know exactly the truth or falsity value of the conditions in each branch in the program. Namely, assuming we are calculating whether or not the key node is reachable from the nodes with value e of variable j being active. Consider a test statement "$if(j < i)$" in the program. We need to know the true value of this test when $j = e$. Hence, we have to examine n possible cases, where n is the size of the domain of variable i. Consequently, in order to check each value of variable j, at this test statement, we need to consider $Size(dom(j)) \times Size(dom(i))$ possible cases which is exponential. In this case, to solve this problem, either we put an environment constraint on i, for example, set i to be e_i, where e_i is an element in $dom(i)$, or we need a more efficient solution.

7.6.3 The Arbiter

Following is the Verilog code of the arbiter controller[11][3].

```
module arbiter-controller;
```

[3]The program has been revised to fit our syntax defined in Section 7.2.1.

```
//Variable declaration
{A, B, C, X} reg sel, id;
reg ack, pass_token;
{IDLE, READY, BUSY } reg  state;
reg is_selected;
//Initialization
Init:
state = IDLE;
ack = 0;
pass_token = 1;
id = A;

epsilon:
  is_selected = (sel == id);
  case(state)
    IDLE:
%     if (is_selected)
        if (req)
          begin
%          state = READY;
          pass_token = 0;
         end
        else
          pass_token = 1;
      else
        pass_token = 0;
    READY:
```

```
      begin
      state = BUSY;
      ack = 1;
      end
    BUSY:
      if (!req)
        begin
        state = IDLE;
%        ack = 0;
        pass_token = 1;
        end
  endcase
omega:
```

The reachable states of the above model are shown as follows.

```
Reachability analysis results:
FSM depth =                    3
reachable states =            36
```

What we want to prove is that if arbiter controller A receives a request req, then it will eventually acknowledge this request by sending ack, i.e., $\mathbf{AG}((req = 1) \rightarrow \mathbf{AF}(ack = 1))$.

The latches in the program are

- $\{A, B, C, X\}$ reg sel, id;
- reg $ack, pass\_token$;
- $\{IDLE, READY, BUSY\}$ reg $state$;

- reg *is_selected*;

The variables of interest are *req* and *ack*. From the program, we see that the property does not depend on variable "*pass_token*", so this variable can be removed. Among the values of "*sel*", "A" is the active value, and "B, C, X" are the deactive values. Consequently, we can use "$\{A, B\}$ reg *sel*" instead of "$\{A, B, C, X\}$ reg *sel*". The value of "*state*" are all active values, so we have to keep them. The reduced data becomes

- $\{A, B\}$ reg *sel*;
- reg *ack*;
- $\{IDLE, READY, BUSY\}$ reg *state*;
- reg *is_selected*;

The reachable states of the corresponding reduced program are shown as follows.

```
Reachability analysis results:
FSM depth =                  3
reachable states =          10
```

Since the code between ϵ and ω are in a forever loop, the path from a node with value e to the key node may contain many cycles of the loop. In the above example, the value $READY$ of variable *state* is set at statement $state = READY$. The path from this statement to the key node contains at least two cycles, i.e., "$state = READY$"$\rightarrow \ldots \rightarrow$ "ω"$\rightarrow$ "ϵ" $\rightarrow \ldots \rightarrow$ "$Keynode$". In the worst case, to check if key nodes are not reachable from another node in the CFD, we have to check all the paths between the node and the key nodes using the conditions in Definition 7.5.2.

7.7 Summary

In this chapter, we proposed an automatic model reduction approach. The approach uses syntactic analysis and generates a smaller program compared to the original one. It preserves the properties to be verified since the concrete model is simulated by the abstracted model with respect to the properties. While generally applicable, this model reduction approach can be used in compositional verification when the size of a single module of a large system is beyond the capability of model checking. Such an application is displayed in the next case study chapter.

In our approach, the reduction is applied on the syntactic analysis of the program source code. Compared to the on-the-fly explicit state space reduction during the generation of the state space, source code reduction can be combined with other model checking or reduction tools to improve its reduction efficiency even more because the reduced program can be fed into other reduction tools. Moreover, since the size of the source code is usually much less than that of its state space or other intermediate forms, this reduction approach uses less resources (CPU time and memory space). However, in the worst case, the procedure examining the value domain of the variables is exponential with respect to the size of variables in the reachability condition (RC) because it has to traverse all the possible values of the variables in RC. A SAT solver is used to find the true value assignment throughout the reduction. The complexity of the reduction depends on that of the CFD traverse and the SAT solver.

Part III

Case Study

Chapter 8

Case Study on a Nortel ATM Switch Fabric

In the previous chapters, we learned that state space explosion is the biggest problem in the model checking applications. We also learned that there are two approaches to solve these problems: compositional verification and model reduction. However, in the compositional verification, we have to construct the environment of the modules under verification (environment problem); in the model reduction, lots of human intervention makes the application of model reduction difficult. In order to tackle these problems, we proposed practical compositional verification techniques, namely the environment synthesis and the syntactic model reduction. In this chapter, to illustrate our approach, we used an industrial size ATM (Asynchronous Transfer Mode) switch fabric from Nortel Networks [83] as a real case study. Using these techniques, we succeed in verifying the switch fabric whose size is beyond the capability of current plain model checking tools. Throughout the verification, we provide the environment assumptions as temporal logic formulas in **ACTL** and then synthesize the formulas into Verilog modules. We then compose this environment module with the RTL block under verification and fed it into a model checking tool (here VIS [11]). However, in case the size of the composed module is still beyond the

capability of model checking, we use our syntactic model reduction algorithm based on the cone of influence reduction, which analyzes the (Verilog) source code and removes the redundant variables and values. We will follow the general verification framework described in Section 1.2 and illustrated in Figure 1.4.

8.1 The 4×4 ATM Switch Fabric

The basic purpose of an ATM switch fabric is to transport valid (i.e. uncorrupted) ATM cells arriving at its ingress ports to the designated egress ports. Invalid ATM cells are to be discarded. Besides valid and invalid ATM cells, ATM cell streams may also contain idle cells, which serve to adapt the cell streams to the transmission bit rates employed. At the ingress side an ATM switch suppresses idle cells, while at the egress side it inserts idle cells whenever no cell is available for transmission. Figure 8.1 illustrates the switching of ATM cells for a 4×4 switch fabric [83]: all numbered cells are switched onto one of the four egress ports, with the exception of cell 6 which is perceived as a corrupted, invalid cell.

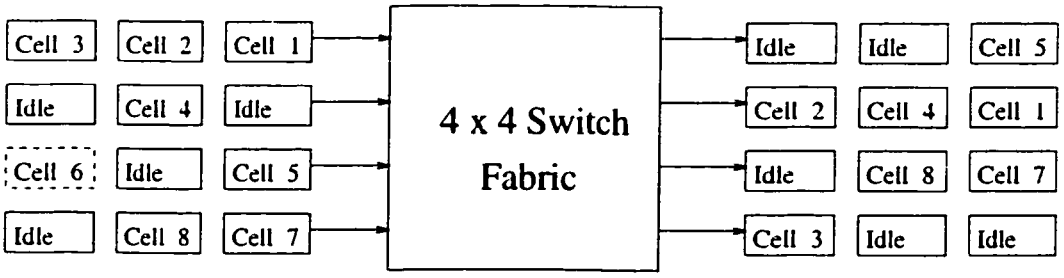


Figure 8.1: Switching of (valid) ATM Cells By a 4×4 Switch Fabric

Cell type identification and cell switching is based on the contents of ATM cells. More precisely, an ATM cell is a fixed-length cell consisting of a 5-octet header field and a 48-octet payload field. The payload field is available for actual user information. The header field carries the information for identification and transportation of the cell. The header of an ATM cell is further decomposed into

subfields as illustrated in Figure 8.2.

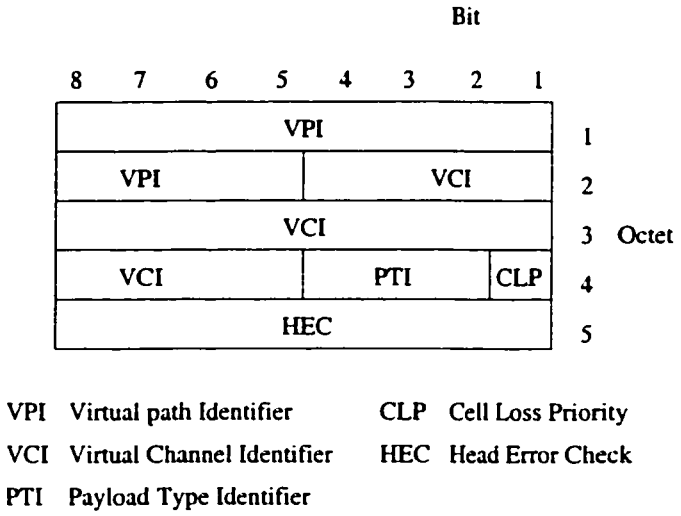


Figure 8.2: Header of an ATM Cell

The virtual path identifier (VPI) comprises the first 12 bits of the cell header, and is followed by a 16-bit virtual channel identifier (VCI). Together, the VPI and VCI constitute the routing field of the cell. Three header bits are used for the payload type identifier (PTI), which indicates the type of information (user data, or operation and maintenance data) contained in the cell payload. One further bit is used to indicate the cell loss priority (CLP). If the CLP bit is 0, the cell has high priority and hence sufficient network resources need be allocated to it. Otherwise, the cell has low priority and is subject to discard depending on current network conditions. The PTI and CLP fields are not used explicitly for cell switching purposes. The last octet of the cell header contains the header error check (HEC) sequence used to check the integrity of the other header subfields. The value of the HEC sequence is defined as "the remainder of the division (modulo 2) by the generator polynomial $x^8 + x^2 + x + 1$ of the product of x^8 and the content of the first four octets of the header, added (modulo 2) to the fixed pattern 01010101". It should be noted that the ATM cell header format depicted in Figure 8.2 is the format used at Network-Node Interfaces (NNI's). A slightly different format, not

