Modeling and Formal Verification of a Telecom System Block Using MDGs

Md Hasan Zobair

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

Montréal, Québec, Canada

April 2001

Abstract

Modeling and Formal Verification of a Telecom System Block Using MDGs

Md Hasan Zobair

Simulation-based verification cannot uncover all errors in an implementation because only a small fraction of all possible cases can be considered. Formal verification is a different technique that can alleviate this problem. Because the correctness of a formally verified design implicitly involves all cases regardless of the input values.

This thesis demonstrates the effectiveness of Multiway Decision Graphs (MDG) to carry out the formal verification of an industrial Telecom hardware which is commercialized by PMC-Sierra Inc. To handle the complexity of the design, we adopted a hierarchical proof methodology as well as a number of model abstraction and reduction techniques. Based on the hierarchy of the design, we followed a hierarchical approach for the equivalence checking of the TSB. We first verified that the RTL implementation of each module complies with the specification of its behavioral model. We also succeeded to verify the full RTL implementation of the TSB against its top level specification. Besides equivalence checking, we furthermore applied model checking to ascertain that both the specification and the implementation of the TSB satisfy some specific characteristics of the system. To measure the performance of the MDG verification, we also conducted the verification of the same TSB with Cadence FormalCheck. The experimental results show that in some cases, the MDG based modeling with abstract state variables allows more efficient verification than that of the boolean modeling in FormalCheck.

Acknowledgments

The contributions of others to the completion of this thesis have been substantial and varied. I will here acknowledge those whose names most immediately comes to mind and apologize those names I have omitted inadvertently.

I would like to express my indebtness to my supervisor Dr. Sofiène Tahar. His guidance, assistance and encouragement were an asset during the thesis work in particular and in my stay at Concordia, in general. Professor Tahar also read and criticized many versions of this thesis. The final version owes much to him, the mistakes are my own.

I would like to thank Mr. Jean Lamarche, Manager, PMC-Sierra Inc. for providing the RASE telecom system block design as a research case study. Without his cooperation this research work would not be possible. In particular, for his valuable time to explain us the functionalities of the design.

I would like to thank the examination committee members, Dr. A. Al-Khalili, Dr. P. Sinha, Dr. O. Ait-Mohamed for serving in my thesis defence. Their comments and feedback were of great value in improving earlier versions of this thesis.

To all my friends and colleagues in the Hardware Verification Group (HVG) at Concordia, thank you for your encouragement, thoughtful discussions, and help. I wish to express my sincere and heartfelt thanks to my friends Syed Asif Iqbal and Mohammed Wahidul Islam who have helped me to proofread my thesis drafts many times.

Last but not least I would like to thank my wife, my parents and the rest of my family members in Bangladesh for their constant moral support and encouragement which were invaluable in completing this thesis.

Table of Contents

List of Figures

List of Tables

List of Acronyms

- ACTL: **A**bstract **C**omputational **T**ree **L**ogic
- ATM: **A**synchronous **T**ransfer **M**ode
- APSC: **A**utomatic **P**rotection **S**witching **C**ontrol
- BDD: **B**inary **D**ecision **D**iagram
- ROBDD: **R**educed **O**rdered **BDD**
- EOBDD: **E**xtended **O**rdered **BDD**
- BMD: **B**inary **M**oment **D**iagram
- HDD: **H**ybrid **D**ecision **D**iagram
- MTBDD: **M**ulti **T**erminal **BDD**
- MDG: **M**ultiway **D**ecision **G**raph
- CTL: **C**omputational **T**ree **L**ogic
- TSB: **T**elecom **S**ystem **B**lock
- RASE: **R**eceive **A**utomatic Protection Switch Control, **S**ynchronization Status Extraction and Bit **E**rror Rate Monitor, a commercial product of PMC-Sierra Inc.
- SONET: **S**ynchronous Transport **O**ptical **Net**work
- SSF: **S**ynchronization **S**tatus **F**iltering, S1 byte filtering of SONET frame
- BER: **B**it **E**rror **R**ate
- BIP: **B**it **I**nterleaved **P**arity
- BERM: **B**it **E**rror **R**ate **M**onitoring on B2 bytes of a SONET frame
- SPE: **S**ynchronous **P**ayload **E**nvelope of a SONET frame
- STS-1: **S**ynchronous **T**ransport **S**ignal, basic electrical signal of a SONET frame.
- STS-3: **S**ynchronous **T**ransport **S**ignal, multiplexed from STS-1 signal
- TOH: **T**ransport **O**ver**h**ead section of a SONET frame
- SOH: **S**ection **O**ver**h**ead
- LOH: **L**ine **O**ver**h**ead
- POH: **P**ath **O**ver**h**ead
- UTOPIA: **U**niversal **T**est and **O**perations **P**HY Interface for **ATM**
- TMRS: **T**ransmit **M**aster/**R**eceive **S**lave, a commercial product of PMC-Sierra Inc.
- RTL: **R**egister **T**ransfer **L**evel
- RIN: **R**eceive **In**put data stream
- FSM: **F**inite **S**tate **M**achine
- PHY: **Phy**sical Layer
- VIS: **V**erification **I**nteracting with **S**ynthesis
- SMV: **S**ymbolic **M**odel **V**erifier

Chapter 1

Introduction

1.1 Motivation

Design errors in digital designs are destructive. The devices are manufactured in vast quantities and errors cannot be remedied in the field by patching. Recently discovered errors in several industrial designs demonstrate that these concerns are not academic. In addition to economic considerations, the degree of design correctness is necessary to the use of digital systems in critical applications. As IC chips grew from tens of thousands of gates in the eighties to more than 10 millions today, designers and verifiers of these massive new circuits face a pair of conflicting goals. They must find ways to contain the expansion of verification time, effort, and cost. They also must increase verification coverage and quality to ensure single-pass success [34].

Simulation-based method is currently used by the industrial community for system-level verification, since it can handle the entire design at a time. When a simulation trace exposes a design error, a verifier analyzes the trace and rectifies the design. However, the rectification might be inadequate because the trace shows only one specific behavior of the system and one cannot confirm that no other trace exposes the error. This handicap is the motivation for the need of new methods to achieve economical and reliable verification of

digital systems. Formal verification technique had paved a path, showing the utility of finding bugs early in the design cycle. It now affords some promising new methods for the validation of digital designs. Some automatic formal verification techniques are gradually finding their place in the verification process as complement to logic simulation. FSM-based automatic verification techniques have proven to be successful formal verification technique that can be applied to real industrial design. However, since it requires the design to be described at the boolean level, they often fail to verify a large-scale design because of the *state space explosion* problem caused by the large datapath. On the other hand while scalable to large designs, theorem proving based interactive verification techniques require a great expertise by the verifier.

Thus, the motivation behind this work was to investigate Multiway Decision Graphs (MDGs) in verifying a large-scale industrial design, and proposes a hierarchical approach for organizing the verification of the investigated design using MDGs. We also provide a comparative evaluation of the experimental results from the model checking of the proposed design using Cadence FromalCheck and the MDG tools.

1.2 Background and Related Work

A cornerstone of many of the Asic CAD suites is the ability to design hierarchically. Complex designs need a mechanism to reduce their complexity for both the designer and verifier. It is difficult to understand a design with hundreds of components; it is easier to understand the same design with only a few components. Using hierarchies to handle complexities is an old methods. It does not mean that the design becomes less complex (sometimes it becomes more complex instead), but it becomes easier to understand for both the designer and verifier. To accelerate the design flow and assure the correctness of

complex digital systems, a hierarchical design approach is usually adopted by the industrial community (see Figure 1.1) [29].

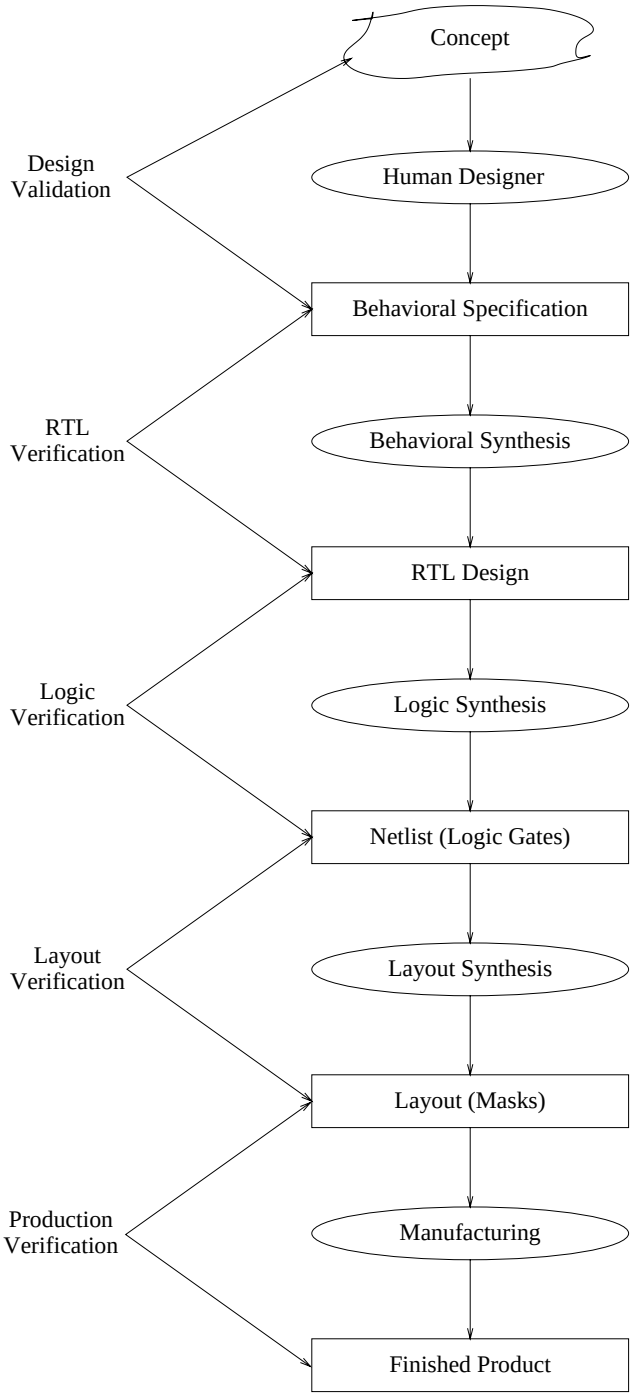


Figure 1.1: Flow of hierarchical design and verification

The system architect first manually derives the requirements of the system as the system behavioral specification. This specification is then refined manually or using CAD tools (e.g., Synopsys) into more detailed descriptions such as register-transfer (RT), logic and mask level descriptions. As the late detection of design errors is largely responsible for unexpected delays in realizing the hardware design, it is extremely important to ensure correctness in each design step. With correct-by-construction design style, automatic tools, such as behavioral and logic synthesis techniques can be used to ensure behavioral and gate level design correctness. However, the refinement process from high-level specification to synthesizable design usually requires manual fine tuning to achieve high performance. More progress is needed to automate the design process at higher levels in order to produce designs of the same quality as is achievable by hand. It is thus essential that the specification (or behavior) and the intermediate design stages be verified for consistency and correctness with respect to some user-specified properties or a previous level of the specification, thus making post-design verification essential. Formal verification techniques are useful in verifying designs between different levels of hierarchy. Such as using sequential equivalence checking, we can verify that whether the RTL model of the design satisfies its specification or not. We can also apply formal verification techniques to verify the gate level netlists of the design. Currently combinational equivalence checking is used in post synthesis design verification where often manual design changes focus on speed, power or testability considerations. In order for a combinational equivalence checker to work, the designs must have a one-to-one mapping of registers/flip-flops. Model checking can be applied to prove temporal properties, i.e., specification, on a design model under all possible and allowable conditions. Here the design under verification can be an

RTL model or a behavioral model. Formal verification techniques, however, cannot be applied to detect a fabrication fault resulting from layout defects during the fabrication process, which may lead to an incorrect behavior of the design.

1.2.1 Formal Verification

Formal verification technique is the mathematical demonstration of consistency between specification and implementation of a design. The ability of formal verification is to check “corner-cases”, which are difficult or infeasible to test through simulation. These include especially complex scenarios unanticipated by the designers. Decreased time to market comes from the ability to apply formal verification earlier in the design cycles and thus find bugs sooner than is possible with simulation. Verifiers who adopted formal verification methods cut the time required to verify a complex design’s implementation from months to weeks or days. Because formal verification techniques consider all cases implicitly, a verifier does not need to generate test cases to verify a design which takes much of the verification time during traditional simulation phase. Due to the above reasons formal verification is growing as a powerful complementary approach to simulation in the industrial community [39].

1.2.2 Formal Verification Techniques

In general, formal verification consists of mathematically establishing that an implementation satisfies its specification. The implementation refers to the system design which is to be verified. This entity can represent a design description at any level of the system abstraction hierarchy. The specification usually refers to the property with respect to

which correctness is to be determined. It can be expressed in a variety of ways, such as behavioral description, an abstract structural description, a timing diagram which reflects the behavior of the system at different time points, a temporal logic formula, etc.

Formal verification techniques naturally group themselves into theorem proving methods and automated FSM state enumeration based methods [31, 37].

1.2.2.1 Theorem Proving

Theorem proving is the most general verification technique, in which a specification and its implementation are usually expressed as first-order or higher-order logic formulae. Their relationship, equivalence or implication, is regarded as a theorem to be proven within the proof system using a set of axioms and inference rules. Theorem proving have had their greatest successes in verifying datapath dominated circuits as it supports the verification of parameterized datapath dominated circuits [51]. It has high abstraction and powerful logic expressiveness capabilities. Using theorem proving, designs can thus be represented at different abstraction levels rather than only at the boolean level [35]. Therefore, it allows a hierarchical verification methodology which can effectively deal with the overall functionality of designs having complex datapaths. However, in contrast to more automated formal verification methods, such as *model checking* or e*quivalence checking*, it is currently a memory and time consuming methods. This is specially so when a real industrial design is to be considered. Users need expertise to verify any design using theorem proving which is the major difficulty for applying the method on industrial designs. Some of the widely used theorem prover in the hardware verification community are HOL (Higher-Order Logic) [30], PVS (Prototype Verification System) [46], Nqthm (a Boyer-

Moore theorem prover) [8], ACL2 (Industrial strength version of the Boyer-Moore theorem prover) [36].

1.2.2.2 FSM-based Techniques

Automated finite state based methods can also be classified into the following categories [31]:

- **Model Checking**: The specification is in the form of a logic formula, the truth of which is determined with respect to a semantic model provided by an implementation.
- **Equivalence Checking**: The equivalence of a specification and an implementation is checked, e.g., equivalence of functions, equivalence of finite-state automata, etc.

Model checking [17, 18] is an automatic technique for verifying finite state concurrent systems. It has a number of advantages over traditional approaches to the problems that are based on simulation and deductive reasoning. The method has been used successfully in practice to verify complex sequential circuit designs and communication protocols. In model checking, ideally the verification is completely automatic. There are two main approaches within model checking — logic based and automa-based [37]. The logic paradigm is based on *temporal logics* [10, 27]. On the other hand, the automata paradigm is based on *language containment* [38]. These two are not at all fundamentally same and each can be described in terms of the other.

A Model checker provides means for checking that a model of the design satisfies a given specification, but it is impossible to determine whether the given specification covers all the properties that the system should satisfy. The main challenge in model checking

is dealing with the *state space explosion* problem [13]. This problem occurs in systems with many components that can interact with each other or systems that have data structures that can assume many different values, e.g., the datapath of a design. In such cases the number of global states can be enormous. One successful technique which attempts to reduce the state space explosion problem is known as symbolic model checking [22, 23]. In this technique, the transition and output functions as well as the sets of states generated during the reachability analysis are encoded by Bryant's Reduced and Ordered Binary Decision Diagrams (ROBDDs) [11]. A number of ROBDD-based verification tools have been developed which are used by the hardware verification community are VIS (Verification Interacting with Synthesis) [9], SMV (Symbolic Model Verifier) [44]. Both of these tools are based on CTL [8]. However, the ROBDD-based methods are not capable to verify designs having non-trivial datapaths [23, 16]. Other tools like, FormalCheck [7] perform model checking based on ω-automata [33] which is commercialized by Cadence design systems. It has some built-in localization reduction algorithm that can handle large size industrial design [7, 14].

Equivalence checking [37] is used to proof the functional equivalence of two design representations modeled at the same or different levels of abstraction. It can be divided into two categories: one is combinational equivalence checking, and the other is sequential equivalence checking. Equivalence checking verifies for all input sequences that an implementation has the same outputs as the specification.

In *combinational equivalence checking*, the function of the two descriptions are converted into canonical forms which are then structurally compared. The major advantage of BDDs is their efficiency for a wide variety of practically relevant combinational circuits.

Two combinational designs functionally modeled by the Boolean functions are proven equivalent by compairing their normal forms. If ROBDDs are used which have the canonical form property, it is sufficient to test both graphs for isomorphism. Subject to a reduction criterion and for a fixed variable ordering, BDDs are a canonical representation for Boolean functions, in the sense that the BDDs for two functions f_1 and f_2 are isomorphic iff $f_1 = f_2$. Current designs which are clock-driven synchronized designs, need to be separated into small portions. The tool then maps each register of one model into another and compares their combinational circuits between every two consecutive registers.

Sequential equivalence checking is used to verify the equivalence between two sequential designs at each state. Sequential equivalence checking considers just the behavior of two designs while ignoring their implementation details such as register mapping. It can verify the equivalence between RTL and netlist or RTL and behavioral model which is very important in design verification. To verify the equivalence of these models, we need efficient representation for the manipulation of next-state and output functions and the set of states. Two models are considered equivalent iff for each input sequence, both machines generate the same output sequence. The problem of showing that two machines are equivalent can be reduced to the problem of finding the reachable state set of a product machine. The reachable state set of sequential circuit can be computed using a fixpoint calculation, where the next-state relation derived from the next-state function and current inputs to the circuit. The number of iterations required to compute the reachable state set can be linear in the number of states for the circuits that have “long” cycles of states. The disadvantage of sequential equivalence checking is that it cannot handle a large design because it encounters state space explosion problem very fast.

Recently, a number of ROBDD extensions such as BMDs [12], HDDs [19] and K*BMDs [26] have been developed to represent arithmetic functions more compactly than ROBDDs to accelerate the verification of complex arithmetic circuits. EOBDDs (can be improved by labelling the leaf nodes with terms containing abstract sorts. MDGs (Multiway Decision Diagrams) [20], successor of EOBDDs (Extended Ordered Binary Decision Diagrams) [42], include the labelling of edges to be first order terms and non-terminal nodes to be abstract variables. ROBDDs, a special cases of MDGs, can be turned into MDGs by transforming them from graphs representing functions into graphs representing relations. In this thesis, we will use MDG-based verification. Details on MDG are described in Chapter 2.

1.2.3 Related Work

In this section, we review several existing related works on the formal verification of different moderate sized digital systems using FSM-based methods. Some of these case studies are used to illustrate the limitations of current formal verification technique in verifying industrial like designs. Among these limitations, state space explosion is the well-known problem faced by the FSM-based methods when verifying designs with a substantial datapath. Researchers of these works had presented different reduction and abstraction techniques to cope with this limitation.

