On the Formal Verification of a SONET Data Stream Processor

Sofiène Tahar[1] and M. Hasan Zobair[1]
Xiaoyu Song[2]

[1]Concordia University
Dept. of Electrical and Computer Engineering
1455 de Maisonneuve W., Montreal, Quebec, H3H 1M8, Canada
{mh_zobai,tahar}@ece.concordia.ca

[2]Portland State University
Dept. of Electrical and Computer Engineering
1800 SW 6th Ave., Portland, OR 97201, USA
song@ee.pdx.edu

Abstract

In this paper, we describe the formal verification of an industrial hardware design from PMC-Sierra, Inc. The design under investigation is a Telecom System Block, which processes a portion of the SONET (Synchronous Optical Network) line overhead of a received data stream. We adopted a hierarchical modeling and verification approach which follows the natural design hierarchy. The formal specification and verification have been carried out based on MDG (Multiway Decision Graphs), a new decision diagram subsuming the traditional binary decision diagrams and allowing abstract data and functions. The verification has been performed using both equivalence and model checking. To measure the performance of the MDG based model checker, we also conducted a comparative verification of the same design using Cadence FormalCheck.

1 Introduction

Simulation-based methods are currently being used by the industrial community for system-level verification, since they can handle the entire design at a time. Simulation, however, cannot provide a high coverage ratio due to the exponential number of test cases to be developed and verified. This handicap is the reason why new methods are needed for the economical and reliable verification of digital systems. Formal verification [12] have recently paved a path,

showing the utility of finding bugs early in the design cycle. Formal verification techniques are usually classified in two categories [12]: interactive theorem proving and automatic decision diagram based model checking and equivalence checking. In model checking, one checks if the design satisfies some properties (formal specification). With equivalence checking, we check if two designs exhibit the same behavior. The latter techniques have been successfully applied to various industrial designs. However, since most tools are based on Reduced Ordered Binary Decision Diagrams (ROBDDs [7]), they require the design to be described at the Boolean level. In practice, they often fail to verify a large-scale design because of the so-called state space explosion [12].

In this paper, we present a methodology for the formal verification of a real industrial design using the Multiway Decision Graphs (MDG) [9] tools. MDGs subsume the traditional binary decision diagrams while extending them with abstract data sorts and uninterrupted function symbols [9]. The design we considered is a Telecom System Block (TSB) from PMC-Sierra, Inc., called RASE—Receive, Automatic Protection Switch Control, Synchronization Status Extraction and Bit Error Rate Monitor [16]. It processes a portion of the SONET (Synchronous Optical Network) [5] line overhead of a received data stream. The main aspect of this paper is to illustrate the ability to carry out the verification of an industrial size design using MDGs. Furthermore, we conducted a comparison between the experimental results obtained with the MDG model checker and the model checking of the same design using Cadence FormalCheck [8].

The rest of this paper is organized as follows: Section 2 reviews some related work. Section 3 gives an overview of Multiway Decision Graphs (MDGs). In Section 4 we describe the functionality and structure of the RASE TSB and discuss their modelings with MDG. Section 5 describes our hierarchical verification methodology using both MDG equivalence and model checking. Section 6 presents a comparison of the verification process between MDG and FormalCheck. Section 7 finally concludes the paper.

2 Related Work

There exists a few related work on the application of the MDG tools in verification of telecommunication systems. For instance, Tahar *et al.* [18] verified the Fairisle [13] ATM (Asynchronous Transfer Mode [11]) switch fabric in an automatic fashion using MDGs by property and equivalence checking. The original design was modeled in Qudos HDL, containing 4200 equivalent gates implemented in Xilinx FPGAs. The authors used model abstraction techniques to reduce the state space of the gate level netlist based on abstract sorts and uninterpreted functions within MDG. They were then able to verify the whole switch fabric without experiencing any state space explosion problem. In comparison to the above ATM switch fabric, our investigated design represents with its 11400 equivalent gates [16] a significantly larger case study. Moreover, unlike the design presented in [18] in which an academic ATM switch fabric designed

at Cambridge University is investigated, our work presents a telecommunication design which is a commercial product. Furthermore, we performed both safety and liveness properties model checking on the investigated design, but for the work in [18], the authors applied only invariant (safety) property checking on the ATM design. Hence, in difference to [18], we were able to make a direct comparison with other model checking tools such as Cadence FormalCheck.

Another notable related work is the one done by Balakrishnan and Tahar [3]. The authors used the MDG tools to model and verify an embedded system of a mouse control application based on the PIC 16C71 Microcontroller from Microchip Technology, Inc. [15] They modeled the system at different levels of design hierarchy. 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. Although this work represents the sole commercial design verified by MDGs before ours, its application was concerned with software assembly aspects rather than hardware design and implementation.

Zhou *et al.* [21] demonstrated the MDG-based formal verification on the example of an Island Tunnel Controller design. In this work, they studied in detail the non-termination problem of abstract state enumeration [9] 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 [14] and VIS [6]. We are conducting a similar comparison with a more sophisticated tool, FormalCheck of Cadence, to check the efficiency of our MDG based verification.

