A Practical Model Checking Approach Using FormalCheck

Leila Barakatain

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfilment of the Requirements
for the Degree of Master of Applied Science at
Concordia University
Montreal, Quebec, Canada

August 2000

© Leila Barakatain, 2000

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Leila Barakatain

Entitled: A Practical Model Checking Approach Using FormalCheck

and submitted in partial fulfilment of the requirements for the degree of

Master of Applied Science (Electrical and Computer Engineering)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________ Asim Al-Khalili

______________________ Jean Lamarche

______________________ Ferhat Khendek

______________________ Sofiène Tahar

Approved by ______________________________________
Chair of Department or Graduate Program Director

__________ 20___ ______________________________
Dean of Faculty

Abstract

A Practical Model Checking Approach Using FormalCheck

Leila Barakatain

Verification of industrial designs is becoming more challenging as technology advances and demand for higher performance increases. One of the most suitable debugging aids is automatic formal verification, which tests behaviors under all possible executions of a system. However, automatic formal verification is limited by the state explosion problem. This thesis presents a practical verification approach using FormalCheck, which helps reducing the state space explosion problem when verifying the high level descriptions of practical systems. This approach relies on the design's built-in hierarchy as the mechanism to conquer its complexity during verification. Then an assume guarantee paradigm is used to verify functional units built on top of instantiated and previously verified modules. We applied this approach to an industrial design (Transmit Master/Receive Slave (TMRS) Telecom System Block) as a case study. The TMRS was thoroughly verified and inconsistencies in the design with respect to its specification were uncovered through model checking. The main contributions of this thesis are, (1) the application of a variety of model checking techniques to a real size design and (2) proposing a number of improvements to the design flow which can accelerate the whole verification process.

Acknowledgements

This work would not have been possible without the help of many people. It is impossible to compose an exhaustive list, which would include almost every one of my friends, the fellows in our research group, the colleague in PMC-Sierra, and the faculty members.

However, I must give special thanks to my research advisor, Dr. Sofiène Tahar. As a constant source of technical feedback and encouragement, he has shown me how to conduct research and to present it. His encouragements and believe in me was the main motivation for me during my research. Also, I would like to thank him for choosing me and giving me the opportunity to work with PMC-Sierra Inc. on their special project on formal verification.

I would like to thank Mr. Jean Lamarche for assisting me in keeping a consistent direction in my research while I was working in PMC-Sierra Inc. as a co-operative student, and for his serving in my examining committee. I am also grateful to Dr. Ferhat Khendek who kindly served in my examining committee. I would like to thank our Graduate Program Director, Dr. Asim Al-Khalili for being so considerate, kind, and patient towards the students. I must thank him for all his help, support, and guidance throughout my studies in Concordia, and for chairing my examining committee.

This work was partially done while in internship at PMC-Sierra Inc. I am grateful to Mr. Alan Nakamoto, Mr. Bob Blais, Mr. Jean-Marc Gendreau, Ms. Peggy McGee, and Mr. Francois Lambert who despite their busy schedule, always found a time spot for me and answered my questions willingly.

Throughout my study in Concordia many people have encouraged and helped me through many obstacles. In particular I would like to thank the members of the Hardware Verification Group for discussions and helping me understand the concepts and clearing up my concerns, specially Dr. Yassine Mokhtari, Hong Peng, and Mohammad Ishtiaque Khan. I thank my close friends Akbar Garjani and Azimeh Sefidcon who always suited me by their wise advice and by showing me a new meaning of life. I would like to thank my caring supervisor in the Boeing company, Mr. Terrill Hendrickson who always believed in me and considered me special enough to grant me the educational leave and to give me the opportunity to grow.

If not for the love and support of my parents, my husband, Mohammad, my daughter, Mina, and my sisters, I would not be able to complete my M.A.Sc. program at Concordia University. I am particularly grateful for my husband's love, care and understanding, especially during the busy periods. My daughter, although only four years old, was always understanding and kept me going with her unconditional love.

Last but not least, everything that I have is given by God, and my eternal gratitude goes to him.

Leila Barakatain

August, 2000

Table of Contents

Chapter 1 Introduction **1**
1.1 Motivation and Goal ... 1
1.2 Background and Related Work ... 3
1.2.1 Formal Verification ... 3
1.2.2 Verification Techniques ... 4
1.2.3 Related Work ... 7
1.3 Scope of the Thesis ... 10

Chapter 2 Introduction to FormalCheck **12**
2.1 Definition of Basic Terms in FormalCheck ... 13
2.1.1 Properties, Constraints and Queries in FormalCheck ... 13
2.1.2 Query Result and Status ... 16
2.1.3 State Variables ... 17
2.2 Run Options in FormalCheck ... 18
2.2.1 Algorithms ... 18
2.2.2 Deadlocks ... 19
2.2.3 Run Time ... 20
2.2.4 Advanced Options ... 20
2.3 Modes of Verification ... 21
2.4 Reduction Algorithms in FormalCheck ... 21
2.5 Reduction Seeds ... 22
2.6 Electronic Scissors ... 23

Chapter 3 A Practical Model Checking Approach **24**
3.1 Basis of the Proposed Approach ... 25
3.2 Horizontal Verification ... 27
3.3 Vertical Verification ... 28
3.4 Validating the Verification Results ... 30
3.5 Reduction and Abstraction Methods ... 32
3.6 Recommendations to Accelerate the Verification Process ... 35

3.7 Combining Simulation and Model Checking 37
3.8 Conclusion and Summary 40

Chapter 4 Model Checking of TMRS Using FormalCheck: A Case Study 42
4.1 The TMRS Telecom System Block 43
4.1.1 The TMRS Icon and Pin Descriptions 44
4.1.2 Block Diagram of the TMRS 50
4.1.3 SCI-PHY Receive Slave Operation 50
4.2 Model Checking of the TMRS Based on SCI-PHY Level 2 Protocol 51
4.2.1 Environment for the SCAN Block 53
4.2.1.1 Model and Environment Abstractions 54
4.2.1.2 Defining Proper Environment and Datapath Abstractions 55
4.2.1.3 State Variables 58
4.2.2 Expression Macros 59
4.2.3 Properties 60
4.2.4 Experimental Results 72
4.2.5 Further Errors Found 75
4.3 Conclusions 78

Chapter 5 Conclusion and Future Work 81

Bibliography 85

Appendix A Properties of the TMRS 91

List of Figures

Figure 2.1 : The organization of a FormalCheck project 13

Figure 2.2 : The explored state space in simulation and model checking 14

Figure 2.3 : Verification Window 15

Figure 3.1 : Hierarchy in high level design process 26

Figure 3.2 : A design unit 27

Figure 3.3 : Linear dependency between design units A and B 28

Figure 3.4 : Interacting design units 29

Figure 3.5 : Parallel processes 34

Figure 4.1 : 8-bit SCI-PHY Cell Format 44

Figure 4.2 : TMRS Icon 45

Figure 4.3 : Block diagram of the TMRS 49

Figure 4.4 : The OCA_O and OCA_OEB signals in the TMRS, and RCA signal in SCI-PHY protocol 52

Figure 4.5 : RADDR[4:0] bus to TMRSSEL signal 52

List of Tables

Table 4.1: SCI-PHY Signal Cross-Reference 53

Table 4.2: Properties of Receive Slave SCI-PHY mode of the TMRS 61

Table 4.3: Verification results of model checking on the abstracted model of TMRS . 74

Table 4.4: Verification results of model checking on the original model of TMRS 75

Table 4.5: Detailed time frame of the verification of the TMRS case study........ 79

Chapter 1

Introduction

1.1 Motivation and Goal

Industrial designs are becoming more and more complex as technology advances and demand for higher performance increases. In today's red-hot economy, staying on schedule to hit a narrow window of market opportunity often means the difference between product dominance and product death. With the volume and complexity of logic required to satisfy function-hungry consumers comes exponential growth in the difficulty of making sure it all works. The validation of a design involves checking that the physical design does indeed meet its specification [22]. The mere scale of complexity of newer designs, makes it likely that the designer will fail to anticipate some possible interactions between different components of the system.

In a traditional design flow, validation is accomplished through simulation and testing. Some errors inside a design may exhibit nondeterministic behaviors, and therefore, will

not be reliably repeatable. This makes testing and debugging using simulation difficult. Also, exhaustive testing for nontrivial designs is generally infeasible, therefore, testing provides at best only a probabilistic assurance. Formal Verification [22], in contrast to testing, uses rigorous mathematical reasoning to show that a design meets all or parts of its specification. For that reason, formal verification is on the critical path for today's IC designers, no matter what type of system they are building [21].

Formal verification has problems of its own class too. The major problem with automatic formal verification is that a large amount of memory and time is often required, because the underlying algorithm in these methods usually involves systematic examination of all reachable states of the system to be verified. As the number of reachable states increases rapidly with the size of a system, the basic algorithm by itself becomes impractical: the number of states for a system is often too large to check exhaustively within the limited time and memory that is available. This phenomenon is known as the state space explosion problem [12].

The goal of this thesis is to introduce and apply practical model checking approaches, which offer some solutions to alleviate the state space explosion problem for verification of high level descriptions of real systems, and to suggest improvements in the design flow which can accelerate the verification process.

There are different academic and commercial model checking tools available. In this work we chose FormalCheck [5] as the verification tool for the following reasons.

1. FormalCheck supports both VHDL and Verilog (the two hardware description languages widely used in industrial designs), unlike many formal verification tools (specially the academic ones), which use some special hardware description language.

2. The automatic reduction algorithms of FormalCheck allow it to verify larger designs than any other model checker in the market.
3. An important feature in a model checker is that constraints can be assumed to prove properties. In FormalCheck, both constraints and properties can be expressed in the same format, therefore, turning around constraints into properties to be proven and vice versa is very easy.
4. Usually to verify a design, a proper working environment has to be built for the target design. In FormalCheck this environment can be defined using the facilities provided by FormalCheck.
5. FormalCheck provides some semi-automatic techniques to reduce the state space explored during the verification process.

1.2 Background and Related Work

1.2.1 Formal Verification

From the early 1980s until the mid-1990s, digital logic simulation alone handled nearly all functional and timing verification in IC design. As chips grew from tens of thousands of gates in the '80s to more than 1 million today, simulation run time stretched from a few hours to days, or even weeks. The volume of stimulus patterns (vectors) required to check every logic function and timing path threatened to overwhelm even the best-funded development organizations. Consequently, reliance on simulation as the sole source of verification has proven impractical [21].

Formal hardware verification attempts to overcome the weakness of non-exhaustive simulation by proving the correspondence between some abstract specification and the

design in hand. There are 3 different techniques of formal hardware verification [22], namely:

- Deductive Methods (Theorem Proving)
- Equivalence Checking
- Model Checking

1.2.2 Verification Techniques

One of the earliest approaches to formal hardware verification was to describe both the implementation as well as the specification in a formal logic. The correctness result was then obtained by proving in the logic, that the specification and implementation were suitably related [39]. *Theorem provers* were developed to handle most of the tedious application of proof rules, and to make sure that each step in a proof is correct [43]. Among the best known interactive theorem provers are the Boyer-Moore Theorem Prover 'Nqthm' [6], PVS [32] and the Cambridge HOL System [18]. One of the main strengths of the theorem proving approach is its ability to provide effective means to guarantee the correctness of safety critical systems, in which the consequences of an error outweighs the time and the financial investment required to construct the proof.

Unfortunately, theorem proving based verification requires a large amount of effort on the part of the user in developing specifications of each component and in guiding the theorem prover through a large number of lemmas. Therefore, for designs that are not safety critical, the assertional proof and automated theorem proving techniques are too expensive. Furthermore, if a system contains an error, it is impossible to construct a proof, and often, no useful diagnostic information is generated.

On the other hand there are *automatic formal verification techniques* which are based on state enumeration. Although these techniques are specialized for finite-state systems, they are easy to use and provide useful diagnostic messages if the system fails to observe the required properties. Both *Equivalence Checking* and *Model Checking* are of this type. Zafiropulo et al. [45] originally proposed the state enumeration method for protocol verification.

Equivalence Checking is used to prove functional equivalence of two design representations modeled at different levels of abstraction [40]. Equivalence Checking can be divided into two categories: *Combinational Equivalence Checking* and *Sequential Equivalence Checking*.

In *Combinational Equivalence Checking*, the functions of the two circuits to be compared are converted into canonical representations of Boolean functions [8], typically Binary Decision Diagrams (BDDs) or their derivatives, which are then structurally compared. Since current designs are mainly clock-driven synchronized, to perform the Combinational Equivalence Checking between two different sequential models, a design has to be divided into pieces. Then each register (or flip-flop) of one model has to be mapped into the other model and their combinational circuits have to be compared between every two consecutive registers. The drawback of this type of verification is that it cannot handle the Equivalence Checking between RTL and behavioral model. Usually RTL and behavioral models are developed separately and should have the same outputs at certain clock cycles, but it is impossible to map each register in the RTL model to one in the behavioral model. As an example, Verity [23] is a verification system aimed at checking equivalence between RTL specification and gate or transistor level implementation for large, hierarchically

structured designs. Examples of Combinational Equivalence Checking tools are Cadence Affirma, Synopsys Formality, IBM Boole Eye, etc.

In Sequential *Equivalence Checking*, given two sequential circuits using the same state encoding, their equivalence can be established by building the product finite state machine and checking whether the values of two corresponding outputs are the same for any initial states of the product machine. Sequential Equivalence Checking only considers the behavior of the two designs while ignoring their implementation details such as latch mapping. Therefore, Sequential Equivalence Checking is able to verify the equivalence between RTL and behavioral model. The drawback of this technique is that it cannot handle a large design due to state space explosion problem. MDG [13] and VIS [7] are examples of Sequential Equivalence Checking tools.

Model Checking is an algorithm that can be used to determine the validity of formulas written in some temporal logic with respect to a behavioral model of a system. Model Checking is based on the state space exploration technique, and uses the reachability state graph as a Kripke structure, which encodes the set of all possible sequences of states for a system over computation trees. Examples of Model Checkers are SMV [28], VIS [7], SPIN [20], Murφ [15], RuleBase [4], and FormalCheck [5] [24].

Model Checking tools are effective as debugging aids for industrial designs, and since they are fully automated, minimal user effort and knowledge about the underlying technology is required to be able to use them. However, there are two drawbacks with Model Checking. The first, the state space explosion and how to avoid it, and the second is the difficulty of judging whether the verified property completely characterizes the desired behavior of the system.

In the verification techniques explained above, a verifier faces the trade-off between choosing an automated technique, which is in general subject to limitations in applicability, and more general but also more work-intensive approaches. This trade-off can be avoided by *combining Model Checking and Deductive Reasoning* if possible. In this method, one can apply automated techniques to subsystems or simpler obligations and then use deductive approaches to combine the results thus obtained into an overall correctness result. STeP [27] is an example of an integrated system (integrates Model Checking and Theorem Proving) for reasoning about reactive systems.

1.2.3 Related Work

In this section, several case studies of mostly industrial designs of protocols and communication hardware is presented, in order to provide a sense of how designs have been verified in practice and the techniques the researchers used to avoid the state space explosion. Comparing these case studies provides insight into the scope and limitations of currently available techniques of formal verification.

Chen et al. at Fujitsu Digital Technology Ltd. [10] exploited symbolic model checking to detect a design error in an ATM (Asynchronous Transfer Mode) circuit. The circuit consists of about 111K gates and supports high-speed switching operations at 156 MHz. When the circuit was manufactured it showed an abnormal behavior under certain circumstances. Using SMV, they identified the error by checking some properties. To avoid state space explosion, they abstracted the width of addresses from 8 bits to 1 bit, and the number of addresses in a Write Address first-in-first-out (FIFO) from 168 to 5. However, in some cases a property could not be verified because of this reduction and a detailed gate

level model was needed for certain blocks to pinpoint the source of the error.

Harkness and Wolf [19] applied symbolic model checking to the Summit bus converter. The bus converter provides a communication link between a high-speed processor bus and a low speed I/O bus. They informally abstracted away aspects of the design that were considered unimportant to the verification conditions they had in mind. Model checking was then applied to an abstracted model of the design, which was formalized in SMV to demonstrate the presence of deadlock on the systems link queues in earlier versions of the design, as well as the absence of deadlock in a revised version.

Eiriksson and McMillan [16] described the verification of the cache coherence protocol for a Silicon Graphics multiprocessor using SMV model checker. To make the verification tractable, the high level specification was kept as abstract as possible and dependent variables were eliminated.

Eisner [17] described the verification of the cache coherent non-uniform memory access (CC-NUMA) cache coherence protocol using RuleBase. CC-NUMA is a distributed shared memory multi-processor. The author made an abstract model of the protocol which supported 5 out of 16 commands and 1 out of 4 configuration bit settings, which was the smallest configuration deemed to be meaningful in this project. The specification was manually translated into a formal model in the (SMV-like) RuleBase language. The results of this verification was finding a coherency bug inside the design.

Pong et al. [36] modeled the S3.mp cache coherence protocol in the Murφ language and system. They modeled only one memory block and it was assumed that other blocks do not interfere. The symmetry of the model was exploited to group equivalent symmetric states. Using these abstraction methods, several protocol errors were detected and cor-

rected. Numerous other coherence protocols have been verified, including those for Gigamax by McMillan and Schwalbe [29], Futurebus+ by Clark et al. [11], and SCI architectures by Stern and Dill [41].

Lu and Tahar [25] verified a 4x4 ATM switch fabric using VIS. Since they did not succeed in using the original fabric to check the properties (due to state space explosion) they abstracted the model by reducing the datapath from 8 bits to 1 bit. After this reduction in the model, they succeeded verification of the properties, but the verification time of each property was unreasonably long. To make this process more efficient, they used property division and latch reduction techniques, which allowed them to verify a number of safety and liveness properties of the abstract model in reasonable CPU time. In [3], we reverified the 4x4 ATM switch fabric of [25] using FormalCheck and compared the results of the two experiments. We were able to verify both the abstracted model and the model with 8 bit datapath with very reasonable amount of CPU time and memory usage.

The efficiency of state exploration and model checking methods depend heavily on the size of the reachability state graph. As observed from the above examples of verifications, the authors had to manually make an abstraction of their model to be able to perform verification on their design. The reduction capabilities in FormalCheck makes it more attractive compare to other automatic formal verification tools (this will be explained in Chapter 3).

So far, the only work reported in the open literature using FormalCheck is the work of Xu et al. [44], where they verified a Frame Multiplexer/Demultiplexer (FMD) Chip from Nortel Semicinductors. The FMD chip is part of a system used in multiplexing/demultiplexing framed data between various channels and a SONET line. The authors constructed

(in Verilog) a non deterministic model of the user which mimics the normal operating environment. The model abstraction was done by applying automatic and semi-automatic reduction facilities provided in FormalCheck. The authors succeeded detecting two design errors in the implementation of the FMD model.

Writing extra code to mimic the working environment is not very reliable, and it can add to the state space explored during the verification. In this thesis it is explained how to use FormalCheck, so that not only the model abstraction and reduction can be done by this tool, but also the normal operating environment of the design can be defined inside FormalCheck as well.