Chen *et al*. at Fujitsu Digital Technology Ltd. [16] identified a design error in an ATM (Asynchronous Transfer Mode) [32] circuit using the SMV tool [44] by verifying some properties expressed in CTL [8]. When the circuit was manufactured it showed an abnormal behavior under certain circumstances. Identifying the specific error during simulation,

they established an abstracted model to cope with the state space explosion and verified the abstracted model by SMV. The design error was detected during the model checking. However, the ATM model was reduced and abstracted a lot from the original design according to the specific error, and the same ATM model may not be used to verify other properties of the original design.

Garcez [28] has also verified some properties on the implementation of the Fairisle ATM switch fabric [41] using HSIS [3] model checking tool. The author described the netlist implementation of the ATM switch fabric using a subset of Verilog, and checked properties on submodules of the fabric using model checking. During the model checking, he did not consider the whole switch fabric to avoid state space explosion. Moreover, he changed the implementation of the fabric to ease the verification process.

Lu and Tahar [42, 41] verified a the Fairisle ATM switch fabric [41] using VIS [9]. Since they did not succeed in using the original switch fabric to verify relevant liveness and safety properties (due to *state space explosion*) they abstracted the model by datapath reduction technique. After this reduction in the model, they successfully verified several properties, but the verification time of each property was unreasonably long. To reduce the verification time, they applied *property division* and *latch reduction* techniques. They also conducted equivalence checking between the behavioral and structural specifications of the submodules written in Verilog HDL. The VIS tool failed to complete the equivalence checking of even a very reduced model of the whole switch fabric due to state space explosion.

Hong and Tahar [49] used an ATM Bit Error Rate Monitor (BERMON) design to illustrate their methodology for the compositional verification of IP-block based designs using

VIS tool. In this work, they focused on the reasoning of the compositional verification of the IP based design and on the issue of how interface behavior should be provided with an IP block to make the verification feasible.

Rajan *et al.* [50] used a combination of theorem proving, model checking and simulation to verify a high-level ATM model. They used model checking to verify some control components in the design, and applied exhaustive simulation to verify some operational components. Theorem prover was applied to verify the whole ATM model. They discovered bugs in the high-level ATM model which was assumed correct during simulation.

Barakatain and Tahar [5] applied model checking techniques for the formal verification of a SCI-PHY Level 2 protocol (a super set of UTOPIA Level 2 protocol [2]) Engine. They used Cadence FormalCheck [14] to formally verify the RTL implementation of the Receive Slave SCI-PHY mode of the Transmit Master/Receive Slave (TMRS) design [48]. The TMRS is an existing industrial design of PMC-Sierra Inc., with a 7500 equivalent gate-count. During the verification process, they used several model abstraction and reduction techniques within FormalCheck to avoid state space explosion, and then verified a number of relevant liveness and safety properties on the TMRS. They succeeded the discovery of a number of mismatches between the TMRS RTL design, document specification and UTOPIA Level 2 protocol description.

Xu *et al.* [53] verified a Frame Multiplexer/Demultiplexer (FMD) chip from Nortel Semiconductors using Cadence FormalCheck [14]. The FMD chip is part of a system used in multiplexing/demultiplexing framed data between various channels and a SONET line. The authors constructed a non deterministic model to simulate the normal operating environment. Tool guided model reduction was used to build an abstracted model which in

turn reduced the state space of the original design. During the verification process, they detected two errors in the implementation of the FMD model.

Tahar *et al*. [52] verified the Fairisle ATM switch fabric [41] in an automatic fashion using MDGs by property and equivalence checking. The original design was modeled in Qudos HDL, containing 4200 equivalent gate-count. They used model abstraction techniques to reduce the state space of the gate level netlist. Using the abstract sort and uninterpreted functions within MDG, they were able to verified the whole switch fabric without having any *state space explosion* problem.

Zobair *et al.* [60] used the Fairisle ATM switch fabric [41] to investigate the impact of design changes on formal verification using the MDG tools [59]. In this work, they showed that design for *verifiability* can have significant effect on the speed of verification using automated decision diagram based technique. The same result was obtained by Curzon *et al.* [25], using interactive proof with the HOL theorem prover [30] for the same design verification. The difference in nature of these two verification methodologies suggests design for *verifiability* can be widely applicable as design for *testability.*

Z. Zhou *et. al* [58] demonstrated the MDG-based formal verification technique on the example of the Island Tunnel Controller (ITC) design. In this work, they studied in detail the non-termination problem of abstract state enumeration and presented a heuristic state generalization technique to solve this problem. They also provided comparative experimental results for the verification of a number of safety properties using two well-known ROBDD-based verification tools SMV [44] and VIS [9].

Balakrishnan and Tahar [4] verified an Embedded System of a Mouse Controller named PIC 16C71 from Microchip Technology Inc. using MDGs [20]. They modeled the

system at different levels of design hierarchy, i.e., the microcontroller RT level, the microcontroller Instruction Set Architecture (ISA), the embedded software assembly code level and the embedded software flowchart specification. The verification was conducted using equivalence checking and property checking. They detected inconsistencies in the assembly code with respect to the specification during the verification phase.

1.3 Scope of the Thesis

In this thesis, we present a methodology for the formal verification of a real industrial design using Multiway Decision Graphs (MDGs) [20]. The design we considered is a Telecom System Block (TSB) — Receive APS Control, Synchronization Status Extraction and Bit Error Rate Monitor Telecom System Block (RASE TSB) which is a commercial product of PMC-Sierra Inc. [47]. The main aspect of this work is to illustrate the ability of the Multiway Decision Graphs (MDG) tools to carry out a verification process of a large industrial design. Equivalence checking and model checking have been carried for the verification process. The RASE TSB processes a portion of the SONET [6] line overhead of a received SONET data stream. It processes the first STS-1 of an STS-N data stream which can be configured to be received in byte serial format at 6.48 Mbps (STS-1) or 19.44 Mbps (STS-N).

As observed from the examples described in the related work section, the Fairisle ATM switch fabric (with its 4200 equivalent gates) is the largest design to be modeled and verified by the MDG tools. In contrast, the RASE TSB design contains 11400 equivalent gates which is much larger than any other design verified by the MDG tools.

The outline of this thesis is as follows: Chapter 2 gives a brief introduction to Multiway Decision Graphs and its related verification techniques. Chapter 3 illustrates, through simple examples, the hierarchical proof and abstraction methodologies for the modeling and verification of a Telecom System Block (TSB) using MDGs. Chapter 4 describes the functionality of the RASE TSB and applies the proposed modeling and verification approaches on the target TSB. In Chapter 4, we also present a comparison of experimental results obtained using MDG and Cadence FormalCheck. Conclusions and ideas on future work are presented in Chapter 5.

Chapter 2

Multiway Decision Graphs

Multiway Decision Graphs (MDGs), a new class of decision graphs, have been proposed [20] as a solution to the state space explosion problem of ROBDD based verification tools. These decision graphs subsume the class of Bryant's reduced ordered binary decision diagrams (ROBDD) [11], while accommodating abstract sorts and uninterpreted function symbols. MDGs are thus much more compact than ROBDDs which enhances its capability to verify a broader range of circuits [52].

2.1 Multiway Decision Graphs

The underlying formal system of MDGs is a subset of many-sorted first-order logic augmented with a distinction between *abstract sorts* and *concrete sort.* Concrete sorts have finite enumerations, while abstract sorts do not. The enumeration of a concrete sort α is a set of distinct constants of sort α. The constants occurring in enumerations are referred to as individual constants, and the other constants as generic constants and could be viewed as 0-ary function symbols. The distinction between abstract and concrete sorts leads to a distinction between three kinds of function symbols. Let *f* be a function symbol of type $\alpha_1 \times \alpha_2 \times ... \times \alpha_n \rightarrow \alpha_{n+1}$. If α_{n+1} is an abstract sort, then *f* is an *abstract function symbol*. If all the $\alpha_1 ... \alpha_{n+1}$ are concrete, then *f* is a *concrete function symbol*. If α_{n+1} is

concrete while at least one of the $\alpha_1 ... \alpha_n$ is abstract, then *f* is referred to as a *cross-operator*. Concrete function symbols must have explicit definition; they can be eliminated and do not appear in MDGs. Abstract function symbols and cross-operators are *uninterpreted*. An MDG is finite, directed acyclic graph (DAG). An internal node of an MDG can be a variable of concrete sort with its edge labels being the individual constants in the enumeration of the sort; or it can be a cross-term (whose function symbol is a cross-operator). An MDG may only have one leaf node denoted as T, which means all paths in the MDG are true formulae. Thus MDGs essentially represent relations rather than functions. In MDG, a data value can be represented by a single variable of abstract type rather than by concrete (e.g., 32 bits) boolean variables. Variables of concrete sorts are used for representing control signals. Using MDGs, a data operation is represented by an uninterpreted function symbol. As a special case of uninterpreted functions, *cross-operators* are useful for modeling feedback from the datapath to the control circuitry.

In MDG, a state machine is described using finite sets of input, state and output variables, which are pairwise disjoint. The behavior of a state machine is defined by its transition/output relations including a set of reset states. An abstract description of the state machine, called Abstract State Machine (ASM) [21], is obtained by letting some data input, state or output variables be of an abstract sort, and the datapath operations be uninterpreted function symbols. As ROBDDs are used to represent sets of states and transition/output relations for finite state machines (FSM), MDGs are used to compactly encoded sets of (abstract) states and transition/output relations for ASMs. This technique replaces the *implicit enumeration technique* [22] with the *implicit abstract enumeration* [15].

The notion of *abstract description of state machines* is hence a theoretical fundament for a verification methodology that makes it possible to verify sequential circuits automatically at the RT level using abstract sort and uninterpreted function symbols. In this sense, we can say that the verification method is applicable to designs where the data operations are viewed as black boxes [57]. Such a verification process fits well in the verification of RTL designs generated by high-level synthesis. This is because high-level synthesis algorithms schedule and allocate data operation without being concerned with the specific nature of operations. In the next sections, we describe the modeling and verification features of the MDG tools. Interested readers are referred to [15, 20, 21, 54, 56, 57, 59] for more details on the MDG algorithms and tools.

2.2 Modeling Hardware with MDGs

MDGs describe circuits at the RT level as a collection of components interconnected by nets that carry signals. Each signal can be an abstract variable or a concrete variable. The input language for MDG based applications is a Prolog-style HDL, called MDG-HDL. This hardware description language allows the use of abstract variables for representing data signals and uninterpreted function symbols for representing data operations. MDG-HDL supports structural description, behavioral ASM descriptions, or a mixture of structural and behavioral descriptions. A structural description is usually a netlist of components connected by signals. A behavioral description is given by a tabular representation of the transition/output relation or the combinational function block. A complete reference of MDG-HDL can be found in [59].

For logic gates, the input and output signals are always of concrete sort, i.e., boolean type. Figure 2.1 shows an OR gates and its MDG representation for a specific ordering of the variables. Boolean MDGs are essentially the same as ROBDDs.

Feedbacks from datapath to control circuitry are represented using *cross-operators*. Figure 2.2(a) shows a comparator that used for the control portion of a circuit. The comparator produces a control signal *y* from two data inputs x_1 and x_2. Both x_1 and x_2 are variables of abstract sort while *y* is a boolean variable. An uninterpreted function symbol *eq* of type $[wordn, wordn] \rightarrow bool$ [1] is used to denote the functionality of this comparator. If the meaning of *eq* matters, rewrite rules, such as $eq(x, x) \rightarrow 1$ should be used. An MDG of the comparator is shown in Figure 2.2(b).

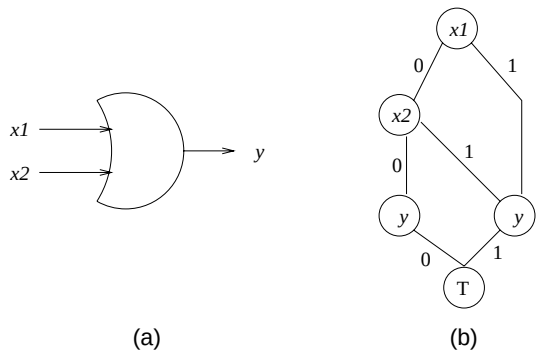


Figure 2.1: The MDG for an OR gate

1. The notation *f*: $[\alpha \rightarrow \beta]$ implies that the function *f* has argument of sort α and range of sort β.

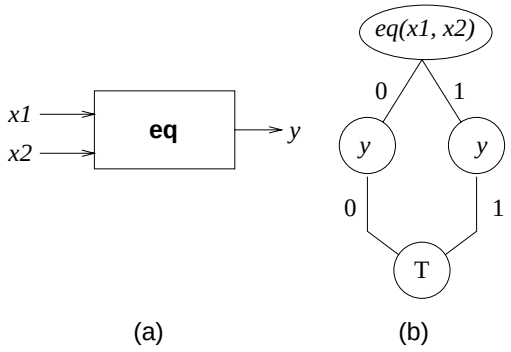
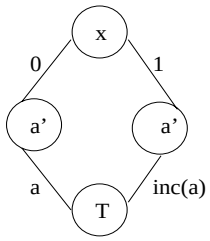


Figure 2.2: The MDG for a comparator

Using MDGs tabular construct, we can represent an ITE (If -Then-Else) or CASE formulas. It is analogous to directed formula where a row gives the symbolic values of the variables in the head of the table as a disjunct of the directed formula. Figure 2.3(a) shows a tabular description of a sample ASM, where *x* is a boolean input, *a* is an abstract state variable and *a'* is its next state variable. It performs *inc* operation (an uninterpreted function) when *x* = 1. Figure 2.3(b) shows its MDG representation.

x	a'
0	a
1	inc(a)

(a)



(b)

Figure 2.3: The MDG for an ASM

2.3 MDG-based Verification Techniques

The MDG software package includes algorithms for *disjunction*, *relational product*, *pruning-by-subsumption (PbyS)*, and *reachability analysis*. Except for *PbyS*, the operations are a generalization of first-order terms of algorithms on ROBDD, with some restrictions on the appearance of abstract variables in the arguments. In the reachability analysis procedure, starting from the initial set of states, the set of states reached in one transition is computed by the relational product operation. The frontier set of states is obtained by removing the already visited states from the set of newly reached states using the pruning-by-subsumption (*PbyS)* operation. If the frontier set of states is empty, then the reachability analysis procedure terminates, since there are no more unexplored states. Otherwise the newly reached states are merged (using disjunction) with the already visited states and the procedure continues where the next iteration with the states in the frontier set as the initial set of states. A facility to carry out simple rewriting of terms that appear in the MDGs is also included. This allow us to provide a partial interpretation of an uninterpreted function symbol. For example, if *zero* is an abstract generic constant of sort *wordn* and *eqz*(*x*) a cross-operator of type [*wordn*→*bool*], then we could provide a partial interpretation of *eqz* using the rewrite rule *eqz*(*zero*)→*1,* indicating that *equal-to-zero* is 1 when the argument is *zero* (but not revealing anything about the other values). User selected rewrite rules are applied anytime a new term is formed during MDG operations. In general, rewriting simplifies MDGs and helps to remove false negatives during safety property checking. Thus likely avoiding non-termination of the *reachability analysis* procedure for designs that depend on interpretation of operators for correct operation. A detailed description of the operations and algorithms can be found in [15, 20, 21, 57]; some possible solutions to

the non-termination problem are addressed in [1, 58]. The following sub-sections describe the verification techniques provided by the MDG tools.

2.3.1 Combinational Equivalence Checking

The MDGs representing the input-output relation of each circuit are computed using the relational product of the MDGs of the components of the circuits. Then taking advantage of the canonicity of MDGs, it is verified whether the two MDG graphs are isomorphic. Using this technique, we can verify the equivalence of two combinational circuits. This technique can also be used to compare two sequential circuits when a one-to-one correspondence between their registers exists, e.g., equivalence checking between RTL model and gate level netlist of a design. However, combinational equivalence checking cannot handle the equivalence checking between RTL and behavioral models because these models are developed separately and it is not possible to map each register in the RTL model to that of the behavioral model.

2.3.2 Invariant Checking

Using the symbolic *reachability analysis* technique, the state space of a given sequential circuit is explored in each state. It is verified that an invariant, i.e., a logical expression, holds over all reachable states. The transition relation of an ASM is represented by an MDG computed by the *relational product* algorithm from the MDGs of components which are themselves abstract machines. In other words, the relational product computes the (synchronous) product machine of the components ASMs. Using invariant checking, we can verify safety properties of a digital system.

2.3.3 Sequential Equivalence Checking

The behavioral equivalence of two sequential circuits can be verified by checking that the circuits produce the same sequence of outputs for every sequence of inputs. This is achieved by forming a circuit from two circuits feeding the same inputs to both of them and verifying an invariant asserting the equality of the corresponding outputs in all reachable states. It can verify the equivalence between RTL and gate level netlist or RTL and behavioral model which is very important in design verification. The drawback of this technique is that it cannot handle a large design due to state space explosion problem. With the increasing number of state components in synchronous digital design, the state space grows exponentially, which is more severe in the product machine generated for sequential equivalence checking.

2.3.4 Model Checking

MDG model checker provides both safety and liveness property checking facilities using the implicit abstract enumeration of an abstract state machine [54]. In MDG model checking, the design is represented by an ASM and the properties to be verified are expressed by formulae in the first-order ACTL-like temporal logic called L_{MDG} [55]. The ASM model of L_{MDG} is composed of the original model along with a simplified invariant [55], and the simplified invariant is checked on the composite machine using the implicit abstract enumeration of an ASM. However, only universal path quantification is possible with the current version of MDG model checker.

2.3.5 Counterexample Generation

Using counterexamples, a user can trace errors in the design during a verification process. If an invariant is violated at some stages of the reachability analysis, a counterexample facility generates a sequence of input-state pairs leading from the initial state to the faulty behavior.

Chapter 3

Modeling and Verification Methodology

Abstraction and hierarchical proof can simplify the verification process vastly. To accelerate the design flow and assure the correctness of a complex digital system, a hierarchical design approach shown in Figure 1.1 (Chapter 1) is usually anticipated. This approach has many advantages and is commonly used in practice. It is particularly useful in the context of formal verification. In addition to hierarchical proof, one must use abstraction mechanisms for relating formal descriptions of hardware designs at different levels of design hierarchy [45]. The following two sub-sections will present more about these two methodologies and their applications to the modeling and verification of a Telecom System Block (TSB) using MDGs.

3.1 Hierarchical Proof Methodology

When a design to be proved correct is large, formal method is usually applied hierarchically [24]. The design is structured into a hierarchy of modules and sub-modules, and specifications that describe “primitive components” at one level of the hierarchy then become specifications of the intended behavior at the next level down. The structure of the proof mirrors this hierarchy: the top-level specification is shown in Figure 3.1 to be satisfied by an appropriate connection of modules. At the next level down, each of these modules is shown to be correctly implemented by a connection of sub-modules, and so on — down to the lowest level of the hierarchy. Hierarchical organization of a design not only

makes the verification process natural, it also makes the task tractable. Dealing with the complexity of a complete system description of even modest size, by standards today, is out of bounds for most verification techniques. By splitting this large problem into smaller pieces that can be handled individually, the verification problem is made mangeable. It effectively increases the range of circuit sizes that can be handled in practice.