Besides the above MDG related work, several other projects on the model checking of different moderate sized telecommunications digital systems are reported in the open literature. Some of these case studies are used to illustrate the limitations of current formal verification techniques in verifying industrial like designs. Among these limitations, state space explosion is the well-known problem faced by model checking methods when verifying designs with a substantial datapath. Researchers of these work presented different reduction and abstraction techniques to cope with this limitation. We discuss in following those work most related to ours.

Barakatain and Tahar [4] applied model checking techniques for the formal verification of a SCI-PHY Level 2 protocol engine (SCI-PHY is a super set of UTOPIA standard [2]). The authors used FormalCheck to formally verify the RTL (Register Transfer Level) implementation of the Receive Slave SCI-PHY mode of the Transmit Master/Receive Slave (TMRS) design [17]. The TMRS is a commercial 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, its document specification and the UTOPIA Level 2 protocol standard.

Xu *et al.* [20] verified a Frame Multiplexer/Demultiplexer (FMD) chip from Nortel Semiconductors using FormalCheck. The FMD chip is part of a system used in multiplexing/demultiplexing framed data between various channels and a SONET line [5]. 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.

3 Multiway Decision Graphs

Multiway Decision Graphs (MDG) [9] have been proposed to solve the state space explosion problem of ROBDD (Reduced Ordered Binary Decision Diagram) [7] based verification. While accommodating a higher level of abstraction as with theorem proving, the MDG tools offer automation in the verification process like ROBDD based tools. The formal logic underlying MDGs is a many-sorted first-order logic, augmented with a distinction between concrete and abstract sorts. This is motivated by the traditional division of datapath and control circuitry of RTL (Register Transfer Level) design. A concrete sort has an enumeration while an abstract sort does not. A small example of a multiplexer is given in Figure 1, illustrating its ROBDD, concrete and abstract MDG encodings.

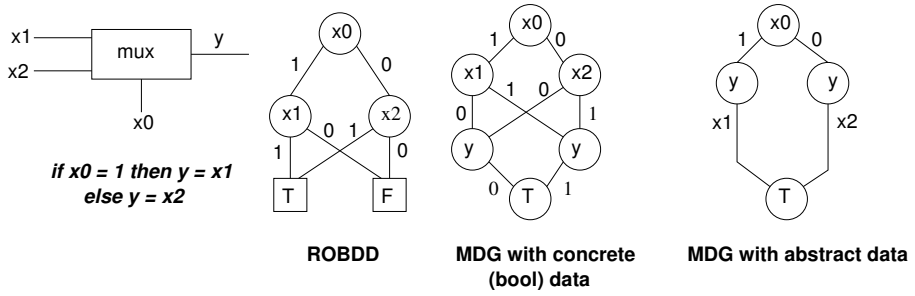

Figure 1: ROBDD and MDGs for a Multiplexer

An MDG is a finite, directed acyclic graph. 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 variable of abstract sort and its edges are labeled with abstract terms of the same sort; or it can be a cross-term (whose function symbol is a cross-operator). For example, given x and y two variables of the same abstract sort, we can define, *leq (x,y)*, for *less-or-equal* as an uninterpreted cross-operator which supports a Boolean value. Variables of concrete and abstract sort can be used to model control and data signals, respectively, while uninterpreted function symbols and cross-operators denote data operations and feedback from the datapath to the control circuitry,

respectively [9]. 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 [9]. An MDG may have only one leaf node denoted as **T**, which means all paths in an MDG are true formulae. Thus, MDGs essentially represent relations rather than functions in a canonical form. MDGs can also represent sets of states. Like ROBDDs, MDGs must be reduced and ordered.

Based on MDGs, abstract descriptions of state machines, called Abstract State Machines (*ASM*)[1] are used to model the system. An ASM is obtained by letting some data inputs, states or output variables be of abstract sort, and the datapath operations are uninterpreted function symbols. They admit non-finite state machines as models in addition to their intended finite interpretations. The MDG tools [22] provide algorithms for equivalence checking, invariant checking and model checking, which are based on the reachability analysis of all states. The equivalence verification procedures are combinational and sequential verification. The model checking supports a first-order temporal logic, called $\mathcal{L}_{\mathcal{MDG}}$ [19], which is an extension of universally quantified branching time first-order temporal logic. The MDG tools run on a Prolog platform and accepts a Prolog-style HDL (Hardware Description Language) as its input language, called MDG-HDL [22], which allows the use of abstract variables for representing data signals. MDG-HDL supports structural descriptions, behavioral descriptions, or a mixture of both. 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, etc.). A behavioral description can be given by high-level constructs such as ITE (If-Then-Else) or CASE statements, which 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.

The MDG tools have some significant practical limitations: For instance, due to the non-interpretation of data operators, the reachability analysis of abstract states may not terminate. If this situation occurs, a dedicated heuristic has to be used from a set of algorithms developed in [1] and [21]. Another practical drawback of the MDG tools with respect to an industrial setting is that they do not accept VHDL or Verilog HDL as input language. However a project is currently underway to translate Verilog to MDG-HDL which will be available in the near future. This paper is to advocate MDG, where we will hence use small examples from the RASE TSB model to illustrate the MDG modeling of behavior and structure.

