System-on-Chip Protocol Compliance Verification Using Interval Property Checking

Verifikation von System-on-Chip-Protokollimplementierungen durch intervallbasierte Eigenschaftsprüfung

Vom Fachbereich Elektrotechnik und Informationstechnik
der Technischen Universität Kaiserslautern
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
genehmigte Dissertation

von
M.Sc. Duc-Minh Nguyen
geb. in Hanoi, Vietnam

D 386

Dekan:	Univ.-Prof. Dr. Steven Liu
Gutachter:	Prof. Dr.-Ing. Wolfgang Kunz, Technische Universität Kaiserslautern Prof. Dr.-Ing. Hans Eveking, Technische Unversität Darmstadt
Datum der Disputation:	16 Januar 2009

Acknowledgments

This thesis is the result of 5 year work done at the Electronic Design Automation Group at the University of Kaiserslautern. Additionally, the problem of false negatives in interval property checking was discovered during a short but intensive stay at the Formal Verification Section in Infineon Technologies AG, that is now OneSpin Solutions GmbH. There are a numerous people who have contributed to this research in various ways and whom I would like to thank.

First of all, I am deeply indebted to my advisor, Prof. Wolfgang Kunz. He has provided me not only the opportunity to work in very interesting research field but also considerable guide and constant support during my long stay with his group. The ideas described in this thesis were greatly enriched during the long, intensive discussions with Wolfgang in which he has been very patient and encouraging.

I would like to thank Prof. Hans Eveking for reviewing this thesis and contributing helpful feedback.

The industrial designs and the verification tool were provided by OneSpin Solutions GmbH, Infineon, and Siemens, for which I appreciate. I owe special thanks to Dr. Klaus Winkelmann for his first introduction of practical verification problems when I was at the Formal Verification Section in Infineon.

I am very grateful to Dominik Stoffel, Markus Wedler for the time they spent proofreading and correcting my papers and also for their fruitful collaborations. Many thanks to Max Thalmaier for helping to conduct experiments with the BDD-based algorithms. I would like to thank Sacha Loitz for his proof reading of this thesis. Furthermore, I would like to thank the other colleagues in the Design Automation Group in Kaiserslautern, especially Ingmar Neumann, Evgeny Karibaev, Evgeny Pavlenko, Bernard Schmidt and Carmen Vicente for the interesting fun talks and support.

I am indebted my mother and my Late father for their great personal sacrifices when I was in Germany away from their difficulties. Last, but not least, I would like to thank Chi, my wife, for her understanding and caring.

Nguyen Duc Minh Kaiserslautern, February 4th, 2009.

To my parents

Contents

Chapter 1

Introduction

1.1 Hardware Verification of System-on-Chip Designs

In present-day hardware design flow, one of the most challenging tasks is to guarantee that designs are free of bugs. While design tools and technologies for *system-on-chip* (SoC) and reusable *Intellectual Property Components* (IP cores) allow more complex hardware systems to be designed effectively the verification methodologies are less effective. This leads to an enlarging productivity gap between design and verification such that verification can constitute about 70% of the total design effort. Therefore, there are increasing demands on effective verification flows that can help to reduce the verification cost.

In verification practice, simulation is a broadly adopted technique for all verification tasks. Although simulation can detect simple and easy-to-find bugs it often fails to find corner-case bugs. Especially, when a design becomes large only a portion of the design behavior can be triggered and be checked by applying input patterns (simulation stimuli) to the design. Generating stimuli that make the design exhibit an interesting, corner-case behavior is difficult. As a consequence, there are hard-to-detect bugs that escape from being detected by simulation techniques. To detect such bugs, formal verification can be used as an alternative to simulation techniques.

Formal verification analyzes a mathematical model of the design and proves that the design exhibits the desired behavior for all possible input patterns. Hence, in principle, formal verification can detect all bugs in the design model. Because of this advantage with respect to simulation, formal verification plays an increasingly important role in system-on-chip design. There are two main application domains of formal verification: *Equivalence Checking* and *Property Checking*.

1.1.1 Equivalence Checking

Equivalence Checking is to verify the functional equivalence of two models of the same design. The design models are functionally equivalent if they produce the same output behavior for all possible input patterns. One of the two models serves as the original model, which is assumed correct (golden design). The other model results from the former

after several modification or optimization steps are applied. Those modifications and optimization steps are done either manually by a designer or automatically by a synthesis tool. This can introduce errors into the design because of human mistakes or due to bugs of the automatic tools. Hence, it must be verified that the transformations of the design have not changed the function of the design after each design step. The task of equivalence checking is to guarantee the correctness of all automatic or manual transformation steps that are applied to the design.

Formal equivalence checking techniques have been well studied and developed for recent decades [Kun93, JMF95, Mat96, KK97, SWWK04]. One of the first successful techniques is proposed by Kunz [Kun93] for combinational designs and is extended by Stoffel [SWWK04] for sequential designs. This technique is based on the fact that there are many similarities between the structures of the golden model and the modified model. The similarities are exploited to analyze and to prove the equivalence between two models. The formal equivalence checking techniques are successfully applied to verify large designs that consist of millions of gates. Now, they are standard components in most industrial design flows.

1.1.2 Property Checking

Property Checking is used to verify the intended behavior of a design. At the beginning of the design process, an initial model of the design is created from the informal specification. From this specification a designer can also formulate properties that he wants the design to fulfill. The properties are verified to be valid for the initial model of the design. Property checking cannot only find bugs in the initial phase of the design process; it can also help to avoid misunderstandings of the specification by the designer. Therefore, bugs can be detected in an earlier phase of the design process with less cost.

Properties can be checked against the design models using *theorem proving* or *model checking*. In contrast to theorem proving that needs human interaction, model checking is a fully automated formal verification method. Hence, it requires less manual efforts than theorem proving.

Because this thesis will focus on improvements in model checking, in the following, the incoming data and the core algorithms of model checking will be described briefly.

Design Models and Property Languages

The two incoming data models for property checking algorithms are mathematical models of the properties and the design.

The mathematical models of the design are used to describe the combinational or sequential behavior of the design. The combinational behavior of the design is the functional relation among the inputs and outputs of the design at a certain time. The sequential behavior of the designs is the functional relation among the inputs and outputs of the design over time. The combinational behavior can be described by propositional logic in terms of input and output variables of the design. In a concrete hardware design, where input

and output variables are encoded by Boolean signals, the theory of Boolean (switching) algebra is often used. In order to describe the sequential behavior of the design, the *finite state machine* (FSM) model is used. The FSM model of the design can be translated to a *state transition graph* (STG) or a labeled STG, i.e., a so-called *Kripke model*.

In order to describe the ordering of desired events happening during the operation of the design, the properties can be formulated as an automaton (for example, Büchi automaton) or in a temporal logic language such as *Computation Tree Logic (CTL)* defined by Clarke and Emerson [CE81] or *Linear Temporal Logic (LTL)* defined in [Kur94].

Two typically desired types of properties are *safety property* and *liveness property*. A safety property requires that the design never exhibit any bad behaviors. A liveness property is to check that the design eventually exhibits an intended behavior.

Core Approaches

Model checking [CE81] is used to verify properties that are expressed in temporal logic against a design described as a Kripke model. The method is an iterative fixed-point computation which traverses the state transition graph of the Kripke model and checks the property in each state of the graph until the whole graph is traversed, i.e., the fixed point is reached. The algorithm calculates the set of reachable states of the design starting from the initial states or from the property states, i.e., the states in which the property is satisfied. As proposed by Clarke in [CE81], it is a graph traversal algorithm where the state sets are enumerated and stored explicitly. This is obviously infeasible when the number of states is large. In general, the state space to be traversed grows exponentially with the number of state components in the design. This is a serious problem which is known as state explosion problem. This problem limits the size of the design that can be verified in practice. Therefore, efficient data structures of the mathematical models as well as data manipulation operations are crucial for the success of model checking approaches.

Data Structures and Decision Algorithms

Binary decision diagrams (BDDs), proposed by Bryant [Bry86], are efficient, canonical representations of Boolean functions. In other words, many practically relevant Boolean functions can be represented by compact BDDs and every distinct function has exactly one unique BDD representation. Moreover, the manipulation operations on BDDs have polynomial complexity in terms of the size of BDDs. Such properties make BDDs a useful data structure for model checking approaches.

McMillan proposed *Symbolic Model Checking (SMC)* [McM93] where the sets of states are calculated and stored implicitly by BDDs. Instead of enumerating state sets, the characteristic functions of the sets of states as well as of the sets of transitions among states are represented by BDDs symbolically. By introducing symbolic model checking, larger designs with more than 10^{20} reachable states can be automatically verified. This improvement in the capability of model checking enables it to be applied in real indus-

trial verification flows. However, in some cases the size of BDDs is still exponential in the number of state components. Therefore, in spite of intensive research in this area, symbolic model checking can only handle designs with up to few hundred state variables. Many industrial-size designs remain too complex for symbolic model checking techniques.

Recently, *Boolean Satisfiability (SAT)* has emerged as a promising reasoning approach for model checking and formal verification. A *SAT-instance* (or a SAT-problem) is usually given as a *conjunctive normal form (CNF)* of a Boolean function. A SAT-instance is solved by searching a variable assignment under which the function evaluates to true. In recent years, many advances in SAT-solvers [MMZ+01, ES03] have been developed so that significantly larger SAT-problems with thousands of variables can be solved.

Biere et.al. proposed *Bounded Model Checking (BMC)* [BCCZ99] where the model checking problem is mapped on a SAT-instance. In BMC, the design is modeled by an iterative circuit model (or so-called time frame expansion) where the design is unrolled for a certain number of time frames. The transition relation of the iterative model and an LTL property are translated into the SAT-instance. The SAT-solver will solve the SAT-instance to decide if there exists a run of the design, i.e., a sequence of states starting from initial states that violates the property. BMC has been used successfully for finding bugs in many industrial designs. However, BMC often fails to prove that the property holds in the design where the complete proof requires the design to be unrolled for a large number of time frames (the diameter of the FSM modeling the design).

Improvements in Model Checking

As mentioned above, symbolic model checking suffers from the state explosion problem while bounded model checking is only good for falsification. Many enhancements have been proposed to overcome these limitations. Here, a few common approaches are summarized.

Researchers suggested *decomposition approaches* to reduce the size of the problem [BCL91, GB94, HKB96, CHJ+90, CM90, CCLQ97, CHM+96a, GDHH98]. The design, its state space, and its transition functions can be divided into separate partitions, which are considered individually. This results in many feasible problems, however, it may also result in inaccurate solutions, i.e., approximation solutions. The precision of the approximation is obtained by heuristic partitioning methods or by repeated analysis of the partitions.

In contrast to decomposition, where all partitions of the design are considered, *abstraction refinement approaches* [CGJ+00,CGKS02,WLJ+06,CCK+02,MA03,GGYA03, JKSC08] check only a small abstract model of the design against the property. The abstract model is constructed conservatively such that if the property holds in the abstract model it will also hold in the design. However, similar to decomposition approaches, inaccuracy of the abstraction may lead to spurious falsification of the property (so-called *false negatives* or *false counterexamples*). The spurious counterexamples are eliminated by refining the abstract model. The refinement process is often guided by the counterexamples

in well-known *Counterexample-Guided Abstraction Refinement (CEGAR) approaches*.

The incompleteness problem of BMC can be solved by induction proofs [SSS00]. In induction proofs, the property is proven in two steps: base step and inductive step. In the base step, the property is proven to hold for k time frames starting from initial states. The inductive step proves that if the property holds for k time frames starting from any arbitrary state, then it also holds for time frame $k + 1$.

Another SAT-based model checking technique called *interval property checking (IPC)* is applied very successfully in industrial practice. It has its origin already in the mid 1990s in the Siemens company and is commercially available [One09]. In the IPC methodology, bounded behaviors of the design are specified using the so called *Interval Language (ITL)*. ITL properties are checked against the time frame expansion model of the design starting from an arbitrary state.

1.1.3 Block-based Verification of SoC Designs

In spite of continuous progress in formal verification technology, a complete SoC design clearly remains beyond the capacity of a state-of-the-art formal property checker. Therefore, formal property checking is usually applied at the block level (block-based verification) and is used to achieve high quality IP blocks. In this way, also simulation at the chip level becomes less expensive since it can concentrate on the global system behavior and is relieved from hunting bugs in the individual SoC modules.

In an SoC design, IP blocks (modules) such as hardware accelerators, I/O controllers, or programmable blocks such as digital signal processors and standard processor cores are connected via sophisticated communication infrastructures. These communication structures usually use some standard protocols, e.g., ARM's *Advanced Microcontroller Bus Architecture (AMBA) Advanced High-performance Bus (AHB)* [ARM99]. These protocols are used to create common interfaces between modules and ensure proper behavior of the entire chip.

Consequently, verification of IP blocks can be divided into two different tasks: computational verification and communicational verification.

Computational verification is to assure the correctness of individually intended behaviors of a module, for example, the module computes data correctly. Communicational verification is to verify the communication interface of the IP block. When it comes to ensuring system integrity, it is of great importance to verify that the components connected to the system busses comply with the respective protocols. This certifies that the verified IP block does not compromise the on-chip communication when connected to the on-chip communication infrastructure. Communicational verification is also called protocol compliance verification. Such a certification becomes especially important when IP blocks from third-party vendors are used. While computational verification for arithmetic data path has made fast progress recently [WSBK07], communicational verification is still a major area of potential innovation in industrial SoC verification.

1.2 Motivation and Thesis Overview

Even though the state-of-the-art in formal verification technology provides a rich set of methods and tools to verify the interfaces between SoC modules formally, severe problems still occur in practice.

Since the control designs for standard SoC protocols typically contain thousands of state variables all methods that rely on an exact traversal of the state space are prohibitively expensive. More advanced methods are available that decompose the state space and/or perform abstractions on it. However, most of these methods are of generic nature and have been developed for a general spectrum of property checking applications. Unfortunately, they tend to fail in industrial practice when being applied specifically to the interfaces of modules. Either their state space approximation is too coarse resulting in false negatives of the property checker, or they run out of computational resources. BMC, in principle, can handle the complexity of SoC module interfaces, however, proofs on bounded time intervals often remain incomplete when it considers the design starting from initial states.

Provided enough resources, IPC can provide a complete proof of a property. It has been proven very efficient for a wide range of computational verification tasks. However, it has to cope with the problem of identifying reachability information while handling communicational verification. Because IPC deals with the time frame expansion model of the design, starting from arbitrary states it can produce false negatives. The false negatives correspond to runs of the design starting from unreachable states. If this happens, the model needs to be augmented with important reachability information in order to rule out the spurious counterexamples. In the present-day IPC methodology flow, invariants have to be identified manually by the verification engineer by analyzing the counterexamples and the source code of the design. This requires tedious and time-consuming manual effort, and, reduces the productivity of the formal verification process.

Additionally, in present-day verification methodology flows, verification engineers try to construct a correct and complete set of properties. A property can be erroneous if it specifies an unrealizable behavior of the design. An incomplete property set fails to cover all indented behaviors of the design and then fails to identify bugs in uncovered behaviors. Therefore, constructing a set of properties is also a challenging task, and reduces productivity of the formal verification process. However, protocol compliance verification has some special characteristics when compared to general computational verification. In principle, the properties for protocol compliance can be formulated independently of the particular design under verification so that they can be re-used for other designs as well. It is therefore beneficial to have a generic, reusable set of protocol compliance properties.

Because of the two problems being addressed above, applying formal property checking still requires a significant amount of manual effort that sometimes hampers adoption of the formal verification methodology flow by industry.

Therefore, the purpose of this thesis is to improve the productivity of the industrial, IPC-based formal verification methodology. First, this thesis proposes a decomposition-based reachability analysis to solve the problem of identifying reachability information

automatically. Second, this thesis develops a generic, reusable set of properties for protocol compliance verification.

The thesis is organized as follows. Chapter 2 introduces the reader into some fundamentals and terminology of graph algorithms and representation of Boolean functions which are basic to the subject of this thesis.

Chapter 3 introduces the concepts of model checking and reviews some recent results on model checking techniques.

Chapter 4 describes the interval property checking techniques and discusses the common property template in ITL. It will outline the problems that reduce the productivity of the IPC-based verification methodology flow. This chapter also discovers an interesting feature of the common property template and the corresponding hierarchical structure of designs, namely the existence of a dedicated central finite state machine that controls the overall behavior of the design. This central finite state machine is called main-FSM and is used to formulate properties.

Chapter 5 develops a new decomposed FSM traversal algorithm to identify reachability information, which is used to strengthen the models in the IPC-based methodology and helps to eliminate the spurious counterexamples. The new algorithm exploits the main-FSM to decompose the reachability analysis problem and to reduce the computational complexity of the traversal algorithm. Experiments have been conducted to demonstrate the effectiveness of this approach in increasing the productivity of the IPC-based methodology.

Chapter 6 develops a new verification methodology for protocol compliance that uses a recorder *finite state structure (FST)* and a set of properties, which are generic and reusable. It develops a systematic way to construct the recorder-FST from the informal specification. The recorder-FST represents the bus protocol status. It is used to formulate the generic set of ITL properties. It will be shown how recorder-FST and the property set are checked against protocol implementation designs to assure that the designs comply with the protocol standard. This can be done by using IPC-based methodology and the FSM traversal algorithm. The chapter illustrates the usefulness of the proposed methodology by developing a representative recorder-FST and a property set for ARM AMBA and by using them to verify several industrial designs.

Chapter 7 concludes the thesis and plans some future directions of research.

Chapter 2

Fundamentals

In the first section of this chapter, basic graph algorithms are reviewed because they serve as primary algorithms from which most algorithms in this thesis are derived. In the second section, the basic notations of Boolean functions and its common representations are briefly introduced. The graph algorithms are summarized of graph chapters in [CLR94]. For a more detailed introduction into the Boolean functions and their representations, the reader may refer to a standard textbook, e.g., [HS96].

2.1 Graphs and Graph Algorithms

2.1.1 Definitions

In formal verification of hardware designs, many problems can be modeled by graphs and the algorithms on these problems are based on graph algorithms. For example, the circuit of a design can be modelled as a directed graph whose vertices are electronic gates. The sequential behavior of a design can be modeled as a state transition graph. Traversing a circuit or analyzing the state transition graph of the design is a graph traversal. The concepts of graphs and graph algorithms are fundamental to all subjects that will be presented in this thesis.

Definition 2.1. *(***Graph***) A graph G is a tuple (V, E), where*

- *V is a set of vertices (or nodes),*
- *$E \subset V \times V$ is a set of edges,*

E is a binary relation on V. Each edge $e \in E$ connects two nodes $u, v \in V$, i.e., $e = (u, v)$. If the edge set E consists of unordered pairs of vertices, G is an undirected graph, otherwise G is a directed graph. □

In a directed graph, given an edge $e = (u, v)$, the node u is the *immediate predecessor* of v, and the node v is called the *immediate successor* of u. The set of all immediate successors of a given node u is also referred to as the *fanout* of the node, which is denoted

by $FanOut(u)$. Similarly, the set of all immediate predecessors of a node u are called its *fanin* being denoted by $FanIn(u)$.

A *path* of length k from a vertex u to a vertex u' in a directed graph G is a sequence $(v_1, v_2, \ldots, v_k)$ of vertices such that $u = v_1, u' = v_k$ and $(v_i, v_{i+1}) \in E$ for $i = 1, 2, \ldots, k-1$. If there is a path p from u to u', we say that u' is *reachable* from u via p, which is denoted as $u \overset{p}{\rightsquigarrow} u'$. In this case, u is called a predecessor of u' and u' is called a successor of u.

In a directed graph, a path $(v_1, \ldots, v_k)$ forms a cycle if $v_1 = v_k$ and the path contains at least one edge. A directed graph with no cycles is called *directed acyclic graph* (DAG).

A directed graph is *strongly connected* if every two vertices are reachable from each other. The *strongly connected components* (SCCs) of a graph are the equivalence classes of vertices under the *"are mutually reachable"* relation. In other words, a strongly directed component of a directed graph G is a maximal set of vertices $U \subset V$ such that for every pair of vertices $u, v \in V$, we have both $u \rightsquigarrow v$ and $v \rightsquigarrow u$; that is, vertices u and v are reachable from each other.

2.1.2 Graph Algorithms

Breadth-first Search Algorithm

The *breadth-first search (BFS)* algorithm is one of the basic algorithms for traversing a graph. It can be considered as the archetype of reachability analysis algorithms, which will be represented in Chapter 3 and Chapter 5.

Given a graph $G = (V, E)$ and a distinguished source vertex s, breadth-first search iteratively moves along the edges of G to visit all vertices that are reachable from s. The key idea of breadth-first search is to expand the set of reachable nodes systematically across the breadth of the traversed node. That is, the algorithm traverses all nodes at distance k from s before visiting any nodes at distance $k + 1$. The frontier between distance k and $k + 1$ in the graph is marked by coloring the discovered and undiscovered nodes with different colors. A queue is used to store the nodes that are at the frontier, i.e., at distance k. These nodes will be considered while visiting nodes at distance $k + 1$. Figure 2.1 shows the breath-first search algorithm as being presented in [CLR94].

Depth-first Search Algorithm

The *depth-first search (DFS)* algorithm explores the longest path in the graph whenever possible, i.e., as deeply as possible. A node v is marked as *"visited"* if all immediate successors u of v have been visited. Therefore, in order to visit the node u, the algorithm visits all its immediate successors recursively. The algorithm as shown in Figure 2.2 timestamps each node. Each node v has two timestamps: the first timestamp $d[v]$ records when v is first visited, and the second timestamp $f[v]$ records when v is marked as "visited".

```
1: BFS(V, E, s) {
2:     for each u ∈ V ∧ u ≠ s {
3:         color[u] ← WHITE;
4:         d[u] ← ∞;
5:         π[u] ← ∅;
6:     }
7:     color[s] ← GRAY;
8:     d[s] ← 0;
9:     π[s] ← ∅;
10:    Q ← {s};
11:    while Q ≠ ∅ {
12:        u ← head[Q]; {
13:        for each v ∈ Adj[u] {
14:            if color[v] = WHITE {
15:                color[v] ← GRAY;
16:                d[v] ← d[u] + 1;
17:                π[v] ← u;
18:                ENQUEUE(Q, v);
19:            }
20:        }
21:        DEQUEUE(Q);
22:        color[u] ← BLACK;
23:    }
24: }
```

Figure 2.1: Breadth-first search algorithm

```
1: DFS(V, E) {
2:     for each u ∈ V {
3:         color[u] ← WHITE;
4:         π[u] ← ∅;
5:     }
6:     time ← 0;
7:     for each u ∈ V {
8:         if color[u] = WHITE {
9:             DFS_VISIT(V, E, u);
10:        }
11:    }
12: }

13: DFS_VISIT(V, E, u) {
14:    color[u] ← GRAY;
15:    d[u] ← time ← time + 1 {
16:    for each v ∈ Adj[u] {
17:        if color[v] = WHITE {
18:            π[v] ← u;
19:            DFS_VISIT(V, E, u);
20:        }
21:    }
22:    color[u] ← BLACK;
23:    f[u] ← time ← time + 1;
24: }
```

Figure 2.2: Depth-first search algorithm

Strongly Connected Components

Another important graph algorithm that will be used in this thesis is decomposing a directed graph into its *strongly connected components (SCCs)*. This decomposition can be done using two depth-first searches on the directed graph G and on its *transpose graph* G^T. The transpose of a graph $G = (V, E)$ is a directed graph $G^T = (V, E^T)$, where $E^T = \{(u, v) : (v, u) \in E\}$. The algorithm to identify SCCs of a graph is shown in Figure 2.3.

```
1: SCC(V, E) {
2:     Compute finishing times f[u] ← DFS(V, E)
3:     Compute E^T;
4:     Call DFS(V, E^T), but in the main loop of DFS
5:       consider the vertices in order of decreasing f[u];
6:     For each vertex u in the main loop of DFS (line 7)
7:       output reachable vertices of u
7:       as vertices in one SCC corresponding to u;
8: }
```

Figure 2.3: Strongly connected component algorithm

2.2 Representations of Boolean Functions

2.2.1 Boolean Functions

We consider the two-valued *Boolean algebra* that is composed by a Boolean set $B = \{0, 1\}$, the operations of conjunction (AND), disjunction (OR) and complementation (NOT). The elements 0 and 1 of B are called *Boolean values* or *Boolean constants*. *Boolean variables*, denoted by lower case letters, for example x, y, v, can be assigned a Boolean value.

An *n-variable vector of Boolean variables*, denoted by an upper case letter, e.g., $X = \langle x_1, \ldots, x_n \rangle$, can be assigned an element of the n-dimensional Boolean space B^n. A point in the n-dimensional Boolean space B^n can be called an assignment to the Boolean vector X and is denoted as $A(X) = \langle a_1, \ldots, a_n \rangle$. Given an assignment $A(X)$ of a Boolean vector X, the value of a variable $x_i \in X$ is denoted by $A(x_i) = a_i$.

An n-variable Boolean function f, $f : B^n \mapsto B$ maps a point in the n-dimensional Boolean space to a Boolean value. The set of points of the Boolean space that are mapped to $1(0)$ is the *on-set(off-set)* of the Boolean function f, respectively.

A *vector of Boolean functions* $F : B^n \mapsto B^m$ consists of m individual Boolean functions $F = \langle f_1, \ldots, f_m \rangle$.

Given two Boolean functions f, g, a new function h can be obtained by applying Boolean operators to f and/or g. These applications of Boolean operators AND, OR, NOT and XOR are denoted by $h = AND(f, g) = f \wedge g$, $h = OR(f, g) = f \vee g$, $h = NOT(f) = \overline{f}$, $h = XOR(f, g) = f \oplus g$, respectively.

In addition to the standard Boolean operators such as AND, OR, NOT, etc., we are interested in other operators, which will be used in this thesis.

Definition 2.2. *(**Implication**) Given two Boolean functions f, g, the implication operator being denoted as $f \rightarrow g$ is defined by: $h = f \rightarrow g = \overline{f} \vee g$* □

The implication operator has some important properties which will be used later in this thesis.

Lemma 2.1. *Let f, g, h be Boolean functions. If $f \rightarrow g = 1$ and $g \rightarrow h = 1$, then $f \rightarrow h = 1$.*

Lemma 2.2. *Let f, g, h be Boolean functions. If $f \rightarrow g = 1$, then $f \wedge h \rightarrow g \wedge h = 1$.*

Lemma 2.3. *Let f, g, h be Boolean functions. If $f \wedge g = 0$ and $h \rightarrow f = 1$, then $h \wedge g = 0$.*

Proof. Lemmas 2.1, 2.2 and 2.3 hold according to the implication definition and properties of Boolean operations. □

Definition 2.3. *(**Cofactor**) Given an n-variable Boolean function $f(x_1, \ldots, x_n)$, the positive cofactor f_{x_j} and the negative cofactor $f_{\overline{x}_j}$ of f with respect to a variable x_j are:*

- $f_{x_j} = f(x_1, \ldots, x_{j-1}, 1, x_{j+1}, \ldots, x_n)$.
- $f_{\overline{x}_j} = f(x_1, \ldots, x_{j-1}, 0, x_{j+1}, \ldots, x_n)$. □

Based on the cofactors of a Boolean function, we can define its quantification as follows:

Definition 2.4. *(**Existential Quantifier**) Given an n-variable Boolean function $f(x_1, \ldots, x_n)$, the existential quantifier operator being applied to f with respect to a variable x_j is:*

$$\exists_{x_j} f = f_{x_j} \vee f_{\overline{x}_j}$$

□

Given a Boolean function $f(V)$, we can apply the existential quantifier with respect to a sub-vector of V, $X = \langle x_1, x_2, \ldots, x_n \rangle \subseteq V$ as follows:

$$\exists_X f(X, V \setminus X) = \exists_{x_1}(\ldots(\exists_{x_{n-1}}(\exists_{x_n} f(V \setminus X)))) \tag{2.1}$$

Lemma 2.4. *Let f be a Boolean function and X be a vector of variables. Then $f \rightarrow \exists_X f = 1$.*

Proof. Lemma 2.4 holds according to the quantifier definition and properties of Boolean operations. □

The cofactors can be generalized to apply to a function f with respect to another function g as defined below:

Definition 2.5. *(***Generalized Cofactor***) Given two n-variable functions $f : B^n \mapsto B$ and $g : B^n \mapsto B$. Let the distance between two points $A_1, A_2 \in B^n$ be given by*

$$D(A_1, A_2) = \sum_{i=1}^{i=n} 2^{n-i} \cdot (A_1(x_i) + A_2(x_i)) \bmod 2.$$

Then the generalized cofactor of F with respect to g being denoted by $f \downarrow g$ is:

$$f \downarrow g(X) = f(\mu_g(X))$$

where $\mu_g(X) : B^n \mapsto B^n$ maps each point $A_1 \in B^n$ to its nearest neighbor $A_2 \in B^n$ according to $D(A_1, A_2)$ such that $g(A_2) = 1$. □

The generalized cofactor of a function f with respect to g is a function that equals to f when $g = 1$. In other word, if we interpreted the pair (f, g) as an *incompletely specified* function whose onset is $f \wedge g$ and *don't care set* is $\overline{g}$, the generalized cofactor can be considered as a heuristic to select a representative of the incompletely specified function (f, c). Note that the generalized cofactor doesn't need to be uniquely determined.

Definition 2.6. *(***Support Set***) The support set of an n-variables Boolean function $f : B^n \mapsto B$ is*

$$supp(f) = \{x_j \in X | f_{x_j} \neq f_{\overline{x}_j}\}$$

For all $x \in supp(f)$, we say f depends on x. □

Boolean functions can be represented by combinational circuits, truth tables, *disjunctive normal forms (DNFs), conjunctive normal forms (CNF),* or *reduced ordered binary decision graphs (ROBDDs).* In the following, a few of these representations are briefly described.

2.2.2 Combinational Circuits

Definition 2.7. *(***Syntax of Combinational Circuits***) A combinational circuit, (a combinational network, or a circuit gate netlist), is a DAG $(I \cup O \cup G, E, Op)$, where*

- *The set of vertices is partitioned in the following way:*
  - *The set of primary inputs $I \subseteq V$ is a set of nodes without predecessors, i.e., $FanIn(i) = 0$,*

  - *The set of primary outputs $O \subseteq V$ is a set of nodes without successors. Every primary output has only one fan-in, i.e., $FanOut(o) = 0$ and $|FanIn(o)| = 1$,*
  - *The set of internal gates G,*

- *E is the set of edges connecting two gates, and edges are called signals,*
- *$Op : V \mapsto Ops$ is the labeling function that maps gates to gate types:*

$$Ops = \{PI, PO, AND, NAND, OR, NOR, NOT, XOR, \ldots\}$$

The labeling function Op is defined such that:

- *For all inputs $i \in X$, $Op(i) = PI$,*
- *For all outputs $o \in O$, $Op(o) = PO$,*
- *For all gates $g \in G$, $Op(g) \in Ops \setminus \{PI, PO\}$.* □

Definition 2.8. *(***Semantics of Combinational Circuits***) Given a combinational circuit (V, E, Op) and a vector of Boolean variables $X = \langle x_1, \ldots, x_n \rangle$, where $n = |I|$. Each gate $g \in V$ is associated with a Boolean function $f^g : (B^n \mapsto B)$ which is defined recursively as follows:*

- *if $g \in I$, then there exists only one $x \in X$ such that $f^g = x$,*
- *if $g \in O$ and $FanIn(g) = g_1$, then $f^g = f^{g_1}$,*
- *if $g \in G$, and $FanIn(g) = \{g_1, g_2, \ldots, g_k\}$ then*

$$f^g = Op(g)(f^{g_1}, f^{g_2}, \ldots, f^{g_k}).$$

□

A combinational circuit is displayed as a directed graph in which gates are depicted differently according to their gate types. Table 2.1 illustrates some standard gate types and their corresponding operations being applied to the Boolean functions.

An example of a combinational circuit is shown in Figure 2.4. In the example, primary inputs i_1, i_2, i_3 are associated with Boolean variables x_1, x_2, x_3, respectively. For gate g_3 with $Op(g_3) = OR$ and $FanIn(g_3) = \{g_1, g_2\}$, the associated Boolean function is $f^{g_3} = g_1 \vee g_2$. Applying Definition 2.8 recursively results in the function associated with the primary output o $f^o = (\overline{x}_1 \wedge x_2) \vee (x_1 \wedge x_3)$

The combinational circuit representation of Boolean functions is used in several important algorithms like *Automatic Test Pattern Generation (ATPG), structural equivalence checking* [Kun93], *structural FSM traversal* [SWWK04], *circuit-based satisfiability solver* [KGP01].

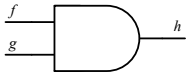
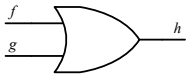
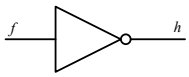
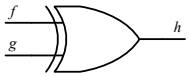
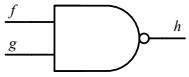
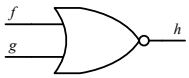
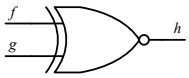
Gate symbol	Name	Boolean operator	Characteristic function as CNF
	AND	$h = f \wedge g$	$(\overline{v}_h \vee v_f) \wedge (\overline{v}_h \vee v_g) \wedge$ $(v_h \vee \overline{v}_f \vee \overline{v}_g)$
	OR	$h = f \vee g$	$(v_h \vee \overline{v}_f) \wedge (v_h \vee \overline{v}_g) \wedge$ $(\overline{v}_h \vee v_f \vee v_g)$
	INVERTER	$h = \overline{f}$	$(v_h \vee v_f) \wedge (\overline{v}_h \vee \overline{v}_f)$
	XOR	$h = f \wedge \overline{g} \vee \overline{f} \wedge g = f \oplus g$	$(\overline{v}_h \vee \overline{v}_f \vee \overline{v}_g) \wedge (\overline{v}_h \vee v_f \vee v_g) \wedge$ $(v_h \vee \overline{v}_f \vee v_g) \wedge (v_h \vee v_f \vee \overline{v}_g)$
	NAND	$h = \overline{f \wedge g}$	$(v_h \vee v_f) \wedge (v_h \vee v_g) \wedge$ $(\overline{v}_h \vee \overline{v}_f \vee \overline{v}_g)$
	NOR	$h = \overline{f \vee g}$	$(\overline{v}_h \vee \overline{v}_f) \wedge$ $(\overline{v}_h \vee \overline{v}_g) \wedge (v_h \vee v_f \vee v_g)$
	XNOR	$h = f \wedge g \vee \overline{f} \wedge \overline{g} = \overline{f \oplus g}$	$(v_h \vee \overline{v}_f \vee \overline{v}_g) \wedge (v_h \vee v_f \vee v_g) \wedge$ $(\overline{v}_h \vee \overline{v}_f \vee v_g) \wedge (\overline{v}_h \vee v_f \vee \overline{v}_g)$

Table 2.1: Standard gates and corresponding Boolean operators, CNFs

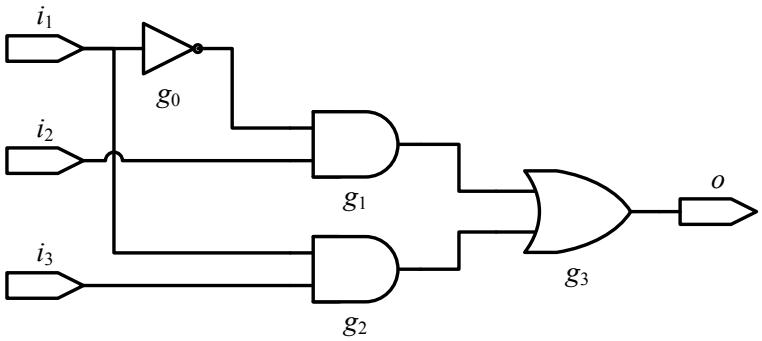


Figure 2.4: Example of a combinational circuit

2.2.3 Reduced Ordered Binary Decision Diagrams

Bryant [Bry86] introduced *reduced ordered binary decision diagrams* (ROBDDs) that are canonical representations of Boolean functions. The canonicity of the representations is very important in many applications, especially in formal verification.

The *ordered binary decision diagram (OBDD)* representation of a Boolean function is constructed recursively based on Shannon's expansion theorem. Each node $v \in V$ of the graph is associated with a Boolean input variable in the vector $X = (x_1, x_2, \ldots, x_n)$ or a Boolean value *value*$(v) \in B$. If a node is associated with a Boolean value, it is called a leaf node. For each non-leaf node v, there are two edges from v to two "children" *low*(v) and *high*(v).

Definition 2.9. *(**Syntax of OBDDs**) Given a vector of Boolean variables $X = \langle x_1, x_2, \ldots, x_n \rangle$, an OBDD is a labeled DAG $(V \cup L, E, var, val)$, where:*

- *V is a set of n nodes,*
- *L is a set of leaf nodes,*
- *E is a set of edges such that*
  - *leaf nodes do not have immediate successor,*
  - *each node $v \in V$ has only two immediate successor $low(v)$ and $high(v)$, which are called two "children" of v,*
- *$var : V \mapsto X$ is an attribute function which maps a node $v \in V$ to a Boolean variable such that if $var(v) = x_j$, $var(low(v)) = x_k$ and $var(high(v)) = x_l$, then $j > k$ and $j > l$,*
- *$val : L \mapsto B$ is a value function which maps a leaf $v \in L$ to a Boolean value.* □

Definition 2.10. *(**Semantics of OBDDs**) Given an OBDD and its labeling variables X, each node $v \in V \cup L$ is associated with a Boolean function f^v which is defined recursively as follows:*

- *if $v \in L$, then $f^v = val(v)$,*
- *if $v \in V$ with $var(v) = x_j$ and two children $low(v)$, and $high(v)$, then*

$$f^v = (\overline{x}_j \wedge f^{low(v)}) \vee (x_j \wedge f^{high(v)}).$$

□

An OBDD can be reduced to obtain a ROBDD such that it contains no node v with *low*$(v) =$ *high*(v), nor any nodes $v, v' \in V$ such that the subgraphs rooted in v and v' are isomorphic, i.e., $f^v = f^{v'}$.

Bryant developed effective algorithms for Boolean manipulations using ROBDDs. For example, Boolean operations such as AND, OR, XOR, etc., being applied to functions

f_1 and f_2 can be calculated in polynomial time in the sizes of ROBDDs representing f_1 and f_2, i.e., $O(|f_1| \cdot |f_2|)$. Such ROBDD operations are implemented in the well-known package CUDD [Som].