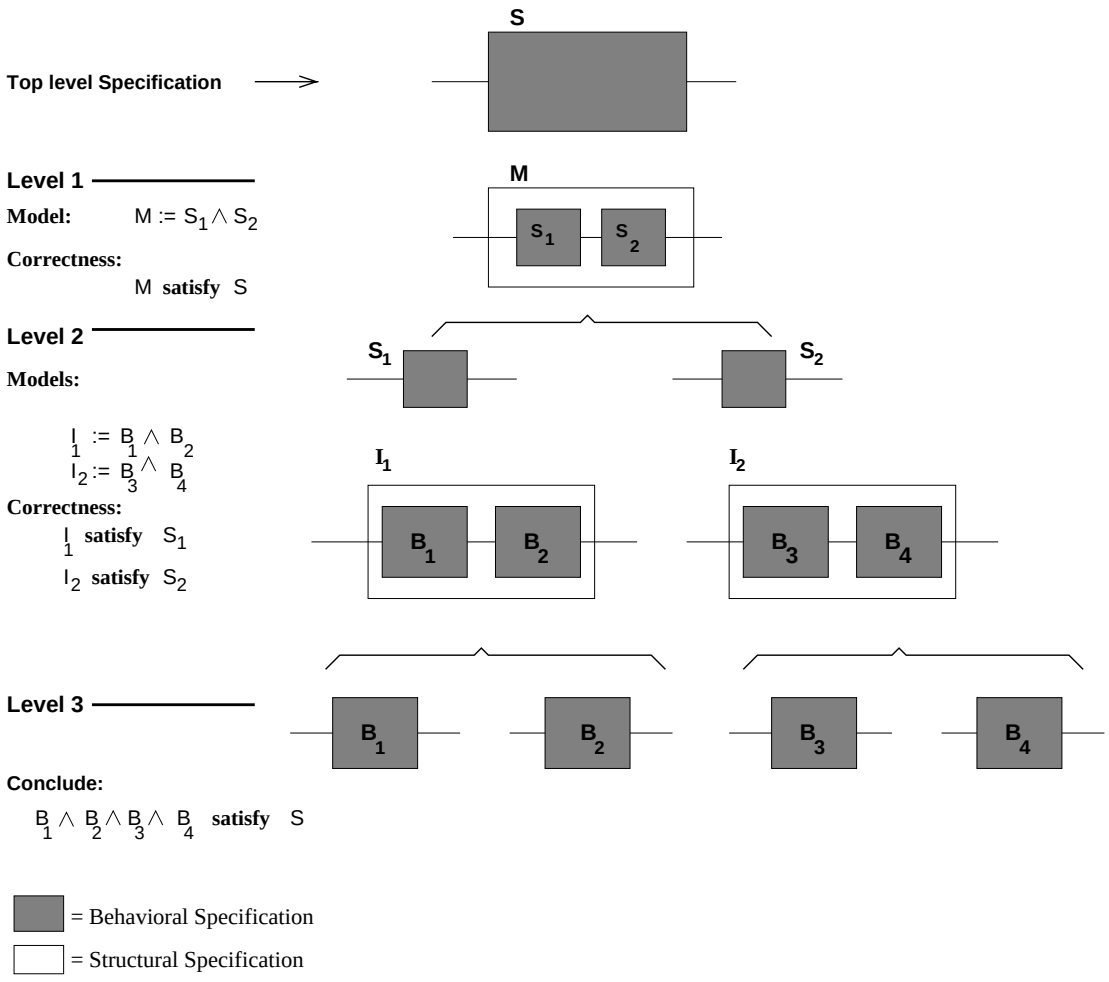


Figure 3.1: An example of hierarchical proof

We can illustrate the hierarchical proof methodology by using Figure 3.1 with a small example. The design we considered is structured into a three-level hierarchy of modules.

At the top level, i.e., Level 1, there are two modules S_1 and S_2, interconnected by internal signals. At this level these modules are considered to be primitive devices. The description M implements the behavior of the entire system at this level. It is constructed by composing the modules S_1 and S_2 and hiding the internal signals. The correctness statement at this level of the proof asserts that the model M satisfies the specification of the whole system, represented by symbol S. At the next level down, i.e., Level 2, S_1 and S_2 become specifications of required behavior for the two sub-systems implemented by I_1 and I_2. These models are constructed from the specifications of the primitive modules at Level 3, i.e., B_1, B_2, B_3 and B_4. At level 2, we have two separate correctness theorems to prove (see Figure 3.1). These theorems assert that the sub-systems modeled by I_1 and I_2 correctly implement the abstract behaviors given by the specifications S_1 and S_2, respectively. We can conclude from this, and from the correctness result for M proved at the top level, that integrating the two sub-systems modeled by I_1 and I_2 give a concrete implementation of the entire design which is correct with respect to the top-level specification S.

A hierarchical proof of correctness usually has many intermediate levels between the concrete design model and the top-level specification. At each level, correctness theorems relate each sub-module to an abstract specification at the next higher level. In general, there may be many separate theorems at each level, one for each different kind of module used at that level. To obtain a completed proof, we must integrate all these intermediate correctness results into a single correctness theorem that relates a fully concrete model of the entire design to the top-level specification of the intended behavior. There are three separate theorems in the hierarchical proof shown in Figure 3.1.

They are as follows:

Theorem 1: M ***satisfies*** S

Theorem 2: I_1 ***satisfies*** S_1

Theorem 3: I_2 ***satisfies*** S_2

Proving these theorems shows only that each module in the hierarchy is correct with respect to its specification. To complete the proof, we must derive a theorem stating the correctness of the entire design with respect to the top-level specification. To do so, the following theorem must be proved.

Theorem 4: $B_1 \wedge B_2 \wedge B_3 \wedge B_4$ ***satisfies*** S

This theorem states that a complete and fully detailed design model constructed from the primitive modules B_1, B_2, B_3, and B_4 satisfies the top-level abstract specification S. This hierarchical approach to hardware verification is possible in logic because design models and specification use the same language (syntax). Both of them are simply boolean terms, and the model-building operation of composition ($\wedge$) can be applied to both of them. Terms used as abstract specifications at one level in a hierarchical proof can therefore be treated as models at the next higher level. In a formalism in which specifications and models are syntactic entities of two distinct types, this direct approach to hierarchical verification is not possible.

This hierarchical proof approach has the added advantage that if we reuse sub-modules in the other modules we do not repeat work unnecessarily. Also if we change the implementation of the some modules, we do not need to reverify the whole design. We just need to prove correctness theorems for the new implementations of the modules and recombine the correctness theorems. A further advantage is that separate subtrees of the design can be

verified independently by different people. The interface between teams occurs at the point where the subtrees are joined. Here, provided the behavioral specifications of the modules are agreed on, the upper levels can also be verified independently

3.2 Abstraction and Reduction Techniques

As the complexity of a functional blocks increase, the default setting used by most formal verification tools may not be sufficient. We need some kind of abstraction techniques in order to reduce the state space of the design under verification. The more extensive the reachable states, the more CPU time and memory it takes to verify a system. Abstraction is the process by which the important properties of a complex object are isolated for further use and the remaining ones ignored as being irrelevant to the task at hand. An example is the process of procedural abstraction in high level programming languages. Programming languages support this abstraction for dealing with the complexity of programming. In a similar way, abstraction plays an important role in hardware verification. Here, an abstraction mechanism establishes a relationship of abstraction between a complex description of hardware behavior and simpler one. This provides a means for controlling the complexity of both specifications and proofs of correctness. By suppressing the irrelevant information in detailed descriptions of hardware behavior, and thereby isolating the properties of these descriptions which are most important. An effective abstraction and reduction mechanism helps to reduce the size and complexity of the design at each level in the hierarchically-structured design. There are four different kinds of abstraction techniques used in hardware formal verification [45].

3.2.1 Behavioral Abstraction

Behavioral abstraction involves proving the correctness of designs with respect to partial specifications of intended behavior. *Partial* specification does not completely define the full range of behavior that a system can exhibit, but only defines its behavior in environments or states that are of particular interest. In logic, specifications are expressed by constraints on the values allowed on the external signals of a design. The *range of behavior* defined by a specification is given by the set of values that satisfy these constraints. A *partial* specification constraints a design's signals to have certain values in situations that are significant or relevant, but leaves unconstrained the signal values in all other situations. This means that in the situations of "undefined" behavior, the predicate defining a partial specification will be satisfied by signals values that would not be allowed by a more complete specification of the system. Thus, the partial specification of a system imposes weaker constraints on its signal values than a complete specification would. For example, in a target system there are several components, e.g., register file and input/output multiplexor, in addition to the main block which controls the main functionality of the system. Also assume that only the verification of the main block is of interest. In that case, the additional blocks can be removed from the top-level of the design, provided that this removal does not change the behavior of the system. The inputs of the main block which are fed by the removed blocks, can be set to primary inputs of the system.

3.2.2 Structural Abstraction

Structural abstraction is the most fundamental abstraction technique to hardware verification. It suppresses information about a design's internal structure, i.e., only the behavior

of the external inputs and outputs of a module is of interest. The basic idea of structural abstraction is that the specification of a device should not reflect its internal construction. To illustrate this abstraction technique, we can have an example of a *BYTE_EXTRACTOR* from our case study (see Chapter 4). To keep our description simple, we are taking only a portion of the byte extraction circuit.

In Figure 3.2, the behavioral description of the *BYTE_EXTRACTOR* does not contain any information about its internal structure. It reflects only the characteristics of the external signals. All the internal signals, (e.g., *c1*, *c3*, *cr1*, *cr3*, *cr* and *s1*), are hidden in the description of specification. But the implementation contains explicit information about the internal structure.

Specification in HDL model:

```
case r1 = 1:
    if (st1 and cl1) then
        next(out[7:0]) = in[7:0]
    else next(out[7:0]) = “00000000”
case r3 = 1:
    if (st3 and cl3) then
        next(out[7:0]) = in[7:0]
    else next(out[7:0]) = “00000000”
```

Structural Model of *BYTE_EXTRACTOR*:

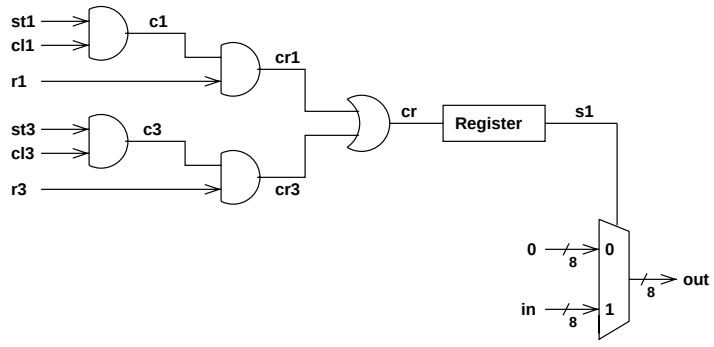


Figure 3.2: An example of structural abstraction

3.2.3 Data Abstraction

The formal description of a system can be more abstract than its realistic model. The specification of intended behavior for a system may be expressed in terms of an abstract notion of the types of values it operates on. The free variables in such a formal description will not stand for the values actually present on the external wires of a design. Instead it will represent more abstract externally observable quantities. The description of a system can be expressed in terms of operations appropriate to these abstract quantities, rather than the operations carried out by the actual hardware on a more concrete representation of these values. The logical types of the variables that represent these abstract values will therefore generally differ from those of the variables in the original design model. A satisfaction relation based on data abstraction must therefore relate concrete variables of one type in the model to more abstract variables of another type in the specification. Both the model and specification express a constraint on free variables that directly correspond to physical wires, but use different logical types to represent the range of values that can appear on them. The model and specification will then be terms of the forms $M[c_1, c_2, . ., c_n]$ of type f_c and $S[a_1, a_2, . ., a_n]$ of type f_a, respectively. In this case, each variable a_i in the specification represents the same externally observable value as the corresponding variable c_i in the implementation. The specification, however, is expressed as a constraint on abstract values of type f_a, instead of the concrete values of type f_c that represent actual physical values in the implementation. To formulate a correctness statement that relates these two specifications, we need an appropriately-defined data abstraction function to map these two different descriptions. Given such a mapping function f: $f_c \rightarrow f_a$, a correctness

statement which expresses a relationship of data abstraction between the model and the specification can be formulated by the following theorem [45]:

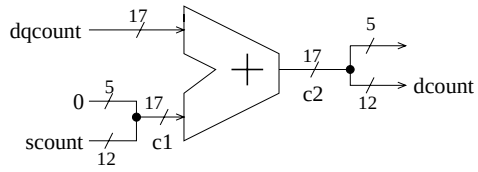
$M[\ c_1, c_2, ..., c_n\ ]$ **satisfies** $S[\ f(c_1), f(c_2), ..., f(c_n)\ ]$

This theorem states that every combination of values $c_1, c_2, ., c_n$ that corresponds to the model *M*, actually appears on the external wires of the system is a concrete representation at a lower level of data abstraction for a combination of more abstract values $f(c_1), f(c_2), ., f(c_n)$ which is allowed by the specification of that system. The resulting correctness statement asserts that the operations on concrete variables actually carried out by the model correctly implement the required operations on abstract values expressed by the specification. The advantage of data abstraction is that it allows specifications of intended behavior to be written in terms of abstract *high-level* operations on data, without having to specify precisely how this data is represented.

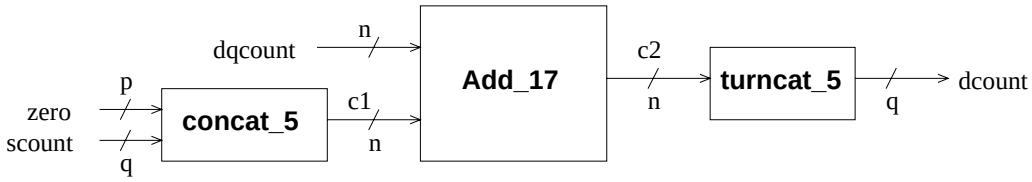
In this thesis, we use MDG-HDL, which is the input language of the MDG tools, to model the design under investigation. One of the major advantages in using MDGs is the ability to handle abstract descriptions. This avoids all the ponderous procedure of defining each bit of a vector of boolean variables. Rather, a vector of boolean variable can be viewed as a single abstract variable. Thus a 24-bit frame-counter can be modeled as a variable of abstract sort *worda24* instead of a concrete sort with enumeration {0, 1, . . ., 16777215}. Another advantage in using MDGs is the ability to represent data operations by *uninterpreted* function symbols. This enables the arithmetic and logical blocks to be viewed as black boxes. As a special case of uninterpreted function, *cross-operators* are useful for modeling feedback from the datapath to the control circuitry. We can illustrate the idea of data abstraction in MDGs using an example (see Figure 3.3) from our case study. The circuit in the example is performing data operations

over two operands of different size. It is concatenating 5-bits for matching the size of the operands to be used for addition and extracting twelve bits from the least significant bit positions of the output by truncating the upper bits. Using MDG-HDL, we can abstract the width of the datapath as well as the functionality of the original model. The data operations (e.g., addition, concatenation) can be modeled using *uninterpreted* function symbols (e.g., *Add_17* of type $[worda17, worda17] \rightarrow worda17$), applied to the operands.

Original Structural Model:



Abstracted Model:



MDG-HDL model:

```
Definitions:
   var(dqcount,worda17)
   var(scount,worda12)
   var(c1,worda17)
   var(c2,worda17)
   var(dcount,worda12)
   constant(zero,wordn)
   f(add_17,inputs[worda17,worda17],output[worda17])
   f(concat_5,inputs[worda12,wordn],output[worda17])
   f(turncat_5,inputs[worda17],output[worda12])
HDL Model:
  concat_5(inputs([zero,scount]), output(c1))
  add_17(inputs([dqcount,c1]), output(c2))
  turncat_5(inputs([c2]), output(dcount))
```

Figure 3.3: An example of data abstraction

3.2.4 Temporal Abstraction

In temporal abstraction, the sequential or time-dependent behavior of a system is viewed at different grains of discrete time. An example of temporal abstraction is the unit delay or register, implemented by an edge triggered flip flop. At the abstract level of description the device is specified as a unit delay, i.e., one unit of discrete time corresponding to the clock period. This type of abstraction technique has not been applied to our case study as the design is not pipelined and every function is taking one clock period to complete.

3.2.5 Model Reduction

In our case study (Chapter 4), we applied the model reduction technique to abstract the behavior of the telecom system block by removing some modules, provided that the main functionality of the system will not be changed. Besides five main functional blocks, the design has one common-bus-interface module and two input/output multiplexors. The common-bus-interface module is used mainly for configuration and testing the interface of the telecom system block. This module contains several read/write registers to store the outputs of other four functional blocks and the interrupt signals are generated by reading these registers. The input/output multiplexors are used for simulation purposes. In our modeling and verification of the Telecom System Block, we can eliminate these three modules which have no effect on the functionality of the system. The system is modeled in such a way that all the intermediate signals between the common-bus-interface and other five modules are to be considered as primary inputs/outputs of the system. We can

conclude from the above descriptions that model reduction of the telecom system block does not change its main functionality at all.

Chapter 4

Modeling and Verification of RASE TSB: A Case Study

The commercial design that is investigated in this work is a Telecom System Block (TSB) named **R**eceive **A**utomatic Protection Switch Control, **S**ynchronization Status **E**xtraction and Bit **E**rror Rate Monitor (RASE) which is commercialized by PMC-Sierra Inc. [47]. In this Chapter, we apply the hierarchical verification and abstraction methodologies described in Chapter 3 on this TSB. We also present experimental results using MDG and FormalCheck.

4.1 The RASE Telecom System Block

The RASE Telecom System Block (TSB) [47] consists of three types of components: Transport overhead extraction and manipulation, Bit Error Rate Monitoring (BERM) and Interrupt Server (see Figure 4.1). In addition to these blocks, it has an interface with Common Bus Interface (CBI) block which is used mainly for the configuration and testing the interface of the TSB and two inputs/outputs multiplexors. The transport overhead extraction and manipulation functions are implemented by three sub-modules (transport overhead bytes extractor, automatic protection switch control and synchronization status filtering). In this study, all the above modules are of interest except the CBI block and inputs/outputs multiplexer which were used for simulation purposes.

The RASE TSB extracts the Automatic Protection Switch (APS) bytes, i.e., K1 and K2 bytes, and the Synchronization Status byte, i.e, S1 byte, from a SONET frame (see Figure 4.2). After extracting the above bytes, it processes them according to some requirements set by the SONET standard [6].

