On the Modeling and Verification of a Telecom System Block Using MDGs

M. Hasan Zobair and Sofiène Tahar

Electrical & Computer Engineering Department, Concordia University
Montreal, Quebec, Canada
Email: {mh_zobai, tahar}@ece.concordia.ca

Technical Report

December 2000

Abstract. *In this report, we investigate the ability of MDGs (Multiway Decision Graphs) to carry out a verification process of a large industrial Telecom hardware which is commercialized by PMC-Sierra Inc. Until recently, the Cambridge Fairisle ATM switch fabric with 4200 equivalent gates was the largest industrial like design verified with the MDG tools. The design we consider in this study is a Telecom System Block (TSB), called RASE, containing 11400 equivalent gates. For the formal verification, we adopted a hierarchical proof methodology to handle the complexity of the design. We then carried out MDG based equivalence checking as well as model checking. To measure the performance of the MDG verification, we also conducted the verification of the same TSB with Cadence FormalCheck. The experimental results showed that in some cases, the MDG model checker was more efficient due to the ability with MDGs to use abstract state variables and uninterpreted function symbols rather than simply a Boolean modeling as in FormalCheck.*

1. Introduction

Every design, no matter how strategic or complex, requires multifaceted verification before marketing. Starting from final manufacturing and moving back through previous phases of the design process, we might find many objects for verification. We restrict our interest to one particular item — the verification of functional correctness through formal verification. Simulation is the most widely used technique for checking hardware designs. In the early stages of design, simulation enables the designer to find and fix errors. In the final stages, however, simulation is not as effective, and some errors can remain hidden. The most serious problem is that simulation does not definitely ensure the conformance of designs to specifications. This handicap is the reason why we need alternative verification methods such as formal verification [9]. 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. Since automated decision graphs based verification techniques are relatively easier to apply than simulation, which needs test vectors and a test bench, we may use them when the design is partially defined. Findings bugs early in the design cycle is a well-known accelerant of design development. Model checkers and equivalence checkers paved a path, showing the utility of finding bugs in the design cycle. Due to the above reasons formal verification has gained a growing interest in the industrial community [10].

In this report, we present a methodology for the formal verification of a real industrial design using Multiway Decision Graphs (MDG) [4]. The design we considered is a Telecom System Block (TSB) — the **R**eceive **A**utomatic Protection Switch Control, **S**ynchronization Status Extraction and Bit **E**rror Rate Monitor Telecom System Block (RASE TSB), a commercial product of PMC-Sierra, Inc.[15]. The main aspect of this work is to illustrate the ability to carry out the verification process of a large industrial design using MDGs. The verification process was carried through both equivalence and model checking. Until recently the Fairisle ATM (Asynchronous Transfer Mode) Switch Fabric [16] is the large design verified by MDGs which needs 4200 equivalent gates implemented in Xilinx FPGAs. In comparison to this number, our investigated design has 11400 equivalent gates [15].

The RASE TSB processes a portion of the SONET (Synchronous Optical Network) [1] line overhead of a received SONET data stream. It processes the first Synchronous Transport Signal (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). The RASE TSB exists in both single and dual BERM (Bit Error Rate Monitor) versions. Both versions use the same logic, register addressing and provide the same outputs. For cost sensitive applications only the Signal Failure (SF) BERM is present and Signal Degrade (SD) BERM is removed as well as its normal mode registers. As both modes use the same logic and provide the same functionality, we are concerned only with the single mode version.

The outline of this paper is as follows: In Section 2, we give an overview of Multiway Decision Graphs (MDGs). In Section 3, we describe the functionality of the RASE TSB. In Section 4, we present the hierarchical behavioral modeling of the TSB. Section 5 describes the implementation of the TSB. In Section 6, we show our hierarchical verification methodology including a comparison of the verification process between MDG and Cadence FormalCheck. Section 7 concludes the paper.

2. Multiway Decision Graphs

Hardware formal verification techniques naturally group themselves into theorem proving and automated decision diagram based methods [9]. In the theorem proving approach, the specification and implementation are usually expressed in first-order or higher-order logic. Theorem proving methods have showed their efficiency in verifying datapath-dominated designs. It is more powerful and expressive but is not automated and needs expertise. The decision diagram based methods use state space algorithms on finite state machine models to check behavioral equivalence or certain properties on the system. As the control circuitry is naturally modelled as FSM, automata-oriented methods including propositional temporal logic, have shown their greatest promise in handling the control portion of the design. Though all the Reduced and Ordered Binary Decision Diagrams (ROBDD) based tools (e.g., SMV [13] and VIS [3]) are fully automatic, they are unable to handle very large designs due to the well known state space explosion [7]. Multiway Decision Graphs (MDGs) [4] have been proposed to solve this state space explosion problem of ROBDD based verification tools. While accommodating higher level of abstraction as theorem prover, the MDG tools offer automation in the verification process like ROBDD based tools. The underlaying logic of the MDGs is a subset of many-sorted first-order logic with a distinction between concrete and abstract sorts. A concrete sort has an enumeration while an abstract sort does not. Hence, a data signal can be represented by a single variable of abstract sort, rather than a vector of boolean variables, and data operations can be viewed as black boxes and

represented by uninterpreted function symbols. The MDG tools are thus much more efficient than ROBDD based tools for designs containing wider datapaths [21].

An MDG is a 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. MDGs incorporate variables of *concrete* as well as *abstract* sort to model control and data signals, respectively. It also supports *uninterpreted* function symbols to denote data operations. Constants as well can be either of concrete or abstract sort. The latter are called *generic* constants. As a special case of uninterpreted functions, *cross-operators* are useful for modeling feedback from the datapath to the control circuitry. MDGs can also represent sets of states. They are thus much more compact than ROBDDs for designs containing datapath, and sequential circuits can be verified independently of the width of the datapath.