A combinational circuit of Boolean functions can be translated into ROBDD representations by traversing the transpose graph of the gate netlist using depth-first search algorithm. In the algorithm shown in Figure 2.2, when a gate is marked as "visited", the corresponding ROBDD of the Boolean function that is associated with this gate is calculated by applying ROBDD operations on the ROBDDs of the intermediate successor.

The OBDD and ROBDD for the combinational circuit in Figure 2.4 is shown in Figure 2.5. In the figure, the edge from a node v to its $low(v)$ is denoted by a dotted line.

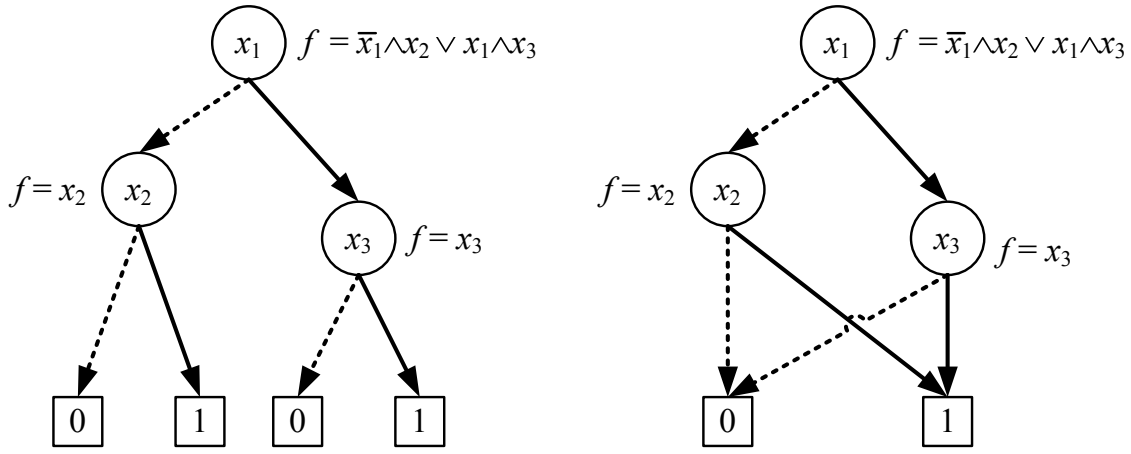


Figure 2.5: Example of the OBDD and the ROBDD for the circuit in Figure 2.4

2.2.4 Conjunctive Normal Forms

Definition 2.11. *(***Conjunctive Normal Forms***) Conjunctive Normal Forms (CNFs) represent Boolean functions as conjunctions of clauses, each clause is a disjunction of literals, and each literal is a Boolean variable or its negation.* □

A CNF of Boolean functions that are represented as a combinational circuit is a conjunction of all CNFs representing nodes of the circuit. Table 2.1 shows in the last column the CNF representing the corresponding gate. The CNF representing a combinational circuit can be constructed by conjoining CNFs of all gates in the circuit. For the circuit in

Figure 2.4, the corresponding CNF is:

$$\begin{aligned} CNF_o(x_1, x_2, x_3, g_0, g_1, g_2, g_3, o) = & (x_1 \vee g_0) \wedge (\overline{x}_1 \vee \overline{g}_0) \\ & (\overline{g}_1 \vee g_0) \wedge (\overline{g}_1 \vee x_2) \wedge (g_1 \vee \overline{g}_0 \vee \overline{x}_2) \\ & (\overline{g}_2 \vee x_1) \wedge (\overline{g}_2 \vee x_3) \wedge (g_2 \vee \overline{x}_1 \vee \overline{x}_3) \\ & (g_3 \vee \overline{g}_1) \wedge (g_3 \vee \overline{g}_2) \wedge (\overline{g}_3 \vee g_1 \vee g_2) \\ & (o \vee \overline{g}_3) \wedge (\overline{o} \vee g_3) \end{aligned} \tag{2.2}$$

The most important algorithm which operates on CNFs is the satisfiability algorithm. Given a CNF F, a SAT-solver finds a value assignment A to the variables of the CNF for which F evaluates to 1.

Most SAT-solvers are developed from the basic DPLL algorithm [DLL62]. Recent advances in SAT-solvers such as VSIDS decision heuristic, lazy two-literal watching scheme [MMZ+01], conflict-driven learning [MSS99] have enabled SAT-solvers to handle significant larger CNFs.

SAT-solvers can be modified to identify all satisfying assignments (all minterms) to the variables of F as proposed in [McM02]. The modified SAT-solvers, which are called *ALLSAT*-solvers continue to search for new solutions instead of terminating whenever a satisfying assignment is found. The assignments that have been found are ruled out by so-called *Blocking Clauses*. The blocking clause, which is in conflict with the satisfying assignment, is identified by analyzing the implication graph and added to the SAT-instance.

Chapter 3

Model Checking

This chapter introduces the concepts of model checking and reviews some recent results on model checking techniques. The first section defines the models which are used to describe the sequential behavior of a design and the model checking problem. In the second section, basic symbolic model checking based on BDDs is briefly described. The third section will review some related works that propose improved algorithms for symbolic model checking. Bounded model checking methods are presented in the last section.

3.1 Models and Properties of Sequential Systems

3.1.1 Models of Sequential Systems

Finite State Machines

In Section 2.2.2, it has been shown that Boolean functions can be represented by combinational circuits whose primary outputs correspond to Boolean functions. In other words, the Boolean functions can be considered as the mathematical model of the combinational circuits.

A sequential system, whose values of outputs depend not only on the present values of inputs but also on the past values of inputs, is often modeled as a *finite state machine (FSM)*.

Definition 3.1. *(***Finite State Machine***) A Finite State Machine (FSM) of Mealy-type is a 6-tuple* $(I, S, S_0, \delta, O, \lambda)$*, where:*

- I *is the input alphabet, i.e., a finite, non-empty set of input values,*
- S *is the (finite, non-empty) set of states,*
- $S_0 \subseteq S$ *is the set of initial states,*
- $\delta : S \times I \mapsto S$ *is the next-state function,*
- O *is the output alphabet,*

- $\lambda : S \times I \mapsto O$ *is the output function.* □

For a *Moore*-type FSM, the output function $\lambda : S \mapsto O$ does not depend on the present inputs.

In order to implement and represent an FSM in two-valued Boolean algebra, its input alphabet, output alphabet, and set of states S are encoded by a vector of *primary inputs (PIs)* $X = \langle x_1, x_2, \ldots, x_n \rangle$, a vector of *primary outputs (POs)* $Y = \langle y_1, y_2, \ldots, y_m \rangle$, and a vector of Boolean *state variables* $V = \langle v_1, v_2, \ldots, v_p \rangle$, respectively. This results in an *encoded FSM* being defined as follows:

Definition 3.2. *(***Encoded Finite State Machine***) A Boolean encoded finite state machine is a 6-tuple* $(I, S, S_0, \Delta, O, \Lambda)$*, where*

- $I \subseteq B^n$ *is the input alphabet,*
- $S \subseteq B^p$ *is the state set,*
- *the next-state function* $\Delta = \langle \delta_1, \delta_2, \ldots, \delta_p \rangle$ *is a vector of Boolean functions, where* $\delta_j \in \Delta$ *is the next-state function for the* next state variable v'_j*, i.e.,* $v'_j = \delta_j(V, X)$*,*
- $O \subseteq B^m$ *is the output alphabet,*
- *the output function* $\Lambda = \langle \lambda_1, \lambda_2, \ldots, \lambda_m \rangle$ *is a vector of Boolean functions, where* $\lambda_j \in \Lambda$ *is the output function for the primary output* y_j*, i.e.,* $y_j = \lambda_j(V, X)$*.* □

In an encoded FSM, a state $s \in S$ (an input letter $i \in I$, an output letter $o \in O$) is given a unique encoding. We can use the same notation s (i, o) for the state (the input letter, the output letter) and its encoding value. The value of a state variable $v_j \in V$ at a state $s \in S$ is denoted by $s(v_j)$. For a sub-vector $U = \langle v_j, v_{j+1}, \ldots, v_k \rangle$ of V and a state $s \in S$, we use the notation $s(U)$ to denote the projection of s on the state variables $v \in U$. Furthermore, with $\Delta_U = (\delta_j \ldots, \delta_k)$ we denote the Boolean functional vector of next-state functions for state variables in U. Similarly, the value of a primary input $x_j \in X$ (a primary output $y_j \in Y$) in an input (output) letter $i \in I$ ($o \in O$) is denoted by $i(x_j)$ ($o(y_j)$).

The FSM can be implemented as a sequential circuit which contains a set of *memory elements (latches)* and a combinational circuit. The latches are used to remember the values of state variables and to connect *next-state variables* $v'_j \in V'$ with the corresponding state variables $v_j \in V$. The combinational circuit represents the vector of next-state functions and the vector of output functions. The input vector of the combinational circuit is $X \cup V$, where $v \in V$ is called a *pseudo input (PSI)*. The output vector is $Y \cup V'$, where $v' \in V'$ is called a *pseudo output (PSO)*. Figure 3.1 shows a Mealy-type FSM being implemented as a sequential circuit.

An FSM without outputs can be viewed as a *finite state transition structure (FST)*: (I, S, S_0, Δ). With an FST we focus on modeling how the underlying sequential system changes its states in response to input symbols. Such state transitions of an FST are represented by the *transition relation* T being defined as follows:

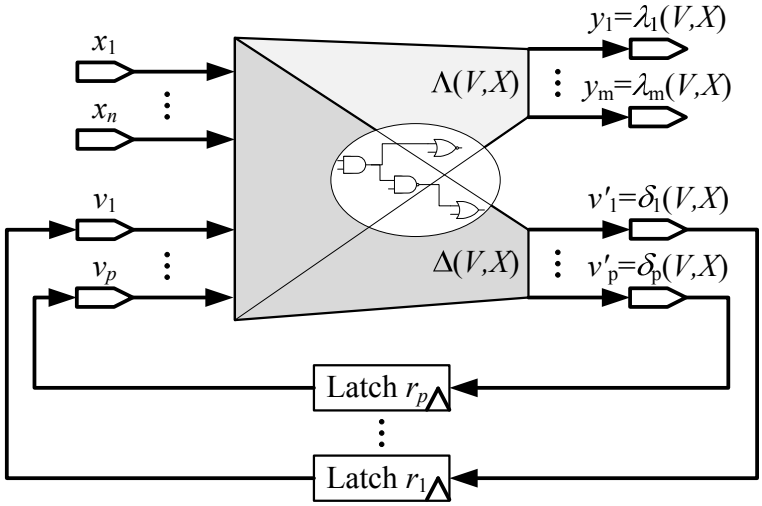


Figure 3.1: Sequential circuit implementing a Mealy FSM

Definition 3.3. *(***Transition Relation***) The transition relation of an FST* (I, S, S_0, Δ) *is a triple* $T \subseteq S \times I \times S$ *such that* $T = \{(s, i, s')|s' = \Delta(s, i)\}$. □

Note that in case the inputs are not of interest the transition relation can be quantified with respect to the inputs. This leads to a simplified transition relation as follows:

$$T = \{(s, s')|\exists i \in I : s' = \Delta(s, i)\}$$

In the sequel, if T is denoted as the function in terms of a pair of states this means the simplified transition relation.

Definition 3.4. *(***State Transition Graph***) An FST can be viewed as a labeled directed* state transition graph (STG) (S, E, L)*, where*

- *the set of states* S *of the FST is the set of vertices,*
- $E = \{(s, s')|\exists i \in I : s' = \Delta(s, i)\}$ *is the set of edges,*
- $L_{(s,s')} = \{i \in I|s' = \Delta(s, i)\}$ *is the label of the edge* $(s, s') \in E$. □

A behavior of the sequential system can be expressed as a *path* (or a *run*) of the FSM, which is defined as:

Definition 3.5. *(***Path of FSM***) Let* (S, E, L) *be the STG of an FSM, a path (or a run) of the underlying FSM is an infinite sequence of states,* $\pi = (s_1, s_2, \ldots)$ *such that for* $j \geq 1$*,* $(s_j, s_{j+1}) \in E$*. If the starting state* $s_1 \in S_0$ *we say that the path is initialized.* □

We denote the j-th state s_j in the path π by $\pi(j)$ and by $\pi_j = (s_j, s_{j+1}, \ldots)$ the suffix of π starting from state s_j.

Definition 3.6. *(***Diameter of FSM***) Consider an FSM* M*, the diameter* $rd(M)$ *is the length of the longest shortest path from an initial state to any reachable states in the STG of* M. □

Kripke Structures

Definition 3.7. *(**Kripke Structure**) Given a set of atomic Boolean propositions $\mathcal{AP}$, a Kripke structure is defined as a 4-tuple $M = (S, S_0, T, L)$, where*

- *S is the set of states,*
- *$S_0 \subseteq S$ is the set of initial states,*
- *$T \subseteq S \times S$ is the transition relation between states,*
- *$L : S \mapsto 2^{\mathcal{AP}}$ is the function that labels each state with a set of atomic propositions.* □

A Kripke structure can be derived from an FSM by taking a Cartesian product of S, I, and O as the set of states in the Kripke structure. In case the inputs and the outputs are not of interest the transition relation can be identical to the transition relation of the FSM.

We also use $\pi = (s_1, s_1, \ldots)$ to denote a path of the Kripke structure, where for $j \geq 1$ it is $(s_j, s_{j+1}) \in T$.

3.1.2 Property Languages

While sequential systems are described by Kripke structures the intended properties to be proven are usually formulated in temporal logic. There are two main types of temporal logic: *Computation Tree Logic (CTL)* and *Linear Temporal Logic (LTL)* that are both sub-logics of a more expressive computation tree logic CTL*.

For a Kripke structure M and an initial state, a *computation tree* is constructed by unwinding the structure into an infinite tree with the initial state as the root. The computation tree represents all possible initialized paths of the Kripke structure from the given initial state.

Formulas in CTL* describe properties of the computation tree. CTL* formulas consist of Boolean variables, Boolean operators and additional *path quantifiers* and *temporal operators*. The path quantifiers A ("for all computation paths") or E ("for some computation path") are used to specify that a property is considered in all paths or in some paths of the tree. The temporal operators are used to select the states along a path of the tree at which a property is considered. Given a path, the basic temporal operators are:

- X specifying that a property is considered at the second state of the path,
- F specifying that a property is considered at some states of the path,
- G specifying that a property is considered at all states of the path,
- U specifying that a property is considered at all states from the beginning of the path to the state at which another property is true,

- R specifying that a property is considered at all states from the beginning of the path to the state at which another property is false.

In practice, CTL or LTL are often used. In the sequel, the two languages will be briefly described.

Computation Tree Logic

Definition 3.8. *(**Syntax of CTL**) Let $\mathcal{AP}$ be a set of atomic propositions. State or path formulas in CTL are constructed by applying the following syntax rules recursively.*

1. *Each atomic proposition $\phi \in \mathcal{AP}$ is a CTL state formula.*
2. *If ϕ, ψ are state formulas, then $\phi \vee \psi, \phi \wedge \psi, \overline{\phi}$ are also state formulas.*
3. *If ϕ, ψ are state formulas, then $\mathsf{X}\,\phi$, $\mathsf{F}\,\phi$, $\mathsf{G}\,\phi$, $\phi\mathsf{U}\,\psi$, $\phi\mathsf{R}\,\psi$ are path formulas.*
4. *If ϕ is a path formula, then $\mathsf{A}\,\phi$, $\mathsf{E}\,\phi$ are state formulas.* □

Properties of a Kripke model are state formulas in CTL. They are always interpreted with the Kripke structure M. The notation $M, s \models \phi$ means that a state formula ϕ holds (or is true) in state s in the Kripke structure M. From the syntax definition of CTL, there are ten basics operators occurring in CTL state formulas which are AX , EX , AF , EF , AG , EG , AU , EU , AR , ER . Each of the ten operators can be expressed in terms of three operators EX , EG , E U . Therefore, we define the semantics of these basic operators as follows:

Definition 3.9. *(**Semantics of CTL**)*

- $M, s \models \phi$ *iff* $\phi \in L(s)$
- $M, s \models \overline{\phi}$ *iff* $M, s \not\models \phi$
- $M, s \models \phi \vee \psi$ *iff* $M, s \models \phi$ *or* $M, s \models \psi$
- $M, s \models \phi \wedge \psi$ *iff* $M, s \models \phi$ *and* $M, s \models \psi$
- $M, s \models \mathsf{EX}\,\phi$ *iff there exists a path* $\pi = (s_0, s_1, \ldots)$ *such that* $M, s_1 \models \phi$
- $M, s \models \mathsf{EG}\,\phi$ *iff there exists a path* $\pi = (s_0, s_1, \ldots)$ *such that for all* $j \geq 0$, $M, s_j \models \phi$
- $M, s \models \mathsf{E}\,\phi\mathsf{U}\,\psi$ *iff there exists a path* $\pi = (s_0, s_1, \ldots)$ *and* $j \geq 0$ *such that for all* $0 \leq k < j$, $M, s_k \models \phi$ *and* $M, s_j \models \psi$ □

The remaining CTL operators are determined as follows:

- $\mathsf{AX}\,\phi = \overline{\mathsf{EX}\,\overline{\phi}}$

- $\mathsf{EF}\,\phi = \mathsf{E}\,[True\ \mathsf{U}\ \phi]$
- $\mathsf{AG}\,\phi = \overline{\mathsf{EF}\,\overline{\phi}}$
- $\mathsf{AF}\,\phi = \overline{\mathsf{EG}\,\overline{\phi}}$
- $\mathsf{A}\,[\phi \mathsf{U}\,\psi] = \overline{\mathsf{E}\,[\overline{\phi}\ \mathsf{U}\ (\overline{\phi} \wedge \overline{\psi})]} \wedge \overline{\mathsf{E}\,\mathsf{G}\,\overline{\psi}}$
- $\mathsf{A}\,[\phi R \psi] = \overline{\mathsf{E}\,[\overline{\phi} \mathsf{U}\,\overline{\psi}]}$
- $\mathsf{E}\,[\phi R \psi] = \overline{\mathsf{A}\,[\overline{\phi} \mathsf{U}\,\overline{\psi}]}$

Linear Temporal Logic

LTL only has one path quantifier, A. Hence, the path quantifier is omitted from formulas. The syntax of LTL is very similar to the syntax of CTL, where atomic propositions $\phi, \psi \in \mathcal{AP}$ are LTL formulas, $\phi \vee \psi$, $\phi \wedge \psi$, $\overline{\phi}$ are LTL formulas, $\mathsf{X}\,\phi$, $\mathsf{G}\,\phi$, $\mathsf{F}\,\phi$, $\phi \mathsf{U}\,\psi$ are LTL formulas.

Since LTL does not express the branching time behaviors of the system the semantics of LTL is defined along paths of the Kripke structure. The notation $M, \pi \models \phi$ meaning that ϕ holds on $\pi = (s_0, s_1, \ldots)$ is defined as follows:

Definition 3.10. *(***Semantics of LTL***)*

- $M, \pi \models \phi$ *iff* $\phi \in L(s_0)$
- $M, \pi \models \overline{\phi}$ *iff* $M, \pi \not\models \phi$
- $M, \pi \models \phi \vee \psi$ *iff* $M, \pi \models \phi$ *or* $M, \pi \models \psi$
- $M, \pi \models \phi \wedge \psi$ *iff* $M, \pi \models \phi$ *and* $M, \pi \models \psi$
- $M, \pi \models \mathsf{X}\,\phi$ *iff* $M, \pi_1 \models \phi$
- $M, \pi \models \mathsf{G}\,\phi$ *iff for all* $j \geq 0$, $M, \pi_j \models \phi$
- $M, \pi \models \mathsf{F}\,\phi$ *iff there exists* $j \geq 0$ *such that* $M, \pi_j \models \phi$
- $M, \pi \models \phi \mathsf{U}\,\psi$ *iff there exists* $j \geq 0$ *such that for all* $0 \leq k < j$, $M, \pi_k \models \phi$ *and* $M, \pi_j \models \psi$ □

3.2 Symbolic Model Checking

3.2.1 Introduction of Model Checking

Definition 3.11. *(***Model Checking***) Given a Kripke structure $M = (S, S_0, T, L)$ and a CTL formula ϕ, the model checking problem is the problem of finding a set of states in which the property ϕ holds ($S_\phi = \{s | M, s \models \phi\}$) and checking if this set includes all initial states, i.e., $S_0 \subseteq S_\phi$.* □

The set S_ϕ can be referred to as the set of states satisfying the formulas ϕ. In the sequel, if it is clear from the context, we use the property formula to denote the set of property states. The set of states ϕ can be calculated recursively as follows:

- if ϕ is an atomic proposition $\phi = \{s | \phi \in L(s)\}$.
- $\mathsf{EX}\,\phi = Pre(T, \phi) = \{s' | \exists s \in \phi : (s', s) \in T\}$
- $\mathsf{EF}\,\phi = \mu Z.(\phi \vee \mathsf{EX}\,Z)$
- $\mathsf{EG}\,\phi = \nu Z.(\phi \wedge \mathsf{EX}\,Z)$
- $\mathsf{E}\,[\phi \mathsf{U}\,\psi] = \mu Z.(\psi \vee (\phi \wedge \mathsf{EX}\,Z))$

where μ and ν are least and greatest fixed-point operators, respectively. Figure 3.2 shows the procedure to calculate $\mathsf{EF}\,\phi$ as the least fixed-point computation. For the other operators, readers are referred to [CGP99].

```
1: EvalEF (M, φ) {
2:     Z ← φ;
3:     Z' ← Z ∪ Pre(T, Z);
4:     while Z' ≠ Z {
5:         Z ← Z';
6:         Z' ← Z ∪ Pre(T, Z);
7:     }
8:     return Z;
9: }
```

Figure 3.2: Backward reachability analysis to evaluate $\mathsf{EF}\,\phi$

The fixed-point algorithm in Figure 3.2 can be considered as the BFS algorithm which operates on the STG of the Kripke model. The algorithm starts with the set of states satisfying the formula $Z = \phi$ and iteratively visits predecessor states of Z by the *preimage computation* $Pre(T, Z)$. Therefore, the algorithms evaluating CTL formulas can be considered as a backward reachability analysis on the STG of the Kripke model.

As a special case, a safety property $\mathsf{AG}\,\phi$, where ϕ is an atomic formula, can be checked using forward reachability analysis instead of backward reachability analysis. The algorithm shown in Figure 3.3 checks the formula $\mathsf{AG}\,\phi$ by traversing the STG of the Kripke model M from the initial states. The algorithm starts with the set of initial states $Z = S_0$ and the set of "bad" states bad, in which ϕ fails, i.e., $\overline{\phi}$ holds. Then, it iteratively visits successor states of Z by the *img computation* $Img(T, Z) = \{s' | \exists s \in Z : (s, s') \in T\}$. At every image computation during the reachability analysis, the set of reachable states is checked against the set of "bad" states whether there is any bad state in the current set of reached states.

```
 1: CheckAG (M, φ) {
 2:     Z ← S_0;
 3:     bad ← ¬φ;
 4:     Z' ← Z ∪ Img(T, Z);
 5:     if Z' ∩ bad ≠ ∅ return fails;
 6:     while Z' ≠ Z {
 7:       Z ← Z';
 8:       Z' ← Z ∪ Img(T, Z);
 9:       if Z' ∩ bad ≠ ∅ return fails;
10:     }
11:     return holds;
12: }
```

Figure 3.3: Forward reachability analysis to check $\mathsf{AG}\,\phi$

The core computation steps of both algorithms, shown in Figure 3.2 and Figure 3.3, are *image* and *preimage* computations which calculate the immediate successors and predecessors of a set of states. *Explicit model checking* performs directly on the Kripke structure, i.e., on the explicit STG of the Kripke structure. This is done by considering all edges connecting the current set of states to the other states as in lines $13 - 20$ in the graph BFS algorithm in Figure 2.1. This is very complex and the major bottleneck in model checking. An effective way to perform the image computation is crucial in model checking. In the next subsection, we will discuss symbolic model checking where the image computations are performed using BDD manipulation.

3.2.2 Symbolic Model Checking

In symbolic model checking [McM93], the set of states as well as the transition relation are represented by BDDs. As consequence, the image computations are evaluated implicitly in a more effective way.

As mentioned in Section 3.1.1, the state sets are encoded by a vector of Boolean variables V. A state $s \in S$ is represented as a Boolean assignment $A = \langle a_1, a_2, \ldots, a_p \rangle$,

where $a_j \in B$ is the Boolean value being assigned to variable $v_j \in V$. Therefore, a set of states $S \subseteq B^n$ can be considered as a set of Boolean assignments whose characteristic function is defined as follows:

Definition 3.12. *(***Characteristic Function of Set***) Let a set of states $S \subseteq B^p$ be encoded by a p-variable Boolean vector $V = \langle v_1, v_2, \ldots, v_p \rangle$. The characteristic function of S is a unique Boolean function $\chi_S : B^p \mapsto B$ such that:*

$$\chi_S(s) = \begin{cases} 1 & \text{if } s \in S \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

□

Similarly, the transition relation can also be represented by its Boolean characteristic function. For an FSM with the vector of transition functions $\Delta = \langle \delta_1, \delta_2, \ldots, \delta_p \rangle$, the transition relation can be calculated by conjoining the transition relations for all state variables as in the following equation:

$$\chi_T(V, X, V') = \bigwedge_{j=1}^{j=p} (v'_j \equiv \delta_j(X, V)) \tag{3.2}$$

In the above equation, $T_j = (v'_j \equiv \delta_j(X, V))$ is called a *bit relation* and T is a *monolithic transition relation*.

In the sequel, the notations of sets and their characteristic functions are used interchangeably.

If the transition relation of an FSM is given, it is straightforward to compute the image or preimage of a state set using the Boolean existential quantifier as follows:

$$Z'(V') = Img(T, Z) = \exists_X, \exists_V T(V, X, V') \wedge Z(V) \tag{3.3}$$

Similarly, the preimage of the set of states Z' can be calculated as:

$$Z(V) = PreImg(T, Z') = \exists_X, \exists_{V'} T(V, X, V') \wedge Z'(V') \tag{3.4}$$

By using Equation 3.3 and Equation 3.4, immediate successor or predecessor states are visited in one calculation step instead of being visited individually as in explicit model checking.

3.2.3 State Space Explosion Problem

Model checking has the well-known *state space explosion problem*. This problem is caused by the fact that the state space of the system is of *exponential* size in the size of the system description. Hence, even relatively small systems can have huge Kripke structures which cannot be constructed. If the system is represented as a sequential circuit with p state variables the state space of the corresponding Kripke structure has up to 2^p states.

Symbolic model checking does not construct Kripke structures explicitly but represents them symbolically. Therefore, it can handle larger systems than explicit model checking. However, for many industrial systems symbolic model checking still faces the state space explosion problem. Several techniques were proposed to solve this problem. In the following section, two major methods will be described: decomposition and abstraction refinement.

3.3 Improvements of Symbolic Model Checking

There is a large amount of literature on previous work dealing with partitioning, approximation or abstraction in order to overcome the limitations of symbolic model checking when large designs need to be handled. This section outlines three types of techniques.

3.3.1 Improvement of Image Computation

As the first attempt, researchers try to improve the performance of the image computations as well as to reduce the size of the BDDs during the computations. It is done by partitioning techniques that decompose the characteristic functions of the state set and the transition relation into partitions which are small enough for symbolic representation. There are two main partitioning techniques: i) conjunction based techniques [BCL91, GB94, HKB96] and ii) splitting based techniques [CHJ$^+$90, CM90, CCQ96, CCLQ97, NIJ$^+$97, QCC96]. In addition, a hybrid technique that uses both partitioning techniques has been proposed in [MKRS00]. In the following, the underlying formulas of these techniques will be described shortly. For more details, the readers are referred to [MKRS00].

Conjunction based techniques do not calculate the monolithic transition function as in Equation 3.2 before quantifying the inputs and the present state variables as in Equation 3.3. Instead, the partitioned transition relation for groups of state variables are conjoined step by step followed by early quantification. For a group of state variables V_j, the conjunction step conjoins the transition relation $T_j(W)$ for V_j with the transition relation $T_k(U, W)$ for the state variables that have been considered before. Before conjunction, all variables U that appear only in $T_k(U, W)$ but not in $T_j(W)$ can be quantified:

$$\exists_U(T_k(U, W) \wedge T_j(W)) = \exists_U(T_k(U, W)) \wedge T_j(W) \tag{3.5}$$

Splitting based techniques decompose the image computation by means of input and output splitting. In input splitting, the image computation is decomposed with respect to an input variable or a present state variable $v \in V \cup X$ as by the following equation:

$$\begin{aligned} \exists_X \exists_V \bigwedge_{i=1}^{i=p} T_i(V, X, V') \wedge Z(V) = & \exists_X \exists_V \bigwedge_{i=1}^{i=p} T_i(V, X, V')_v \wedge Z(V)_v \vee \\ & \exists_X \exists_V \bigwedge_{i=1}^{i=p} T_i(V, X, V')_{\overline{v}} \wedge Z(V)_{\overline{v}} \end{aligned} \tag{3.6}$$

On the other hand, output splitting decomposes the image computation with respect to a next state variable $v'_j \in V'$ as follows:

$$\begin{aligned}\exists_X \exists_V \bigwedge_{k=1}^{k=p} T_k(V,X,V') \wedge Z(V) = & v'_j \wedge \exists_X \exists_V \bigwedge_{k=1}^{k=p} T_k(V,X,V')_{v'_j} \wedge Z(V) \vee \\ & \overline{v'}_j \wedge \exists_X \exists_V \bigwedge_{k=1}^{k=p} T_k(V,X,V')_{\overline{v'}_j} \wedge Z(V)\end{aligned} \tag{3.7}$$

Equation 3.7 can be further simplified by using the generalized cofactor of T_k with respect to δ_j and $Z(V)$:

$$\begin{aligned}\exists_X \exists_V \bigwedge_{k=1}^{k=p} T_i(V,X,V') \wedge Z(V) = & v'_j \wedge \exists_X \exists_V \bigwedge_{k=1}^{k=p,k\neq j} ((T_k(V,X,V') \downarrow Z(V)) \downarrow \delta_j) \vee \\ & \overline{v'}_j \wedge \exists_X \exists_V \bigwedge_{k=1}^{k=p,k\neq j} ((T_k(V,X,V') \downarrow Z(V)) \downarrow \overline{\delta}_j))\end{aligned} \tag{3.8}$$

3.3.2 Approximate FSM Traversal

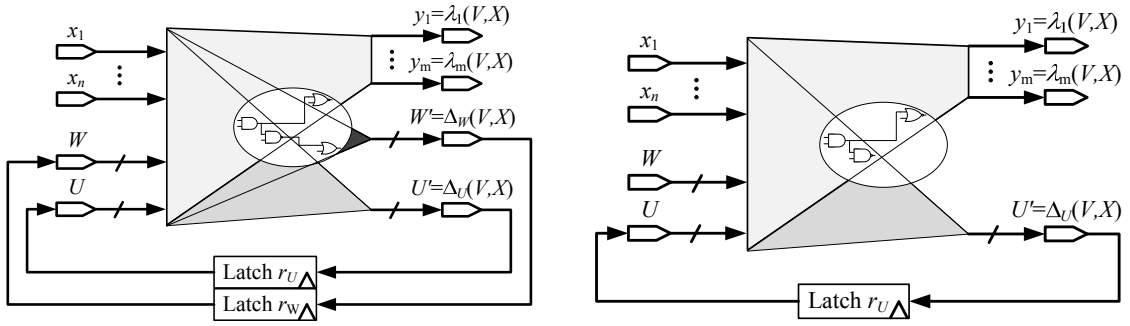
Even though in practice the above described decomposition is shown to improve the efficiency of the image computation it may still suffer from the explosion problem for the size of BDDs. Therefore, researchers proposed approximate computations.

Cho *et al.* presented an approximative FSM traversal method based on state space decomposition [CHM+96a]. In this approach, the approximate set of reachable states is calculated as the conjunction of subsets that can be represented by BDDs. The basic idea is to partition the original FSM into several sub-FSMs and to perform a symbolic traversal for each individual sub-FSM.

Definition 3.13. ***(Sub-FSMs)*** *Given an FSM $M = (I, S, S_0, \Delta, \Lambda)$ and a sub-vector of state variables $U = \langle v_j \ldots, v_k \rangle \subseteq V$, the sub-FSM corresponding to U is $\tilde{M} = (\tilde{I}, \tilde{S}, \tilde{S}_0, \tilde{\Delta}, \tilde{\Lambda})$, where*

- *$\tilde{I} = I \times B^{p-k+j-1}$ is the set of input symbol and is encoded by input variables and state variables which do not belong to U, i.e., $\tilde{X} = X \cup (V \setminus U)$.*
- *$\tilde{S} \subseteq B^{k-j+1}$ is the set of sub-states.*
- *$\tilde{S}_0 \subseteq \tilde{S}$ is the set of initial sub-states.*
- *$\tilde{\Delta} = \langle \delta_j, \ldots, \delta_k \rangle$ is the sub-vector of the next state functions of the original FSM.*
- *$\tilde{\Lambda} = \Lambda$ is the output function.* □

The sequential circuit of the sub-FSM is obtained from the sequential circuit of the original FSM by removing the latches corresponding to state variables $W = V \setminus U$. Also the portions of the combinational circuit on which only the next state functions of W depend are removed. Figure 3.4 illustrates the construction of the sequential circuit of the sub-FSM for the sub-vector U.



(a) original FSM (b) sub-FSM

Figure 3.4: Construction of the sequential digital circuit for the sub-FSM

Definition 3.14. ***(Dependency of Sub-FSMs)*** *Consider two sub-FSMs $\tilde{M}_k$ and $\tilde{M}_l$ corresponding to two vectors of state variables V_k and V_l. $\tilde{M}_k$ depends on $\tilde{M}_l$ if there is at least one state variable $v \in V_l$ and at least one state variable $v_j \in V_k$ such that $\delta_j \in \Delta_k$ depends on v. In other words, $\exists v_j \in V_k$ such that $V_l \cap supp(\delta_j) \neq \emptyset$.* □

Given a partition of the vector of state variables $(V_1, \ldots, V_h)$, we can traverse the corresponding sub-FSMs $(\tilde{M}_1, \ldots, \tilde{M}_h)$ to compute the over-approximate sets of reachable states $\tilde{S}_1, \ldots, \tilde{S}_h$. Each sub-FSM $\tilde{M}_l$, where $1 \leq l \leq h$, is traversed separately to compute the set of reachable sub-states. In the traversal algorithm, the pseudo inputs $V \setminus V_l$ can be treated as free primary inputs. However, these pseudo inputs can also be constrained to model the interaction among sub-FSMs. This will lead to a more precise approximation if the correct constraints are imposed properly to the pseudo inputs. The authors of [CHM+96a] proposed a number of algorithms to traverse the sub-FSMs and to model the interaction among them. There are two main algorithms i) *Machine By Machine (MBM)* and ii) *Frame By Frame (FBF)*. The latter has two variants called *Reached FBF (RFBF)* and *To FBF (TFBF)*.

Machine By Machine Approximate FSM Traversal

The MBM algorithm as shown in Figure 3.5 processes the sub-FSMs serially and iteratively. It starts with a coarse approximation of the $\tilde{S}_l$, which consist of all states of the sub-FSM $\tilde{M}_l$, for all $1 \leq l \leq h$. It means that all states are assumed to be reachable. Then, it tries to refine the approximation by traversing the sub-FSMs iteratively. Before a sub-FSM $\tilde{M}_l$ is traversed the pseudo inputs are constrained by the sets $\tilde{S}_k$ for all $1 \leq k \leq h$.

This is done by evaluating the generalized cofactor of the transition relation $\tilde{T}_l$ with respect to the sets $\tilde{S}_k$. Then, the *BFS_Traversal* algorithm is used to calculate the set of reachable states $\tilde{S}_l$ for $\tilde{M}_l$. The *BFS_Traversal* algorithm is similar to the algorithm in Figure 3.3 except that it returns the set of reachable states and it does not check reachable states against *bad* states. An event $traverse_l$ is used to drive the traversal algorithm. The sub-FSM $\tilde{M}_l$ is traversed again when $traverse_l = \mathbf{true}$. This happens only if the sets $\tilde{S}_k$ of the sub-FSMs $\tilde{M}_k$ on which $\tilde{M}_l$ depends are changed. The algorithm finishes when the $\tilde{S}_l$ sets do not shrink any more, i.e., $traverse_l = \mathbf{false}$ for all $1 \leq l \leq h$.

```
1: MBM_Traversal(⟨V_1, ..., V_h⟩, S_0) {
2:     for each l ∈ 1...h {
3:         S̃_l ← 1;
4:         S̃_{l_0} ← ∃_{V\Ṽ_l} S_0;
5:         traverse_l ← true;
6:     }
7:     converged ← false;
8:     while ( not converged ) {
9:         for each l ∈ 1...h {
10:            if traverse_l {
11:                old_S̃_l ← S̃_l;
12:                T̃_l ← T̃_l ↓ ⋀_{k=1}^{k=h} S̃_k;
13:                S̃_l ← BFS_Traversal(M̃_l);
14:                traverse_l ← false;
15:                if old_S̃_l ≠ S̃_l {
16:                    for each k ∈ 1...h {
17:                        if M̃_k depends on M̃_l
18:                            traverse_k ← true;
19:                    }
20:                }
21:        }
22:        converged ← not ⋁_{k=1}^{k=h}(traverse_k);
23:    }
23:    return {S̃_l | 1 ≤ l ≤ h}
24: }
```

Figure 3.5: Machine by machine approximate FSM traversal

Frame By Frame Approximate FSM Traversal

In contrast to the MBM algorithm, which traverses each sub-FSM completely before processing the next sub-FSM, the FBF algorithm handles all sub-FSMs in parallel. The generic FBF algorithm as shown in Figure 3.6 iteratively calculates the immediate successor states of the set $constraint_l$ for sub-FSMs. Before image computation is performed for the sub-FSM $\tilde{M}_l$, the transition relation $\tilde{T}_l$ is constrained by the sets $constraint_k$ for all $1 \leq k \leq h$ by the generalized cofactor. After image computation is done for all sub-FSMs, the results to_l of image computation are used to update the sets of reachable states $\tilde{S}_l$, the sets $constraint_l$ and to check the convergence of the algorithm. Two algorithms RFBF and TFBF are derived from the generic FBF algorithm by specifying how to update the sets $constraint_l$ and how to perform the convergence check.

```
1: FBF_Traversal(⟨V_1, ..., V_h⟩, S_0) {
2:     for each l ∈ 1...h {
3:         S̃_{l_0} ← ∃_{V\Ṽ_l} S_0;
4:         S̃_l ← S̃_{l_0};
4:         constraint_l ← S̃_{l_0};
6:     }
7:     converged ← false;
7:     j ← 0;
8:     while ( not converged ) {
9:         for each l ∈ 1...h {
12:            T̃_l ← T̃_l ↓ ⋀_{k=1}^{k=h} constraint_k;
13:            to_l^{j+1} ← Img(T̃_l, constraint_l);
14:            S̃_l ← S̃_l ∪ to_l^{j+1};
21:        }
21:        Update_Constraints;
21:        j ← j + 1;
21:        converged ← Convergence_Check;
23:    }
23:    return {S̃_l | 1 ≤ l ≤ h}
24: }
```

Figure 3.6: Frame by frame approximate FSM traversal

The algorithm RFBF uses the sets of reachable states $\tilde{S}_l$ as the sets $constraint_l$. It converges when the sets of reachable states do not change.