1.3 Scope of the Thesis

This thesis first describes a practical verification approach using FormalCheck. The approach introduces model checking early in the design cycle in order to minimize the overall verification effort. It relies on the built-in hierarchy of the design as the mechanism to conquer its complexity during verification. This approach minimizes state space explosion problems by concentrating the bulk of the verification effort to RT level modules suitable for model checking. Also it suggests a semi-formal verification, by exploiting the benefits of both simulation and model checking to achieve better test coverage. Later, some modifications to the design flow are suggested, which can help accelerating the verification process. To complete the discussions about model checking, we explore different abstraction and reduction techniques available in the open literature. At the end, the proposed approach and abstraction techniques are applied to verify the Transmit Master/ Receive Slave TMRS Telecom System Block (TSB) from PMC-Sierra Inc. [34], as a case

study.

The rest of the thesis is organized as follows: Chapter 2 is a brief introduction to FormalCheck and its automatic and semi-automatic reduction techniques. Chapter 3 introduces a step by step model checking approach based on FormalCheck. Chapter 4 gives a description of the Transmit Master/Receive Slave (TMRS) Telecom System Block (TSB) and applies the proposed verification approach on the target TSB. Conclusions and ideas on future work are presented in Chapter 5.

Chapter 2

Introduction to FormalCheck

FormalCheck [9] is a model checking tool commercialized by Cadence to be used to augment conventional simulation techniques. It is used to verify synthesizable Register Transfer Level (RTL) VHDL or Verilog design models using the model checking method. The patented reduction algorithms of FormalCheck, allow it to handle larger designs than any other model checker on the market. In [9] it is reported that properties on designs of up to 5,000 latches and 100,000 combinational variables have been verified with this tool. FormalCheck provides an intuitive graphical interface to simplify the verification process. This chapter presents a brief overview of FormalCheck and explains the facilities provided in this tool which makes the process of formal verification much easier and more efficient than other model checking tools available.[1]

1. The material in this chapter are based on the information in FormalCheck User's guide [5].

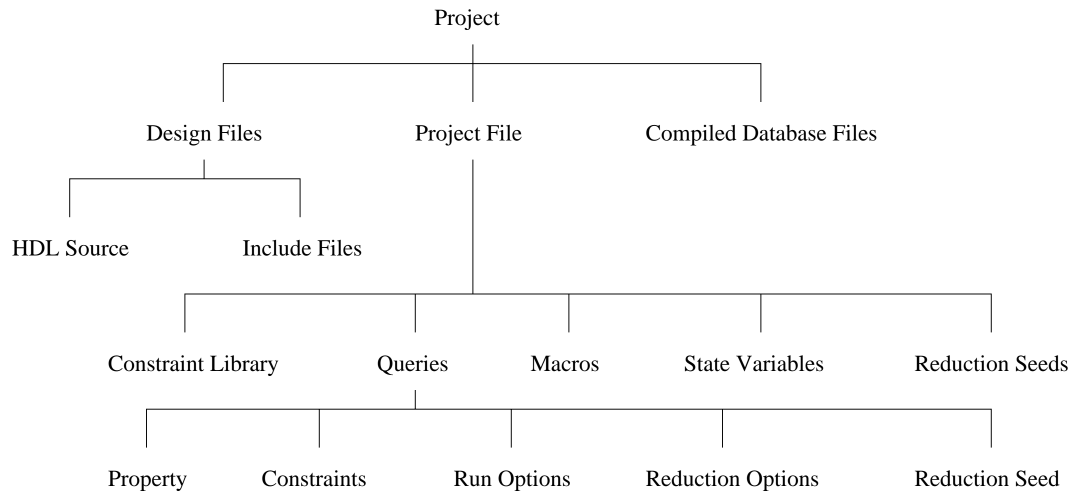

Figure 2.1 : The organization of a FormalCheck project

2.1 Definition of Basic Terms in FormalCheck

Applying FormalCheck to verify a design involves first creating a project. A FormalCheck project is a container (directory structure) that holds all information relevant to verification including (refer to Figure 2.1):

- pointers to the design model file
- information needed to compile the design, including the HDL language used, top level entity name and hierarchical dependencies
- *queries* containing *properties* and *constraints*
- current status of *queries* and the results of *queries* already verified.

2.1.1 Properties, Constraints and Queries in FormalCheck

A model checker verifies that a design model exhibits specific behaviors (*properties*) that are required by the design specification. Properties that form the basis of a model

checker's queries fall into two categories: *safety* and *liveness*. *Safety* properties describe behaviors that can be shown to be false by a finite simulation trace. Safety properties use one of the two formats: the *always* format and the *never* format. *Liveness* properties describe behaviors that are eventually exhibited. Liveness properties cannot be checked with a simulation tool, unless the maximum number of steps before the fulfillment of the eventuality is known. Liveness properties use one of the three formats: the *Eventually* format, the *Eventually Always* format, and the *Strong Liveness* format.

By default, a model checker attempts to verify properties under all possible input scenarios. However, many designs are intended to function properly only under more constrained circumstances. Therefore the default check could return irrelevant failures. *Constraints* can be used to limit the input scenarios used in verification, and as a result, limit the state space of the design model that is verified (Figure 2.2) [9].

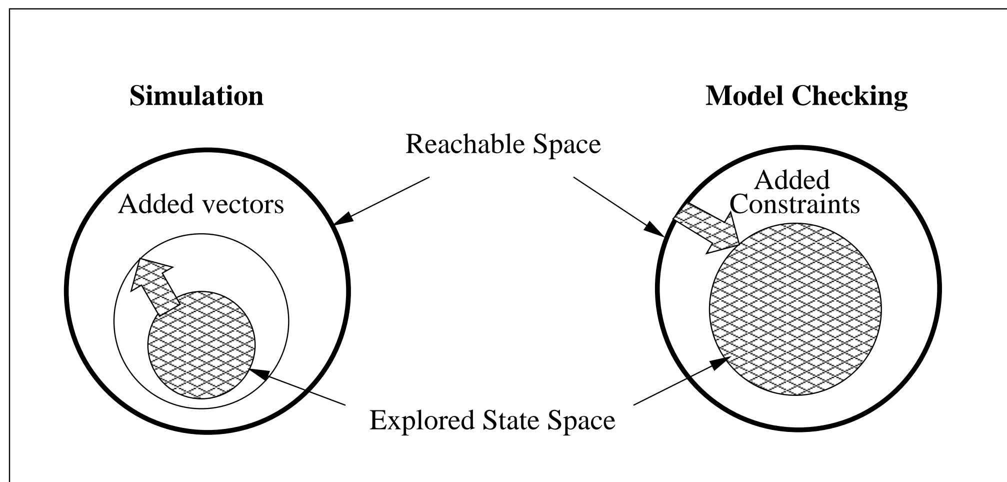

Figure 2.2 : The explored state space in simulation and model checking

Properties and Constraints simply express behaviors. The difference is that a property ensures that a behavior occurs, and a Constraint assumes that the behavior occurs for the

purpose of verification. Each behavior is defined in terms of one to three conditions, namely: *Fulfilling Condition*, *Enabling Condition*, and *Discharging Condition*. A *Fulfilling Condition* is required for all behaviors. In FormalCheck, a Fulfilling Condition needs to be qualified and by default this condition is checked over the entire state space. A *Verification Window* can be specified by adding an *Enabling Condition* and/or *Discharging Condition* (Figure 2.3). Commonly, the behavior described by a Fulfilling Condition is not required until after it is triggered by an event. This event is defined by adding an *Enabling Condition*. The requirement for a Fulfilling Condition may be canceled by a *Discharging Condition*. Once discharged, the Fulfilling Condition is not required to occur unless it is retriggered by an Enabling Condition.

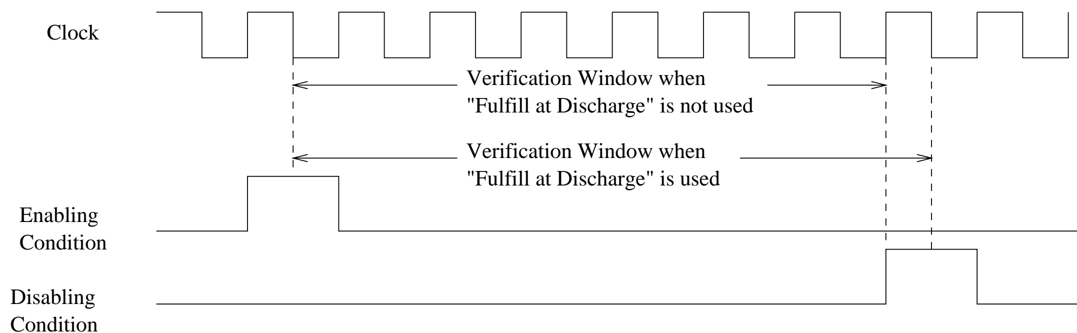

Figure 2.3 : Verification Window

The *Verification Window* is defined to begin immediately after the Enabling condition becomes true, and it terminates immediately upon the Discharging Condition becoming true (Figure 2.3). The Verification Window can be further controlled (for Never, Always, and Eventually behaviors) by using the settings *Fulfill Delay*, *Duration* and *Of Edge*. The *Fulfill Delay* field postpones the beginning of the verification window beyond the assertion of the Enabling Condition until a given number of occurrences of an event (defined by

a Boolean expression) in the *Of Edge* field. The Verification Window remains open for a given duration in the *Duration* field.

FormalCheck works by checking a *query*. A query is a group of properties that are to be checked together subject to specified Constraints. Constraints that are marked as "default" are automatically added to a new query when it is created. Several queries, each containing a different set of properties and Constraints, can be defined for a project. However, it is recommended to include only one property per each query.

2.1.2 Query Result and Status

There are eight possible results that a query can have: *New*, *Failed*, *Verified!*, *Running*, *Terminated*, *Scheduled*, *No Error*, and *Vacuous*.

- *New,* indicates a new query on which no verification attempt has been run yet.
- *Failed*, indicates the previous verification attempt ended in finding an error in the query. In this case an error track is available for debugging. In FormalCheck a back-referencing utility permits the user to click on an error in the error track waveform, and get a pop-up window of the VHDL or Verilog source, with the cursor on the assignment which gave the indicated variable the indicated value [9] [24].
- *Verified!*, indicates the query was previously verified.
- *Running*, indicates a verification attempt is currently underway.
- *Terminated*, indicates the verification attempt was stopped by the user or by an abnormal condition. Termination usually occurs when the verification run hits a time or memory limit set by the user or when the verification runs out of memory. A query termination can also be caused by errors. In case of a query termination, the "verify.*"

files should be viewed for more information. These files are found under the specific project and query directory.

- *Scheduled*, indicates the query has been scheduled for verification.
- *No Error*, indicates no failures were found. This result is not the same as *Verified!*. This result is obtained when Auto-Restrict is chosen as the algorithm for verification run (this will be explained in Section 2.2.1). If the result is *No Error* then FormalCheck should be re-run using the Symbolic State Enumeration or Explicit State Enumeration algorithms for full verification.
- *Vacuous*, indicates that the Enabling Condition of some property was never satisfied, thus the Fulfilling Condition was never checked. Vacuity also can occur due to overconstraining the property. In that case FormalCheck automatically performs a check to detect some forms of Constraints conflict [9].

There are two possible values for a query's status: *Current* and *Outdated*. *Current* indicates that the verification result accurately reflects the latest compiled version of the design model and the query. *Outdated* indicates the verification result is out of date as a result of new compiled version of the design model or new version of the query.

2.1.3 State Variables

Boolean expressions are adequate to capture a behavior at some specific point in time. Often the condition to be checked is actually the culmination of a sequence of events. Such sequential conditions are defined in terms of *State Variables*. A State Variable consists of a name, range, initial value and its defined behavior. The behavior of the State Variable is defined using "If then else" constructs. Each additional state doubles the state

space, therefore, the range of a State Variable should be defined as narrow as possible to avoid adding unnecessary states to the verification. In general, State Variables are used for the following reasons:

1. To count events: A State Variable can be used to count events such as clock edges. However, counting clock edges is not a common function for user defined State Variables any more, since it is automated with the “edgeof” feature (Section 2.1.1).
2. To parallelize serial data or to detect complex sequences of events: Paralelizing a serial data can be used to detect flags on a serial channel.
3. To be used as a memory element: A State Variable can be defined to remember the previous value of a signal.

2.2 Run Options in FormalCheck

FormalCheck provides run options that are used to select which algorithms are used in a verification run. Proper use of run options will ensure an efficient verification run. Run options in FormalCheck include verification algorithms, deadlocks, run time, and advanced options.

2.2.1 Algorithms

FormalCheck provides three algorithms to perform verification, namely, *Symbolic State Enumeration* (using ordered Binary Decision Diagrams, BDDs [8]), *Explicit State Enumeration*, and *Auto-Restrict*. *Symbolic State Enumeration* is the default algorithm in FormalCheck. This algorithm is generally useful for verifying models with a very large set of states. However, the BDDs can become intractably large, especially when the design

model includes many arithmetic expressions.

Explicit State Enumeration is generally useful for verifying models with fewer than 1000 primary input values per state and which may contain a large number of arithmetic expressions. It may succeed in other cases in which Symbolic State Enumeration runs out of memory, as long as the number of input values per state is not large and the total number of states is limited to 10 million. If errors are likely to be found in the design model, then this algorithm may find the errors faster than Symbolic State Enumeration.

The *Auto-Restrict Algorithm* attempts to narrow down the portion of the design in which a failure of the query (if any) is likely to be found, and performs the analysis there. It can dramatically accelerate the discovery of a failure. However it is very important to note that if no failures are found when using this option, it may not guarantee that the design model is verified. In such cases, FormalCheck reports “No Errors” instead of the usual “Verified!”, and the verification has to be re-run using the Symbolic State Enumeration or Explicit State Enumeration algorithms for full verification. Auto-Restrict is useful when other algorithms run out of memory.

2.2.2 Deadlocks

A deadlock state is a state from which the only transition is back to itself [1]. By default, FormalCheck does not consider deadlocks as errors. Choosing this option will treat deadlocks as errors and report any that are found. Selecting this option might increase the verification time. Note that a deadlock is not possible if the query is subject to a clock

1. This is the definition of deadlock in FormalCheck. In literature a deadlock state is a state from which there are no transitions at all.

Constraint. The perpetual motion of the clock precludes the possibility of a deadlock state.

2.2.3 Run Time

By Default there are three cases that the verification run will stop:

- after finishing exhaustively checking the entire design model
- when an error is found in the design model
- when the available memory is exhausted.

By choosing the run time option a time limit, in CPU minutes, can be placed on the verification run.

2.2.4 Advanced Options

There are three different options that can be added to the run options of a query.

- "-L" specifies a limit on memory used. The verification begins a wrap-up sequence when the memory limit is reached. (The wrap-up sequence generates results and reports that can be used for troubleshooting.)
- "-#hardlimit=" stops verification without the wrap-up sequence. When this option is used, a limit equal to the user accessible physical memory is recommended.
- "-DVACCHECK" is used to check for a query that is vacuous because its Constraints are identically false on the design model or the query is over-constrained. "-DVACCHECK" removes properties from the verification and checks only Constraints in conjunction with the design model. If the result of this check is "Verified" then Constraints are conflicting with each other (or query is vacuous). If the result of this

check is “Failed” then the existing Constraints are okay (query is not vacuous).

2.3 Modes of Verification

FormalCheck is designed to operate in one of two ways during the design process: the *Debug Mode* and the *Regression Mode*. The *Debug Mode* is used during the early stages of the design process to verify a small portion of the design. FormalCheck is used to edit a VHDL or Verilog model, compile it, verify a query, debug any errors, and repeat. Typically only one query is verified at a time.

The *Regression Mode* is used only when the entire design is presumed correct. In this case, the design is close to completion and no known bugs exist. FormalCheck is used to compile a complete design, verify multiple queries to check all behaviors in the specification, and identify any unexpected errors. FormalCheck uses a patented algorithm which seeks to replace verification with a checksum computation to make sure the latest design fix did not affect the previously verified properties. When this succeeds, the entire verification of hundreds of queries can complete in a few minutes, consuming very little computational resources.

2.4 Reduction Algorithms in FormalCheck

FormalCheck uses a patented localization reduction algorithm to reduce the size of the design model relative to the property being tested. The localization reduction algorithm may significantly reduce the verification run time and the computing resources required for verification. By default, a *1-Step* version of this algorithm is used. The *1-Step* algorithm makes automatic reductions based on circuit topology and requires no additional

input from the user. This is a good starting point, however for more computational intensive designs the 1-Step reduction may not be enough. Complex designs may require the *Iterated* reduction algorithm. The *Iterated* algorithm can take several iterations to reduce the design model but will generally result in a smaller reduced model than if the 1-Step algorithm is used. The Iterated algorithm takes longer to reduce the design model but the computer memory required maybe significantly less. The Iterated reduction algorithm does not work with the Auto-Restrict verification algorithm.

A *Reduction Seed* can be supplied to specify the starting point for the algorithm (refer to Section 2.5). A good starting point allows faster completion but does not affect the logical outcome.

2.5 Reduction Seeds

The Iterated reduction algorithm seeks to find a portion of the design model on which it is sufficient to run the verification. This portion by construction will be such that if the query is verified on this portion, it is guaranteed to be true on the original model. Conversely, if the query is falsified on this portion, then the algorithm expands the error track to an error track of the original 1-Step reduction model.

The Iterated algorithm iterates through a succession of intelligent guesses at a suitable reduced model. The closer the starting point for the algorithm is to a portion of the design model that is sufficient to check the query, the faster the algorithm will run. Therefore it is useful for the user to make an educated guess at what a sufficient portion could be. A sufficient portion is one which contains sufficient logic to check the query. A reduction seed is not required but is recommended when performing Iterated reduction.

The user designates a candidate portion of the design model by marking its boundary signals *Start As Input*, which in effect disconnects those signals from the driving expressions and turns these signals into primary inputs. If it turns out that some driving expressions in fact are needed to verify the query, then the algorithm will reinstate them, cutting the design model signals at an alternate point instead.

2.6 Electronic Scissors

Similar to Reduction Seeds, Electronic Scissors are user defined designations of design model signals that are forever disconnected from their driving expression, and thus treated as primary inputs. These signals are designated as *Make Input*. This provides the user with a means to isolate a portion of the design model. Signals thus designated retain this designation, and this designation may be used with both the 1-Step and the Iterated algorithms. If FormalCheck returns the status “Verified!”, it means it would also have returned “Verified!” if the query had been run on the full model (without any signals designated Make Input). However, if FormalCheck returns an error track, this may be an artifact of the reduction, and not be possible in the unreduced model.

Chapter 3

A Practical Model Checking Approach

In this chapter a practical model checking approach using FormalCheck is described, which introduces model checking early in the design cycle in order to minimize the overall verification effort. It relies on the built-in hierarchy of the design as the mechanism to conquer its complexity during verification. This approach minimizes state explosion problems by concentrating the bulk of the verification effort to Register Transfer Level (RTL) modules suitable for model checking. Then, considering the structure of the design in hand, a pattern is used to verify functional units (cores) built on top of instantiated and previously verified modules. In [37] the idea of verifying the functional units of a system using FormalCheck was introduced very briefly. In this thesis, the idea of verifying the cores of a system advances into much more details by concentrating on different possible structures of a design and offering solutions of how to deal with each one of them.