[1] The notion of *Abstract State Machines* (ASM) has been defined in the MDG literature [9], which by coincidence matches the same naming as the widely known ASM introduced by Gurevich [10].

4 The RASE Telecom System Block

The design under investigation is the RASE Telecom System Block (TSB). It processes a portion of the SONET (Synchronous Optical Network) [5] line overhead of a received SONET data stream. The RASE TSB consists of three types of components: Transport overhead extraction and manipulation, Bit Error Rate Monitoring (BERM) and Interrupt Server (see Figure 2) [16]. In addition to these blocks, it has an interface to a Common Bus Interface (CBI) block which is used mainly for the configuration and testing of the TSB interface and two inputs/outputs multiplexers. The transport overhead extraction and manipulation functions are implemented by three sub-modules (Transport Overhead bytes extractor, Automatic Protection Switch (APS) control and Synchronization Status filtering).

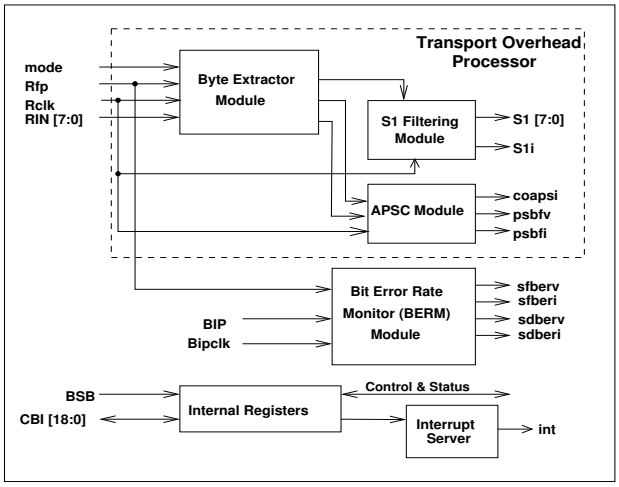

Figure 2: The RASE Telecom System Block

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 3). After extracting the above bytes, it processes them according to some requirements set by the SONET standard. The APS control block filters and captures the received APS channel bytes (K1 and K2), allowing them to be read via the CBI bus. The synchronization status filtering block captures and filters the S1 status bytes, allowing them to be read via the CBI bus. The RASE TSB also performs Bit Error Rate Monitoring using the Bit Interleaved Parity (BIP)-24/8 line of a frame, i.e., B2 bytes (Figure 3). The received line 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 (predefined 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. 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).

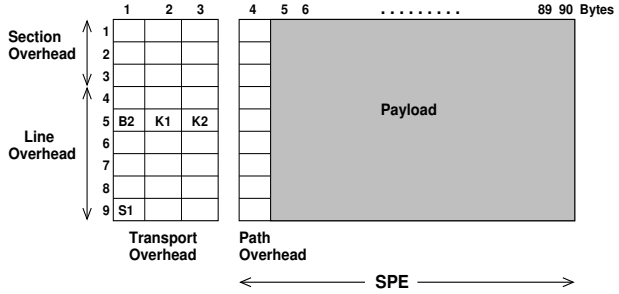

Figure 3: The STS-1 SONET Frame Structure

We propose 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. The detail descriptions of this approach can be found in [23]. Figure 4 displays a tree showing the level of design hierarchy of the RASE TSB. Inspired by [16], we derived a behavioral model of the RASE TSB which consists of five main functional blocks – Transport Overhead Extractor (TOH), Automatic Protection Switch (APS), Synchronization Status Filtering, Bit Error Rate Monitoring (BERM) 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 the RASE telecom system block. For the formal verification, we are interested in the above five control blocks only to eliminate the possibility of state space explosion. During our modeling of the verification, we eliminated those blocks which have no effect on the functionality of the TSB.

4.1 MDG Modeling of the RASE Implementation

In this section, we describe the implementation of the TSB at the Register Transfer Level (RTL). For the MDG-based verification, we translated the original VHDL models into very similar models using MDG-HDL. 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 variables can be viewed as a single abstract variable. Thus a 24-bit frame-counter can be modeled as a variable of abstract sort, say *worda24*, instead of a concrete sort with enumeration 0, 1, . . ., 16777215. Another advantage in using MDG 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

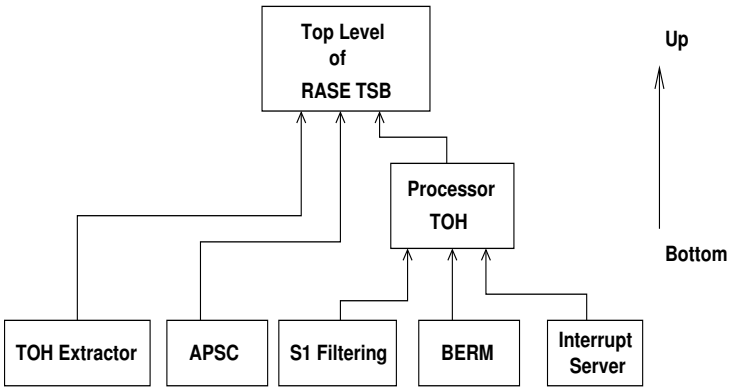

Figure 4: The RASE TSB Design Hierarchy

function, cross-operators are useful for modeling feedback from the datapath to the control circuitry. To handle the complexity of the design, we hence adopted a module abstraction technique for the RTL model in MDG. This idea is illustrated using an example (see Figure 5) from our case study.