The algorithm TFBF uses the sets to_l as the sets $constraint_l$. It converges when the conjunction of the sets to_l is repeated. Formally, the algorithm converges at the jth iteration if there exists a previous iteration k such that $\bigwedge_{l=1}^{l=h} to_l^j = \bigwedge_{l=1}^{l=h} to_l^k$, where to_l^j, to_l^k denote the sets to_l at iteration j and k, respectively.

The FBF algorithms often result in a more precise approximation than the MBM algorithm. However, they need more image computations to converge. In [MKSS99], extensions were made to improve the convergence of the RFBF algorithm by proposing the *Least fixed-point Machine By Machine* (LMBM) algorithm.

Partitioning the Design into Sub-FSMs

Before the approximative FSM algorithms can be used, sub-FSMs of the design need to be identified. A *good* partition will improve the accuracy of the algorithms. In [CHM+96b], Cho *et al.* proposed methods to automatically determine *good* partitions of the design. Another method to partition the design using *overlapping projections* was presented in [GDHH98].

3.3.3 Abstraction Refinement

While approximative reachability analysis traverses all sub-FSMs of the system independently of properties to be proven, abstraction refinement checks a specific property against only one *abstract model* which is, in special cases, one sub-FSM of the system. This abstraction is also an approximate model of the system and is relevant to proving the property. In addition, the abstraction will be refined if the approximation does not contain sufficient information to prove the property.

Abstraction refinement concepts were comprehensively described in [ME99, Kur94]. Here, the conceptual definitions and the framework for abstraction refinement algorithms will be shortly described.

An abstraction function h for a Kripke model $M = (S, S_0, T, L)$ is a surjection $h : S \mapsto \hat{S}$, which maps a concrete state $s_j \in S$ to an abstract state $\hat{s}_k = h(s_j) \in \hat{S}$. Given an abstract state $\hat{s}_k \in \hat{S}$, the set of concrete states corresponding to $\hat{s}_k$ is denoted by $h^{-1}(\hat{s}_k) = \{s \in S | h(s) = \hat{s}_k\}$.

Definition 3.15. ***(Minimal Abstraction)*** *[CCK+02] The minimal abstract model $\hat{M} = (\hat{S}, \hat{S}_0, \hat{R}, \hat{L})$ corresponding to a Kripke model $M = (S, S_0, T, L)$ and an abstraction function h which is defined by:*

1. $\hat{S} = \{\hat{s} | \exists s \in S : h(s) = \hat{s}\}$
2. $\hat{S}_0 = \{\hat{s} | \exists s \in S_0 : h(s) = \hat{s}\}$
3. $\hat{T} = \{(\hat{s}_1, \hat{s}_2) | \exists (s_1, s_2) \in T : h(s_1) = \hat{s}_1 \wedge h(s_2) = \hat{s}_2\}$
4. $\hat{L}(\hat{s}) = \bigcup_{h(s)=\hat{s}} L(s)$ □

Given an abstraction function h, an atomic formula ϕ is called to *respect* h if $\forall s \in S : h(s) \models \phi$. The following theorem is fundamental to all abstraction refinement frameworks.

Theorem 3.1. *(**Abstraction Preservation**) [ME99] Let $\hat{M}$ be an abstract model of M corresponding to the abstraction function h and ϕ be a propositional formula that respects h. Then, $\hat{M} \models \mathsf{AG}\,\phi \Rightarrow M \models \mathsf{AG}\,\phi$* □

According to Theorem 3.1, if a property holds in the abstraction of a concrete model, then the property also holds in the concrete model. However, the converse of the theorem is not true. If the property fails in the abstract model, it may still hold in the concrete model. In this case, the counterexample for the property in the abstract model is a *spurious* counterexample, i.e., a *false negative* because it does not correspond to a path in the concrete model. The abstraction function is not sufficient to prove the property and needs to be refined.

Definition 3.16. *(**Refinement**) [ME99] An abstraction function h' is a refinement for the abstraction function h and M if:*

1. $\forall s_1, s_2 \in S : h'(s_1) = h'(s_2) \Rightarrow h(s_1) = h(s_2)$
2. $\exists s_1, s_2 \in S : h(s_1) = h(s_2) \wedge h'(s_1) \neq h'(s_2)$ □

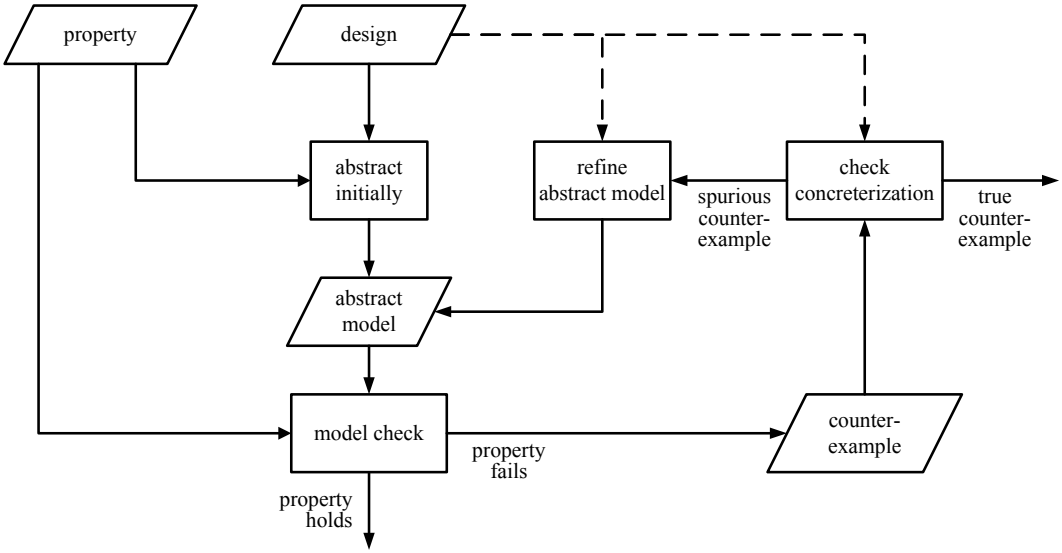


Figure 3.7: Abstraction refinement verification methodology flow

The basic verification methodology flow of abstraction refinement methods is presented in Figure 3.7. In the flow, first the initial abstract model is generated. Then the property is checked against the abstract model. If the property fails in the abstract model, the abstract counterexample is validated against the concrete model to check if it is a real counterexample or a false negative. If the counterexample is spurious, the abstract model is refined. The two steps of model checking and refinement are repeated until the property is proven to hold or a real counterexample is found. In the following, techniques for abstraction and refinement will be described.

Abstraction

Generally, creating a minimal abstraction for a concrete model M and an abstraction function h is often too expensive or even impossible [ME99]. There are two main techniques to derive an approximative abstract model, i.e.:

1. localization reduction,

2. predicate abstraction.

If the model is given as an encoded FSM, a minimal abstract model can be approximated by the localization reduction method [ME99, Kur94]. In localization reduction, the vector of state variables is partitioned into sets of *visible($\mathcal{V}$)* and *invisible ($\mathcal{I}$)* state variables. The abstract model $\hat{M}$ is the sub-FSM corresponding to the vector of visible state variables $\mathcal{V}$. Given the vector of visible state variables $\mathcal{V} = \langle v_1, \ldots, v_k \rangle \subseteq V$, the abstraction function being defined in localization reduction is $h(s) = s(\mathcal{V})$.

In predicate abstraction [GS97, TR01, ECW03, JKSC08], the abstraction function is defined by a set of predicates. These predicates are the Boolean propositions that label the concrete and abstract states. Given a set of predicates $\mathcal{AP} = \{P_1, \ldots, P_q\}$, the abstract state space $\hat{S}$ is encoded by a vector of abstract state variables $\hat{V} = \{\hat{v}_1, \ldots, \hat{v}_q\}$, where each abstract state variable $\hat{v}_j$ corresponds to a predicate P_j. Given a concrete state s, the valuation of the set of predicates is a Boolean vector $A = \langle P_1(s), \ldots, P_q(s) \rangle$, where $P_j(s) = 1$ iff $s \models P_j$. This vector A encodes the abstract state $\hat{s}$ to which the concrete state s is mapped by abstraction function h. More formally, the abstraction function corresponding to a given set of predicates $\mathcal{AP}$ is:

$$h(s) = \{\hat{s} \in B^q | \bigwedge_{j=1}^{j=q} \hat{s}(\hat{v}_j) \equiv P_j(s)\} \tag{3.9}$$

From Equation 3.9, the abstract transition relation of the abstract model $\hat{M}$ is defined as follows:

$$\hat{T} = \{(\hat{s}_1, \hat{s}_2) | \exists s_1, s_2 \in S : \bigwedge_{j=1}^{j=q} \hat{s}_1(\hat{v}_j) \equiv P_j(s_1) \wedge T(s_1, s_2) \wedge \bigwedge_{j=1}^{j=q} \hat{s}_2(\hat{v}'_j) \equiv P_j(s_2)\}$$

In [ECW03, JKSC08], the abstract transition relation is approximated by enumerating all possible pairs $(\hat{s}_1, \hat{s}_2)$ by identifying all satisfying assignments of the SAT-instance in the following Equation 3.10, and then projecting the satisfying assignments to the abstract current and next state variables $\hat{V}, \hat{V}'$.

$$\bigwedge_{j=1}^{j=q} \hat{s}_1(\hat{v}_j) \equiv P_j(s_1) \wedge T(s_1, s_2) \wedge \bigwedge_{j=1}^{j=q} \hat{s}_2(\hat{v}'_j) \equiv P_j(s_2) \tag{3.10}$$

Here, $T(s_1, s_2)$ is the SAT-instance representing the concrete transition relation. In order to reduce the complexity of the expensive ALLSAT procedure, the set of predicates is partitioned into *clusters*, i.e., small sets of predicates [JKSC08]. The abstract models are, then, calculated for all clusters, which are then conjoined to construct monolithic abstract transition function. This will lead to a coarser abstraction.

Refinement

In the refinement step, the *counterexample-guided refinement* approach [CGJ+00] is used. The abstract counterexample is checked if it has a corresponding concrete counterexample. If it does not have one, it is analyzed to refine the abstraction. Here, it will be briefly described how to analyze an abstract counterexample which is a path of abstract states $\hat{\pi} = (\hat{s}_1, \ldots, \hat{s}_k)$. The reader is referred to [CGJ+00] for a more detailed framework to analyze abstract counterexamples which include loops.

For an abstract path $\hat{\pi}$, the set of corresponding concrete paths denoted by $h^{-1}(\hat{\pi})$ is defined by the following equation:

$$h^{-1}(\hat{\pi}) = \{(s_1, \ldots, s_k) | \bigwedge_{j=1}^{j=k} h(s_j) = \hat{s}_j \wedge S_0(s_1) \wedge \bigwedge_{j=1}^{j=k-1} T(s_j, s_{j+1})\} \tag{3.11}$$

The set of concrete paths can be considered as a sequence of sets S_j consisting of concrete states corresponding to the abstract state $\hat{s}_j$. S_j is calculated by the image computation as follows:

$$S_j = Img(T, S_{j-1}) \cap h^{-1}(\hat{s}_j) \tag{3.12}$$

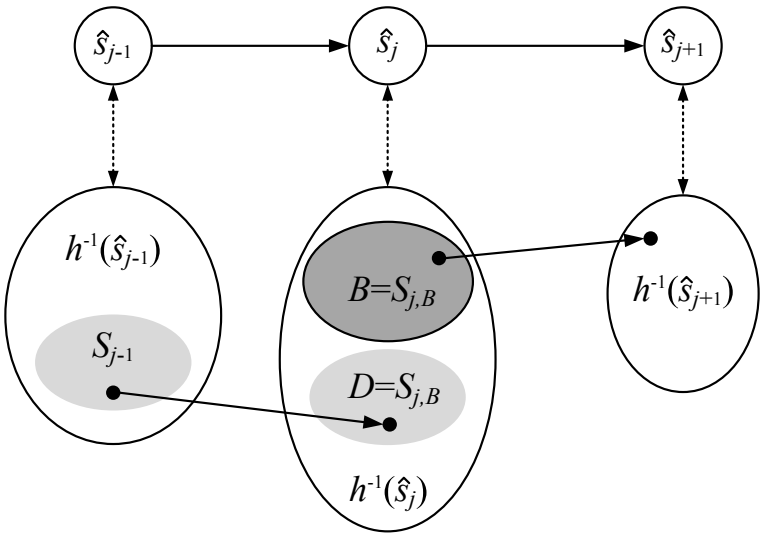


Figure 3.8: Sets of dead-end and bad states corresponding to spurious abstract transition $(\hat{s}_j, \hat{s}_{j+1})$

If $h^{-1}(\hat{\pi})$ is empty, the abstract counterexample is spurious and there exists a minimal j ($1 \leq j \leq k$) such that $S_{j+1} = \emptyset$. The set of concrete states $h^{-1}(\hat{s}_j)$ consists of two different sets of i) *dead-end* states and ii) *bad* states. Figure 3.8 illustrates the concept of dead-end states and bad-states.

- The states in the set S_j are *dead-end* states because there isn't any concrete transition from a state in S_j to a state in $h^{-1}(\hat{s}_{j+1})$.
- Since the transition $(\hat{s}_j, \hat{s}_{j+1})$ exists, there must be a concrete transition from a state in $h^{-1}(\hat{s}_j)$ to a state in $h^{-1}(\hat{s}_{j+1})$ even though there is no transition from S_j to $h^{-1}(\hat{s}_{j+1})$. The states in $h^{-1}(\hat{s}_j)$ whose next states are in $h^{-1}(\hat{s}_{j+1})$ are called *bad* states. The set of bad-states $S_{i,B}$ can be calculated by the preimage computation as:

$$S_{i,B} = PreImg(T, h^{-1}(\hat{s}_{j+1})) \cap h^{-1}(\hat{s}_j)$$

The set $S_{i,B}$ is considered as the reason of the spurious counterexample. Therefore, refining the abstract model to eliminate the counterexample becomes separating the bad-states from the dead-end states. This is done by making some invisible variables visible in case of localization reduction or by adding some predicates in case of predicate abstraction.

Because the minimum separation problem is known to be NP-hard [CGJ+01], many heuristic approaches based on SAT or BDD have been proposed to find a small set of invisible state variables [CCK+02, CGKS02, WLJ+06]. For predicate abstraction, the authors of [JKSC08] proposed to calculate *weakest preconditions* and then to use them as predicates to be added.

3.4 Bounded Model Checking

Bounded Model Checking (BMC) based on SAT-solvers was proposed by Biere [BCCZ99] and has been used as a complementary technique to SMC. While SMC handles the complete STG of the system, BMC only works on a bounded iterative model of the system. Consequently, BMC aims at searching for bugs in the runs of the system, where the length of these runs is bounded by some integer k. In other words, BMC handles only a portion of the STG consisting of the states that are at distance k from the initial states. This portion of the STG is modeled by the *time frame expansion* of the considered FSM.

Definition 3.17. *(**Time Frame Expansion Model**) Given an FSM M and $k \geq 0$, the k time frame expansion of M is a Boolean function:*

$$[\![M]\!]_k = \bigwedge_{j=0}^{j=k-1} T(s_j, i_j, s_{j+1}) \tag{3.13}$$

where $s_j, s_{j+1} \in S$ are states and $i_j \in I$ is input letter of the FSM M. □

Consider an encoded FSM being implemented as a sequential circuit, the time frame expansion of the encoded FSM is represented as an *iterative circuit model*, which is constructed from the sequential circuit by making k copies of the combinational circuit representing the transition function and the output function. Each copy represents the behavior of the original circuit at a specific time. The output values of the original circuit at time frame j are produced in the iterative circuit at the outputs Y^j and are computed based on state variables V^j and inputs X^j. The combinational circuit representing transition functions are concatenated such that the next-state variables V'^j of the j-th copy are the present-state variables V^{j+1} of the $(j+1)$-th copy. Figure 3.9 illustrates the iterative circuit model expanded for the k time frames. For the iterative circuit model shown in Figure 3.9, Equation 3.13 becomes:

$$[\![M]\!]_k = \bigwedge_{j=0}^{j=k-1} T(V^j, X^j, V^{j+1}) \tag{3.14}$$

Here, V^j, X^j, V^{j+1} denote the copies of the state variables, the inputs at time frame j, and the state variables at time frame $j+1$.

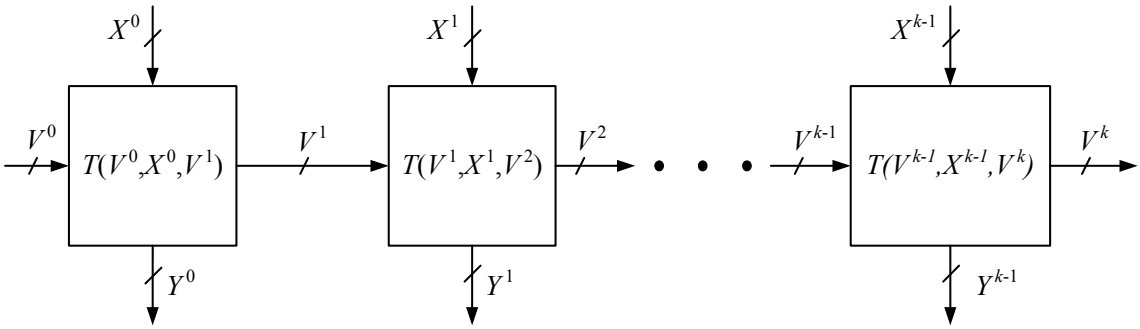


Figure 3.9: Iterative circuit of an encoded FSM

$[\![M]\!]_k$ is a Boolean function of variables X^j, V^j for all $0 \leq j < k$ and can be considered as a characteristic function of the set of paths $\pi = (s_0, s_1, \ldots, s_k)$, whose length is $k+1$. In order to represent the set of initialized paths the constraint of the initial states S_0 is added to the time frame expansion model, which results in:

$$[\![M]\!]_k^0 = S_0(V^0) \wedge \bigwedge_{j=0}^{j=k-1} T(V^j, X^j, V^{j+1}) \tag{3.15}$$

In order to prove a property ϕ being specified in LTL, the inverted property $\psi = \overline{\phi}$ is translated into a Boolean proposition $[\![\psi]\!]_k$ that constrains the set of paths π to fulfill ψ, i.e., to violate ϕ. The translation is described in detail in [BCCZ99] and can be summarized by the following definitions.

Definition 3.18. *(**(k,l)-loop Path**) For $k \geq l \geq 0$, a path $\pi = (s_0, \ldots, s_l, \ldots, s_k)$ is a (k,l)-loop path if there is a state transition from s_k to s_l. In other words, π is a (k,l)-loop if $T(V^k, X^k, V^l) = 1$. Therefore, the set of (k,l)-loop paths is constrained by:*

$$_lL_k = T(V^k, X^k, V^l)$$

□

Definition 3.19. *(**Paths with Loop**) For $k \geq 0$, a path π consists of at least a loop if there exists $0 \leq l \leq k$ such that π is (k,l)-loop. π is called a k-loop path. The set of k-loop paths is constrained by:*

$$L_k = \bigvee_{l=0}^{l=k} {}_lL_k$$

□

Definition 3.20. *(**Successor in a Loop**) For $k \geq l, i \geq 0$, the successor $succ(i)$ of i in a (k,l)-loop path π is:*

$$succ(i) = \begin{cases} i+1 & \text{if } i < k \\ l & \text{if } i = k \end{cases}$$

□

Definition 3.21. *(**Translation of an LTL Formula for a Loop**) Given an LTL formula ψ, the set of paths with loop satisfying ψ is constrained by a Boolean proposition being defined recursively as follows:*

- *if ψ is an atomic proposition, ${}_l[\![\psi]\!]_k^i = \psi(V^i)$*
- *if ψ is an atomic proposition, ${}_l[\![\overline{\psi}]\!]_k^i = \overline{\psi(V^i)}$*
- ${}_l[\![\psi \wedge \phi]\!]_k^i =_l [\![\psi]\!]_k^i \wedge_l [\![\phi]\!]_k^i$
- ${}_l[\![\psi \vee \phi]\!]_k^i =_l [\![\psi]\!]_k^i \vee_l [\![\phi]\!]_k^i$
- ${}_l[\![\mathsf{X}\,\psi]\!]_k^i =_l [\![\psi]\!]_k^{succ(i)}$
- ${}_l[\![\mathsf{G}\,\psi]\!]_k^i =_l [\![\psi]\!]_k^i \wedge_l [\![\mathsf{G}\,\psi]\!]_k^{succ(i)}$
- ${}_l[\![\mathsf{F}\,\psi]\!]_k^i =_l [\![\psi]\!]_k^i \vee_l [\![\mathsf{F}\,\psi]\!]_k^{succ(i)}$

□

Definition 3.22. *(**Translation of an LTL Formula Without a Loop**) Given an LTL formula ψ, the set of paths without a loop satisfying ψ is constrained by a Boolean proposition being defined recursively as follows:*

- *Inductive Case: $\forall i \leq k$*
  - *if ψ is an atomic proposition, $[\![\psi]\!]_k^i = \psi(V^i)$*

  - *if ψ is an atomic proposition,* $[\![\overline{\psi}]\!]_k^i = \overline{\psi(V^i)}$
  - $[\![\psi \wedge \phi]\!]_k^i = [\![\psi]\!]_k^i \wedge [\![\phi]\!]_k^i$
  - $[\![\psi \vee \phi]\!]_k^i = [\![\psi]\!]_k^i \vee [\![\phi]\!]_k^i$
  - $[\![\mathsf{X}\,\psi]\!]_k^i = [\![\psi]\!]_k^{i+1}$
  - $[\![\mathsf{G}\,\psi]\!]_k^i = [\![\psi]\!]_k^i \wedge [\![\mathsf{G}\,\psi]\!]_k^{i+1}$
  - $[\![\mathsf{F}\,\psi]\!]_k^i = [\![\psi]\!]_k^i \vee [\![\mathsf{F}\,\psi]\!]_k^{i+1}$
- *Base Case:* $[\![\psi]\!]_k^{k+1} = 0$ □

Definition 3.23. *(***Translation of an LTL Formula***) Given an LTL formula ψ and a Kripke structure M, the set of paths satisfying ψ is constrained by:*

$$[\![\psi]\!]_k = \Big(\big(\overline{L}_k \wedge [\![\psi]\!]_k^0\big) \vee \bigvee_{l=0}^{l=k} \big({}_lL_k \wedge {}_l[\![\psi]\!]_k^0\big) \Big) \tag{3.16}$$

□

From Equation 3.15 and Equation 3.16, the characteristic function for the set of initialized paths that fulfill the property ψ is:

$$[\![M, \phi]\!]_k = S_0(V^0) \wedge \bigwedge_{j=0}^{j=k-1} T(V^j, X^j, V^{j+1}) \wedge \Big(\big(\overline{L}_k \wedge [\![\psi]\!]_k^0\big) \vee \bigvee_{l=0}^{l=k} \big({}_lL_k \wedge {}_l[\![\psi]\!]_k^0\big) \Big) \tag{3.17}$$

To check the property ϕ, the characteristic function $[\![M, \psi]\!]_k$ is converted into a SAT-instance. If this SAT-instance is satisfiable the satisfying assignment corresponds to the path that fulfills the property ψ. This path is the counterexample violating the property ϕ within the bound k. If the SAT-instance is unsatisfiable, we can conclude that there are no violating k-state paths. Yet, we can not conclude that the property is proven. Normally, the satisfication of $[\![M, \psi]\!]_k$ is repeatedly checked for increasing values of k until either a counterexample is discovered or k equals to the *reachability diameter* of the Kripke structure or the computing resources (memory, runtime) is exceeded.

In practice, calculating the reachability diameter is a very expensive task. Hence, the reachability diameter is often unknown. Moreover, the reachability diameter is often large such that BMC often terminates because of running out of resources. Therefore, BMC is practically considered as an incomplete method: it is able to disprove a property but is not able to prove the property.

Induction based BMC [SSS00] can be used to prove a safety property $\mathsf{G}\,\phi$. To prove the property by inductive proof with depth n, the SAT-solver is used to check the satisfication of two Equations $[\![M, \phi]\!]_{base}$ and $[\![M, \phi]\!]_{induc}$.

$$[\![M, \phi]\!]_{base} = S_0(V) \wedge \bigwedge_{j=0}^{j=n-1} T(V^j, X^j, V^{j+1}) \wedge \bigwedge_{0 \le i < j \le n} (V^i \neq V^j) \wedge \bigvee_{j=0}^{j=n} \overline{\phi(V^j)} \tag{3.18}$$

$$[\![M, \phi]\!]_{induc} = \bigwedge_{j=0}^{j=n} \phi(V^j) \wedge \bigwedge_{j=0}^{j=n} T(V^j, X^j, V^{j+1}) \wedge \bigwedge_{0 \leq i < j \leq n} (V^i \neq V^j) \wedge \overline{\phi(V^{n+1})} \quad (3.19)$$

In order to improve the performance of BMC, one can use an approximate set of reachable states to prune the search space of the SAT-instance as proposed in [GGW+03, CNQ04]. The approximate set of reachable states are computed by the reachability analysis algorithms in Section 3.3.2. This set is then translated into CNF which is added to the SAT-instance. This can result in faster run time of the SAT-solver as in [CNQ04] or in smaller depth n of the inductive proof as in [GGW+03].

Chapter 4

Interval Property Checking

In this chapter, the interval property checking methodology for protocol compliance verification will be presented. In the first section, the common template that is often used to formulate properties in protocol compliance verification is presented. In the second section, interval property checking is described. The section will discuss how false negatives which occur when properties are checked using the IPC-based methodology can be eliminated by *reachability invariants*. In the present-day IPC-based methodology flow these invariants can only be identified manually, which requires a lot of efforts.

4.1 Properties in Interval Property Checking

4.1.1 Operational Interval Properties

In Chapter 3, it is assumed that the desired behavior of a system can be described by formulas in CTL or LTL. However, such a property often leads to a complex problem for property checkers. Hence, in practice, properties are often formulated in a restricted format, for example, as a safety property $\mathsf{G}\,\phi$. In this section, a special property format which is very common in SoC verification will be considered.

In practical SoC verification we want to verify a piece of the design behavior where certain conditions of the environment occur and the design is at certain internal states. These input and state conditions are specified in safety properties together with the desired behavior in the implication format as follows:

$$\phi = \mathsf{G}\,(a \rightarrow c) \tag{4.1}$$

These properties specify that the system exhibits the desired behavior described in c whenever the external input and internal states of the system fulfill the constraints described in a. In the following, a and c are called the *assumption* and the *commitment* of a property, respectively.

This work focuses on specifying and verifying the correctness of SoC modules and their interfaces at the *register transfer level (RTL)*. At the RTL, designers very commonly

adopt an *operational* view of the system where the desired behavior is described clock tick by clock tick within a certain time interval. Consequently, the properties needed to verify this behavior can be specified by an assumption a and a commitment c describing a finite behavior of the system. This finite behavior can be specified in a restricted version of LTL. In this restricted version, only a finite number of instances of the temporal operator X is used. It results in a special style of writing properties that is very common in SoC verification. In this style, the finite behavior is specified as conjunctions of propositions for the inputs and the outputs of the design at time j, as follows:

$$a = a_s(V) \wedge \bigwedge_{j=0}^{j=n-1} \mathsf{X}^j(a_j(X)) \tag{4.2}$$

$$c = \bigwedge_{j=0}^{j=n-1} \mathsf{X}^j(c_j(X,Y)) \wedge \mathsf{X}^n(c_e(V)) \tag{4.3}$$

In these formulas,

- n denotes the number of clock cycles that the design needs to perform a specific operation,
- $a_j(X)$ is a Boolean proposition and specifies certain *input* conditions at time point j,
- $c_j(X,Y)$ is a Boolean proposition and specifies certain *output* conditions at time point j,
- $a_s(V), c_e(V)$ are Boolean propositions and specify a *starting assumption* and an *ending commitment* in terms of the *state variables* of the design,
- $\mathsf{X}^n\phi$ is defined by $\mathsf{X}^n\phi = \mathsf{X}\mathsf{X}^{n-1}\phi$ for $n > 0$ and $\mathsf{X}^0\phi = \phi$.

Let us consider the process of setting up properties to describe the finite behaviors in more detail. Usually, when inspecting the design to set up properties, as a first step, the designer identifies a set of *important control states* which serve as an orientation in the overall verification methodology. For each operation two important control states are chosen as its *starting* and its *ending state*. In the property this is reflected by the *starting assumption* $a_s(V)$ and the *ending commitment* $c_e(V)$. The time window between the starting and the ending state is called *inspection interval* which has finite length n. For example, when verifying a processor the decode and the write-back phase of an instruction could be chosen to define the inspection interval of an operation. During the inspection interval the correct input/output behavior is described by specifying the input conditions $a_j(X)$ and output conditions $c_j(X,Y)$. The input and output conditions can be given by an arbitrary relation specifying the functional operation of the system. These conditions can be derived from the system specification. Besides the starting and ending states, in practice, a property may also relate to some intermediate states. This may sometimes

be required to further constrain the behavior. On the other hand, since such additional state information can be tedious to derive the verification engineer will prefer to limit the state specifications of an operational property to the important control states whenever possible.

property operation_property **is**
  assume:
    at t: $a_s(V)$;
    at t: $a_0(X)$;
    at t+1: $a_1(X)$;
    (...);
    at t+n-1: $a_{n-1}(X)$;
  prove:
    at t: $c_0(X,Y)$;
    at t+1: $c_1(X,Y)$;
    (...);
    at t+n-1: $c_{n-1}(X,Y)$;
    at t+n: $c_e(V)$;
end property;

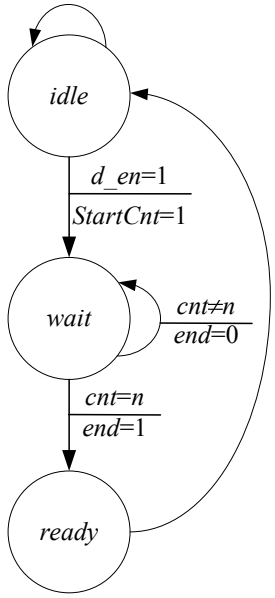
Figure 4.1: ITL operational property (= 'interval property')

Since this style of writing properties leads to an intuitive methodology and is quite popular among industrial designers commercial property checkers support this by proprietary languages. In the proprietary *interval language (ITL)* [BJW04, WSFT04] the above property looks as shown in Figure 4.1. ITL, in practice, allows to express more complex Boolean conditions a_j, c_j . For example, it allows us to include internal gates of the circuit into the property specification. It also allows to use the temporary variables which store the values of gates at different time points in the Boolean propositions a_j, c_j. However, by tracing the predecessors of these gates it is always possible to generate an equivalent property in the above scheme. In addition, as long as the maximum duration of an operation is bounded it is also possible to express behaviors of variable length by disjunction of commitments for different time points. ITL supports such specifications by providing additional syntactic features such as a temporal operator *within*$[t_a, t_b]$. In this thesis, for sake of simplicity, only the property template mentioned above is considered. However, it does not limit the proposed method in being applied to other properties in ITL.

The important control states provide an intuitive way of setting up a property suite by linking the individual properties by their starting and ending states. In most cases, the important control states of a design are specified by only constraining a few state bits of the global state vector. For example, when verifying the instructions of a processor only the state variables related to important control information such as opcode or certain status bits need to be specified. Nothing is said about the other state bits such as the pipeline buffers. The important control states of a design are typically related to a small set of

```
case CurrentState is
  when idle ⇒
    if (d_en = 1) then
      NextState ⇐ wait; StartCnt ⇐ 1;
    else NextState ⇐ idle;
    end if;
  when wait ⇒
    if (cnt = n) then
      NextState ⇐ ready; end ⇐ 1;
    else NextState ⇐ wait;
    end if;
  when ready ⇒
    NextState ⇐ idle;
    (...);
end case;
```

(a) main-code



(b) main-STG

Figure 4.2: Example of the RTL code and the STG of a main-FSM

state variables. As the verification engineer moves along the important control states of a design he traverses an *abstract* machine whose states are given by the important control states and whose transitions correspond to the operations that are verified. This abstract machine in the following will be called *main-FSM* and plays a key role in the proposed approximative reachability analysis.

Fortunately, complex control structures often expose hierarchical structures and a main-FSM can be identified easily. For reasons to be explained in Section 4.2 such a main-FSM is mostly used in the context of protocol compliance verification. The following section will present how to exploit this hierarchy and will describe a scheme for protocol compliance checking where a_s and c_e can be defined based on the states of a main-FSM in the design.

4.1.2 Interval Properties Based on the Main-FSM

In common design styles for implementing standard protocols the overall behavior of a design is controlled by a dedicated FSM. This FSM can be used as the main-FSM to identify the important control states and to formulate properties in the template presented in the previous section.

Although the main-FSM in protocol implementations is usually quite small it plays a very important role in the design. As an example, in an RTL description we may find a piece of code as illustrated in Figure 4.2(a) which implements the main-FSM. The main-FSM is encoded by a small sub-vector $\hat{V} = (v_p, \ldots, v_q)$, with $1 \leq p < q \leq m$, of the overall state variable vector V. It can be represented by a *state transition graph* (STG)

where the vertices represent the *main-states* $\hat{S}$ and the edges indicate *main-transitions* $\hat{T}$, as shown in Figure 4.2(b). The size of the main-FSM is usually small so that its STG can be represented explicitly.

When the design performs a specific functionality its main-FSM is supposed to go through a certain sequence of main-states $\pi_{\hat{s}} = (\hat{s}_1, \ldots, \hat{s}_{n+1})$. From this sequence the verification engineer can derive the following starting assumption and ending commitment for the corresponding property:

$$\begin{array}{ll} a_s & = \bigwedge_{i=p}^{q} (\hat{v}_i \equiv \hat{s}_1(\hat{v}_i)) \\ c_e & = \bigwedge_{i=p}^{q} (\hat{v}_i \equiv \hat{s}_{n+1}(\hat{v}_i)) \end{array} \tag{4.4}$$

In short notation, we also write:

$$\begin{array}{ll} a_s & = (\hat{V} \equiv \hat{s}_1) \\ c_e & = (\hat{V} \equiv \hat{s}_{n+1}). \end{array} \tag{4.5}$$

We obtain the template in Equation 4.6 for operational properties based on the main-FSM. In ITL, the property is shown in Figure 4.3.

$$\phi = \mathsf{G}\left[\Big((\hat{V} \equiv \hat{s}_1) \wedge \bigwedge_{j=0}^{j=n-1} \mathsf{X}^j(a_j(X))\Big) \rightarrow \Big(\bigwedge_{j=0}^{j=n-1} \mathsf{X}^j(c_j(X,Y)) \wedge \mathsf{X}^n(\hat{V} \equiv \hat{s}_{n+1})\Big)\right] \tag{4.6}$$

property operation_s_1_to_s_n **is**
  assume:
    at t: $\hat{V} \equiv \hat{s}_1$;
    at t: $a_0(X)$;
    at t+1: $a_1(X)$;
    (...);
    at t+n-1: $a_{n-1}(X)$;
  prove:
    at t: $c_0(X,Y)$;
    at t+1: $c_1(X,Y)$;
    (...);
    at t+n-1: $c_{n-1}(X,Y)$;
    at t+n: $\hat{V} \equiv \hat{s}_{n+1}$;
end property;

Figure 4.3: Property template based on main-states in ITL

For illustration of this template consider again the STG depicted in Figure 4.2(b). In order to check that the timing for a data transfer is met the verification engineer

may specify that output signal end will be asserted n clock cycles after the data transfer has been enabled ($d\_en = 1$) provided that the design has been in the $idle$ state. During execution of this transfer the main-FSM will traverse the following path in its STG $(idle, wait, \ldots, wait, ready)$. In the $idle$ state, the main-FSM enables an external counter to count the number of transferred blocks by setting the corresponding enable signal $StartCnt$ of the counter. The output cnt of the counter will be evaluated in the $wait$ state of the main-FSM to decide whether or not the transfer is complete. If the transfer is complete the main-FSM will make a transition into state $ready$. Without looking at the details of the counter and by only inspecting the code of the main-FSM the above property template can be used to specify this operation as follows:

$$f = \mathsf{G}\left[\Big((\hat{V} \equiv idle) \wedge (d\_en \equiv 1)\Big) \rightarrow \Big(\mathsf{X}^{n}(end \equiv 1) \wedge \mathsf{X}^{n+1}(\hat{V} \equiv ready)\Big)\right] \quad (4.7)$$

property exTemplate **is**
  assume:
    at t: $\hat{V} \equiv idle$;
    at t: $d\_en \equiv 1$;
  prove:
    at t+n: $end \equiv 1$;
    at t+n+1: $\hat{V} \equiv ready$;
end property;

Figure 4.4: Example of property in proposed template

The corresponding ITL template is given in Figure 4.4. Note that the starting and ending states are main-states and no other state information is specified in the property. Section 4.2 will elaborate how, in general, a_s and c_e need to be refined in order to check this kind of property without a complete reachability analysis of the design.

4.2 Interval Property Checking

This section will describe how to verify operational properties that are formulated in ITL. The approach is based on combining a bounded circuit model with certain invariants.