further considered here, applies at User-Network Interfaces (UNI's). The octets of an ATM cell are transmitted in increasing order, starting with octet one. Thus, the cell header is transmitted first, followed by the payload field (octets 6 to 53). Bits within each octet are transmitted in decreasing order, starting with bit 8 (the most significant bit). ATM cell switching can now be described in brief as follows. After receiving a cell at one of its ingress ports, an ATM switch fabric determines whether the cell is a corrupted or idle cell. A corrupted cell is a cell with an incorrect HEC sequence. An idle cell is a cell with its VPI, VCI and PTI bits all set to 0 and its CLP bit set to 1, and with a correct HEC sequence. If the ingress ATM cell is corrupt or idle, it is discarded. Otherwise, an attempt is made to translate the value of the VPI/VCI field in the cell header into a new VPI/VCI value and an egress port number by means of a VPI/VCI lookup-table. If the lookup-table contains an enabled entry for the VPI/VCI value of the ingress cell, this value is replaced by the new VPI/VCI value and a new correct HEC sequence is generated. The resulting cell (i.e. with the new VPI/VCI value and HEC sequence) is then switched onto the designated egress port. If no enabled entry is available for the VPI/VCI value of the ingress cell, the cell is discarded. Cell buffers can be placed at various "locations" within a switch fabric, depending on the architecture of the switch. Most commonly used are input buffers and output buffers. In case of the former, cells are buffered at the ingress ports immediately upon their arrival. In case of the latter, cells are buffered at the egress ports after the VPI/VCI translation. Lastly, a switch fabric preserves the sequence order among valid cells received at the same ingress port and destined for the same egress port.

The particular 4×4 ATM switch fabric incorporates the following features to perform ATM cell switching:

- *cell HEC verification/generation*

  The correctness of the header of each ingress cell is verified by checking the value of its HEC sequence. An ingress cell with an incorrect HEC sequence is

discarded. The header of each egress cell has a correct HEC sequence.

- *Idle cell suppression/insertion*

  Idle cells in the ingress ATM cell streams are suppressed. Idle cells are inserted into the egress ATM cell streams whenever no cell is available for transmission.

- *Table-controlled cell VPI/VCI translation*

  The value of the VPI/VCI field of an ingress cell is translated into a new VPI/VCI value and the designated egress port number for the modified cell. The translation is based on a VPI/VCI lookup-table whose format is as shown in Figure 8.3. The "enable flag" field occupies 1 bit. If the value of the enable flag is 1, the corresponding table entry is enabled; otherwise it is disabled. The "old VPI/VCI" and "new VPI/VCI" fields are each 28 bits wide, in accordance with the ATM cell header structure in Figure 8.2 The "egress port" field is 2 bits wide, allowing each of the four egress ports of the 4×4 ATM switch fabric to be addressed. The VPI/VCI lookup-table can hold up to 1024 entries. If there is no enabled entry for the value of the VPI/VCI field of an ingress cell, the cell is discarded.

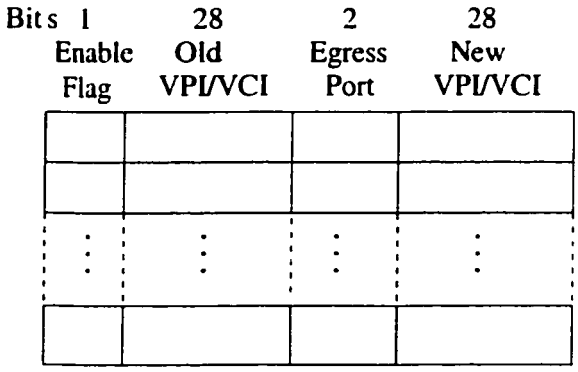


Figure 8.3: Format of the VPI/VCI Lookup-table

- *Output buffering*

Successfully translated ingress cells are buffered at the appropriate egress ports prior to transmission. Each of the four egress ports of the switch fabric can queue up to 10 ATM cells. When a given output queue is full, cells destined for the corresponding egress port are discarded.

- *Cell discard counters*

  Three counters named *HEC_DISCARD_COUNT*, *VPI_VCI_DISCARD_COUNT* and *BUFFER_DISCARD_COUNT* are maintained to track the number of ingress cells discarded due to an incorrect HEC sequence, a non-matching VPI/VCI value and a buffer overflow, respectively. Not taking into account idle cells, the total number of ingress cells of the switch fabric should be equal to the total number of egress cells plus the sum of the above three counters.

8.2 Modeling the Switch Fabric

There are mainly four modules in the ATM switch fabric, *ATM_SWITCH*, *ATM_MON*, *FIFO_QUEUE*, and *ATM_GEN* as shown in Figure 8.4. *ATM_SWITCH* is the root module which includes the ATM cell routing functions. *ATM_MON* is the ingress part of the fabric which includes the ATM cell monitor and detection functions. *FIFO_QUEUE* is the queuing module. *ATM_GEN* is the egress part of the fabric which includes the ATM cell restructure functions.

Each *ATM_MON* or *ATM_GEN* has primary inputs *CLOCK_IN*, *DATA_IN*, *EOCN_IN*, and *CLOCK_OUT*, *DATA_OUT*, *EOCN_OUT*, respectively. *CLOCK_IN* and *CLOCK_OUT* are the transmission clocks. *DATA_IN* and *DATA_OUT* are the transmission data bus which is 8-bit wide. *ECON_IN* and *ECON_OUT* are the end-of-cell marks. The functional timing of the ingress and egress interfaces of the 4×4 ATM switch fabric is shown in Figure 8.5 and 8.6, respectively. ATM cells should be sent to ingress port i ($i = 0, \ldots, 3$) at the falling edge of *CLOCK_IN_i*, and received at egress port i ($i = 0, \ldots, 3$) at the rising edge of *CLOCK_OUT_i*.

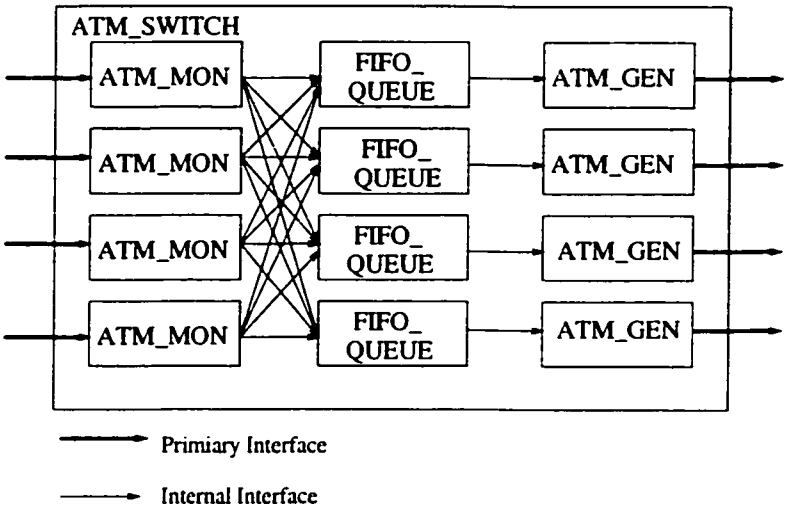


Figure 8.4: ATM Modules

The corresponding end-of-cell marks should go low at the 52nd octet of each cell sent or received.

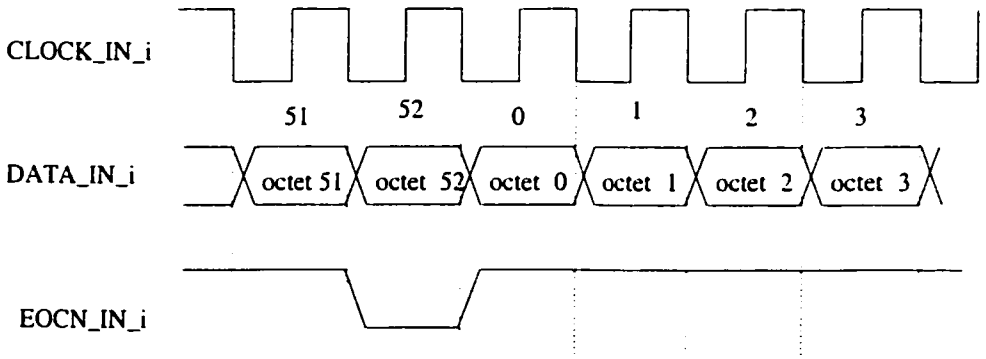


Figure 8.5: Ingress Interface Timing

8.3 Specifying Local Properties

The major property of such an ATM switch fabric is that

"Valid cells (with good HEC and matching VPI/VCI) are switched correctly"

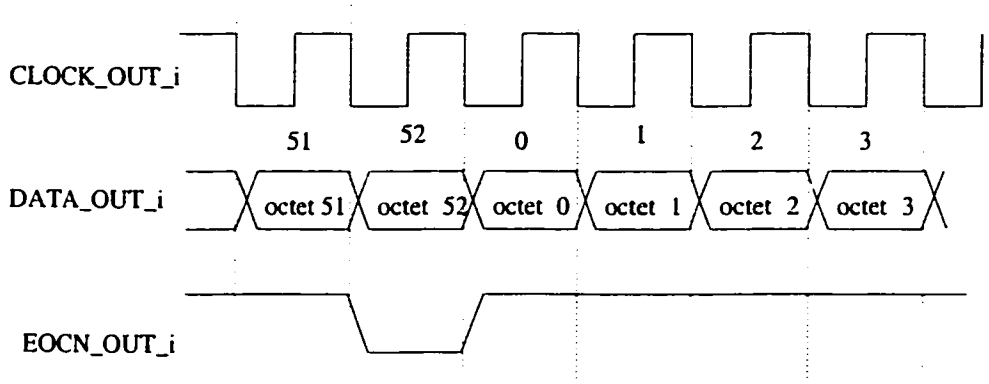


Figure 8.6: Egress Interface Timing

Trying to prove this property directly using model checking will fail because of state space explosion, even after model reduction. In order to prove this property, compositional verification is necessary. Here, since all the cells are queued in the *FIFO_QUEUE* module, we specify the ingress part and the egress part separately and extract the local properties respectively. Namely, in the ingress part, valid cells (with good HEC and matching VPI/VCI) are switched into the queue, and in the egress part, cells in the queue are restructured and sent.