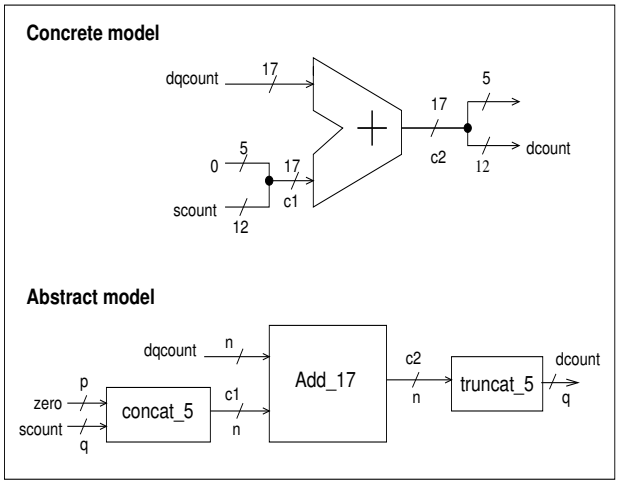

Figure 5: An Example of Module Abstraction

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 applied to the operands (e.g. *A*dd_17,

*C*oncat_5). Similarly, we can define feedback signals using uninterpreted cross-operators, e.g. a function *geq (dcount,declare_th)*, whitch models a *greater-or-equal* test for the counter *dcount* against a threshold value, *declare_th* and outputs a Boolean result.

4.2 MDG Modeling of the RASE Behavior

Based on the product documentation [16] provided to us, we derived a high-level MDG behavioral model of the RASE TSB using Abstract State Machines (ASMs). To illustrate the behavioral modeling approach adopted for the RASE TSB, we are presenting here only the BIP counting abstract state machine and its MGD-HDL model in pseudo-code (see Figure 6). The BIP line counter has three possible states—S0, S1 and S2. The state variable *Bcount* of abstract sort *worda12*, stores the current count value of the BIP line. The symbol *n_Bcount* in Figure 6 represents the next state value of the BIP line counter. 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 *worda12*. After initialization, if the input *bip* = "1" then the next state will be S1, where the counter is incremented until it reaches the threshold. The uninterpreted function *inc_12* denotes the increment-by-one operation of abstract words. 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 nature of some variables and function symbols. The reachability analysis algorithm of the MDG tools is based on abstract implicit state enumeration [9]. Due to the *non-termination* of abstract state enumeration all states may not be reached [1]. To illustrate this limitation of MDG-based verification, we can look at the example of Figure 6, where a generic constant *zero* of the abstract sort denotes the initial value of *Bcount*. The MDG representing the set of reachable states of the BIP counting ASM would contain states of the form (*Bcount*, *inc_12*(. . . *inc_12*(*zero*). . .)) for a 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, since the structure of the MDG will become arbitrarily large. This typical form of non-termination can be avoided by using some heuristic techniques described, e.g., in [1] and [21]. One such method is based on the *generalization* of initial state that causes divergence, like the variable *Bcount* in Figure 6. Rather than starting the reachability analysis with an abstract constant *zero* as the initial value of *Bcount*, a *fresh*[2] abstract variable (e.g., *C*) is assigned to *Bcount* at the beginning of the analysis. Because of this *fresh* variable, the initial set of states represented by *Bcount* thus represents any state, hence any increment of *Bcount* leads the ASM to a state where the new value of *Bcount* is an instance of its arbitrary

[2] A *fresh* variable is disjoint from all other variables.