Similar to BMC, in order to verify the property in Equation 4.6, the inverted property is translated into a Boolean proposition that represents the set of k-state paths which violate the property. Let us consider the inverted property ψ of ϕ being written by using

the duality of LTL operators and De-Morgan's laws as follows:

$$\psi = \overline{\phi} = \mathsf{F}\left[\Big((\hat{V} \equiv \hat{s}_1) \wedge \bigwedge_{j=0}^{j=n-1} \mathsf{X}^j(a_j(X))\Big) \wedge \Big(\bigvee_{j=0}^{j=n-1} \mathsf{X}^j(\overline{c_j(X,Y)}) \vee \mathsf{X}^n(\hat{V} \not\equiv \hat{s}_{n+1})\Big)\right] \tag{4.8}$$

First, we consider the set of paths with length $k = n$ and without a loop fulfilling ψ. This set of paths is constrained by the proposition that is translated from the LTL formulas as follows:

Definition 4.1. *(***Translation of an LTL Formula for IPC***) Given an LTL formula ψ in the format in Equation 4.8, the set of paths satisfying ψ is constrained by a Boolean proposition being defined recursively as follows:*

- *if ψ is an atomic proposition,* $[\![\psi]\!]^i = \psi(V^i)$
- *if ψ is an atomic proposition,* $[\![\overline{\psi}]\!]^i = \overline{\psi(V^i)}$
- $[\![\psi \wedge \phi]\!]^i = [\![\psi]\!]^i \wedge [\![\phi]\!]^i$
- $[\![\psi \vee \phi]\!]^i = [\![\psi]\!]^i \vee [\![\phi]\!]^i$
- $[\![\mathsf{X}\,\psi]\!]^i = [\![\psi]\!]^{i+1}$
- $[\![\mathsf{F}\,\psi]\!]^i = [\![\psi]\!]^i$ □

When Definition 4.1 is applied to a safety property in ITL, it leads to a much simpler proposition than the translation of an LTL property in the traditional BMC. The translation for the property in Equation 4.8 is:

$$[\![\psi]\!] = (\hat{V} \equiv \hat{s}_1) \wedge \bigwedge_{j=0}^{j=n-1} (a_j(X^j)) \wedge \Big(\bigvee_{j=0}^{j=n-1} (\overline{c_j(X^j, Y^j)}) \vee (\hat{V}^n \not\equiv \hat{s}_{n+1})\Big) \tag{4.9}$$

In the conventional BMC method, the set of paths with length k without a loop fulfilling ψ at at least one state s_t (where $0 \leq t \leq k$) is represented by the following proposition:

$$[\![M, \psi]\!]^k_{\text{BMC}} = S(V^0) \wedge \bigwedge_{j=0}^{j=k+n-1} T(V^j, X^j, V^{j+1}) \wedge \bigvee_{t=0}^{t=k} [\![\psi]\!]_t \tag{4.10}$$

In Equation 4.10, $[\![\psi]\!]_t$ constrains the state s_t of the path π to satisfy ψ, that is defined by:

$$[\![\psi]\!]_t = (\hat{V}^t \equiv \hat{s}_1) \wedge \bigwedge_{j=0}^{j=n-1} (a_j(X^{t+j})) \wedge \Big(\bigvee_{j=0}^{j=n-1} (\overline{c_j(X^{t+j}, Y^{t+j})}) \vee (\hat{V}^{t+n} \not\equiv \hat{s}_{n+1}) \Big) \tag{4.11}$$

The proposition in Equation 4.10 is translated into a SAT instance that is solved by a SAT-solver to find a path that fulfills ψ, and, hence, violates ϕ. In the conventional method, k starts at 0 and increases until a counterexample is found or the diameter of the design is reached. Normally, huge values of k are required so that the resulting SAT instances, in most cases, would grow beyond the capacity of the solver. In order to overcome this problem we consider only a single instance of ψ_t and replace the constraints for the initial state by an invariant $\mathcal{I}$. This leads to a simplified proposition as follows:

$$[\![M, \psi]\!]^t_{\text{IPC}} = \mathcal{I}(V^t) \wedge \bigwedge_{j=0}^{j=n} T(V^{t+j}, X^{t+j}, V^{t+j+1}) \wedge [\![\psi]\!]_t \tag{4.12}$$

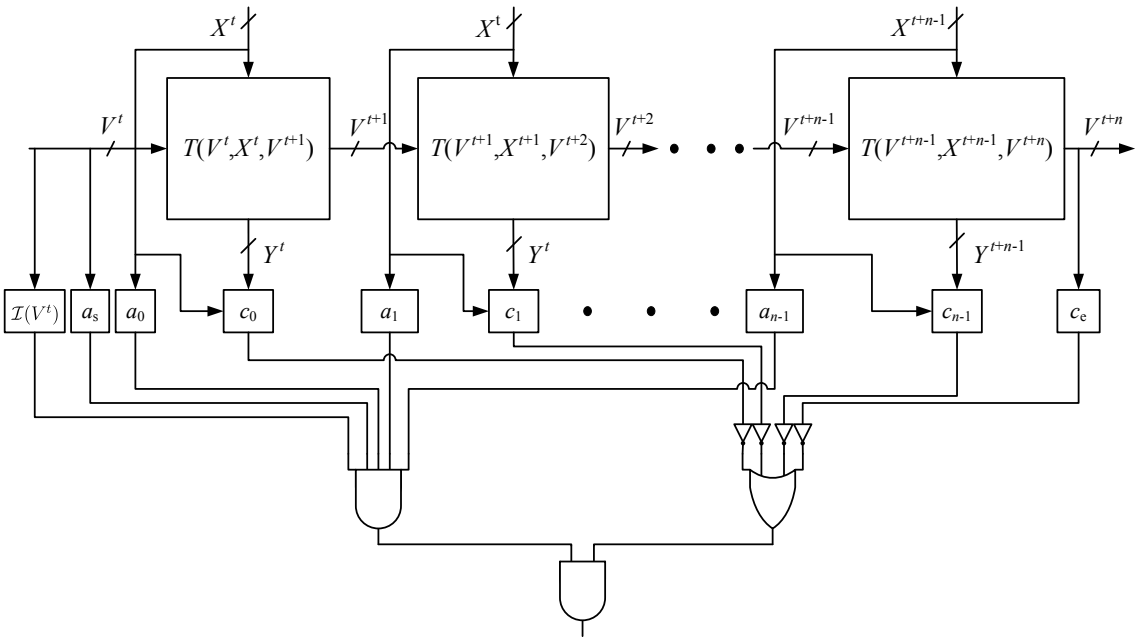


Figure 4.5: Circuit representing the set of paths with length n fulfilling the property ψ

This proposition consists of only a single translation $[\![\phi]\!]$ of the property and of a time frame expansion model for n cycles. Note that n only results from the length of the property and does not depend on the diameter of the design. It is also different from k in BMC. Furthermore, the state vector V^t for the starting states of the time frame expansion are constrained by the invariant $\mathcal{I}$. The proposition is represented as the iterative circuit

being connected with the circuits that represent the assumption a, the commitment c, and the invariant $\mathcal{I}$. Figure 4.5 illustrates this circuit.

In Equation 4.12, the invariant $\mathcal{I}$ includes the initial states and is closed under the image computation. It is defined as follows:

Definition 4.2. ***(Invariant)*** *Given an encoded FSM $M = (I, S, S_0, \Delta, O, \Lambda)$, an invariant $\mathcal{I}$ of M is a Boolean proposition that fulfills the following conditions:*

- $S_0 \rightarrow \mathcal{I} = 1$
- $\mathcal{I} = Img(T, \mathcal{I}) = \exists_X, \exists_V T(V, X, V') \wedge \mathcal{I}(V)$ □

Lemma 4.1. *Let $\mathcal{I}$ be an invariant. Then, for any t*

$$S_0(V^0) \wedge \bigwedge_{j=0}^{j=t-1} T(V^j, X^j, V^{j+1}) \rightarrow \mathcal{I}(V^t) = 1$$

Proof. The lemma can be proven by induction.

- Base case: It is from Definition 4.2, $S_0 \rightarrow \mathcal{I} = 1$
- Inductive case: If it is:

  $$S_0(V^0) \wedge \bigwedge_{j=0}^{j=t-1} T(V^j, X^j, V^{j+1}) \rightarrow \mathcal{I}(V^t) = 1$$

  Then, according to Lemma 2.2, it is:

  $$S_0(V^0) \wedge \bigwedge_{j=0}^{j=t-1} T(V^j, X^j, V^{j+1}) \wedge T(V^t, X^t, V^{t+1}) \rightarrow \mathcal{I}(V^t) \wedge T(V^t, X^t, V^{t+1}) = 1$$

  According to Lemma 2.4, it is:

  $$\mathcal{I}(V^t) \wedge T(V^t, X^t, V^{t+1}) \rightarrow \exists_X \exists_{V^t} \mathcal{I}(V^t) \wedge T(V^t, X^t, V^{t+1}) = 1$$

  According to Lemma 2.1 and Definition 4.2, we have:

  $$\mathcal{I}(V^t) \wedge T(V^t, X^t, V^{t+1}) \rightarrow \mathcal{I}(V^{t+1}) = 1$$

  Hence, it is:

  $$S_0(V^0) \wedge \bigwedge_{j=0}^{j=t} T(V^j, X^j, V^{j+1}) \rightarrow \mathcal{I}(V^{t+1}) = 1$$

□

Lemma 4.2. *If $[\![M, \psi]\!]^t_{IPC} = 0$, then $[\![M, \psi]\!]^k_{BMC} = 0$ for every k.*

Proof. $[\![M,\psi]\!]^t_{\text{IPC}} = 0$ means that, for any t, it is:

$$\mathcal{I}(V^t) \wedge \bigwedge_{j=0}^{n} T(V^{t+j}, X^{t+j}, V^{t+j+1}) \wedge [\![\psi]\!]_t = 0$$

According to Lemma 4.1, and Lemma 2.3 , we can replace $\mathcal{I}(V^t)$ by

$$S_0(V^0) \wedge \bigwedge_{j=0}^{j=t-1} T(V^j, X^j, V^{j+1})$$

to have:

$$S_0(V^0) \wedge \bigwedge_{j=0}^{j=t-1} T(V^j, X^j, V^{j+1}) \wedge \bigwedge_{j=0}^{n} T(V^{t+j}, X^{t+j}, V^{t+j+1}) \wedge [\![\psi]\!]_t = 0$$

Or it is:

$$S_0(V^0) \wedge \bigwedge_{j=0}^{j=t+n-1} T(V^j, X^j, V^{j+1}) \wedge [\![\psi]\!]_t = 0$$

Let

$$\bigwedge_{j=t+n}^{j=k+n-1} T(V^j, X^j, V^{j+1})$$

be "anded" to the above equation, we finally have:

$$S_0(V^0) \wedge \bigwedge_{j=0}^{j=k+n-1} T(V^j, X^j, V^{j+1}) \wedge [\![\psi]\!]_t = 0$$

Consider all $t = 0 \ldots k$ for the above equation, and sum up the results, we have:

$$[\![M,\psi]\!]^k_{\text{BMC}} = S(V^0) \wedge \bigwedge_{j=0}^{j=k+n-1} T(V^j, X^j, V^{j+1}) \wedge \bigvee_{t=0}^{t=k} [\![\psi]\!]_t = 0$$

□

Theorem 4.1. *If $[\![M,\psi]\!]^t_{IPC}$ is unsatisfiable, then $M \models \phi$.*

Proof. According to Lemma 4.2, if $[\![M,\psi]\!]^t_{\text{IPC}}$ is unsatisfiable, $[\![M,\psi]\!]^k_{\text{BMC}}$ is also unsatisfiable for any value of k. Hence, we can not find any k such that $M \models \psi$. It means that $\overline{M \models \psi}$. And, therefore, it is $M \models \phi$. □

Theorem 4.1 states that an unbounded proof of the original property in Equation 4.6 can be done by checking the satisfiability of $[\![M,\psi]\!]^t_{\text{IPC}}$. This formulates the concept of *Interval Property Checking (IPC)*.

As another aspect, IPC can be elaborated as invariant checking [BCL+94]. The set of "bad" states that violates ϕ is represented by:

$$\bigwedge_{j=0}^{j=n} T(V^{t+j}, X^{t+j}, V^{t+j+1}) \wedge [\![\psi]\!]_t$$

Therefore, checking ϕ is equivalent to choosing an invariant $\mathcal{I}$ and checking that $\mathcal{I}$ and the set of "bad" states are disjoint.

Depending on what invariant $\mathcal{I}$ is chosen in Equation 4.12 a false counterexample may be generated based on some state for V^t that is not reachable in the system. Hence, the quality of the invariant $\mathcal{I}$ is crucial for the success of this method.

Fortunately, for most standard cases of SoC module verification the trivial invariant $\mathcal{I} = 1$ is sufficient. This may be surprising at first glance but becomes intuitive if we realize that the iterative circuit model itself contains a lot of local reachability information which is sufficient for many interval properties. As a rule of thumb, we may state that verifying modules for *computation* like hardware accelerators or processor cores usually does not require sophisticated invariants. This only changes when *communication* modules are considered. Here, global dependencies between state information over long time windows exist and require significant effort to provide an invariant $\mathcal{I}$ of sufficient strength. This is the reason why this proposed approach is developed with a focus on protocol compliance verification.

Due to the nature of the iterative circuit model, even in the case of communication modules the required invariants are much weaker than we would expect from other property checking formulations. Moreover, they are often intuitive to the designer. Therefore, in the case of communication modules it is common practice to manually inspect the code and to define invariants that are appropriate for proving properties of the type described above.

This is illustrated by means of the example in Figure 4.2. The design uses a counter to keep track of time. The output cnt of the counter is evaluated whenever the design is in state $wait$ and a transition from $wait$ to state $ready$ is performed if and only if $cnt = n$. However, when the property in Equation 4.7 is constructed, the verification engineer does not know anything about how the main-states are related to the states in other parts of the design. Determining this information by manual code inspection is very tedious and should be avoided whenever possible. Therefore, the verification engineer initially chooses the invariant $\mathcal{I} = 1$. If the resulting proposition in Equation 4.12 is proven to be unsatisfiable this implies that the property holds and the verification engineer has identified a piece of proper behavior. However, the proof of the property in Equation 4.7 fails and a counterexample is generated when Equation 4.12 is set up with the invariant $\mathcal{I} = 1$. This counterexample, however, turns out to be a false negative. For example, the property checker may claim that $(\hat{V} \equiv idle \wedge cnt \equiv 1)$ leads to $(end \equiv 1)$ after $n - 1$ cycles rather than n cycles. The obvious reason for this false negative to appear is that the value of the counter at the beginning of the considered transaction is neither defined by the property nor defined by the invariant $\mathcal{I}$. Anyhow, inspecting the design reveals

that the counter will always take the value $cnt = 0$ at this point in time. Specifying this in the property would violate our template because the counter is not part of the main-FSM. Note that the missing constraint is independent of the property and results from reachability in the overall design. Therefore, this information is taken into account by a refinement of the invariant $\mathcal{I}$. The new invariant becomes:

$$\mathcal{I}(V^t) = ((\hat{V}^t \equiv idle) \rightarrow (cnt^t \equiv 0)) \tag{4.13}$$

This leads to a strengthened ITL property as in Figure 4.6. The implicative structure of the invariant required in this example is a typical representative for the invariants usually required when following the above methodology. Since each property based on the proposed template assumes a concrete state $\hat{s}$ for the main-FSM it is usually sufficient to restrict the values for certain non main-state variables (i.e., sub-FSM state variables) such that the encoded states are reachable simultaneously with the main-FSM state $\hat{s}$. This results in the invariant of the following form:

$$\mathcal{I}(V^t) = \big((\hat{V} \equiv \hat{s}) \rightarrow a_{\hat{s}}), \tag{4.14}$$

where $a_{\hat{s}}$ is a Boolean expression which has to be satisfied by the values of the sub-FSM variables that occur whenever the main-FSM is in state $\hat{s}$.

Taking into account such reachability constraints is very important to avoid false negatives when using IPC based on Equation 4.12. Note that the validity of reachability constraints obtained from manual analysis must be proven by writing additional properties.

property strengthenedProperty **is**
  assume:
    at t: $\hat{V} \equiv idle$;
    at t: $d\_en \equiv 1$;
    at t: $cnt \equiv 0$;
  prove:
    at t+n: $end \equiv 1$;
    at t+n+1: $\hat{V} \equiv ready$;
end property;

Figure 4.6: Example of the property strengthened with reachability constraints.

The IPC-based methodology for verifying protocol implementations in SoCs is summarized in Figure 4.7. The main-FSM is usually determined manually in the RTL code of the design by identifying a set of important control states, as described before. Moreover, commercial tools such as [Sof09] are available that can be used to automatically extract a main-FSM from the RTL description and to visualize its state transition graph. By inspecting the RTL code and by seeking explanations in the specification the verification engineer formulates properties that link implementation and specification. Using

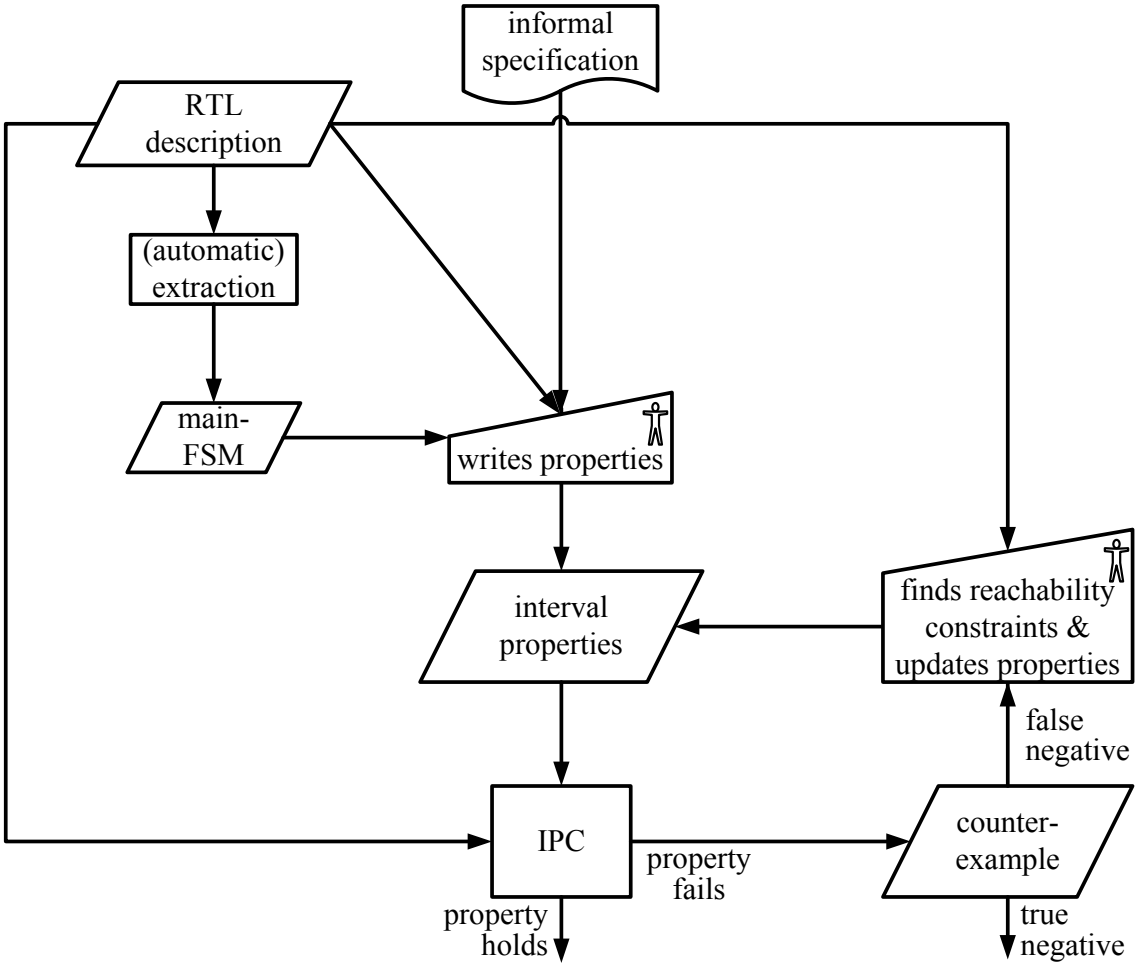


Figure 4.7: IPC-based verification flow

the templates proposed in the previous section properties are written which specify functional operations of the design corresponding to a sequence of states in the main-FSM. Inspecting only the main-FSM of a design requires a lot less effort than analyzing the entire source code. On the other hand, properties based on only the main-FSM are likely to produce false negatives because reachability constraints linking main-states to other states in the design are ignored. As explained above, false negatives may occur and it is industrial practice to derive reachability constraints manually. This increases the verification effort drastically because it requires detailed inspection of larger parts of the source code in addition to that of the main-FSM.

In the next chapter, it will be described how to reduce this effort by the *Transition-by-Transition* (TBT) FSM traversal for approximative reachability analysis.

Chapter 5

Transition by Transition FSM Traversal

In this chapter, the proposed reachability analysis algorithms are described. The reachability analysis is tailored to providing IPC with the essential invariant $\mathcal{I}$ such that false negatives can be avoided. As the properties are set-up based on transitions of the main-FSM, it is natural to explore the main-FSM to identify the invariant $\mathcal{I}$. In this chapter, we will see how the the existence of the main-FSM can be exploited systematically in reachability analysis and how to partition both the transition relation and the state space such that the computational complexity is reduced drastically.

This chapter, in the first section, will first present an exact reachability analysis which decomposes the state space and the transition relation using the main-FSM. Then, in the second section, an approximative algorithm will be described. The SAT-based implementation of the algorithms is represented in the third section. The fourth section will compare the proposed method with previous work represented in Chapter 3. Finally, experimental results of the application of the proposed algorithms in the IPC-based verification methodology are reported.

5.1 Decomposition of State Space and Transition Relation

5.1.1 Definition of the Main-FSM

The main-FSM has been introduced conceptually in the previous chapter. It plays an important role in the system since it controls the overall behavior of the system. First, let us define the main-FSM of an encoded FSM formally.

Definition 5.1. ***(Main-FSM)*** *Let T be the transition relation of an encoded FSM $M = (I, S, S_0, \Delta, O, \Lambda)$. Let V and $\hat{V} \subseteq V$ be the vectors of state variables and main state variables of M. The STG of the main-FSM of M is a triple $(\hat{S}, \hat{S}_0, \hat{T})$ where*

- $\hat{S} = \{\hat{s} \in B^{|\hat{V}|} | \exists s \in S : s(\hat{V}) = \hat{s}\}$ *is the set of main-states*
- $\hat{S}_0 = \{\hat{s}_0 \in \hat{S} | \exists s_0 \in S_0 : s_0(\hat{V}) = \hat{s}_0\}$ *is the set of initial main-states*

- $\hat{T} = \{(\hat{s}_1, \hat{s}_2) \in \hat{S}^2 | \exists s_1, s_2 \in S \wedge \exists i \in I : (s_1, i, s_2) \in T \wedge s_1(\hat{V}) = \hat{s}_1 \wedge s_2(\hat{V}) = \hat{s}_2\}$ *is the main-transition relation.* □

In other words, the triple $(\hat{S}, \hat{S}_0, \hat{T})$ describes the behavior of the isolated main-FSM. The states in $\hat{S}$ are encoded solely by the state variables belonging to the main-FSM such that an encoded main-state is a sub-vector of the entire state vector V.

Given a main state variable $\hat{v}_j \in \hat{V}$, the next state function of $\hat{v}_j$ is denoted by $\hat{\delta}_j$. The vector of next state functions corresponding to the main state variable $\hat{V}$ is denoted by $\hat{\Delta} = \Delta_{\hat{V}}$.

Even though the main-FSM as defined above is a sound abstraction of the system, in contrast to other model checking techniques with abstraction, this approach does not rely on proving properties in the abstract machine. In fact, an abstract machine that would allow us to prove properties of the format given in the previous section would be significantly more complex than the main-FSM. Such an abstraction would not only contain the main-states but also many other states representing the input/output behavior of the concrete system. Instead, the main-FSM is used only to derive a decomposition of the concrete system and to conduct an approximative reachability analysis.

The main-FSM of the system can easily be extracted from the RTL description. Note that the extraction technique must yield a main-FSM conforming to Definition 5.1, i.e., all main-states must be identified and no state may be missed. Since the main-FSM is very small, in practice, this is easy to guarantee.

The finite state machine of the FSM M can be viewed as the product machine of all FSMs in M, including the main-FSM. The main-FSM and the sub-FSMs often communicate via a request/acknowledge mechanism. The main-FSM enters a main-state and sends a request to a sub-FSM which performs the required task. The main-FSM waits in that main-state until it receives the acknowledge from the sub-FSM, and then goes to another main-state. Please observe that the state space of the design is much smaller than the Cartesian product of two sets of states of the main-FSM and the sub-FSM.

In the example design shown in Figure 4.2, when the main-FSM is in states $idle$ or $ready$ the sub-FSM remains in state $cnt = 0$. The value of cnt only changes when the main-FSM is in state $wait$. Therefore, the state space of the design has only $n + 2$ states instead of $3 * n$ states as resulting from the Cartesian product.

5.1.2 Decomposition Using the Main-FSM

The state space of the system can be partitioned into sets of system states corresponding to main-states. Whenever the main-FSM is in one of its states, the other FSMs in the system may each be in one of many different states. Each single state of the main-FSM therefore corresponds to a set of states of the complete FSM M:

Definition 5.2. ***(Constrained Set of States)*** *Let $\hat{s} \in \hat{S}$ be a main-state of an FSM M. The set of states corresponding to $\hat{s}$ is*

$$S_{\hat{s}} = \{s \in S | s(\hat{V}) = \hat{s}\}$$

□

In the same way, we can describe the transitions of the complete FSM M in terms of transitions of the main-FSM. When the main-FSM moves from one of its states, $\hat{s}_1$, to another state, $\hat{s}_2$, the FSM M correspondingly moves from a state in $S_{\hat{s}_1}$ to another state in $S_{\hat{s}_2}$. By considering all possible transitions between such states corresponding to $\hat{s}_1$ and $\hat{s}_2$, we define the transition relation of the system corresponding to a transition of the main-FSM:

Definition 5.3. *(***Constrained Transition Relation***) Let $(\hat{s}_1, \hat{s}_2) \in \hat{T}$ be a main-transition of an FSM M. The transition relation corresponding to $(\hat{s}_1, \hat{s}_2)$ is*

$$T_{\hat{s}_1 \to \hat{s}_2} = \{(s_1, i, s_2) \in T | s_1 \in S_{\hat{s}_1} \wedge s_2 \in S_{\hat{s}_2}\}$$

□

Given this definition of the transition relation $T_{\hat{s}_1 \to \hat{s}_2}$, we define image computation corresponding to a transition $\hat{s}_1 \to \hat{s}_2$ of the main-FSM.

Definition 5.4. *(***Constrained Image Computation***) Let $(\hat{s}_1, \hat{s}_2) \in \hat{T}$ be a main-transition of an FSM M. The constrained image computation corresponding to $(\hat{s}_1, \hat{s}_2)$ of a set of states $Z \in S$ is*

$$img_{\hat{s}_1 \to \hat{s}_2}(T, Z) = \{s_2 \in S | \exists s_1 \in Z \wedge \exists i \in I \wedge (s_1, i, s_2) \in T_{\hat{s}_1 \to \hat{s}_2}\}$$

□

The following two lemmas show how the STG of an FSM M can be explored based on single transitions of its main-FSM.

Lemma 5.1. *The set of states of an FSM M can be decomposed into disjoint sets of states corresponding to main-states:*

$$S = \bigcup_{\hat{s} \in \hat{S}} S_{\hat{s}}$$

Proof. We prove the lemma in three steps:

1. Consider a main-state $\hat{s}_1$ and a state $s_1 \in S_{\hat{s}_1}$. By Definition 5.2 it is $s_1 \in S$. Therefore, it is
$$\bigcup_{\hat{s}} S_{\hat{s}} \subseteq S$$

2. Consider a state $s_2 \in S$, since $\hat{V} \subseteq V$ there exists a main-state $\hat{s}_2 \in \hat{S}$ such that $s_2(\hat{V}) = \hat{s}_2$. Hence, it is $s_2 \in S_{\hat{s}_2}$ and
$$S \subseteq \bigcup_{\hat{s}} S_{\hat{s}}$$

3. Next we prove that $S_{\hat{s}_1} \cap S_{\hat{s}_2} = \emptyset$ for all $\hat{s}_1 \neq \hat{s}_2$. Suppose that there exists a state $s \in S_{\hat{s}_1} \cap S_{\hat{s}_2}$ then, by Definition 5.2 it is $\hat{s}_1 = s(\hat{V}) = \hat{s}_2$.

From (1), (2) and (3), the lemma is proven. □

Lemma 5.1 states that the states of the FSM can be computed implicitly and exactly as the union of states corresponding to main-states.

Lemma 5.2. *Let $Z_{\hat{s}_1}$ be a set of states corresponding to a main-state $\hat{s}_1$. The next-states of $Z_{\hat{s}_1}$ can be calculated as:*

$$img(T, Z_{\hat{s}_1}) = \bigcup_{\forall \hat{s}_2 : (\hat{s}_1, \hat{s}_2) \in \hat{T}} img_{\hat{s}_1 \to \hat{s}_2}(T, Z_{\hat{s}_1})$$

Proof. The lemma is proven in two steps:

1. Consider a main-transition $(\hat{s}_1, \hat{s}_2) \in \hat{T}$ and a state $s_2 \in img_{\hat{s}_1 \to \hat{s}_2}(T, Z_{\hat{s}_1})$. By Definition 5.4 there exist a state $s_1 \in Z_{\hat{s}_1}$ and an input $i \in I$ such that $(s_1, i, s_2) \in T_{\hat{s}_1 \to \hat{s}_2}$. By Definition 5.3 it is $(s_1, i, s_2) \in T$. Hence, it is $s_2 \in img(T, Z_{\hat{s}_1})$. Therefore it is

$$\bigcup_{\forall \hat{s}_2 : (\hat{s}_1, \hat{s}_2) \in \hat{T}} (img_{\hat{s}_1 \to \hat{s}_2}(T, Z_{\hat{s}_1})) \subseteq img(T, Z_{\hat{s}_1})$$

2. If we consider a state $s_2 \in img(T, Z_{\hat{s}_1})$ then there exist a state $s_1 \in Z_{\hat{s}_1}$ and an input $i \in I$ such that $(s_1, i, s_2) \in T$. By Definition 5.1 there exists a transition $(\hat{s}_1, \hat{s}_2) \in \hat{T}$ such that $s_1(\hat{V}) = \hat{s}_1 \wedge s_2(\hat{V}) = \hat{s}_2$. By Definition 5.3 it is $s_2 \in img_{\hat{s}_1 \to \hat{s}_2}(T, Z_{\hat{s}_1})$. Therefore, it is

$$img(T, Z_{\hat{s}_1}) \subseteq \bigcup_{\forall \hat{s}_2 : (\hat{s}_1, \hat{s}_2) \in \hat{T}} (img_{\hat{s}_1 \to \hat{s}_2}(T, Z_{\hat{s}_1}))$$

From (1) and (2), the lemma is proven. □

By Lemma 5.2 the next-states of a set of states corresponding to a main-state can be calculated using the *constrained image* computation for every outgoing transition of the corresponding main-state.

This is the basis of the proposed *transition-by-transition* FSM traversal algorithm. Note that TBT traversal does not only create a partition of the state set but also partitions the transition relation of the design.

The algorithm in Figure 5.1 calculates reachable states of an FSM by decomposing the state space and the transition relation based on the main-FSM. The procedure *tbt_traversal* takes the transition relation, the set of initial states and the main-FSM of the FSM as inputs. It returns the sets of reachable states corresponding to main-states.

At the beginning of a procedure call, the sets of states corresponding to the main-states are empty except for the sets of states corresponding to the initial main-states which

```
0: procedure tbt_traversal(T, S_0, Ŝ, T̂, Ŝ_0) {
1:     queue ← Ŝ_0;
2:     for each ŝ ∈ Ŝ {
3:       if (ŝ ∈ Ŝ_0) S_ŝ ← {s_0 ∈ S_0 | s_0(V̂) ≡ ŝ};
4:       else S_ŝ ← ∅;
5:     }
6:     while (queue ≠ ∅){
7:       ŝ ← dequeue(queue);
8:       for each ŝ' where (ŝ, ŝ') ∈ T̂ {
9:         S_{ŝ→ŝ'} ← img_{ŝ→ŝ'}(T, S_ŝ);
10:        S_ŝ' ← S_ŝ' ∪ S_{ŝ→ŝ'};
11:        if (S_ŝ' changes) enqueue(ŝ', queue);
12:      }
13:    }
14:    return({S_ŝ});
15: }
```

Figure 5.1: TBT traversal algorithm.

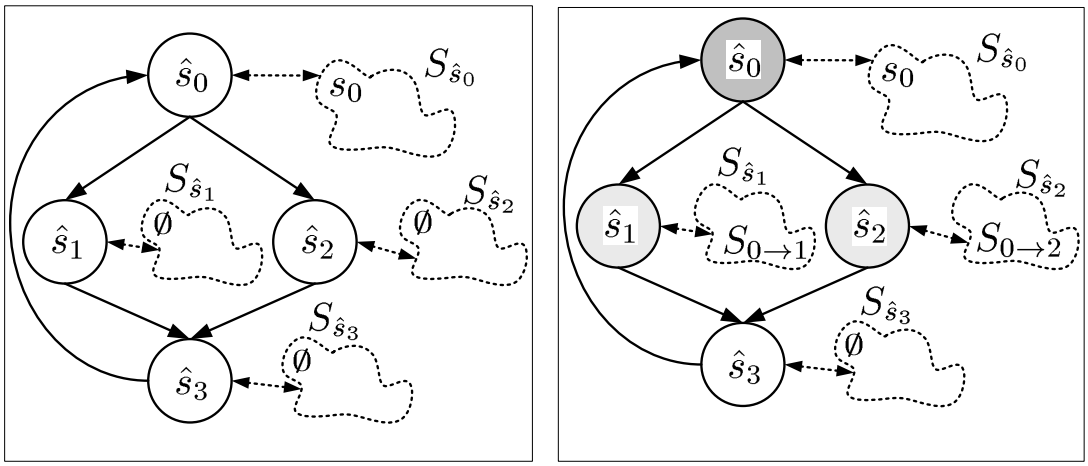
contain the given initial states. A *queue* is used to keep track of the main-states that need to be processed next. First, the queue contains the initial main-states. A main-state $\hat{s}$ is added to the queue if and only if its corresponding set of states changes. In this case, the sets of states corresponding to the next-states of $\hat{s}$ have to be recalculated. For a main-state $\hat{s}$ in the queue, the algorithm considers all its next-states. The set of states $S_{\hat{s}\rightarrow\hat{s}'}$ corresponding to a transition $(\hat{s}, \hat{s}')$ is calculated by the *constrained image* computation. The set of states $S_{\hat{s}'}$ corresponding to main-state $\hat{s}'$ is extended by $S_{\hat{s}\rightarrow\hat{s}'}$. If new states are added to $S_{\hat{s}'}$ by this computation, $\hat{s}'$ is entered into the queue. Procedure *tbt_traversal* implements a fixed-point calculation for the sets of reachable states corresponding to main-states. The fixed-point is reached if and only if the queue is empty.

The algorithm is illustrated in Figure 5.2 where the main-FSM of the FSM consists of four main-states $\hat{s}_0, \ldots, \hat{s}_3$. We want to calculate the sets of states $S_{\hat{s}_0}, \ldots, S_{\hat{s}_3}$ corresponding to the main-states.

At the beginning, in Figure 5.2(a), the set of states $S_{\hat{s}_0}$ contains the initial state s_0. The other sets of states are empty and the *queue* consists of $\hat{s}_0$.

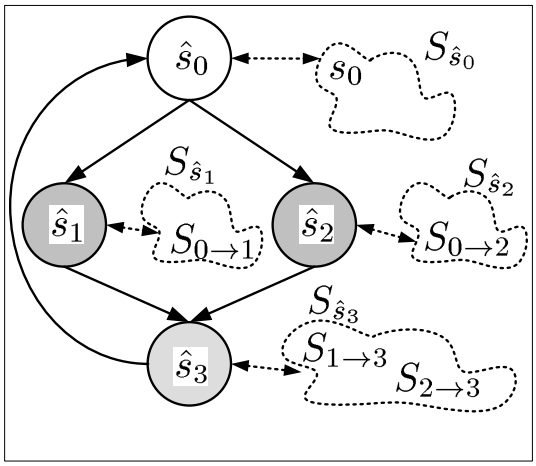
Next, in Figure 5.2(b), the main-state $\hat{s}_0$ is removed from the *queue*. The algorithm calculates the sets of states $S_{0\rightarrow1}$ and $S_{0\rightarrow2}$ corresponding to the outgoing transitions $(\hat{s}_0, \hat{s}_1)$ and $(\hat{s}_0, \hat{s}_2)$ of $\hat{s}_0$ applying the *constrained image* computation. $S_{0\rightarrow1}$ and $S_{0\rightarrow2}$ are united with the old sets $S_{\hat{s}_1}$ and $S_{\hat{s}_2}$, respectively, to obtain the new sets $S_{\hat{s}_1}$ and $S_{\hat{s}_2}$. Because $S_{\hat{s}_1}$ and $S_{\hat{s}_2}$ change, the main-states $\hat{s}_1$ and $\hat{s}_2$ are added into the *queue*.