While performing model checking, it is often quite difficult to judge whether the verified property completely characterizes the desired behavior of the system, so in this chap-

ter we will explain how to validate the verification results in FormalCheck. The other problem in model checking is the state space explosion problem and how to avoid it. Later in this chapter, we will talk about some reduction and abstraction methods in order to reduce the state space explored during the verification process.

The time spent for the verification of a product is as much or may be more than the time spent to design it. In this chapter we are proposing several improvements in the design flow which can result in preventing bugs from being introduced into the design, as well as making the verification process faster. Considering the strengths and weaknesses of both simulation and model checking, we suggest a semi-formal verification approach, which exploits the positive features of both techniques to achieve much better test coverage of designs.

3.1 Basis of the Proposed Approach

Most design processes rely on hierarchy as the mean of dealing with complexity. Generally these processes consist of developing a specification for the entire chip and then, repeatedly partitioning the specification into smaller and more manageable *functional building blocks*. These functional building blocks are usually further partitioned into much smaller design units represented as *synthesizable hardware components* at the RTL. Once the design units are verified for basic functionality, they are combined to implement functional building blocks, often referred to as *cores*. These cores represent the actual implementation for distinct levels of functionality directly derived from the architectural partition of the chip. Finally, these cores are combined to assemble the entire chip (Figure 3.1).

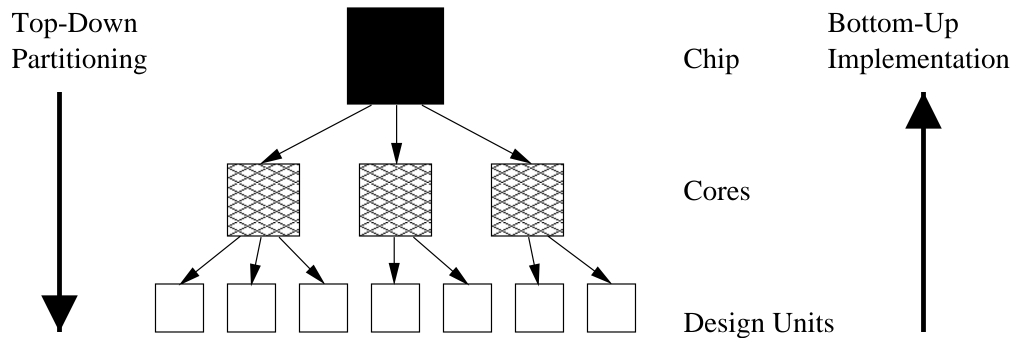

Figure 3.1 : Hierarchy in high level design process

A typical design flow relies on a top-down partitioning phase where the specifications are broken down into focused areas of functionality (cores and design units). This top-down partitioning phase is followed by a bottom-up implementation phase (Figure 3.1). At the implementation phase, designers mix and match legacy components, third-party IP cores, and newly developed pieces of logic to create their designs. During this phase, designers not only spend most of their time writing RTL code, but simulating and debugging the interfaces between the third-party solutions and their own designs.

Using a model checker (in our case FormalCheck) early in the design cycle addresses the functional verification problem as the implementation phase evolves, that is, in a bottom-up fashion. As each design unit becomes available, a model checker is used to assess their correct functionality. Properties assumed at the inputs of each design unit/core are verified on the components driving them. This *horizontal verification* is followed up by a *vertical verification* phase, where properties already proven (or provable) at lower levels in the chip hierarchy are assumed to conduct the verification of properties at higher levels. The vertical verification was first introduced very briefly in [37] and was claimed that it was applicable to all types of design structures. In this thesis we describe the vertical veri-

fication in detail. We explain this approach considering different possible structures of a design, and how to verify the properties of each specific structure. We will also explain that the vertical verification is applicable to most of the design structures, however, there are some specific cases, where the application of vertical verification will lead to a circular behavior (refer to Section 3.3).

3.2 Horizontal Verification

As design units are created, a model checker is used to evaluate their functionality under all possible scenarios allowed within their operating conditions. The goal is to attain a high level of confidence on the correct functionality of each design unit as a stand alone entity. When using FormalCheck, this is verified by ensuring that all properties in a design unit hold true (Verified!) within the Constraints set forth by assuming certain environmental conditions. For example, consider the design unit shown in Figure 3.2. The property of A (P_A) is verified assuming the legal environmental conditions.

$$\text{Assume (Legal Env.)} \xrightarrow{\text{Prove}} P_A$$

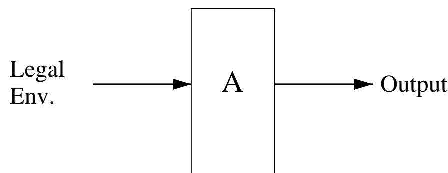

Figure 3.2 : A design unit

An important feature in a model checker is that properties can be assumed to prove other properties. This feature allows for easy verification of interfaces. Turning around assumed properties (Constraints) into properties to be proven and vice versa, facilitates the

verification of interfaces by providing the means to ensure that assumptions made on the inputs of a design unit are validated as properties held by the designing unit driving it. In the case of two interacting design units, properties on one design unit are assumed to hold while verifying the properties of the other design unit. This case is explained in more details in Section 3.3.

3.3 Vertical Verification

An important feature in FormalCheck is that the properties can be assumed (Constraints) to prove other properties. FormalCheck supports assuming lower level properties (sub-properties) along with the legal environmental conditions to conduct the verification of larger property. This capability of being able to verify a property based on the sub-properties being implied supports a vertical verification process exploiting the built-in hierarchy of the chip [37].

In vertical verification, properties that are already proven at lower levels in the hierarchy of the chip, are assumed to conduct the verification of properties at higher levels. At this level, when verifying a property of one component we must make assumptions about the behavior of the other components. Let's start with the simplest case. Assume the design units inside a functional building block (core) have linear dependencies. For example, consider the dependencies between blocks A and B of the core shown in Figure 3.3.

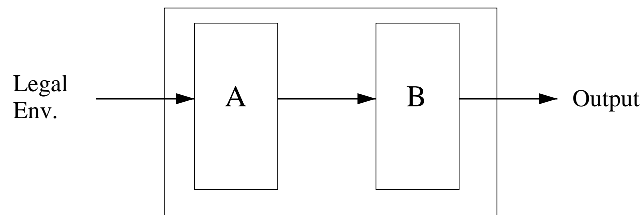

Figure 3.3 : Linear dependency between design units A and B

To verify the property of B (P_B), The following steps should be followed.

- The property of A (P_A) should be verified along with legal environmental conditions. This part is usually done when verifying the design units.

$$\text{Assume (Legal Env.)} \xrightarrow{\text{Prove}} P_A$$

- The property of B (P_B) is verified, assuming the property of A (P_A is turned to a Constraint) and the legal environmental conditions.

$$\text{Assume (Legal Env. and } P_A\text{)} \xrightarrow{\text{Prove}} P_B$$

For more detailed example of this kind refer to Chapter 4, Section 4.2.3, Property_6A.

In some cases the relationship between the design units of a core is not linear and the two design units are interacting. For example, consider the two blocks C and D of the core shown in Figure 3.4. The interacting blocks can have two different kinds of structure. One type of structure is when there is one way dependency between blocks C and D, meaning only the internal output signal of block C depends on the value of the signal coming from D (and not vice versa). In that case, to verify the property of D the following steps should be followed.

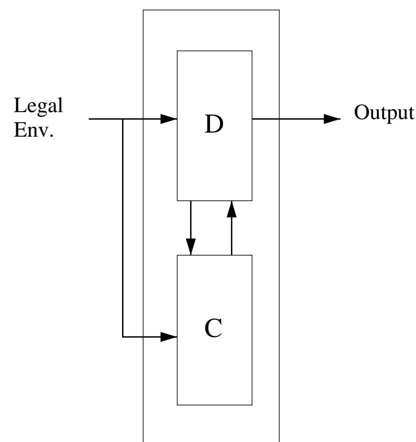

Figure 3.4 : Interacting design units

- Properties on design unit D (P_D) are assumed to hold (Constraint), while verifying the properties of design unit C (P_C).

$$\text{Assume (Legal Env. and } P_D) \xrightarrow{\text{Prove}} P_C$$

- Conversely, properties in design unit D (P_D) are proven assuming that the properties of design unit C (P_C) hold (Constraint).

$$\text{Assume(Legal Env. and } P_C) \xrightarrow{\text{Prove}} P_D$$

For more detailed example of this type of verification, refer to Chapter 4, Section 4.2.3, Property_5.

The other type of interacting blocks are the ones that have two way dependency between blocks C and D, meaning the internal output of one block (block C) depends on the internal output of the other block (block D), and vice versa. In this case the solution explained above, results in circular behavior. The danger with such circularity is if the assumed properties are false we could get trivially satisfied properties, but we cannot deduce the implementations are false [1]. This circularity can be broken by induction over time [30], which is out of the scope of this thesis.

3.4 Validating the Verification Results

The last step of a verification process is to validate the verification results of the queries to make sure the design environment was captured properly. Usually, proper capture of the design environment is one of the most problematic steps. This step is limited not just to the proper definition of Clock and Reset pins, but includes the specification of all combina-

tional and sequential input Constraints required to establish a reasonable interface to verify the design. Underconstraining the inputs leads to spurious failures, requiring tedious iterations of the environment model. What is worse, if the environment overconstrains the inputs, then potentially fatal input sequences will be overlooked. Since verification addresses all possible behaviors, the issue of a correct and adequate environment model is more critical, and thus requires more precision than the conventional approach used for simulators [38].

FormalCheck performs a number of automatic checks to find likely instances of an overconstrained design, however, it cannot detect all possible scenarios that can cause the overconstrain condition. We recommend performing sanity checks on queries that are reported as "Verified!". The first step for sanity check is to validate the Constraints of a verified query by running the -DVACCHECK on it. This check is independent of properties and therefore should only be done once for any set of Constraints. The other indicator is to analyze the size of the nominal and reachable state space and average variable coverage. For instance if the amount of reached space or the State Variable average coverage is noticeably small, the query should be suspected for overconstrain condition. Other clues for the source of overconstrain can be found in the *Missing Values* report (in Query Manager) of FormalCheck. A common indicator is when there are values that some signal never attains.

Adding a Discharging Condition to a property, results in the reduction of the state space being explored during the verification. Defining the verification window too short can cause the property to be trivially verified, therefore, properties which include Discharging Condition should be carefully reviewed for the overconstrained condition as well.

3.5 Reduction and Abstraction Methods

The efficiency of state exploration and model checking methods depends heavily on the size of the reachable states. The more extensive the reachable states, the more time and memory it takes to verify a system. The biggest obstacle of model checking is often unmanageably huge number of reachable states. FormalCheck relies on the localization of a property into a portion of the design that is relevant to what is being verified. However, such localization by itself may not succeed in performing the verification unless assisted by the user. As the complexity of functional building blocks increases, the default setting used by most formal verification tools may be insufficient. One needs some reduction and abstraction methods in order to reduce the state space explosion problem. Common abstraction techniques are *model abstraction using symmetry* [31], *datapath abstraction* [33], *environment abstraction* [37], and *property decomposition* [12].

Finite state concurrent systems frequently contain replicated (symmetrical) components. It is possible to find symmetry in memories, caches, register files, bus protocols, network protocols, or any design that has a lot of replicated structure. The idea of *model abstraction using symmetry* is based on the observation that having symmetry in the system implies the existence of nontrivial variant groups that preserve both the state labeling and the transition relation. Such groups can be used to define an equivalence relation on the state space of the system. The smaller model produced by this relation is often smaller than the original model. Thus, it can be used to verify any property of the original model [12] [31]. Therefore, the idea behind model abstraction using symmetry is to remove the redundant structure of a design and perform model checking only on required circuits [26]. The critical aspect of symmetry reduction in verification is detecting symmetries. As

an example consider a network circuit which is capable of transferring data cells coming form 32 external channels. If a smaller model which supports 8 external channels can be brought out from the original model, which has the same state labelling and the transition relations as the original model, then the smaller model can be used to verify the properties of the original model. In [26] symmetry reduction was performed on an already abstracted ATM switch model. In our case study in Chapter 4, we applied this reduction technique to reduce a real size design.

Datapath abstractions are used to minimize the impact of datapaths on the performance of model checking engines. Usually, the majority of the state holding devices (registers and latches) are in the datapath. The basic idea behind this technique is to identify the datapath storage elements that do not contribute to the control flow of the design [33]. The user simply restricts the verification of a property to a handful of different and interesting scenarios. For example, consider a network protocol which is capable of transferring data cells with different lengths, and the cell length is determined by the information included in the first bytes of the data cell (header). The length of the cell usually affects the computational parts of the design and does not change the control flow of the finite state machine (FSM) of the design. Therefore, the minimum cell length can be used in order to reduce the state space.

Proper capture of the design environment includes the specification of all combinational and sequential input Constraints required to establish a reasonable interface to verify the target design [38]. By *environment abstraction,* the user instructs the model checker what areas in the design must be excluded and replaced by assumptions of their behavior. This technique simplifies the complexity of the verification task. In Formal-

Check, environment abstraction is possible using Reduction Seeds, Electronic Scissors, and Constraints (Refer to Chapter 2 Sections 2.1.1, 2.5 and 2.6). For example assume in the target model there is a test block in addition to the main block which controls the main functionality of the model. Also assume only the verification of the main block is of interest. In that case, the test block can be removed using the Electronic Scissors. The inputs of the main block which are fed by the test block, can be set to neutral constant values using Constraints in FormalCheck.

Many finite state systems are composed of multiple processes running in parallel. The specifications for such systems can often be decomposed into properties that describe the behavior of small parts of the system [12]. *Property decomposition* is the process where a property is broken up into several sub-properties such that verification of sub-properties guarantees the verification of the original property [37].

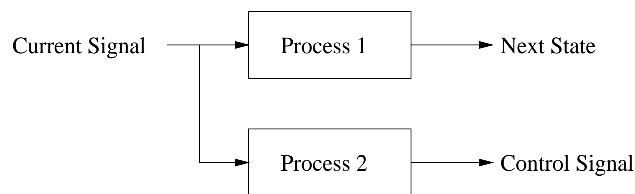

Figure 3.5 : Parallel processes

For example, assume the objective design unit contains two parallel processes (Figure 3.5). One process handles the state transition, and the other one produces the control signals (notice that each specific control signal has to be asserted for a specific state of the system, therefore, these two processes are not considered independent). Also assume the user wants to verify a property which concludes the next state and the control signal that has to be asserted at the same time for that specific state of the FSM. This property can be

divided to two properties, one which checks for the next state of the FSM and the other one checks for the control signal that the system produces for that specific state. For a more detailed example refer to Chapter 4, Section 4.2.3, Property_1.

The reduction techniques expressed in this section were applied to reduce the industrial design TMRS [34], discussed in Chapter 4.

3.6 Recommendations to Accelerate the Verification Process

Designers of today's massive new circuits face a pair of conflicting goals. They must increase verification coverage and quality to ensure single-pass success. They also must find ways to contain the expansion of verification time, effort, and cost. One of the biggest concerns of companies is "What kind of changes in the design flow can accelerate the verification process?".

To find a way to reduce the time spent for verification, including simulation and formal verification, first we have to answer this question: Where do bugs come from? Bugs can be introduced to the design through several channels [14] such as:

- Incorrect specifications
- Misinterpretation of specifications
- Misunderstanding between designers
- Missed cases
- Protocol non-conformance
- Resource conflicts, etc.

The only way to detect bugs inside a design is to define the right queries (for model checking) and/or to write the proper test benches (for simulation) to uncover them. Test

processes and methodologies can considerably reduce the risk of releasing systems which could fail in the field. In [42] the author introduces smart testing to improve the quality of functional testing (simulation) and validation. In this part of the thesis we propose improvements in the design flow which result in preventing bugs from being introduced into the design, as well as making the verification process faster. We are mainly focusing on model checking and ways to improve the design flow, in order to ease and accelerate the query definition and verification process. These enhancements are listed in following.

- The first and probably most important improvement in the design process is having a good quality (clear, complete, and up-to-date) specification for each product life cycle step. Applying the hierarchical verification methodology explained in this chapter will contribute to this improvement. As mentioned before, the model checking of the design units starts as soon as they are ready. At that time, the specifications for the embedded design unit is still fresh in designer's mind. Therefore, applying a model checker at the early stages of the design cycle results in the formal documentation for each design unit.

- The specification for the different integration levels should use the same signal names. Also, the lower level specifications should be built on the higher level specification functional definitions, rather than redefining them.

- It is a well known fact that the detail available in the specification of the targeted design is usually not enough to fully verify the functionality of it, therefore, a test plan should be created early in the product design and verification cycle. A test plan identifies all the features whose functionality must conform to the specification. Taking this approach will notably ease defining suitable queries for verification of the design

model.

- Each specification should contain the complete functionality requirement at that particular integration level. The more source documents one has to use to define the queries, the higher the probability that the functionality of the model will be misunderstood, therefore, the chance of defining wrong queries will be higher.
- The automated reduction and abstraction algorithms provided by the model checker should be used as much as possible. Reductions and abstractions that are done manually can be complicated, error-prone and slow. Of-course the apparent effectiveness of the model checkers depends on the size of the design. Therefore, the manual abstraction should only be used, if the automated reduction and abstraction algorithms cannot reduce the model efficiently to avoid the state space explosion.
- Designs usually are modified in some point in time. Properties verified on the earlier version of the design should be reused (with minor modifications) to verify the behavior of the new design. Since the properties usually do not contain the cycle by cycle implementation details found in test vectors or test benches, they are more easily reused when verifying a modified version of the previous work.

3.7 Combining Simulation and Model Checking

Many designers hope that model checking will be a solution to the problems involved with simulation, such as increasing test coverage of design, and shortening the time to market. Model checking will help (indeed, it will be essential) in niches where it is particularly effective, however, it does not replace traditional functional simulation. Model checking is used to expand these verification techniques.

Since datapaths increase the number of selection variables per state, datapaths and computational intensive designs tend to rapidly cause state space explosion. Also, in a lot of instances the data values do not affect the control of the design. Hence, model checking is most effective when used for the verification of control-oriented designs such as control logic, finite state machines (FSMs) and protocols. On the other hand simulation is better suited for datapath analysis and computation results analysis. When verifying a system, the verification engineer has two questions in mind: (1) does the system work properly when legal inputs are present?, (2) what is the behavior of the system in case of the presence of illegal inputs? Depending on the data content implied by the chip level property, design verification engineers set apart properties suitable for model checking from those properties requiring traditional simulation methods for their verification.

For example, consider a network block, which transfers data cells of a certain length. Also, assume this block has to assert a control signal named Start_Of_Cell, when transferring the first byte of a data cell. The first thing the verification engineer would possibly verify is, (assuming legal environment) when there is a good cell available to transfer, the Start_Of_Cell signal will eventually be asserted. This type of properties which check the control parts of the design (such as control signals, FSM, etc.) can be verified using model checking.