In MDG-based verification, abstract descriptions of state machines, called abstract state machines (ASM) are used to model the systems. An ASM is obtained by letting some data inputs, states or output variables be of abstract sort, and the datapath operations be uninterpreted function symbols. They admit non-finite state machines as models in addition to their intended finite interpretations. This makes it possible to verify a circuit at the RTL functional model [20]. Because of this, the use of ASMs raises the level of abstraction of automated verification methods to approach those of interactive theorem proving methods, without sacrificing automation.

The MDG tools [21] provide algorithms for equivalence checking, invariant checking and model checking, which are based on the reachability analysis of all states. The MDG tools package the MDG operators and verification procedures. The equivalence verification procedures are combinational and sequential verification. The combinational verification provides the equivalence checking of two combinational circuits based on isomorphism checking of the canonical form of MDGs. The sequential verification provides behavioral equivalence checking of two ASMs. The model checking supports an ACTL-like temporal logic, L_{MDG} [18]. The MDG tools run on a Prolog platform and accept a Prolog-style HDL — MDG-HDL, which allows the use of abstract variables for representing data signals. MDG-HDL supports structural descriptions, behavioral descriptions, or a mixture of structural and behavioral description. A structural description is usually a (hierarchical) network of components (modules) connected by signals. The MDG-HDL comes with a large library of predefined, commonly used, basic components (such as logic gates, multiplexers, registers, bus drivers, ROMs, etc.). A behavioral description is given by high-level constructs as ITE (If-Then-Else) formulas, CASE formulas or tabular representations. These high-level constructs are based on MDG tables. An MDG table is similar to a truth table but allows first-order terms in rows. The internal MDG data structure compiles this MDG-HDL description into the ASM model before building the MDGs for the particular verification procedure. For more information about MDG and the tools readers are referred to [4, 5, 18, 19, 20, 21, 22].

3. The RASE Telecom System Block

The RASE Telecom System Block (TSB) [15] consists of three types of components: Transport overhead extraction and manipulation, Bit Error Rate Monitoring (BERM) and Interrupt Server (see Figure 2). 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.

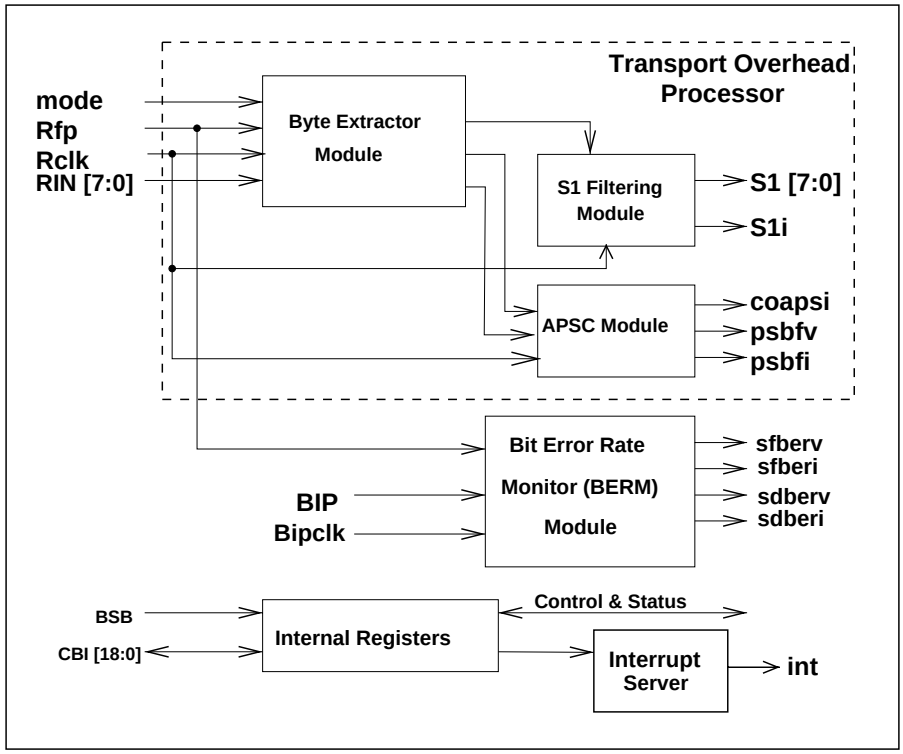


Figure 1: The RASE telecom system block

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 1). After extracting the above bytes, it processes them according to some requirements set by the SONET standard [1].

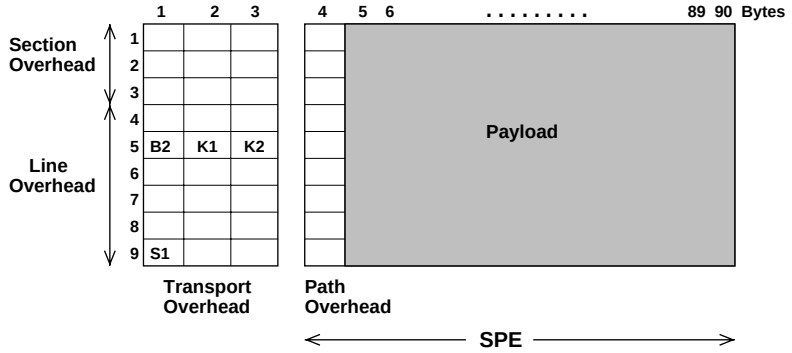


Figure 2: The STS-1 SONET frame structure

The TSB also performs Bit Error Rate Monitoring using the BIP-24/8 line of a frame, i.e., B2 bytes (Figure 1). 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 logic and offer the same functionality.