In Figure 5.2(c), $\hat{s}_1$ and $\hat{s}_2$ are removed from the *queue*. Similarly, the sets of states $S_{1\rightarrow3}$ and $S_{2\rightarrow3}$ corresponding to the outgoing transitions $(\hat{s}_1, \hat{s}_3)$ and $(\hat{s}_2, \hat{s}_3)$ are calcu-

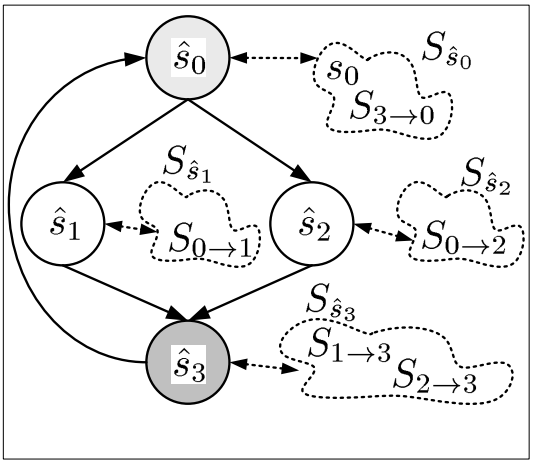


(a) $queue = \{\hat{s}_0\}$

(b) $queue = \{\hat{s}_1, \hat{s}_2\}$



(c) $queue = \{\hat{s}_3\}$



(d) $queue = \emptyset$

Figure 5.2: Example of TBT traversal algorithm

lated. $S_{1\rightarrow 3}$ and $S_{2\rightarrow 3}$ are added to $S_{\hat{s}_3}$. Because $S_{\hat{s}_3}$ changes we add $\hat{s}_3$ into the *queue*.

In Figure 5.2(d), we remove $\hat{s}_3$ from the *queue* and calculate $S_{3\rightarrow 0}$ corresponding to the transition $(\hat{s}_3, \hat{s}_0)$. $S_{3\rightarrow 0}$ is then added to the old $S_{\hat{s}_0}$ to obtain the new $S_{\hat{s}_0}$. Let us assume that $S_{\hat{s}_0}$ does not change because $S_{3\rightarrow 0}$ does not contain any new states. Therefore, $\hat{s}_0$ is not entered into the *queue*. Now, because the *queue* is empty the algorithm terminates.

Theorem 5.1. *Procedure tbt_traversal calculates all reachable states of the FSM exactly.*

Proof. Consider a main-state $\hat{s}$ in the queue, the *for* loop in lines $6-12$ iterates over all outgoing transitions of $\hat{s}$. Therefore, it calculates all next-states for the set of states corresponding to $\hat{s}$ according to Lemma 5.2. This means that all next-states of the set of states corresponding to a main-state in the queue are visited by the loop. Let state s' be a newly visited next-state and let it be added to the set of states corresponding to main-state $\hat{s}'$. It changes the set $S_{\hat{s}'}$ and state $\hat{s}'$ is written into the queue. Therefore, the next-states of s' are visited later when $\hat{s}'$ is read from the queue. At the beginning the queue contains the initial main-states with the corresponding sets of states containing all initial states. Thus, all next-states of the initial states are visited. Recursively, all reachable states of the FSM are visited by the procedure. According to Lemma 5.1, the calculated reachable states are partitioned into sets of states corresponding to the main-states. □

According to Theorem 5.1, the *tbt_traversal* algorithm is an exact reachability analysis algorithm. Even though the proposed decomposition strategy helps to reduce the computation complexity the algorithm is still not able to handle industrial designs. Hence, we need to further decompose the design to calculate the approximate set of reachable states as described in the next section.

5.2 Decomposing Using Sub-FSMs

The previous section outlined how to partition the traversal of an FSM using the transitions of a main-FSM. As a next step, we study how the main-FSM can be used for automatically decomposing the FSM into sub-FSMs to be individually traversed. The individual traversals are performed following the hierarchy of the sub-FSM. This reachability analysis for the overall design is approximative because we may miss correlations between sub-FSMs.

5.2.1 Partitioning the Design into Sub-FSMs

The design is partitioned into sub-FSMs in three steps:

1. For each next-state function δ_i the support set is calculated under the constraints imposed by main-FSM transitions.

2. Based on the constrained support sets, a variant of the latch dependency graph is built and decomposed into strongly connected components.

3. The latches within each strongly connected component are grouped together and treated as a sub-FSM. The hyper-graph of strongly connected components each representing a sub-FSM is a DAG and can be levelized. This results in a hierarchy for the sub-FSM.

For the remainder of this subsection, these steps will be described with the definitions of the required notations and procedures.

Functional Dependencies of Latches

The goal of the first step is to determine functional dependencies between the design registers under the individual transitions of the main-FSM. The support set of a Boolean function has been defined in Chapter 2. In this section, only the latches in the support set of the next-state functions are considered; the primary inputs are ignored.

Definition 5.5. *(**Latch Support Set**) Let V be the set of state variables of an encoded FSM $M = (I, S, S_0, \Delta, O, \Lambda)$. The latch support set of a state variable $v_j \in V$ is:*

$$supp(v_j) = supp(\delta_j) \cap V$$

□

As a next step we further constrain the support set by considering a specific main-transition $(\hat{s}_1, \hat{s}_2) \in \hat{T}$.

Definition 5.6. *(**Constrained Latch Support Set**) Let V be the set of state variables of an FSM $M = (I, S, S_0, \Delta, O, \Lambda)$. Let $(\hat{s}_1, \hat{s}_2) \in \hat{T}$ be a main-transition of the main-FSM of M. The constrained latch support set of a state variable $v_j \in V$ corresponding to $(\hat{s}_1, \hat{s}_2)$ is:*

$$supp_{\hat{s}_1 \rightarrow \hat{s}_2}(v_j) = supp(\delta_j \downarrow (\hat{V} \equiv \hat{s}_1) \downarrow (\hat{\Delta} \equiv \hat{s}_2)) \cap V$$

□

Note that by definition, the generalized cofactor of δ_j with respect to $(\hat{\Delta} \equiv \hat{s}_2)$ is not uniquely determined. Hence, the constrained latch support set of a latch is not uniquely determined. However, as it turns out, small constrained latch support sets are beneficial to partition the FSM into sub-FSMs. Section 5.3 will describe a SAT-based implementation to approximate a minimal constrained latch support.

Constrained Latch Dependency Graph

In the second step, the constrained latch support sets are used to build a constrained latch dependency graph as follows:

Definition 5.7. *(**Constrained Latch Dependency Graph**) Let $\hat{T}$ be the main-transition relation of an FSM M. The constrained dependency graph of the state variables is a*

directed graph $G(V, E)$ where the set of vertices is given by the set of state variables V. The set of edges E is defined as

$$E = \{(v_i, v_j) \in V^2 | v_i \in \bigcup_{(\hat{s}_1, \hat{s}_2) \in \hat{T}} \mathit{supp}_{\hat{s}_1 \to \hat{s}_2}(v_j)\}.$$

□

Note that the constrained latch dependency graph differs from the usual latch dependency graph based on the standard support set computation where the edges are defined by

$$E' = \{(v_i, v_j) \in V^2 | v_i \in \mathit{supp}(\delta_j)\}.$$

The difference of the non-constrained versus the constrained latch dependency graph is illustrated by means of an example. Consider an FSM with a main-FSM encoded by a single state variable $\hat{v}$. Further, let the FSM have two non-main-state variables v_1, v_2. Finally, the transition functions for $\hat{v}$ and v_2 shall be given by $v_2' = v_1 \wedge \hat{v}$ and

$$\hat{v}' = \begin{cases} \hat{v}, & \text{if } v_1 \equiv 0 \\ \overline{\hat{v}} & \text{else.} \end{cases}$$

In this case, it is $v_1 \in \mathit{supp}(\delta_2)$. On the other hand, v_1 is constant under any transition of the main-FSM and therefore will not be element of any constrained support set.

Partitioning Designs into Sub-FSMs

The constrained latch dependency graph is partitioned using the SCC algorithm represented in Chapter 2 to identify strongly connected components (SCC). Each SCC corresponds to a sub-FSM. The graph of sub-FSMs is a directed acyclic graph which can be levelized by a depth-first search algorithm. The level of each sub-FSM k is denoted by $\mathit{level}(k)$.

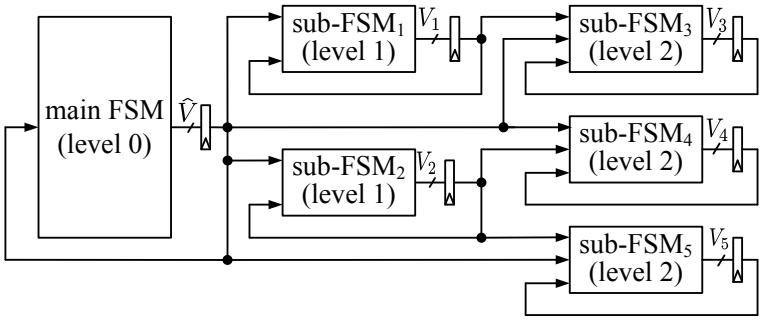


Figure 5.3: Partitioned structure of an FSM

An example of the hierarchical structure of the FSM after being partitioned is shown in Figure 5.3. The FSM consists of one main-FSM and five sub-FSMs. The sets of state

variables of the main-FSM and the five sub-FSMs are $\hat{V}, V_1, \ldots, V_5$, respectively. Sub-FSMs 1 and 2 are on level 1, the other sub-FSMs are on level 2. As shown in Figure 5.3, the SCC decomposition produces *uni-directional* interaction among the sub-FSMs of the FSM. The control information stored in the state variables is only passed in one direction from the lower-level sub-FSMs to the higher level sub-FSMs. Note that a non-constrained decomposition into SCCs is generally not useful for industrial designs because this type of decomposition often produces only large SCCs. If, however, the dependency graph is constrained by the transitions of the main-FSM the situation is different and many small SCCs can often be identified.

5.2.2 Traversing Sub-FSMs

0: procedure tbt_subFSM$(T, S_0, \hat{S}, \hat{T}, \hat{S}_0, V_k, \{\tilde{S}_{\hat{s},l} | \forall \hat{s} \in \hat{S}, \forall l < k\})$ {
1: queue $\leftarrow \hat{S}_0$;
2: **for each** $\hat{s} \in \hat{S}$ {
3: **if** $(\hat{s} \in \hat{S}_0)$ $\tilde{S}_{\hat{s},k} \leftarrow \{\exists_{V \setminus V_k \setminus \hat{V}} s_0 \in S_0 | s_0(\hat{V}) \equiv \hat{s}\}$;
4: **else** $\tilde{S}_{\hat{s},k} \leftarrow \emptyset$;
5: }
6: **while** (queue $\neq \emptyset$){
7: $\hat{s} \leftarrow$ dequeue(queue);
8: $\tilde{T}_{\hat{s},k} \leftarrow \bigwedge_{v_i \in \{\hat{V} \cup V_k\}} (v_i' \equiv \delta_i) \downarrow \bigwedge_{l<k} \tilde{S}_{\hat{s},l}(V_l)$;
9: **for each** $\hat{s}'$ **where** $(\hat{s}, \hat{s}') \in \hat{T}$ {
10: $\tilde{S}_{\hat{s} \to \hat{s}',k} \leftarrow img_{\hat{s} \to \hat{s}'}(\tilde{T}_{\hat{s},k}, S_{\hat{s}})$;
11: $\tilde{S}_{\hat{s}',k} \leftarrow \tilde{S}_{\hat{s}'} \cup \tilde{S}_{\hat{s} \to \hat{s}'}$;
12: **if** $(\tilde{S}_{\hat{s}'}$ changes$)$ enqueue$(\hat{s}'$, queue$)$;
13: }
14: }
15: **return**$(\{\tilde{S}_{\hat{s}}\})$;
16: }

Figure 5.4: Approximative TBT traversal algorithm for a sub-FSM k

For the individual traversal of the sub-FSMs determined in the previous section we consider the corresponding state vectors $V_1, \ldots, V_r$. Without loss of generality we may assume that these vectors are topologically sorted with respect to the SCC graph, i.e., *level*$(j) <$ *level*(k) for $1 \leq j < k \leq r$. For each sub-FSM $k = 1, \ldots, r$ together with the main-FSM, an abstract model M_k is generated by applying localization abstraction to the state variables $v_j \notin \hat{V} \cup V_k$. In other words, in M_k only the state variables of the sub-FSM V_k and the main-FSM $\hat{V}$ are considered as state variables. All other state variables are treated as pseudo-inputs. Similarly to the machine-by-machine approach of [CHM+96a]

the individual abstract machine M_k is traversed separately. When the sub-FSM is traversed, the pseudo-inputs $V_1, \ldots, V_{k-1}$ are constrained with the sets of reachable states that have been calculated for the corresponding sub-FSMs in previous runs. Because the TBT algorithm calculates only the sets of reachable states that correspond to main-states, the transition relation $\tilde{T}_{\hat{s}_1,k}$ of the sub-FSM k corresponding to a main-state $\hat{s}_1$ is constrained by the reachable states $\tilde{S}_{\hat{s}_1,l}$ of all lower level sub-FSMs $l < k$ as follows:

$$\tilde{T}_{\hat{s}_1,k} = \bigwedge_{v_i \in \{\hat{V} \cup V_k\}} (v_i' \equiv \delta_i(X,V)) \downarrow \bigwedge_{l<k} \tilde{S}_{\hat{s}_1,l}(V_l). \tag{5.1}$$

Consequently, the *tbt_traversal* algorithm in Figure 5.1 is modified to approximate the set of reachable states of a sub-FSM k. The modified algorithm *tbt_subFSM* being presented in Figure 5.4 takes the transition relation, the set of initial states, the main-FSM, a vector of state variables V_k for a sub-FSM k and the sets of reachable states for sub-FSMs $l < k$ as inputs. It calculates an approximative sets of reachable states for sub-FSM k. The only difference from the *tbt_traversal* algorithm is that the transition relation is constrained in line 8 by the set of reachable states of the lower level sub-FSMs as described above. Furthermore, this constrained transition relation is used in the constrained image computation to calculate the set of next states corresponding to each main-transition.

```
0: procedure approx_tbt(T, S_0, Ŝ, T̂, Ŝ_0) {
1:     (V, E) ← Constrained_Dependency(T, T̂);
2:     ⟨V_1, V_2, ..., V_r⟩ ← SCC(V, E);
3:     for each V_k {
4:       {S̃_ŝ,k} ← tbt_subFSM(T, S_0, Ŝ, T̂, Ŝ_0, V_k, {S̃_ŝ,l | ∀ŝ ∈ Ŝ, ∀l < k});
5:     }
6:     return({S̃_ŝ,l | ∀ŝ ∈ Ŝ, ∀1 ≤ l ≤ r};
7: }
```

Figure 5.5: Approximative TBT traversal algorithm for all sub-FSMs

Procedure *approx_tbt()* as depicted in Figure 5.5 approximates the set of reachable states of the overall design using the procedure *tbt_subFSM*. The procedure takes the transition relation, the initial states, and the main-FSM as its input parameters. First, it calls the procedure *Constrained_Dependency* which constructs the constrained latch dependency graph based on Definition 5.7. The dependency graph is then partitioned into strongly connected components using the standard SCC procedure. Each SCC contains state variables of a sub-FSM. Next, all sub-FSMs are traversed by the procedure *tbt_subFSM* in topological order. Because of the uni-directional communication among sub-FSMs the next-state function of a sub-FSM k only depends on the state variables in sub-FSMs l with *level*$(l) <$ *level*(k). Due to the topological ordering of the sub-FSMs it

is $l < k$. This justifies why, in contrast to the MBM approach in [CHM+96a], we do not repeat the traversal for lower-level sub-FSMs $l < k$ using the reachability constraints $\tilde{S}_k$.

Up to now the general theory of the exact and the approximative version of our TBT traversal algorithm have been discussed. The following sections will present some relevant implementation details for a SAT-based version of the algorithm.

5.3 SAT-based Implementation of TBT

The discussions of the previous subsections revealed that the constrained image and the constrained support set computations are key for an efficient implementation of the (approximative) TBT traversal algorithm. This subsection provides some details on how these computations can be implemented using a SAT solver. Max Thalmaier has implemented the proposed algorithm based on BDDs. The readers are referred to [NTW+08] for details of the BDD-based implementation.

5.3.1 SAT-based Constrained Image Calculation

Within a SAT solver the transition relation as well as the characteristic function for a set of states Z are represented as *CNF*. The same applies for the reachability constraints obtained by the traversal of subordinate sub-FSMs.

First, let us consider the constrained transition relation corresponding to a main-transition relation $(\hat{s}_1, \hat{s}_2)$. This constrained transition relation defined in Definition 5.3 can be represented by the following CNF:

$$\text{CNF}_{\hat{s}_1 \rightarrow \hat{s}_2} = \text{CNF}_T \wedge \bigwedge_{v_i \in \hat{V}} \{l_i\} \wedge \bigwedge_{v'_i \in \hat{V}'} \{l'_i\} \tag{5.2}$$

In the above equation, we have:

- CNF_T is the CNF representing the transition relation T of FSM M
- l_i is a literal representing the value of the main-state variable $v_i \in \hat{V}$ that encodes main-state $\hat{s}_1$. l_i is defined by:

$$l_i = \begin{cases} \overline{v_i} & \textit{if } \hat{s}_1(v_i) \equiv 0 \\ v_i & \textit{if } \hat{s}_1(v_i) \equiv 1 \end{cases} \tag{5.3}$$

- l'_i is a literal representing the value of the next main-state variable $v'_i \in \hat{V}'$ that encodes main-state $\hat{s}_2$. l'_i is defined by:

$$l'_i = \begin{cases} \overline{v'_i} & \textit{if } \hat{s}_2(v_i) \equiv 0 \\ v'_i & \textit{if } \hat{s}_2(v_i) \equiv 1 \end{cases} \tag{5.4}$$

Then, a simple ALL-SAT approach is used to enumerate all possible assignments of the next-states variables. In the current implementation, states are stored in disjunctive normal form (DNF) with each main-state.

Procedure *img* in Figure 5.6 calculates the next-states of the set of states Z corresponding to a main-transition $(\hat{s}_1, \hat{s}_2)$. First, the SAT instance *CNF* is calculated from the transition relation, the main-states $\hat{s}_1, \hat{s}_2$ and the set of states Z. A satisfying assignment A of the SAT instance is then found by the SAT solver. The values of the next-state variables encoding the next-state s' are extracted from A as a partial assignment. State s' is added to the set of next states $Z'_{\hat{s}_2}$. In addition, similarly to the approach in [McM02], a blocking clause $\text{CNF}_{s'}$ preventing the next-state variables from being reassigned is added to the SAT instance. The blocking clause $\text{CNF}_{s'}$ of s' is defined by:

$$\text{CNF}_{s'} = \bigvee_{v'_i \in V'} \overline{l'_{s',i}} \tag{5.5}$$

where

$$l'_{s',i} = \begin{cases} \overline{v'_i} & \textit{if } s'(v'_i) \equiv 0 \\ v'_i & \textit{if } s'(v'_i) \equiv 1 \end{cases} \tag{5.6}$$

The SAT instance is solved again to find another assignment. The procedure finishes when the SAT instance becomes unsatisfiable, i.e., $A \equiv \emptyset$.

```
0:procedure img(ŝ1, ŝ2, Z, T, V) {
1:    Z'_ŝ2 ← ∅;
2:    CNF ← CNF_T ∧ ⋀_{v_i∈V̂} {l_i} ∧ ⋀_{v'_i∈V̂'} {l'_i} ∧ CNF_Z;
2:    A ← SAT_solve(CNF);
3:    while (A ≠ ∅){
5:       s'(V') ← A(V');
6:       Z'_ŝ2 ← Z_ŝ2 ∪ s';
7:       CNF_s' ← ⋁_{v'_i∈V'} ¬l'_{s',i};
8:       CNF ← CNF ∧ CNF_s';
4:       A ← SAT_solve(CNF);
9:    }
10:   return(Z_ŝ2);
11:}
```

Figure 5.6: Constrained image computation

Procedure *img* is illustrated by the following example. Let the main-state variables be v_1, v_2 and the other state variables be v_3, v_4. Consider a main-transition $\hat{s}_1, \hat{s}_2$. In the main-state $\hat{s}_1$ the values of state variables v_1 and v_2 are 0. In the main-state $\hat{s}_2$ the values of next-state variables v'_1 and v'_2 are 0 and 1, respectively. The set of states corresponding to the main-state $\hat{s}_1$ is $S_1(v_3, v_4) = \{00, 10\}$. We want to calculate the set of next-states corresponding to the main-state $\hat{s}_2$. The CNFs are calculated as:

1. $\text{CNF}_{\hat{s}_1 \to \hat{s}_2} = \text{CNF}_T \wedge \{\overline{v_1}\} \wedge \{\overline{v_2}\} \wedge \{\overline{v_1'}\} \wedge \{v_2'\}$.

2. $\text{CNF}_Z = \text{CNF}(\overline{v_3} \wedge \overline{v_4} \vee v_3 \wedge \overline{v_4})$.

Solving the SAT instance we get an assignment of the next-state variable $A(v_3', v_4') = 01$. Therefore, a next-state is $s' = 01$. We add a blocking clause to the SAT instance, $\text{CNF}_{s'} = v_3' \vee \overline{v_4'}$. The SAT instance is given to the solver again. However, the solver proves the instance to be unsatisfiable. Consequently, the set of next-states corresponding to the main-transition $(\hat{s}_1, \hat{s}_2)$ is $S_{1\to 2} = \{01\}$.

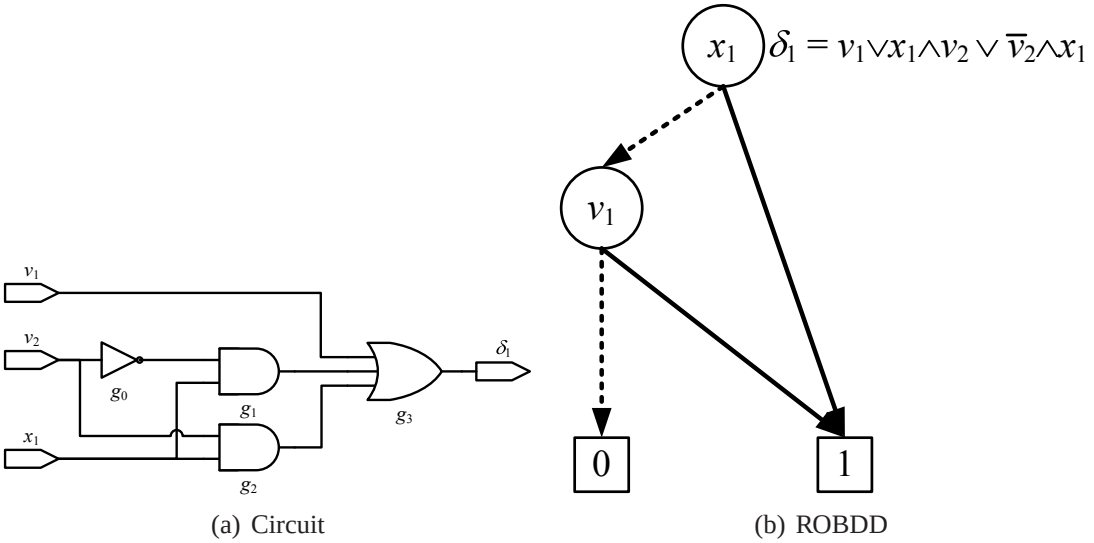
Last but not least, note that the constraints obtained from the main-FSM transition prune the search space such that the enumeration of next states becomes feasible. Moreover, the incremental SAT interface of the underlying solver [ES03] turned out to be beneficial. It should be mentioned that more sophisticated methods for ALL-SAT as proposed in [GGA04] could further improve the efficiency of this enumeration.

5.3.2 SAT-based Constrained Support Set Calculation

In Chapter 2, the support set of a Boolean function has been defined. When a function is represented as a combinational circuit, the support set can be approximated by calculating the *cone of influence* of the output of the circuit. The cone of influence of an output of a combinational circuit, which can be called the *syntactic support set*, is the set of all inputs that are reachable from the output in the transpose graph of the circuit. This set can be determined by a structural traversal of the circuit. Obviously, the cone of influence may include pseudo-dependencies that do not exists. This pseudo-dependencies can be illustrated by an example shown in Figure 5.7(a). In the figure, the combinational circuit representing the function $\delta_1(x_1, (v_1, v_2)) = v_1 \vee x_1 \wedge v_2 \vee \overline{v_2} \wedge x_1$ is shown. Analyzing the combinational circuit, we can see that the cone of influence of δ_1 includes v_2 while the support set of δ_1 does not.

When computing the constrained support set $\mathit{supp}_{\hat{s}_1 \to \hat{s}_2}(v_j)$ the complete, concrete design is considered. Since the syntactic cone of influence of a next-state variable v_j' usually includes most other state variables an exact analysis for calculating $\mathit{supp}_{\hat{s}_1 \to \hat{s}_2}(v_j)$ is prohibitively expensive. Given a main-transition $(\hat{s}_1, \hat{s}_2) \in \hat{T}$, the combinational circuit is used to approximate this constrained support set as illustrated in Figure 5.8.

First, the state values $\hat{s}_1$ and $\hat{s}_2$ are assigned to the main-state variables $\hat{V}$ and the main-next-state variables $\hat{V}'$, respectively. These Boolean constraints are then propagated to identify constant nodes in the circuit network. Next, the support set of δ_j under the constraint of main-transition $(\hat{s}_1, \hat{s}_2)$ is approximated by tracing the circuit structure backwards from the next-state variable v_j' to the primary inputs or the state variables using a modified graph traversal algorithm. This modified procedure traverses the transposed graph of the circuit network. Whenever it reaches a constant node tracing for this path is stopped. All state variables v_i reached during this analysis are added to the constrained support set approximation that is returned by the backtracing procedure.



(a) Circuit (b) ROBDD

Figure 5.7: Cone of influence is different from support set

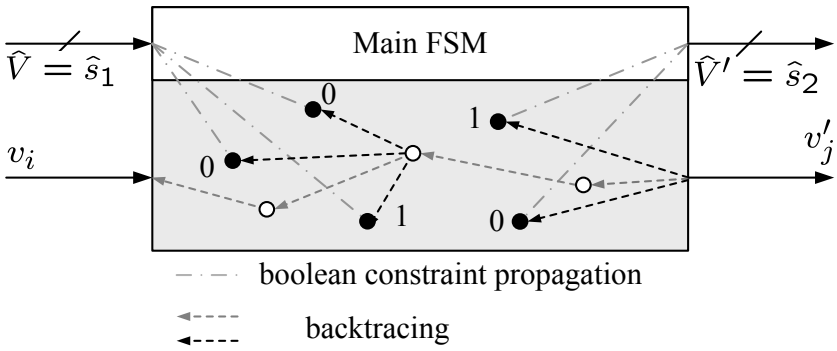


Figure 5.8: Determining the constrained support set

5.4 TBT Traversal in IPC-based Verification Flow

In this section, we will consider integrating the proposed TBT FSM traversal algorithm into the IPC-based verification flow. The TBT algorithm can be used as a preprocessing phase to identify the invariant $\mathcal{I}$ for IPC. The TBT algorithm analyzes the design and its main-FSM to approximate the sets of reachable states corresponding to the main-states. The characteristic functions of these sets are translated into macros in ITL of the IPC-based tool. These macros are then used as additional propositions in the assumption parts of the properties. When the property in Figure 4.3 is proven it is strengthened with the characteristic function representing the set of reachable states corresponding to $\hat{s}_1$, i.e., $\bigwedge_{\forall k} \tilde{S}_{\hat{s}_1,k}$. The strengthened property is formulated in ITL as shown in Figure 5.9. Since both the property and the sets of reachable states correspond to the main-states, only one individual reachability constraint is used in the property. It helps to reduce the complexity of the SAT-instance for the property. Moreover, note that the automatically identified invariants need not to be proven in the properties unlike the manually found invariants.

property strengthened_operation_s_1_to_s_n **is**
  assume:
    at t: $\hat{V} \equiv \hat{s}_1$;
    at t: $\bigwedge_{\forall k} \tilde{S}_{\hat{s}_1,k}$;
    at t: $a_0(X)$;
    at t+1: $a_1(X)$;
    (...);
    at t+n-1: $a_{n-1}(X)$;
  prove:
    at t: $c_0(X,Y)$;
    at t+1: $c_1(X,Y)$;
    (...);
    at t+n-1: $c_{n-1}(X,Y)$;
    at t+n: $\hat{V} \equiv \hat{s}_{n+1}$;
end property;

Figure 5.9: Strengthened property template based on main-states in ITL

In short, the modified IPC-based verification flow is shown in Figure 5.10. The TBT traversal algorithm is added as the preprocessing phase to the methodology flow, which is displayed as grey boxes in the figure. The ultimate goal of the modified flow is to avoid false negatives completely so that time-consuming code inspections of design source code beyond the main-FSM are no longer needed. In other words, the manual process of identifying reachability constraints, which is displayed as dotted boxes in the figure, can be removed from the verification flow.

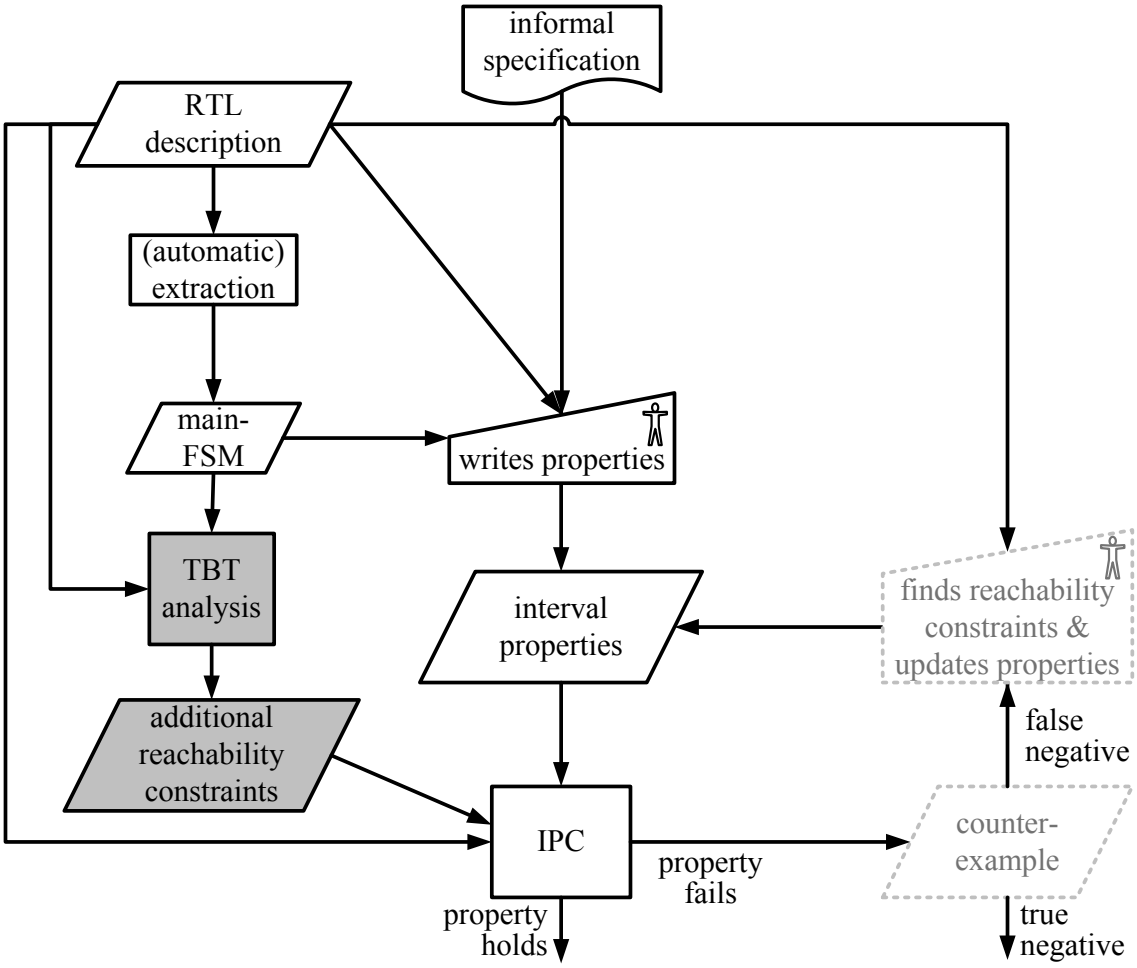


Figure 5.10: IPC-based verification flow with TBT traversal

5.5 Comparison with Previous Work

5.5.1 Comparison with Improvements in Image Computation

Section 3.3.1 describes some methods to decompose the image computation. The proposed *TBT* reachability analysis algorithm uses a special set of state variables, namely *main-state-variables*, as splitting variables. Since the main-FSM controls the other state variables of the design the complexity of the image computation is reduced more effectively. Moreover, the proposed algorithm does not calculate the disjunction in Equation 3.8 explicitly but it decomposes the image computation according to *main-transitions*.

5.5.2 Comparison with Approximative Reachability Analysis Methods

In Section 3.3.2, the previous approximative FSM traversal algorithms have been presented. The approximative algorithm proposed in this thesis explores the *main-FSM* to partition the design into sub-FSMs which are level-ordered *strongly connected components* of a *refined dependency graph* of state variables. This will result in two improvements compared to the previous approximation methods:

1. The approximated sets of reachable states are more exact and more suitable for the intended verification methodology.

2. The proposed algorithm terminates earlier since it does not need to search for a fixed point when it traverses the sub-FSMs.

5.5.3 Comparison with Abstraction Refinement Methods

There are several differences between the proposed methodology in this thesis and the abstraction refinement methodology being described in Section 3.3.3.

First, even though in the proposed methodology, the main-FSM of the system can be understood as an abstract model, the main-FSM is not used to prove the considered properties as in the abstraction refinement methodology. It is due to the fact that the main-FSM is not sufficient as an abstraction to prove the properties. Therefore, the main-FSM is only used to decompose the state space of the concrete model, and the property is verified against the concrete model using IPC.

Second, the abstraction employed in IPC with invariants, in general, cannot be expressed efficiently by a single partitioning of the state set in abstract and concrete state variables. The properties proven in IPC may relate to a large number of state variables. If all these state variables were concretized the resulting model would become prohibitively complex. Moreover, in order to capture the state information implicitly contained in the iterative circuit it would be necessary to concretize specific state variables only at certain time points, but not at others.

Third, since the proposed methodology uses reachability analysis as a processing phase of the IPC-based verification methodology it avoids two expensive computational procedures of abstraction refinement methods:

1. the construction of an abstract model in predicate abstraction and
2. the state separation in counterexample-guided refinement.

Finally, the proposed methodology is only tailored towards RTL property checking of SoC modules and interfaces while abstraction refinement is more generic with the cost of more complex algorithms.

5.5.4 Comparison with BMC-based Methods

In this work, protocol compliance properties are handled by a variant of BMC. In order to prove the properties, the approximated set of reachable states are also used. However, since the properties are formulated based on the main-FSM, the proposed methodology is more effective than the generic methodology in [GGW+03, CNQ04] because of two reasons:

1. IPC only checks simpler SAT-instances than induction based property checking does.
2. The proposed reachability analysis can find more essential reachability constraints.

5.6 Experimental Results

This section will elaborate the evaluation of the proposed verification methodology and algorithms. For this purpose, the approach is applied in a number of verification projects. The characteristics of these projects are reported in Section 5.6.1. The results obtained in these verification projects will be described in Section 5.6.2 and 5.6.3. Furthermore, in Sections 5.6.4 and 5.6.5 a comparison with several symbolic model checking and reachability analysis algorithms as implemented in VIS [BHSV+96] is presented.

5.6.1 Design Characteristics

In order to evaluate the overall methodology experiments were conducted on four industrial design projects and on three public domain designs. The benchmark suite consists of two industrial implementations for a memory controller (denoted by *fcdp1* and *fcdp2*), an industrial AHB master (*ahb-master*) and an industrial bus bridge converting from BVCI to AHB (*bvci2ahb*). The open core designs implement a *Peripheral Component Interconnect (PCI)* target (*pci-target*), an ahb-master (*open-ahb*) and an SDRAM controller (*SDRAM*). Table 5.1 reports design characteristics. Ordered by columns, the number of latches, the size of the main-FSM (number of states/number of transitions) and the number of sub-FSMs are reported.

Design	#latches	#main-states	#main-transitions	#sub-FSMs
fcdp1	498	38	103	460
fcdp2	646	38	103	611
ahb-master	3373	14	54	1117
bvci2ahb	145	10	20	119
pci-target	220	6	24	120
open-ahb	450	9	24	390
SDRAM	110	9	24	78

Table 5.1: Characteristics of experimental designs

5.6.2 Performance Measurements for TBT Traversal

Design	CPU-time		Memory	#SAT var.	#SAT clauses	#blocking clauses	#Image comp.
	partition	traversal					
fcdp1	0:00:05	0:05:59	20	1043 2410 2044	508 2877 1786	4 36955 177	2 275 36
fcdp2	0:00:13	0:16:26	25	1077 4738 2831	570 7038 2933	177 38998 2943	1 265 34
ahb-master	0:00:10	1:05:01	54	680 1720 1226	630 3567 1570	4 229511 458	3 578 13
bvci2ahb	0:00:00	0:13:38	18	338 508 431	221 1050 291	8 327703 3330	4 54 9
pci-target	0:00:00	0:03:42	17	1770 2620 2235	1522 4685 3727	177 9222 880	1 4109 291
open-ahb	0:00:01	0:44:25	19	645 1192 745	707 2058 980	16 165394 1694	8 536 18
SDRAM	0:00:00	0:03:57	17	385 3329 1338	247 5385 1691	1 16280 320	1 11897 227

Table 5.2: Performance of SAT-based implementation of TBT algorithm

In the preprocessing phase of the proposed verification flow, the approximative TBT traversal algorithm was used to analyze the above designs. The performance data on the

SAT-based implementation of the TBT traversal algorithm is shown in Table 5.2.

The table is organized as follows. The table reports CPU times (hh:mm:ss), separated into the time spent in the decomposition and in the traversal phase in columns $2-3$. The peak memory consumption (MB)is reported in column 4. The remaining columns of the table contain machine-independent data for the individual traversals of the sub-FSMs k. The table always reports the minimum, maximum and the average value over all sub-FSM traversals for:

- the number of variables and clauses required to represent the constrained abstract transition relations T_k,
- the number of blocking clauses generated during the traversal of the individual machine M_k,
- the number of image computations

5.6.3 Effect of Generated Reachability Constraints on IPC

The effect of the generated reachability constraints within the verification flow based on IPC is studied in the sequel.

A Case Study with the Flash Memory Controller

As an example, let us focus on the first design (*fcdp1*) in Table 5.1 implementing an AMBA-flavor protocol and representing a module of a telecommunication SoC developed by Infineon. During the design process of this new SoC the proposed verification methodology was evaluated with respect to its productivity.