8.3.1 Specifying the Ingress Local Properties

The ingress part of the ATM switch fabric is shown in Figure 8.7. There are three

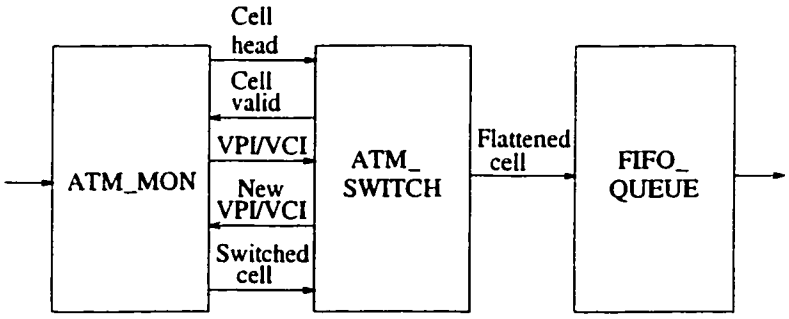


Figure 8.7: Switch Fabric Ingress

modules, *ATM_MON*, *ATM_SWITCH*, and *FIFO_QUEUE*. The names over the

arrow lines are the interface signals between modules. For example, *ATM_MON* detects the head-bit (HEC, IDLE) of a cell and gives them to *ATM_SWITCH*. *ATM_SWITCH* checks if these bits are valid and notifies *ATM_MON* module. Because there are circular signals between the modules, we must decompose the system into a linear one with respect to the time so that we can use the compositional reasoning.

In order to decompose such a system, we illustrate the system into a more detailed level, which is shown in Figure 8.8. In this illustration, the interface signals

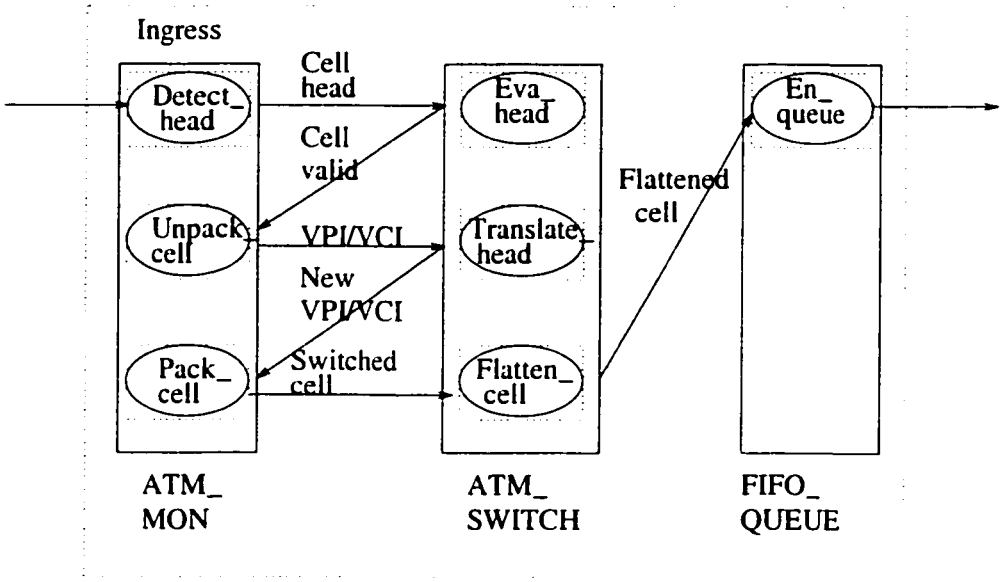


Figure 8.8: Detailed Illustration of the Ingress

are not altered. However, this time, we can see that the communication between modules are handled by some blocks, namely *Detect_head*, *Unpack_cell*, *Pack_cell*, and so on. We thus can decompose the system into linear blocks with respect to the time sequence.

In order to prove that valid cells (with good HEC and matching VPI/VCI) are switched correctly, we need to prove that each corresponding block works respectively. Namely, *Check_head* can detect the head, and *Eva_head* can evaluate the head correctly, and *Translate_head* can translate VPI/VCI correctly, and so on.

These local properties are illustrated in the following.

Ingress_P$_1$: In this property, we require that the ingress port will receive a cell if a cell is coming into the port. Formally,

$\mathbf{AF}(New\_cell\_recieved = 1)$

where *New_cell_recieved* is set when a cell with integral structure is received.

Ingress_P$_2$: In this property, we check the HEC detection mechanism in the ingress part, given there is a cell ready. Namely,

$\mathbf{AG}(HEC\_OK = 1)$

where *HEC_OK* is set if the cell under test has a good HEC value.

Ingress_P$_3$: In this property, we check the IDLE detection mechanism in the ingress part, given there is a cell ready. Formally,

$\mathbf{AG}((WORD[0] = 0) \wedge (WORD[1] = 0) \wedge (WORD[2] = 0) \wedge (WORD[3][7 : 1] = 0) \wedge (WORD[3][0] = 1) \rightarrow (IS\_IDLE = 1))$

meaning that when the byte stream (*WORD*) in a cell satisfying the above format (all 0 except the last bit), then this cell is judged to be an idle cell.

Ingress_P$_4$: In this property, we check that a cell is unpacked correctly. Formally,

$\mathbf{AG}((VPI[11 : 4] = WORD[0]) \wedge (VCI[11 : 4] = WORD[2]) \wedge (VPI[3 : 0] = WORD[1][7 : 4]) \wedge (VCI[15 : 12] = WORD[1][3 : 0]) \wedge (VCI[3 : 0] = WORD[3][7 : 4]) \wedge (PTI[2 : 0] = WORD[3][3 : 1]) \wedge (CLP = WORD[3][1]))$

where *WORD* is the input byte stream and *VPI, VCI, CLP, PTI* are the formatted cell headers.

Ingress_P$_5$: In this property, we check that if the incoming VPI_VCI satisfies our specification (bit 27 to 4 are 0), then it will find a match in the routing table. Formally,

$(\mathbf{AF}(((VPI\_VCI\_IN[27 : 4] = 0) \wedge (MATCH\_FOUND = 1))))$

where VPI_VCI_IN is the VPI_VCI value of the input cell. $MATCH\_FOUND$ is set when VPI_VCI_IN can find a match in the routing table.

$Ingress\_P_6$: In this property, we check that all incoming VPI_VCI that do not satisfy our specification cannot find a match in the routing table. Formally,
$(\mathbf{AG}((\neg(VPI\_VCI\_IN[27:4]=0)) \rightarrow (MATCH\_FOUND=0)))$
This is a safety property of the routing table, which has the similar form as $Ingress\_P_5$.

$Ingress\_P_7$: In this property, we check that the cell is packed correctly. Formally,
$\mathbf{AG}((VPI[11:4]=WORD[0]) \wedge (VCI[11:4]=WORD[2]) \wedge (VPI[3:0]=WORD[1][7:4]) \wedge (VCI[15:12]=WORD[1][3:0]) \wedge (VCI[3:0]=WORD[3][7:4]) \wedge (PTI[2:0]=WORD[3][3:1]) \wedge (CLP=WORD[3][1]))$
This property is similar with $Ingress\_P_4$.

$Ingress\_P_8$: In this property, we check that the cell is flattened correctly, namely the word structure of a cell can be correctly flattened into a bit stream. Formally,
$\mathbf{AG}((FLATTENED\_CELL[7:0]=WORD[0]) \wedge (FLATTENED\_CELL[15:8]=WORD[1]))$
where $FLATTENED\_CELL$ is the corresponding bit stream of the cell.

$Ingress\_P_9$: In this property, we check that the flattened cell can be enqueued correctly, namely the flattened cell is put into the queue and the pointer of the queue is changed accordingly. Formally,
$\mathbf{AG}(\neg IS\_FULL \rightarrow \mathbf{AF}((Queue.HEAD=FLATTENED\_CELL) \wedge (HEAD=HEAD+1)))$
where IS_FULL is set when the queue is full; The property means that if the queue is not full, then the cell will find a place in the queue.

These local properties only work under corresponding certain environments assumptions. The assumptions of these environments are described as **ACTL** formulas, which can be synthesized into Verilog modules using our environment synthesis approach.

8.3.2 Specifying the Egress Local Properties

The egress part of the fabric is shown in Figure 8.9.

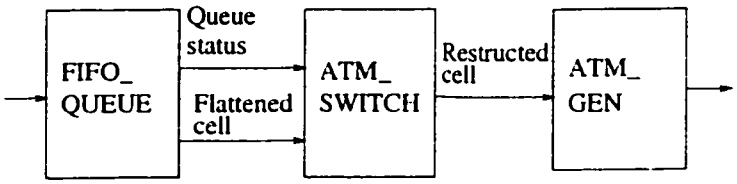


Figure 8.9: Switch Fabric Egress

Using the same approach as the previous chapter, we can decompose the system into a linear one as shown in Figure 8.10.