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. 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.

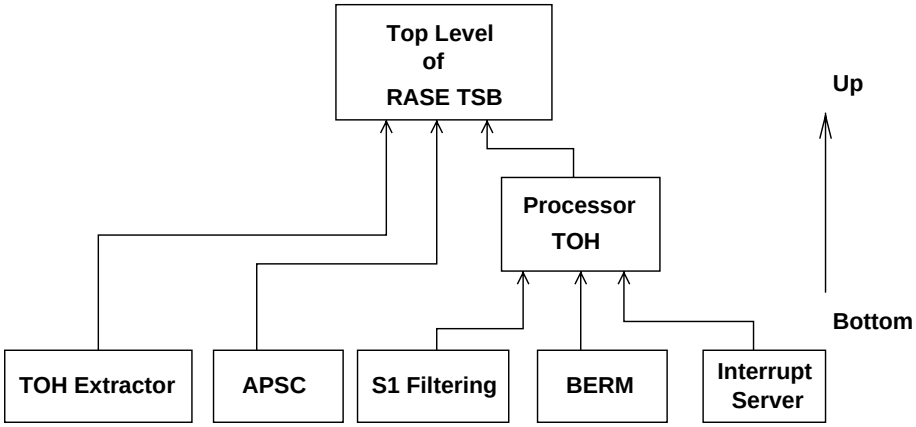


Figure 3: The hierarchy tree of the TSB

Inspired by [15], 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.

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.

4.1 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 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 [1]. 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 indi-

cated by the pointer bytes in the LOH which is not our interest in this work. The behavior of the extraction module based on a row and a column counting abstract state machine (ASM) rather than a finite state machine and an extractor which extracts the specific byte (i.e., *RIN*) within a SONET frame (i.e., *RIN*). The column counting ASM has five states — S0, S1, S2, S3 and S4 (see Figure 4), while the row counting ASM has three states — S0, S1 and S2 (see Figure 5).

The column counting state machine accepts five signals *clk, rst, rfp, sts* and *col_eq* as its input control signals (see Figure 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.*

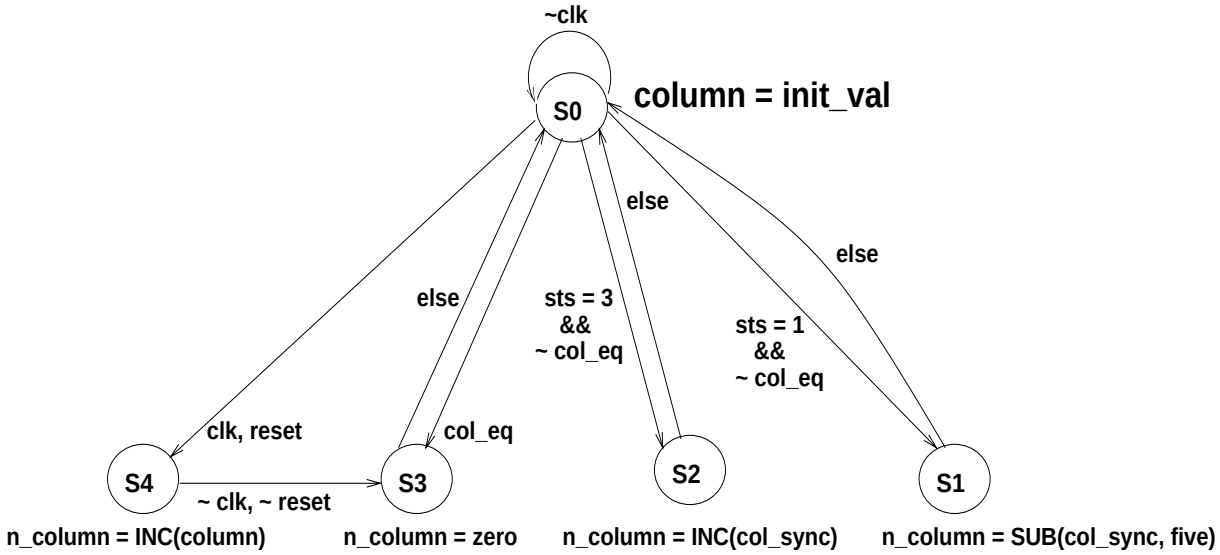


Figure 4: The column counting ASM

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.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 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.

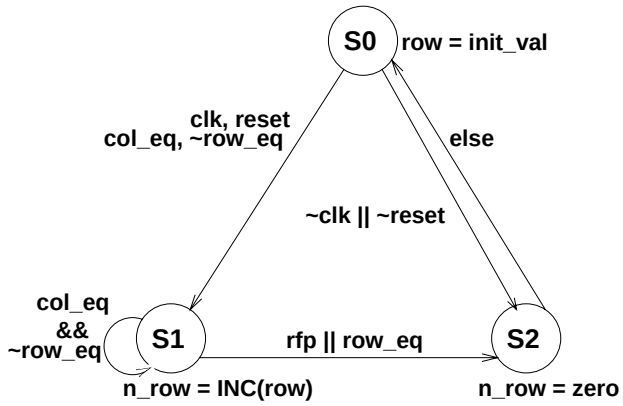
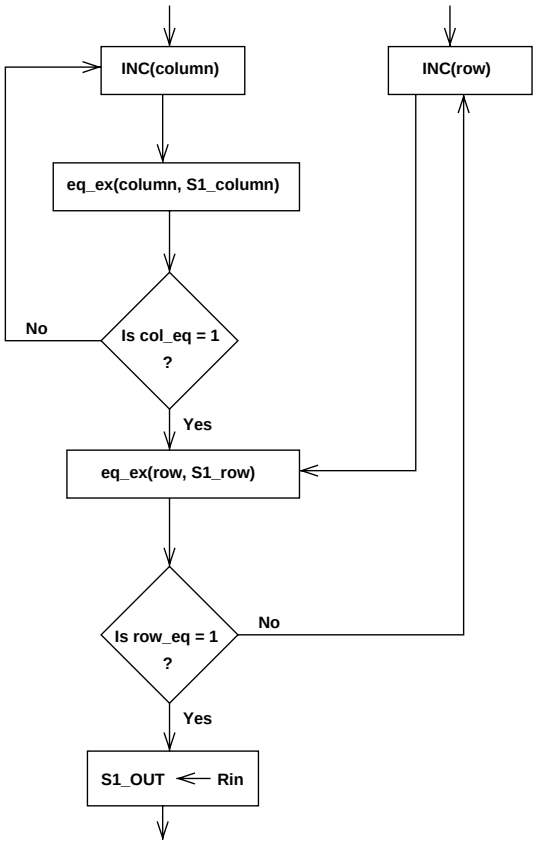