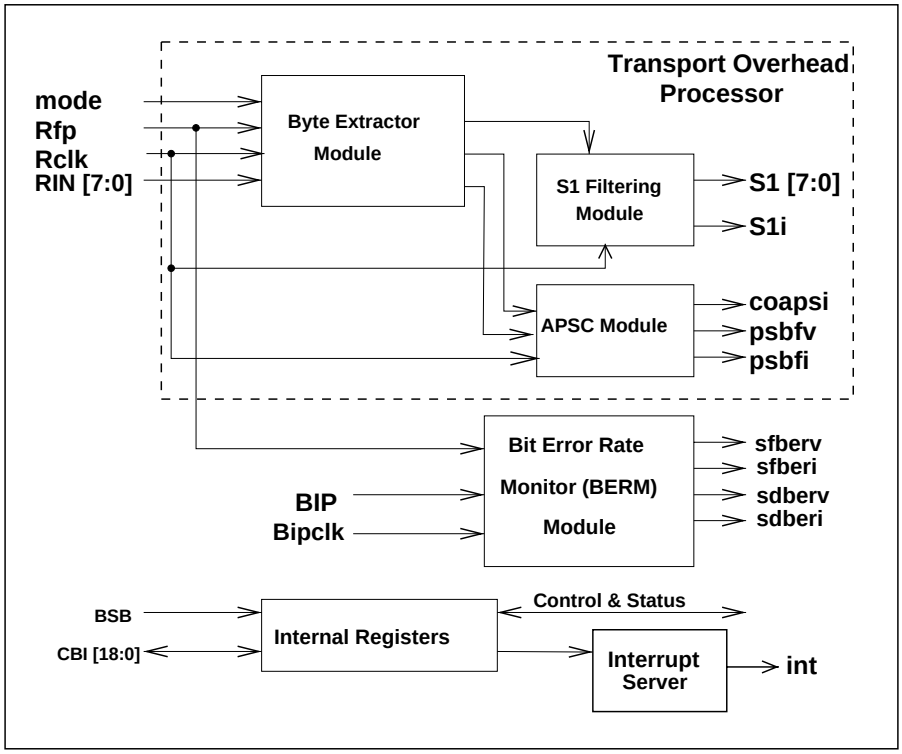


Figure 4.1: The RASE telecom system block

The TSB also performs Bit Error Rate Monitoring using the BIP-24/8 line of a frame, i.e., B2 bytes (Figure 4.2). The received line Bit Interleaved Parity (BIP) error detection code is based on the line overhead and synchronous payload envelope of the received data stream. The line BIP code is a bit interleaved parity calculation using even parity. The calculated BIP code (pre-defined by programmable registers) is compared with the BIP code extracted from the B2 bytes of the following frame. Any differences indicate that a

line layer bit error has occurred and an interrupt signal will be activated in response to this error. A maximum 192000 (24 BIP/frame x 8000 frames/second) bit error can be detected for Synchronous Transport Signal (STS) -3 rate and 64000 (8 BIP/frame x 8000 frames/ second) for the STS-1 rate. The RASE TSB contains two BERM blocks. One BERM is dedicated to monitor for the Signal Failure (SF) error rate and the other BERM is dedicated to monitor for the Signal Degrade (SD) error rate. They work on the same module and offer the same functionality.

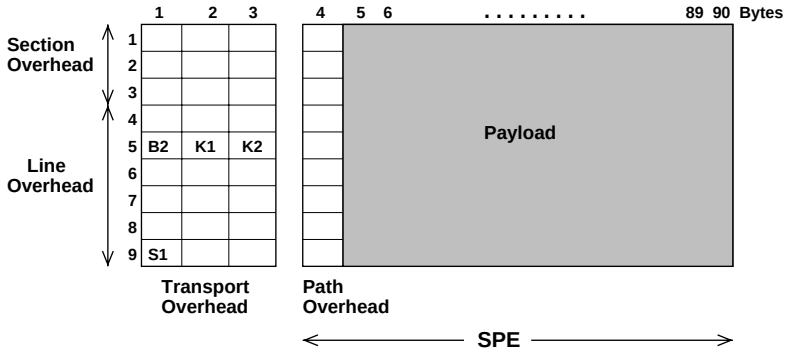


Figure 4.2: The STS-1 SONET frame structure

The Automatic Protection Switch (APS) control block filters and captures the receive automatic protection switch channel bytes (K1 and K2), allowing them to be read via CBI bus. These bytes are grouped and filtered for 3 frames before being written to these registers. A protection switching byte failure alarm is declared when 12 successive frames have been received without 3 consecutive frames having the same APS bytes. When 3 consecutive frames have identical APS bytes, the alarm will be removed. The detection of invalid APS codes is done in software by polling the APS K1 and K2 registers, which is not of interest in the current study.

The synchronization status filtering block captures and filters the S1 status bytes, allowing them to be read via CBI bus. This block can be configured to capture the S1 nibble of eight consecutive frames and filters the nibbles/bytes for the same value. It can also be configured to perform filtering based on the whole S1 byte.

The interrupt server activates an interrupt signal if there is a change in APS bytes, a protection switch byte failure, a change in the synchronization status, or a change in the status of Bit Error Rate Monitor (BERM) occur.

The Common Bus Interface (CBI) block provides the normal and test mode registers. The normal mode registers are required for normal operation while the test mode registers are used to enhance the testability of the TSB. The input test multiplexer selects normal or test mode inputs to the TSB. The output test multiplexer selects the outputs modes.

4.2 Behavioral Modeling of the TSB using MDGs

The description of a system can be a specification or an implementation. A specification refers to the description of the intended behavior of the hardware design. An implementation refers to the hardware design of the system which can be at any level of the design, i.e., in RT level or gate level netlist. In the MDG system, an abstract description of a state machine (ASM) can be used to describe a specification or an implementation. We adopt a hierarchical approach to model the TSB behavior at different levels of the design hierarchy which in turn enables the verification process to be done at different levels. Figure 3 represents a tree showing the level of design hierarchy of the RASE TSB.

Inspired by [47], we derived a behavioral model of the RASE TSB which consists of five main functional blocks — Transport Overhead Extractor, Automatic Protection Switch,

Synchronization Status Filtering, Bit Error Rate Monitoring and Interrupt Server. These are the basic building blocks of the TSB. We composed the behavioral model of each basic building block in a bottom-up fashion until we reached the top-level specification of RASE telecom system block. In the following sub-sections, we represent the behavioral model of each basic module of the TSB which will be composed to form the complete behavior of the TSB.

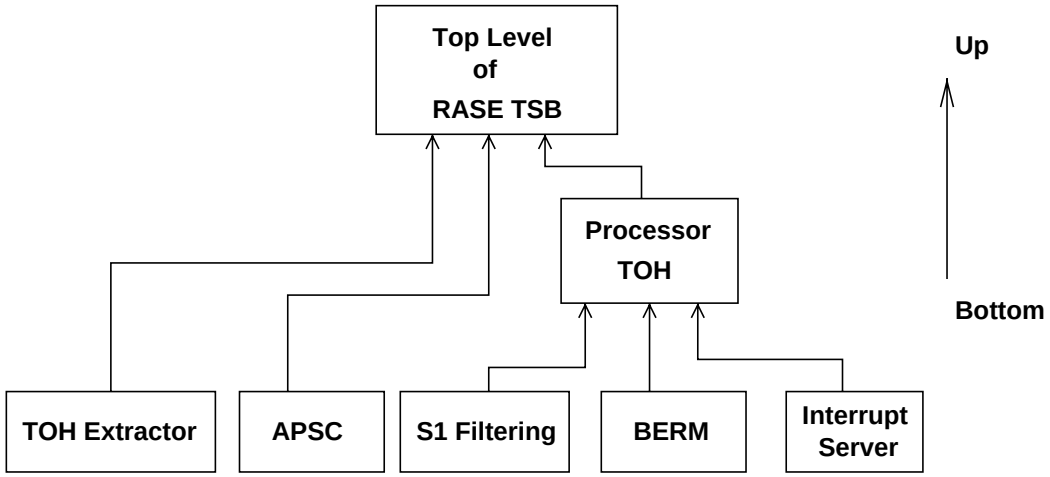


Figure 4.3: The hierarchy tree of the TSB

Examples of sorts and uninterpreted functions that are used to model the *RASE TSB* are as follows:

- concrete sort *bool* = {0, 1}.
- abstract sort *worda8* (used to represent 8-bits word).
- generic constant *zero* of sort *wordn.*
- cross-operator *eq_ex* of type $([worda8, worda8] \rightarrow bool)$ is used to compare the equality.

- uninterpreted function symbol *inc* of type $[worda8 \rightarrow worda8]$ is used as an incrementer of 8-bits.
- cross-operators *bit0,.., bit3* of type $([worda8] \rightarrow bool)$ are used to extract the boolean value from an abstract variable.

4.2.1 Transport Overhead Extraction

To derive the behavior of transport overhead extraction, we need to have a look into the structure of a SONET frame in Figure 4.1 and the locations of S1, K1, K2 and B2 line overhead bytes within that frame. The basic signal of SONET is the STS-1 electrical signal. The STS-1 frame format is composed of 9 rows of 90 columns of 8-bits bytes, in total 810 bytes [6]. The byte transmission order is row-by-row, left to right. At a rate of 8000 frames per second, that works out to a rate of 51.84 Mbps. The STS-1 frame consists of Transport Overhead (TOH) and Synchronous Payload Envelope (SPE). The Transport Overhead is composed of Section Overhead (SOH) and Line Overhead (LOH). The SPE is divided into two parts: the STS Path Overhead (POH) and the Payload. The first three columns of each STS-1 frame make up the TOH and the last 87 columns make up the SPE. The SPE can have any alignment within the frame and this alignment is indicated by the pointer bytes in the LOH which is not of our interest in this work. The behavior of the extraction module is based on a row and a column counting abstract state machine (ASM) rather than finite state machines and an extractor which extracts the specific byte (i.e., *RIN*, receive input data stream) within a SONET frame. The column counting ASM has five states — S0, S1, S2, S3 and S4 (see Figure 4.4), while the row counting ASM has three states — S0, S1 and S2 (see Figure 4.5). The symbols '&&', '||', and '~' in all the figures, denote logical AND, OR and negation of the signals, respectively.

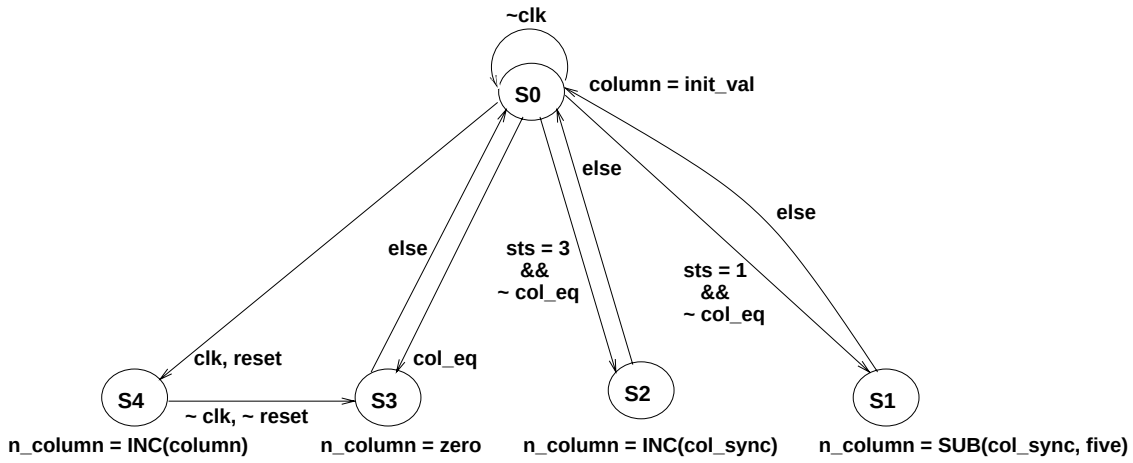


Figure 4.4: The column counting ASM

The column counting state machine accepts five signals *clk, rst, rfp, sts* and *col_eq* as its input control signals (see Figure 4.4). The presence and absence of a frame is indicated by *rfp* signal at high. A SONET frame can be either in STS-1 or STS-3 mode which is indicated by the signal *sts.* An abstract state variable *column* represents the current count number of the columns. At each time, the counting state machine has a transition to different state according to the control input signals. In this abstract description of the counter, the count *column* is of an abstract sort, say *wordn*. The input control signals, (e.g., *clk, rst, rfp* and *sts),* are of concrete sort *bool* with the enumeration {1, 0}. The uninterpreted function *inc* of type $[worda8 \rightarrow worda8]$ denotes the increment-by-one operation. The cross-operator *eq_ex(column,constant_signal)* of type $([worda8, worda8] \rightarrow bool)$ is used to model the feedback to the column counting state machine. This cross-operator represents a comparator which accepts two operands of abstract sort, i.e., *column* and *constant_signal,* and sets the control signal *col_eq* ='1' whenever the inputs are equal. State S0 is the reset state from there can be four transitions

depending on the input control signals. In state S1, a data operation will be performed to adjust the column number as per frame modes and the result of the operation will be assigned to the count value. In state S2, the constant signal *column_sync* will be incremented by one to adjust the frame mode i.e., STS-1, STS-3 and the incremented value will be assigned to the count value. State S3 is the counter roll-over state which depends on the control signal *col_eq* and *reset*. In state S4, the counter will be incremented by one in each clock cycle if no other transitions are possible and it will remain in that state unless a transition is possible to other state depending on the inputs.

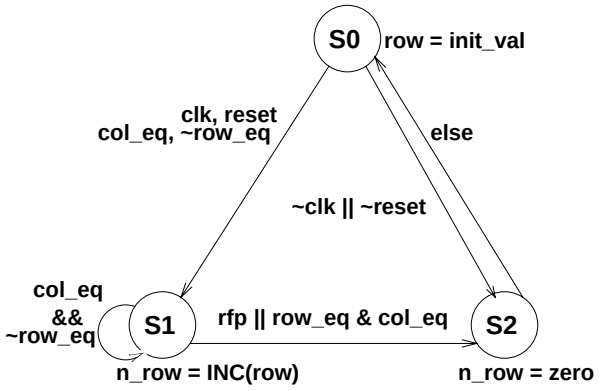
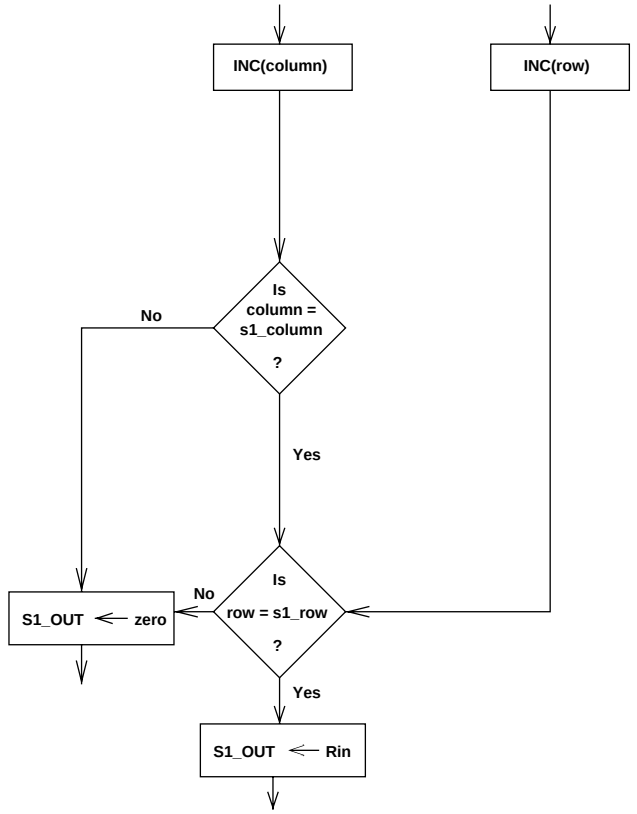


Figure 4.5: The row counting ASM

The row counting state machine, having three states, uses *col_eq* and *row_eq* control signals generated by the *eq_ex* cross-operator to increment or roll-over the row counting variable *row* (Figure 4.5). State S0 is the initial state where the variable *row* initialize by its reset value *init_val*. Any frame start pulse *rfp* or both *col_eq* =’1’ and *row_eq* =’1’ makes a transition to state S2 where state variable *row* assigned to be *zero* which is a generic constant of abstract sort *wordn.* The abstract state variable *row,* in state S1, will be incremented by one using the uninterpreted function symbol *inc* of type

[$worda8 \rightarrow worda8$]. Whenever two control signals *col_eq* ='1' and *row_eq* = '0', given by the cross-operators *eq_ex(column_count, constant_signal)* and *eq_ex(row, constant_signal)*, the abstract state variable *row* increments by one. The symbol '||', '&&' and '~' denote logical OR, AND and negation of the signals, respectively.

Flowchart:



MDG-HDL model:

```
initialize column = zero
initialize row = zero
next(column) = INC(column)
next(row) = INC(row)
if (column = s1_column)
   then if (row  = s1_row)
         then next(S1_out) = RIN
         else next(S1_out) = zero
   else next(S1_out) = zero
```

Figure 4.6: Flowchart specification of byte extractor and its MDG model

The behavior of an extractor can be described using a flowchart and its model in pseudo-MDG-HDL as shown in Figure 4.6. We can see from Figure 4.6 that the extraction of the line overhead byte from a frame is performed by comparing the count values, from the column and row counting ASMs (Figures 4.4 and 4.5), with two constant values representing the index of byte's location within a frame. The transport overhead bytes (i.e., S1, K1 and K2) are extracted from the received data stream (*RIN*) of a SONET frame. In Figure 4.6, the column and row counters initialized by variables *a* and *b* on each clock cycle and compared with an index to locate S1 byte within a SONET frame. If row_eq and col_eq are true then the byte will be extracted from a SONET frame. Otherwise the value of S1_byte will be zero and the value of both counters will be increased by one.

An abstract state machine can have an infinite number of states due to the abstract variable and the uninterpreted nature of the function symbols. The reachability analysis algorithm of MDGs is based on the abstract implicit state enumeration. The major draw back of this algorithm is that a least fixed point may not be reached during reachability analysis. Because of this limitation, a *non-termination* of abstract state enumeration may occur when computing the set of reachable states. To illustrate this limitation of MDG-based verification, we can have an example of Figure 4.6, where state variables *column* and *row* of abstract sort represent the column and row counter of a SONET frame, a generic constant *zero* of the same abstract sort denotes the initial value of *column* and *row*, and an abstract function symbol *INC* describes how the counters are incremented by one. The MDG representing the set of reachable states of the column/row counting ASM (see Figures 4.4 and 4.5) would contain states of the form

(*row, INC*(. . . *INC*(*zero*). . .))

for the number of infinite iterations. As a consequence, there is no finite MDG representation of the set of reachable states and the reachablility algorithm will not terminate. This typical form of non-termination can be avoided by using some techniques described in [1, 20, 58]. For instance, in [20] the authors present a method based on the *generalization* of initial state that causes divergence, like the variable *column*/*row* in Figure 4.6. Rather than starting the reachability analysis with an abstract constant *zero* as the value of column/row, a *fresh* abstract variable (e.g., *a* or *b*) is assigned to *column*/*row* at the beginning of the analysis. As a consequence, the initial set of states represented by column/ row thus represents any state, hence any incrementing of *column*/*row* leads the ASM to a state where the new value of *column*/*row* is an instance of its arbitrary value of the initial state. We can also terminate the reachability analysis of abstract implicit state enumeration by using rewrite rules for the *cross-operator*. Rewrite rules for *cross-operators* shrink the size of MDG. For example, if there is a path in an MDG which has *cross-operator eq_ex*(X, X) = 0 with *eq_ex* intends for equality. We can use the cross-term rewrite rule *eq_ex*(X, X) `->1` to eliminate this path in the MDG.