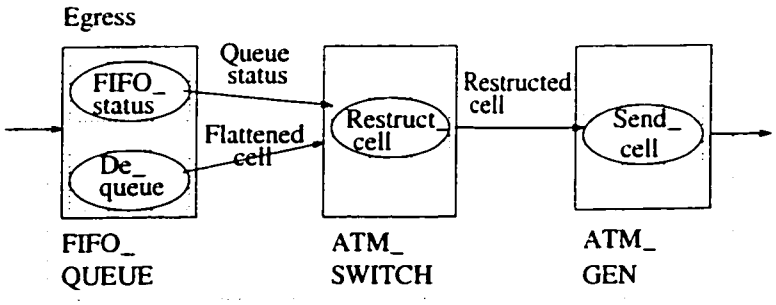


Figure 8.10: Blocked Switch Fabric Egress

The local properties are as following.

Egress_P$_1$: In this property, we check that the status of the queue is empty if the head pointer equals to the tail pointer. Formally,

$\mathbf{AG}((HEAD = TAIL) \rightarrow (EMPTY = 1))$

where $EMPTY$ is set when the queue is empty.

Egress_P$_2$: In this property, we check that if there is at least one cell in the queue, then this cell can be de-queued correctly. Formally,

$\mathbf{AG}(!EMPTY \rightarrow ((FLATTENED\_CELL = FIFO[TAIL])))$

meaning that if the queue is not empty, then a cell can be dequeued.

Egress_P$_3$: In this property, we check that the flattened bit stream cell can be restructured into a word format cell. Formally,

$\mathbf{AG}((RESTRUCTED\_CELL[0] = FLATTENED\_CELL[7:0]) \wedge$
$RESTRUCTED\_CELL[53] = FLATTENED\_CELL\ [423:416]))$

meaning that the dequeued cell (*FLATTENED_CELL*) can be restructured into a formatted cell (*RESTRUCTED_CELL*);

Egress_P$_4$: In this property, we check that the de-queued cell can be sent out to the egress port. Formally,

$\mathbf{AG}(NEWCELL\_READY = 1)$

where *NEWCELL_READY* is set when a cell has been sent out successfully.

8.4 Verification of the Switch Fabric

Because we cannot make model checking of the global property

Valid cells (with good HEC and matching VPI/VCI) are switched correctly

against the whole fabric due to state space explosion, we have to use compositional verification to verify the modules in the system.

According to compositional rules, we have to prove:

1. Every block satisfies its local properties under certain environments (stimulus).
2. The environments should be discharged as well.

In this fabric case, we need to verify that the blocks in the ingress part, i.e., *Detect_head*, *Eva_head*, *Translate_head*, etc., and the blocks in the egress part, i.e.,

FIFO_status, *De_queue*, etc., satisfy their local properties given a cell coming in. Here, we only show how to prove local properties $Ingress\_P_1$, $Ingress\_P_5$, and $Ingress\_P_6$. The other properties can be proved in a similar way.

$Ingress\_P_1$ specifies that *ATM_MON* will check the input stream of *DATA_IN* port to see if there is a bit structure satisfying the interface timing and the ATM cell format. If so, a flag *New_cell_recieved* will be set to indicate that an ATM cell is received from the port. In the verification, we assume that an ATM cell will eventually get to the port according to the interface timing. However, an ATM cell has 53-byte. If we assume the behavior of an ATM cell (53-byte) and prove that this cell will be received in *ATM_MON* (another 53-byte), the state space explodes immediately. In this case, model reduction is necessary.

When *ATM_MON* receives a 53-byte structure from the port, it only checks the first 5 bytes of the structure to see whether these 5 bytes satisfy the ATM cell format (i.e., in frame). Once the first 5 bytes are in frame, we judge that an ATM cell is received no matter what the payload bytes (byte 6 to 53) are. Thus, we can remove the payload bytes when we verify whether or not *ATM_MON* can receive a cell.

Consequently, the environment assumption can be written as an **ACTL** formula:

$$\mathbf{AF}(B0 \wedge \mathbf{AX}(B1 \wedge \mathbf{AX}(B2 \wedge \mathbf{AX}(B3 \wedge \mathbf{AX}(B4 \wedge EOCN)))))$$

where $B0$, $B1$, $B2$, $B3$ and $B4$ are the first 5 bytes of a cell, respectively (the synthesized Verilog code is shown in Appendix A.1). The assumption property means that a $B0B1B2B3B4$ structure will eventually be fed into the data port of *ATM_MON* and the timing satisfies *ECON*. This assumption needs not to be discharged since it is the primary input. In the verification of *ATM_MON*, we will check property $AF(New\_cell\_recieved = 1)$, where *New_cell_recieved* is set when a structure *B0B1B2B3B4* is received. The verification of $Egress\_P_4$ is similar to that of $Ingress\_P_1$. The only difference is that the direction of the data flow is

reversed. Because VIS can finish the verification of the property $Ingress\_P_1$, no model reduction is needed in this case.

In the verification of $Ingress\_P_5$ and $Ingress\_P_6$, what we want to check is that the correct *VPI_VCI* of the incoming cell can find a match in the routing table, while the corrupted *VPI_VCI* of the incoming cell cannot find a match in the routing table. Hence, the environment assumption of these properties is the value of the *VPI_VCI*, i.e., *VPI_VCI_IN*. Since in the switch fabric, only those *VPI_VCI_IN* with bit 27 to 4 being 0 can find a match, the corresponding environment **ACTL** formula is

$$\mathbf{AF}(VPI\_VCI\_IN[27:4] = 0)$$

which is discharged if the blocks before "Translate_head" can be proved. This assumption can be synthesized into Verilog behavior code (see Appendix A.2). The synthesized code then can be composed with the block under verification, i.e., *Translate_head*. However, since the routing table is involved in the verification, and the size of the routing table is 1024×58-bit, no model checking tool can accept such a large model. We have to apply model reduction (Figure 1.4 **Step 5**) with respect to the properties. By observing the properties, we find that we are just verifying the behavior of variable *MATCH_FOUND*. According to the model reduction approach, those values that do not affect the value of *MATCH_FOUND* can be abstracted using one typical value. In the program of the model, only the first item in the routing table with bit 27 to 4 equaling to 0 can change the value of *MATCH_FOUND*, so this value is kept as active values, while all other values in the routing table which do not affect the behavior of *MATCH_FOUND* can be removed. So, we can keep only two items in the routing table and remove the other 1022 items. In this way, the model under verification is reduced. Then, we are able to verify the reduced module in VIS.

The verification results are shown in Table 8.1, The verification is performed using the VIS model checker [11] on a SUN Enterprise server with 6GB memory.

Throughout the model checking, we set VIS with the options: implicit clocking and advanced ordering. The CPU time reported is the real time. The BDD size in the table represents those states of the system that satisfy the formula.

Generally, properties $Ingress\_P_3$, $Ingress\_P_4$, $Ingress\_P_7$, $Ingress\_P_8$, and $Egress\_P_1$ can be verified in VIS without the model reduction. However, the other properties failed in VIS due to state space explosion. The symptom was that VIS would run out of all the memory resources and be deactived by the system.

Prop.	Status	Model Checking		
		CPU	Mem	BDD-node
$Ingress\_P_1$	Verified	19.5	0.908M	42722
$Ingress\_P_2$	Verified	272.4	1.908M	18446
$Ingress\_P_3$	Verified	1.4	1.308M	15073
$Ingress\_P_4$	Verified	3.8	9.9M	7130
$Ingress\_P_5$	Verified	11.3	8.54M	164033
$Ingress\_P_6$	Verified	11.6	88.51M	383969
$Ingress\_P_7$	Verified	3.7	9.918M	7104
$Ingress\_P_8$	Failed	-	-	-
$Ingress\_P_9$	Verified	15.1	100.6	490923
$Egress\_P_1$	Verified	2.5	1.44M	15764
$Egress\_P_2$	Verified	16.5	112.3	632434
$Egress\_P_3$	Failed	-	-	-
$Egress\_P_4$	Verified	6.7	12.2M	137724

Table 8.1: Verification Results of Sample Properties in VIS

In Table 8.1, "*Failed*" means that property checking failed due to bugs in the verification tool. For example,

```
always begin
        a = 1;
end
```

satisfies formula $\mathbf{AG}(a = 1)$, however,

```
always begin
        a = 1;
        a = 1;
end
```

does not satisfy the same formula. This is obviously a bug in the tool. In this case, we conducted the particular property verification in another tool (here FormalCheck) to make sure that its model checking really succeeds.

For the purpose of comparison, we verified the above models in FormalCheck [15] on the same machine. However, this time, we do not do the reduction using our model reduction approach. The verification results are shown in Table 8.2. The reduction algorithm selected in FormalCheck is iterated with empty reduction seed because there are no constraints on the primary inputs, and the run option is symbolic BDD because it allows a more efficient model checking. The CPU time in the table is the real time and "States" are the states reachable.

Property	Status	CPU(s)	Memory(MB)	States
$Ingress\_P_1$	Failed	-	-	-
$Ingress\_P_2$	Verified	1036	29.64	2.02e+03
$Ingress\_P_3$	Verified	4	3.121	4
$Ingress\_P_4$	Verified	22	6.71	1.02e+03
$Ingress\_P_5$	Non-terminated	-	-	-
$Ingress\_P_6$	Non-terminated	-	-	-
$Ingress\_P_7$	Verified	32	13.75	3.36e+07
$Ingress\_P_8$	Verified	8	3.69	1.31e+05
$Ingress\_P_9$	Non-terminated	-	-	-
$Egress\_P_1$	Verified	365	0.55	2.62e+05
$Egress\_P_2$	Non-terminated	-	-	-
$Egress\_P_3$	Verified	605	115.07	6.67e+02
$Egress\_P_4$	Failed	-	-	-

Table 8.2: Verification Results of Sample Properties in FormalCheck

In Table 8.2, "*Non-terminated*" means that the verification failed due to state space explosion. The reason for this is either that the property under verification involves so many variables in the program that the reduction algorithms in FormalCheck are of no help (in this case, FormalCheck gives an internal bug report), or the model under verification is too large to be even complied by the tool (in this case, the tool will stay in a dead lock state until all the memory is consumed). The "*Failed*" in the table means that the property cannot be verified by this tool because the environment assumptions could not be specified. Although we can use temporal constraints to express some environments, the tool itself does not provide a mechanism to establish this environment. From the data in table 8.2, we noticed that $Ingress\_P_2$ needs much more time in FormalCheck than other properties because the reduction time of this property is longer.