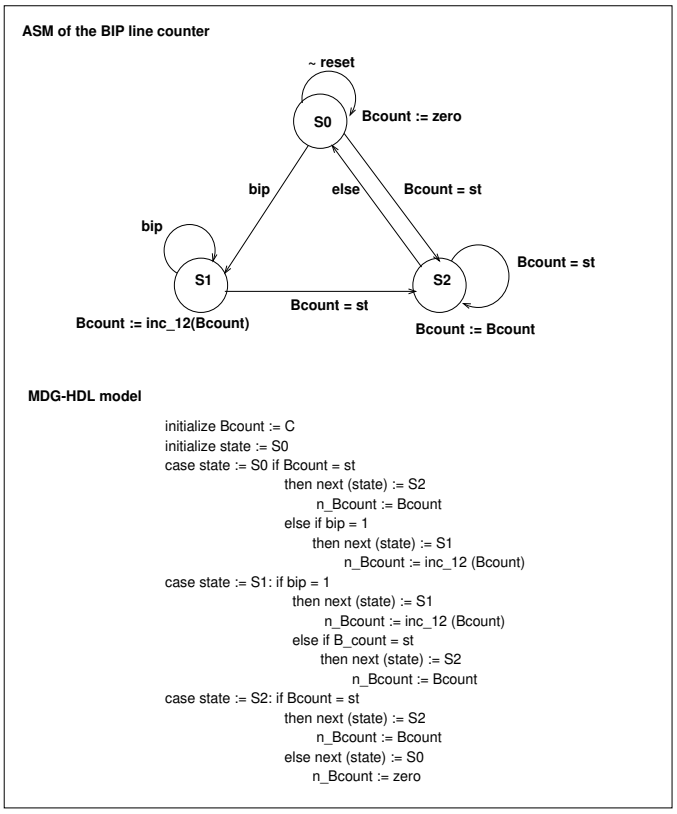

Figure 6: Example of an ASM with its MDG-HDL Model

value of the initial state.

5 Verification of the RASE TSB

The functional simulation for the TSB was performed using a VHDL test bench. Two different types of data structure have been used. These data structures are implemented for two different functions: the SONET frame data information and the BIP line data. In contrast to simulation, formal verification has a built-in technique that allows a non-deterministic choice of values for the primary inputs. This non-determinism will eliminate the use of (explicit) deterministic data structure as used for simulation. We have used a non-deterministic environment to perform the MDG model checking and equivalence checking of the TSB. The *non-deterministic* environment refers to that fact that no constraints are applied on the primary input of the systems. But in the case of FormalCheck, we required to constraint some of the primary inputs of the system, which was essential for terminating some properties verification.

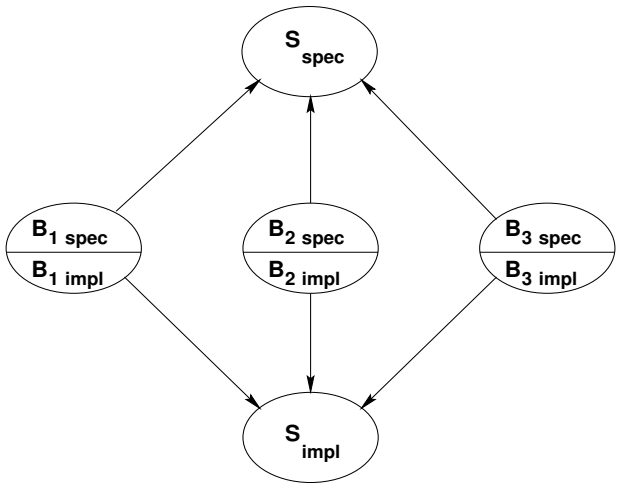

Figure 7: Hierarchical Proof Methodology for Equivalence Checking

5.1 Hierarchical Verification Approach

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 system, S, having three sub-modules, named *B1*, *B2* and *B3*, which may or may not be interconnected between them by control signals Figure 7. In the verification phases, first we proved that the implementation of each sub-module (i.e., $Bi[impl]$, where i = 1,.., 3) is equivalent to its specification, (i.e. $Bi[spec]$, where i = 1,.., 3), which can be done automatically within the MDG system. Then we derive a specification for the whole system, $S[spec]$, as a conjunction of the specification of each sub-module. Similarly, we also derive an implementation of the whole system, $S[impl]$, as a conjunction of the implementation of each sub-module. 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. This is done naturally using our MDG-HDL, which is Prolog based, and hence provides a predicate description of the design components. Finally, we deduce that the specification of the whole system is equivalent to the top level implementation of the system. It is to be noted that we must also ensure that the specification itself is correct with respect to its desired behavior. This is done by model checking a set of properties.

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. 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 8,

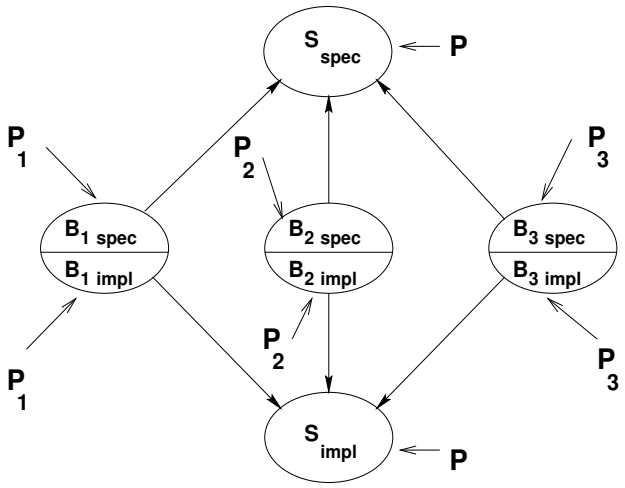

Figure 8: Hierarchical Proof Methodology for Model Checking

where properties *P1*, *P2* and *P3* are the features for the blocks *B1*, *B2*, and *B3*, 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 MDG equivalence checking and model checking, respectively. More details of the overall verification approach can be found in [23].

5.2 Model Checking of the RASE TSB

We first 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 verification of the properties has been carried out using the model checking facility of MDG [19].