In order to prove the compliance of the module with the protocol specification and to verify that all functional operations are correctly executed 25 properties were written. The properties specify the behavior of the design during transitions between certain important main-states. Note that, some of the main-states are not used as starting or ending state of a transaction. As explained before, for verifying complex controllers of this kind it is common practice to structure the verification process by following the main-states of a design or using some related notions.

The responsible verification engineer spent about one week to understand the protocol, to analyze the main-FSM and to write the required properties following the property template in Figure 4.3.

Unfortunately, all properties failed because of false negatives. Using the conventional verification methodology, three more weeks were needed to manually inspect the source code of the design in order to identify and prove valid invariants that are sufficient to prove the developed properties. This effort results from the need to not only inspect the main-FSM but also most sub-FSMs. This industrial experience demonstrates that property checking based on Equation 4.12 has its pros and cons. On the one hand, the formulation as an IPC problem makes it possible in the first place to handle industrial designs of this

complexity. On the other hand, identifying the right invariant $\mathcal{I}$ can be quite difficult. This has been measured to account for 75 percent of the verification costs of the flash memory controller.

Main-state	manually found	automatically found
$ilde$	14	63
bm_wr_fifo	12	65
bm_rd_fifo	11	69
bm_rd_start	11	79
nf_wr_fifo	9	79
nf_rd_fifo	8	84
nf_rd_start	9	79
$Total$	74	655

Table 5.3: Number of reachability constraints in main-states found manually and automatically for the flash memory controller

Table 5.3 quantifies the manual effort by counting the reachability constraints valid in some main-states of the flash memory controller. Identifying these reachability constraints manually is a process of trial and error. As explained before, whenever a false negative occurs the verification engineer needs to inspect the source code of the design and to identify reachability constraints to eliminate the false negative. Therefore, the number of manually found reachability constraints in Table 5.3 is proportional to the number of false negatives that have occurred and, thus, to the number of iterations in the non-shaded part of Figure 5.10. In other words, the number of reachability constraints in column 2 of Table 5.3 is proportional to the manual verification effort. By contrast, using the proposed approach reachability constraints are automatically determined in a pre-processing phase.

In order to determine whether the proposed techniques in this paper are sufficient to reduce this effort the shaded part of the flow in Figure 5.10 was evaluated. For this purpose, the manually identified invariants for the flash memory controller (*fcdp1*) were replaced by the reachability constraints generated automatically by TBT traversal as described in Section 5.4. In fact, *all* properties could still be proven without false negatives. This means that it was sufficient to understand the design at a global level and to formulate properties by only inspecting the main-FSM. The tedious process of manually inspecting the source code of the design and the counterexamples (accounting to 3 person weeks in the original flow) could be completely avoided.

As a next step of the evaluation the effect of the reachability constraints on the performance of the property checker is measured. Table 5.4 reports the CPU times and memory usage required by the property checker to prove the set of properties. Within the table the properties are sorted by their starting states. Properties $1-12$ have starting state $idle$. Properties $13-17$ have starting state nf_wr_fifo. Properties $18-19$ have starting states bm_rd_start and bm_rd_fifo. Properties $20-23$ have starting states nf_rd_start and nf_rd_fifo. Properties $24-25$ have starting state bm_wr_fifo. Note that some

properties have two different starting states because they are combined from two different properties. The length in terms of clock cycles and the size in terms of gates of the properties are shown in columns 2 and 3. The CPU time and memory consumption are shown in the other columns. It is apparent that the computational effort varies only little. For the whole property set the CPU time as well as the memory requirement are comparable regardless of whether the manually or the automatically detected constraints are used. Besides the usual variations that SAT techniques show when slightly modifying the problem no serious performance differences could be observed.

Property	Length	Size	CPU time (sec.)		Mem (MB)	
			manual	auto	manual	auto
1	9	13977	0.37	0.38	66	70
2	4	570	0.03	0.01	69	70
3	4	832	0.03	0.58	142	70
4	6	5270	0.08	3.24	99	107
5	4	726	0.02	0.01	99	107
6	10	13269	0.27	0.32	99	107
7	20	54954	12.73	0.22	144	107
8	14	35145	0.81	0.07	145	107
9	23	51846	3.58	0.28	149	107
10	5	2136	0.08	14.07	111	113
11	13	31627	0.75	0.86	141	143
12	4	591	0.03	3.62	141	143
13	14	34181	2.44	0.1	153	143
14	12	22218	0.94	0.76	153	144
15	15	35673	1.03	0.02	217	144
16	19	51168	1.56	1.16	217	144
17	22	36358	1.48	0.86	217	144
18	9	12399	0.46	1.67	67	144
19	11	25424	1.41	3.77	99	164
20	11	18449	0.86	0.02	141	164
21	10	15783	0.6	2.94	141	164
22	16	31527	1.64	0.9	142	164
23	20	44425	3.18	1.26	142	164
24	11	17246	0.3	1.5	99	164
25	11	15195	0.23	1.38	99	164
Total			34.91	40	217	164

Table 5.4: CPU times and memory usages to prove properties of design *fcdp1* using constraints generated manually and automatically

An Example Property for the PCI Target

In order to make the experiments more transparent to the reader, experiments using a public domain PCI target block are represented in this section. The PCI target block was extracted from the PCI local bus in [Azi97]. To avoid the explosion problem of the state space the creator of the benchmark has reduced the width of the address/data bus to 3. In these experiments, the width of the address/data bus was re-expanded to 32 bits. This makes the design more realistic although it is still quite small compared with the industrial designs.

In the following, a specific example based on the PCI design is used to describe how reachability constraints are added to the properties. A property has been written which checks the *turnaround cycle* of the PCI protocol. The property states that if the PCI target is in main-state *idle* and the bus command is *read*, then the next cycle must be the turnaround cycle. This means that the PCI target does not drive the bus lines in this cycle [SIG95]. The PCI target floats the bus lines by setting control signals *OE_TRDY*, *OE_DEVSEL* and *OE_AD* to 0. Consequently, the property is formulated based on the main-states as:

$$\begin{aligned} \mathsf{G}\Big[&(\mathit{main\_state} \equiv \mathit{idle} \wedge \mathit{ReadCmd} \equiv 1) \\ &\rightarrow \mathsf{X}\,(\mathit{OE\_TRDY} \equiv 0 \wedge \mathit{OE\_DEVSEL} \equiv 0 \wedge \mathit{OE\_AD} \equiv 0)\Big] \end{aligned} \tag{5.7}$$

In ITL, the property is written as shown in Figure 5.11.

```
property turn_around is
  assume:
    at t: main_state ≡ idle;
    at t: ReadCmd ≡ 1;
  prove:
    at t+1: OE_TRDY ≡ 0;
    at t+1: OE_DEVSEL ≡ 0;
    at t+1: OE_AD ≡ 0;
end property;
```

Figure 5.11: Property for the turn-around cycle of PCI protocol

The IPC-based property checker fails to prove this property. There are many false counterexamples. One of them is that the values of *OE_TRDY*, *OE_DEVSEL* and *OE_AD* are 1 when the main-state is *idle*. Therefore, to avoid this false negative the following reachability constraint needs to be added to the property.

$$\begin{aligned} &(\mathit{main\_state} \equiv \mathit{idle} \rightarrow \\ &\mathit{OE\_TRDY} \equiv 0 \wedge \mathit{OE\_DEVSEL} \equiv 0 \wedge \mathit{OE\_AD} \equiv 0) \end{aligned} \tag{5.8}$$

Instead of identifying these values manually we performed our TBT algorithm to automatically identify reachability constraints and added them to the properties. Thus, the property is strengthen as shown in Figure 5.12. This modified property can now be proven without any false negative. Also for the PCI block all false negatives could be eliminated using the generated constraints.

property $turn\_around$ **is**
  assume:
    at t: $main\_state \equiv idle$;
    at t: $ReadCmd \equiv 1$;
    at t: $constraints\_at\_idle$;
  prove:
    at t+1: $OE\_TRDY \equiv 0$;
    at t+1: $OE\_DEVSEL \equiv 0$;
    at t+1: $OE\_AD \equiv 0$;
end property;

Figure 5.12: Strengthened property for the turn-around cycle of PCI protocol

Other Designs

To complete the evaluation and to check whether the approach is also viable for a broader range of communication IP blocks, sets of representative properties were developed for all other designs in the benchmark suite. Without constraints the property checker always reports false negatives when using the trivial invariant $\mathcal{I} = 1$. Table 5.5 reports the results for proving the properties with the automatically identified invariants. In this experiment, the invariants were identified by the BDD-based implementation of the TBT traversal [NTW+08]. For each design, the table shows the number of properties in column 2, the lengths of the properties(minimum/maximum) in terms of clock cycles in column 3 and CPU time and peak memory consumption for proving the property set in columns $5, 6$. All experiments confirm the experience with the flash memory controller (*fcdp1*) that the constraints generated by TBT traversal are sufficient to completely avoid false negatives for the respective property sets. Consequently, the verification effort was reduced dramatically.

5.6.4 Comparison with Other Model Checking Techniques Using Approximation or Abstraction

In this section, the results of a comparison between the proposed approach and the large set of model checking techniques implemented in VIS (version 2.1) [BHSV+96] are presented.

Design	number of properties	length of properties	IPC time	IPC peak memory
fcdp1	25	23/4	0:00:39	218
fcdp2	25	4/23	0:00:56	95
ahb-master	7	7/23	0:07:00	1301
bvci2ahb	12	2/8	0:00:16	350
pci-target	4	2/4	0:00:27	59
open-ahb	15	2/7	0:00:01	143
SDRAM	2	8/13	0:00:01	69

Table 5.5: Proving set of properties

As a first step of this comparison a representative property for each design is selected. This property was manually converted to VIS CTL and checked with abstraction and refinement and other model checking techniques provided by VIS. Except for the property on the small designs PCI target and bvci2ahb all the model checking runs were aborted due to memory-out or time-out. The results for this comparison are reported in Table 5.6. In detail, three abstraction/refinement algorithms, traditional model checking algorithms such as *breath first search*, *high density reachability analysis*, and two approximative model checking algorithms are applied to verify the properties. For the algorithms that terminated within a timeout limit of 5 hours and with a memory limit of 2GB, CPU-time and memory requirement are reported. Timeout and out-of-memory are indicated by TO and MO, respectively. Table 5.6 clearly shows that the problem complexity of the verification task is beyond the capacity of the generic methods provided by VIS.

In the table the first column refers to the applied techniques as follows:

- *AR(1), AR(2), AR(3)* denotes the abstraction/refinement algorithms *grab_test* [WLJ+06], *iterative_model_check* [JMH00] and *check_invariant -A3*.
- The *breadth-first-search* algorithm *check_invariant -A0* is identified by the abbreviation *BFS*.
- *HD* is the abbreviation for the VIS implementation of the *High Density Reachability Analysis* algorithm [RS95].
- The model checking algorithms *check_invariant -A2* based on approximative reachability analysis techniques [CHM+96a, CHM+96b, MKSS99] are listed with the same mnemonics as used in the citations under *MBM, RFBF, TFBF, TMBM, LMBL, TLMBM*.
- Finally, the approximative model checking algorithm *approximate_model_check* is denoted as *ACTL mc*.

The presented results clearly show that the generic algorithms in VIS are outperformed by TBT. Even for the far simpler problem of checking the invariant generated by TBT all generic techniques fail for some larger industrial benchmarks.

Design	**fcdp1**		**fcdp2**		**ahb-master**		**bvci2ahb**	
Method	Time (hh:mm:ss)	Mem (MB)	Time (hh:mm:ss)	Mem (MB)	Time (hh:mm:ss)	Mem (MB)	Time (hh:mm:ss)	Mem (MB)
TBT	0:06:04	50	0:16:39	92	1:05:11	92	0:13:38	36
+IPC	$<$ 1sec	76	$<$ 1sec	73	02:25	619	$<$ 1sec	52
AR (1)	-	MO	-	MO	-	MO	-	MO
AR (2)	-	MO	-	MO	TO	-	TO	-
AR (3)	-	MO	-	MO	-	MO	-	MO
BFS	-	MO	-	MO	-	MO	0:09:38	729
HD	-	MO	-	MO	TO	-	TO	-
MBM	-	MO	-	MO	false negative		false negative	
RFBF	-	MO	-	MO	false negative		false negative	
TFBF	-	MO	-	MO	false negative		false negative	
TMBM	-	MO	-	MO	false negative		false negative	
LMBM	-	MO	-	MO	false negative		false negative	
TLMBM	-	MO	-	MO	false negative		false negative	
ACTL mc	-	MO	-	MO	TO	-	TO	-

Design	**pci-target**		**open-ahb**		**SDRAM**	
Method	Time (hh:mm:ss)	Mem (MB)	Time (hh:mm:ss)	Mem (MB)	Time (hh:mm:ss)	Mem (MB)
TBT	0:03:42	55	0:44:26	50	0:03:57	40
+IPC	$<$ 1sec	50	$<$ 1sec	52	$<$ 1sec	48
AR (1)	0:00:06	752	-	MO	-	MO
AR (2)	TO	-	TO	-	0:41:51	75
AR (3)	0:00:04	603	-	MO	-	MO
BFS	-	MO	-	MO	0:00:15	65
HD	-	MO	-	MO	0:00:27	70
MBM	false negative		false negative		false negative	
RFBF	false negative		false negative		false negative	
TFBF	false negative		false negative		false negative	
TMBM	false negative		false negative		false negative	
LMBM	false negative		false negative		false negative	
TLMBM	false negative		false negative		false negative	
ACTL mc	1:03:52	83	TO	-	2:29:02	78

Table 5.6: Comparison techniques implemented in VIS and (SAT-based) TBT/IPC for proving a representative property

5.6.5 Comparison with Other Approximative Reachability Analysis Algorithms

When it comes to determining reachability constraints from scratch rather than proving given candidates for invariants the approximative FSM traversal techniques described in [CHM+96a, CHM+96b, MKSS99] can be considered. In a last experiment, the generic approximative FSM traversal algorithms as implemented in VIS are applied to the flash memory controller (*fcdp1*). It was studied whether the generated constraints are sufficient to prove the properties. The results of the study are reported in Table 5.7.

Main-state	**necessary constraints**	**with VIS sub-FSMs**	**VIS with TBT sub-FSMs**
$ilde$	14	3	12
bm_wr_fifo	12	2	9
bm_rd_fifo	11	3	9
bm_rd_start	11	2	9
nf_wr_fifo	9	2	7
nf_rd_fifo	8	3	6
nf_rd_start	9	3	7
$Total$	74	18	59

Table 5.7: Number of necessary reachability constraints that can be identified by VIS for the flash memory controller

For each main-state of the design, the number of reachability constraints manually detected by the verification engineer is reported in the second column of Table 5.7. A post-processing has been applied to ensure that no constraint can be removed without introducing a false negative for some of the properties.

Column 3 presents how many of these constraints could be generated with the approximative algorithms in VIS using the respective decomposition into sub-FSMs. In order to again simplify the problem for VIS, the sub-FSM decomposition generated by TBT was provided to the VIS algorithms. Column 4 of the table reports how many of the necessary constraints could be found in this experiment.

In both cases VIS generates a number of additional constraints not listed above. However, in both cases even the complete set of constraints generated by VIS was not sufficient to eliminate false negatives when proving the properties for the design. In other words, the approximative FSM traversal algorithms in VIS are generic approximative algorithms and are not able to identify suitable constraints as they are needed to prove the properties.

Chapter 6

Reuse Methodology for Protocol Compliance Verification

This chapter will present a way to increase the productivity of the specification phase in the IPC-based flow based on reuse concepts. In the first section, the previous work on specifying and verifying protocol compliance will be briefly reviewed. In that section, the idea behind the proposed methodology is also introduced. The second section will define a *finite state transition structure (FST)* called *recorder* which is key in our reuse methodology. It also will describe a systematic procedure for designing the recorder as well as for specifying the bus protocol. In Section 6.3, the overall verification methodology is presented. In the last section, the proposed approach is experimentally evaluated.

6.1 Introduction

In communication verification, it is verified that the communication interface of an IP block complies with a standard protocol. In contrast to properties for verifying design-specific functionality, the properties for verifying protocol compliance can, in principle, be formulated independently of the particular design under verification so that they can be re-used for other designs as well. It is therefore beneficial to distinguish between design-specific and protocol-specific verification tasks. However, linking design-independent properties to a design in a way such that usability and tractability of the verification is not compromised is not an easy problem.

This chapter presents a new methodology for formally specifying communication-on-chip bus protocols and for verifying protocol compliance of communication blocks in System-on-Chip (SoC) designs. In this methodology, the bus protocol is specified in a design-independent way by a set of protocol compliance properties based on a generic *recorder* finite state transition system (FST). The proposed methodology clearly differentiates between design-specific and protocol-specific aspects of the overall verification task and exploits the nature of typical SoC protocol specifications and implementations. In this way, the proposed methodology contributes to reaching two important goals: firstly,

it reduces the algorithmic complexity of automatically determining the reachability information needed to avoid false negatives. Secondly, it incorporates re-use concepts to further increase industrial productivity when verifying SoC interfaces.

6.1.1 Related Work

In the past, several approaches for formal property compliance verification have been proposed. Each consists of a style or language for property specification as well as corresponding proof methods. A set of compliance properties can either be specified in a monitor circuit as in [SDJ00, YHY+05, HCY03] or in a standard property specification language such as PSL [Acc04]. Besides these formal approaches there also exists a wealth of non-formal verification IPs for checking protocol compliance by simulation.

A monitor circuit is implemented in a hardware description language (HDL) and is added to the system bus in oder to observe the bus behavior. When a bus participant violates the protocol specification the monitor will trigger an error output that corresponds to the failing participant. Protocol compliance can be verified by using a standard model checker to verify that the error output is never triggered. Since the monitor is a synthesizable design written in an HDL it can be easily applied in practice. Monitor style specification has been used successfully to verify the compliance of PCI protocol implementations as well as an Intel Itanium Processor Bus Protocol.

In [SDJ00] the monitor consists of a set of properties in an *implication style* of the form $past\_condition \rightarrow current\_state$, where $past\_condition$ is the set of constant values of the bus signals in the past. The set of properties is extracted from the rules in the protocol specification. This type of properties covers local, unconnected behaviors of the protocol. Although this monitor style can be used to check some compliance aspects of the protocol it cannot describe the complete behavior of the protocol because it does not store enough "history" of the bus behavior. A complex protocol such as AMBA AHB in which single transactions may include long sequences on the bus is difficult to specify using this approach.

In [YHY+05], a specification FSM is used as the monitor to check the protocol compliance of the design. The specification FSM includes two specific states *dc* and *vio*. When the design under verification violates the protocol specification the specification FSM goes to state *vio*. When an input sequence which is not supported by the device under verification (DUV) appears the specification FSM goes to state *dc*. The compliance of the design with the specification FSM is checked using a *branch-and-bound* algorithm. The algorithm calculates all possible pairs of design states and specification states and checks that the states *dc* and *vio* are not contained in any state pair. Due to complexity the branch-and-bound algorithm can be applied only to FSMs with a small number of states.

The protocol can be also specified in *generalized symbolic trajectory evaluation* (GSTE) *assertion graphs*. The assertion graph can be used to automatically generate a monitor as proposed in [HCY03]. The experiments in [HCY03] are promising, however, they also show that the generated monitors can become very large. The resulting verification model may exceed the capacity of the formal checker.

Recently, several standards for property specification languages have been proposed. These languages can be used to specify the protocol specification as assertions in the design. The design assertions can be verified dynamically by simulation or by static analysis methods. As an example, the partial specification of the AMBA AHB protocol is described in [Dea03].

The set of properties written in standard property languages can also be translated into a monitor. In [MAB06], the properties specified in PSL are split into primitive parts. These parts are translated into primitive digital components and then combined into a monitor. Then, a theorem prover is used to prove that the monitor never asserts its error output when connected to the DUV.

Although many different protocol compliance verification methodologies have been proposed, to the author's best knowledge, there are no systematic design principles to construct a standard set of properties, neither in property languages nor in monitors, that can completely capture the protocol behavior and still be generic and independent of a specific design.

6.1.2 Contribution of this Chapter

This chapter proposes a novel protocol compliance verification methodology which uses a *recorder* to remember previous bus states. The recorder is very similar to a monitor in the sense that it watches the behavior of the bus signals. However, it doesn't check the correctness of the bus behavior (it does not have a monitor output and no error states). It is merely used to formulate a set of compliance properties. It can be combined easily with IPC and is constructed in such a way that only little manual intervention is needed to compute invariants of sufficient strength without a full-blown reachability analysis.

The recorder is specified in an HDL to record important states of the bus. It is connected with the design using only protocol-specific signals. Additionally, a set of compliance properties is formulated in ITL. The properties are based on the recorder which is independent of the design implementation. Both, the recorder and the set of compliance properties only need to be developed once for a particular bus protocol and can be re-used on any design implementing the same protocol. As opposed to previous works, monitors are not used to check the protocol compliance but the recorder is used as a basis and guidance for formulating implementation-independent properties. Splitting the specification into a recorder and a property set allows for an intuitive way of describing a protocol behavior. Moreover, it provides a natural partitioning of the verification problem that can be effectively exploited by the proof algorithms and which leads us to a simplified reachability analysis for IPC. In this way, all reachability information needed to cover long-term dependencies in the control behavior of the module can be obtained. And hence, complex properties related to the data path of the communication module can be proven.

6.2 Recorder-FST and ITL Properties

In protocol specification documents such as that for the AMBA bus [ARM99], the concept of *bus states* is often used. A bus state represents the history of the bus behavior. It represents sufficient information to determine all possible current behaviors. For example, in the AHB bus protocol, a new transaction is started with bus signal *HTRANS*=2 only if the bus state is *IDLE*. In another example, a burst transaction is continued with bus signal *HTRANS*=3 only if the bus state is *"in-the-middle-of-a-burst-transaction"*. Consequently, the bus behavior can often be described in the format *"if the bus state is IDLE, a data or IDLE transaction can be started with HTRANS=2 or HTRANS=0, respectively"*. The bus states, therefore, are essential to describe the bus behavior. The bus states are usually identified and specified before properties for the bus protocol are written. However, it is hard to specify bus states using only bus signals because a bus state represents information on the history of the bus behavior. In order to specify that the bus is in bus state *IDLE*, all possible prior sequences of the signal values on the bus leading the bus to state *IDLE* need to be specified. A recorder can be used to store bus history in *recorder states*.

In order to specify the bus states, not all bus signals need to be considered. The bus states can be expressed in a subset of the bus control signals. Usually, the other signals are needed to communicate data. Their behavior is determined by the control signals of the bus. For example, in the case of the AHB bus, the signals *HTRANS*, *HGRANT*, *HRESP*, *HREADY* and *HBURST* are sufficient to determine the bus states. The other signals such as *HSIZE*, *HWRITE*, *HADDR*, etc., do not need to be considered in the recorder.

Consequently, the formal specification of the bus protocol can be partitioned into two parts:

1. The recorder, for identification of the bus states (also called *recorder states* in the sequel) based on the history of the bus signals;

2. The set of properties, checking the protocol-compliant behavior of the design with respect to possible transitions between bus states and the absence of unexpected transitions. This includes both control-related and data-related behavior.

In the following, the concept of the recorder will be defined and elaborated using an example of the AHB protocol. Then, a systematic (manual) synthesis procedure for such a recorder will be described. The procedure is an intuitive, top-down approach that constructs first a bus state-related "skeleton" in the form of a non-deterministic FST. The non-deterministic FST is then extended using auxiliary FSTs to yield the final recorder.

6.2.1 Recorder-FST

Consider a bus with a set of control signals $\hat{X} = \{x_1, \ldots, x_n\}$. These control signals have a finite Boolean or natural domain and encode an alphabet $\Sigma = \{\sigma_0, \ldots, \sigma_m\}$ of m *bus symbols*. The control-related bus behavior can be considered as a *bus language* $\mathcal{L} \subset \Sigma^*$. The recorder, which is related to the bus language, is defined as follows:

Definition 6.1. *(**Recorder FST**) Given the bus with set of bus symbols Σ, the recorder of a bus is a labeled, incompletely specified, finite state transition structure (FST) $(\Sigma, S, s_0, \delta, L)$, where*

1. *Σ is the set of input symbols (bus symbols),*
2. *S is the set of recorder states,*
3. *s_0 is the initial state,*
4. *$\delta : S \times \Sigma \mapsto S$ is the incompletely specified transition function,*
5. *$L : S \mapsto 2^{\Sigma}$ is the labeling function.*

The labeling function assigns to each state $s \in S$ the input symbols $L(s)$ that can occur in the bus according to the specification when the recorder is in state s. The transition function $\delta(s, \sigma)$ is defined for every state s and for every input symbol $\sigma \in L(s)$ possible in that state s, and undefined otherwise. □

Note that the recorder has only a single initial state. The synthesis procedure starts with a set of bus symbols corresponding to an initial *idle* condition of the bus. This set labels the starting state of the recorder.

Definition 6.1 can be illustrated with the AHB protocol as an example. In this protocol, there are some control signals *HTRANS*, *HGRANT*, *HRESP*, *HBURST*, that encode the alphabet Σ of bus symbols. Tables 6.1 and 6.2 describe a recorder for the AHB protocol as it will be constructed by the proposed synthesis procedure introduced later in this section. To keep things simple, a small part of the recorder – the one concerned with the *INCR4* burst transaction is presented.

state s_i	**label** $\Sigma_i = L(s_i)$	**comment**
s_0	$\Sigma_0 = (\mathit{htrans} \equiv 0 \vee \mathit{htrans} \equiv 2) \wedge (\mathit{hresp} \equiv 0)$	bus is idle
s_1	$\Sigma_1 = (\mathit{htrans} \equiv 3 \vee \mathit{htrans} \equiv 1) \wedge (\mathit{hburst} \equiv 3)$	in first phase of an *INCR4* transaction
s_2	$\Sigma_2 = (\mathit{htrans} \equiv 3 \vee \mathit{htrans} \equiv 1) \wedge (\mathit{hburst} \equiv 3)$	in second phase of an *INCR4* transaction
s_3	$\Sigma_3 = (\mathit{htrans} \equiv 3 \vee \mathit{htrans} \equiv 1) \wedge (\mathit{hburst} \equiv 3)$	in third phase of an *INCR4* transaction
s_4	$\Sigma_4 = (\mathit{htrans} \equiv 0 \vee \mathit{htrans} \equiv 2)$	in fourth phase of an *INCR4* transaction
...	...	...

Table 6.1: Example recorder states for AHB protocol

Table 6.1 shows some states of the recorder and their labels. Since the set of bus symbols, Σ, can be quite large sets of such symbols are represented by characteristic functions, i.e., predicates on bus control variables that evaluate to true iff a variable assignment encodes an element from the set. For simplicity, set and characteristic function notations are used interchangeably. The meaning is always obvious from the context. In Table 6.1, the label $L(s_i)$ of each state s_i is given by the characteristic Boolean function Σ_i. As stated above, the label of a state is the set of those bus symbols that are expected to occur in that state. For example, when the recorder is in state s_0 (the bus is idle), there are two possible transactions: Either a *NONSEQ*, i.e., *burst*, transaction can be started with *HTRANS*=2, or the *IDLE* transaction is continued with *HTRANS*=0. In both cases, the slave is expected to respond with *OKAY*, i.e., *HRESP*=0. The other bus signals may have arbitrary values. Hence, the set of symbols that may appear on the bus when the recorder is in its initial state s_0 is represented by the function $\Sigma_0 = (\mathit{htrans} \equiv 0 \vee \mathit{htrans} \equiv 2) \wedge (\mathit{hresp} \equiv 0)$. When the recorder is in any one of the states s_1, s_2, s_3, the *INCR4* transaction is continued with *HTRANS*=3 (i.e., *SEQ*), or *HTRANS*=1 (i.e., *BUSY*). Therefore, the three different states s_1, s_2, s_3 are all labeled with the same set of bus symbols as characterized by $\Sigma_1 = \Sigma_2 = \Sigma_3 = (\mathit{htrans} \equiv 3 \vee \mathit{htrans} \equiv 1) \wedge (\mathit{hburst} \equiv 3)$.

current state s_i	**triggering condition** $\Sigma_{i,j} \subseteq \Sigma_i$	**next state** s_j
s_0	$\Sigma_{0,0} = (\mathit{htrans} \equiv 0) \wedge \mathit{hgrant} \wedge (\mathit{hresp} \equiv 0)$	s_0
	$\Sigma_{0,1} = (\mathit{htrans} \equiv 2) \wedge \mathit{hgrant} \wedge (\mathit{hresp} \equiv 0) \wedge (\mathit{hburst} \equiv 3)$	s_1
	...	...
s_1	$\Sigma_{1,2} = (\mathit{htrans} \equiv 3) \wedge \mathit{hgrant} \wedge (\mathit{hresp} \equiv 0) \wedge (\mathit{hburst} \equiv 3)$	s_2
	...	...
s_2	$\Sigma_{2,3} = (\mathit{htrans} \equiv 3) \wedge \mathit{hgrant} \wedge (\mathit{hresp} \equiv 0) \wedge (\mathit{hburst} \equiv 3)$	s_3
	...	...
s_3	$\Sigma_{3,4} = (\mathit{htrans} \equiv 3) \wedge \mathit{hgrant} \wedge (\mathit{hresp} \equiv 0) \wedge (\mathit{hburst} \equiv 3)$	s_4
	...	...
s_4	$\Sigma_{4,0} = (\mathit{htrans} \equiv 0) \wedge \mathit{hgrant} \wedge (\mathit{hresp} \equiv 0)$	s_0
	$\Sigma_{4,1} = (\mathit{htrans} \equiv 2) \wedge \mathit{hgrant} \wedge (\mathit{hresp} \equiv 0) \wedge (\mathit{hburst} \equiv 3)$	s_1
	...	...

Table 6.2: Example state transition table for AHB protocol

Table 6.2 presents the state transition table of the recorder. The middle column shows the triggering conditions under which the recorder moves from a current state s_i to a next state s_j. For example, from state s_0, the recorder can go to state s_1 when an *INCR4* burst is started and the master is still granted, i.e, when *HBURST*=3 (i.e., *INCR4*) and *HGRANT*=1. The triggering condition, $\Sigma_{0,1} = (\mathit{htrans} \equiv 3) \wedge \mathit{hgrant} \wedge (\mathit{hresp} \equiv$

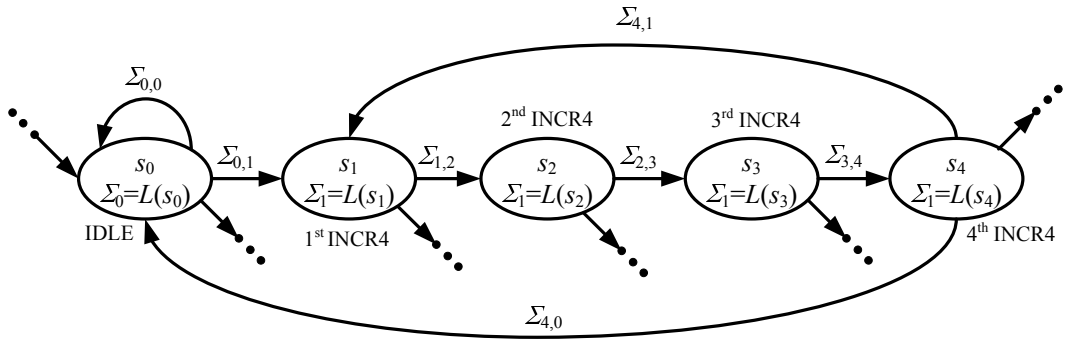


Figure 6.1: Extracted part of the recorder for AHB protocol

$0) \wedge (\mathit{hburst} \equiv 3)$, represents the set of symbols that make the recorder transition from s_0 to s_1. Figure 6.1 shows the corresponding (partial) state transition diagram of the recorder consisting of the recorder states in Table 6.1. In the figure, the recorder states are labeled with the set of symbols $L(s_i)$ and the state transitions are marked with triggering conditions $\Sigma_{i,j}$.

Lemma 6.1. *For any possible word $w = \sigma_0\sigma_1 \ldots \sigma_t \in \mathcal{L}$, there exists a sequence of recorder states $(s_0, \ldots, s_t)$ such that $\delta(s_i, \sigma_i) = s_{i+1}$ for $0 \leq i < t$.*

Proof. Lemma 6.1 is proven by an inductive proof as follows:

1. Induction base: Consider a word $w = \sigma_0$. By the definition of the labeling function, $\sigma_0 \in L(s_0)$. Thus, the transition function is defined, i.e.,

   $$\exists s_1 \in S : \delta(s_0, \sigma_0) = s_1$$

   Therefore, the word w corresponds to the state sequence (s_0, s_1).

2. Induction step: Consider a word $w = \sigma_0\sigma_1 \ldots \sigma_{k-1}$ being recognized by the recorder. It means that there exists a sequence of states $s_0, s_1, \ldots, s_k$ such that $s_{i+1} = \delta(s_i, \sigma_i)$ for $0 \leq i \leq k-1$. Let σ_k be a bus symbol that can occur when the recorder is in state s_k. Similar as for the base step, $\sigma_k \in L(s_k)$, thus $\delta(s_k, \sigma_k) = s_{k+1}$. Therefore, the word $w' = w\sigma_k$ corresponds to sequence of states $(s_0, s_1, \ldots, s_k, s_{k+1})$.

□

Lemma 6.1 states that for every word of the bus language, there exists a corresponding path in the STG of the recorder FST. Note that forbidden words (words not in $\mathcal{L}$) are not handled by the recorder. It is not the task of the recorder to recognize forbidden words, it only needs to make defined transitions for correct words. The ITL properties being represented in Section 6.2.2 are used to check that only allowed symbols appear on the bus.

The labeling function plays an important role in building the state transition graph (STG) of the recorder. Basically, the set Σ_s labeling a recorder state s describes a certain "situation" on the bus that may occur in several different states. The labeling alone is not sufficient to describe the state of the bus, since two different recorder states (bus states) s_1, s_2 can be labeled by the same set of possible input symbols $\Sigma_{s_1} = \Sigma_{s_2}$. Obviously, however, such states occur in a similar context. We can partition the state transition graph of the recorder by the labels. All recorder states with the same label are in a common block of the partition.

This idea is used for a systematic synthesis of the recorder which consists of two steps.

- In a first phase, only the individual "situations" that the bus can be in are considered. In other words, sets of allowed bus symbols that will be used as labels in our recorder are identified. In addition, the possible transitions between the labels are determined. This leads to a non-deterministic FST which controls the overall behavior of the recorder. This FST is called the *main-FST* of the recorder.
- In a second step, we add details in the form of additional sub-FSTs communicating with the main-FST to remove the non-determinism. This step yields the recorder.

The main-FST and its recorder are constructed such that the main-FST satisfies the following definition:

Definition 6.2. ***(Recorder Main-FST)*** *The* main-FST *of a recorder* $(\Sigma, S, s_0, \delta, L)$ *is an incompletely specified non-deterministic FST* $(\Sigma, \hat{S}, \hat{s}_0, \hat{\delta})$*, where*

- Σ *is the set of input symbols (bus symbols),*
- $\hat{S} = \{\Sigma_s \in 2^{\Sigma} | \exists s \in S : L(s) = \Sigma_s\}$ *is the set of main-states,*
- $\hat{s}_0 = \Sigma_0$ *is the initial main-state,*
- $\hat{\delta} : \hat{S} \times \Sigma \mapsto 2^{\hat{S}}$ *is the incompletely specified main transition function.*

A main-state is a set of symbols that is the label of at least one recorder state. The main transition function $\hat{\delta}$ is incompletely specified. It is defined for every state $\hat{s}$ and for every symbol $\sigma \in \hat{s}$ allowed in that state. More specifically, the main transition function is related to the recorder in the following way:

$$\hat{\delta}(\hat{s}_i, \sigma) = \{\hat{s}_j \in \hat{S} | \exists s, s' \in S : L(s) = \hat{s}_i \wedge L(s') = \hat{s}_j \wedge \delta(s, \sigma_k) = s'\}$$

□

Figure 6.2 shows the labeled state transition graph of the main-FST for the recorder in Figure 6.1. In the figure, the three main-states are sets of bus symbols Σ_0,Σ_1 and Σ_2. A main-state may correspond to several recorder states, as is the case in the example: Σ_1 corresponds to recorder states s_1, s_2 and s_3. The edges between pairs of main-states are

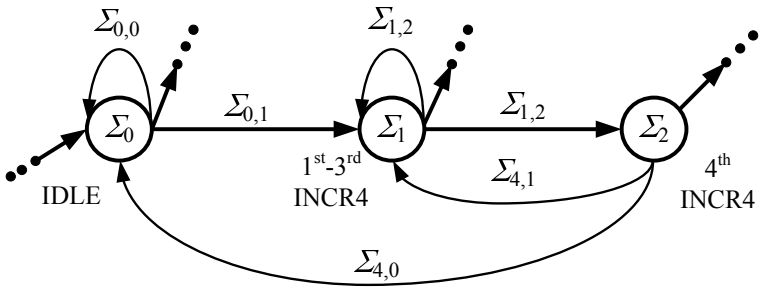


Figure 6.2: Extracted part of the main-FST of the recorder

labeled with the characteristic function of the set of bus symbols triggering the transition between the states. The example also shows non-determinism: two outgoing transitions from main-state Σ_1 are labeled with the same set of triggering bus symbols $\Sigma_{1,2}$. When refining state Σ_1 into recorder states s_1, s_2 and s_3, this non-determinism is removed.

The procedure to construct the main-FST is shown in Figure 6.3. It is a systematic procedure to derive the state transition graph of the main-FST from the protocol specification document. It successively identifies the main-states $\hat{S}$ and the main-transitions. The procedure is illustrated by means of the example shown in Figure 6.2. The starting point of the procedure is the main-state Σ_0 which represents the situation when the bus is idle. For this main-state, all possible next main-states are determined as sets of bus symbols that result from applying every symbol $\sigma \in \Sigma_0$ to the bus. (This is done, e.g., by evaluating the protocol specification document.) In the example, the set Σ_1 is one of the sets found, constituting one new main-state. The transition between the current main-state and a newly-identified next main-state under the input symbol σ is represented by updating the transition function accordingly: $\hat{\delta}(\Sigma_0, \sigma) = \Delta$. This process is repeated for every newly-identified main-state. In the example, applying all input symbols $\sigma \in \Sigma_1$ to the bus leads to two possible next main-states: Σ_1 and Σ_2. The transition function $\hat{\delta}(\Sigma_1, \sigma)$ is updated for every bus symbol accordingly. Afterwards, the procedure is repeated for the new main-state Σ_2.