Figure 5: The row counting ASM

The behavior of an extractor can be described using a flowchart and its model in pseudo-MDG-HDL as shown in Figure 6. We can see from Figure 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 (Figure 4 and Figure 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 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.

Flowchart:



MDG-HDL model:

```
initialize column = zero
initialize row = zero
next(column) = INC(column)
next(row) = INC(row)
if (column = s1_column)
   then col_eq =1
   if (row  = s1_row)
      then row_eq =1
if (col_eq and row_eq)
   then next(S1_out) = RIN
else next(S1_out) = zero
```

Figure 6: Flowchart specification of byte extractor and its MDG model

4.2 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 [1].
- Generates two interrupt signals whenever these two APS bytes do not meet their specifications.

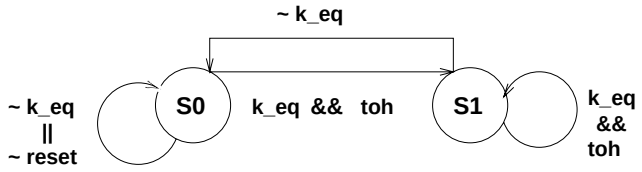
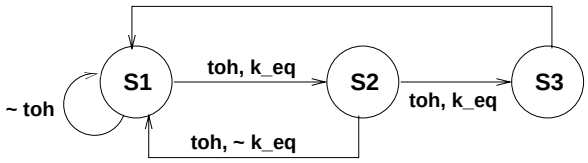
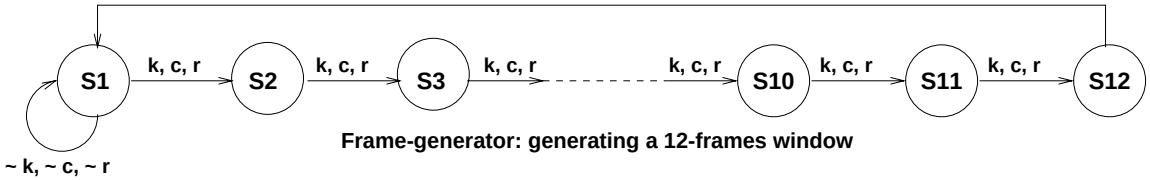


Figure 7: Filtering ASM for K1 and K2 bytes

The Filtering abstract state machine, having two states (S0 and S1), is shown in Figure 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. The symbol '||', '&&' and '~' denote logical OR, AND and negation of the signals, respectively. According to [1, 15], K1 and K2 bytes 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.



ASM_match: detecting identical K-bytes



Frame-generator: generating a 12-frames window

Figure 8: Set of ASMs to declare the APS failure alarm

The automatic protection switch failure monitoring is a complex behavior of the system. According to [1, 15], 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 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 8, the symbols *k*, *c* and *r* denote the arrival of overhead bytes ready to be processed, system clock and active low reset, respectively. The symbol "~" denotes the negation of the above signals.