The next thing the verification engineer wants to verify is the behavior of the system when there is a bad cell in the stream of input data cells. Is the model capable of skipping the bad cell and going back to the normal mode of operation or will it have a lock up situation? This type of property is very hard or even impossible to verify by model checking for the following reasons.

1. When defining this type of property, a bad cell has to be introduced to the model. Usu-

ally there is a FSM which indicates if the received cell is good or not. Since the information about the cell is provided inside its header, either the datapath or the assumed property of it has to be present to verify the behavior of this FSM. Therefore, the design verification engineer might face state space explosion.

2. Assuming unlimited amount of memory is available and state space explosion is not an issue, the other problem that the verification engineer will face is defining a property, which expresses the following behavior:

   - The bad cell will be skipped and the system will be back into normal mode of operation and it wont fall into a lock up situation.

   - The bad cell will be skipped but the following good cells are guaranteed to be transferred properly.

   To define and verify the above behavior, a sequence of events need to be provided, consequently, defining this property for a model checker will be very hard.

3. Assume the verification engineer succeeds defining and verifying this property. The next step of verification is sanity check. Model checkers provide useful diagnostic messages if the system fails to observe the required properties, however, for the properties that are reported as “Verified” there are no waveforms or diagnostic messages. Making sure that the verified property completely characterized the desired behavior and was not verified trivially will be very hard and time consuming, specially when dealing with illegal inputs.

Considering the problems explained here, time, resources, and energy needed to define and verify these types of properties, one can easily decide that simulation is a better approach for this type of verification.

3.8 Conclusion and Summary

In this chapter a practical verification approach applicable at the RT level using FormalCheck was presented. In this approach sub-properties are assumed while conducting the proof for chip level properties. This approach for verification is possible since the properties of a complex design rest upon the properties of its embedded components and their interactions. In case the embedded components have two way dependencies, the application of this method can lead to circular reasoning. In this type of situation induction over time is implied.

The model checking process is started by design engineers to verify the functionality of each design unit. Applying a model checker at the early stages of the design cycle results on the formal documentation for each design unit. As the design units are combined to implement cores of functionality, designers and verification engineers start working together. Designers define and verify fundamental properties within the core itself while verification engineers use some of these properties to verify their interfaces to third-party IP cores.

Changes on chip level specification should trigger a controllable and predictable amount of changes on previously verified properties. In FormalCheck, a property immune to the latest design fix re-verifies in a matter of minutes, regardless of how long it took to verify the first time. However, if the outcome of a previously verified property is affected by the latest design fix, FormalCheck re-starts the verification for such property from scratch.

Since model checkers cannot handle all design styles, a sound hierarchical verification approach relies in their inter-dependence with simulation. Simulation is used primarily on

the data path oriented and computational intensive parts of the design while model checking is intended for use on the control intensive parts. Both technologies are very capable of finding bugs associated with safety properties. However, only model checking has the ability of providing a definite answer on properties dealing with eventualities.

In this chapter we also proposed some improvements in the design flow which are capable of preventing bugs from being introduced into the design, as well as making the overall verification process faster. By focusing on model checking aspect of verification, some suggestions are made to enhance the design flow, in order to ease and accelerate the query definition and verification process.

In the verification approach described in this chapter, usually the verification of the design units is assuming no state space explosion problems. This problem arises when trying to verify a core. This methodology was applied to a real size circuit, the Transfer Master/Receive Slave (TMRS) Telecom System Block (TSB) from PMC-Sierra Inc. [34].

In the next chapter, we present the methods and results on formal verification of the TMRS. We briefly explain the functionality and the structure of the TMRS when in receive slave SCI-PHY [35] mode of operation, and then construct a signal cross-reference between the TMRS and the target protocol in order to match their interface signals. To avoid state space explosion, we perform environment abstraction and datapath abstraction techniques presented in Section 3.5. Also, an abstracted model of the TMRS is built using symmetry to speed up the verification process. After establishing a proper environment, we consider 13 queries of the TMRS, including liveness properties and safety properties. Property decomposition was performed to uncover a bug inside the design. In this experiment, we discovered several mismatches between the TMRS design, its specification, and the SCI-PHY protocol.

Chapter 4

Model Checking of TMRS Using FormalCheck: A Case Study

In the previous chapter a practical verification approach was explained. This chapter presents a case study which exercises this methodology. The design units (the basic building blocks at RT level) are usually not very large, and as a result state space explosion is not a problem in the verification of each design unit. This problem is frequently encountered when verifying the design at the higher levels of the implementation like verifying a core or a whole chip.

This chapter presents the methods and results on formally verifying the implementation of the Receive Slave SCI-PHY (Saturn Compatible Interface for ATM-PHY devices) [35] mode of the Transmit Master/Receive Slave (TMRS) design [34] using FormalCheck [5]. The TMRS Telecom System Block (TSB) (designed by PMC-Sierra Inc.), was chosen to be verified as a research case to experiment the benefits of formally verifying the control blocks of a design using FormalCheck over using simulations to find bugs. This *megacell*

implements the output port of a cell interface. It can be configured to operate either as a bus master or a bus slave. The TMRS supports SCI-PHY Level 2 [35] protocol and ANY-PHY (proprietary protocol of PMC-Sierra). SCI-PHY Level 2 protocol is a superset of UTOPIA (Universal Test & Operations PHY Interface for ATM) Level 2 protocol [2].

The TMRS is made of two major modules namely “Internal Register CBI interface” and “SCAN Interface”. The SCAN module is responsible for the main functionality of the TMRS block, whereas the CBI (Configuration Block Interface) module handles the configuration and the test route of it. In this experience the verification of the SCAN module was of interest.

4.1 The TMRS Telecom System Block

The TMRS [34] implements the output port of a cell interface. It can output the cell data either in 8-bit or 16-bit wide format at clock rates up to 52 MHz. Data transfers are cell-based, that is an entire cell is transferred to one PHY device before another is selected. It outputs cells on SCI-PHY compatible interface.

The 8-bit wide, variable data structure in SCI-PHY interface is shown in Figure 4.1. A user defined (UDF) byte is included in the data structure to allow Header Error Control (HEC) generation to be performed either in the ATM layer device or the PHY layer device. The prepended bytes are used by ATM switch cores in system specific ways to route the cell through those cores [35].

The TMRS block is designed to interface directly to a Multi-channel Cell FIFO (MCF). It directly supports up to 32 logical channels each corresponding to a physical layer ATM device. Each logical channel corresponds to a FIFO channel in the external FIFO. When

the TMRS is operated as a bus slave, it autonomously multiplexes the traffic from up to 32 logical channels and presents them as a single cell stream. The logical channel is identified by the first word of the cell data received from the FIFO.

	Bit 7	Bit 0
Word 0 (optional)	Extended Address	PHYID[4:0]
Word 1 (optional)	User Prepend	
Word 2 (optional)	User Prepend	
Word 3 (optional)	User Prepend	
Word 4 (optional)	User Prepend	
Word 5	H1	
Word 6	H2	
Word 7	H3	
Word 8	H4	
Word 9 (optional)	H5/UDF	
Word 10	PAYLOAD1	
	. . .	
Word 57	PAYLOAD48	

Figure 4.1 : 8-bit SCI-PHY Cell Format

4.1.1 The TMRS Icon and Pin Descriptions

The Icon of the TMRS is presented in Figure 4.2. In this study only the control part of TMRS (not the data-path) in Receive-Slave SCI-PHY mode is of interest, therefore we only describe the control pins used in this mode of operation.

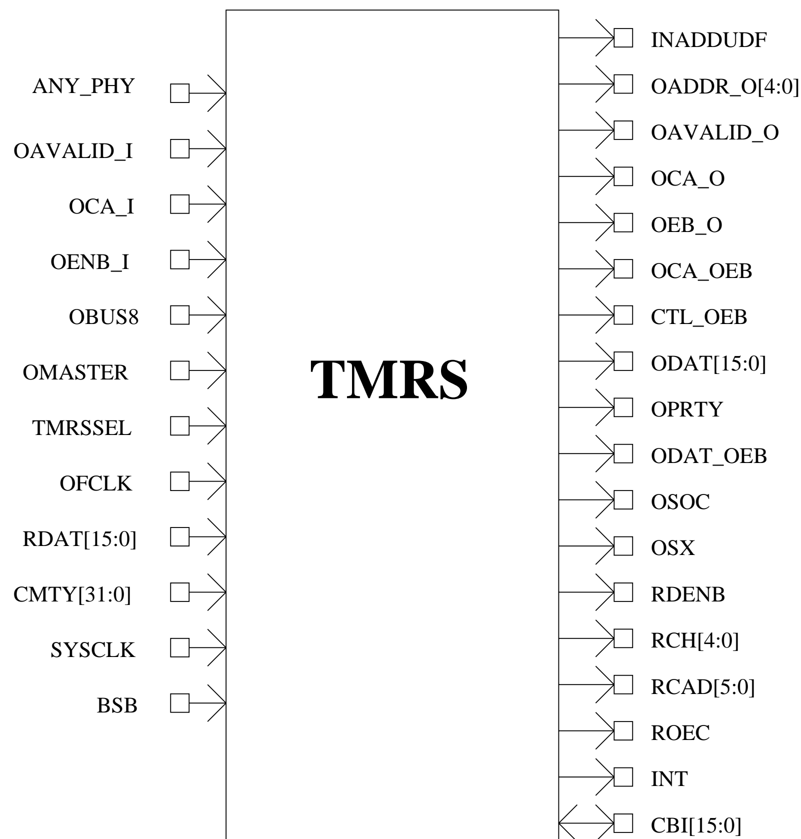

Figure 4.2 : TMRS Icon

OMASTER: The Output Port Master Select (OMASTER) pin determines the direction of the output port control signals. When OMASTER is low, the output port of the device in which the TMRS is integrated is a bus slave and complies with the SCI-PHY or ANY-PHY receive protocol. The TMRSSEL, OAVALID_I, OENB_I, OCA_O, and OCA_OEB signals are used.

ANY_PHY: When set to low, the TMRS supports SCI-PHY (UTOPIA Level 2) protocol, otherwise it supports ANY-PHY protocol.

TMRSSEL: The TMRS Select (TMRSSEL) pin selects the TMRS for polling and cell transfer operations. This pin is only used when the TMRS is configured as a slave, i.e.,

OMASTER is low. If the TMRSSEL is sampled high when OAVALID_I is high, the TMRS is enabled for polling. The TMRS is selected for a cell transfer when TMRSSEL and OENB_I were sampled high in the previous clock cycle and OENB_I is sampled low in the current clock cycle. The cell transfer is performed while OENB_I is maintained low.

OAVALID_I: The Output Port Address Valid (OAVALID_I) output indicates that the bus master is asserting a valid PHY address for polling purposes. In SCI-PHY mode, OAVALID_I is active high. In slave mode (OMASTER is low), the OCA_O output is driven on the clock cycle following OAVALID_I and TMRSSEL being sampled high. If OAVALID_I is sampled low, OCA_O becomes high impedance. The TMRS supports polling in contiguous cycles if OAVALID_I is held high.

OENB_I: The active low Output port Enable (OENB_I) input is used to initiate the transfer of cells from the cell output port to a PHY device when TMRS is in Slave mode. In SCI-PHY mode, when OENB_I is sampled low and the TMRS is selected, a word is output on bus ODAT[15:0] on the following first clock cycle. The selection occurs when OENB_I and TMRSSEL were last sampled high. To transfer a complete cell OENB_I has to be maintained low for the period of cell transfer. The period varies depending on the bus width and cell format. OENB_I can be deasserted any time during the cell transfer to perform word level flow control, but it is assumed that no other devices will be selected when OENB_I is reasserted.

OCA_O: The Output Cell Available (OCA_O) output provides cell level flow control. OCA_O is only used if the MASTER input is low (Slave mode). The TMRS indicates the presence of a cell to transfer via the OCA_O output. When OAVALID_I and

TMRSSEL are asserted, OCA_O is driven to high if at least one line of the CMTY[31:0] status bus is low and the prefetch of the next FIFO is finished. In SCI-PHY mode, the OCA_O assertion occurs on the first clock cycle following OAVALID_I and TMRSSEL assertion. If all CMTY[31:0] lines are high or the prefetch of the next FIFO is not finished, OCA_O is deasserted. Note that the CMTY line corresponding to the FIFO of the current transfer is always masked (i.e., always considered to be set to high).

OCA_OEB: The Output Cell Available Output Enable (OCA_OEB) is an active low signal intended to be wired directly to an active low pad output enable. It is used to control the direction of the OCA pin at the device level. In Slave mode (OMASTER low), OCA_OEN becomes low if the TMRS is selected for polling; otherwise, OCA_OEB is high.

OBUS8: The Output port Bus width select (OBUS8) selects the interface bus width. When OBUS8 is high, only ODAT[7:0] present valid data and ODAT[15:8] are forced low. When OBUS8 is low, all ODAT[15:0] inputs are used.

OFCLK: The Output FIFO Clock (OFCLK) is used to read words from the TMRS cell output port. OFCLK must cycle at a 52 MHz or lower instantaneous rate. All output port SCI-PHY bus timing is relative to the rising edge of OFCLK.

CMTY[31:0]: The Channel Empty bus (CMTY[31:0]) input reports the cell availability of external FIFOs or of an external FIFO channel. A logic low on a line of the CMTY[31:0] indicates the corresponding channel of the FIFO has at least one complete cell available to transfer. CMTY[31:0] is sampled on the rising edge of OFCLK.

SYSCLK: The System Clock (SYSCLK) input is used to synchronize the FIFO interface. SYSCLK must cycle at a 52 MHz or lower instantaneous rate.

OSOC: The Output Start Of Cell (OSOC) marks the start of the cell on the ODAT[15:0] bus. When operating in SCI-PHY mode, OSOC assertion indicates that the first word of the cell structure is present on the ODAT[15:0] stream. When operating in Slave mode, OSOC is valid when OENB_I was asserted in the previous cycle.

ODAT_OEB: The Output Data Enable (ODAT_OEB) is an active low signal intended to be wired directly to an active low pad output enable for the ODAT[15:0] bus, OSOC and OPRTY signals.

PRELEN[1:0]: The Prepend Length (PRELEN[1:0]) bits determine the number of prepended bytes or words in each cell. The prepended bytes/words are added in addition to the address prepend used in Slave or ANY-PHY mode. When OBUS8 is high, PRELEN can have the values 0, 1, 2, and 4, which is the number of prepended bytes. When OBUS8 is low, PRELEN can have values 0, 1, and 2, which is the number of prepended words.

HECUDF: The HECUDF bit determines whether or not the HEC/UDF octets are included in the cell received from the FIFO. When set to high (the default), the HEC and UDF octets are included. When set to low, the third word of the 27-word ATM cell is omitted and a 26-word cell (plus appended words) is transferred. If OBUS8 is high, the fifth byte of the 53-octet ATM cell is omitted and a 52-byte cell (plus appended bytes) is transferred.

INADDUDF: The INADDUDF output indicates the state of the INADDUDF bit in the configuration and status register. This bit relocates the word identifying the logical

channel in the H5/UDF field when operating in SCI-PHY slave mode. When this bit is set to high and the 8-bit cell format is used, the logical channel is identified in the H5 byte (Figure 4.1).

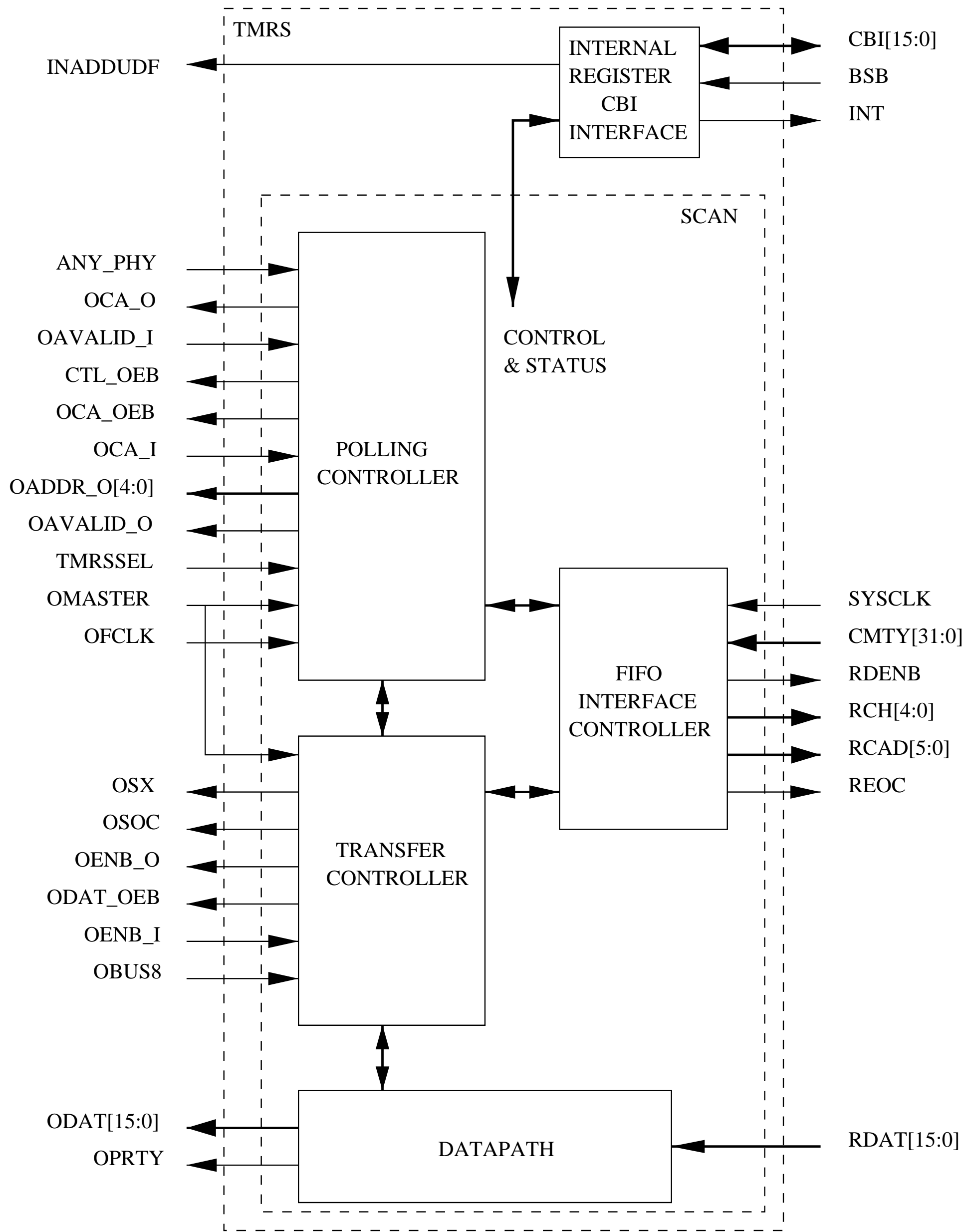

Figure 4.3 : Block diagram of the TMRS

4.1.2 Block Diagram of the TMRS

The TMRS consists of two major blocks, the CBI block and SCAN block (Figure 4.3). It consists of about 7500 gates. The CBI block is used only for the configuration and test interface and the SCAN block drives the main functionality of the TMRS. The SCAN block of TMRS consists of four main blocks, namely, Datapath, Polling controller, FIFO interface controller and Transfer controller. The Polling controller block handles the control signals related to the polling of the TMRS in the Receive Slave mode and polling the PHY devices in Transmit Master mode. The FIFO interface controller block decides if any of the 32 FIFOs have a cell available and also decides which FIFO channel should be selected. The Transfer controller block determines when the transmission of a cell starts, and the Datapath block actually handles the transmission of data.