In the second phase, the main-FST is extended such that the resulting recorder can fully describe the bus behavior. This is done by adding auxiliary sub-FSTs to the system in order to remove non-determinism. The state set of the recorder is encoded by the state variables of the main-FST and the sub-FSTs, i.e.,

$$S \subseteq \hat{S} \times \tilde{S}_1 \times \tilde{S}_2 \times \ldots$$

where the $\tilde{S}_i$ are the state sets of the sub-FSTs.

This step is illustrated using the running example of the AHB bus recorder. After having identified the main-FST as shown in Figure 6.3, we need to add details in order to remove the non-determinism in main-state Σ_1. The STG in Figure 6.2 corresponds to an *INCR4* burst transaction which consists of 4 consecutive data transfers (as shown in

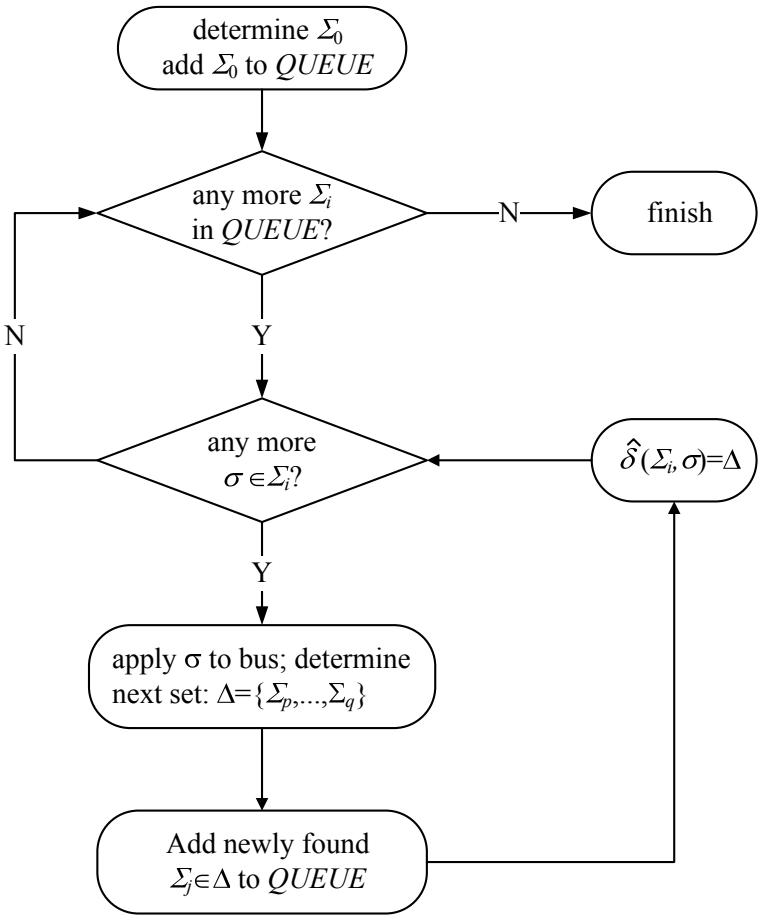


Figure 6.3: Constructing the main-FST of the recorder

Figure 6.1). Of these transfer cycles, 3 are identical with respect to the bus signals. These cycles are represented by Σ_1, which is a label for 3 consecutive recorder states (s_1, s_2 and s_3 in Fig. 6.1). To implement these three states, we add a sub-FST implementing a counter *cnt* with the state set $\tilde{S} = \{1, 2, 3\}$. The state set of the recorder now includes three distinct states $s_1 = (\Sigma_1, 1)$, $s_2 = (\Sigma_1, 2)$ and $s_3 = (\Sigma_1, 3)$, all labeled with the bus symbol set Σ_1.

6.2.2 ITL Properties

This section describes the second step in the process of formally specifying protocol-compliant behavior: formulating properties for the design based on the recorder. In order to construct a set of design-independent properties, only the bus signals and the state variables of the recorder are referred to in the properties. The bus signals are used to specify the correctness of the input/output behavior of the design with respect to the bus. The state variables of the recorder are used to specify the history of the bus signals.

In the following, the control and data bus signals are denoted by $\hat{X}$ and $\bar{X}$. The set of bus signals is the union of both, i.e., $X = \hat{X} \cup \bar{X}$. In addition, V, $\hat{V}$ and $\tilde{V}$ are used to denote the state variables of the recorder, of the main-FST and of the sub-FSTs, respectively.

The properties are constructed following the template as introduced in Chapter 4. They describe:

1. the control-related and data-related behavior of the bus signals and

2. the state sequences of the recorder.

Since the recorder is constructed by extending a main-FST, the latter, obviously, controls the overall behavior of the recorder. The main state variables describe the general context of the bus signals and its states correspond to “important steps” in a bus transaction. Hence, the main states are used to define the starting assumption and ending commitment of the property in Equation 4.6.

Because the recorder does not depend on a concrete implementation, properties can also refer to signals of sub-FSTs of the recorder to further specify the bus transactions. This does not violate the property template.

Suppose that the transaction to be described corresponds to a sequence of recorder main-states $(\hat{s}_1, \ldots, \hat{s}_{n+1})$, the starting assumption and ending commitment can be determined as follows:

$$a_s(V, \hat{X}) = (\hat{V} \equiv \hat{s}_1) \wedge \tilde{a}_s(\tilde{V}) \tag{6.1}$$

In Equation 6.1, the starting assumption is composed from two conditions: $(\hat{V} \equiv \hat{s}_1)$ states that the recorder should start in state $\hat{s}_1$ with starting assumptions about sub-FSTs of the recorder given by $\tilde{a}_s(\tilde{V})$.

Additionally the bus signals $\hat{X}$ should be selected from the set $\Sigma_{1,2}$, triggering a state transition from $\hat{s}_1$ to $\hat{s}_2$ and initiating the bus transaction to be checked by the property. It

implies that the input condition at time point 0 is defined as follows:

$$a_0(X) = \Sigma_{1,2}(\hat{X}) \tag{6.2}$$

For the remaining time points, the bus symbols triggering intermediate state transitions on the path $(\hat{s}_2, \ldots, \hat{s}_{n+1})$ are partially constrained by environment conditions $a_j(\hat{X})$. These conditions are used to constrain the behavior of the other bus participants rather than the design under verification. (Therefore, they are also not a part of the commitment to be checked.)

The commitment c is composed of a series of intermediate commitments $c_j(X)$ and an ending commitment c_e. The intermediate commitments c_j represent the intended behavior of the bus signals. We can distinguish between control-related behavior as represented by the labels Σ_j of recorder states and data-related behavior, $c_j(\bar{X})$, as follows:

$$c_j(X) = \Sigma_j \wedge c_j(\bar{X}) \tag{6.3}$$

Note that it is crucial that the commitments $c_j(X)$ include Σ_j. Remember that the recorder is incompletely specified. Its behavior is left undefined for inputs that do not comply with the protocol specification. The recorder does not check protocol compliance of the design; this needs to be done with the properties by adding the constraints Σ_j to the commitments $c_j(X)$.

Finally, the ending commitment, c_e, is composed from conditions on the ending recorder main-state $\hat{s}_{n+1}$ and ending states of the sub-FSTs of the recorder, as given by $c_e(\tilde{V})$ in Equation. 6.4.

In ITL, the property can be stated as shown in Figure 6.4.

$$c_e(V) = (\hat{V} \equiv \hat{s}_{n+1}) \wedge c_e(\tilde{V}) \tag{6.4}$$

In the *assumption* part, the property specifies that

- at the beginning of the transaction, the main-FST of the recorder is in state $\hat{s}_1$, i.e., the main-state variables take the values of the state code of $\hat{s}_1$ (line 2),
- at the beginning, the sub-FSTs are constrained by $\tilde{a}_s(\tilde{V})$ (line 3),
- at the beginning, the bus control signals $\hat{X}$ initiate the transaction with the triggering condition $\Sigma_{1,2} \subseteq \Sigma_1$ (line 4),
- during the transaction, the bus control signals $\hat{X}$ fulfill the *environment constraints* described by $a_1, \ldots, a_{n-1}$ (lines 5 to 7).

In the *commitment* part, the property requires that

- during the transaction, the bus behavior of the data signals $\bar{X}$ is as specified by the constraints $c_0, \ldots, c_{n-1}$ at time points $t, t+1, \ldots, t+n-1$, respectively (lines 9, 10 and 13),

property protocol_behavior;
1: **assume:**
2: **at** t: $\hat{V} \equiv \hat{s}_1$;
3: **at** t: $\tilde{a}_s(\tilde{V})$;
4: **at** t: $\Sigma_{1,2}(\hat{X})$;
5: **at** t+1: $a_1(\hat{X})$;
6: **at** ...;
7: **at** t+n-1: $a_{n-1}(\hat{X})$;
8: **prove:**
9: **at** t: $c_0(\bar{X})$;
10: **at** t+1: $c_1(\bar{X})$;
11: **at** t+1: Σ_1;
12: **at** ...;
13: **at** t+n-1: $c_{n-1}(\bar{X})$;
14: **at** t+n-1: Σ_{n-1};
15: **at** t+n: $\hat{V} \equiv \hat{s}_{n+1}$;
16: **at** t+n: $c_e(\tilde{V})$;
end property;

Figure 6.4: Protocol compliance property template based on the recorder

- during the transaction, the bus behavior of the control signals $\hat{X}$ is as specified by the constraints $\Sigma_1, \ldots, \Sigma_{n-1}$ at time points $t+1, \ldots, t+n-1$ (lines 11 and 14),
- finally, the main-FST ends up in main-state $\hat{s}_{n+1}$ (line 15), and
- the sub-FSTs fulfill their ending condition $c_e(\tilde{V})$ (line 16).

This property template for protocol compliance is illustrated using the AHB example. Let us formulate a property for one phase in the *INCR4* transaction of the AHB protocol, where we are "right in the middle" of the actual burst transfer. The main-FST of the recorder makes a transition from its main-state $\hat{s}_2$ back to $\hat{s}_2$. This is expressed by the property in Figure 6.5.

Line 3 shows the starting assumption for the states of the sub-FSTs: the counter has not yet reached its final value. The set of bus symbols that trigger the main-transition $\hat{s}_2 \rightarrow \hat{s}_2$ is represented by its characteristic function (line 4). Lines 6 and 7 specify the behavior of a data signal, *HADDR*. (Note that a is a symbolic value linking the expressions of lines 5 and 7; the property is not bound to a specific value of *HADDR*). The intended bus behavior is specified in the commitment part. Under the assumptions, the recorder should end in main-state $\hat{s}_2$ and the value of the counter should have been incremented (lines 9 and 10). The constraint in line 8 checks that when the recorder is in state $\hat{s}_2$, the bus control signals $\hat{X}$ indeed only take values from Σ_2. This check verifies that the recorder is only "fed" with allowed bus symbols when reaching state $\hat{s}_2$.

property protocol_behavior;
1: **assume:**
2: **at** t: $\hat{V} \equiv \hat{s}_2$;
3: **at** t: $cnt < 3$;
4: **at** t: $(\mathit{htrans} \equiv 3) \wedge \mathit{hgrant} \wedge (\mathit{hresp} \equiv 0) \wedge (\mathit{hburst} \equiv 3)$;
5: **prove:**
6: **at** t: $\mathit{haddr} \equiv a$;
7: **at** t+1: $\mathit{haddr} \equiv a + (1 << \mathit{hsize})$;
8: **at** t+1: $(\mathit{htrans} \equiv 3 \vee \mathit{htrans} \equiv 1) \wedge (\mathit{hburst} = 3)$;
9: **at** t+1: $\hat{V} \equiv \hat{s}_2$;
10: **at** t+1: $cnt \equiv cnt + 1$;
end property;

Figure 6.5: Property for middle phase of *INCR4* transaction

Up to this point, the method for specifying bus behavior using the recorder and a set of ITL properties has been presented. By splitting the specification process into these two steps and by defining the recorder formally, the formal specification can be constructed in a systematic way. This helps to reduce the manual effort of formal specification, which is essential in any verification methodology.

6.3 Methodology

In the previous section, the specification model of the bus protocol has been presented. This section proposes a methodology to verify the specification model against designs implementing the protocol.

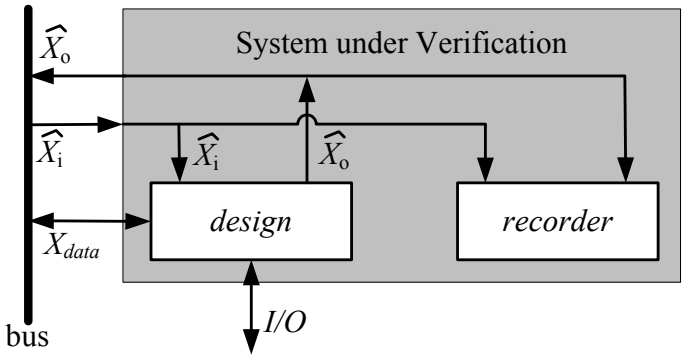


Figure 6.6: Connecting the design with the recorder

First, a combined DUV is created by connecting the recorder with the design implementing the protocol as shown in Figure 6.6. In the figure, the design is connected with

the bus signals X via its bus inputs X_i and its bus outputs X_o, i.e., $X_i \cup X_o = X$. The recorder is connected via only the control signals $\hat{X} = \hat{X}_i \cup \hat{X}_o$, where $\hat{X}_i \in X_i$ and $\hat{X}_o \in X_o$. Note that the recorder only reads the control signals from the bus.

The set of compliance properties is verified against the combined DUV. Many verification techniques are available that may be considered to solve this problem. For example, induction-based BMC [SSS00] and abstraction/refinement [CCK$^+$02] are promising candidates since they are less susceptible to the state explosion problem than conventional symbolic model checking.

However, in the case of industry-strength protocol compliance verification these techniques also may suffer from complexity problems. Let us conceptually analyze the application of these two techniques to prove the example property shown in Figure 6.5. To prove this property abstraction refinement approaches should consider an abstraction model that consists of at least the signals appearing in the property, namely, *Haddr*, *cnt*, $\hat{V}$. Unfortunately, this abstract model is already beyond the ability of model checking techniques in abstraction/refinement approaches. Moreover, in our observation, this abstraction model still needs to be refined to prove the property, thus, further increasing the computational complexity. In *k-induction-based* BMC, an iterative model consisting of k time-frames of the design is analyzed using SAT. However, since a bus transaction is often very long the considered behaviors of the communication modules often span over many cycles. Therefore, k is often large and unknown. This prevents induction-based BMC from proving the property.

As presented in Chapters 4 and 5, the IPC-based verification flow can be used to verify the property set. IPC leads to an efficient computational model that can be handled by a SAT-Solver even for large designs. On the other hand, IPC may fail for communication modules due to long-term state dependencies. Hence, the properties need to be strengthen with invariants to eliminate false negatives.

As mentioned in Chapter 4, a "good" invariant $\mathcal{I}$ is essential for an IPC-based methodology. Obviously, an exact reachability analysis is not feasible, otherwise the properties could be proven using traditional symbolic model checking. In Chapter 5, an invariant $\mathcal{I}$ can often be determined for IPC by traversing many small abstract models of the DUV. However, in the situation considered here, the properties are formulated based on the main-states of the recorder which is constructed in terms of only the bus control signals. Therefore, only reachability constraints related to bus control signals are needed. In the experiments of this work, these reachability constraints can be identified by traversing only a single abstract model consisting of state variables that relate to the bus control signals. This, however, makes the abstract model tractable for symbolic traversal algorithms.

The verification methodology for protocol compliance checking is extended from the proposed IPC verification methodology flow proposed in Chapter 5. It is presented in Figure 6.7. First, the designer or verification engineer constructs the protocol specification consisting of the recorder and a property set in a systematic way as proposed in Section 6.2. Next, the recorder is connected with the communication module to create the extended DUV as shown in Figure 6.6. Symbolic FSM-traversal is then used to identify the reachable states of the abstract model whose latches are listed manually by the veri-

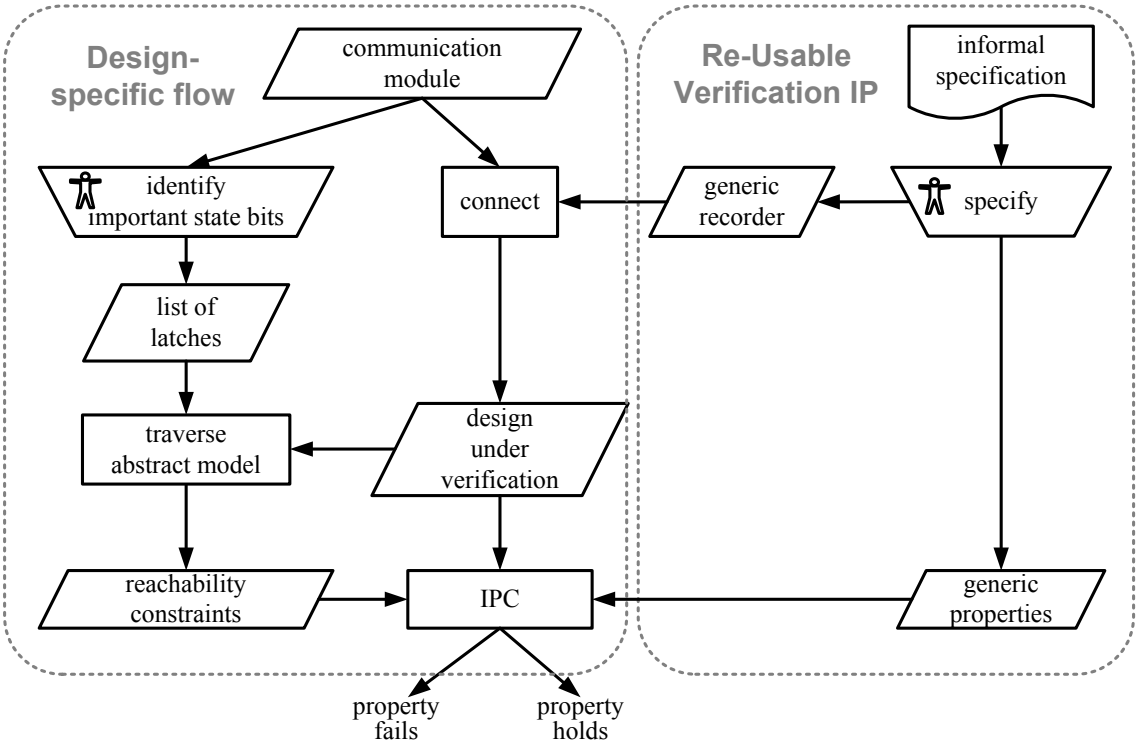


Figure 6.7: Protocol compliance verification methodology

fication engineer. Finally, the set of reachable states in the abstract model is used as an invariant for IPC.

The proposed methodology consists of two manual processes:

1. specification of the recorder circuit and the property set and
2. identification of the latches for the abstract model.

Note that the first task is design-independent. The recorder and the property set are generic and need to be constructed only once for each standard protocol. Moreover, based on the construction procedure proposed in Section 6.2, they can be constructed in an efficient way.

Only the second task is design-specific and needs to be completed for each implementation. In the IPC-based methodology based on operational properties it is standard practice to identify a set of important control states that are used as starting and ending states in the properties. Since the abstract model contains only state variables that relate to the bus control signals only a small portion of the design needs to be inspected by the verification engineer to identify these state variables. Normally, the verification engineer needs to consider design state variables that are:

- in the cone of influence of the bus control signals,
- the important control state variables of the design.

These important design state variables are then combined with the state variables of the recorder to compose the abstract model for reachability analysis.

6.4 Experiments

The proposed methodology was applied to verify three industrial and one public domain AHB-master designs.

6.4.1 Recorders

The recorder is constructed to verify protocol compliance of masters in the AHB protocol as described in Section 6.2. It has been pointed out that it is beneficial to design a recorder that only reads the bus control signals as opposed to designing a monitor which also checks the correctness of transactions and, thus, also relates to the data path of the communication module. By separating the generic protocol specification in a recorder and in a set of properties we preserve the natural distinction between control and data path. In order to measure the effects of distinguishing between control-related and data-related bus behaviors the proposed recorder has been extended to a full monitor, i.e., an error output was added such that the control- and data-related behavior are not only specified but also checked. In the following, $Ctrl$ is used to denote the results from the recorder version and $CtrlData$ is used to denote the results from the monitor version of our DUV.

The size of the recorder and the monitor are reported in Table 6.3. The table shows the number of lines of code, the number of inputs, the number of state variables, the number of main-states and the number of main-transitions for the first version $Ctrl$, and for the second version $CtrlData$ in columns 2 and 3, respectively.

	Ctrl	**CtrlData**
LOCs	583	666
no. of inputs	11	47
no. of latches	15	51
no. of gates	3647	13923
no. of main-states	39	39
no. of main-transitions	163	163

Table 6.3: The size of the recorder for the AHB protocol

The recorder or monitor are connected with the designs to form a DUV. Since we use a recorder and a monitor, 8 different designs under verification are considered.

6.4.2 Properties

The behavior of the bus corresponding to the *idle, single* and the *incrementing* burst transfer is specified in the template presented in section 6.2.2. When the properties are checked

against the DUV, a SAT-instance is constructed. For each property the size of the decision problem resulting from the property and its cone of influence in the DUV is measured. For each property, Table 6.4 shows the property length n in terms of clock cycles, the number of signals and the number of gates (2-input NAND equivalents) for each instance. The results in columns $Ctrl$ refer to the property sets being checked against the recorder, column $CtrlData$ shows the results for the monitor solution. The last row in the table shows the maximum number of signals and gates occurring in the property sets.

Property	**length**	**Ctrl**		**CtrlData**	
		Signals	Gates	Signals	Gates
nogrant_nogrant	2	28	1227	100	1486
nogrant_grant_idle	2	28	2163	100	2431
idle_nohready_idle	2	191	1323	191	1321
idle_idlegrant_idle	2	22	1165	22	1171
idle_idlenogrant_nogrant	2	22	737	22	742
idle_singlegrantokay_idle	10	815	18117	922	65211
idle_singlegrantretry_idle	11	752	19524	895	72691
idle_singlegrantretrynogrant	11	752	18355	895	71457
idle_singlegranterror_idle	10	744	17874	851	64838
idle_singlenograntgrant_idle	11	121	17099	517	70299
idle_singlenograntnogrant	11	121	15935	517	69069
idle_burstgrantokay	10	857	23187	929	81094
idle_burstgrantretrygrant	11	794	24011	938	87106
idle_burstgrantretrynogrant	19	891	38249	1323	151120
idle_burstgranterror	10	825	23208	897	81105
idle_burstnograntokay	19	671	34282	1355	150713
idle_burstnograntretry	19	639	33523	1323	146563
idle_burstnogranterror	11	574	19276	934	74855
trans_burstgrantokay	10	861	23375	933	81040
trans_burstgrantretrygrant	11	798	24155	942	87016
trans_burstgrantretrynogrant	19	895	38541	1327	151177
trans_burstgranterror	10	829	23398	901	81061
trans_burstnograntokay	19	675	34649	1359	150859
trans_burstnograntretry	19	643	33821	1327	146647
trans_burstnogranterror	11	578	19263	938	74607
trans_busygrant	2	253	1413	325	4027
trans_busynograntgrant	10	341	15994	701	69442
trans_nohready_trans	2	290	791	326	1464
Max		895	38541	1359	151177

Table 6.4: The size of the properties for the AHB protocol

6.4.3 Verifying Compliance Based on a Recorder using IPC and Reachability Analysis

Reachability Analysis

The BDD-based implementation of the TBT traversal algorithm [NTW+08] is used to identify reachability constraints. However, since the abstract model is fairly small any conventional reachability analysis algorithm can be used here as well. As described in Section 6.3 the abstract model is created by code inspection identifying a set of important control state bits of the design. They are provided to TBT-traversal as a sub-FSM. The TBT-algorithm then traverses this sub-FSM together with the recorder to identify the reachability constraints. The reachability constraints are used to strengthen the properties as described in Chapter 5.

The results are shown in Table 6.5 which is organized as follows. For each design the results are reported for 2 DUVs:

- the DUV that consists of the design and the recorder (denoted by *+Ctrl*)
- the DUV consisting of the design and the monitor (denoted by *+CtrlData*). The DUV sizes in terms of the number of latches and the number of gates are reported in column 2.

In column TBT the CPU time and memory consumption of TBT are shown when it traverses the abstract model.

Design	no. of latches	no. of gates	TBT		IPC	
			CPU (sec)	Mem (MB)	CPU (sec)	Mem (MB)
fcdp-ahb+Ctrl	661	20366	2069	294	27	149
fcdp-ahb+CtrlData	697	29157	6557	301	36	188
ahb-master+Ctrl	3388	109421	70	136	293	530
ahb-master+CtrlData	3429	119674	102	138	6109	6897
bvci2ahb+Ctrl	161	6838	300	138	256	480
bvci2ahb+CtrlData	197	17239	410	119	609	963
open-ahb+Ctrl	466	15656	578	488	194	489
open-ahb+CtrlData	502	25896	462	390	130	169

Table 6.5: CPU times and memory usages for traversing abstract models and for proving properties

Verifying Properties Using IPC

After the reachability constraints are identified, the IPC tool is used to verify the set of properties. Table 6.5 shows the total CPU-time and the peak memory usages needed by

the IPC tool to prove the set of properties. Column IPC in Table 6.5 shows the total CPU-time and the peak memory usages needed by the IPC tool to prove the set of properties.

As shown in Tables 6.5 the algorithms spent less time and memory for the $Ctrl$ than for the $CtrlData$ recorder in 3 designs. The 4^{th} design is an exception due to a confirmed bug in the design. While the monitor directly reveals this bug several recorder properties are proven to hold before the failing property is processed.

Chapter 7

Summary and Future Work

7.1 Summary

Since an increasing number of functionalities is integrated into present-day SoC designs, SoC designs are usually constructed with a block-based methodology. Consequently, when SoC designs which are specified in RTL are verified it is possible to pursue an overall verification strategy commonly referred to as *correctness by integration*. This means that the overall verification task is divided into local and global sub-tasks. The local sub-tasks consist in the verification of the individual SoC modules. The global task is to verify the global behavior of the entire chip. The global verification problem can be simplified drastically if the individual modules have been verified before-hand. If their correctness is already guaranteed chip-level verification can actually concentrate on the global system behavior and is relieved from hunting bugs in local modules. When verifying the individual SoC modules it is beneficial to distinguish between *computational* modules such as processors and hardware accelerators and *communication* modules which implement the interfaces between these modules. If both, *correctness of computation* and *correctness of communication* are guaranteed by verifying all SoC modules at the RTL it is envisioned that chip-level verification can move from the RTL to the next higher level of abstraction.

In recent years, formal property checking has become adopted successfully in industry and is used increasingly to solve the local verification tasks described in the above scenario. This success results from property checking formulations that are well adapted to specific methodologies. In particular, assertion checking and property checking methodologies based on Bounded Model Checking (BMC) [BCCZ99] or related techniques have matured tremendously during the last decade and are well supported by industrial methodologies. This is particularly true for formal property checking of *computational* SoC modules.

This work is based on a SAT-based formulation of property checking called Interval Property Checking (IPC). IPC originates in the Siemens company and is in industrial use since the mid 1990s [One09]. IPC handles a special type of safety properties, that specify operations in intervals between abstract starting and ending states. This paves the way for

extremely efficient proving procedures.

However, there are still two problems in the IPC-based verification methodology flow that reduce the productivity of the methodology and sometimes hamper adoption of IPC. First, IPC may return false counterexamples since its computational bounded circuit model only captures local reachability information, i.e., long-term dependencies may be missed. If this happens, the properties need to be strengthened with reachability invariants in order to rule out the spurious counterexamples. Identifying strong enough invariants is a laborious manual task. Second, a set of properties needs to be formulated manually for each individual design to be verified. This set, however, isn't re-usable for different designs.

This work exploits special features of communication modules in SoCs to solve these problems and to improve the productivity of the IPC methodology flow.

7.1.1 Approximate TBT FSM Traversal for IPC Verification Flow

The first problem in the IPC methodology flow especially applies to communication modules as their state machines have to cope with long transactions. These transactions may span over long time intervals and impose non-trivial reachability constraints on the state variables of communication modules.

As presented in Chapter 4, properties for a communication module are unbounded safety properties. These properties specify the behavior of the module by relating inputs, internal signals and outputs of the design within a finite number of clock cycles. Thus, they can be proven with a SAT-based property checker starting from an arbitrary state, provided that a sufficient invariant can be calculated. This invariant should contain enough reachability information to eliminate potential false negatives. Furthermore, Chapter 4 illustrates an important observation that communication modules often consist of a central, abstract FSM, namely a main-FSM. The main-FSM controls the overall behavior of the design and is also used to formulate the properties.

For the purpose of identifying reachability information, Chapter 5 presents the approximate FSM traversal algorithm called TBT traversal. This algorithm uses the main-FSM of the design to decompose both, the state space and the transition relation of the design. These decompositions turn out to be the key and make the approach tractable for large industrial designs.

The approach has been used in "real-life" industrial verification projects. The proposed methodology has shown to be very successful in reducing the manual effort for setting up property sets for communication modules.

7.1.2 Re-usable Properties and Generic Recorder

Communication modules often implement a standard bus protocol that is used as the communication infrastructure for SoC designs. Consequently, a set of properties can be constructed independently of the design. A module fulfilling the property set does not violate the protocol standard and can be integrated into the system.

Chapter 6 presented a method to formulate a design-independent set of properties for protocol compliance verification. The property set is formulated based on a generic recorder FST.

The recorder FST remembers the bus status as its states. The recorder is constructed in two phases. In the first phase, a non-deterministic main-FST of the recorder is built using a manual, systematic procedure. This procedure identifies states of the main-FST as bus symbols that can occur in the bus and the transitions among bus symbols. In the second phase, the main-FST is extended by a set of sub-FSTs to remove its non-deterministic transitions. The recorder is, then, connected with a design via bus signals to create a design under verification (DUV).

The properties specify the bus transactions, where each transaction starts from a recorder state to another recorder state. The properties are formulated using the template presented in Chapter 4, where the main states are the states of the recorder. The properties are checked against the DUV using IPC with strengthening invariants that can be identified by traversing a small abstract model of the DUV.

The re-usable recorder and property set are important integral parts of the overall *correctness by integration* verification strategy. The proposed methodology in Chapter 6 provides two benefits:

1. It incorporates a re-use concept to reduce the manual effort to construct the protocol compliance properties in the IPC verification flow. Hence, it helps to increase the productivity of the flow.

2. It simplifies the problem of identifying invariants for IPC.

7.2 Future Work

This thesis has concentrated on improving the productivity of the industrial IPC-based verification flow for communication modules in SoC designs. In the flow, it attempts to replace the manual step of identifying reachability invariants with an approximate FSM traversal algorithm and the manual step of constructing a property set by a generic, re-usable recorder and a set of properties. The following sections outline ideas for future research.

7.2.1 Identifying Reachability Invariants

Identifying Main-FSMs

The main-FSM of a design is the key element of the proposed work. It is used not only in the property template but also in the approximate FSM traversal. In this work, it is supposed to be extracted manually or automatically by analyzing the RTL description of the design syntactically. However the design may have several important state variables, e.g., separate flags, that form its main-FSM. Those main-state-variables are not necessarily

obvious in the RTL description. In this case, the semantic functional dependency among latches of the design can be used to extract the main-FSM. A first attempt [TNW+07] has succeeded in extracting the correct main-FSM for some designs. More attempts should be made to further increase the exactness of the method for other designs.

In addition, properties can be analyzed to identify the important state variables. The starting condition of properties often consist of the important state variables that can be used as the main state variables.

Backward TBT

In the proposed methodology, the approximate TBT traversal is used as preprocessing phase for the IPC checker. It is independent from properties to be checked by the IPC checker. It is interesting to investigate whether TBT can be used with respect to a property to be checked. This means that the TBT traversal is only used by the IPC checker whenever it needs reachability invariants. Moreover, it is worth to examine how to analyze the counterexamples being produced by the IPC checker to identify suitable sub-FSMs that need to be traversed to identify the reachability information.

A first possible idea is to use a backward TBT traversal algorithm. The backward traversal is used to check whether the starting states of the counterexamples are reachable or not. An exact backward FSM traversal is usually infeasible for large designs. However, the first SAT-based implementation of an approximate backward TBT algorithm is fast to determine that a counterexample is unreachable. This is elaborated by the following example.

Let us consider a design and a sequence of main-states $(\hat{s}_1, \hat{s}_2, \ldots \hat{s}_n)$. The design consists of a counter cnt that is set to 0 whenever the main-FSM takes transition $(\hat{s}_1, \hat{s}_2)$. When the main-FSM goes through the state sequences $(\hat{s}_2, \ldots, \hat{s}_n)$, the counter does not change its value. Hence, at main-state $\hat{s}_n$ the counter should be 0. However, when IPC checks the property starting from main-state $\hat{s}_n$, it produces a counterexample. At the starting state of the counterexample, the counter equals to an arbitrary $value \neq 0$. When the counter and the main-FSM are traversed by backward TBT the preimage computation under the constraints of main-transition $(\hat{s}_1, \hat{s}_2)$ will return an empty set of states. Hence, the backward traversal can reach its fixed-point and finish quickly.

In preliminary experiments, unreachable counterexamples are used to update the invariant $\mathcal{I}$. The property is then checked again by the IPC checker. However, this requires a large amount of iterations of the IPC checker and decreases the performance of the IPC checker.

Future work may examine how to identify the sub-FSMs from the results of the backward traversal and use the forward reachability analysis on these sub-FSMs. It may be possible to use proof analysis techniques in [GGYA03]. In the above example, the preimage computation under the constraint of the main-transition $(\hat{s}_1, \hat{s}_2)$ is an unsatisfiable instance. Hence, the proof of this UNSAT instance can be analyzed to identify the UNSAT cores which will consist of the state variables of the counter.

7.2.2 Applying Approximate FSM Traversal in Abstraction Refinement

In [WLJ+06], Wang et.al. proposed to use approximate reachability constraints in an abstraction refinement framework. In this method, the approximate set of reachable states is calculated in a preprocessing phase and then is used as additional constraints in pseudo inputs of the abstract model whenever the property is checked.

Inspired by this idea, it is possible to develop a tighter integration of approximate reachability analysis and abstraction refinement. Future work will examine how to exploit the sub-FSM decomposition based on the abstract FSM to prevent the abstract model from growing too large. The basic idea is illustrated in Figure 7.1. When a list of state variables to be refined is suggested by the refinement procedure, the abstract model is not extended by these state variables. Instead, the list is decomposed into sub-FSMs by the technique presented in Section 5.2.1. Each sub-FSM is traversed together with the abstract model to compute the approximate set of reachable states. This set is then used to constrain the abstract model in the next run of the model checking algorithm. The abstract model is extended only if the approximate set of reachable states can not help to eliminate the spurious counterexample.

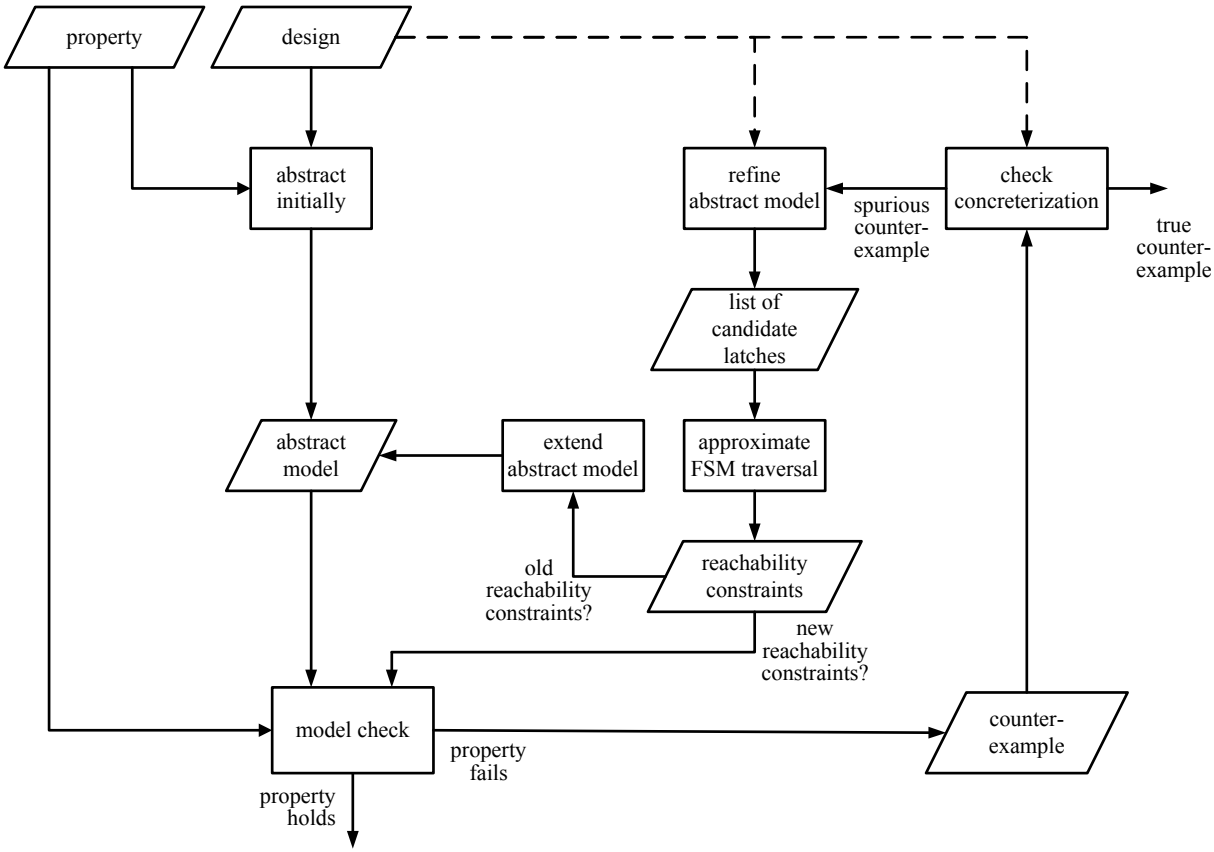


Figure 7.1: Integrating approximate FSM traversal into abstraction refinement

Chapter 8

Zusammenfassung (in Deutsch)

Der Entwurf eines System-on-Chip (SoC) muss heutzutage angesichts der hohen Komplexitäten stark modular und in einer Top-Down-Vorgehensweise erfolgen. Die eigentliche Implementierung der Komponenten wird dabei auf der Register-Transfer-Ebene mittels Hardwarebeschreibungssprachen wie VHDL oder Verilog vollzogen. Die einzelnen Module werden dann zu einem Gesamtsystem zusammengesetzt (*system integration*). Auch bei der Verifikation heutiger Systeme muss ein modularer Ansatz gewählt werden, da eine Simulation des Gesamtsystems entweder nicht mit vertretbarem Zeitaufwand möglich ist oder nicht die gewünschte Qualität liefern kann. Man teilt das Verifikationsproblem daher in lokale und globale Verifikationsaufgaben auf. Die Überprüfung eines Moduls dieses Systems stellt eine lokale Verifikationsaufgabe dar. Die globale Verifikationsaufgabe besteht darin, das Verhalten des aus Modulen zusammengesetzten Systems zu überprüfen. Das globale Verifikationsproblem kann drastisch vereinfacht werden, wenn die Korrektheit der Einzelmodule bereits garantiert wurde. Dann kann sich die Verifikation auf Chip-Ebene auf das Zusammenspiel der Module konzentrieren und wird von der Fehlersuche in den Modulen entlastet.