Overall, since the verification in VIS is based on the reduced model while the verification in FormalCheck is based on the concrete model, the former is efficient with respect to CPU time and memory because the latter has to do the reduction work by itself.

After the above verification, we actually proved every block satisfies its local properties, given certain environment assumptions. Moreover, because these environment assumptions are the outputs of the blocks in the system, they are discharged in the verification of the local properties. We apply the compositional rule

as follows, where $\mathcal{T}_\varphi$ means the synthesized Verilog module of formula φ

$$\frac{\begin{array}{r} Detect\_head \models Ingress\_P_1 \\ Eva\_head \| \mathcal{T}_{Ingress\_P_1} \models Ingress\_P_2 \\ Eva\_head \| \mathcal{T}_{Ingress\_P_1} \models Ingress\_P_3 \\ Unpack\_cell \| \mathcal{T}_{Ingress\_P_2} \| \mathcal{T}_{Ingress\_P_3} \models Ingress\_P_4 \\ Translate\_head \| \mathcal{T}_{Ingress\_P_4} \models Ingress\_P_5 \\ Translate\_head \| \mathcal{T}_{Ingress\_P_4} \models Ingress\_P_6 \\ Pack\_cell \| \mathcal{T}_{Ingress\_P_5} \| \mathcal{T}_{Ingress\_P_6} \models Ingress\_P_7 \\ Flattened\_cell \| \mathcal{T}_{Ingress\_P_7} \models Ingress\_P_8 \\ En\_queue \| \mathcal{T}_{Ingress\_P_8} \models Ingress\_P_9 \\ FIFO\_Status \models Egress\_P_1 \\ De\_queue \models Egress\_P_2 \\ Restruct\_cell \| \mathcal{T}_{Egress\_P_1} \| \mathcal{T}_{Egress\_P_2} \models Egress\_P_3 \\ Send\_cell \| \mathcal{T}_{Egress\_P_3} \models Egress\_P_4 \end{array}}{\begin{array}{l} \mathcal{T}_{Ingress\_P_1} \| Detect\_head \| Eva\_head \| Unpack\_cell \| Translate\_head \\ \| Pack\_cell \| Flattened\_cell \| En\_queue \| FIFO\_Status \| De\_queue \\ \| Restruct\_cell \| Send\_cell \models Egress\_P_4 \end{array}}$$

Actually, the global property: "Valid cells (with good HEC and matching VPI/VCI) are switched correctly" is given by assuming $Ingress\_P_1$ (valid cells) and deducing $Egress\_P_4$ (correct switch). This way, we are checking the satisfaction of the global property against the whole design.

8.5 Errors Discovered

The first of these is in the *SEND_CELL* block. Normally, if there is no cell to be sent, the *SEND_CELL* block continuously sends *IDLE* cell via the output port. If there is a real cell to be sent, the *SEND_CELL* block will send this real cell after an *IDLE* cell. The procedure looks like the following.

```
always begin: SEND_CELL
   if (not cell_ready)
      cell = IDLE CELL;

   SEND cell TO THE OUTPUT PORT;
   ...
end
```

where, *cell* has been assigned to be the real cell in the *Restruct_cell* block. However, the *cell_ready* signal is set by another *always* block which is parallel to the *SEND_CELL* block. So, the global picture of sending a cell is as follows.

```
always begin: Restruct_cell
   cell = REAL CELL;
end

always begin: Cell_ready
   if (cell == REAL CELL)
      cell_ready = 1;
   ...
end

always begin: SEND_CELL
   if (not cell_ready)
      cell = IDLE CELL;

   SEND cell TO THE OUTPUT PORT;
   ...
end
```

In the normal case, *cell_ready* signal is set as soon as the real cell is ready, then this cell can be sent to the output port by the *SEND_CELL* block. Namely property $Egress\_P_4$: $\mathbf{AG}(NEWCELL\_READY = 1)$ is satisfied. However, if for some reason, the *cell_ready* signal is delayed which may be caused by additional delay units placed by the RTL designers to meet some timing specification or even by the clock distribution networks, the real cell will be covered by the *IDLE* cell in statements:

```
if (not cell_ready)
   cell = IDLE CELL;
```

where, the *IDLE* cell will be sent to the output port instead of the real cell. The real cell is lost and cannot be recovered any more, and the worst thing is that the system does not know a cell has been lost. In model checking, property $\mathbf{AG}(NEWCELL\_READY = 1)$ failed in this case and the counter example indicates that the real cell equals to the IDLE cell. To correct this bug, we can change the *SEND_CELL* block as follows to solve the problem.

```
always begin: SEND_CELL
   case (cell_ready)
   'false:   SEND IDLE CELL TO THE OUTPUT PORT;
   'true :   SEND cell TO THE OUTPUT PORT;
   endcase
   ...
end
```

The other two bugs are in the *Translate_head* block. The *Translate_head* block is responsible for translating the old *VPI_VCI_IN* of the incoming cell to the new *VPI_VCI_NEW* if there is a match for the old *VPI_VCI* in the *LOOKUP_TABLE*. The procedure is shown as follows.

```
while (!MATCH_FOUND && i <= MAX_CONNECTIONS)
   if (LOOKUP_TABLE[i].VPI_VCI_IN == VPI_VCI_IN ) begin
      MATCH_FOUND = 1;
      VPI_VCI_NEW = LOOKUP_TABLE[i].VPI_VCI_NEW
   end
endwhile
```

where, *MAX_CONNECTIONS* is the number of items in the *LOOKUP_TABLE*.

If there is one item which *VPI_VCI_IN* domain equals the input *VPI_VCI_IN*, then *MATCH_FOUND* is set to 1 meaning that a match is found and *VPI_VCI_NEW* is set to the value of the *VPI_VCI_NEW* domain of the *LOOKUP_TABLE*. In this procedure, the value of *VPI_VCI_IN* should not be changed until the comparison is finished, namely *VPI_VCI_IN* has to be registered instead of wired. However, in the real design, *VPI_VCI_IN* is a wire variable instead of a register variable. In the case of wired variable, unless the timing between the *Translate_head* and its environment is perfectly matched, the design will lead to errors. In model checking, this corner case will lead to the failure of property $Ingress\_P_5$ i.e., $(\mathbf{AF}(((VPI\_VCI\_IN[27:4] = 0) \wedge (MATCH\_FOUND = 1))))$, because there is a path in the state space that $(VPI\_VCI\_IN[27:4] \neq 0) \wedge (MATCH\_FOUND = 1)$. This bug is solved by changing the data type of *VPI_VCI_IN* from *wire* to *reg*.

Another bug in this block is the statement

```
if (LOOKUP_TABLE[i].VPI_VCI_IN == VPI_VCI_IN ) begin
```

is mistaken as

```
if (LOOKUP_TABLE[i].VPI_VCI_IN == VPI_VCI_IN & 28'hFF7FFFF) begin
```

where 28 is the length of *VPI_VCI_IN*. In this case, cells with *VPI_VCI* equalling to $008000X$ are matched, but should not, since according to the specification, only

the cells with *VPI_VCI* equalling to $000000X$ can be matched. This bug actually is difficult to be found using simulation because one has to simulate that all the cells with *VPI_VCI* not equalling to $000000X$ cannot be matched. With formal verification, one can easily detect this bug using property $Ingress\_P_6$. According to this property, every state in the state space should be $(VPI\_VCI\_IN \neq 000000X) \wedge (MATCH\_FOUND = 0)$ provided that the incoming *VPI_VCI_IN* does not equal to $000000X$. This bug is also corrected by simply removing $\&28'hFF7FFFF$ in the *while* statement.

8.6 Summary

In this chapter, we demonstrated our compositional verification framework (Figure 1.4) against a Nortel 4×4 ATM switch fabric which is beyond the capability of model checking tool VIS. In the verification, we first decompose the fabric into many blocks, then we obtain the local properties with respect to each block. In the verification of each block, we first generate the environment for the blocks. If the block and the environment are within the capability of VIS, we apply VIS directly; else we have to reduce the block size using the syntactic model reduction approach. Because the reduction approach preserves the correctness of the properties, we can safely verify the reduced module. Finally, we can deduce the satisfaction of the global properties from the partial results. As a comparison, we also verify the blocks in FormalCheck. The results confirm that our techniques have advantages over FormalCheck when there are big datapaths in the system.

Chapter 9

Conclusion and Future Work

With the increasing complexity of large scale ASIC/SoC designs, simulation cannot cover all the corner cases in a reasonable time frame. Formal verification is emerging as a supplementary approach to mainstream simulation. Among various formal verification approaches, model checking is a fully automatic approach to verify a finite state machine against its temporal specifications. However, its application is limited by the size of the system to be verified. Current model checking tools [11, 61, 15, 22] are limited to several hundred boolean state variables due to state space explosion. There are two main methods to tackle this problem: *compositional verification* and *model reduction*. Compositional verification is to verify each partition in the system separately and then derive the system specification from the partial proofs. Model reduction is to reduce the size of the system such that it can be handled by a model checking tool. In this thesis, we integrate these two approaches to perform model checking.

In a compositional verification, properties are only true under certain environments. One of the problems in the compositional reasoning approach is to generate the *environment assumption*, i.e., stimulus for the module (partition) under verification. In our approach, we provide the environment assumptions as temporal logic formulas in **ACTL** and then construct the reduced tableau of the formulas, which

cover all the possible cases of the environment. In the construction, the formulas are interpreted over the tableau on a three-value domain so that the size of the tableau is smaller. We also deploy rewriting rules to simplify the formulas. Compared with existing related work, the proposed approach has a smaller environment size, which is a key factor in the compositional verification. Based on the reduced tableau, we synthesize Verilog modules, which are composed with the RTL design block under verification and fed into a model checking tool (here VIS [11]).