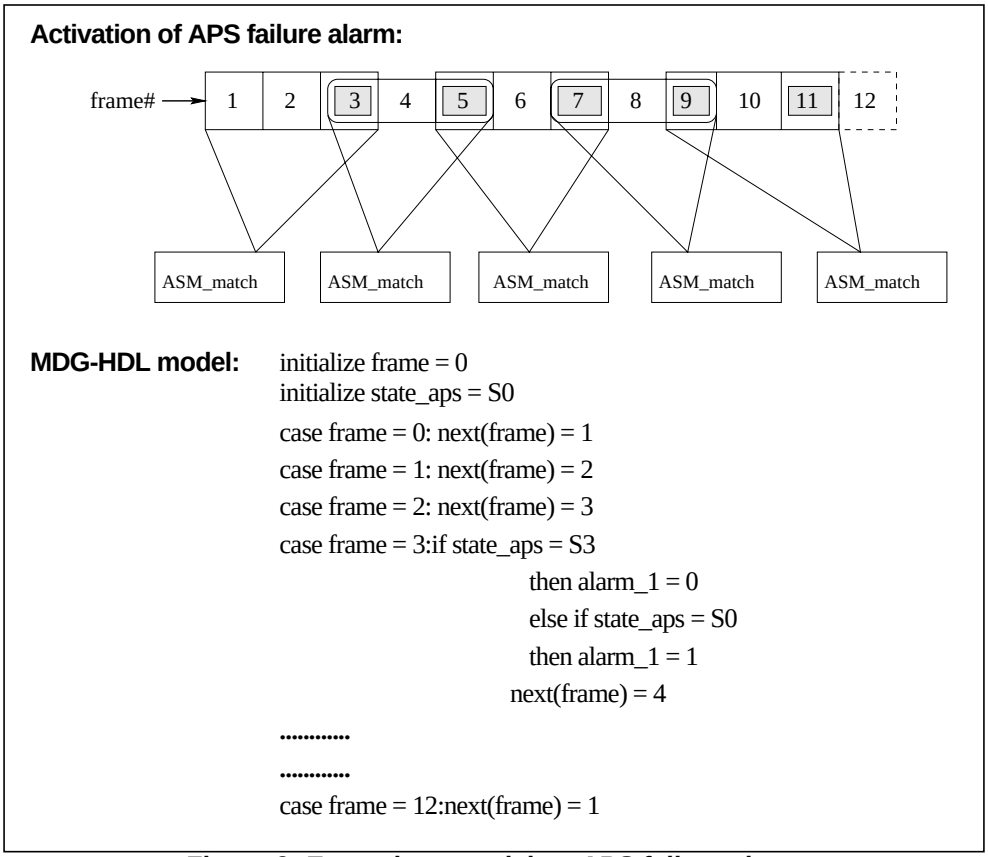


Figure 9: Example to model an APS failure alarm

The automatic protection switch failure monitoring is performed by the combination of two ASMs in Figure 8. A graphical representation with MDG modeling of the methodology to activate the automatic protection switch failure alarm is shown in Figure 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. Figure 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 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.

4.3 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 [1, 15]. 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 10). 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 10, represent the transport overhead byte ready to be processed and the result of comparison between present and previous values of S1 byte, respectively.

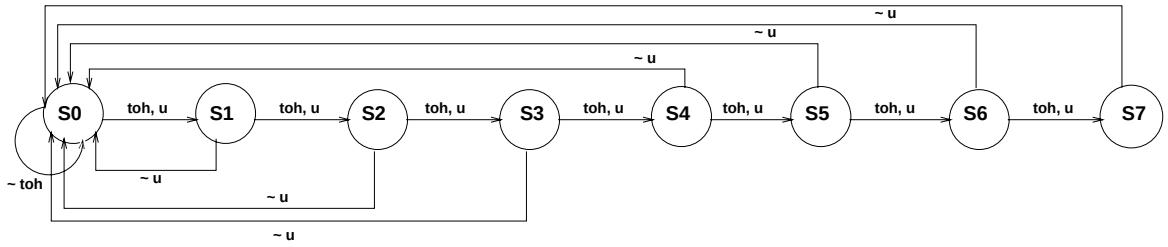
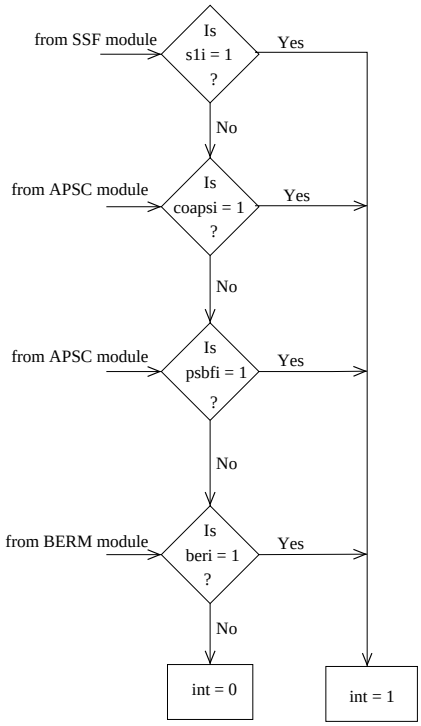


Figure 10: Abstract state machine to filter the S1 bytes

4.4 The Interrupt Server

The interrupt server has a very simple characteristic. Whenever a change in automatic protection switch, a protection switch failure, a change of the synchronization status, 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 11. We can see from Figure 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 11: Flowchart specification of interrupt server and its MDG model

4.5 Bit Error Rate Monitoring (BERM)

The Bit Error Rate monitoring is performed by a sliding window algorithm [15]. 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 need 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 12).

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.

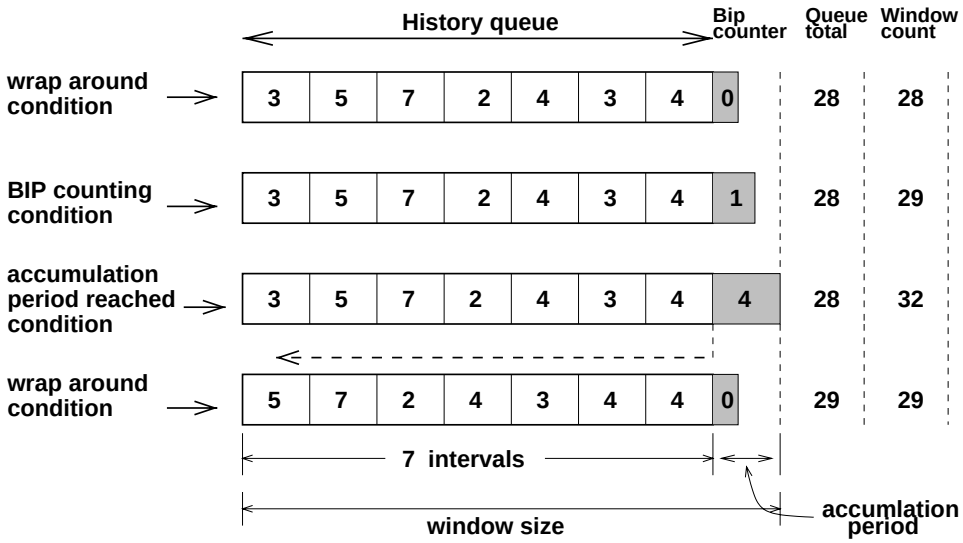
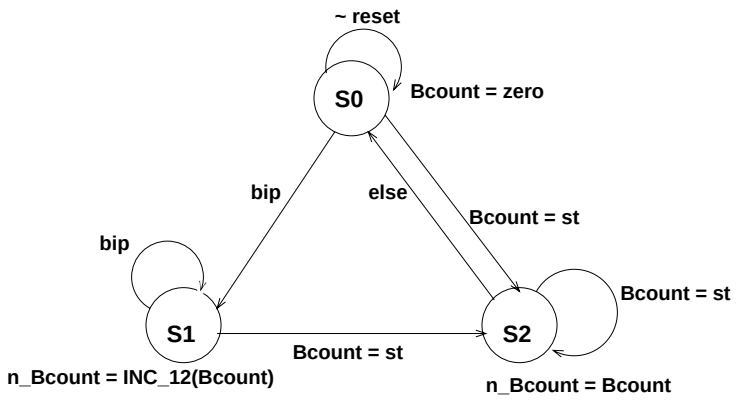


