Processing APL Properties to Generate Verification-Ready MDG Model

Kamran Hussain*, Otmane Ait Mohamed* and Sa'ed Abed†
*Electrical and Computer Engineering Department, Concordia University, Canada
Email: {k_hussai,ait}@ece.concordia.ca
† Computer Engineering Department, Hashemite University, Jordan
Email: sabed@hu.edu.jo

***Abstract*— Multiway Decision Graphs (MDGs) are special decision diagrams that subsume Binary Decision Diagrams (BDDs) and extend them by a first-order formulae suitable for model checking of datapath circuits. In this paper we propose a new specification language, Abstract Property Language (APL), for the MDG model-checker. The APL language eradicates the restrictions present in the existing $\mathcal{L}_{MDG}$ specification language and introduces new operators to improve expressiveness. The paper also presents the design of a front-end translator that accepts specification in APL and builds composite MDG model with the specification directly embedded into its MDG-HDL representation. Finally, some experimental results are presented to show the performance of the APL-Tool and the analysis of the generated code executed on benchmark properties.**

I. Introduction

Model checking is an important formal verification technique that starts integrating digital system designs. It aims by exploring the reachable state space of a model to verify that an implementation satisfies a specification. Specifications are expressed in a propositional temporal logic. An algorithm is employed to automatically explore the state-space and ascertain if the specification is verified by the state-transition graph. Model checking is focused mostly on automatic decision procedures for solving the verification problem. In case the verification fails, the user can track with the counter example produced as to why it failed.

Binary Decision Diagram (BDD) [1] is a canonical representation for Boolean functions that addresses this problem by providing an efficient encoding for the state space at the Boolean level. This representation allows model checkers to verify large systems. Still, most model checkers face the state space explosion problems while verifying large systems even using Symbolic Model Checking.

The MDG system [2] is a decision diagram based verification tool, primarily designed for hardware verification. It supports both equivalence checking and model checking. It is based on multiway decision graphs which extend the Reduced-Ordered Binary Decision Diagram (ROBDD [1]) with abstract sorts and uninterpreted function symbols. To represent and reason about designs using abstract sorts and uninterpreted function symbols we need a first-order logic. A many-sorted first-order logic has been used with a distinction between abstract and concrete sorts that mirrors the hardware distinction between datapath and control. MDGs are canonical representations of a certain class of quantifier-free formulaes to represent the transition and output relations of a state of machine, as well as the set of states.

In this paper, we extend the $\mathcal{L}_{MDG}$ by APL property specification language that facilitates the formal representation of desired specification in MDG based model-checking methodology. We also present how the written specifications are processed in MDG based model checking and introduce the design of a front-end translator that accepts specification in APL and builds composite MDG model with the specification directly embedded into its MDG-HDL representation. Finally, we support our methodology and tool by experimental results executed on benchmark properties of the Read-port module of the Look-Aside Interface.

The structure of the rest of this paper is as follows: Section II describes some related work. Section III introduces some preliminaries about Model Checking, MDG and APL language Specification. The main contribution of the paper describing our methodology is presented in Section IV. Section V presents some experimental results. Finally, Section VI concludes the paper and gives some future research directions.

II. Related Work

As related work to ours, we cite the work of Eric Gascard [3]. He presented a process of generating deterministic finite automata from a given PSL SERE expression [4]. The work includes a developed tool that can produce a VHDL description of a monitor checking circuit given a PSL SERE formula. This is achieved by creating a parser for PSL language. The syntax tree produced by the PSL parser is then given to the automata construction tool. The VHDL monitors are directly translated from the expressions. These monitors are later synthesized on FPGA (Field-Programmable Gate Array).

Marc Boulé and Zeljko Zilic described a way to generate hardware assertion checkers to be used in an emulation environment [5]. To use assertions in hardware emulation, they have introduced a checker generator tool, called MBAC, that can transform PSL assertion units into Verilog modules. The transformation includes boolean layer, temporal layer and *assert* directive of the verification layer. The Verilog modules are synthesized on FPGA. The assertion signals are externally monitored by the verification engineer.

Morin-Allory and Borrione showed a unique method of synthesizing monitors from PSL assertions [6]. In their work, primitive monitor blocks are built for each foundation language operator of PSL. Generic monitors are built for operators that can have different number of operands. These monitors are written in synthesizable subset of VHDL. Construction of complex monitors are done by interconnecting primitive monitors. They have proved the correctness of the monitors using PVS theorem prover [7]. To do this, each monitor is converted to its respective finite state machine (FSM) and translated into PVS input formalism.

The major difference between the work presented in this paper and the works presented in the above paragraphs is the target application. Most are presented with application to either simulation or emulation based verification environment in mind. The research closely related to ours is the work presented in [8], in creating $\mathcal{L}_{MDG}$ language and its processing tools, called $\mathcal{L}_{MDG}$-Tools. In this case the targeted application is MDG based model checking. In this paper, we present a new language based on $\mathcal{L}_{MDG}$, with added operators from PSL, and present an improved tool to perform the processing of specifications written in terms of properties. Given a model of a hardware design in MDG-HDL and its respective specification in terms of properties, the outcome of both $\mathcal{L}_{MDG}$-Tools and our proposed tool is a verification-ready composite model. This model is a combination of the original model and the generated monitor(checker) circuit, representing the specification, in MDG-HDL.