However, in case the size of the composed module is still beyond the capability of model checking, we use a novel syntactic model reduction algorithm based on cone of influence ideas, which analyzes the source code and removes the redundant variables and values. The reduction is based on the static analysis of control flow diagram of the program. The values of state variables in the program are partitioned into *active values*, and *deactive values* according to their dependency to the properties. The deactive values then can be replaced by an *abstract* value, and thus the value domains of the variables are much smaller than the original ones. Accordingly, we can have a reduced program with respect to the abstracted variables and the state space of the reduced program is smaller than that of the original one, while the correctness of the properties is preserved. Compared with existing related work, the proposed approach is an automatic approach and has better performance in the reduction of data-path intensive designs.

We provide formal proofs for the soundness of the proposed environment synthesis and syntactic model reduction approaches. The environment synthesis has been implemented in Java. The implementation of syntactic model reduction in C++ is in progress in the Hardware Verification Group of Concordia University.

To illustrate our approaches, we used a Nortel ATM (Asynchronous Transfer Mode) switch fabric as a real case study. We succeed in verifying this switch fabric, which size is beyond the capability of current plain model checking tools.

Future Research Directions

The proposed techniques open a way for the development of new verification tools. There are many opportunities for further work to improve the proposed approaches. We discuss here three selected open issues.

- *Versatile environment assumptions*

  For now, the proposed environment synthesis approach requires **ACTL** formulas to describe the environment of the module under verification, then the approach will synthesize the **ACTL** formulas into the Verilog program. This requires the verifier to have knowledge on the syntax and semantics of **ACTL**, which will limit its applicability in industry. So, it is very important to develop a user friendly environment description interface, which can accept easier and more understandable environment descriptions and then internally translate these engineer-oriented environment descriptions to **ACTL**.

- *Automatic design and property partition*

  In the compositional verification framework in Figure 1.4, there are two steps which are still manually done, namely, design and global property partitioning. This is a limitation, which may prevent its applications. The implementation of the automatic design and property partition is difficult because a verification tool does not have knowledge on how the design is partitioned and how the local properties are connected to each module in the design. A possible solution is that the designers put design constraints (local properties) into the RTL code so that the verification tool can get the information on how the system is partitioned by analyzing the constraints. In this way, the verification tool can establish the link between the design and the constraints. This is the so-called "assertion based HDL language" approach, which is an active research area and is being prompted by Open Verilog International and VHDL International.

- *Automata reduction*

  In the reduced tableau construction, we used two approaches to reduce the size of the tableau. One is the rewriting, and the other is the three-value domain. These two methods can reduce the size of the input **ACTL** formulas and the number of states in the tableau, respectively. However, besides these, another possible approach to reduce the tableau size further is to work on the tableau directly and to try to reduce the number of states in the tableau $\mathcal{K}$ (automata) and generate a new tableau $\mathcal{K}'$ such that $\mathcal{K}$ and $\mathcal{K}'$ are equivalent with respect to their languages. One such approach is proposed by K. Etessami et. al [32] to reduce an automata generated by linear temporal logic (**LTL**) tableau construction. The approach is based on the color-refinement algorithm of [45] and it eliminates the situation that one transition from a state "subsumes" another. In our tableau construction procedure, a similar approach can be used. However, more extensive explorations are needed in this topic.

Appendix A

Examples of Synthesized Environments

A.1 $Ingress\_P_1$

The **ACTL** environment assumption of properties $Ingress\_P_1$ is

$$\mathbf{AF}(B0 \wedge \mathbf{AX}(B1 \wedge \mathbf{AX}(B2 \wedge \mathbf{AX}(B3 \wedge \mathbf{AX}(B4 \wedge EOCN)))))$$

The synthesized Verilog code [1] of this assumption is shown as follows. Lines L0 to L8 are comments. *B0, B1, B2, B3, B4, EOCN* are set as outputs of the module *tableau*. Lines L10 to L14 are to declare the variables. Lines L15 to L25 are to set the initial states, where *S_INIT_W* indicates the initial states and *S_INIT_W_TMP* is a temporary variable. In Lines L26 to L35, wire variables *Sx_NEXT_W* describe the transitions of the states, i.e., what is the next state of current state *Sx*. *Sx_NEXT_W_TMP* are the temporary variables. Lines L36 to L41 are the non-deterministic assignments of the outputs. Lines L42 to L86 are the behaviors of this environment.

```
L0:  //`define TRUE 1'b1
```

[1] Verilog subset acceptable in VIS model checker

```
L1:  //`define FALSE 1'b0
L2:  //`define S0 0
L3:  //`define S1 1
L4:  //`define S2 2
L5:  //`define S3 3
L6:  //`define S4 4
L7:  //`define S5 5
L8:  //`define S6 6

L9:  module tableau(B4,B3,B2,B1,B0,EOCN);
L10: output B4,B3,B2,B1,B0,EOCN;

L10: //Variable declaration
L11: reg B4,B3,B2,B1,B0,EOCN;
L12: wire B4ND_W,B3ND_W,B2ND_W,B1ND_W,B0ND_W,EOCNND_W;
L13: reg [2:0] STATE;
L14: wire [2:0] S_INIT_W_TMP, S_INIT_W, S0_NEXT_W, S1_NEXT_W,
S2_NEXT_W, S3_NEXT_W, S4_NEXT_W, S5_NEXT_W, S6_NEXT_W;

L15: //Initialiazation
L16: assign S_INIT_W_TMP = $ND(0, 1, 2, 3, 4, 5, 6);
L17: assign S_INIT_W = ((S_INIT_W_TMP == 2) || (S_INIT_W_TMP == 3) ||
                        (S_INIT_W_TMP == 4) || (S_INIT_W_TMP == 5) ||
                        (S_INIT_W_TMP == 6)) ? 1 : S_INIT_W_TMP;

L18: initial begin
L19:     STATE = S_INIT_W;
L20:     B4 = 0;
```

```
L21:      B3 = 0;
L22:      B2 = 0;
L23:      B1 = 0;
L24:      EOCN = 0;
L25: end // Initial

L26: //Combinational part
L27: assign S1_NEXT_W = 2;
L28: assign S2_NEXT_W = 3;
L29: assign S3_NEXT_W = 4;
L30: assign S4_NEXT_W = 5;
L31: assign S5_NEXT_W = 6;
L32: assign S6_NEXT_W = 6;
L33: wire [2:0] S0_NEXT_W_TMP;
L34: assign S0_NEXT_W_TMP = $ND (0, 1, 2, 3, 4, 5, 6);
L35: assign S0_NEXT_W = ((S0_NEXT_W_TMP == 3) || (S0_NEXT_W_TMP == 4) ||
                         (S0_NEXT_W_TMP == 5) || (S0_NEXT_W_TMP == 6))
                         ? 2 : S0_NEXT_W_TMP;
L36: assign B4ND_W = $ND(0, 1);
L37: assign B3ND_W = $ND(0, 1);
L38: assign B2ND_W = $ND(0, 1);
L39: assign B1ND_W = $ND(0, 1);
L40: assign B0ND_W = $ND(0, 1);
L41: assign EOCNND_W = $ND(0, 1);

L42: //Sequential part
L43: always begin
L44:    case (STATE)
```

```
L45:     1: begin
L46:          B0= 1;
L47:          STATE = S1_NEXT_W;
L48:        end
L49:     2: begin
L50:          B1= 1;
L51:          STATE = S2_NEXT_W;
L52:        end
L53:     3: begin
L54:          B2= 1;
L55:          STATE = S3_NEXT_W;
L56:        end
L57:     4: begin
L58:          B3= 1;
L59:          STATE = S4_NEXT_W;
L60:        end
L61:     5: begin
L62:          B4= 1;
L63:          EOCN= 1;
L64:          STATE = S5_NEXT_W;
L65:        end
L66:     6: begin
L67:          B4 = B4ND_W;
L68:          B3 = B3ND_W;
L68:          B2 = B2ND_W;
L70:          B1 = B1ND_W;
L71:          B0 = B0ND_W;
L72:          EOCN = EOCNND_W;
```

```
L73:          STATE = S6_NEXT_W;
L74:        end
L75:     0: begin
L76:          B4 = 0;
L77:          B3 = 0;
L78:          B2 = 0;
L79:          B1 = 0;
L80:          B0 = 0;
L81:          EOCN = 0;
L82:          STATE = S0_NEXT_W;
L83:        end
L84:     endcase // case (STATE)
L85: end // always begin
L86: endmodule // tableau
```

The fairness constraint generated is shown as follows, namely one of the next states has to be asserted infinitely often.

```
(tableau.STATE[2:0] = 1
 || tableau.STATE[2:0] = 2
 || tableau.STATE[2:0] = 3
 || tableau.STATE[2:0] = 4
 || tableau.STATE[2:0] = 5
 || tableau.STATE[2:0] = 6
);
```