4.1.3 SCI-PHY Receive Slave Operation

When operating in Receive Slave SCI-PHY mode, the TMRS works as a single PHY[1]. The TMRS will respond to a polling request when OAVALID_I and TMRSSEL pins are asserted. The TMRS autonomously reflects the cell availability of all FIFO channels as indicated by the CMTY[31:0] input bus. To compensate for the Multichannel Cell FIFO latency, the TMRS prefetches the first five bytes/words of the next non-empty FIFO before the next transfer occurs. So if it cannot finish the prefetch cycle before the end of the current transfer, OCA_O will not be asserted upon a polling request, even if some FIFOs are

1. The physical layer (PHY) provides for transmission of cells over a physical medium connecting two ATM devices.

non-empty. When a polling request occurs simultaneously with a cell being transferred, the TMRS will mask the CMTY line associated with the FIFO of the current transfer. Upon a polling selection, OCA_OEB signal is synchronous to the OCA_O.

A cell transfer is affected by the assertion of OENB_I. The TMRS is selected for transfer when the TMRSSEL input is high on the clock cycle prior to OENB_I assertion. The first word of the cell data received from the FIFO identifies the logical channel. Alternatively if INADDUDF field in the control register is set to one, the word identifying the FIFO channel is placed in the H5/UDF field (Figure 4.1). Upon selection, the TMRS transfers a cell from the next available FIFO channel. The TMRS services non-empty FIFO channels in a round robin fashion.

4.2 Model Checking of the TMRS Based on SCI-PHY Level 2 Protocol

The SCI-PHY protocol was defined within the SATURN Group [35] as a standardized cell-based interface between ATM layer and PHY layer devices to support single-PHY and multi-PHY applications. SCI-PHY Level 2 is an extension of SCI-PHY, that leaves all of the basic specifications and operations unchanged, but adds two important interface specifications (1) a Physical Medium Dependant (PMD) to Transmission Convergence (TC) interface specification that is compatible with all major vendors of 155 and 622 Mbit/s PMD and TC devices, and (2) an ATM Layer to Switch interface specification that provides a general purpose “extended-cell” format that will accommodate most ATM Layer implementations [35].

To be able to match the TMRS interface signals with SCI-PHY Level 2 interface signals, we based our verifications on the following environment assumptions.

1. The interface signal RCA in SCI-PHY is assumed to be the output of a buffer which its inputs are OCA_O and OCA_OEB, two interface signals of the TMRS. It is assumed that the OCA_O signal is the input of the buffer and the OCA_OEB is an active low enable signal of that buffer, as shown in Figure 4.4.

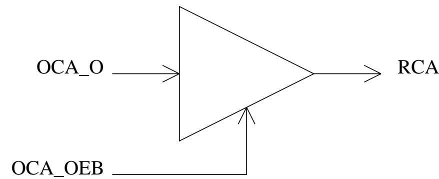

Figure 4.4 : The OCA_O and OCA_OEB signals in the TMRS, and RCA signal in SCI-PHY protocol

2. TMRSSEL input of the TMRS is assumed to be one of 32 output bits of a 5x32 decoder, which decodes RADDR[4:0] of SCI-PHY interface signal, as shown in Figure 4.5.

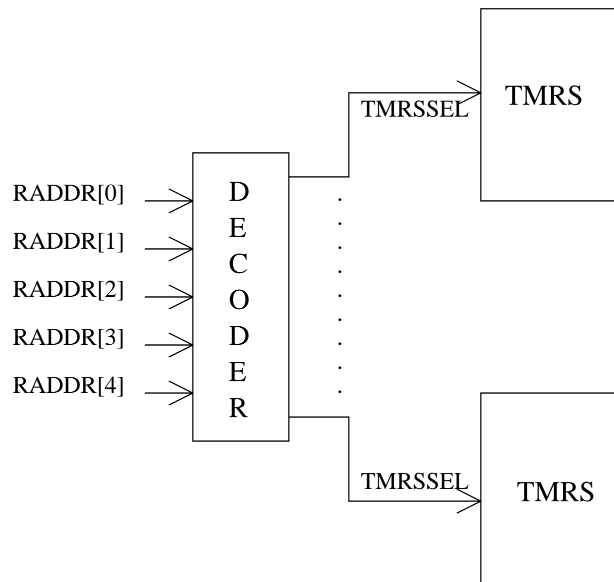

Figure 4.5 : RADDR[4:0] bus to TMRSSEL signal

Table 4.1 gives the name cross-reference between the TMRS pins when in Receive Slave SCI-PHY mode, and Receive interface signals in SCI-PHY protocol. Other TMRS interface signals are set to specific values to work in Receive Slave SCI-PHY mode. They are explained in Section 4.2.1.2.

Table 4.1: SCI-PHY Signal Cross-Reference

SCI-PHY Receive	TMRS Receive Slave
RFCLK	OFCLK
RRDENB	OENB_I
RSOC	OSOC
RCA	OCA_O, OCA_OEB
RDAT[15:0]	ODAT[15:0]
RxPRTY	OPRTY
RADDR[4:0]	TMRSSEL
RVALID	OAVALID_I

4.2.1 Environment for the SCAN Block

State space explosion is a well-known problem in FSM-based verification approaches [12]. In digital hardware designs, the state space increases exponentially with the number of latches of a design. Because model-checking algorithms are based on state space exploration, their efficiency is also limited by this phenomenon. In this section, we describe our methodology of how to deal with the state space explosion in the model checking.

The SCAN block of TMRS consists of four main blocks (Figure 4.3), namely, Datapath, Polling controller, FIFO interface controller and Transfer controller, where the Polling controller, FIFO interface controller and Transfer controller were of interest in our study. In order to make the proper environment for these three blocks and also avoid the state space explosion, we used the following techniques: model abstraction, environment

abstraction using Electronic Scissors and Reduction Seeds, and datapath abstraction using Constraints in FormalCheck.

4.2.1.1 Model and Environment Abstractions

As explained before, we were only interested in the verification of the control blocks of TMRS. To reduce the state space and speed up the verification, we tried to trim the TMRS block by eliminating the blocks that did not have any or very little effect on the three control blocks. The abstractions and reductions adopted are as following:

- As the SCAN block of the TMRS was the core to be verified, we separated this block from the rest of the TMRS design, hence, we eliminated the CBI block which is used for configuration and test purposes.
- The INPUT_MUX module was excluded from the rest of the SCAN block using Electronic Scissors in FormalCheck. Then, assumptions were made about the behavior of this component to create a proper design environment for verification of the control blocks. To do so, we modeled the outputs of this block using Constraints and State Variables. These Constraints and State Variables are expressed in Sections 4.2.1.2 and 4.2.1.3.
- The majority of the state holding registers and latches are in datapath, therefore, the DATAPATH module of the SCAN block was removed using Electronic Scissors in FormalCheck.
- Aside from the state space explosion problem, the other problem we were facing was defining proper queries. The origin of these problems were as following: (1) we had to consider two or sometimes three different specifications to be able to define one query,

(2) the signal names in the specification of the TMRS did not always match with the ones in the design. This inconsistency made it extremely difficult to observe the correct behavior of signals, (3) the specification of the TMRS was not up to date. To speed up the process of defining correct queries and as a result reducing the overall verification time, we made an abstracted model of the SCAN block using symmetry reduction technique. The abstracted model supports only 8 PHY devices. Once the verification of the queries for the abstracted model succeeded, the same queries on the model of TMRS which supports 32 PHY devices, were verified.

- In the verification of TMRS, we used the Iterated algorithm for the properties which were consuming too much of memory. Also to reduce the memory usage even more in those properties, we introduced Reduction Seeds (refer to Chapter 2, Section 2.5). By defining some of the signals as Reduction Seeds, we are reducing the explored state space for model checking and hence improving performance. Assigning Start as Input reduction value to an element will initially free it. It starts as a primary input but FormalCheck can make it active again if it is determined that the logic driving the design element cannot be pruned. The Iterated reduction algorithm was used for most of the properties, when verifying the original model of the TMRS. The reduction algorithms and Reduction Seeds used for each one of the properties are explained in Section 4.2.3.

4.2.1.2 Defining Proper Environment and Datapath Abstractions

In FormalCheck a query is made up of both properties and Constraints, where the Constraints establish the operating environment for the design. In formal verification all reachable states are explored. By adding Constraints to the properties we reduce the state space

explored in verification and therefore improve the overall performance, which means less CPU time and/or memory usage [9]. On the other hand, some of the datapath elements do not contribute to the control flow of the design, therefore, placing restrictions on the data values of these elements will reduce the reachable state space as well. In our experiment, we defined a number of Constraints as default on all of the properties used to verify the TMRS. We can classify these Constraints into two main categories.

I. The class of Constraints which establish an operating environment

- A clock Constraint on OFCLK signal, starting with high for one crank[1] and then low for one crank.
- A Clock Constraint on SYSCLK signal, the same as OFCLK.
- A Reset Constraint on RST signal, starting with high for two cranks and then goes to low forever.
- A Group Constraint named SCIPHY_Slave. This group is used to drive the TMRS to SCI-PHY slave mode. The two signals under this group, ANYPHY and OMASTER signals, are set to low.
- A Group Constraint named STABLE_INPUTS. This Constraint was defined to create a suitable environment for the TMRS. By adding this Constraint the input signals to TMRS will stay stable for the whole duration of the OFCLK and they only change with the rising edge of OFCLK. The signals included in this group are as following:

  1. CMTY[31:0] (or CMTY[7:0] in the abstracted model of the TMRS)

---

1. In FormalCheck, a *crank* is considered as the propagation delay of a flip-flop. Since there is no concept of the absolute time, FormalCheck uses a crank as the unit of time and observes the events relative to this unit.

2. OENB_I input signal

3. TMRSSEL input signal

4. OAVALID_I. We have defined two Constraints on this signal: (1) OAVALID_I has to be stable during one clock cycle to meet the required timing limits. (2) Based on SCI-PHY Level 2 standard, "*To avoid contention while polling, the ATM layer device shall present a valid address no more than once every other RFCLK period. It is recommended that RADDR[4:0] be set to 0x1F when invalid. When supporting 32 PHY links, RAVALID is asserted simultaneously with a valid address on RADDR[4:0]*" [35]. That means in the TMRS, the OAVALID_I signal should not stay high for two consecutive clock cycles. This is the second Constraint for this signal.

- Since the INPUT_MUX block was eliminated from the design by the Electronic Scissors, we had to use Constraints on the outputs which where used by the control blocks of the SCAN block. These general Constraints are as following:

1. We had to define two State Variables, which we named Oavalid_I_Reg and TMRSSEL_Reg (refer to Section 4.2.1.3), to model the behavior of INPUT_MUX block. Then, we used two general Constraints, which were named Oavalid_I_Reg and TMRSSEL_Reg as well, to map the State Variables to the corresponding output signals of the INPUT_MUX block.

2. OMASTER_Noreg, OBUS8_Noreg, and ANYPHY_Noreg. These Constraints simply connect the Omaster, Obus8, and Anyphy input signals, to the Omaster_Noreg, Obus8_Noreg, and Anyphy_Noreg (the outputs of the INPUT_MUX block), respectively.

II. The class of Constraints which perform datapath abstractions

- A Group Constraint named Default_Mode. This group is limiting the TMRS to work in the 8-bit mode and to transfer cells with the minimum length possible. The constrained signals in this group are:

  1. HECUDF set to high.
  2. INADDUDF set to low.
  3. OBUS8 set to high.
  4. PHYDEV set to 31 (set to 7 for the abstracted model of the TMRS).
  5. PRELEN set to "00".

4.2.1.3 State Variables

State Variables can be used as memory elements to either capture a sequence of events or just to be used as a latch. In our experiment, the two State Variables, Oavalid_I_Reg and TMRSSEL_Reg, were defined to model the behavior of two outputs of the INPUT_MUX block. A third State Variable, WasSelected, was used to detect the selection of the TMRS by master right after completing a cell transfer, when it is not in back-to-back transfer mode. These State Variables are defined as following.

- ```
  Oavalid_I_Reg: Range 0 to 1          Initial: 0
  If (not @RstDone) then Oavalid_I_Reg := 0;
  elsif ((@CLKrising) and (@RstDone)) then Oavalid_I_Reg := Oavalid_I;
  else Oavalid_I_Reg := Oavalid_I_Reg;
  end if;
  ```
- ```
  TMRSSEL_Reg: Range 0 to 1            Initial: 0
  ```

```
If (not @RstDone) then TMRSSEL_Reg := 0;
elsif ((@CLKrising) and (@RstDone)) then TMRSSEL_Reg := Tmrssel;
else TMRSSEL_Reg := TMRSSEL_Reg;
end if;
```

- `WasSelected:  Range 0 to 1           Initial: 0`

```
if ((@TMRSselected) and (@CLKrising) and (@RstDone))
then WasSelected := 1
elsif ((@TransferDone) and (Oenb_I = 1) and (@WaitSelectedState))
then WasSelected := 0
else WasSelected := WasSelected;
end if;
```

4.2.2 Expression Macros

Often, the same complex expression is needed in several behaviors. FormalCheck supports expression reuse through Macros. Any expression that is given a user supplied name becomes a Macro, and is globally available [5]. We created some expression Macros inside FormalCheck to make the properties more comprehensive. The following lists the Macro names and also their contents[1].

1. @*CLKrising:* scan:OFCLK = rising
2. @*FIFOnotEmpty:* scan:Polling_Sm_Inst:Allfifoempty = 0
3. @*PrefetchDone:* scan:Transfer_SM_Inst:Prefetch_Not_Rdy = 0
4. @*RCAhigh:* (scan:Oca_O = 1) and (scan:Oca_Oeb = 0)

1. The @ sign in front of a Macro shows that this is a Macro not a signal.

5. @*RstDone:* scan:Rst = 0

6. @*TMRSpolled:* (scan:Tmrssel = 1) and (scan:Oavalid_I = 1)

7. @*TMRSselected:*: (scan:Oenb_I = 0) and (scan:Transfer_Sm_Inst:Oenb_I_ Reg = 1) and (scan:Tmrssel_Reg = 1)

8. @*TransferDone:*: (scan:Transfer_Sm_Inst:Rec_Cpt = scan:Transfer_Sm_Inst:Cell_Length)

9. @*Latch4State:* scan:Transfer_Sm_Inst:Slave_State = Sms_Latch4

10. @*StartTxState:* scan:Transfer_Sm_Inst:Slave_State = Sms_Start_Transf

11. @*TMRSselectedLastCC:* ((scan:Transfer_Sm_Inst:Tmrssel_Reg2 =1) and (scan:Transfer_Sm_Inst:Oenb_I_Reg2 = 1)) and (scan:Transfer_Sm_Inst:Oenb_I_Reg = 0)

12. @*WaitSelectState:* scan:Transfer_Sm_Inst:Slave_State = Sms_Wait_Sel

13. @*IncCptState:* scan:Transfer_Sm_Inst:Slave_State = Sms_Inc_Cpt

14. @MiddleOfTx: (scan:Transfer_Sm_Inst:Rec_Cpt /= scan:Transfer_Sm_Inst:Cell_Length)

4.2.3 Properties

After establishing a proper environment, we consider 13 queries of the TMRS, including liveness and safety properties. Each query contains only one property and the Constraints are all global, except for one (which will be explained on the spot). The following properties have been defined using the SCI-PHY Level 2 protocol, the specification of the TMRS, and the test benches already written to test this design. Usually a test plan can help

defining proper queries, but since there was not a test plan written for this design we tried to take advantage of the test benches. The comment lines in the test benches helped us understand the properties that the verification engineer had in mind to perform functional simulation.

Table 4.2 shows a brief description of the properties established to verify the Receive Slave SCI-PHY mode of the TMRS, the sources of these properties, and the status of the TMRS (Polled, Selected, Cell Transfer, etc.). In this section, we only talk about the properties for which we performed Property Decomposition or Vertical Verification techniques on them. Experimental results of the verification of these properties for the abstracted and complete TMRS models are summarized in Tables 4.3 and 4.4, respectively. To obtain a complete description of all properties and the reduction techniques used for each one, refer to Appendix A.

Table 4.2: Properties of Receive Slave SCI-PHY mode of the TMRS

Property	Source(s)	Brief Description of Property	TMRS Status
Property_1	SCI-PHY Doc. Test bench	OENB_I = 0 & TMRS Selected then OSOC = 1 & @StartTxState	Selected
Property_1A	SCI-PHY Doc. Test bench	TMRSselectedLastCC & @WaitSelectState then @StartTxState	Selected
Property_1B	SCI-PHY Doc. Test bench	OENB_I = 0 & TMRS Selected & (@Latch4State or @WaitSelectState) then OSOC = 1 & @WaitSelect-State	Selected
Property_2	SCI-PHY Doc. TMRS Spec.	If TMRS is polled and all CMTY[31:0] lines are high or the prefetch of the next FIFO is not finished, OCA_O will be deasserted	Polled
Property_3	TMRS Spec.	In Back To Back Transfer Mode: @TransferDone & OENB_I = 0 & AllFifoEmpty = 1 then ODAT_OEB = 1	Selected
Property_4	SCI-PHY Doc.	No PHY shall drive RCA (OCA_O) upon sampling RAVALID (OAVALID_I) low	ANY

Table 4.2: Properties of Receive Slave SCI-PHY mode of the TMRS

Property	Source(s)	Brief Description of Property	TMRS Status
Property_5	SCI-PHY Doc. TMRS Spec.	@TMRSpolled & (more cell available to transfer & @PrefetchDone) then OCA_O = 1	Polled
Property_5A	TMRS Spec.	If there is a cell available to transfer then Eventually @PrefetchDone	ANY
Property_5B	SCI-PHY Doc. TMRS Spec.	@TMRSpolled & (assuming @PrefetchDone) then OCA_O = 1	Polled
Property_6	TMRS Spec.	In Back To Back Transfer Mode: @TransferDone & OENB_I = 0 & New_Cell_Transf_Rdy = 1 then Eventually OSOC = 1 & @StartTxState	Selected
Property_6A	TMRS Spec.	In Back To Back Transfer Mode: @TransferDone & OENB_I = 0 & New_Cell_Transf_Rdy = 1 then Eventually @StartTxState	Selected
Property_6B	TMRS Spec.	In Back To Back Transfer Mode: @TransferDone & OENB_I = 0 & New_Cell_Transf_Rdy = 1 then Eventually OSOC = 1	Selected
Property_7	SCI-PHY Doc. TMRS Spec. Test bench	TMRS always expects a complete cell transfer but supports transfer interruption by deassertion of OENB_I	Cell Transfer
Property_8	SCI-PHY Doc.	If master selects the TMRS before the prefetch cycle is done, the TMRS will not start a cell transfer in the next C.C.	Selected
Property_9	SCI-PHY Doc. TMRS Spec.	When operating in the Slave mode the TMRS monitors the input signal OENB_I to validate the data transfer	Cell Transfer
Property_10	SCI-PHY Doc. TMRS Spec.	TMRS Selected & OENB_I = 0 then ODAT_OEB = 0	Selected
Property_11	TMRS Spec.	TMRS not Selected & @TransferComplete then ODAT_OEB = 1	Transfer Done

Property_1: According to the SCI-PHY protocol: “When RRDENB is sampled low by the PHY layer device, the RSOC signal will be accepted by the ATM layer device on the next rising edge of RFCLK” [35]. We expressed this property for the TMRS as following: “If OENB_I is sampled low by the PHY layer device, and the TMRS is selected (TMRS-SEL and OENB_I signals were sampled high in the previous clock cycle), on the next ris-

ing edge of OFCLK the transmission of a full cell will start and the OSOC signal will be set to high”. In FormalCheck, this property is expressed as follows.

```
Property: property_1
Type: Always
After: (@TMRSselected) and (@WaitSelectedState) and (@CLKrising)
Always: OSOC = 1 and @StartTxState
Options: Fulfill Delay: 0  Duration: 1  Count of @CLKrising
```

Tables 4.3 and 4.4 report that the verification of Property_1 was “Terminated”. Based on the specification of the TMRS “the START_TRANSF state, starts the transfer by asserting OSOC signal”, therefore, Property_1 expects the FSM to move to START_TRANSF state and the OSOC signal to be set to high in the same clock cycle. By examining the “verify.out” file, it was observed that no transition was enabled for this property during the verification process. This means the Enabling Condition was irrelevant to the Fulfilling Condition. The Transfer_Slave state machine is actually made of two parallel processes (state machines), one of them generates the OSOC as well as other control signals, and the other one determines the state of the Transfer_Slave state machine in the next clock cycle. To discover the cause of termination, this property was decomposed to the following two properties, Property_1A which checks for the state transition to START_TRANSF state, and Property_1B which checks for the assertion of the OSOC signal. These two properties are stated as following.