In [58], the authors present a heuristic state generalization method based on the following observation: reachability analysis terminates if we generalize any state within a processor-like loop. Once we generalize a state in a loop, it covers all the abstract states having the same control state values, thus a termination is possible. In [1], the authors propose another solution to the non-termination problem based on retiming and additional circuit transformations that preserve the design's hebavior. In retiming, the registers are placed in appropriate positions, so that the critical paths they embrace are as short as

possible. Moreover, the retiming corresponds to minimizing the overall number of registers.

4.2.2 Automatic Protection Switch Control

The Filtering and Triggering behavior of the Automatic Protection Switch Control (APSC) module is cyclic for every frame. In each frame, it does the following tasks.

- Waits for the transport overhead bytes (K1 and K2) ready to be processed which is indicated by the control signal *toh* =1, inserted by the extraction module.
- Filters the K1 and K2 bytes according to their specifications mentioned in [6].
- Generates two interrupt signals whenever these two APS bytes do not meet their specifications.

The Filtering abstract state machine, having two states (S0 and S1), is shown in Figure 4.7. The symbols *reset*, *toh* and *k_eq* denote active low reset, arrival of transport overhead bytes from extraction module and comparison between current and previous values of K1 and K2 bytes, respectively. According to [6, 47], the K1 and K2 bytes must be the same for three consecutive frames before a new value is accepted. The algorithm can be derived by an abstract state machine where S0 is the reset state. The abstract state will remain in state S0, if two consecutive frames do not contain identical K1 and K2 bytes. Whenever two consecutive frames contain identical K1 and K2 bytes i.e., *k_eq* = 1, a transition to state S1 will possible. It will remain in the same state, if the next frame contains identical K1 and K2 bytes, unless it will back to state S0. While in state S1, the filtered K1 and K2 bytes need to be checked. Any change to the current filtered bytes with respect to the previous value, will cause an interrupt which indicates the change of automatic protection switch bytes

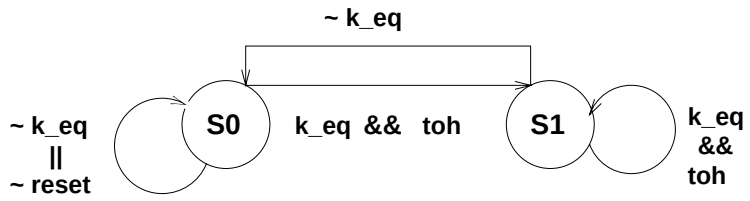


Figure 4.7: Filtering ASM for K1 and K2 bytes

The automatic protection switch failure monitoring is a complex behavior of the system. According to [6, 47], whenever any of the K1 and K2 bytes does not stabilized over a period of twelve consecutive frames, where no three successive frames contain identical K1/K2 bytes, the automatic protection switch failure alarm will be set to high. A set of abstract state machines shown in Figure 8, is used to model this failure alarm monitoring. Among two ASMs, one generates a 12-frames window and another detects the equality of K1 or K2 bytes within three consecutive frames of a 12-frames window. On each state, the frame generator will wait for the transport overhead bytes to be ready, i.e., $k = 1$. Whenever all other control inputs set to '1' (*c* and *r*), there will be a transition to next state and it will continue until reaching state S12. From state S12, it returns to its initial state S1 and waits for the transport overhead bytes to be ready. Detection of equality among K1 or K2 bytes within three successive frames can be modeled by an abstract state machine called *ASM_match* (see Figure 4.8) which has three states — S1, S2 and S3. In each state, it waits for the transport overhead bytes to be ready (indicated by *toh* = 1). State transition between two consecutive states depends on the equality of K1 or K2 bytes within two successive frames, i.e., $k\_eq = 1$. If no equality is detected, i.e., $k\_eq = 0$, there will be a transition to state S1 from any other state. To simplify the presentation in Figure 4.8, the symbols *k*, *c*

and *r* denote the arrival of overhead bytes ready to be processed, system clock and active low reset, respectively.

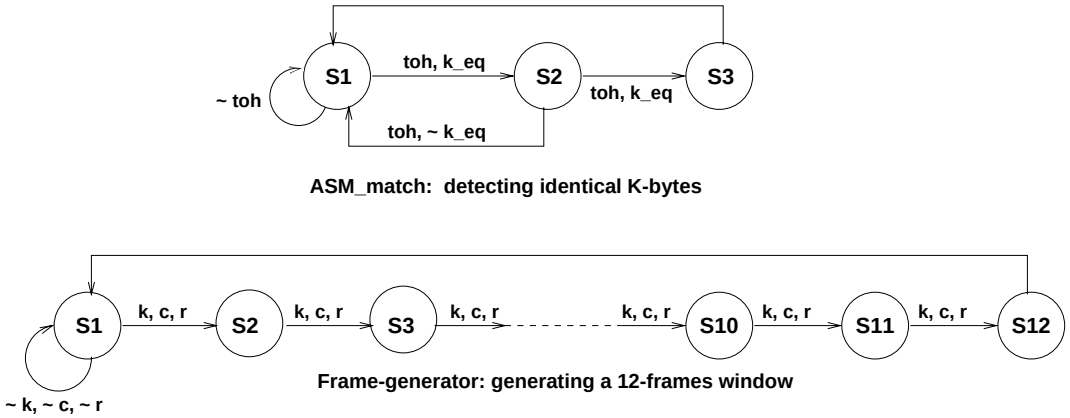
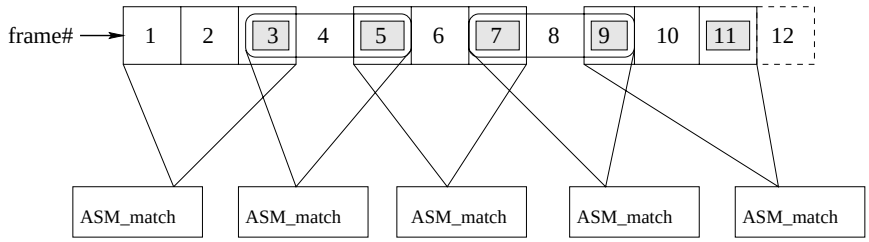


Figure 4.8: Set of ASMs to declare the APS failure alarm

The automatic protection switch failure monitoring is performed by the combination of two ASMs in Figure 4.8. A graphical representation with MDG modeling of the methodology to activate the automatic protection switch failure alarm is shown in Figure 4.9. In a 12-frames window, if we do not find any identical K1 or K2 bytes between frame#10 and frame#11, frame#12 does not take into account to activate the failure alarm. Because it is sure that we will not find two more matches within next two frames (frame#11 and frame#12). So, considering only 11 frames, the failure alarm can be activated.

Activation of APS failure alarm:



MDG-HDL model:

```
initialize frame = 0
initialize state_aps = S0
case frame = 0: next(frame) = 1
case frame = 1: next(frame) = 2
case frame = 2: next(frame) = 3
case frame = 3:if state_aps = S3
                        then alarm_1 = 0
                   else if state_aps = S0
                        then alarm_1 = 1
                   next(frame) = 4
.............
.............
case frame = 12:next(frame) = 1
```

Figure 4.9: Example to model an APS failure alarm

Figure 4. 9 shows that *ASM_matching* is looking for identical K1 or K2 bytes among 3 frames within a 12-frames window which can be divided in 5 over-lapped sub-windows. Each of the sub-window consists of 3 frames. By using the state variable *S3* of the *ASM_matching* at the intersection points of 5 sub-windows (shaded points 3, 5, 7, 9 and 11 in Figure 4.9), we can determine whether any of the 5 sub-windows containing identical bytes or not. Taking the conjunction of the results at these 5 points, we can determine whether a failure has been occurred or not. The failure alarm will be set if the result of this conjunction is '1'. An interrupt will be triggered if there is a change in the present alarm condition with respect to its previous value.

4.2.3 Synchronization Status Filtering

The behavior of this module is cylic for every frame. In each frame, it waits for the transport overhead byte (S1 byte) ready to be processed according to their specification in [6, 47]. The network elements will be synchronized if S1 bytes are identical for eight successive frames before a new value is accepted. Whenever there are no identical S1 bytes within eight consecutive frames, an interrupt needs to be triggered. Based on this specification, we can represent the behavior of this module using an abstract state machine having eight states — S0, S1, S2, S3, S4, S5, S6 and S7 (see Figure 4.10).

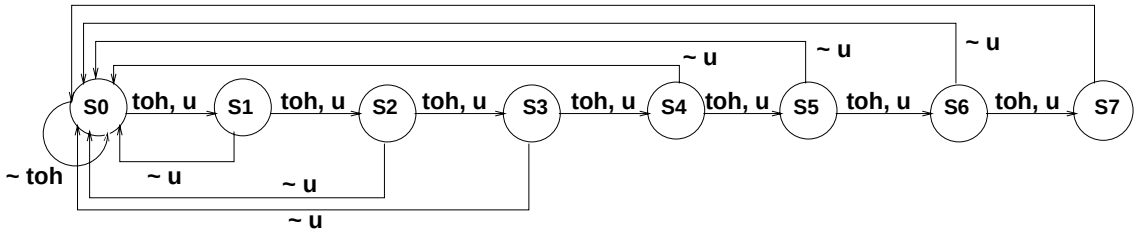


Figure 4.10: Abstract state machine to filter the S1 bytes

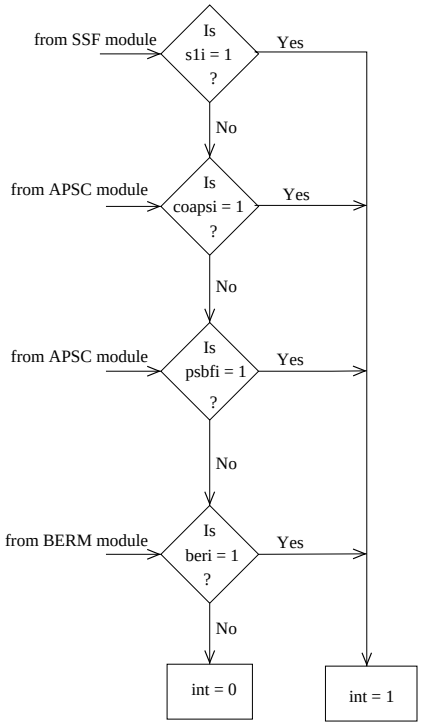
In each state, whenever a match between two consecutive S1 bytes is found, a transition to next state is possible. After the *toh* signal goes high, it performs several routine tasks, i.e., comparing present and previous values of S1 byte, updating the filtered S1 byte and generating an interrupt, if necessary. The symbols *toh* and *u* in Figure 4.10, represent the transport overhead byte ready to be processed and the result of comparison between present and previous values of S1 byte, respectively.

4.2.4 The Interrupt Server

The interrupt server has a very simple characteristic. Whenever a change in the automatic protection switch, a protection switch failure, a change of the synchronization sta-

tus, or a change of the BERM status alarm is detected on its event capturing input, the interrupt line *int* goes high, otherwise it remains low forever. The behavior of an interrupt server can be described using a flowchart and its model in pseudo-MDG-HDL as shown in Figure 4.11. We can see from Figure 4.11 that the interrupt line of the TSB will go high, i.e., *int=1*, whenever any of the sub-modules' (e.g., APSC, SSF and BERM*)* interrupt line sets to high. The interrupt line will be low, if all the interrupt lines from sub-modules are low.

Flowchart:



MDG-HDL model:

```
initialize int = 0
if s1i = 1
   then next(int) = 1
   else if coapsi = 1
       then next(int) = 1
       else if psbfi = 1
          then next(int) = 1
          else if beri = 1
             then next(int) = 1
             else next(int) = 0
```

Figure 4.11: Flowchart specification of interrupt server and its MDG model

4.2.5 Bit Error Rate Monitoring (BERM)

The Bit Error Rate monitoring is performed by a sliding window algorithm [47]. The evaluation period for the sliding window has a variable length which can be chosen by the users. This evaluation period is broken into eight sub-accumulation periods. Thus, the BERM status is evaluated many times per evaluation period and not only once. This gives a better detection time as well as a better false detection immunity. The sliding window algorithm is selected to keep track of the history of the BIP count. In order to add the new BIPs at one end of the window and subtract them at the other end, the queue of the BIPs count needs to be stored in a history queue register. This algorithm is chosen for its superior performance when compared to other algorithms. It offers a lower false declaration probability at the expense of a more complex behavior. The window is progressing by hops that are much smaller than the window size. It is broken into eight sub-intervals which is forming a queue of BIP's history (see Figure 4.12).

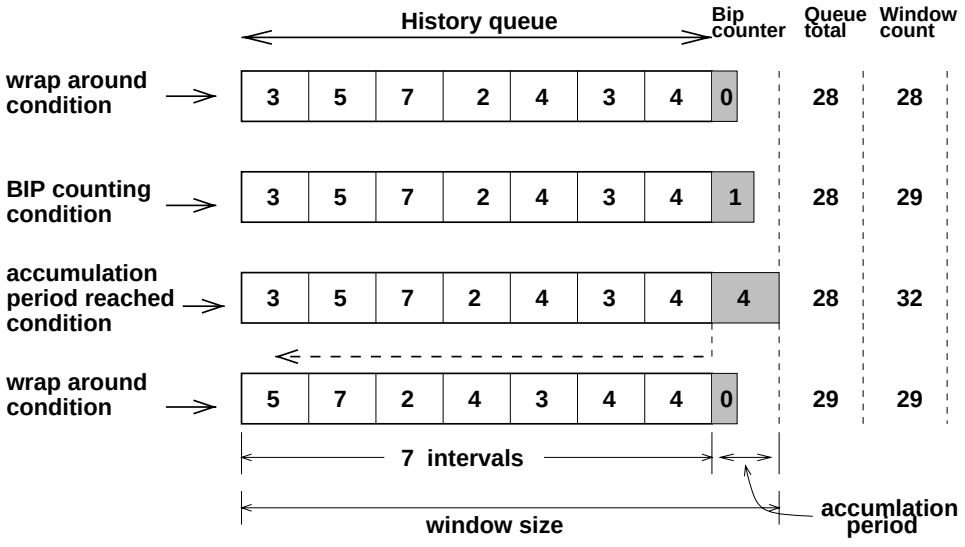


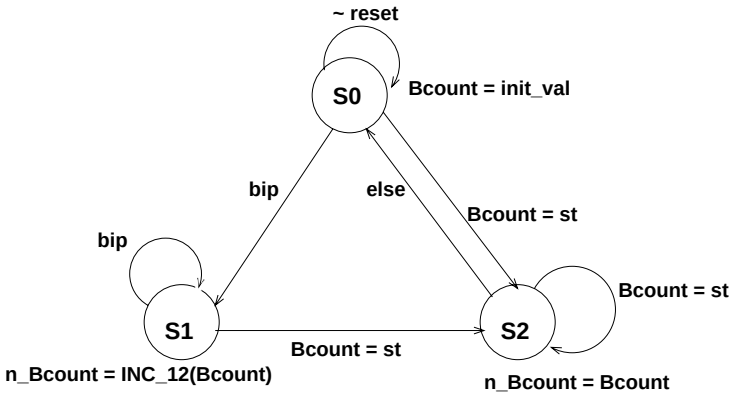
Figure 4.12: BERM sliding window algorithm

After initialization, the Bit Error Rate Monitoring can be done by following sequence of steps:

- Counting of frames for a given sub-accumulation period.
- Accumulates BIP errors over a declaration period of time which is the number of frames.
- Compare the accumulated count of line BIP against a programmable declaration threshold value which indicates the BER to be monitored.
- When the accumulated count of line BIP exceeded the threshold value, the BERM declaration status alarm goes high.
- Then, the BERM starts to monitor the clearing threshold. If the BER goes under a clearing threshold, the BERM status alarm goes low.

An interrupt is triggered, whenever there is a change of the current BERM status from its previous value. The BIPs accumulation is done in the eighth sub-interval of the sliding window (see Figure 4.12). When the frame counter is reached to a certain threshold value, the latest BIP count will be put in the history queue. The summation of eight sub-interval BIPs stored in the history queue is periodically compared against a declaration or clearing threshold value to set or reset the BERM status alarm. Whenever the BERM status alarm condition is set, it will be compared against the clearing threshold value. On the other hand, if the alarm is in the reset condition, the declaration threshold value is used for monitoring and comparing. To keep our description simple, we are presenting only the BIP counting abstract state machine and its pseudo-MDG-HDL model in Figure 4.13.

ASM of the BIP line counter:



MDG-HDL model:

```
initialize Bcount = C
initialize state = S0
case state = S0: if bip = 1
                    then next(state) = S1
                         Bcount = INC_12(Bcount
                    else if Bcount = st
                         then next(state) = S2
                              Bcount = Bcount
case state = S1: if bip = 1
                    then next(state) = S1
                         Bcount = INC_12(Bcount
                    else if Bcount = st
                         then next(state) = S2
                              Bcount = Bcount
case state = S2: if Bcount = st
                     then next(state) = S2
                          Bcount = Bcount
                     else next(state) = S0
```

Figure 4.13: An ASM to count the BIP line and its MDG-HDL model

The BIP line counter has three possible states — S0, S1 and S2. The state variable *Bcount* stores the count value of BIP line. The symbols *st* and *bip* are the inputs to the state machine. They represent the saturation threshold value of the counter and the received BIP line, respectively. In state S0, the counter has been initialized to *zero* which is a generic

constant of *abstract* sort. After initialization, if the input *bip* = '1' then the next state will be S1. In state S1, using the uninterpreted function symbol *inc_12* of type [$worda12 \rightarrow worda12$] the count value *Bcount* will be incremented by one. When the count value is equal to the saturation threshold value *st,* there will be a transition to state S2. In state S2, the value of the counter will remain unchanged until *Bcount* is not equal to *st.*

An abstract state machine can have an infinite number of states due to the abstract variable and the uninterpreted nature of the function symbols. The reachability analysis algorithm of MDGs is based on the abstract implicit state enumeration. The major draw back of this algorithm is that a least fixed point may not be reached during reachability analysis. Because of this limitation, a *non-termination* of abstract state enumeration may occur when computing the set of reachable states. To illustrate this limitation of MDG-based verification, we can have an example of Figure 4.13, where state variable *Bcount* of abstract sort represents the BIP counter of a SONET frame, a generic constant *zero* of the same abstract sort denotes the initial value of *Bcount,* and an abstract function symbol *INC* describes how the counters are incremented by one. The MDG representing the set of reachable states of the BIP counting ASM (see Figure 4.13) would contain states of the form

(*Bcount*, *INC*(. . . *INC*(*zero*). . .))

for the number of infinite iterations. As a consequence, there is no finite MDG representation of the set of reachable states and the reachability algorithm will not terminate. This typical form of non-termination is due to the fact that the structure of MDG can be arbitrarily large, and it can be avoided by using some techniques described in [1, 58]. In those papers, the authors present one of the methods based on the *generalization* of initial state that causes divergence, like the variable *Bcount* in Figure 4.13. Hence, rather

than starting the reachability analysis with an abstract constant *zero* as the value of *Bcount*, a *fresh* abstract variable (e.g., *C*) is assigned to *Bcount* at the beginning of the analysis.