The properties are described using a property specification language called $\mathcal{L}_{\mathcal{MDG}}$ [19], which is a universally quantified first-order temporal logic. We have verified a set of 12 properties. For illustration purposes, in the following, we present three sample properties on the RASE TSB, Property 1, Property 7 and Property 12. A complete description of a longer list of properties can be found in Appendix A. In the properties description, the temporal operators, AG, X, and F mean *always*, in the *next* state, and *eventually* in the future, respectively. The symbols "!", "&", "|", and "⇒" mean logical *not*, *and*, *or* and *implication*, respectively.

Property 1: According to the specification of SONET Transport System [5], the filtered S1 byte of a SONET frame needs to be identical for seven con-

secutive frames. If seven consecutive frames do not contain identical S1 bytes, an interrupt is generated to indicate that the filtered S1 value has changed.

```
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)))
```

where *state_ssd* = 6 means that no seven consecutive frames contain identical S1 bytes. *s1_in* is the synchronization status byte of input SONET data stream, and *s1_last_reg* and *s1_filter_reg* are the previous and filtered values of the S1 bytes, respectively. *s1i* is the interrupt signal which should not go high if the next frame does not contain identical S1 bytes.

Property 7: When the calculated Bit Error Rate (BER) counters (*dcount* and *ccount*) exceed their programmable alarm declaration threshold value (*declare_th*) and alarm clearing threshold value (*clear_th*), respectively, an alarm (*berv*) will be triggered to indicate that the calculated BER on the SONET input data stream exceeds its programmable threshold value.

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

where *Bipclk* is the BIP line referenced clock and, *dcount* and *ccount* counters count up to 4k depending on the configuration of the threshold values. These counters are modeled with abstract data sorts, hence removing the possibility of state space explosion in MDG.

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.

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

where *berv* is the Bit Error Rate declaration threshold value. *beri* is the interrupt signal which will go high if the calculated threshold value exceeds its programmable one.

We applied model checking in a hierarchical fashion on both the block and top levels of the TSB. As there are five blocks in our TSB design, we grouped the properties into five block-level properties and finally merged them into a set of twelve properties featuring the top level verification of the TSB. The experimental results from the verification of a set of 12 properties on both the RASE specification and implementation, are given in Table 1, including the particular module affected, the CPU time in seconds, memory usage in MB and the number of generated MDG nodes. Experimental results in Table 1 and subsequent tables have been carried out on a Sun Ultra Sparc 2 machine with 768 MB of memory.

Table 1: Model Checking Results using MDG

Property	Module	Implementation			Specification		
		CPU Time (sec.)	Memory (MB)	MDG Nodes	CPU Time (sec.)	Memory (MB)	MDG Nodes
Property 1	S1 Filter	82.47	15.60	53139	59.91	13.22	21690
Property 2	APSC	82.62	14.98	52094	71.43	13.70	21333
Property 3	APSC	54.31	17.12	52375	55.39	12.46	20984
Property 4	APSC	78.05	15.24	51354	56.67	12.01	21060
Property 5	TOH Proc.	76.57	15.53	51533	45.59	13.19	51527
Property 6	TOH Ext.	81.65	15.80	52094	53.59	14.50	20837
Property 7	BERM	82.54	15.86	51482	52.76	12.92	21243
Property 8	BERM	64.30	15.72	51410	44.83	14.34	21060
Property 9	RASE	78.06	16.65	51330	56.28	12.37	21036
Property 10	RASE	58.41	16.12	51277	54.06	13.29	51253
Property 11	APSC	81.42	16.22	51530	54.99	12.59	51214
Property 12	BERM	85.72	15.98	51564	87.66	12.46	21178

5.3 Equivalence Checking of the RASE TSB

Once a certain confidence in the correctness of the models has been established, we proceed with the verification of the RASE TSB models using equivalence checking following the bottom-up hierarchical approach described earlier. For instance, 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. Experimental results for the equivalence checking between the behavioral models against their RTL implementations are given in Table 2, including the module verified, CPU time in seconds, memory usage in MB and the number of MDG nodes generated.

The verifications of the first four modules consumed less CPU time and memory, because they are less complex and have less abstract state variables and cross-operators than those of the last three modules (see Table 2). 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 the 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, especially 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

obviously takes more CPU time and memory during the verification process than the other modules.

Table 2: Equivalence Checking Results Using MDG

Module	CPU Time (sec.)	Memory (MB)	MDG Nodes
TOH Extractor	3.88	2.33	2806
APSC	17.37	7.91	9974
S1 Filter	22.22	6.81	14831
Interrupt Server	0.48	0.09	180
BERM	80.53	21.31	35799
TOH Processor	89.03	27.79	60068
RASE TSB	437.15	47.36	135658

6 Comparison with FormalCheck

One of the motivations of this work was to compare the model checking of the RASE TSB using the MDG model checker with existing commercial model checking tools, here Cadence FormalCheck [8]. The performance metrics of the comparison were CPU time, memory usages and state variables. Similar to our MDG verification, we also eliminated here some blocks which do not affect the behavior of the system. Based on the design hierarchy, the model checking of the lower level modules has been done in first place. Finally, we integrated all sub-modules into a top level structural model of the RASE TSB and performed the model checking on it. In our performance comparison between FormalCheck and MDG model checking, we consider only the top level verification. A full specification of the properties in the FormalCheck syntax, as well as the defined constraints and macros are given in Appendix B.