Property_1A: If the TMRS is in the WAIT_SEL state and the TMRS was selected in the previous clock cycle (TMRSSEL_REG2 and OENB_I_REG2 signals are sampled high),

and OENB_I signal was sampled low in the previous clock cycle (OENB_I_REG is low), on the next rising edge of OFCLK the TMRS will be in START_TRANSF sate. In FormalCheck, this property is expressed as follows.

```
Property: property_1A
Type: Always
After: (@TMRSselectedLastCC) and (@WaitSelectState) and (@CLKrising)
Always: @StartTxState
Options: Fulfill Delay: 0  Duration: 1  Count of @CLKrising
```

Property_1B: If the TMRS is in the LATCH4 or WAIT_SEL states and OENB_I signal is sampled low and the TMRS is selected (TMRSSEL_REG and OENB_I_REG signals are sampled high), on the next rising edge of OFCLK the transmission of a full cell will start (the OSOC signal will be set to high) and the TMRS will be in WAIT_SEL sate. In FormalCheck, this property is expressed as follows.

```
Property: property_1B
Type: Always
After: (@TMRSselected) and (@Latch4State or @WaitSelectState) and
       (@CLKrising)
Always: Osoc = 1 and @WaitSelectState
Options: Fulfill Delay: 0  Duration: 1  Count of @CLKrising
```

The verification of Property_1A and Property_1B revealed the origin of the problem, which caused terminating the verification of Property_1. The Transfer_Slave state machine proceeds to START_TRANSF state one clock cycle after the OSOC signal (Output Start Of Cell) is asserted. Therefore, the OSOC signal and START_TRANSF state are not synchronous.

Property_3: In Receive Slave SCI-PHY back-to-back transfer mode, when the external master device does not deassert OENB_I at the end of a transfer, the TMRS remains selected for another cell transfer. If all FIFOs are empty the TMRS will deassert ODAT_OEB and will wait to be reselected [34]. In FormalCheck, this property is expressed as follows.

```
Property: property_3
Type: Always
After: (@TransferDone) and (Oenb_I = 0) and
       (Polling_Sm_Inst: Allfifoempty = 1) and (@RstDone)
Always: Odat_Oeb = 1
Options: Fulfill Delay: 0 Duration 1 count of @CLKrising
```

It is observed from Tables 4.3 and 4.4, that the verification of Property_3 failed in both models. This means even after a cell transfer is done and there are no more cells to transfer, ODAT_OEB will still be asserted. This problem was taken into consideration by the designer of the TMRS and was fixed. In the new version of the design, when in the back-to-back transfer mode, after completing a cell transfer the ODAT_OEB will stay asserted and the data on the ODAT bus will be the byte/word which was transferred last. The master is capable of determining the end of a cell (by counting the number of bytes/words it has received), so the master waits for the assertion of the OSOC signal (from the TMRS).

Property_5: According to the SCI-PHY protocol: “Each PHY link shall have a unique address corresponding to a value between 0 and 31. Upon sampling its address with the rising edge of the RFCLK, a PHY must drive RCA to indicate whether it has an entire cell

in its buffer" [35]. In TMRS this property is expressed as following: "When Oavalid_I and TMRSSEL are asserted (TMRS is polled), OCA_O is driven to high in the next clock cycle, if at least one of the CMTY[31:0] status bus is low and the prefetch of the next FIFO is finished" [34]. In FormalCheck, this property is expressed as follows.

```
Property: property_5
Type: Always
After: (@TMRSpolled) and (@FIFOnotEmpty) and (@PrefetchDone) and
       (@CLKrising) and (@RstDone)
Always: @RCAhigh
Options: Fulfill Delay: 0 Duration: 1 Count of @CLKrising
```

In Property_5, the Polling and Transfer controllers are interacting blocks. The Transfer controller uses the internal output of the Polling controller to determine whether a cell is available, and asserts its internal output to the Polling controller when the prefetch cycle of that cell is done. When the TMRS is polled by the master, the Polling controller uses the internal output of the Transfer controller to assert/deassert the RCA signal. The vertical verification described in Section 3.3 could be applicable for this property in case of state explosion. To verify Property_5 using sub-properties, we need to add the following State Variable and Constraints.

1. A State Variable Named New_Cell_Available. This State Variable becomes 0 when there are no more cells (besides the one that is currently being transferred) available, and 1 when there is at least one more cell to transfer. This State Variable was defined to be able to describe the New_Cell_Tx_Rdy Constraint. It will be explained more later.

2. Since Property_5 is verified under the condition that there exists a cell to transfer, a Constraint named Channel_Not_Empty has to be defined. This Constraint makes sure that there is a FIFO that has a cell to transfer (besides the cell that is currently being transferred).
3. A Constraint to assume the property of Polling_Sm block (this Constraint is verified as Property_6A1). This Constraint is defined as following:

```
Constraint: New_Cell_Tx_Rdy

Type: Always

After: New_Cell_Available = 1 and @CLKrising

Assume Always: Polling_Sm_Inst:New_Cell_Transf_Rdy = 1

Unless: New_Cell_Available = 0

Options: (None)
```

In FormalCheck when defining the Fulfill Delay to slide the verification window (refer to Chapter 2), it is mandatory to define the Duration of the Verification Window as well. New_Cell_Transf_Rdy signal becomes high after two clock cycles and, it stays high unless there are no more cells available to transfer. Since the duration of the New_Cell_Transf_Rdy signal being high is not known, we could not use the Fulfill Delay option, therefore, we defined the State Variable New_Cell_Available to specify the behavior of New_Cell_Transf_Rdy signal. Now we can define the property of Transfer controller (Property_5A) as following.

Property_5A: Prefetch of a cell will eventually be done, assuming there is at least one cell available and the New_Cell_Transf_Rdy is asserted (the assertion of New_Cell_Transf_Rdy happens two clock cycles after a cell is available).

```
Property: property_5A

Type: Eventually

After: (@CLKrising) and (@RstDone)

Eventually: @PrefetchDone

Options: (None)
```

After verifying the property of Transfer controller block (Property_5A), this property is assumed to verify the property of Polling controller (Property_5B). Therefore, turning Property_5A to a Constraint we have:

```
Constraint: PrefetchIsDone

Type: Eventually

After: (Polling_Sm:New_Cell_Transf_Rdy = 1) and (@CLKrising) and (@RstDone)

Assume Eventually: @PrefetchDone

Unless: Polling_Sm:New_Cell_Transf_Rdy = 0

Options: (None)
```

Then, assuming the PrefetchIsDone Constraint, Property_5B is defined as following.

Property_5B: When Oavalid_I and TMRSSEL are asserted, OCA_O is driven to high in the next clock cycle, if at least one of the CMTY[31:0] status bus is low and assuming the prefetch of the next FIFO is finished. In FormalCheck, this property is expressed as follows.

```
Property: property_5B

Type: Always

After: (@TMRSpolled) and (@FIFOnotEmpty) and (@CLKrising) and (@RstDone)

Always: @RCAhigh

Options: Fulfill Delay: 0 Duration: 1 Count of @CLKrising
```

The verification of Property_5 was done with no state space explosion problem. However for experimental purposes, Properties 5A and 5B were verified for the abstracted model of the TMRS. As we can see, the application of the vertical verification on Property_5 was possible, because of the one way interdependencies between the Polling and Transfer controller blocks in this case.

Property_6: Based on the TMRS specification, "in Receive Slave SCI-PHY back-to-back transfer, when the external master device does not deassert OENB_I at the end of a transfer, the TMRS remains selected for another cell transfer" [34]. Hence, when the external master does not deassert OENB_I at the end of a transfer, and at least one of the FIFOs have a cell to transfer, the TMRS will eventually start transmitting the new cell.

We acquired from Property_1, that the OSOC signal and START_TRANSF state of the Transfer_Slave state machine are not synchronous. Accordingly, we had to decompose Property_6 to two properties to be able to check for both the START_TRANSF state of the FSM (Property_6A) and the OSOC signal (Property_6B), while in back-to-back transfer mode. The queries of these two properties contain another Constraint, named BackToBackTx, in addition to the default Constraints explained in Section 4.2.1.2. This Constraint puts the TMRS in the back-to-back transfer mode. Also to make matters easier, we assumed no interruptions from the master will occur while transferring a cell, as well as after the completion of a cell transfer (since a cell transfer with interruptions from the master was verified through other queries, this assumption is considered safe). The BackToBackTx Constraint is expressed in FormalCheck as follows.

```
Constraint: BackToBackTx

Type: Always

After: (@IncCptState) and (@CLKrising)

Assume Always: Oenb_I = 0

Unless: (@StartTxState) or (@WaitSelectState)

Options: (None)
```

In FormalCheck, Propery_6A and Property_6B are expressed as following.

Property_6A:

```
Property: property_6A

Type: Eventually

After: (@TransferDone) and (Transfer_Sm_Inst:New_Cell_Transf_Rdy = 1) and
       (@Clkrising)

Eventually: @StartTxState

Options: (None)
```

Property_6B:

```
Property: property_6B

Type: Eventually

After: (@TransferDone) and (Transfer_Sm_Inst:New_Cell_Transf_Rdy = 1) and
       (@Clkrising)

Eventually: (Osoc = 1) and (@LatchNext2State)

Options: (None)
```

The relationship between the Polling controller and Transfer controller in Property_6A and Property_6B is linear, therefore, the vertical verification (refer to Section 3.3) could be

applied in case of state explosion. To do so, each one of these properties can easily be divided to two sub-properties, one of which will be the property of Polling controller block (Property_6A1 and Property_6B1), and the other one the property of Transfer controller block (Property_6A2 and Property_6B2). These properties are explained as following.

Property_6A1:

```
Property: property_6A1
Type: Always
After: (New_Cell_Available = 1) and (@Clkrising)
Always: Polling_Sm:New_Cell_Transf_Rdy = 1
Unless: New_Cell_Available = 0
Options: (None)
```

Property_6B1 is exactly the same as Property_6A1, therefore we avoid rewriting this property. Now assuming there is a cell available in the FIFO (Constraint Channel_Not_Empty) and assuming Property_6A1 (Constraint New_Cell_Tx_Rdy), Property_6A2 and Property_6B2 should be verified.

Property_6A2:

```
Property: property_6A2
Type: Eventually
After: (@TransferDone) and (@Clkrising)
Eventually: @StartTxState
Options: (None)
```

Property_6B2:

```
Property: property_6B2

Type: Eventually

After: (@TransferDone) and (@Clkrising)

Eventually: (Osoc = 1) and (@LatchNext2State)

Options: (None)
```

As we can see applying the vertical verification when the blocks have linear dependencies, is very straightforward. Properties 6A1 (6B1), 6A2, and 6B2 were verified on the abstracted model of the TMRS.

4.2.4 Experimental Results

All verifications in this work were executed on a HP9000 (440MHz) with 6144 MB RAM and HP-UX11 operating system. For the rest of this chapter, we will call the model of the TMRS which supports 32-PHY devices, the *original model*, and the model of the TMRS which supports 8 PHY devices, the *abstracted model* to simplify the text. As mentioned in Section 4.2.1.1, we first designed an abstracted model of the TMRS (which supports 8 PHY devices) and verified it. After completing the verification of the abstracted model of the TMRS, we verified the original model of the TMRS using reduction techniques. The experimental results are shown in Tables 4.3 and 4.4, respectively. These tables include the reduction algorithm used for each query, the result (status) of the verification, the number of reached states, the number of states in the model, the average state coverage, the CPU time (real time) in seconds, and the memory usage in megabytes.

As shown in Tables 4.3 and 4.4, the 1-Step reduction algorithm was used for the verification of Property_3 in both models. The reason for that is, when the Iterated algorithm was used the result of the verification was "Terminated", but with 1-Step algorithm the verification would finish with no problem. This shows that the Iterated algorithm is not powerful enough to reduce the model efficiently for all kinds of queries, and in that case the user has no choice but using the 1-Step algorithm. It must be added here that, if a query is verified using the Iterated algorithm, the verification result is guaranteed to be valid.

Properties 1A, 1B, 2, 4, 5, 6A, 6B, and 7 consume less CPU time and memory for the original model compared to the abstracted model. The reason for that is, we used the Iterated algorithm and Reduction Seed for the original model, whereas for the abstracted model the default (1-Step algorithm) was used. Therefore, the reduction algorithm used for each property is an important key to reduce the CPU time and the memory usage. Also we can see that if the properties are verified under the same conditions, by increasing the number of PHY devices supported by the TMRS, the number of State Variables and the memory usage will increase. The CPU time for property checking is related to different parameters such as memory usage, number of State Variables, and the reduction method used for that specific property. Feedbacks from the internal blocks can increase the CPU time also.

Table 4.3: Verification results of model checking on the abstracted model of TMRS

Properties	Reduction Algorithm	Status	States Reached	State Variables	State Var. Avg. Coverage	Real Time (seconds)	Memory Usage (MB)
Property_1	Iterated	Terminated	N/A	N/A	N/A	N/A	N/A
Property_1A	1-Step	Verified	5.09e+09	114	97.37%	493	34.61
Property_1B	1-Step	Verified	5.09e+09	114	97.37%	189	34.59
Property_2	1-Step	Verified	5.09e+09	114	97.81%	126	34.73
Property_3	1-Step	Failed	3.28e+08	115	95.22%	160	36.23
Property_4	1-Step	Verified	5.09e+09	114	97.81%	84	34.20
Property_5	1-Step	Verified	5.09e+09	115	97.39%	256	36.82
Property_5A	1-Step	Verified	1.81e+06	55	93.64%	32	22.93
Property_5B	1-Step	Verified	9.58e+05	42	96.43%	21	22.12
Property_6	1-Step	Terminated	N/A	N/A	N/A	N/A	N/A
Property_6A	1-Step	Verified	3.83e+08	114	98.25%	161	29.90
Property_6A1	1-Step	Verified	1.32e+06	42	97.62%	16	22.14
Property_6A2	1-Step	Verified	3.31e+05	56	93.75%	20	22.93
Property_6B	1-Step	Verified	3.83e+08	114	98.25%	256	29.51
Property_6B1	1-Step	Verified	1.32e+06	42	97.62%	16	22.14
Property_6B2	1-Step	Verified	3.31e+05	57	93.86%	31	22.94
Property_7	1-Step	Verified	5.09e+09	114	98.25%	208	34.74
Property_8	1-Step	Verified	5.09e+09	114	97.37%	34	5.27
Property_9	1-Step	Verified	5.45e+09	115	97.39%	23	5.27
Property_10	1-Step	Verified	4.61e+03	20	97.50%	18	21.68
Property_11	1-Step	Verified	5.14e+09	116	97.41%	136	34.63

Table 4.4: Verification results of model checking on the original model of TMRS

Properties	Reduction ALgorithm	Status	States Reached	State Variables	State Var. Avg. Coverage	Real Time (seconds)	Memory Usage (MB)
Property_1	Iterated	Terminated	N/A	N/A	N/A	N/A	N/A
Property_1A	Iterated	Verified	2.93e+18	195	97.95%	29	7.22
Property_1B	Iterated	Verified	2.94e+18	195	97.95%	31	7.22
Property_2	Iterated	Verified	4.12e+25	193	99.48%	21	7.21
Property_3	1-Step	Failed	1.76e+18	251	97.41%	10370	126.79
Property_4	Iterated	Verified	2.93e+18	196	98.21%	76	42.50
Property_5	Iterated	Verified	2.93e+18	197	97.97%	73	38.87
Property_6	Iterated	Terminated	N/A	N/A	N/A	N/A	N/A
Property_6A	Iterated	Verified	2.06e+17	195	98.46%	86	42.03
Property_6B	Iterated	Verified	2.06e+17	195	98.46%	92	43.16
Property_7	Iterated	Verified	2.93e+18	195	98.46%	73	42.50
Property_8	Iterated	Verified	2.93e+18	195	97.95%	72	37.79
Property_9	Iterated	Verified	3.09e+18	196	97.96%	69	38.33
Property_10	Iterated	Verified	2.93e+18	195	98.21%	75	38.96
Property_11	Iterated	Verified	2.94e+18	196	98.21%	380	38.90

4.2.5 Further Errors Found

In our study and during the verification process, we found several errors in the specification of the TMRS. These errors are not only related to the Receive Slave SCI-PHY mode of TMRS, but also to the other modes such as Transmit Master for SCI-PHY/ANY-PHY and Receive Slave ANY-PHY modes of operation. These errors were found by property checking and also by code inspection. The errors are listed as following:

- The following fundamental mismatch was found between the specification of the TMRS [34] and the SCI-PHY protocol [35].

  According to the SCI-PHY protocol "To ensure backwards compatibility with single-

PHY devices, the PHY for which a cell transfer is in progress shall not be polled until completion of the cell transfer" [35]. The TMRS was basically designed to be polled while transmitting a cell, so naturally it is not compatible with single-PHY devices. In Receive Slave SCI-PHY mode the TMRS relies on the master device. If the master device is not compatible with single-PHY devices and it polls the TMRS during a cell transfer, the TMRS will reply and it will not care about being backwards compatible with single-PHY devices.