Figure 12: BERM sliding window algorithm

The BIPs accumulation is done in the eighth sub-interval of the sliding window (see Figure 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, 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 13. 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*.

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

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 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 13, where state variable *Bcount* of abstract sort represent 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 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 fact that the structure of MDG can be arbitrarily large, and it can be avoided by using some techniques described in [14, 21]. 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 13. 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.

5. 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.) [21]. To handle the complexity of the design which consists of a network of 11400 equivalent gates, we adopted a module abstraction technique.

For example, we can take the BERM module which is the largest component of the RASE TSB to illustrate the module abstraction technique (see Figure 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 14) using an *uninterpreted* function symbol **add_17** of type $[worda17 \rightarrow worda17]$. Likewise, We can increment the 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 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.*

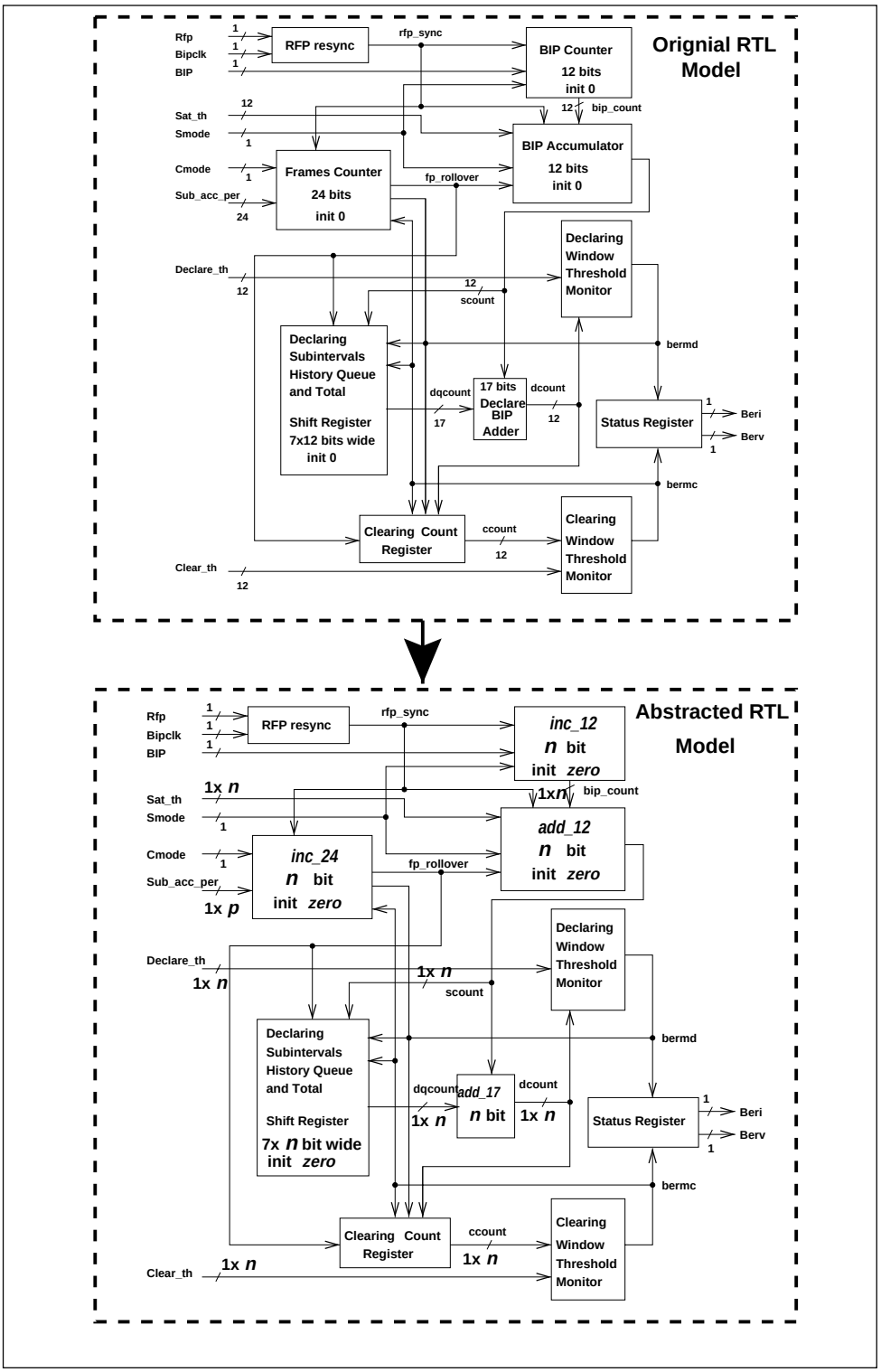


Figure 14: Module abstraction of the BERM block

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.