Es ist für die Lösung des Verifikationsproblems sinnvoll, bei den SoC-Modulen zwischen *Berechnungsmodulen* wie Prozessoren und Hardware-Beschleunigern und *Kommunikationsmodulen*, welche die Schnittstellen zwischen diesen Modulen darstellen, zu unterscheiden. Sind durch die Überprüfung aller SoC-Module auf RT-Ebene sowohl die korrekte Berechnung als auch die korrekte Kommunikation garantiert, ist es sogar vorstellbar, dass die Verifikation auf der Chip-Ebene statt auf der Register-Transfer-Ebene auf der nächst höheren Abstraktionsebene durchgeführt werden kann.

In den letzten Jahren ist die formale Eigenschaftsprüfung erfolgreich in der Industrie eingeführt worden und wird in zunehmendem Maße für die Modulverifikation verwendet. Der Erfolg resultiert zu einem großen Teil aus neuen Eigenschaftssprachen, die auf eine spezifische Methodik zugeschnitten sind. Insbesondere in den vergangenen zehn Jahren haben sich *assertion checking*, die auf Bounded Model Checking (BMC) basierende Eigenschaftsprüfung und ähnliche Techniken durch die Entwicklung neuer Berechnungsmodelle und Algorithmenn außerordentlich weiterentwickelt. Darüberhinaus werden diese Techniken durch auf sie zugeschnittene neue Verifikationsmethodiken unterstützt. Dies

gilt besonders für die formale Eigenschaftsprüfung von SoC-Berechnungsmodulen.

Die vorliegende Arbeit basiert auf einer SAT-basierten Formulierung der Eigenschaftsprüfung, die *Interval Property Checking (IPC)* genannt wird. IPC wurde von der Firma Siemens entwickelt und wird industriell seit Mitte der neunziger Jahre verwendet [One09]. Mit IPC lässt sich eine bestimmte Klasse von Sicherheitseigenschaften beweisen, mit denen sich das Verhalten eines Entwurfs in einem begrenzten Zeitintervall spezifizieren lässt. Dabei wird ausgehend von einem abstrakten Anfangszustand der Übergang zu einem abstrakten Endzustand beschrieben. Die dardurch mögliche Modellbildung erlaubt die Anwendung von extrem leistungsfähigen Beweismethoden.

Auch bei der IPC-basierten Verifikationsmethodik gibt es jedoch noch einige Probleme, die ihre Erfolge schmälern und manchmal die Akzeptanz von IPC behindern. Zwei dieser Probleme werden im Kern dieser Arbeit angegangen.

Zum Einen bedingt das der Eigenschaftsprüfung zugrundeliegende Berechnungsmodell, dass eventuell auftretende Gegenbeispiele nicht echt sind (sogenannte *false negatives* oder *false counterexamples*). Die betrachteten Zustände des Systems sind vom Anfangszustand nicht erreichbar, d.h., das Gegenbeispiel kann in einer richtig initialisierten Schaltung niemals beobachtet werden. Das Auftreten solcher falscher Gegenbeispiele resultiert aus der Beschränkung der zeitlichen Betrachtung auf endliche Zeitintervalle mit beliebigem Anfangszeitpunkt. Diese Betrachtungsweise erfasst nur lokale Erreichbarkeitsinformationen, kann aber langfristige Abhängigkeiten von Signalen übersehen. In der Praxis werden zu beweisende Eigenschaften häufig mit globaler Erreichbarkeitsinformation in Form von Invarianten verstärkt, um das Auftreten von falschen Gegenbeispielen zu vermeiden. Das manuelle Ermitteln solcher Invarianten ist jedoch eine mühsame und zeitraubende Arbeit, die vom Verifikationsingenieur geleistet werden muss.

Zum Anderen ist bereits die Aufgabe, für jeden Entwurf einen Satz spezifischer Eigenschaften aufzustellen, mit großem Aufwand verbunden. Im Allgemeinen ist ein Eigenschaftssatz spezifisch für das Design, für das er erstellt wurde, und nur selten für andere Entwürfe wiederverwendbar. In bestimmten Fällen, (zum Beispiel bei Kommunikationsmodulen), ist eine Wiederverwendung bei entsprechender Aufbereitung der Verifikationsinhalte jedoch denkbar, so dass sich der Aufwand für die Modulverifikation signifikant verringern ließe.

Diese Arbeit stellt sich diesen Problemen. Die vorgestellten Lösungsansätze nutzen dabei die besonderen Beschaffenheiten und Eigenschaften typischer Kommunikationsmodule in Systems-on-Chips aus.

8.1 Approximative TBT FSM-Traversierung für IPC

Das Problem der falschen Gegenbeispiele tritt bei IPC selten bei Berechnungsmodulen, dafür umso häufiger bei Kommunikationsmodulen auf. Dies liegt daran, dass die beteiligten Zustandsmaschinen lange Transaktionen verarbeiten müssen. Solche Transaktionen können sich über große Zeitintervalle erstrecken und es können daher zwischen den Zustandsvariablen der Kommunikationsmodule Beziehungen bestehen, die sich nur durch

nichttriviale Erreichbarkeitsbedingungen ausdrücken lassen.

Wie in Kapitel 4 dargestellt wird, handelt es sich bei IPC-Eigenschaften um *unbeschränkte* Sicherheitseigenschaften, die auf Zeitintervallen formuliert sind. Diese Eigenschaften spezifizieren das Verhalten des Moduls, indem sie die logischen Werte der Eingangssignale und Ausgangssignale und der internen Signale innerhalb einer begrenzten Anzahl von Taktzyklen angeben. So können sie, ausgehend von einem beliebigen Zustand der Schaltung, mit einem SAT-basierten Eigenschaftsprüfer bewiesen werden, sofern eine ausreichende Invariante berechnet werden kann. Diese Invariante sollte genügend Erreichbarkeitsinformation enthalten, um mögliche falsche Gegenbeispiele auszuschließen. Weiterhin wird in Kapitel 4 die wichtige Beobachtung getroffen, dass Kommunikationsmodule häufig eine zentrale, abstrakte Zustandsmaschine aufweisen. Dieser zentrale Automat wird in dieser Arbeit als *main-FSM* bezeichnet. Die main-FSM steuert das Gesamtverhalten des Entwurfs. Darüberhinaus wird sie für Formulierung der Eigenschaften verwendet.

Um Erreichbarkeitsinformation zu identifizieren, stellt Kapitel 5 einen approximativen FSM-Traversierungsalgorithmus vor. Dieser Algorithmus verwendet die main-FSM des Entwurfs, um sowohl den Zustandsraum als auch die Übergangsrelation des Entwurfs zu partitionieren. Daher wird der Algorithmus auch *Transition-by-Transition*-Traversierung (TBT-Traversierung) einer FSM genannt. Die in TBT angewendeten Dekompositionen sind der Schlüssel, um eine geeignete Approximation des erreichbaren Zustandsraums für große industrielle Entwürfe zu ermöglichen.

Die berechneten Approximationen sind erfolgreich zur Verifikation industrieller Schaltungsentwürfe eingesetzt worden. Das vorgeschlagene Verfahren ist geeignet den manuellen Aufwand beim Erstellen von Eigenschaftssätzen für Kommunikationsmodule zu verringern.

Dieser Beitrag der Arbeit zum Verifikationsablauf soll kurz anhand von Abbildung 8.1 erläutert werden. TBT wird als Vorverarbeitungsschritt verwendet, um eine Invariante $\mathcal{I}$ für IPC zu identifizieren. TBT analysiert den Entwurf und seine main-FSM, um Mengen erreichbarer Zustände zu berechnen, die den Zuständen der main-FSM entsprechen. Die charakteristischen Funktionen dieser Mengen werden dann als zusätzliche Vorraussetzung in den Eigenschaften verwendet. Letzten Endes ist es das Ziel des geänderten Ablaufs, falsche Gegenbeispiele vollständig zu vermeiden. Damit beschränkt sich die manuelle Inspektion des Quellcodes eines Entwurfs auf die Identifikation der main-FSM. Das zeitaufwändige manuelle Bestimmen von Erreichbarkeitsbedingungen entfällt – dieser Schritt kann aus dem Verifikationsablauf entfernt werden.

8.2 Wiederverwendbare Eigenschaften und generischer Recorder

Kommunikationsmodule in SoCs implementieren für gewöhnlich ein Standardbusprotokoll, das als Kommunikationsinfrastruktur verwendet wird. Daher kann unabhängig vom konkreten Entwurf ein Satz von Eigenschaften aufgestellt werden, der garantiert, dass ein Modul, das die spezifizierten Eigenschaften erfüllt, den Protokollstandard nicht verletzt.

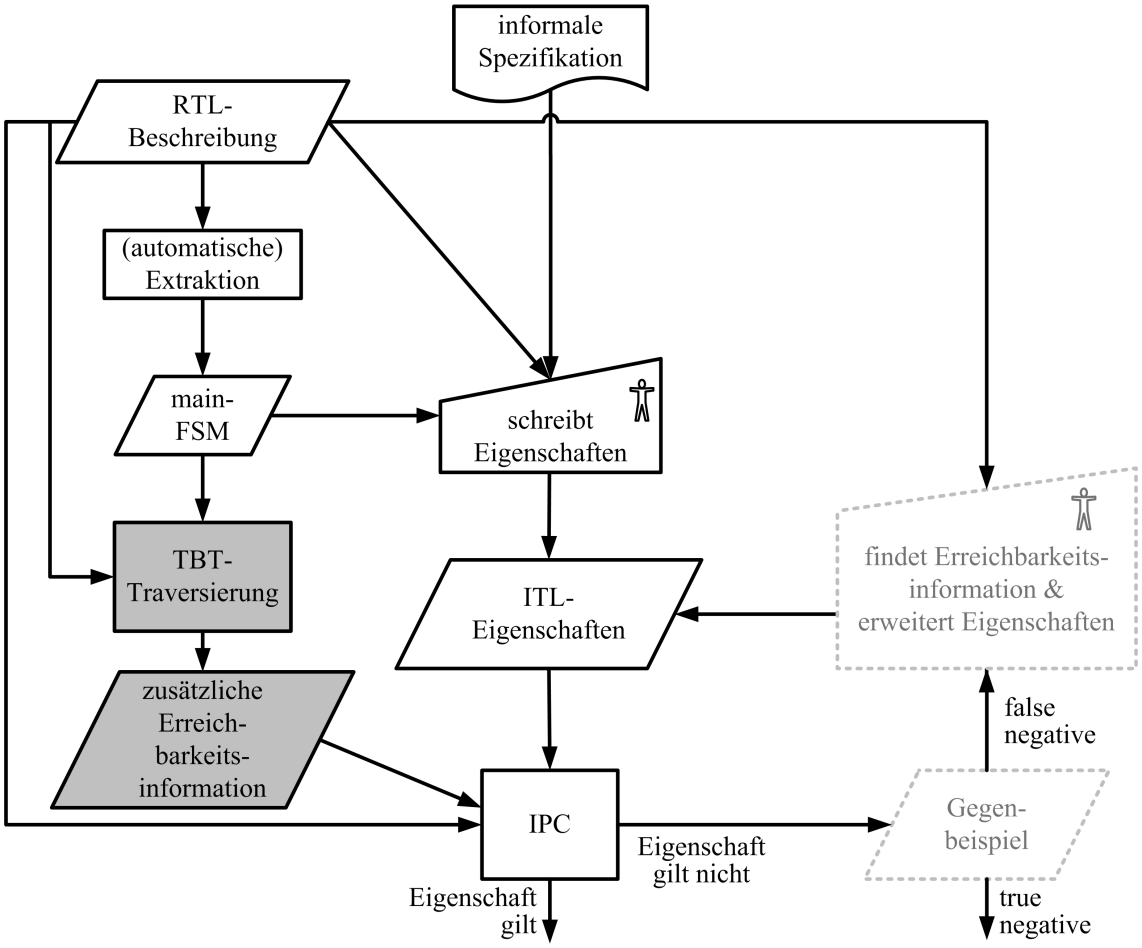


Abbildung 8.1: IPC-basierter Verifikationsablauf mit TBT-Traversierung

Auf diese Weise zertifizierte Module können daher gefahrlos in das System integriert werden – sie halten sich an den Kommunikationsstandard.

In Kapitel 6 wird eine Methodik zur Erstellung von entwurfsunabhängigen Eigenschaftssätzen für die Protokollverifikation dargestellt. Diese Eigentschaftssätze werden jeweils auf Basis eines generischen Halbautomaten, einer sogenannten *Recorder-FST* (FST = *finite state transition structure*) formuliert. Die Aufgabe der Recorder-FST ist nicht, Verletzungen des Busprotokolls festzustellen. Sie beobachtet lediglich während des Betriebs der Schaltung den Bus und stellt in jedem Zeitpunkt den jeweiligen Bus-Zustand fest. Das Soll-Verhalten des Entwurfs wird in IPC-Eigenschaften beschrieben, die sich dann auf den Buszustand beziehen können.

Die Recorder-FST wird in zwei Phasen entworfen:

1. In der ersten Phase wird eine nichtdeterministische main-FST des Recorders unter Verwendung eines manuellen, systematischen Verfahrens gebildet. Dieses Verfahren identifiziert Zustände der main-FST als Bussymbole, die auf den Steuersignalen des Buses übermittelt werden.

2. In der zweiten Phase wird die main-FST um sub-FSTs erweitert, mit denen nichtdeterministische Übergänge entfernt werden. So erhält man einen deterministischen Gesamtautomaten, die Recorder-FST.

Sowohl der Recorder als auch der zu verifizierende Schaltungsentwurf werden mit den Bussignalen verbunden und bilden ein gemeinsames Verifikationsmodell (engl. *design under verification*, *DUV*).

Das Soll-Verhalten dieses DUV während des Übergangs zwischen jeweils zwei Recorderzuständen wird dann mit dazu spezifizierten generischen Eigenschaften überprüft. Diese Eigenschaften werden unter Verwendung eines Templates formuliert, das in Kapitel 4 vorgestellt wird.

Die Eigenschaften werden für das DUV mittels IPC überprüft. IPC wird dabei durch Invarianten unterstützt, die identifiziert werden können, indem man ein kleines abstraktes Modell des DUV traversiert. Der wiederverwendbare Recorder und der dazugehörige generische Eigenschaftssatz bilden einen wichtigen und wesentlichen Beitrag zur gesamten *correctness-by-integration*-Strategie.

Das vorgeschlagene Verfahren in Kapitel 6 hat zwei Vorteile:

1. Aufgrund der Wiederverwendbarkeit verringert sich der mannuelle Aufwand, der zum Aufstellen der Eigenschaften benötigt wird. Dies erhöht die Produktivität der Verifikation erheblich.

2. Die Bestimmung von geeigneten Invarianten für IPC wird automatisiert.

Der in Abbildung 8.2 dargestellte Verifikationsablauf fasst den Beitrag der Arbeit in Kapitel 6 zusammen. Vorab konstruiert der Designer oder der Verifikationsingenieur systematisch die Protokollspezifikation, die aus dem Recorder und einem Eigenschaftssatz

besteht, wie sie in Abschnitt 6.2 vorgeschlagen werden. Anschließend wird der Recorder mit dem Kommunikationsmodul verbunden, um das erweiterte DUV zu erstellen. Symbolische FSM-Traversierung wird verwendet, um die erreichbaren Zustände eines geeigneten abstrakten Modells zu identifizieren, dessen Zustandsvariablen zuvor manuell vom Verifikationsingenieur ermittelt wurden. Schließlich wird diese Menge erreichbarer Zustände des abstrakten Modells als Invariante benutzt, damit IPC die generischen Eigenschaften für das konkrete DUV prüfen kann.

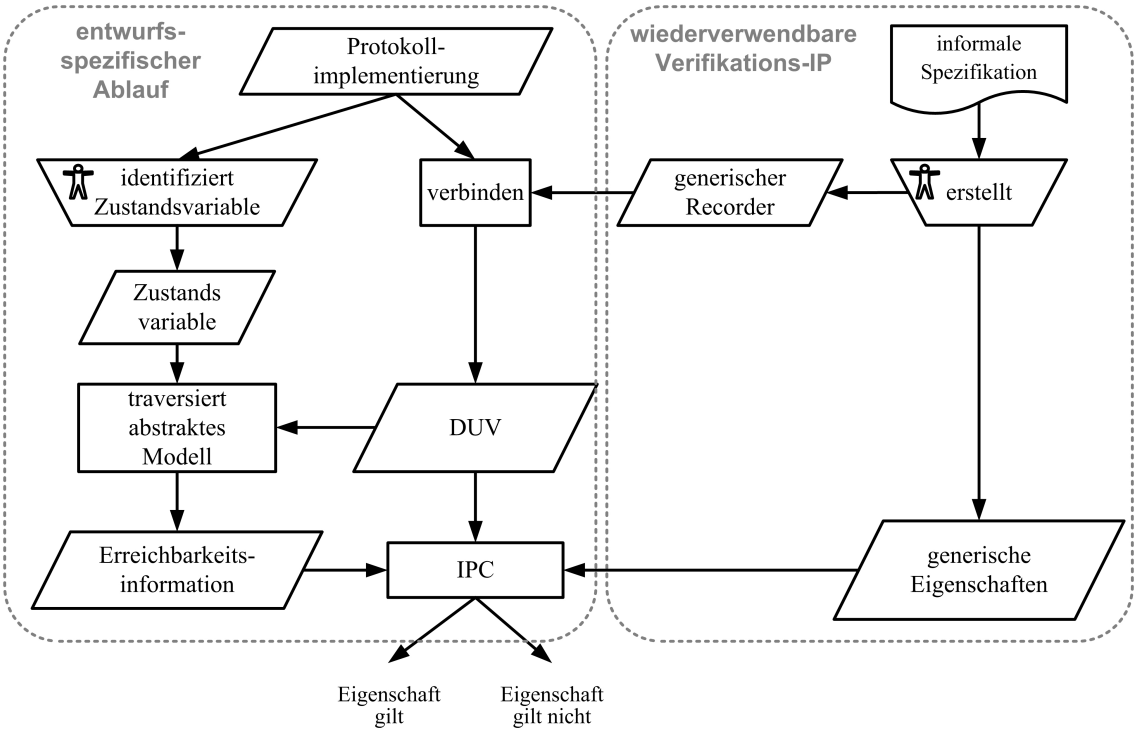


Abbildung 8.2: Methodik zur Verifikation von Protokollimplementierungen

List of Figures

List of Tables

Bibliography

[Acc04] ACCELLERA ORGANIZATION INC.: *Property Specification Language - Reference Manual, version 1.1*. http://www.eda.org/vfv/docs/PSL-v1.1.pdf, June 2004

[ARM99] ARM LIMITED: *AMBA Specification (Rev 2.0)*. http://www.arm.com, 1999

[Azi97] AZIZ, Adnan: *Texas-97 benchmarks*. 1997. – http://www-cad.EECS.Berkeley.EDU/ Respep/Research/Vis/texas-97

[BCCZ99] BIERE, A. ; CIMATTI, A. ; CLARKE, E. ; ZHU, Y.: Symbolic Model Checking Without BDDs. In: *Proc. Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 1999

[BCL91] BURCH, J. R. ; CLARKE, E. M. ; LONG, D. E.: Representing Circuits more Efficiently in Symbolic Model Checking. In: *Proc. ACM/IEEE International Design Automation Conference (DAC)*, ACM Press, 1991. – ISBN 0–89791–395–7, S. 403–407

[BCL+94] BURCH, J. R. ; CLARKE, E. M. ; LONG, D. E. ; MCMILLAN, K. L. ; DILL, D. L.: Symbolic Model Checking for Sequential Circuit Verification. In: *IEEE Transactions on Computer-Aided Design* 13 (1994), April, Nr. 4, S. 401–424

[BHSV+96] BRAYTON, R. K. ; HACHTEL, G. D. ; SANGIOVANNI-VINCENTELLI, A. ; SOMENZI, F. ; AZIZ, A. ; CHENG, S.-T. ; EDWARDS, S. ; KHATRI, S. ; KUKIMOTO, Y. ; PARDO, A. ; QADEER, S. ; RANJAN, R. K. ; SARWARY, S. ; SHIPLE, T. R. ; SWAMY, G. ; VILLA, T.: VIS: A system for Verification and Synthesis. In: ALUR, R. (Hrsg.) ; HENZINGER, T. (Hrsg.): *Proc. International Conference on Computer-Aided Verification (CAV)* Bd. 1102. New Brunswick, NJ : Springer-Verlag, July 1996, S. 428–432

[BJW04] BRINKMANN, R. ; JOHANNSEN, P. ; WINKELMANN, K.: Application of Property Checking and Underlying Techniques. In: DRECHSLER, Rolf (Hrsg.): *Advanced Formal Verification*. Boston, MA, USA : Kluwer Academic Publishers, 2004

[Bry86] BRYANT, Randal E.: Graph-based Algorithms for Boolean Function Manipulation. In: *IEEE Transactions on Computers* 35 (1986), August, Nr. 8, S. 677–691

[CCK+02] CHAUHAN, P. ; CLARKE, E. ; KUKULA, J. ; SAPRA, S. ; VEITH, H. ; WANG, D.: Automated Abstraction Refinement for Model Checking Large State Spaces using SAT based Conflict Analysis. In: *Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2002

[CCLQ97] CABODI, Gianpiero ; CAMURATI, Paolo ; LAVAGNO, Luciano ; QUER, Stefano: Disjunctive Partitioning and Partial Iterative Squaring: an Effective Approach for Symbolic Traversal of Large Circuits. In: *Proc. Annual Conference on Design Automation*, 1997. – ISBN 0–89791–920–3, S. 728–733

[CCQ96] CABODI, Gianpiero ; CAMURATI, Paolo ; QUER, Stefano: Improved Reachability Analysis of Large Finite State Machines. In: *Proc. International Conference on Computer-Aided Design (ICCAD)* IEEE, 1996, S. 354–360

[CE81] CLARKE, E. M. ; EMERSON, E.A.: Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. In: *Lecture Notes in Computer Science* 131 (1981)

[CGJ+00] CLARKE, Edmund M. ; GRUMBERG, Orna ; JHA, Somesh ; LU, Yuan ; VEITH, Helmut: Counterexample-Guided Abstraction Refinement. In: *Proc. International Conference on Computer Aided Verification (CAV)*, 2000, S. 154–169

[CGJ+01] CLARKE, Edmund ; GRUMBERG, Orna ; JHA, Somesh ; LU, Yuan ; VEITH, Helmut: Progress on the State Explosion Problem in Model Checking. In: *Lecture Notes In Computer Science* (2001), S. 176 – 194

[CGKS02] CLARKE, Edmund ; GUPTA, Anubhav ; KUKULA, James ; STRICHMAN, Ofer: SAT based Abstraction-Refinement using ILP and Machine Learning Techniques. In: *Proc. International Conference on Computer Aided Verification (CAV)*, 2002

[CGP99] CLARKE, E. M. ; GRUMBERG, O. ; PELED, D. A.: *Model Checking*. London, England : MIT Press, 1999

[CHJ+90] CHO, Hyunwoo ; HACHTEL, Gary ; JEONG, Seh-Woong ; PLESSIER, Bernard ; SCHWARZ, Eric ; SOMENZI, Fabio: ATPG Aspects of FSM Verification. In: *Proc. International Conference on Computer-Aided Design (ICCAD)*, IEEE, 1990

[CHM+96a] CHO, Hyunwoo ; HACHTEL, G. D. ; MACII, E. ; PLESSIER, B. ; SOMENZI:, F.: Algorithms for Approximate FSM Traversal Based on State Space Decomposition. In: *IEEE Transactions on Computer-Aided Design* 15 (1996), December, Nr. 12, S. 1465–1478

[CHM+96b] CHO, Hyunwoo ; HACHTEL, G. D. ; MACII, E. ; PONCINO, M. ; SOMENZI, F.: Automatic State Space Decomposition for Approximate FSM Traversal Based on Circuit Analysis. In: *IEEE Transactions on Computer-Aided Design* 15 (1996), December, Nr. 12, S. 1451–1464

[CLR94] CORMEN, Thomas H. ; LEISERSON, Charles E. ; RIVEST, Ronald L.: *Introduction to Algorithms*. McGraw-Hill Book Company, 1994

[CM90] COUDERT, Olivier ; MADRE, Jean christophe: A Unified Framework for the Formal Verification of Sequential Circuits. In: *Proc. International Conference on Computer-Aided Design (ICCAD)*, 1990

[CNQ04] CABODI, Gianpiero ; NOCCO, Sergio ; QUER, Stefano: Improving SAT-based Bounded Model Checking by Means of BDD-based Approximate Traversals. In: *Journal of Universal Computer Science* 10 (2004), S. 1693–1730

[Dea03] DEADMAN, Bernard: *PSL/Sugar-Tutorial: Protocol Modeling*. Design and Verification Conference (DVCon), 2003

[DLL62] DAVIS, Martin ; LOGEMANN, George ; LOVELAND, Donald: A Machine Program for Theorem Proving. In: *Communications of the ACM* 5 (1962), S. 394–397

[ECW03] EDMUND CLARKE, Helmut V. ; WANG, Dong: SAT Based Predicate Abstraction for Hardware Verification. In: ENRICO, Giunchiglia (Hrsg.) ; ARMANDO, Tacchella (Hrsg.): *Conference on Theory and Applications of Satisfiability Testing* Bd. 2919, Springer Berlin / Heidelberg, 78-92

[ES03] EEN, N. ; SOERENSSON, N.: An Extensible SAT-solver. In: *Proc. International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2003

[GB94] GEIST, Daniel ; BEER, Ilan: Efficient Model Checking by Automated Ordering of Transition Relation Partitions. In: *Proc. International Conference on Computer Aided Verification (CAV)*, Springer-Verlag, 1994. – ISBN 3–540–58179–0, S. 299–310

[GDHH98] GOVINDARAJU, Shankar G. ; DILL, David L. ; HU, Alan J. ; HOROWITZ, Mark A.: Approximate Reachability with BDDs Using Overlapping Projections. In: *Proc. International Design Automation Conference (DAC)*, 1998, S. 451–456

[GGA04] GANAI, M. K. ; GUPTA, A. ; ASHAR, P.: Efficient SAT-based unbounded symbolic model checking using circuit cofactoring. In: *Proc. International Conference on Computer-Aided Design (ICCAD)*. Washington, DC, USA : IEEE Computer Society, 2004. – ISBN 0–7803–8702–3, S. 510–517

[GGW+03] GUPTA, Aarti ; GANAI, Malay ; WANG, Chao ; YANG, Zijiang ; ASHAR, Pranav: Abstraction and BDDs Complement SAT-based BMC in DiVer. In: *Proc. International Conference on Computer-Aided Design (ICCAD)*, 2003

[GGYA03] GUPTA, Aarti ; GANAI, Malay ; YANG, Zijiang ; ASHAR, Pranav: Iterative Abstraction using SAT-based BMC with Proof Analysis. In: *Proc. International Conference on Computer-Aided Design (ICCAD)*. Washington, DC, USA : IEEE Computer Society, 2003. – ISBN 1–58113–762–1, S. 416

[GS97] GRAF, Susanne ; SAÏDI, Hassen: Construction of Abstract State Graphs with PVS. In: *Proc. International Conference Computer Aided Verification (CAV)* Bd. 1254. Springer-Verlag. – ISBN 3–540–63166–6, 72–83

[HCY03] HU, Alan J. ; CASAS, Jeremy ; YANG, Jin: Efficient Generation of Monitor Circuits for GSTE Assertion Graphs. In: *Proc. International Conference on Computer-Aided Design (ICCAD)*, 2003. – ISBN 1–58113–762–1, S. 154–159

[HKB96] HOJATI, Ramin ; KRISHNAN, Sriram C. ; BRAYTON, Robert K.: Early Quantification and Partitioned Transition Relations. In: *Proc. International Conference on Computer Design, VLSI in Computers and Processors*, IEEE Computer Society, 1996. – ISBN 0–8186–7554–3, S. 12–19

[HS96] HACHTEL, Gary D. ; SOMENZI, Fabio: *Logic Synthesis and Verification Algorithms*. Boston : Kluwer Academic Publishers, 1996

[JKSC08] JAIN, Himanshu ; KROENING, Daniel ; SHARYGINA, Natasha ; CLARKE, Edmund M.: Word-Level Predicate-Abstraction and Refinement Techniques for Verifying RTL Verilog. In: *IEEE Transactions on Computer-Aided Design* 27 (2008), Nr. 2, S. 366–379

[JMF95] JAIN, J. ; MUKHERJEE, R. ; FUJITA, M.: Advanced Verification Techniques Based on Learning. In: *Proc. International Design Automation Conference (DAC)*, 1995, S. 420 – 426

[JMH00] JANG, J. ; MOON, In-Ho ; HACHTEL, G.: Iterative Abstraction Based CTL Model Checking. In: *Proc. International Conference on Design, Automation and Test in Europe (DATE)*, 2000

[KGP01] KUEHLMANN, Andreas ; GANAI, Malay K. ; PARUTHI, Viresh: Circuit-based Boolean Reasoning. In: *Proc. International Design Automation Conference (DAC)*. New York, NY, USA : ACM, 2001. – ISBN 1–58113–297–2, S. 232–237

[KK97] KUEHLMANN, Andreas ; KROHM, Florian: Equivalence Checking Using Cuts and Heaps. In: *Proc. International Design Automation Conference (DAC)*, 1997, S. 263–268

[Kun93] KUNZ, W.: An Efficient Tool for Logic Verification Based on Recursive Learning. In: *Proc. International Conference on Computer-Aided Design (ICCAD)*, 1993, S. 538–543

[Kur94] KURSHAN, R. P.: *Computer-Aided Verification of Coordinating Processes – The Automata-Theoretic Approach*. Princeton, New Jersey : Princeton University Press, 1994

[MA03] MCMILLAN, Kenneth L. ; AMLA, Nina: Automatic Abstraction without Counterexamples. In: *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2003, S. 2–17

[MAB06] MORIN-ALLORY, Katell ; BORRIONE, Dominique: Proven correct monitors from PSL specifications. In: *Proc. International Conference on Design, Automation and Test in Europe (DATE)*, 2006. – ISBN 3–9810801–0–6

[Mat96] MATSUNAGA, Y.: An Efficient Equivalence Checker for Combinational Circuits. In: *Proc. International Design Automation Conference (DAC)*, 1996, S. 629–634

[McM93] MCMILLAN, K. L.: *Symbolic Model Checking*. Boston : Kluwer Academic Publishers, 1993

[McM02] MCMILLAN, Kenneth L.: Applying SAT Methods in Unbounded Symbolic Model Checking. In: *Proc. International Conference on Computer Aided Verification (CAV)*, 2002, S. 250–264

[ME99] M.CLARKE, Edmund ; E.LONG, David: Model Checking and Abstraction. In: *ACM Transactions on Programming Languages and Systems* 16 (1999), S. 1512 – 1542

[MKRS00] MOON, In-Ho ; KUKULA, James H. ; RAVI, Kavita ; SOMENZI, Fabio: To Split or to Conjoin: the Question in Image Computation. In: *Proc. International Design Automation Conference (DAC)*, 2000, S. 23–28

[MKSS99] MOON, In-Ho ; KAKULA, James ; SHIPLE, Tom ; SOMENZI, Fabio: Least Fixpoint Approximations for Reachability Analysis. In: *Proc. International Conference on Computer-Aided Design (ICCAD)*, 1999, S. 41–44

[MMZ+01] MOSKEWICZ, Matthew W. ; MADIGAN, Conor F. ; ZHAO, Ying ; ZHANG, Lintao ; MALIK, Sharad: Chaff Engineering an Efficient SAT solver. In: *Proc. International Design Automation Conference (DAC)*, 2001

[MSS99] MARQUES-SILVA, Joao P. ; SAKALLAH, Karem A.: GRASP: A Search Algorithm for Propositional Satisfiability. In: *IEEE Transactions of Computers* 48 (1999), S. 506–521

[NIJ+97] NARAYAN, Amit ; ISLES, Adrian J. ; JAIN, Jawahar ; BRAYTON, Robert K. ; SANGIOVANNI-VINCENTELLI, Alberto L.: Reachability Analysis Using Partitioned-ROBDDs. In: *Proc. International Conference on Computer-Aided Design*, IEEE Computer Society, 1997. – ISBN 0–8186–8200–0, S. 388–393

[NTW+08] NGUYEN, Minh D. ; THALMAIER, Max ; WEDLER, Markus ; BORMANN, Jörg ; STOFFEL, Dominik ; KUNZ, Wolfgang: Unbounded Protocol Compliance Verification using Interval Property Checking with Invariants. In: *IEEE Transactions on Computer-Aided Design* 27 (2008), November, Nr. 11, S. 2068–2082

[One09] ONESPIN SOLUTIONS GMBH, GERMANY: *OneSpin 360MV*. `http://www.onespin-solutions.com`, 2009

[QCC96] QUER, Stefano ; CABODI, Gianpiero ; CAMURATI, Paolo: *Decomposed Symbolic Forward Traversals of Large Finite State Machines*. 1996

[RS95] RAVI, K. ; SOMENZI, F.: High Density Reachability Analysis. In: *Proc. International Conference on Computer-Aided Design (ICCAD)*, 1995, S. 154–158

[SDJ00] SHIMIZU, K. ; DILL, L. ; J.HU, A.: Monitor-Based Formal Specification of PCI. In: *Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2000

[SIG95] SIG, PCI: *PCI Local Bus Specification 2.1*. http://www.pcisig.com/specifications. `http://www.pcisig.com/specifications`. Version: 1995

[Sof09] SOFTWARE Novas: *Debussy®nWave™*. `http://www.novas.com/Solutions/Debussy/`, 2009

[Som] SOMENZI, Fabio: *CUDD: CU Decision Diagram Package*. `http://vlsi.colorado.edu/~fabio/CUDD/`,

[SSS00] SHEERAN, M. ; SINGH, S. ; STALMARCK, G.: Checking Safety Properties Using Induction And A SAT-Solver. In: *Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2000

[SWWK04] STOFFEL, D. ; WEDLER, M. ; WARKENTIN, P. ; KUNZ, W.: Structural FSM-Traversal. In: *IEEE Transactions on Computer-Aided Design* 23 (2004), May, Nr. 5, S. 598–619

[TNW+07] THALMAIER, Max ; NGUYEN, Minh D. ; WEDLER, Markus ; STOFFEL, Dominik ; KUNZ, Wolfgang: Formale Verifikation von SoC Protokollimplementierungen. In: *Tagungsband 1.GMM/GI/ITG-Fachtagung Zuverlässigkeit und Entwurf*, 2007

[TR01] T.BALL ; R:MAJUMDAR: Automatic Predicate Abstraction Of C Programs. In: *Proc. Conference on Programming Language Design and Implementation*, 2001

[WLJ+06] WANG, C. ; LI, B. ; JIN, H. ; HACHTEL, G. ; SOMENZI, F.: Improving Ariadne's Bundle by following Multiple Threads in Abstraction Refinement. In: *IEEE Transactions on Computer-Aided Design* 25 (2006), S. 2297 – 2316

[WSBK07] WEDLER, M. ; STOFFEL, D. ; BRINKMANN, R. ; KUNZ, W.: A Normalization Method for Arithmetic Data-Path Verification. In: *IEEE Transactions on Computer-Aided Design* 26 (2007), November, Nr. 11, S. 1909–1922

[WSFT04] WINKELMANN, K. ; STOFFEL, D. ; FEY, G. ; TRYLUS, H.: Cost-Efficient Block Verification for a UMTS Up-Link Chip-Rate Coprocessor. In: *Proc. International Conference on Design, Automation and Test in Europe (DATE)*. Paris, France : IEEE Computer Society, February 2004

[YHY+05] YANG, Ya-Ching ; HUANG, Juinn-Dar ; YEN, Chia-Chih ; SHIH, Che-Hua ; JOU, Jing-Yang: Formal Compliance Verification Of Interface Protocols. In: *Proc. IEEE International Symposium on VLSI*, 2005

Lebenslauf

Name:	Nguyen Duc-Minh
Anschrift:	Kurt-Schumacher-Str. 22/App 0.1 67663 Kaiserslautern
Geburtsdatum:	25.03.1976
Geburtsort:	Hanoi/Vietnam
Familienstand:	verheiratet

Ausbildung

1982-1991	Grundschule in Hanoi
1991-1994	Hanoi-Amsterdam Gymnasium in Hanoi Abschluss: Abitur
1994-1999	Hochschulstudium zum Elektronik-und-Telekommunikation Ingenieur an der Hanoi Technischen Universität / Hanoi-Vietnam Bachelorabschluss: 5.1999
2001-2003	Masterstudium zum M.Sc. Eletrical Engineering an der Technischen Universität Kaiserslautern Masterabschluss: 6.2003

Berufstätigkeit

1999-2001	Wissenschaftlicher Mitarbeiter an der Hanoi Technische Universität / Hanoi-Vietnam
seit 8/2003	Wissenschaftlicher Mitarbeiter von Prof. Dr.-Ing. Wolfgang Kunz am Lehrstuhl Entwurf Informationstechnischer Systeme des Fachbereichs Elektrotechnik und Informationstechnik der Technischen Universität Kaiserslautern