The summary of the comparison between these two verification systems are given in Table 3, where '*' means that the verification did not terminate after a substantial run-time (several days). Some properties verification in FormalCheck (Properties 5, 7, 8, 9, 10 and 12) did not terminate due to state space explosion, even though we used different tool guided reduction and abstraction techniques in FormalCheck. The number of state variables used in FormalCheck is less than MDG, because FormalCheck has a built-in state reduction technique which is not available within the MDG system.

Properties 7, 8 and 12 belong to the BERM module, which is the largest and most complex of the RASE TSB. The verification of Properties 7, 8 and 12 did not terminate as these are dealing with control signals having width of 12 to 24 bits. Moreover, some complex data operations between large sized state variables were involved. For instance, 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. In general, ROBDD-based verification cannot be directly applied to a mixed control-datapath design which has large data words as the size of an ROBDD grows exponentially with the width of the Boolean variables. The verification of the BERM module falls into the above mentioned category which comprises lots of wide state variables.

Our experimental results show that FormalCheck whose underlying structure is automata oriented [12] is more efficient in verifying FSM-based modules, i.e., concrete variables, than the MDG tools. For instance, Table 1 shows that Properties 1, 2, 3, 4, 6 and 11 take less verification time in FormalCheck than in MDG. These properties are related to the APSC, Synchronization status and TOH Extraction modules which are completely FSM-based designs. Properties 5, 9 and 10 did not terminate on the top level using FormalCheck, as these properties are verifying the integrated functionality of several modules.

Table 3: Comparative Model Checking Results using MDG and FormalCheck

Property	**MDG**			**FormalCheck**		
	CPU Time (sec.)	**Memory (MB)**	**State Variables**	**CPU Time (sec.)**	**Memory (MB)**	**State Variables**
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	*	*	*

Human effort to formal verification of any design is an important issue to the industrial community. In FormalCheck, we do not need any intervention for variable ordering while for the MDG tools, we used manual variable ordering since no heuristic ordering algorithm is available in the current version. The translation of the original VHDL design description to MDG-HDL structural model was also time consuming. In contrast to this, no time was spent on the RTL modeling for FormalCheck which accepts the original VHDL structural model without any translation.

7 Conclusions

In this paper, we demonstrated that the MDG tools have the capability to verify a moderate size industrial telecommunication hardware, the RASE TSB. Based on the product documentation provided by PMC-Sierra, Inc., we derived a behavioral model of the Telecom System Block. The specification was given as English text 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.

Although some of the ideas presented in this paper may be well known, this is the first time the MDG tool has been applied to a commercial telecommunication system. Unlike other conventional (simulation) tools, applying MDG to a design verification is not straight forward or well defined. To apply this tool, especially for a design like the RASE TSB, one needs to come up with a methodology on "how to apply" MDG to this kind of large telecommunication hardware. On the other hand, one of the motivations of this work was to compare the verification of the TSB using MDGs with the verification done by a sophisticated formal verification tool, Cadence FormalCheck. The verification in FormalCheck has one major practical advantage over MDG, namely the VHDL/Verilog front-end which enables a seamless integration with the design flow. The MDG-based approach on the other hand can handle arbitrary data widths using abstract sort and uninterpreted functions which is a solution to the state space explosion problem. Finally, our experimental results show that in some cases, FormalCheck fails short due to state space explosion for wider datapath.

The experimental results in this work suggest that a hybrid MDG-Formal Check 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 industrial size designs where the circuit is composed of FSM-based control and datapath oriented modules.

8 Acknowledgments

We would like to thank PMC-Sierra, Inc. for providing us with the documentation and VHDL models of the RASE TSB. We are in particular grateful to Jean Lamarche, whose cooperation contributed to the initiating and maturity of this work. We are also thankful to the reviewers, whose thoughtful and extensive comments significantly improved the quality of this paper.

References

[1] O. Ait Mohamed and X. Song and E. Cerny. On the Non-termination of MDG-based Abstract State Enumeration, Theoretical Computer Science, Vol.300, pp. 161-179, 2003.

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

[3] 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.

[4] L. Barakatain, S. Tahar, J. Lamarche and J.-M. Gendreau. Functional Verification of a SCI-PHY Level 2 Protocol Engine. Proc. IEEE International Conference on Information, Communications & Signal Processing, Singapore, October 2001.

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

[6] 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.

[7] 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.

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

[9] 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.

[10] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In Specification and Validation Methods. Oxford University Press, 1995.

[11] R. Handel, M.N. Huber and S. Schroeder. ATM Networks: Concepts, Protocols, Applications, Harlow and Addison-Wesley, 1998.

[12] 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.

[13] I. Leslie and D. McAuley. Fairisle: A ATM Network for Local Area. ACM Communication Review, Vol. 19, pp. 237-336, September 1991.

[14] K.L. McMillan. Symbolic Model Checking. Norwell, MA, Kluwer, 1993.