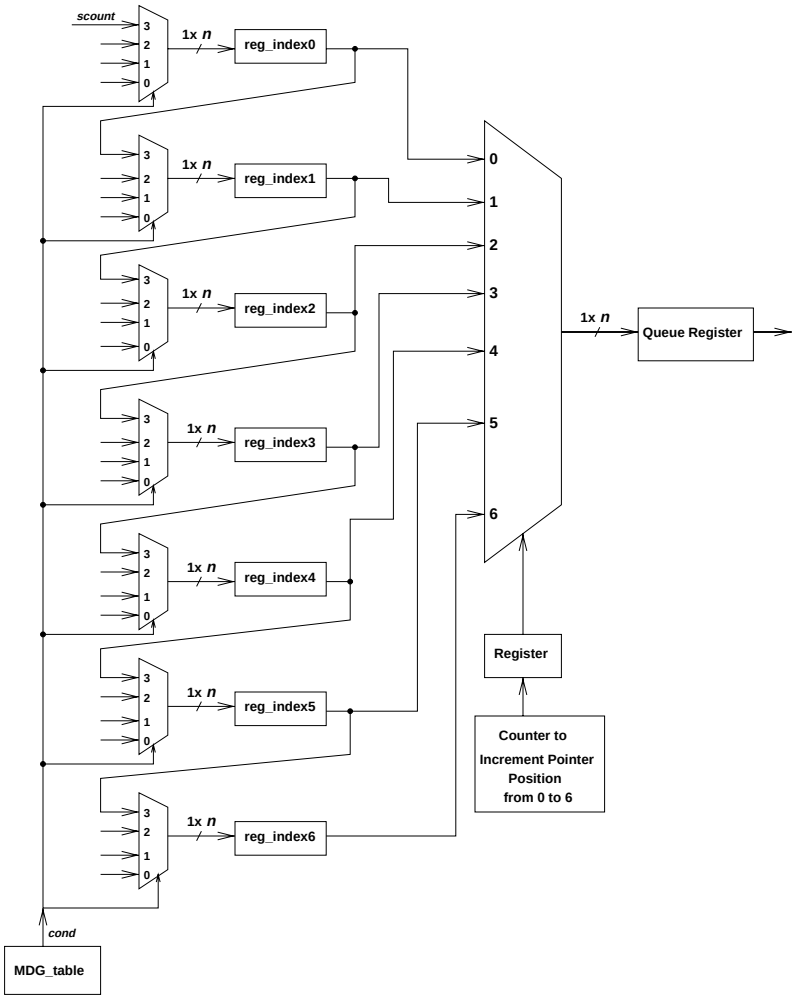


Figure 15: The Implementation of a multi-dimensional array using MDG-HDL

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. We describe our technique as follows (see Figure 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.

6. 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. 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., $B_{i\,[impl]}$, where i = 1,.., 3) is equivalent to its specification, i.e., $B_{i\,[spec]} \Leftrightarrow B_{i\,[impl]}$ 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., $S_{[spec]} = \Lambda\, B_{i\,[spec]}$. Similarly, we also derive an implementation of the whole system as a conjunction of the implementation of each sub-module, i.e., $S_{[impl]} = \Lambda\, B_{i\,[impl]}$. 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 16.

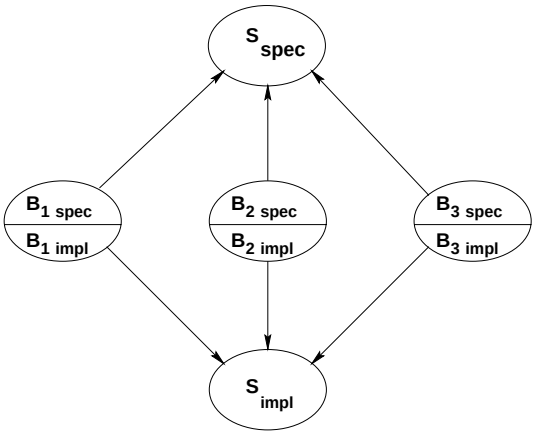


Figure 16: Hierarchical proof method

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. In the following two

sub-sections, we describe the verification process for the Telecom System Block using Equivalence checking and Model checking techniques.

6.1 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 many abstractions we adopted to the overall RASE TSB, we also succeeded to verify the top level specification of the RASE TSB against its total 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 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 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 operation.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 1: Experimental results for equivalence checking

Module name	CPU time (in seconds)	Memory (in MB)	No. 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

6.2 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 is *liveness properties* [7]. We describe several properties and their verification in the following two sub-sections.

6.2.1 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} [17, 19], which is a subset of ACTL that supports abstract data representations [18]. Both safety and liveness properties can be expressed in L_{MDG}, however, only universal path quantification is possible. In the following Abstract-CTL expressions, the symbols "!", "&", "|", "->" denote logical "not", "and", "or" and "imply", respectively. In the following we are describing several safety and liveness properties of the TSB.

Table 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 [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 elements 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*(*col-*

umn, 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))) &((bipclk=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)));

6.2.2 Properties Verification

The verification of the properties has carried out by using the model checking facility of MDG tools [17]. 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 5.1.1 for both specification and implementation, are given in Table 3 and Table 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 3 and 4 show that property checking on the implementation consumed more time and memory as the specification is more abstract than the original model.