- In the Transmit Master SCI-PHY/ANY-PHY mode the following error in the specification of the TMRS [34] was found.

  The Master Transfer state machine is missing the "Prefetch #3 State" and "Prefetch_Next_FIFO 3" states.

- In the Receive Slave SCI-PHY/ANY-PHY mode the following eight errors in the specification of the TMRS were found.

  - The Slave Transfer state machine is missing "Latch_3", "Latch_4", and "Prefetch_Next_FIFO _3" states.

  - The Slave Transfer state machine shows while in "Empty_Pipe" state,

    ```
    if new_cell_transfer_rdy = 0 => Next_State = Load_Pipe_Cache_Pre_State
    ```

    whereas in the TMRS design we have,

    ```
    if new_cell_transfer_rdy = 1 => Next_State = Load_Pipe_Cache_Pre_State
    ```

  - The Slave Transfer state machine shows while in "Empty_Pipe" state,

    ```
    if transfer_end = 0 => Next_State = Wait_State
    ```

    whereas in the TMRS design we have,

    ```
    if transfer_end = 1 => Next_State = Wait_State
    ```

- The Slave Transfer timing diagram for SCI-PHY and ANY-PHY are missing “Pre #3”, “Latch #3”, and “Latch #4” states.

- The Slave Transfer state machine shows, while in “Start_Trans” state,

```
if ANY-PHY = 0 and Transf_En = 1 => Next_State = INC_CPT_State
```

but this FSM does not specify the case of `ANY-PHY = 1`.

- The Slave Transfer timing diagram (ANY-PHY) is showing “Asser_OSOC” state, which does not exist in the TMRS design.

- The Slave Transfer timing diagram (ANY-PHY) shows that the OSX signal is asserted while in “Start_Transfer” state. In reality the OSX signal is asserted while in “Wait_Sel” state.

- The Slave Transfer timing diagram (ANY-PHY) shows that the OSOC signal is asserted while in “Asser_OSOC” state. In reality this signal will be asserted while in “Start_Transfer” state.

• In the Icon specification of the TMRS [34], we found the following errors:

- The CBI[15:0] bus is shown as an input bus, whereas it is an I/O bus.

- ODAT[15:0] data bus is shown as I/O bus, whereas it is actually an output data bus.

- Signals SCAN_IN and SCAN_EN are shown as input signals to the SCAN block and also SCAN_OUT is shown as an output signal from this block. These signals are defined in the code of the TMRS, but they have never been used inside the SCAN block.

All of the suggested modifications over the specification of the TMRS, which were

specified in this section, were considered acceptable by the designer of the TMRS, and the specification was revised to reflect these corrections.

4.3 Conclusions

In this study, we explored the model checking for the Receive Slave SCI-PHY mode of operation of the TMRS TSB. The main contributions of this work are (1) application of the verification approach described in Chapter 3 on a real size design, (2) the establishment of the abstracted model of the TMRS, (3) the definition of a suitable environment inside FormalCheck for the TMRS interface signals, (4) the abstraction of the model using Electronic Scissors in FormalCheck, (5) the definition of a set of properties on the TMRS in FormalCheck, (6) the application of reduction techniques and Reduction Seeds to the model to reduce the state space, (7) the verification of the original model of the TMRS, (8) the discovery of several mismatches between the TMRS design, its specification, and the SCI-PHY protocol, and (9) the uncovering of some errors in the specification of the TMRS.

FormalCheck provides various algorithms to perform verification, such as "Symbolic State Enumeration" (using ordered Binary Decision Diagrams or BDDs), "Explicit State Enumeration" and "Auto Restrict" options [5]. To verify large circuits and to avoid the state space explosion there are several techniques to use: (1) choosing the suitable run option can reduce the run time when verifying large circuits, (2) using a suitable reduction method and reduction seed in FormalCheck can reduce the state space. The only drawback to using the reduction seed is that the person who is verifying the design has to know the design well enough to be able to introduce reduction seeds. The danger behind over con-

straining a design is false verification results (3) If the automatic reduction techniques fail in reducing the state space, an abstracted model of the circuit needs to be developed.

Using the Iterated reduction algorithm in FormalCheck does indeed make the verification much faster than the 1-Step algorithm and the verification results are guaranteed to be valid for the entire design. The only problem with this algorithm is that it does not seem to be able to reduce the design efficiently for all kinds of queries. Using this algorithm for some queries can cause the verification result to be "Terminated", whereas using the 1-Step algorithm, even though in a longer time, will proceed to complete the verification of the same queries (provided having enough resources to avoid state space explosion).

Table 4.5: Detailed time frame of the verification of the TMRS case study

Verification phases	**Time (week)**
Reading documents (SCI-PHY, UTOPIA, and TMRS Spec.) for model checking purposes	2.5
Reading the code for model checking purposes	3
Verifying FSM and Timing Diagrams	2
Making the abstract model of TMRS	0.5
Defining properties and verifying the abstract model	3
Verifying the original model	2
Total (model checking)	13

Human time is a very important factor in verification. In this study as shown in Table 4.5, a lot of time was spent for understanding the specifications to define the right queries and also a lot of time was spent for understanding the implementation to be able to apply the features of FormalCheck like Reduction Seeds, Constraints and State Variables. Since

the designer has a thorough knowledge of the design and the specification, assuming he/she is trained to use FormalCheck, it would take him/her only 2-3 weeks to define the queries and formally verify this design. According to the designer of the TMRS, the RTL model was made in 4 weeks, while writing test benches and running simulation took about 3 months.

The "Back Reference" feature in FormalCheck can help the designer debug the problems found during verification. The trick to finding these problems is to isolate which sequence of events caused the erroneous behavior. This is time-consuming with simulation, because it is difficult for designers to sort through all the events and determine which ones matter. FormalCheck can automatically find a minimal sequence of events that cause the error, when a user asserts a property that expresses that the error condition should never occur.

Chapter 5

Conclusion and Future Work

The aim of model checking is to obtain an economic advantage by catching bugs reliably and early in the design process. Although automatic formal verification is reliable in catching bugs, the state space explosion problem limits its use. In order to increase the effectiveness of the verification process, model checking and simulation should complement each other. Model checking shines when verifying state machine oriented (control logic) modules and protocols, while HDL simulators are good for data intensive designs (such as ALUs, adders, and multipliers) and datapath analysis.

This thesis practices a practical verification approach that is easy to use and safe to apply. The contributions of this work are summarized as follows:

- The validity of the hierarchical verification methodology was demonstrated by verifying a real size integrated circuit using FormalCheck. In this practice the queries were verified in reasonable amount of time, for both the abstracted model and the whole design.

- The manual reductions can be error prone. Therefore, the verification results of a manually abstracted model of a design are not 100% reliable. In our experiment, the state space explosion problem was avoided during verification of the original design by using the automatic reduction algorithms, introducing reduction seeds, and environment abstraction using facilities provided by FormalCheck. Since, no manual reduction techniques were used while verifying the original model, we can claim that the verification results are guaranteed to be correct.

- Usually, the time spent for verification (including simulation and formal verification) of a product is as much, or may be more than the time spent to design it. Therefore, one of the concerns of the companies is to find a way to reduce the time spent on verification while delivering a reliable product. In this work, several fundamental changes in the design flow were proposed in Chapter 3, which are capable of accelerating the verification process.

- One essential difference between the approach proposed in this thesis and the one applied in [44] is that, the authors constructed a non-deterministic model of the environment in an HDL language to mimic the normal operating environment. Their solution actually results in adding to the whole reachable state space, which we are trying to avoid. The proposed verification approach suggests using the facilities provided in FormalCheck (such as Constraints, and if necessary State Variables) to define the proper operating environment. In the verification of the TMRS a suitable operating environment was defined for the design model using mostly Constraints and only three State Variables. This method reduces the reachable state space instead of adding to it.

- Another difference between the approach proposed in this thesis and the one presented in [44] is that the authors used State Variables to make the formulation of the properties easier. They used State Variables to memorize the value of the signals in the previous clock cycle, instead of using Enabling Condition. Each extra State Variable doubles the reachable state space, therefore usage of State Variables should be avoided unless absolutely necessary. In the verification of the TMRS Enabling Condition, Fulfill Delay option, and Macros were used to make properties easier to formulate and easier to understand.

- We also suggested a semi-formal verification approach which exploits the positive features of simulation and model checking to achieve better test coverage of designs. Considering the fact that datapaths and computational intensive designs tend to cause state space explosion, and in a lot of instances the data values do not affect the control of the design, hence, simulation is better suited for datapath analysis and computation results analysis. On the other hand, model checking is most effective when used for the verification of control-oriented designs such as control logic, finite state machines (FSMs) and protocols.

Even though the reduction algorithms of FormalCheck allow it to handle larger designs than most model checkers on the market, it still has limitations. One of the limitations is that in some cases if the user does not specify the Reduction Seeds for the Iterated algorithm, it will not find the smallest model automatically. The other limitation of the Iterated algorithm is that, it cannot be used to reduce the model for all kinds of queries. In some cases using this algorithm causes the verification to be terminated. In those cases the only option is to use the 1-Step reduction algorithm which is time consuming, and if the model

is too big it could lead to state space explosion. Therefore, some improvement in the Iterated algorithm would lead to higher verification quality when using FormalCheck. Also, if manual abstraction of a model is needed due to the limitations of this tool, the verification results have to be validated using simulation. Therefore, if the counter-examples in FormalCheck could be translated to test programs automatically, it could speed up the verification and debugging process.

Even though defining properties in FormalCheck seems to be very easy (since the user does not have to know a temporal logic like CTL), there are times that it can be challenging because of the limitations in FormalCheck. One of the limitations in defining properties is when a user chooses to use the Fulfill Delay option, he/she is demanded to define the Duration of that behavior as well. Consider a property which the Verification Window has to start few clock cycles after the Enabling Condition is fulfilled and continues to be true until a Discharging Condition is present. This type of property is very hard to define in FormalCheck. Therefore, some advancements in defining a Verification Window is required.

This thesis presented some practical approaches to model checking. Future work is needed to investigate the compositional verification in order to handle the formal verification of interacting blocks with two way dependencies.

Bibliography

[1] M. Abadi and L. Lamport: Conjoining Specifications; *ACM Transactions on Programming Languages and Systems*, Vol. 17, No. 4, pp. 1-27, September 1995.

[2] The ATM Forum Technical Committee: *UTOPIA Level 2*; V1.0, June 1995.

[3] L. Barakatain and S. Tahar: Model Checking of the Fairisle ATM Switch Fabric using FormalCheck; *Technical Report*, Concordia University, Department of Electrical and Computer Engineering, September 1999.

[4] I. Beer, S. Ben-David, C. Eisner, A. Landver: RuleBase: an Industry-oriented Formal Verification Tool; *Proc. Design Automation Conference (DAC'96)*, Las Vegas, Nevada, pp. 655-660, June 1996.

[5] Bell Labs Design Automation, Lucent Technologies: *FormalCheck Users Guide*; V2.1, July 1998.

[6] R.S. Boyer and J. S. Moore: *A Computational Logic Handbook*, Academic Press, 1988.

[7] R.K. Brayton, et. al.: VIS; *Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, Lecture Notes in Computer Science, Vol. 1166, Springer-Verlag, Palo Alto, CA., pp. 248-256, November 1996.

[8] R. Bryant: Graph Based Algorithms for Boolean Function Manipulation; *IEEE Transactions on Computers*, Vol. C-35, No. 8, pp. 677-691, August 1986.

[9] Cadence: *Formal Verification Using Affirma FormalCheck*; Manual, V2.3, October 1999.

[10] B. Chen, M. Yamazaki, and M. Fujita: Bug Identification of a Real Chip Design by Symbolic Model Checking; *Proc. International Conference on Circuits And Systems (ISCAS'94)*, London, U.K., pp. 132-136, June 1994.

[11] E. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. Long, K. McMillan, and L. Ness: Verification of the Futurebus+ Cache Coherence Protocol; *Formal Methods in System Design*, Vol. 6, pp. 217-232, 1995.

[12] E. M. Clarke, O. Grumberg, and D. A. Peled: *Model Checking*, The MIT Press, 1999.

[13] F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny: Multiway Decision Graphs for Automated Hardware Verification; *Formal Methods in System Design*, Vol. 10, pp. 7-46, February 1997.

[14] D. Dill: What's Between Simulation and Formal Verification?; *Invited Lecture in Design Automation Conference (DAC'98)*, San Francisco, CA, USA, June 1998.

[15] D. Dill: The Murφ Verification System; *Proc. International Conference on Computer-Aided Verification (CAV'96)*, Lecture Notes in Computer Science, Vol. 1102, Springer-Verlag, New Brunswick, NJ, USA, pp. 390-393, August 1996.

[16] A.T. Eiriksson and K. L. McMillan: Using Formal Verification/Analysis Methods on the Critical Path in System Design; *Proc. International Conference on Computer-Aided Verification (CAV'95)*, Lecture Notes in Computer Science, Vol. 939, Springer-Verlag, pp. 367-380, July 1995.

[17] C. Eisner: Real Value for Minimal Cost: Formal Verification of a Distributed Shared Memory Cache Coherence Protocol (A Case Study); IBM Science and Technology, Haifa Research Laboratory, Haifa, Israel, 1996.

[18] M. Gordon and T. Melham: *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic;* Cambridge, U.K.: Cambridge Univ. Press, 1993.

[19] C. Harkness and E. Wolf: Verifying the Summit Bus Converter Protocol With Symbolic Model Checking; *Formal Methods System Design*, Vol. 4, No. 2, pp. 83-97, February 1994.

[20] G.J. Holzmann: The Model Checker SPIN; *IEEE Transactions on Software Engineering*, Vol. 23, No. 5, pp. 279-295, May 1997.

[21] T. Jackson: Verification Critical Path for Today's IC Designers; *Electronics Journal*, Technical: Feature Article, September 1999.

[22] C. Kern and M. R. Greenstreet: Formal Verification in Hardware Design: A Survey; *ACM Transactions on Design Automation of Electronic Systems*, Vol. 4, No. 2, pp. 123-193, April 1999.

[23] A. Kuehlman, A. Srinivasan, and D. P. LaPotin: Verity- A Formal Verification Program for Custom CMOS Circuits; *IBM Journal* Res. Dev. 39, pp. 149-165, 1/2 (Jan./Mar. 1995).

[24] R. P. Kurshan: Formal Verification in a Commercial Setting; in *Proc. Design Automation Conference (DAC'97)*, Anaheim, California, pp. 258-262, June 1997.

[25] J. Lu and S. Tahar: Practical Approaches to the Automatic Verification of an ATM Switch Fabric using VIS; *Proc. IEEE 8th Great Lakes Symposium on VLSI (GLS-VLSI'98)*, Lafayette, Louisiana, USA, pp. 368-373, February 1998.

[26] J. Lu, S. Tahar, D. Voicu and X. Song: Model Checking of a real ATM Switch; *Proc. IEEE International Conference on Computer Design (ICCD'98)*, Austin, Texas, USA, pp. 195-198, October 1998.

[27] Z. Manna, et. al.: STeP: The Stanford Temporal Prover; *Tech. Rep. STAN-CS-TR-94-1518*, Computer Systems Laboratory, Stanford Univ., Stanford, CA., 1994.

[28] M.L. McMillan: *Symbolic Model Checking.* Norwell, MA: Kluwer, 1993.

[29] K.L. McMillan and J. Schwalbe: Formal Verification of the Encore Gigamax Cache Consistency Protocol; *Proc. Int. Symposium on Shared Memory Multiprocessors*, Tokyo, Japan, pp. 242-251, April 1991.

[30] K.L. McMillan: Verification of an Implementation of Tomasulo's Algorithm by Compositional Model Checking; *Proc. Computer Aided Verification (CAV'98)*, Vancouver, BC, Canada, pp. 110-121, June/July 1998.

[31] C. Norris Ip: State Reduction Methods for Automatic Formal Verification; *Ph.D. Thesis*, Department of Computer Science, Stanford University, December 1996.

[32] S. Owre, J.M. Rushby, and N. Shankar: PVS: a Prototype Verification System; *Proc. International Conference on Automated Deduction*, Saratoga Springs, NY, USA, pp. 748-752, 1992.

[33] V. Paruthi, N. Mansouri and R. Vemuri: Automatic Data Path Abstraction for Verification of Large Scale Designs; *Proc. International Conference on Computer Design (ICCD'98)*, Austin, Texas, pp. 192-194, October 1998.

[34] PMC-Sierra Inc.: *SCI-PHY Transmit Master and Receive Slave TSB Specification*; Issue 2, May 10, 1999.

[35] PMC-Sierra Inc.:*Saturn Compatible Interface Specification for PHY Layer and ATM Layer Devices, Level 2*; *Application Note*, Issue 4: August 1997.

[36] F. Pong, A. Nowatzyk, G. Aybay, and M. Dubois: Verifying Distributed Directory Based Cache Coherence Protocols: S3.mp, A Case Study; *Proc. Conference on Parallel Processing (EURO-Par'95)*, Lecture Notes in Computer Science, Vol. 966, Springer-Verlag, pp. 207-300, 1995.

[37] C.M. Roman: A Hierarchical Verification Methodology; *International Cadence User Conference*, Orlando, Florida, USA, September 1999.

[38] C.M. Roman, G. De Palma, and R. Kurshan: Model Checking Without Hardware Drivers; *International Conference on Correct Hardware and Verification Methods (CHARME'97)*, Invited Lecture, Montreal, Canada, October 1997.

[39] C.J. Seger: An Introduction to Formal Hardware Verification; *Tech. Rep.* 92-13, Dep. of Computer Science, University of British Colombia, Vancouver, B.C., Canada, June 1992.

[40] G. Smith, R. Bahnsen, and H. Halliwell: Boolean Comparison of Hardware and Flowcharts; *IBM Journal of Research and Development*, Vol. 26, pp. 106-116, January 1982.

[41] U. Stern and D. L. Dill: Automatic Verification of the SCI Cache Coherence Protocol; *Proc. Correct Hardware Design and Verification Methods (CHARME'95)*, Lecture Notes in Computer Science, Vol. 987, Springer-Verlag, Frankfurt, Germany, pp. 21-34, October 1995.

[42] S. Stoica: How Much Testing is Enough; *Proc. of IEEE International Test Conference*, Washington, D.C., USA, pp. 1129, August 1998.

[43] L. Wos, R. Overbeek, E. Lusk, and J. Boyle: *Automated Reasoning: Introduction and Applications*, Prentice-Hall, 1984.

[44] Y. Xu, E. Cerny, A. Silburt, A. Coady, Y. Liu and P. Pownall: Practical Application of Formal Verification Techniques on a Frame Mux/Demux Chip from Nortel Semiconductors; *Proc. Correct Hardware Design and Verification Methods (CHARME'99)*, Bad Herrenalb, Germany, pp. 110-124, September 1999.