4.3 Modeling of the RTL Implementation of the RASE TSB

In this section, we give a brief description of the RASE TSB at the RT level. We translated the original VHDL models into very similar models using the Prolog-style MDG-HDL, which comes with a large number of predefined basic components (logic gates, multiplexers, registers etc.) [59]. To handle the complexity of the design which consists of a network of 11400 equivalent gates, we adopted the abstraction techniques described in Chapter 3.

For example, we can take the BERM module which is the largest component of the RASE TSB to illustrate the data abstraction technique (see Figure 4.14). This module contains registers with variable widths which can be 12-bits, 17-bits, 24-bits, or 7x12-bits wide. As the MDG system can handle abstract data sorts, it avoids all the cumbersome procedure of defining each bit of a register. Rather, a register can be viewed as an *abstract* variable of *n*-bit word i.e., *wordn.* Such high-level words are arbitrary size, i.e., generic with respect to the word sizes. We can define each of the datapaths of this module, i.e., 12-bits, 17-bits and 24-bits, as *worda12, worda17* and *worda24* of *abstract* sort. An immediate consequence of modeling the data as a compact word of abstract sort is that we can simplify the modeling of the BERM block by using generic registers of arbitrary size and abstract the functionality of the Declare BIP Adder unit (Figure 4.14) using an *uninterpreted* function symbol **add_17** of type [$worda17 \rightarrow worda17$]. Likewise, we can increment the

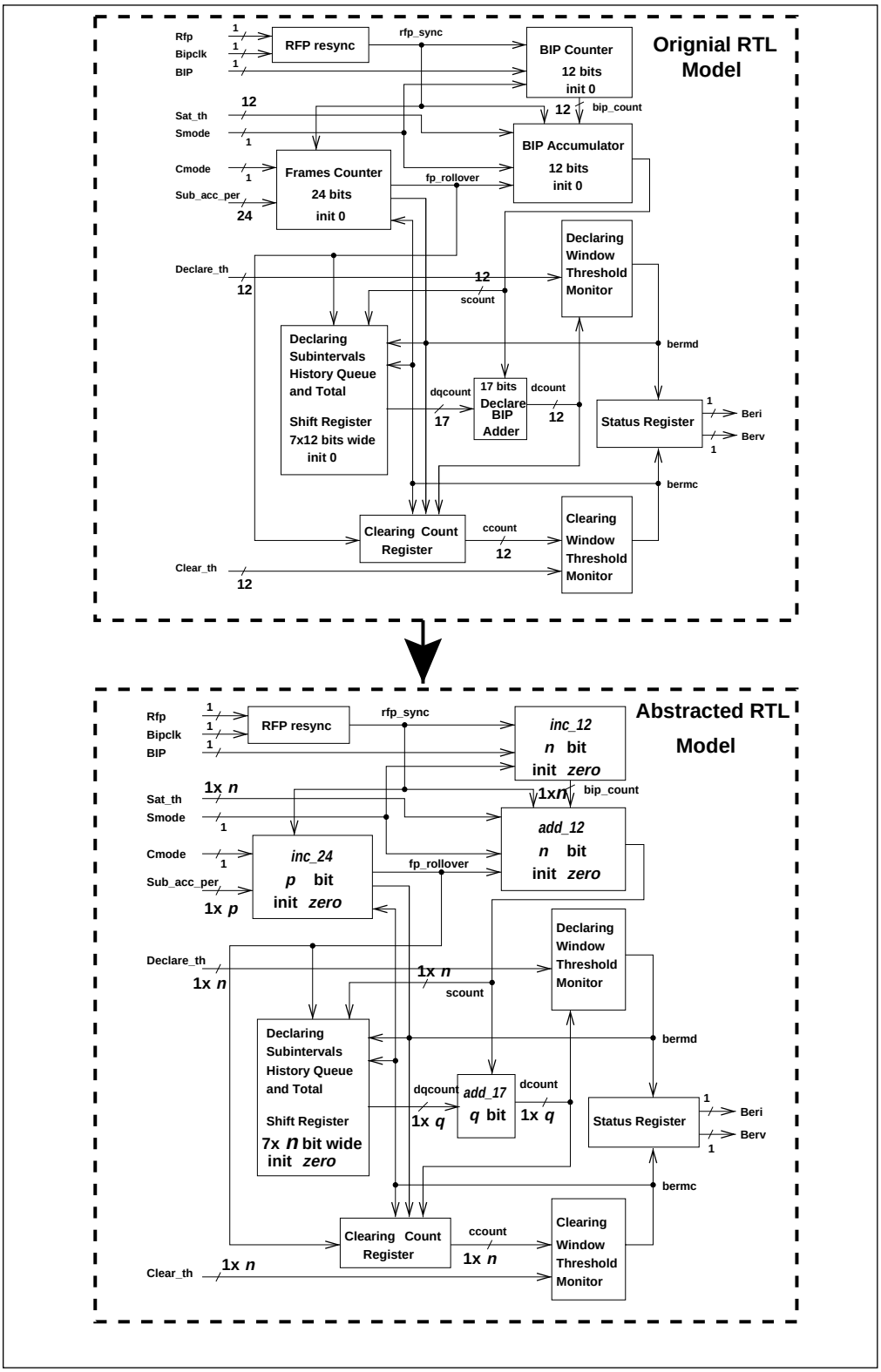


Figure 4.14: Module abstraction of the BERM block

value of an abstract variable using an uninterpreted function symbol **inc** of type $[wordn \rightarrow wordn]$ which in turn reduces the probability of state space explosion. We abstracted the functionality of the frame and BIP counter modules (Figure 4.14) using two uninterpreted function symbols **inc_12** of type $[worda12 \rightarrow worda12]$ and **inc_24** of type $[worda24 \rightarrow worda24]$, respectively. The internal control signal *fp_rollover* is generated by the frame counter module which is an abstracted module, i.e., all the data used by this module are *abstract* sorts. To generate a control signal which is of concrete sort, we need some sort of decoders that accept abstract data and give an output of concrete sort. MDG-HDL provides this type of decoding technique by using cross-operators. A cross-operator is an uninterpreted function of type $([wordn] \rightarrow bool)$ or $([wordn, wordn] \rightarrow bool)$ which may take one or more abstract variables and gives an output of concrete sort. Here *bool* is a concrete sort with enumeration {0, 1} and is used by the control signal *fp_rollover*. Using a cross-operator of type $([worda24, worda24] \rightarrow bool)$, the control signal *fp_rollover* can be generated by the abstract frame counter. In all of these cases, data operations are viewed as black-box operations.

In the original design, a history queue register of 7x12 bits wide is used to store the accumulated values of BIPs for seven clock cycles. In each cycle, the content of each register within the history queue register is updated from its previous stage, e.g., stage-1 will be updated from stage-0. The content of the register in stage-6 is used to calculate the BER using sliding window algorithm described in Section 4.5. As the current version of MDG-HDL does not support any declaration of multidimensional arrays, we need to adopt a technique to cope with this limitation.

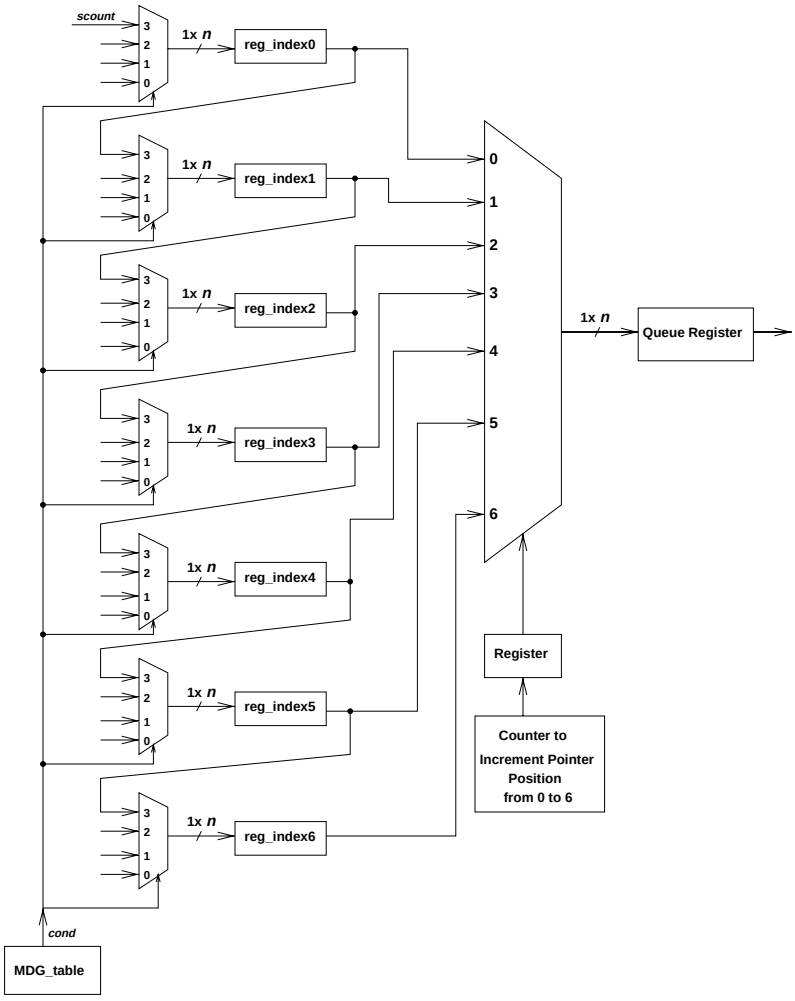


Figure 4.15: The Implementation of a multi-dimensional array using MDG-HDL

We describe our technique as follows (see Figure 4.15): The history queue register is segmented into seven different registers, (e.g., reg_index0, reg_index1, etc.). Depending on the control signals, each segmented register can have four possible values which can be implemented by seven 4x1 multiplexors along with one MDG table. The input selection is controlled by the signal *cond* which is the output of the MDG-table, containing all the required conditions that need to be satisfied. In each cycle, each segmented register will be

updated by its previous stage provided that multiplexors' third inputs are selected by the MDG-table. In the next stage, all the indexed registers are connected to a 7x1 multiplexer. The inputs of the multiplexer are controlled through a counter based pointer. The pointer is incremented in each cycle by one. The input registers are selected by the position of the pointer which is the current value of the counter. Whenever an indexed register is selected, the history queue register is connected to that indexed register, e.g., at the 7th cycle when the counter value is 6, register *reg_index6* will be connected to history queue register and thus can be used to calculate the BER.

4.4 Hierarchical Verification of the RASE TSB

Based on the hierarchy of the design, we adopted a hierarchical proof methodology for the verification of the proposed design as described in Chapter 3. To illustrate our hierarchical proof methodology, we can have a system having three sub-modules, named B_1, B_2 and B_3, which may or may not be interconnected between them by control signals. In the verification phases, first we proved that the implementation of each sub-module (i.e., $B_{j\,[impl]}$, where j = 1,.., 3) is equivalent to its specification, i.e., $B_{j\,[spec]} \Leftrightarrow B_{j\,[impl]}$ which can be done automatically within the MDG system. Then we derive a specification for the whole system as a conjunction of the specification of each sub-module, i.e., $S_{[spec]} = \bigwedge_{j:1..n_s} B_{j[spec]}$. Similarly, we also derive an implementation of the whole system as a conjunction of the implementation of each sub-module, i.e., $S_{[impl]} = \bigwedge_{j:1..n_s} B_{j[impl]}$. The current version of the MDG system does not support an automatic conjunction procedure of sub-modules. To cope with this limitation, we need manual interventions to compose all of the sub-modules (both specification and implementation) until the top level of the system

is reached. Finally, we deduce that the specification of the whole system is equivalent to the top level implementation of the system, i.e., $S_{[spec]} \Leftrightarrow S_{[impl]}$. We must ensure that the specification itself is correct with respect to its desired behavior given by a set of properties. A graphical representation of this method is illustrated in Figure 4.16.

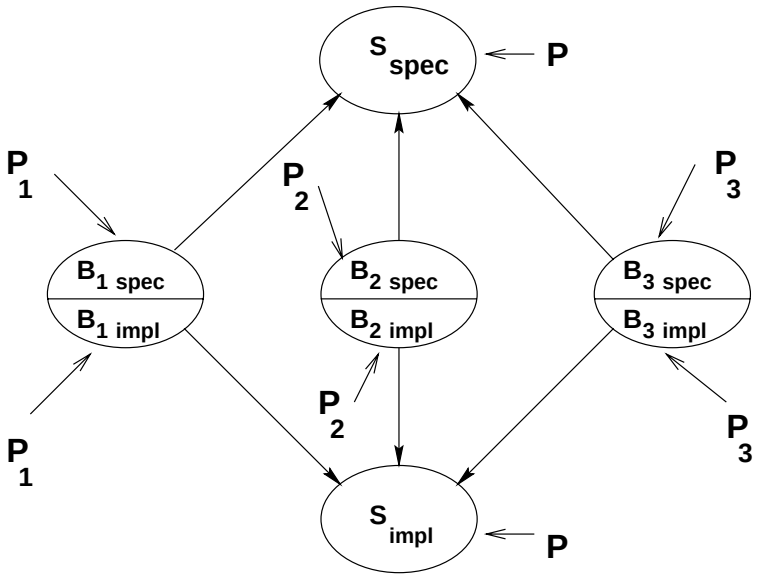


Figure 4.16: Hierarchical proof methodology

The RASE TSB has five modules, each module in the design was verified separately using both property and equivalence checking facilities provided by the MDG tools. At first, we applied property checking on the block level of the TSB. While applying property checking on the block level, we sorted the properties according to the features that are generated by the specific block. To illustrate this idea, we can have an example in Figure 4.16, where properties P_1, P_2 and P_3 are the features for the block B_1, B_2, and B_3 respectively. To perform a hierarchical verification, first we will verify all of these properties on their specific blocks. Finally, we will merge all of these properties into a set

of properties to be checked on the top level of the design, i.e., S_{spec} and $S_{impl.}$ In the following two sub-sections, we describe the verification process for the Telecom System Block using equivalence checking and model checking techniques.

4.4.1 Equivalence Checking

We follow an hierarchical approach for the equivalence checking of the RASE TSB. We verified that the RTL implementation of each module complied with the specification of its behavioral model. Thanks to the data abstraction features in MDG, we also succeeded to verify the top level specification of the RASE TSB against its full RTL implementation. To verify the RTL implementation against the behavioral specification, we made use of the fact that the corresponding input/output signals used in both descriptions have the same sort and use the same function symbols. The two machines are equivalent if and only if they produce the same outputs for all input sequences. Experimental results, including CPU time, memory usages and number of MDG nodes generated, for the equivalence checking between the behavioral model of each module including top level specification of the TSB against their RTL implementation are given in Table 4.1.

The verifications of the first four modules consumed less CPU time and memory, because they have less complexity and abstract state variables and cross-operators than those of the last three modules (see Table 4.1). The BERM module consumed more CPU time and memory during the verification as it performs complex arithmetic operations on abstract data. On the other hand, the verification of the TOH Process module consumed less CPU time and memory, even though it needs more MDG components to model than the BERM module. This is because of the fact that, *TOH Process* module is a state machine based design and in contrast to BERM does not perform any complex data opera-

tion. To model the complex arithmetic operations of the BERM, we need more abstract state variables and uninterpreted functions, specially cross-operators, which have significant effects on the verification of this module. As the top level of the design comprises all the bottom level sub-modules, it takes more CPU time and memory during the verification process than the other modules.

Table 4.1: Experimental results for equivalence checking

Module name	CPU time (in second)	Memory (in MB)	Number of MDG nodes generated
TOH Extraction	3.88	2.33	2806
APSC module	17.37	7.91	9974
Synchronization Status	22.22	6.81	14831
Interrupt server	0.48	0.09	180
BERM module	80.53	21.31	35799
TOH Process module	89.03	27.79	60068
Top level of RASE TSB	437.15	47.36	135658

4.4.2 Validation by Property Checking

We applied property checking to ascertain that both the specification and the implementation of the telecom system block satisfy some specific characteristics of the system. The properties are statements regarding the expected behavior of the design, and significant effort is spent in their development. The properties should be true for the design at any level regardless of the verification technique. It can include details of certain situations which should never occur and others which will happen eventually. The former are called *safety properties* and the later *liveness properties* [17]. We describe several properties and their verification in the following two sub-sections.

4.4.2.1 Properties Description

In order to describe the properties of the system, we need to define a proper environment of the system. As we explained before, we are interested in five control blocks only. During our modeling of the RASE TSB, we eliminated those blocks which have no effect on the functionality of the telecom system block. The environment is built in such a way that it allows a non-deterministic choice of values on the primary inputs. After establishing a proper environment, we consider twelve properties of the RASE TSB, including safety and liveness properties. While using MDG for property checking, the properties are described using a property specification language called L_{MDG}, [55, 56] which is a subset of Abstract-CTL that supports abstract data representations [54]. Both safety and liveness properties can be expressed in L_{MDG}, however, only universal path quantification is possible. In the following Abstact-CTL expressions, the symbols "!", "&", "|", "->", "=>" denote logical "not", "and", "or", "imply" for safety properties, and "imply" for liveness properties, respectively. In the properties descriptions, **AG**, **X** and **F** mean that always for all paths, in the next cycles and sometimes in the future, respectively. In the following, we present two safety and one liveness properties of the TSB and the rest of the properties are given in Appendix A.

Table 4.2: Properties and their corresponding modules of RASE TSB

Property	Module
Property 1	Synchronization Status Filtering (SSF)
Property 2	Automatic Protection Switch Control (APSC)
Property 3	Automatic Protection Switch Control (APSC)
Property 4	Automatic Protection Switch Control (APSC)
Property 5	Transport Overhead Processor
Property 6	Transport Overhead Byte Extractor
Property 7	Bit Error Rate Monitoring (BERM)
Property 8	Bit Error Rate Monitoring (BERM)
Property 9	Transport Overhead Processor, BERM and Interrupt Server
Property 10	Transport Overhead Processor and BERM
Property 11	Automatic Protection Switch Control (APSC)
Property 12	Bit Error Rate Monitoring (BERM)

Property 1: According to the specification of SONET Transport System in [6]: The filtered S1 byte of a SONET frame needs to be identical for eight consecutive frames. If eight consecutive frames do not contain identical S1 bytes an interrupt is generated to indicate that the filtered S1 value has changed. When the TSB is in *state_ssd = 6*, it means that no 7 consecutive frames contain identical S1 bytes. If the next frame does not have identical byte, interrupt *s1i* will go to high in the next cycle. In L_{MDG} this safety property is expressed as follows:

```
AG((!(rstb=0)&(rclk=1)&(toh_ready=1)&(((s1_cap=1)&(state_ssd=6)&
   (s1_in=s1_last_reg)&(!(s1_in=s1_filter_reg))))) -> (X(s1i=1)) );
```

Property 2: According to the specification of the SONET Transport System in [6]: The APS bytes, i.e., K1 and K2 bytes, should be identical for 3 consecutive frames. If there is a change in these APS bytes within 3 consecutive frames, an interrupt will be generated to indicate that a change in APS bytes has occurred. When the TSB in *state_aps = 1* and the current values of APS bytes are not identical with their previous filtered values, the interrupt will go to high to indicate a change in APS bytes. In L_{MDG} this safety property is expressed as follows:

```
AG((!(rstb=0)&(rclk=1)&(toh_ready=1)&(state_aps=1)&(!(k1_fil_reg =
    k1_in)|!(k2_fil_reg=k2_in)) )-> (X(coapsi=1)));
```

Property 12: When the value of BERM declaration threshold alarm is stable, we need to make sure that the interrupt lines related to this value eventually goes low. In L_{MDG} this liveness property is expressed as follows.

```
AG(((berv=berv_last_reg)&(!(rstb=0))&(bipclk=1))=>(F(beri=0)) );
```

4.4.2.2 Properties Verification

The verification of the properties has been carried out using the model checking facility of MDG tools [59]. We checked in each reachable state if the outputs satisfy the logic expression of the property which should be true over all reachable states. The experimental results from the verification of all properties stated in Section 4.4.2.1 for both specification and implementation, are given in Table 4.3 and Table 4.4, respectively. All experimental results were obtained on a Sun Ultra SPARC 2 workstation (296 MHz / 768 MB) and include CPU time in seconds, memory usage in megabytes and the number of MDG nodes generated.