Table 3: Experimental results of property checking on the specification of RASE TSB

Property	Module	CPU time (in seconds)	Memory (in MB)	No. of MDG nodes generated
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

It is known that in some cases the abstract reachability analysis may not terminate. This happens when the set of states is infinite or it cannot be represented finitely using the mechanisms presently available in the MDG system. When the 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 [21]. 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 the initial-state generalization and rewrite rules readers are referred to [14, 21].

Table 4: Experimental results of property checking on the implementation of RASE TSB

Property	Module	CPU time (in seconds)	Memory (in MB)	No. of MDG nodes generated
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

6.2.3 Comparison between Cadence FormalCheck and MDG Model Checking

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. [6]. 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 [2].

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 6.2.1 using FormalCheck property language. A full description of these properties is included in the Appendix. 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 FormalCheck, we constrained these clocks depending on the properties.

Table 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 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 [6] is more efficient in verifying FSM-based design, i.e., concrete data, than the MDG tools. Table 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 2).

7. Conclusions

In this report, we demonstrated that the Multiway Decision Graphs tools have the capability to verify a moderate sized industrial telecom hardware. Based on the product document provided by the PMC-Sierra Inc., we derived a behavioral model of the telecom system block. The specification was given as English test which was modeled in terms of Abstract State Machines using MDG-HDL. To handle the complexity of the RTL model, the module was abstracted by using MDG based abstract sorts and uninterpreted functions. We adopted a hierarchical verification approach to verify the whole TSB through MDG-based equivalence and model checking. Our verification did not find any errors in the existing design. During our RTL modeling of the TSB, we developed a technique to accommodate a multi-dimensional shift-register using basic MDG components.

One of the motivations of this work was to compare the verification of the TSB using MDGs with the verification done by an existing commercialized formal verification tool — Cadence FormalCheck model checker. The verification in FormalCheck has one major practical advantage over MDG, namely the VHDL/Verilog front-end which enables the integration with the design flow. The MDG-based approach can handle arbitrary data widths using abstract sort and uninterpreted functions which is a solution to the state space explosion problem. On the other hand, our experimental result shows that in some cases FormalCheck tool fails short due to state space explosion for wider datapath.

The experimental results in this work 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 wider datapath.

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 for MDG tools, we used manual variable ordering since no heuristic ordering algorithm is available in the current version. While doing the verification of the *RASE TSB*, much of the human time was spent on determining a suitable variable ordering. 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 which accepts the original VHDL structural model without any translation. 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 in FormalCheck, property checking on the implementation took about two man-weeks. Table 6 gives a summary of human time taken to model and verify the design using both MDG and FormalCheck tools, where "*" means that FormalCheck does not need that modeling or verification phase.

Table 6: Summary of human effort for the verification in MDGs and FormalCheck

Modeling and verification phases	MDG	FormalCheck
Understanding and Modeling the Behavior	3 man-months	*
Implementation of RTL Modeling	1 man-month	*
Defining properties and verifying properties	3 man-weeks	2 man-weeks
Equivalence checking between RTL vs. Beh.	1 man-week	*

References

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

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

[3] R. K. Brayton et. al, "VIS: A System for Verification and Synthesis", *Technical Report UCB/ ERL M95*, Electronics Research Laboratory, University of California, Berkeley, December 1995.

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

[5] 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, pp. 79-113, 1997.

[6] Cadence Design Systems, Inc., *Formal Verification Using Affirma FormalCheck*; Manual, Version 2.3, August 1999.

[7] 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.

[8] B. Chen, M. Yamazaki and M. Fujita, “Bug Identification of a Real Chip Design by Symbolic Model Checking” In *Proc. of International Conference on Circuits and Systems (ISCAS’94)*, London, UK, June 1994.

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

[10] P. Kurshan, “Formal Verification in a Commercial Setting”, in *Proc. Design Automation Conference (DAC’97)*, Anaheim, California, pp. 258-262, June 1997.

[11] J. Lu, S. Tahar, D. Voicu and X. Song, “Model Checking of a Real ATM Switch”, *Proc. IEEE International Conference on Computer Design* (ICCD,98), Austin, Texas, USA, IEEE Computer Society Press, pp. 195-198, October 1998.

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

[13] M.L. McMillan, “Symbolic Model Checking”, Norwell, MA, Kluwer, 1993.

[14] O. Ait-Mohamed, X. Song, E. Cerny, “On the Nontermination of MDG-based Abstract State Enumeration”, *Proc. IFIP Conf. correct Hardware and Verification Methods (CHARME’98)*, Montreal, Canada, pp. 218-235, October 1997.

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

[16] 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, pp. 956-972, July 1999.

[17] 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 (CAV’98)*, July 1998.

[18] Y. Xu. *MDG Model Checker User’s Manua*l, Dept. of Information and Operational Research, University of Montreal, Montreal, Canada, September 1999.

[19] 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.

[20] Z. Zhou. *Multiway Decision Graphs and their Applications in Automatic Verification of RTL Designs.* PhD Thesis, Dept. of Information and Operational Research, University of Montreal, Montreal, Canada, 1997.

[21] 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.

[22] Z. Zhou and N. Boulerice. *MDG Tools (V1.0) User’s Manual*, Dept. D’IRO, University of Montreal, 1996.

APPENDIX

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_Count=0)
or(pm5209:Toh_Process_Inst:Apsc_Inst:Psbf_Monitor:Match_Count=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_Last_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(pm5209: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:Bipclk=1andpm5209:Berm_Inst:Berv=stable

Eventually: pm5209:Berm_Inst:beri = 0

Options:(None)

3. Macros Expressions used in the property:

@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)