A.2 *Ingress_P*$_5$

The **ACTL** environment assumption of properties *Ingress_P*$_5$ is

$$\mathbf{AF}(VPI\_VCI\_IN[27:4] = 0)$$

The synthesized Verilog code of this assumption is shown as follows. Lines L0 to L5 are comments. *VPI_VCI_IN[27:4]* is set as an output of the module *tableau*. Lines L9 to L12 are to declare the variables. Lines L14 to L18 are to set the initial states, where *S_INIT_W* indicates the initial states and *S_INIT_W_TMP* is a temporary variable. In Lines L19 to L25, wire variables *Sx_NEXT_W* describe the transitions of the states, i.e., what is the next state of current state *Sx*. *Sx_NEXT_W_TMP* are the temporary variables. Lines L26 to L49 are the non-deterministic assignments of *VPI_VCI_IN[27:4]*. Lines L50 to L70 are the behaviors of this environment.

```
L0: //`define TRUE 1
L1: //`define FALSE 0
L2: //`define S0 0
L3: //`define S1 1
L4: //`define S2 2
L5: //`define S3 3
L6: module tableau(VPI_VCI_IN);
L7: output[27:4] VPI_VCI_IN;
L8: //Variable declaration
L9: reg [27:4] VPI_VCI_IN;
L10: wire [27:4] VPI_VCI_INND_W;
L11: reg [1:0] STATE;
L12: wire [1:0] S_INIT_W_TMP, S_INIT_W,
                S0_NEXT_W, S1_NEXT_W,
                S2_NEXT_W, S3_NEXT_W;
L13: //Initialiazation
L14: assign S_INIT_W_TMP = $ND(0, 1, 2, 3);//$
L15: assign S_INIT_W = ((S_INIT_W_TMP == 3)) ?
           2 : S_INIT_W_TMP;
```

```
L16: initial begin
L17:     STATE = S_INIT_W;
L18: end // Initial
L19: //Combinational part
L20: assign S2_NEXT_W = 3;
L21: assign S3_NEXT_W = 3;
L22: assign S1_NEXT_W = 1;
L23: wire [1:0] S0_NEXT_W_TMP;
L24: assign S0_NEXT_W_TMP = $ND (0,1,2,3);//$
L25: assign S0_NEXT_W = ((S0_NEXT_W_TMP == 1)
     || (S0_NEXT_W_TMP == 3)) ?
        2 : S0_NEXT_W_TMP;
L26: assign VPI_VCI_INND_W[4] = $ND( 0, 1);
L27: assign VPI_VCI_INND_W[5] = $ND( 0, 1);
L28: assign VPI_VCI_INND_W[6] = $ND( 0, 1);
L29: assign VPI_VCI_INND_W[7] = $ND( 0, 1);
L30: assign VPI_VCI_INND_W[8] = $ND( 0, 1);
L31: assign VPI_VCI_INND_W[9] = $ND( 0, 1);
L32: assign VPI_VCI_INND_W[10] = $ND( 0, 1);
L33: assign VPI_VCI_INND_W[11] = $ND( 0, 1);
L34: assign VPI_VCI_INND_W[12] = $ND( 0, 1);
L35: assign VPI_VCI_INND_W[13] = $ND( 0, 1);
L36: assign VPI_VCI_INND_W[14] = $ND( 0, 1);
L37: assign VPI_VCI_INND_W[15] = $ND( 0, 1);
L38: assign VPI_VCI_INND_W[16] = $ND( 0, 1);
L39: assign VPI_VCI_INND_W[17] = $ND( 0, 1);
L40: assign VPI_VCI_INND_W[18] = $ND( 0, 1);
L41: assign VPI_VCI_INND_W[19] = $ND( 0, 1);
```

```
L42: assign VPI_VCI_INND_W[20] = $ND( 0, 1);
L43: assign VPI_VCI_INND_W[21] = $ND( 0, 1);
L44: assign VPI_VCI_INND_W[22] = $ND( 0, 1);
L45: assign VPI_VCI_INND_W[23] = $ND( 0, 1);
L46: assign VPI_VCI_INND_W[24] = $ND( 0, 1);
L47: assign VPI_VCI_INND_W[25] = $ND( 0, 1);
L48: assign VPI_VCI_INND_W[26] = $ND( 0, 1);
L49: assign VPI_VCI_INND_W[27] = $ND( 0, 1);
L50: //Sequential part
L51: always begin
L52:    case (STATE)
L53:    0: begin
L54:        VPI_VCI_IN[27:4] = 1;
L55:        STATE = S0_NEXT_W;
L56:       end
L57:    1: begin
L58:        VPI_VCI_IN[27:4] = 1;
L59:        STATE = S1_NEXT_W;
L60:       end
L61:    2: begin
L62:        VPI_VCI_IN[27:4] = 0;
L63:        STATE = S2_NEXT_W;
L64:       end
L65:    3: begin
L66:        VPI_VCI_IN = VPI_VCI_INND_W;
L67:        STATE = S3_NEXT_W;
L68:       end
L69:    endcase // case (STATE)
```

```
L70: end // always begin
L71: endmodule // tableau
```

The fairness constraint generated is shown as follows, namely one of the next states has to be asserted infinitely often.

```
(tableau.STATE = 1
 || tableau.STATE = 2
 || tableau.STATE = 3
);
```

Appendix B

The Satisfiability Problem

The SATisfiability (SAT) problem is a core problem of a large family of computationally intractable NP-complete problems. Such problems have been identified as central to a number of areas in computing theory and engineering. An instance of the SAT problem is a Boolean formula that has three components [37]:

- A set of n variables: $x_1, x_2, \ldots, x_n$.
- A set of literals. A literal is a variable ($Q = x$) or a negation of a variable ($Q = \neg x$).
- A set of m distinct clauses: $C_1, C_2, \ldots, C_m$. Each clause consists of only literals combined by just *or* ($\vee$) connectives.

The goal of the satisfiability problem is to determine whether there exists an assignment to truth-values to variables that makes the following Conjunctive Normal Form (CNF) formula satisfiable:

$$C_1 \wedge C_2 \wedge \ldots \wedge C_m,$$

where $\wedge$ is a logic *and* connective.

Since SAT is NP-complete, it is unlikely that any SAT algorithm has a fast worst-case time behavior, namely the worst-case time is exponential [37]. However,

clever algorithms can rapidly solver many SAT formulas of practical interest. There has been great interest in designing efficient algorithm to solve most SAT formulas. In practice, SAT is fundamental in solving many problems in automated reasoning, computed-aided design, machine vision, and integrated circuit design automation and computer network design. Therefore, methods to solve SAT formulas play an important role in the development of efficient computing systems [37].

SATO [89, 88] is one of the provers for the SAT problem of propositional logic based on Davis-Putnam method [27]. It uses a set of optimization methods in the analysis of the CNF propositional logic formulas. Compared with other SAT tools, it is the most efficient one for the moment [88].

SATO expects an CNF formula in DIMACS or Lisp format [89]. For example, an CNF formula $(\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2)$ can be expressed as $(-120, 1 - 20)$ in DIMACS format or $((-12), (1 - 2))$ in Lisp format. SATO accepts this formula and will give the models satisfying the formula, i.e., $x_1 = x_2 =$ **true** or $x_1 = x_2 =$ **false**. However, for formula $(x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1))$, no matter what you try, there is no satisfying assignment since x_1 must be **false** (third clause), so x_2 must be **false** (second clause), but then the first clause is unsatisfiable.

Bibliography

[1] Y. Abarbanel, I. Beer, L. Gluhovsky, K. Keidar, and Y. Wolfsthal. FoCs: Automatic generation of simulation checkers from formal specifications. In *Proceedings of Computer Aided Verification*, LNCS 1855, pages 538–542, Chicago, IL, USA, July 2000. Springer Verlag.

[2] F. Van Aelten, S. Y. Liao, J. Allen, and S. Devadas. Automatic generation and verification of sufficient correctness properties for synchronous processors. In *Proceedings of IEEE International Conference on Computer Aided Design*, pages 183–188, Santa Clara, California, USA, November 1992. IEEE computer society press.

[3] R. Alur, L. de Alfaro, T. Henzinger, and Y. C. Mang. Automating modular verification. In *Proceedings of Concurrency Theory*, LNCS 1664, pages 82–97, Eindhoven, the Netherlands, August 1999.

[4] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: modularity in model checking. In *Proceedings of Computer Aided Verification*, LNCS 1427, pages 521–525. Springer Verlag, Vancouver, BC, Canada, June/July 1998.

[5] R. Alur, T. A. Henzinger, and S. K. Rajamani. *Symbolic exploration of transition hierarchies. In Tools and Algorithms for Construction and Analysis of Systems 98.* LNCS 1384, 1998.

[6] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods in System Design: An International Journal*, 15(1):7–48, July 1999.

[7] A. Arora, P. C. Attie, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 173–182, Puerto Vallarta, Mexico, June 1998.

[8] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Proceedings of Computer Aided Verification*, LNCS 1427, pages 319–331, Vancouver, BC, Canada, June/July 1998. Springer Verlag.

[9] J. Bergeron. *Writing Testbenches: Function Verification of HDL Models.* Kluwer Academic Publishers, 2000.

[10] J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra.* Elsevier, 2001.

[11] R. K. Brayton et al. VIS: A system for verification and synthesis. In *Proceedings of Computer Aided Verification*, LNCS 1102, pages 428–432. Springer Verlag, Rutgers University, NY, USA, July 1996.

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

[13] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, USA, June 1990.

[14] C. Capellmann, R. Demant, F. Fatahi-Vanani, R. Galvez-Estrada, U. Nitsche, and P. Ochsenschlaeger. Verification by behavior abstraction: a case study of service interaction detection in intelligent telephone networks. In *Proceedings of*

International Conference on Computer Aided Verification, LNCS 1102, pages 466–478, New Brunswick, NJ, USA, July/August 1996. Springer Verlag.

[15] Cadence Design Systems. *Technical manual of FormalCheck*, v2.3 edition, 1987-1999.

[16] L. Claesen, F. Proesmans, E. Verlind, and H. De Man. A methodology for the automatic verification of MOS transistor level implementations from high level behavioral specifications. In *Proceedings of the International Workshop on Formal Methods in VLSI Design*, Miami, Florida, USA, January 1991.

[17] E. M. Clarke and E. A. Emerson. Design and synthesis of a synchronization skeletons using branching time temporal logic. In *Proceedings of the Logic of Programs Workshop*, Yorktown Heights, NY, USA, May 1981.

[18] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finites-tate concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.

[19] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counter-example guided abstraction refinement. In *Procceding of Computer Aided Verification*, LNCS 1855, pages 154–169, Chicago, IL, USA, July 2000.

[20] E. M. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, Sept 1994.

[21] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking. In *Nato ASI*, volume 152 of *F*. SpringerVerlag, 1996.