Table 4.3: Experimental results of property checking on the specification

Property	Module	CPU time (in sec.)	Memory (in MB)	No. of MDG nodes
Property 1	SSF	59.91	13.22	21690
Property 2	APSC	71.43	13.70	21333
Property 3	APSC	55.39	12.46	20984
Property 4	APSC	56.67	12.01	21060
Property 5	TOH Proc.	45.59	13.19	51527
Property 6	TOH B. Ex.	53.59	14.50	20837
Property 7	BERM	52.76	12.92	21243
Property 8	BERM	44.83	14.34	21060
Property 9	RASE	56.28	12.37	21036
Property 10	RASE	54.06	13.29	51253
Property 11	APSC	54.99	12.59	1214
Property 12	BERM	87.66	12.46	21178

As we discussed in Section 4.2.1, when a design is dependent on a particular interpretation of the function symbols which are uninterpreted in the model, a non-termination of reachability can occur. In our case, we used several uninterpreted functions and abstract variables in the abstracted model of the RASE TSB which created a non-termination problem during the reachability analysis. To cope with the non-termination problem of abstract state exploration, we used initial state generalization technique described in Section 4.2.1. In the case of uninterpreted functions, the non-termination problem has been resolved by

providing a partial interpretation through rewrite rules. For more details about non-termination and rewrite rules readers are referred to [1, 58].

Table 4.4: Experimental results of property checking on the implementation

Property	Module	CPU time (in sec.)	Memory (in MB)	No. of MDG nodes
Property 1	SSF	82.47	15.60	53139
Property 2	APSC	82.62	14.98	52375
Property 3	APSC	54.31	17.12	51832
Property 4	APSC	78.05	15.24	51354
Property 5	TOH Proc.	76.57	15.53	51533
Property 6	TOH B. Ex.	81.65	15.80	52094
Property 7	BERM	82.54	15.86	51482
Property 8	BERM	64.30	15.72	51410
Property 9	RASE	78.06	16.65	51330
Property 10	RASE	58.41	16.12	51277
Property 11	APSC	81.42	16.22	51530
Property 12	BERM	85.72	15.98	51564

4.4.3 Comparison between Cadence FormalCheck and MDG Model checker

One of the motivations of this work was to compare the model checking of the *RASE TSB* using MDG model checker with an existing commercial model checking tools. We chose Cadence FormalCheck as a commercial one to compare with MDG. The performance criteria of the caparison were *CPU-time, memory usages and state variables*.

FormalCheck is a model checking tool developed and distributed by Cadence Design Systems, Inc. [14]. The tool accepts VHDL and Verilog HDL as its input language

provided that RTL design should be modeled in synthesizable VHDL or Verilog HDL code. FormalCheck has an intuitive graphical interface which makes it users friendly. This model checker verifies that a design model exhibits specific properties that are required by the design to meet its specifications. Properties that form the basis of a model are termed as Queries in FormalCheck. FormalCheck supports constraints on the design to be verified to limit the input scenarios which in turn reduce the state space of the design model that is to be verified [7].

As explained in Section 3, we are only interested in five modules of the RASE TSB. Before checking the properties, we need to setup an environment of the system which will reduce the state space and speed up the verification process. To do so, we eliminated the CBI block and two input and output multiplexors from the original VHDL code. The signals related to these modules are used as primary inputs and outputs of the system that allows a non-deterministic choice of values on the inputs. During our verification in FormalCheck, we used the same verification methodology as with the MDG system. Starting from the lower level modules, we reached the top level structural model which includes the whole design of the RASE TSB. We defined all the properties stated in Section 4.4.2.1 using FormalCheck property language. A full description of these properties is included in Appendix B. In all properties, we used reset signal *rstb* as the default constraint where *rstb* is used to initialize the registers. As the RASE TSB uses asynchronous active low reset, the reset input *rstb* starts with low for duration of 2, and then goes to high forever. Depending on the functionality, the modules of the TSB are running under the control of

each of the two different clocks — *rclk* and *bipclk.* During our verification in Formal-Check, we constrained these clocks depending on the properties.

Table 4.5: Property checking on the top level implementation using FormalCheck and MDG

Property	MDG model checker			FormalCheck model checker		
	Time (in Sec.)	Memory (in MB)	State variable	Time (in Sec.)	Memory (in MB)	State variable
Property 1	82.47	15.60	57	60	16.08	54
Property 2	82.62	14.98	57	32	12.81	71
Property 3	54.31	17.12	57	44	14.45	43
Property 4	78.05	15.24	57	44	14.49	44
Property 5	76.57	15.53	56	*	*	*
Property 6	81.65	15.80	55	10	11.75	28
Property 7	82.54	15.86	57	*	*	*
Property 8	64.30	15.72	57	*	*	*
Property 9	78.06	16.65	55	*	*	*
Property 10	58.41	16.12	55	*	*	*
Property 11	81.42	16.22	56	9	2.66	42
Property 12	85.72	15.98	56	*	*	*

The summary of the comparison between these two verification systems with respect to CPU time, memory usages and number of state variables are given in Table 4.5, where '*' means that the verification did not terminate within a substantial verification time. All of the experiments have been carried out on a Sun Ultra SPARC 2 workstation with 296 MHz and 768 MB of memory. While performing the property checking on the top level model of the RASE TSB using FormalCheck, some of the properties verifications (Properties 5, 7, 8, 9, 10 and 12) did not terminate. Although those properties were verified with a reasonable CPU time on a modular basis. These properties were taking too much CPU time

and memory, even though we used different tool guided reduction and abstraction techniques in FormalCheck.

Properties 7, 8 and 12 belong to the BERM module which is the largest and most complex module of the *RASE TSB*. The BERM module has several control state variables which perform complex arithmetic operations between large sized data. The verification of Properties 7, 8 and 12 did not terminate within a substantial CPU time as these three properties are dealing with control signals having width of 24 bits to 12 bits. Moreover, some complex data operations between large sized state variables were involved. In general, if the control information needs *n* bits, then it is impossible to reduce the datapath width to less than *n*. Hence, in this case ROBDD-based datapath reduction technique is no more feasible. On the other hand, using the MDG-based approach, we naturally allow the abstract representation of data while the control information is extracted from the datapath using cross-operators. Because of this, all of these properties were verified within the MDG system without any complexity.

Properties 5, 9 and 10 did not terminate on the top level structural model as these properties are verifying the integrated functionalities of several modules. As the control circuitry naturally modeled as FSM, automata-oriented methods are more efficient in handling FSM-based designs than designs with complex arithmetic data operations. Our experimental result shows that FormalCheck whose underlaying structure is automata oriented [33] is more efficient in verifying FSM-based design, i.e., concrete data, than the MDG tools. Table 4.5 shows that, Property 1, 2, 3, 4, 6 and 11 takes less verification time in FormalCheck than in the MDG tools. These properties are related to the APSC,

Synchronization status and TOH Extraction modules which are completely FSM-based designs (see Table 4.2).

Human effort to formal verification of any design is an important issue to the industrial community. In FormalCheck, we do not need manual intervention for variable ordering while the MDG tools need manual variable ordering since no heuristic ordering algorithm is available in the current version. During the verification of the *RASE TSB*, much of the human time was spent on determining a suitable variable ordering in MDG. Because the verifier needs to understand the design thoroughly, the time spent on understanding and modeling the behavior of the design in MDG-HDL was about three man-months. The translation of the original VHDL design description to a similar MDG-HDL structural model took about one man-month. In contrast to this, no time was spent on the RTL modeling for FormalCheck since it accepts the original VHDL structural model as its input language. Time spent on checking the equivalence of the RTL implementation with its behavioral specification using MDG, was about one man-week. In the property checking, the time required to setup twelve properties, to build the proper environment and to conduct the property checking both on the implementation and the specification was about three man-weeks. On the other hand for FormalCheck, property checking on the implementation took about two man-weeks.

Chapter 5

Conclusions and Future Work

BDD-based symbolic model checking and equivalence checking have proven to be successful formal verification techniques that can be applied to real industrial design. However, since it requires the design to be described at the boolean level, they often fail to verify a large-scale design because of the *state space explosion* problem caused by the large datapath.

This thesis investigates the formal verification using Multiway Decision Graphs (MDGs) of a Telecom System Block from PMC-Sierra Inc. We studied the effectiveness of Multiway Decision Graphs (MDGs) tools in verifying a large-scale industrial design. The design we considered is a TSB named Receive Automatic protection switch control, Synchronization status extraction, and Bit Error Rate Monitor (RASE). The design contains 11,400 equivalent gates which is much larger than any other design verified by MDGs before this work.

The specific contributions of this work are as follows:

1. We suggested a hierarchical approach for organizing the verification of a large-scale industrial design using MDGs. Our hierarchical approach simplifies a large modeling and verification problem into smaller pieces that can be handled on a modular basis. The hierarchical approach is applicable on a partially defined design instead of waiting for the entire design model.

2. Based on the product document provided by PMC-Sierra Inc., we derived a behavioral

model in MDG-HDL of the TSB. The specification was given as English text which was modeled in terms of Abstract State Machines using MDG-HDL. Our behavioral modeling was based on the different levels of design hierarchy.

3. As the complexity of data operations increases, the default setting used by most formal verification tools may not be sufficient to avoid state space explosion. We applied data abstraction to handle the complexity of state space in datapath orientated module. We developed a generic model of datapath orientated modules using abstract data sorts. We succeeded to verify the whole TSB using the MDG tools. The verification process had been carried out by equivalence checking as well as model checking. The validity of our hierarchical approach for organizing the verification of a real industrial design was demonstrated by the experimental results obtained with MDG tools.

4. We compared the model checking of the RASE TSB using MDG model checker with an existing commercial model checking tool, here, Cadence FormalCheck. While performing the property checking on the top level model of the design using FormalCheck, the verification of some of the datapath oriented properties did not terminate. As the MDG-based approach allows the abstract representation of data while the control information is extracted from the datapath using cross-operators, all of these properties could be verified in MDG. Our experimental result shows that FormalCheck is more efficient in verifying FSM-based design, i.e., concrete data, than the MDG tools.

The experimental results in this thesis suggest that a hybrid MDG-FormalCheck model checking approach can be applied to improve the efficiency of formal verification in an industrial setting. This hybrid approach can be widely applicable in verifying a class of

designs where the control portion is composed of FSM-based and datapath orientated modules. Because our experimental results showed that FormalCheck is more efficient in verifying FSM-based module while MDG-based model checking is less efficient in verifying designs with concrete data. On the other hand, MDG-based model checking showed their efficiency in verifying design with abstract datapath.

MDGs open the way to the development of a wide range of new formal verification techniques. The goal of formal verification is to improve the industrial design verification process. To achieve this objective, we need to verify a variety of industrial designs to evaluate and improve the performance of the MDG-based verification techniques. For instance, the present work could be extended to investigate the compositional verification of the RASE TSB with other system blocks.

Bibliography

[1] O. Ait-Mohamed, X. Song, E. Cerny, "On the nontermination of MDG-based abstract state enumeration", In *Proc. IFIP Conference on Correct Hardware and Verification Methods*, Montreal, Canada, October 1997, pp. 218-235.

[2] The ATM Forum Technical Committee: UTOPIA Level 2; Vol. 1, June 1995.

[3] A. Aziz et al. , "HSIS: A BDD-based Environment for Formal Verification", In *Proc. ACM/IEEE Design Automation Conference*, NewYork, June 1994, pp.454-459.

[4] S. Balakrishnan and S. Tahar, "A Hierarchical Approach to the Formal Verification of Embedded Systems Using MDGs", In *Proc. IEEE 9th Great Lakes Symposium on VLSI*, Ann Arbor, Michigan, USA, March 1999, IEEE Computer Society Press, pp. 284-287.

[5] L. Barakatain, S. Tahar, Jean-Marc Gendreau and Jean Lamarche, "Practical Approaches to the Model Checking of a Telecom Megacell using FormalCheck", In *Proc. ACM 11th Great Lakes Symposium on VLSI*, West Lafayette, Indiana, USA, March 2001.

[6] Bell Communication Research (BellCORE), "*SONET Transport Systems: Common Generic Criteria*", GR-253-CORE, issue 2, December 1995.

[7] Bell Labs Design Automation, Lucent Technologies, *FormalCheck Users Guide*, Vol. 2.1, July 1998.

[8] R. S. Boyer and J. S. Moore, "*A Computational Logic Handbook*", Academic Press, Boston, 1988.

[9] R. K. Brayton et. al, “VIS: A System for Verification and Synthesis”, Technical Report UCB/ERL M95, Electronics Research Laboratory, University of California, Berkely, December 1995.

[10] M. C. Browne, E. M. Clarke, D. L. Dill and B. Mishra, “Automatic Verifcation of Sequential Circuits using Temporal Logic”, *IEEE Transactions on Computers*, December 1986, pp. 1035-1044.

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

[12] R. E. Bryant and Y. Chen, “Verification of Arithmetic Circuits with Binary Moment Diagrams, In *32nd ACM/IEEE Design Automation Conference*, San Francisco, California, June 1995.

[13] J.Burch, E. M. Clarke, K. L. McMillan and D. L. Dill, “Sequential Circuit Verification using Symbolic Model Checking”, In *Proc. 27th ACM/IEEE Design Automation Conference, IEEE Computer Society Press*, Los Alamitos, June 1990, pp. 46-51.

[14] Cadence Design Systems, Inc., “*Formal Verification Using Affirma FormalCheck Manual”*, Version 2.3, August 1999.

[15] E. Cerny, F. Corella, M. Langevin, X. Song, S. Tahar, Z. Zhou. “Automated Verification with Abstract State Machines Using Multiway Decision Graphs”, *Formal Hardware Verification Methods and Systems in Comparison*, LNCS 1287, State-of-the-Art Survey, Springer Verlag, 1997, pp. 79-113.

[16] B. Chen, M. Yamazaki, and M. Fujita, “Bug Identification of a Real Chip Design by Symbolic Model Checking”, In *Proc. International Conference on Circuits And Systems*, London, U.K., June 1994, pp. 132-136.

[17] E. M. Clarke, O. Grumberg and D. E. Long, "Model checking and Abstraction", In *Proc. 19th ACM Symp. on Principles of Programming Languages*, January 1992.

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

[19] E. M. Clarke, M. Fujita and X. Zhao, "Hybrid Decision Diagrams", In *Proc. IEEE International Conference on Computer-Aided Design*, San Jose, California, U. S. A, November 1995.

[20] 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, February 1997, pp. 7-46.

[21] F. Corella, M. Langevin, E. Cerny, Z. Zhou and X. Song, , "State Enumeration with Abstract Descriptions of State Machines", In *Proc. IFIP WG 10.5 Advanced Research Working Conf. on Correct Hardware and Verification Methods,* Frankfurt, Germany, October 1995.

[22] O. Coudert, C. Berthet and J.C. Madre, "Verification of Synchronous Sequential machines based on symbolic execution", In J. Sifakis, editor, Automatic Verification Methods for Finite State Systems, LNCS Vol. 407, Springer-Verlag, NewYork, 1989, pp. 365-373.

[23] O. Coudert and J. C. Madre, "A Unified Framework for the Formal Verification of Sequential Circuits", In Proc. Internation Conf. on Computer-Aided Design, pp. 126-129.

[24] P. Curzon, "The Formal Verification of the Fairisle ATM Switching Element", Technical Reports 328 & 329, University of Cambridge, Computer Laboratory, March 1994.

[25] P. Curzon and I. Leslie, "Improving Hardware Designs whilst Simplifying their Proof", *Designing Correct Circuits*, Workshops in Computing, Springer-Verlag, 1996.

[26] R. Drechsler, B. Becker and S. Ruppertz, "K*BMDs: A New Data Structure for Verification", In *Proc. IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods,* Frankfurt, Germany, October 1995.

[27] E. A. Emerson, "Temporal and Modal Logic", In *Handbook of Theoretical Computer Science*, Elsevier Science Publishers B. V., 1990, Chapter 16.

[28] E. Garcez, "The Verification of an ATM Switching Fabric using HSIS tool", Technical Report, *WSI-95-13*, Tubingen University, Germany, 1995.

[29] A. Ghosh, S. Devadas and A. R. Newton, *"Sequential Logic Testing and Verification"*, Kluwer Academic Publishers, 1992.

[30] M. Gordon and T. Melham, *"Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic"*, Cambridge, UK, Cambridge Univ. Press, 1993.

[31] A. Gupta, "Formal Hardware Verification Methods: A Survey", *Journal of Formal Methods in System Design*, Kluwer Academic Publishers, Vol. 1, No. 2/3, 1992, pp. 151-238.

[32] R. Händel, Manfred N. Huber and Stefan Schröder, "ATM Networks: Concepts, Protocols, Applications", 3rd Ed., Harlow and Addison-Wesley, 1998.

[33] R. H. Hardin, Z. Har'El and R. P. Kurshan, "COSPAN", In *Proc. 8th International Conf. on Computer-Aided Verification*. LNCS, vol.1102, Springer-Verlag, New York, 1996, pp. 423-427.

[34] T. Jackson, “Verification Critical Path for Today’s IC Designers”, *Electronics Journal*, Technical: Feature Article, September 1999.

[35] J. J. Joyce, “Multi-level Verification of Microprocessor-based Systems”, PhD Thesis, Computer Laboratory, University of Cambridge, May 1990.

[36] M. Kaufmann and J. S. Moore, ACL2, “An industrial strength version of Nqthm”, *In Proc. 11th Annual Conf. on Computer Assurance,* Gaithersburg, MD, June,1996, S. Faulk and C. Heithayer, Eds. pp. 23-34.

[37] C. Kern and M. Greenstreet, “Formal Verification in Hardware Design: A Survey”, *ACM Transactions on Design Automation of Electronic Systems*, Vol. 4, April 1999, pp. 123-193.

[38] R. P. Kurshan, “Analysis of discrete event coordination”, *Stepwise Refinement of Distributed Systems*, LNCS, Vol. 430, Springer-Verlag, NewYork, 1989, pp.414-453.

[39] R. P. Kurshan, "Formal Verification in a Commercial Setting", In *Proc. Design Automation Conference*, Anaheim, California, June 1997, pp. 258-262.

[40] M. Langevin and E. Cerny, “An Extended OBDD Representation for Extended FSMs”, In *Proc. of EDAC-ETC-EUROASIC,* 1994.