[15] Microchip Technology Inc. PIC16C71. pp. 2.328-2.372. 1994.

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

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

[18] 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.

[19] Y. Xu, E. Cerny, X. Song, F. Corella, O. Mohamed. Model Checking for First-Order Temporal Logic using Multiway Decision Graphs. In Computer Aided Verification, LNCS 1427, Springer Verlag, 1998, pp. 219-231.

[20] 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.

[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, LNCS 1166, Springer Verlag, 1996, pp. 233-246.

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

[23] M.H. Zobair. Modeling and Formal Verification of a Telecom System Block using MDGs. M.A.Sc. Thesis, Concordia University, Department of Electrical and Computer Engineering, Montreal, Canada, April 2001. Available online at: http://hvg.ece.concordia.ca/Publications/Thesis/Hasan-Thesis.pdf

A MDG Properties Description

Property 1: According to the specification of SONET Transport System in [5]: 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 seven 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 $\mathcal{L}_{\mathcal{MDG}}$ 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 [5]: The APS bytes, i.e., K1 and K2 bytes, should be identical for three consecutive frames. If there is a change in these APS bytes within three 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 $\mathcal{L}_{\mathcal{MDG}}$ 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 [5]: An alarm for the automatic protection switch failure will be triggered, i.e., $psbfv = 1$, whenever the TSB receives 12 frames in which three consecutive frames do not contain identical K1 or K2 bytes. In $\mathcal{L}_{\mathcal{MDG}}$ 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 $\mathcal{L}_{\mathcal{MDG}}$ 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 $\mathcal{L}_{\mathcal{MDG}}$ 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 $\mathcal{L}_{\mathcal{MDG}}$ 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 an MDG table. A cross-operator *eq_ex* 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 $\mathcal{L}_{\mathcal{MDG}}$ 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. In the following expression, we use expressions $declare\_thm = 1$ and $mclear\_th = 0$ instead of $(declare\_th \leq dcount)$ and $(count \geq clear\_th)$, respectively. 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 $\mathcal{L}_{\mathcal{MDG}}$ 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., becomes 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 $\mathcal{L}_{\mathcal{MDG}}$ 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 other 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 $\mathcal{L}_{\mathcal{MDG}}$ 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 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 $\mathcal{L}_{\mathcal{MDG}}$ 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 $\mathcal{L}_{\mathcal{MDG}}$ 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 $\mathcal{L}_{\mathcal{MDG}}$ this liveness property is expressed as follows:

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

B FormalCheck Properties Description

1. Constraints for Property Checking in FormalCheck:

```
Clock Constraint: Rclk
  Signal: Rclk
  Extract: No
  Default: No
  Start: Low
  1st Duration: 1
  2nd Duration: 1

Clock Constraint: Bipclk
  Signal: Bipclk
  Extract: No
  Default: No
  Start: Low
  1st Duration: 1
  2nd Duration: 1

Reset Constraint: Rstb
  Signal: 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 (Toh_Process_Inst: Sync_Status_Inst :
       Filter: Match_Count = 6) and
       (Toh_Process_Inst:Sync_Status_Inst:Temp1 = TRUE) and
       (Toh_Process_Inst:Sync_Status_Inst:Temp2 = FALSE)
Always: Toh_Process_Inst:S1i = 1
Options: Fulfill Delay: 0   Duration: 1 counts of
         Toh_Process_Inst : Rclk = rising
```

Property 2:

```
Property: Property_2
Type: Always
After: @k_filter and @k_ready and @k_last
```

```
      and Toh_Process_Inst:Apsc_Inst:Filter_K1k2:Match_Count = 1
Always: Toh_Process_Inst:Coapsi = 1
Options: Fulfill Delay: 0   Duration: 1 counts of Rclk = rising
```

Property 3:

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

Property 4:

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

Property 5:

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

Property 6:

```
Property: Property_6
Type: Always
After: Toh_Process_Inst : Toh_Extract_Inst :Column = 0 and
       Toh_Process_Inst : Toh_Extract_Inst :Row = 8 and
Rclk = 1 and Rstb /= 0
Always: S1 = S1_Tsb
Options: Fulfill Delay: 0   Duration: 1 counts of Rclk = rising
```

Property 7:

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

Property 8:

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

Property 9:

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

Property 10:

```
Property: Property_10
Type: Always
After: Rstb = 0
Always: Toh_Process_Inst : Apsc_Inst :Coapsi = 0 and
        Toh_Process_Inst : Apsc_Inst :Psbfi   = 0 and
  Toh_Process_Inst : Apsc_Inst :Psbfv = 0 and
  Toh_Process_Inst : Sync_Status_Inst :S1i = 0 and
  Berm_Inst :Berv = 0 and Berm_Inst :Beri = 0
Unless: Rstb /= 0
Options:(None)
```

Property 11:

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

Property 12:

```
Property: Property_12
Type: Eventually
After: Rstb/=0 and Bipclk=1 and Berm_Inst :Berv=stable
Eventually: Berm_Inst :beri = 0
Options:(None)
```

3. Macros Expressions used in the Properties:

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

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

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

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

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

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