[22] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[23] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.

In *Proceedings of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, USA, 1977.

[24] D. Cyrluk and M. K. Srivas. Theorem proving: Not an esoteric division, but the unifying framework for industrial verification. In *Proceedings of IEEE International Conference on Computer Design*, pages 538–545, Austin, Texas, USA, October 1995. IEEE computer society Press.

[25] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *Proceedings of 11th International Conference on Computer Aided Verification*, LNCS 1633, pages 160–172, Trento, Italy, July 1999. Springer Verlag.

[26] Jim Davies. *Using Z: specification, refinement, and proof.* Prentice-Hall, 1996.

[27] M. Davis, G. Longemann, and D. Loveland. A machine program for theorem proving. *Communication of the ACM*, 5(7):394-297, July 1962.

[28] M. Dwyer, J. Hatcliff, R. Joehanes, et al. Tool-supported program abstraction for finite-state verification. In *Proceedings of The 23rd International Conference on Software Engineering*, pages 53–71, Toronto, Ontario, Canada, May 2001.

[29] M. B. Dwyer and C. S. Pasareanu. Filter-based model checking of partial systems. In *Proceedings of the ACM SIGSOFT Sixth International Symposium on the Foundation of Software Engineering*, Lake Buena Vista, Florida, USA, November 1998.

[30] E. A. Emerson. *Temporal and modal logic, Handbook of theoretical computer science.* Elsevier Sciences B.V. J. van Leeuwn North-Holland edition, 1990.

[31] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[32] K. Etessami and G. Holzmann. Optimizing Büchi automata. In *Proceedings of Concurrent Theory*, LNCS 1877, Penn State University, State College, Pennsylvania, USA, August 2000. Springer Verlag.

[33] N. Francez. *Program Verification.* Addison-Wesley, 1992.

[34] M. Fujita. RTL design verification by making use of data-path in formation. In *Proceedings of the 1992 IEEE International Conference on Computer Design*, pages 592-597, Cambridge, MA, USA, October 1992. IEEE Computer Society Press.

[35] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

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

[37] J. Gu, P. W. Purdom, J. Franco, and B. W. Wah. Algorithms for satisfiability (SAT) problem: A survey. *Discrete Mathematics and Theoretical Computer Science: Satisfiability (SAT) Problem*, 35:19-152, 1997.

[38] A. Gupta. Formal hardware verification methods. *Formal Methods in System Design*, 1(2/3):151-238, October 1992.

[39] T. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In *Proceedings of Computer Aided Verification*, LNCS 1427, pages 440–451, Vancouver, BC, Canada, June/July 1998. Springer Verlag.

[40] T. Henzinger, S. Qadeer, and S. K. Rajamani. Assume-guarantee refinement between different time scales. In *Proceedings of Computer Aided Verification*, LNCS 1633, pages 208–221, Trento, Italy, July 1999. Springer Verlag.

[41] T. Henzinger, S. Qadeer, S. K. Rajamani, and S.Tasiran. An assume guarantee rule for checking simulation. In *Proceedings of Formal Methods in Computer Aided Design*, volume 1522 of *LNCS 1522*, pages 421–432. Springer Verlag, 1998.

[42] G. J. Holzmann. *Design and validation of computer protocols.* Prentice Hall, 1991.

[43] G. J. Holzmann. The Spin model checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[44] G.J. Holzmann and D. Peled. An improvement in formal verification. In *Proceedings of International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols*, pages 177–194, Bern, Switzerland, 1994.

[45] P. C. Kanellakis and S. A. Smolka. CCS expressioins, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, 1990.

[46] S. Katz, D. Geist, and O. Grumberg. Have I written enough properties? - a method of comparison between specification and implementation. In *Proceedings of Conference on Correct Hardware Design and Verification Methods*, LNCS 1703, pages 280–297, Bad Herrenalb, Germany, September 1999. Springer Verlag.

[47] S. Katz and O. Grumberg. Reduced tableau for safety ACTL. unpublished notes, July 2000.

[48] S. Katz and H. Miller. Saving space by fully exploiting invisible transitions. *Formal Methods in System Design*, 14(3):311–322, May 1999.

[49] C. Kern and M. R. Greenstreet. Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2):123-193, 1999.

[50] Y. Kesten and A. Pnueli. Modularization and abstraction: the key to practical formal verification. In *Proceedings of the 23rd International Symposium Mathematical Foundations of Computer Science*, pages 212–231, Brno, Czech Republic, 1998.

[51] S. A. Kripke. Outline of a theory of truth. *Journal of Philosophy*, 72:690–716, 1975.

[52] T. Kropf. *Introduction to Formal Hardware Verification.* New York, Springer, 1999.

[53] O. Kupferman and M. Y. Vardi. On the complexity of branching modular model checking. In *Proceedings of Concurrent Theory*, pages 408–422, Philadelphia, Pennsylvania, USA, August 1995. Extended abstract.

[54] R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigun. Verifying hardware in its software context. In *Proceedings of International Conference on Computer Aided Design*, pages 742–749, San Jose California, USA, 1997.

[55] R. P. Kurshan. *Computer Aided verification of coordinating processes.* Princeton University Press, 1994.

[56] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, USA, January 1985.

[57] D. E. Long. *Model Checking, Abstraction, and Compositional Verification.* PhD thesis, CMU, 1993.

[58] M. Maidl. The common fragment of ctl and ltl. In *41st Annual Symposium on Foundations of Computer Science*, pages 643–653, Redondo Beach, California, November 2000.

[59] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer Verlag, New York, 1992.

[60] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Safety.* Springer Verlag, New York, 1995.

[61] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[62] K. L. McMillan. Compositional rule for hardware design refinement. In *Proceedings of Computer Aided Verification*, LNCS 1254, pages 24–35, Haifa, Israel, June 1997. Springer Verlag.

[63] K. L. McMillan. Tutorial on model checking, 1998. http://www-cad.eecs.berkely.edu/kenmcmil/.

[64] K. L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *Proceedings of Computer Aided Verification*, LNCS 1427, pages 102–121, Vancouver, BC, Canada, June/July 1998. Springer Verlag.

[65] K. L. McMillan. Compositional methods and symbolic model checking. In *Proceedings of Symbolic Model Checking Workshop*, pages 21–42, Trento, Italy, July 1999.

[66] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[67] K. S. Namjoshi and R. P. Kurshan. Syntactic program transformations for automatic abstraction. In *Proceedings of Computer Aided Verification*, LNCS 1855, pages 433–449, Chicago, IL, USA, July 2000. Springer Verlag.

[68] S. Palnitkar. *Verilog HDL: A Guide to Digital Design and Synthesis.* Prentice Hall, 1st edition, Feb. 1996.

[69] D. Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science: 5th GI-Conference*, pages 167–183, Karlsruhe, Germany, March 1981.

[70] C. S. Pasareanu, M. B. Dwyer, and M. Huth. Assume-guarantee model checking of software: A comparative case study. In *Proceedings of SPIN Workshop*, pages 168–183, Trento, Italy, June 1999.

[71] H. Peng, Y. Mokhtari, and S. Tahar. Model reduction based on value dependency. In *Proceeding of IEEE International ASIC/SOC Conference*, Washigton, DC, USA, September 2001.

[72] H. Peng, Y. Mokhtari, and S. Tahar. Environment synthesis for compositional model checking. In *Proceeding of IEEE International Conference on Computer Design*, Freiburg, Germany, September 2002. IEEE computer society Press.

[73] H. Peng and S. Tahar. Survey on compositional verification. Technical report, Department of Electrical and Computer Engineering, Concordia University, Montreal, Canada, November 1998.

[74] H. Peng and S. Tahar. Compositional verification of an ATM Bit Error Rate MONitor Circuit. Technical report, Department of Electrical and Computer Engineering, Concordia University, Montreal, Canada, March 1999.

[75] H. Peng and S. Tahar. Compositional verification of ip based designs. In *Proceedings of IFIP International Workshop on IP Based Synthesis and System Design*, Grenoble, France, December 1999.

[76] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, 1981.

[77] A. Pnueli. In transition for global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI series. Series F*, pages 123–144. Springer Verlag, 1984.

[78] H. Saidi. Modular and incremental analysis of concurrent software systems. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, Cocoa Beach, Florida, USA, October 1999.

[79] D. A. Schmidt and B. Steffen. Data-flow analysis as model checking of abstract interpretations. In *Proceedings of the 5th Static Analysis Symposium*, pages 170–183, Pisa, Italy, September 1998.

[80] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide.* Computer Science Laboratory, SRI International, Menlo Park, California, USA, September 1999.

[81] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formula. In *Proceedings of Computer Aided Verification*, LNCS 1855, pages 248–263, Chicago, IL, USA, July 2000. Springer Verlag.

[82] Summit design. *VisualHDL user's guide*, v6.0 edition, 1994-1999.

[83] Northern Telecom. *Specification of a 4*4 ATM switch*, November 1998.

[84] M. Y. Vardi. On the complexity of modular model checking. In *Proceedings of the 10th IEEE Symposium on Logic in Computer Science*, pages 101–111, June 1995.

[85] P. Viresh, M. Nazanin, and V. Ranga. Automatic data path abstraction for verification of large scale designs. In *Proceedings of International Conference on Computer Design*, pages 504–509, Austin, Texas, USA, October 1998. IEEE computer society Press.

[86] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, pages 184-192, St. Petersburgh Beach, Florida, USA, January 1986.

[87] K. Yorav. *Exploiting syntactic structure for automatic verification.* PhD thesis, Israel institute of technology, June 2000.

[88] H. Zhang. SATO: An efficient propositional prover. In *Proceeding of International Conference on Automated Deduction 97*, volume 1249 of *LNAI*, pages 272–275, Berlin, German, July 1997.

[89] H. Zhang and M. E. Stickel. Implementing the Davis-Putnam algorithm by tries. Technical report, University of Iowa, Iowa City, 1994.