[41] I. Leslie and D. McAuley, “Fairisle: A ATM Netwrok for Local Area”, *ACM Communication Review*, Vol. 19, pp. 237-336, September 1991.

[42] J. Lu, S. Tahar, D. Voicu and X. Song, “Model Checking of a Real ATM Switch”, In *Proc. IEEE International Conference on Computer Design*, Austin, Texas, USA, IEEE Computer Society Press, October 1998, pp. 195-198.

[43] J. Lu and S. Tahar, “Practical Approaches to the Automatic Verification of an ATM Switch Fabric using VIS”, In *Proc. IEEE 8th Great Lakes Symposium on VLSI*, Lafaytte, Louisiana, USA, February 1998, pp. 368-373.

[44] K.L. McMillan, “*Symbolic Model Checking*”, Norwell, MA, Kulwer, 1993.

[45] T. F. Melham, “Abstraction Mechanisms for Hardware Verification”, In *VLSI Specification Verification and Synthesis*, G. Birtwistle and P. Subrahmanyam, Eds. Kluwer Academic Publishers, Hingham, MA, 1988, pp. 267-291.

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

[47] PMC-Sierra Inc., “*Receive APS, Synchronization Status and BERM Telecom System Block Engineering Document*”, Issue 4, January 29, 1998.

[48] PMC-Sierra Inc., “*SCI-PHI Transmit Master and Receive Slave TSB Specification*”, Issue 2, May 10, 1999.

[49] H. Peng and S. Tahar, “Compositional Verification of IP Based Designs”, In *Proc. IFIP International Workshop on IP Based Synthesis and System Design*, Grenoble, France, December 1999, pp. 189-193.

[50] S. Rajan, M. Fujita, K. Yuan, and M. Lee, “High-Level Design and Validation of ATM Switch”, In *Proc. IEEE International High Level Design Validation and Test Workshop*, Oakland, California, USA, November 1997.

[51] S. Tahar and R. Kumar, “Implementing a Methodology for Formally Verifying RISC Processors in HOL”, *Higher Order Logic Theorem Proving and its Applications*, LNCS 780, Springer Verlag, 1994, pp. 281-294.

[52] S. Tahar, X. Song, E. Cerny, Z. Zhou, M. Langevin and O. Ait- Mohamed, “Modeling and Verification of the Fairisle ATM Switch Fabric using MDGs”, *IEEE Transactions on CAD of Integrated Circuits and Systems*, Vol. 18, No. 7, July 1999, pp. 956-972.

[53] 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”, In *Proc. Correct Hardware Design and Verification Methods*, Bad Herrenalb, Germany, September 1999, pp. 110-124.

[54] Y. Xu, E. Cerny, X. Song, F. Corella, O. Mohamed, “Model Checking for First-Order Temporal Logic using Multiway Decision Graphs”, In *Proc. of Conference on Computer Aided Verification*, July 1998.

[55] Y. Xu. “Model Checking for a First-order Temporal Logic Using Multiway Decision Graphs”, PhD Thesis, Dept. of Information and Operational Research, University of Montreal, Montreal, Canada, 1999.

[56] Y. Xu, “*MDG Model Checker User’s Manual*”, Dept. of Information and Operational Research, University of Montreal, Montreal, Canada, September 1999.

[57] Z. Zhou, “Multiway Decision Graphs and their Applications in Automatic Verification of RTL Designs”, PhD. Thesis, Dept. of Information and Operational Research, Universite of Montreal, Montreal, Canada, 1997.

[58] Z. Zhou, X. Song, S. Tahar, E. Cerny, F. Corella, M. Langevin, “Formal Verification of the Island Tunnel Controller using Multiway Decision Graphs”, in *Formal Methods in Computer-Aided Design*, Lecture Notes in Computer Science 1166, M. Srivas and A. Camilleri, Eds. Berlin, Germany: Springer-Verlag, pp. 233-246, 1996.

[59] Z. Zhou and N. Boulerice, “MDG Tools (v1.0) User’s Manual”, Dept. of Information and Operation Research, University of Montreal, Montreal, Canada, 1996.

[60] M. H. Zobair, S. Tahar, and P. Curzon, “The Impact of Design Changes on Verification using MDGs”, *In Proc. IEEE Canadian Conference on Electrical & Computer Engineering*, Halifax, Nova Scotia, Canada, May 2000. pp. 173-178.

Appendix A

Properties Description for MDG Model Checking

Property 1: According to the specification of SONET Transport System in [1]: The filtered S1 byte of a SONET frame needs to be identical for eight consecutive frames. If eight consecutive frames do not contain identical S1 bytes an interrupt is generated to indicate that the filtered S1 value has changed. When the TSB is in *state_ssd = 6*, it means that no 7 consecutive frames contain identical S1 bytes. If the next frame does not have identical byte, interrupt *s1i* will go to high in the next cycle. In L_{MDG} this safety property is expressed as follows:

```
AG((!(rstb=0)&(rclk=1)&(toh_ready=1)&(((s1_cap=1)&(state_ssd=6)&
(s1_in=s1_last_reg)&(!(s1_in=s1_filter_reg))))) -> (X(s1i=1)) );
```

Property 2: According to the specification of the SONET Transport System in [1]: The APS bytes, i.e., K1 and K2 bytes, should be identical for 3 consecutive frames. If there is a change in these APS bytes within 3 consecutive frames, an interrupt will be generated to indicate that a change in APS bytes has occurred. When the TSB in *state_aps = 1* and the current values of APS bytes are not identical with their previous filtered values, the interrupt will go to high to indicate a change in APS bytes. In L_{MDG} this safety property is expressed as follows:

```
AG((!(rstb=0)&(rclk=1)&(toh_ready=1)&(state_aps=1)&(!(k1_fil_reg
    = k1_in)|!(k2_fil_reg=k2_in))) -> (X(coapsi=1)));
```

Property 3: According to the specification of SONET Transport System in [1]: An alarm for the automatic protection switch failure will be triggered, i.e., *psbfv = 1*, whenever the TSB receives 12 frames in which 3 consecutive frames do not contain identical K1 or K2 bytes. In L_{MDG} this safety property is expressed as follows:

```
AG(((state_psf = 10) & (state_aps = 0) & (! (rstb = 0) ) & (rclk=1)&
    (toh_ready=1)) -> (X(psbfv=1)) );
```

Property 4: The TSB generates the protection switch failure interrupt, i.e., *psbfi* = 1, if the protection switch failure alarm is not stable. This means that an interrupt will never be triggered whenever the current alarm value does not differ from its previous value, i.e., in stable condition. The expression of this safety property in L_{MDG} is as follows:

```
AG( ( ( (psbfv = 0) & ( psbfv_last_reg = 1) ) | ( (psbfv = 1) &
     ((psbfv_last_reg = 0) ))) -> (X(psbfi = 1)));
```

Property 5: The *toh_ready* input is used as a synchronization signal. It must be high for only one clock cycle per SONET frame. The k1_in, k2_in and s1_in inputs are observed only when *toh_ready* is high. When this signal is low, eventually all the inputs related to the transport overhead processing of the TSB will be low. The L_{MDG} expression of these liveness properties are as follows:

```
AG( (toh_ready=0) => (F((s1i = 0)&(coapsi = 0))) );
```

Property 6: In this property, we define the overhead byte extraction behavior of the TSB. As the L_{MDG} syntax does not support abstract variables in the left hand term of the formula, we need to create a concrete variable using original signals related to design ele-

ments and a MDG table. A cross-operator *eq_ex* of type ($[worda8, worda8] \rightarrow bool$]) and two *abstract* variables indicating the location of the overhead bytes are used to create this extra variable of *concrete* sort. In the following formula, *s1_rin_equal* is a concrete signal generated by two cross-operators *eq_ex*(*column, zero*) and *eq_ex*(*row, eight*). The variable *s1_rin_equal* =1, if both of the *cross-operators* give an output equal to 1. The non-filtered value of S1 bytes will be available on the output port, i.e., *s1_tsb,* if the byte extractor extracts the overhead bytes from the first column of the ninth row within a frame. In L_{MDG} this safety property is expressed as follows:

```
AG( (!(rstb=0)&(rclk=1)&(s1_rin_equal=1)) -> (s1_tsb = rin) );
```

Property 7: The function of the BERM is to monitor the BIP error line over a defined declaration period and set an alarm if the declaration threshold is exceeded. When the calculated BER exceeds a *declaration* threshold value, i.e., *declare_th,* the BERM status alarm *berv* goes high. If the calculated BER value is under the *clearing* threshold, the alarm will reset. To check this threshold value, i.e., *dcount*, the BERM module needs to perform several arithmetic operations which include additions and incrementing of larger sized data (see Figure 12 and 14). In the following expression, we use expressions *declare_thm = 1 and mclear_th = 0* instead of (declare_th < = dcount) and (count > = *clear_th*), respectively. Because the L_{MDG} syntax does not support relational expressions like, $X \geq Y$ or $M \leq N$ in the formula. To get the value of *declare_thm* and *mclear_th,* we use an additional MDG table which contains *cross-operator* to compare the input signals. In L_{MDG} this safety property is expressed as follows:

```
AG((! ( rstb = 0) & (bipclk = 1) & (berten = 1) & (declare_thm = 1)
    & (mclear_th = 0) ) -> ( X (berv = 1)) );
```

Property 8: The TSB generates an interrupt, whenever the *berv* status is changed, i.e., unstable. This means that the interrupt will never be triggered, i.e., *beri =1*, if the current value of *berv* does not differ from its previous value, i.e., stable condition. In L_{MDG} this safety property is expressed as follows:

```
AG(( !(rstb = 0) & (bipclk = 1) & ( (( berv = 1) &(berv_last_reg=0
    ) )| ( (berv = 0) & (berv_last_reg = 1)))) -> (X(beri=1)) );
```

Property 9: When an event occurs on the inputs of the interrupt server, the interrupt output of the TSB goes high. The inputs of the interrupt server are connected to the interrupt lines of the BERM, APSC and Sync_Status modules. Whenever any of these interrupt lines, i.e., *beri*, *psbfi*, *s1i* and *coapsi*, goes high, the interrupt line of the TSB will be set, i.e., *int = 1*. In L_{MDG} this safety property is expressed as follows:

```
AG((!(rstb=0)&(int_rd=0)&((rclk=1)&((s1i=1)|(coapsi=1)|(psbfi=1)))
    &((biclk=1)&(beri=1)) -> (int = 1) );
```

Property 10: This reset property checks the reset behavior of the TSB. When the asynchronous active low reset line is active, i.e., *rstb* = 0, all the outputs of the TSB should remain low. In L_{MDG} this safety property is expressed as follows:

```
AG((rstb=0)-> (s1i=0)&(coapsi=0)&(psbfi=0)& (psbfv=0)& (berv=0)
     & (beri=0));
```

Property 11: When the values of an APS failure alarm are stable, we need to make sure that the interrupt line related to this value eventually goes low. In L_{MDG} this liveness property is expressed as follows:

```
AG(((psbfv=psbfv_last_reg)&(!(rstb=0))&(rclk=1)) =>(F(psbfi=0)));
```

Property 12: When the value of BERM declaration threshold alarm is stable, we need to make sure that the interrupt lines related to this value eventually goes low. In L_{MDG} this liveness property is expressed as follows.

```
AG(((berv=berv_last_reg)&(!(rstb=0))&(bipclk=1))=>(F (beri=0)));
```

Appendix B

Properties Description for FormalCheck Model Checking

1. Constraints for Property Checking in FormalCheck:

```
Clock Constraint: Rclk
  Signal: pm5209:Rclk
  Extract: No
  Default: No
  Start: Low
      1st Duration: 1
      2nd Duration: 1
Clock Constraint: Bipclk
   Signal: pm5209:Bipclk
   Extract: No
   Default: No
   Start: Low
      1st Duration: 1
      2nd Duration: 1
Reset Constraint: Rstb
Signal: pm5209:Rstb
Default: Yes
Start: Low
      Transition Duration Value
        Start        2     0
               forever     1
```

2. Properties Description in FormalCheck:

Property 1:

```
Property: Property_1
Type: Always
After: (@s1_ready)and (pm5209:Toh_Process_Inst:Sync_Status_Inst:
       Filter:Match_Count = 6)and
       (pm5209:Toh_Process_Inst:Sync_Status_Inst:Temp1 = TRUE)and
       (pm5209:Toh_Process_Inst:Sync_Status_Inst:Temp2 = FALSE)
Always: pm5209:Toh_Process_Inst:S1i = 1
Options: Fulfill Delay: 0   Duration: 1 counts of
         pm5209:Toh_Process_Inst:Rclk = rising
```

Property 2:

```
Property: Property_2
Type: Always
After: @k_filter and @k_ready and @k_last
      and
pm5209:Toh_Process_Inst:Apsc_Inst:Filter_K1k2:Match_Count=1
Always: pm5209:Toh_Process_Inst:Coapsi = 1
Options: Fulfill Delay: 0   Duration: 1 counts of pm5209:Rclk =
rising
```

Property 3:

```
Property: Property_3
Type: Always
After: @k_ready and (pm5209:Toh_Process_Inst:Apsc_Inst:
       Psbf_Monitor:Mismatch_Count = 10) and
       ((pm5209:Toh_Process_Inst:Apsc_Inst:Psbf_Monitor:Match_Cou
       nt=0)
       or(pm5209:Toh_Process_Inst:Apsc_Inst:Psbf_Monitor:Match_Co
       unt=1 and @k1_neq))
Always: pm5209:Toh_Process_Inst:Psbfv = 1
Options: Fulfill Delay: 0   Duration: 1 counts of pm5209:Rclk =
rising
```

Property 4:

```
Property: Property_4
Type: Never
Never: pm5209:Toh_Process_Inst:Psbfi =1 and
       (pm5209:Toh_Process_Inst:Apsc_Inst:Psbf_Monitor:Temp_Psbfv
       =
       pm5209:Toh_Process_Inst:Apsc_Inst:Psbf_Interrupt:Psbfv_Las
       t_Reg) and @k_ready
Options:(None)
```

Property 5:

```
Property: Property_5
Type: Eventually
After: pm5209:Toh_Process_Inst:Toh_Ready = 0
Eventually: pm5209:Toh_Process_Inst:Coapsi = 0 and
            pm5209:Toh_Process_Inst:S1i = 0
Options:(None)
```

Property 6:

```
Property: Property_6
Type: Always
```

```
After: pm5209:Toh_Process_Inst:Toh_Extract_Inst:Column = 0 and
       pm5209:Toh_Process_Inst:Toh_Extract_Inst:Row = 8 and
       pm5209:Rclk = 1 and pm5209:Rstb /= 0
Always: pm5209:S1 = pm5209:S1_Tsb
Options: Fulfill Delay: 0   Duration: 1 counts of pm5209:Rclk =
rising
```

Property 7:

```
Property: Property_7
Type: Always
After: (@Enable_berm) and
       (pm5209:Berm_Inst:Declare_Th <= pm5209:Berm_Inst:Dcount)
       and
       (pm5209:Berm_Inst:Ccount_Tst >= pm5209:Berm_Inst:Clear_Th)
Always: pm5209:Berm_Inst:Berv = 1
Unless:(pm5209:Berm_Inst:Declare_Th>=pm5209:Berm_Inst:Dcount)or(p
     m5209:Berm_Inst:Clear_Th <= pm5209:Berm_Inst:Ccount)
Options:(None)
```

Property 8:

```
Property: Property_8
Type: Always
After: pm5209:Rstb /= 0 and pm5209:Bipclk = 1 and
       pm5209:Berm_Inst:Berv /= stable and @Enable_berm
Always: pm5209:Berm_Inst:Beri = 1
Options: Fulfill Delay: 0   Duration: 1 counts of pm5209:Bipclk =
rising
```

Property 9:

```
Property: Property_9
Type: Always
After: pm5209:Rstb /= 0 and pm5209:Int_Rd = 0 and (pm5209:Rclk = 1
       and (pm5209:Toh_Process_Inst:Sync_Status_Inst:S1i = 1 or
       pm5209:Toh_Process_Inst:Apsc_Inst:Coapsi = 1 or
       pm5209:Toh_Process_Inst:Apsc_Inst:Psbfi=1))and
       (pm5209:Bipclk=1 and pm5209:Berm_Inst:Beri = 1)
Always: pm5209:Int = 1
Unless: pm5209:Int_Rd = 1
Options:(None)
```

Property 10:

```
Property: Property_10
Type: Always
After: pm5209:Rstb = 0
```

```
Always: pm5209:Toh_Process_Inst:Apsc_Inst:Coapsi = 0 and
        pm5209:Toh_Process_Inst:Apsc_Inst:Psbfi   = 0 and
        pm5209:Toh_Process_Inst:Apsc_Inst:Psbfv = 0 and
        pm5209:Toh_Process_Inst:Sync_Status_Inst:S1i = 0 and
        pm5209:Berm_Inst:Berv = 0 and pm5209:Berm_Inst:Beri = 0
Unless: pm5209:Rstb /= 0
Options:(None)
```

Property 11:

```
Property: Property_11
Type: Eventually
After: pm5209:Rstb /= 0 and pm5209:Rclk = 1 and
pm5209:Toh_Process_Inst:Apsc_Inst:Psbfv = stable
Eventually:pm5209:Toh_Process_Inst:Apsc_Inst:Psbfi = 0
Options:(None)
```

Property 12:

```
Property: Property_12
Type: Eventually
After: pm5209:Rstb/=0andpm5209:Bip-
clk=1andpm5209:Berm_Inst:Berv=stable
Eventually: pm5209:Berm_Inst:beri = 0
Options:(None)
```

3. Macros Expressions used in the Properties:

@s1_ready: ((pm5209:Toh_Process_Inst:Rclk = 1) and (pm5209:Toh_Process_Inst:Rstb /= 0))
and (pm5209:Toh_Process_Inst:Sync_Status_Inst:S1_Ready = 1)

@k_filter: (pm5209:Toh_Process_Inst:Apsc_Inst:K1_In /=
pm5209:Toh_Process_Inst:Apsc_Inst:K1_Filter_Reg) or
(pm5209:Toh_Process_Inst:Apsc_Inst:K2_In /=
pm5209:Toh_Process_Inst:Apsc_Inst:K2_Filter_Reg)

@k_last: (pm5209:Toh_Process_Inst:Apsc_Inst:K1_In =
pm5209:Toh_Process_Inst:Apsc_Inst:K1_Last_Reg)
and(pm5209:Toh_Process_Inst:Apsc_Inst:K2_In =
pm5209:Toh_Process_Inst:Apsc_Inst:K2_Last_Reg)

@k_ready: ((pm5209:Toh_Process_Inst:Apsc_Inst:Rclk = 1)
and(pm5209:Toh_Process_Inst:Apsc_Inst:Rstb = 1)) and
(pm5209:Toh_Process_Inst:Apsc_Inst:K_Ready= 1)

@k1_neq: pm5209:Toh_Process_Inst:Apsc_Inst:K1_In /=
pm5209:Toh_Process_Inst:Apsc_Inst:K1_Last_Reg

@Enable_berm: ((pm5209:Rstb /= 0) and (pm5209:Bipclk = 1)) and (pm5209:berten= 1)