[45] P. Zafiropulo, C. H. West, H. Rudin, D. D. Cowan, and D. Brand: Towards Analyzing and Synthesizing Protocols; *IEEE Transactions on Communications*, Vol. 28, No. 4, April 1980.

Appendix A

Properties of the TMRS

In this section the complete description of the properties defined for the verification of the Receive Slave SCI-PHY mode of the TMRS is presented.

Property_1: According to the SCI-PHY protocol: "When RRDENB is sampled low by the PHY layer device, the RSOC signal will be accepted by the ATM layer device on the next rising edge of RFCLK" [35]. We expressed this property for the TMRS as following: "If OENB_I is sampled low by the PHY layer device, and the TMRS is selected (TMRS-SEL and OENB_I signals were sampled high in the previous clock cycle), on the next rising edge of OFCLK the transmission of a full cell will start and the OSOC signal will be set to high". In FormalCheck, this property is expressed as follows.

```
Property: property_1

Type: Always

After: (@TMRSselected) and (@WaitSelectedState) and (@CLKrising)

Always: OSOC = 1 and @StartTxState

Options: Fulfill Delay: 0  Duration: 1  Count of @CLKrising
```

Reduction options:

```
Reduction Technique: Iterated

Reduction Seed: New (User defined seed)

Start as input: Fifo_Ctl_Inst

Make input: Datapath_Inst and Input_Mux_Inst
```

This property was decomposed to the following two properties, Property_1A which checks for the state transition to START_TRANSF state, and Property_1B which checks for the assertion of the OSOC signal. These two properties are stated as following.

Property_1A: If the TMRS is in the WAIT_SEL state and the TMRS was selected in the previous clock cycle (TMRSSEL_REG2 and OENB_I_REG2 signals are sampled high), and OENB_I signal was sampled low in the previous clock cycle (OENB_I_REG is low), on the next rising edge of OFCLK the TMRS will be in START_TRANSF sate. In FormalCheck, this property is expressed as follows.

```
Property: property_1A

Type: Always

After: (@TMRSselectedLastCC) and (@WaitSelectState) and (@CLKrising)

Always: @StartTxState

Options: Fulfill Delay: 0  Duration: 1  Count of @CLKrising
```

Reduction options:

```
Reduction Technique: Iterated

Reduction Seed: New (User defined seed)

Start as input: Fifo_Ctl_Inst

Make input: Datapath_Inst and Input_Mux_Inst
```

Property_1B: If the TMRS is in the LATCH4 or WAIT_SEL states and OENB_I signal is sampled low and the TMRS is selected (TMRSSEL_REG and OENB_I_REG signals are sampled high), on the next rising edge of OFCLK the transmission of a full cell will start (the OSOC signal will be set to high) and the TMRS will be in WAIT_SEL sate. In FormalCheck, this property is expressed as follows.

```
Property: property_1B

Type: Always

After: (@TMRSselected) and (@Latch4State or @WaitSelectState) and
       (@CLKrising)

Always: Osoc = 1 and @WaitSelectState

Options: Fulfill Delay: 0  Duration: 1  Count of @CLKrising
```

Reduction options:

```
Reduction Technique: Iterated

Reduction Seed: New (User defined seed)

Start as input: Fifo_Ctl_Inst

Make input: Datapath_Inst and Input_Mux_Inst
```

Property_2: According to the SCI-PHY protocol: "Each PHY link shall have a unique address corresponding to a value between 0 and 31. Upon sampling its address with the rising edge of the RFCLK, a PHY must drive RCA to indicate whether it has an entire cell in its buffer" [35]. For TMRS, this property is expressed as following: "When OAVALID_I and TMRSSEL are asserted (TMRS is polled) and all CMTY[31:0] lines are high or the prefetch of the next FIFO is not finished, OCA_O will be deasserted" [34]. In FormalCheck, this property is expressed as follows.

```
Property: property_2

Type: Always

After: (@TMRSpolled) and (Polling_Sm_Inst:Allfifoempty = 1) and (@RstDone)

       and (@CLKrising)

Always: (not @RCAhigh)

Options: (None)
```

Reduction options:

```
Reduction Technique: Iterated

Reduction Seed: New (User defined seed)

Start as input: Transfer_Sm_Inst

Make input: Datapath_Inst and Input_Mux_Inst
```

Property_3: In Receive Slave SCI-PHY back-to-back transfer mode, when the external master device does not deassert OENB_I at the end of a transfer, the TMRS remains selected for another cell transfer. If all FIFOs are empty the TMRS will deassert ODAT_OEB and will wait to be reselected [34]. In FormalCheck, this property is expressed as follows.

```
Property: property_3

Type: Always

After: (@TransferDone) and (Oenb_I = 0) and

       (Polling_Sm_Inst: Allfifoempty = 1) and (@RstDone)

Always: Odat_Oeb = 1

Options: Fulfill Delay: 0 Duration 1 count of @CLKrising
```

Reduction options:

```
Reduction Technique: 1-Step

Reduction Seed: New (User defined seed)

Make input: Datapath_Inst and Input_Mux_Inst
```

Property_4: According to the SCI-PHY protocol: "No PHY shall drive RCA upon sampling RAVALID low" [35]. In FormalCheck, this property is expressed as follows.

```
Property: property_4

Type: Never

After: (Oavalid_I = 0) and (@RstDone) and (@CLKrising)

Never: @RCAhigh

Options: (None)
```

Reduction options:

```
Reduction Technique: Iterated

Reduction Seed: New (User defined seed)

Start as input: Fifo_Ctl_Inst

Make input: Datapath_Inst and Input_Mux_Inst
```

Property_5: According to the SCI-PHY protocol: "Each PHY link shall have a unique address corresponding to a value between 0 and 31. Upon sampling its address with the rising edge of the RFCLK, a PHY must drive RCA to indicate whether it has an entire cell in its buffer" [35]. In TMRS this property is expressed as following: "When Oavalid_I and TMRSSEL are asserted (TMRS is polled), OCA_O is driven to high in the next clock cycle, if at least one of the CMTY[31:0] status bus is low and the prefetch of the next FIFO is finished" [34]. In FormalCheck, this property is expressed as follows.

```
Property: property_5
Type: Always
After: (@TMRSpolled) and (@FIFOnotEmpty) and (@PrefetchDone) and
        (@CLKrising) and (@RstDone)
Always: @RCAhigh
Options: Fulfill Delay: 0 Duration: 1 Count of @CLKrising
```

Reduction options:

```
Reduction Technique: Iterated
Reduction Seed: New (User defined seed)
Start as input: Fifo_Ctl_Inst
Make input: Datapath_Inst and Input_Mux_Inst
```

In Property_5, the Polling and Transfer controllers are interacting blocks. The Transfer controller uses the internal output of the Polling controller to determine whether a cell is available, and asserts its internal output to the Polling controller when the prefetch cycle of that cell is done. When the TMRS is polled by the master, the Polling controller uses the internal output of the Transfer controller to assert/deassert the RCA signal. The verti-

cal verification described in Section 3.3 could be applicable for this property in case of state explosion. To verify Property_5 using sub-properties, we need to add the following State Variable and Constraints.

1. A State Variable Named New_Cell_Available. This State Variable becomes 0 when there are no more cells (besides the one that is currently being transferred) available, and 1 when there is at least one more cell to transfer. This State Variable was defined to be able to describe the New_Cell_Tx_Rdy Constraint. It will be explained more later.
2. Since Property_5 is verified under the condition that there exists a cell to transfer, a Constraint named Channel_Not_Empty has to be defined. This Constraint makes sure that there is a FIFO that has a cell to transfer (besides the cell that is currently being transferred).
3. A Constraint to assume the property of Polling_Sm block (this Constraint is verified as Property_6A1). This Constraint is defined as following:

```
Constraint: New_Cell_Tx_Rdy
Type: Always
After: New_Cell_Available = 1 and @CLKrising
Assume Always: Polling_Sm_Inst:New_Cell_Transf_Rdy = 1
Unless: New_Cell_Available = 0
Options: (None)
```

In FormalCheck when defining the Fulfill Delay to slide the verification window (refer to Chapter 2), it is mandatory to define the Duration of the Verification Window as well. New_Cell_Transf_Rdy signal becomes high after two clock cycles and, it stays high unless there

are no more cells available to transfer. Since the duration of the New_Cell_Transf_Rdy signal being high is not known, we could not use the Fulfill Delay option, therefore, we defined the State Variable New_Cell_Available to specify the behavior of New_Cell_Transf_Rdy signal. Now we can define the property of Transfer controller (Property_5A) as following.

Property_5A: Prefetch of a cell will eventually be done, assuming there is at least one cell available and the New_Cell_Transf_Rdy is asserted (the assertion of New_Cell_Transf_Rdy happens two clock cycles after a cell is available).

```
   Property: property_5A

   Type: Eventually

   After: (@CLKrising) and (@RstDone)

   Eventually: @PrefetchDone

Options: (None)
```

Reduction options:

```
   Reduction Technique: Iterated

   Reduction Seed: New (User defined seed)

   Start as input: Fifo_Ctl_Inst

   Make input: Datapath_Inst, Input_Mux_Inst and Polling_Sm
```

After verifying the property of Transfer controller block (Property_5A), this property is assumed to verify the property of Polling controller (Property_5B). Therefore, turning Property_5A to a Constraint we have:

```
Constraint: PrefetchIsDone

Type: Eventually

After: (Polling_Sm:New_Cell_Transf_Rdy = 1) and (@CLKrising) and (@RstDone)

Assume Eventually: @PrefetchDone

Unless: Polling_Sm:New_Cell_Transf_Rdy = 0

Options: (None)
```

Then, assuming the PrefetchIsDone Constraint, Property_5B is defined as following.

Property_5B: When Oavalid_I and TMRSSEL are asserted, OCA_O is driven to high in the next clock cycle, if at least one of the CMTY[31:0] status bus is low and assuming the prefetch of the next FIFO is finished. In FormalCheck, this property is expressed as follows.

```
Property: property_5B

Type: Always

After: (@TMRSpolled) and (@FIFOnotEmpty) and (@CLKrising) and (@RstDone)

Always: @RCAhigh

Options: Fulfill Delay: 0 Duration: 1 Count of @CLKrising
```

Reduction options:

```
Reduction Technique: Iterated

Reduction Seed: New (User defined seed)

Start as input: Fifo_Ctl_Inst

Make input: Datapath_Inst, Input_Mux_Inst and Transfer_Sm
```

Property_6: Based on the TMRS specification, "in Receive Slave SCI-PHY back-to-back transfer, when the external master device does not deassert OENB_I at the end of a transfer, the TMRS remains selected for another cell transfer" [34]. Hence, when the external master does not deassert OENB_I at the end of a transfer, and at least one of the FIFOs have a cell to transfer, the TMRS will eventually start transmitting the new cell.

We acquired from Property_1, that the OSOC signal and START_TRANSF state of the Transfer_Slave state machine are not synchronous. Accordingly, we had to decompose Property_6 to two properties to be able to check for both the START_TRANSF state of the FSM (Property_6A) and the OSOC signal (Property_6B), while in back-to-back transfer mode. The queries of these two properties contain another Constraint, named BackToBackTx, in addition to the default Constraints explained in Section 4.2.1.2. This Constraint puts the TMRS in the back-to-back transfer mode. Also to make matters easier, we assumed no interruptions from the master will occur while transferring a cell, as well as after the completion of a cell transfer (since a cell transfer with interruptions from the master was verified through other queries, this assumption is considered safe). The BackToBackTx Constraint is expressed in FormalCheck as follows.

```
Constraint: BackToBackTx

Type: Always

After: (@IncCptState) and (@CLKrising)

Assume Always: Oenb_I = 0

Unless: (@StartTxState) or (@WaitSelectState)

Options: (None)
```

In FormalCheck, Propery_6A and Property_6B are expressed as following.

Property_6A:

```
Property: property_6A

Type: Eventually

After: (@TransferDone) and (Transfer_Sm_Inst:New_Cell_Transf_Rdy = 1) and
       (@Clkrising)

Eventually: @StartTxState

Options: (None)
```

Reduction options:

```
Reduction Technique: Iterated

Reduction Seed: New (User defined seed)

Start as Input: Fifi_Ctl_Inst

Make input: Datapath_Inst and Input_Mux_Inst
```

Property_6B:

```
Property: property_6B

Type: Eventually

After: (@TransferDone) and (Transfer_Sm_Inst:New_Cell_Transf_Rdy = 1) and
       (@Clkrising)

Eventually: (Osoc = 1) and (@LatchNext2State)

Options: (None)
```

Reduction options:

```
Reduction Technique: Iterated

Reduction Seed: New (User defined seed)

Start as Input: Fifi_Ctl_Inst

Make input: Datapath_Inst and Input_Mux_Inst
```

The relationship between the Polling controller and Transfer controller in Property_6A and Property_6B is linear, therefore, the vertical verification (refer to Section 3.3) could be applied in case of state explosion. To do so, each one of these properties can easily be divided to two sub-properties, one of which will be the property of Polling controller block (Property_6A1 and Property_6B1), and the other one the property of Transfer controller block (Property_6A2 and Property_6B2). These properties are explained as following.

Property_6A1:

```
Property: property_6A1
Type: Always
After: (New_Cell_Available = 1) and (@Clkrising)
Always: Polling_Sm:New_Cell_Transf_Rdy = 1
Unless: New_Cell_Available = 0
Options: (None)
```

Reduction options:

```
Reduction Technique: Iterated
Reduction Seed: New (User defined seed)
Make input: Input_Mux_Inst, Transfer_Sm_Inst, and Fifo_Ctl_Inst, and
            Datapath_Inst
```

Property_6B1 is exactly the same as Property_6A1, therefore we avoid rewriting this property. Now assuming there is a cell available in the FIFO (Constraint Channel_Not_Empty) and assuming Property_6A1 (Constraint New_Cell_Tx_Rdy), Property_6A2 and Property_6B2 should be verified.

Property_6A2:

```
Property: property_6A2

Type: Eventually

After: (@TransferDone) and (@Clkrising)

Eventually: @StartTxState

Options: (None)
```

Reduction options:

```
Reduction Technique: Iterated

Reduction Seed: New (User defined seed)

Start as Input: Fifi_Ctl_Inst

Make input: Datapath_Inst, Input_Mux_Inst, and Pollins_Sm_Inst
```

Property_6B2:

```
Property: property_6B2

Type: Eventually

After: (@TransferDone) and (@Clkrising)

Eventually: (Osoc = 1) and (@LatchNext2State)

Options: (None)
```

Reduction options:

```
Reduction Technique: Iterated

Reduction Seed: New (User defined seed)

Start as Input: Fifi_Ctl_Inst

Make input: Datapath_Inst, Input_Mux_Inst, Polling_Sm_Inst
```

Property_7: According to SCI-PHY protocol: “The ATM layer device may pause the transfer at any time by deasserting RRDENB” [35]. In TMRS this property is expressed as following: “The Slave Transfer State Machine always expects a complete cell transfer but supports transfer interruption by deassertion of Oenb_I” [34]. In FormalCheck, this property is expressed as follows.

```
Property: property_7
Type: Eventually Always
After: (not @TransferDone) and (Oenb_I =1) and (@RstDone) and (@CLKrising)
Eventually Always: @TransferDone
Unless: (Oenb_I =1)
Options: (None)
```

Reduction options:

```
Reduction Technique: Iterated
Reduction Seed: New (User defined seed)
Start as input: Fifo_Ctl_Inst and Polling_Sm_Inst
Make input: Datapath_Inst and Input_Mux_Inst
```

Property_8: According to SCI-PHY protocol: “When RCA transitions high, the ATM layer device can read a full cell from the PHY layer device. When the empty RCA deassertion option is implemented, RSOC is valid *only* when the RCA signal is coincidentally asserted and RRDENB was asserted low in the previous cycle” [35]. In TMRS we express this property as following: “If the master selects the TMRS before the prefetch cycle is finished, the TMRS will not start a cell transfer in the next clock cycle”. In FormalCheck, this property is expressed as follows.

```
Property: property_8
Type: Never
After: (not @PrefetchDone) and (@TMRSselected) and (@CLKrising) and
       (@RstDone)
Never: Osoc = 1
Options: Fulfill Delay: 0 Duration 1 count of @CLKrising
```

```
Reduction options:
  Reduction Technique: Iterated
  Reduction Seed: New (User defined seed)
  Start as input: Fifo_Ctl_Inst
  Make input: Datapath_Inst and Input_Mux_Inst
```

Property_9: According to SCI-PHY protocol: “When RRDENB is sampled low by the PHY layer device, the RDAT bus will be accepted by the ATM layer device on the next rising edge of RFCLK. When RRDENB is sampled high by the PHY layer device, no transfer is performed in the subsequent RFCLK cycle” [35]. In TMRS this property is expressed as following: “When operating in Slave mode the TMRS monitors the input signal Oenb_I to validate the data transfer” [34]. Therefore, if transferring a cell is interrupted by Oenb_I being set to high, the Odat_Oeb signal will be set to high in the next clock cycle. In FormalCheck, this property is expressed as follows.

```
Property: property_9
Type: Always
After: (not @TransferDone) and (Oenb_I =1) and (@RstDone) and (@CLKrising)
Always: Odat_Oeb = 1
Options: Fulfill Delay: 0 Duration 1 count of @CLKrising
```

```
Reduction options:

  Reduction Technique: Iterated

  Reduction Seed: New (User defined seed)

  Start as input:Fifo_Ctl_Inst and Polling_Sm_Inst

  Make input: Datapath_Inst and Input_Mux_Inst
```

Property_10: According to SCI-PHY protocol: "When RRDENB is sampled low by the PHY layer device, the RDAT bus will be accepted by the ATM layer device on the next rising edge of RFCLK" [35]. In TMRS when operating in Slave mode, this property is expressed as following: "ODAT_OEB is low when OENB_I was asserted low in the previous clock cycle, and the TMRS is selected for a cell transfer" [34]. In FormalCheck, this property is expressed as follows.

```
  Property: property_10

  Type: Always

  After: (@TMRSselected) and (@RstDone) and (@CLKrising)

  Always: Odat_Oeb = 0

  Unless: (Oenb_I =1)

  Options: Fulfill Delay: 0 Duration 1 count of @CLKrising

Reduction options:

  Reduction Technique: Iterated

  Reduction Seed: New (User defined seed)

  Start as input: Fifo_Ctl_Inst and Polling_Sm_Inst

  Make input: Datapath_Inst and Input_Mux_Inst
```

Property_11: This property expresses the reverse case of Property_10. When TMRS is not in back-to-back transfer mode, if a cell transfer is complete but the TMRS is not selected by the master again, the Odat_Oenb will be deasserted unless it is selected again. In FormalCheck, this property is expressed as follows.

```
  Property: property_11

  Type: Always

  After: (WasSelected = 0) and (@RstDone) and (@CLKrising)

  Always: Odat_Oeb = 1

  Unless: (WasSelected = 1)

  Options: (None)

Reduction options:

  Reduction Technique: Iterated

  Reduction Seed: New (User defined seed)

  Start as input: Fifo_Ctl_Inst

  Make input: Datapath_Inst and Input_Mux_Inst
```