III. Background

A. Model Checking

Model checking techniques are attractive verification method because of their high automation. Model checking is historically a decision graph based method. It uses state space algorithms on finite-state models to check if the desired specification is satisfied. Specifications are expressed in a propositional temporal logic. An algorithm is employed to automatically explore the state-space and check if the specification is verified by the state-transition graph. Depending on *success* or *failure*, a model-checking tool gives the answer *yes* or *no* respectively. When the verification fails, a counter example is provided that help the user to trace the source of the error. This counter example is also known as failure trace.

Model checking algorithms have been explored since early eighties and significant results have been published [9]. The introduction of Bryant's (BDD's) [1] piloted a breakthrough in the size of a transition system that can be verified. Since then, a number of researchers have explored BDD-based symbolic technique and have published results [10], [11]. The drawback of this method is that they usually suffer from the state-explosion problem when verifying designs with large datapath.

B. Multiway Decision Graphs

A new class of decision graph, called Multiway Decision Graph (MDG), was proposed as a solution to the state-space explosion problem by Cerny et al. in 1997 [2]. In MDG based model checking approach, data signals are denoted by abstract variables, and data operators are represented by un-interpreted function symbols. As a result, a verification based on abstract-implicit-state-enumeration can be carried out independently of datapath width, substantially lessening the state explosion problem. A model-checking methodology is typically comprised of three major parts: a specification language, a system modeling language and a set of algorithms to perform model-checking.

The MDG model-checking process requires two sets of tools: a back-end model-checker and a front-end property parsing processor. The front-end provides a mechanism to process the desired specification, written in terms of properties. In order to create the verification-ready model, the property formulae are processed to construct additional circuitry that is added to the original model. The result is a composite model (Fig. 1) in MDG-HDL. The composite model is fed to the back-end model-checker, part of MDG-Tools, to formally verify if the specification holds.

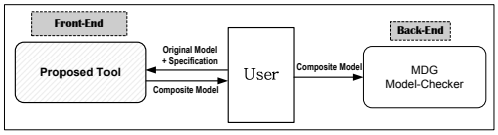

Fig. 1. Proposed front-end of MDG methodology

C. APL language Specification

The APL specification language, designed to be used for MDG model-checking, is based on the CTL* class of temporal logic [8]. It allows properties to be written in first-order temporal logic . The complete specification, both syntax and semantics, of the APL language has been presented by Kamran et al [12]. In APL, all the lexical restrictions that were present in the $\mathcal{L}_{MDG}$ [13] specification language have been lifted. Consequently, users do not need to modify their original model in order to get the verification-ready composite model. While replacing $\mathcal{L}_{MDG}$ with APL, standardized operators were added, and new operators borrowed from PSL [4] were introduced to improve expressiveness.

IV. Generating Verification-Ready MDG Models

The goal of this work is to develop a new front-end as shown in Figure 1, where a single tool will accept the original model with its specifications written in an improved language and will efficiently create the verification-ready composite model. In this section, we present the details of a proposed tool that can process the original model of a design and its specification written in APL and can create a composite model for the MDG model-checking verification. This composite model represents the *design under verification* with some added circuitry [13]. The added circuitry has one boolean output for each property, given as desired specification. These additional circuits are named *monitors* and the boolean outputs are named *flags*.

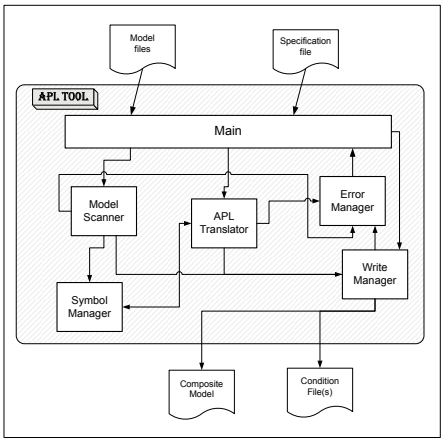

Fig. 2. Structure of APL-Tool.

A. APL Tool Architecture

The tool accepts four files as parameters: three MDG-HDL source files representing the model and a specification file written in APL. It reports syntax and semantic errors found in the specification file. It informs the user whether the construction of the composite model succeeded or not. In case of a failure, it provides detail of the error.
The tool consists of six modules (Fig. 2):

1) Main program: Dispatching tasks to other modules.
2) Model scanner: Scanning the original model files and providing collected data to Symbol Manager.
3) Symbol manager: Managing all symbol data.
4) APL Translator: Parsing the specification file, generating monitors with the help of Symbol manager.
5) Write manager: Writing output files.
6) Error manager: Providing error details.

Model scanner, Symbol manager and APL Translator are collection of sub-modules, performing smaller tasks.

B. APL Translator

The APL Translator (Fig. 3) compiles the specification file and generates, for each property, a monitor circuit with a flag output. Our source language is Abstract Property Language (APL) and the target language is MDG-HDL. It reports syntax and semantic errors to the Error Manager. It sends queries to the Symbol Manager anytime symbol data is required to generate a circuit or to simply check semantics.

The process of generating the monitor circuit from the specification has the following steps:

- Lexical Analyzer tokenizes the specification text based on lexical specification of the *source language*, APL.
- Syntax Analyzer creates annotated abstract syntax tree (AST) based on the grammar of APL [12].
- Simple traversal algorithm is used to handle context and generate the target code at the same time. Whenever it finds a component to be built, it consults the Symbol Manager to check semantics. It reports semantic errors to the Error Manager. If no error exists for the component in concern, it generates MDG-HDL component code and its associated signals. The components are written to the target files by delegating the job to the the Write Manager.

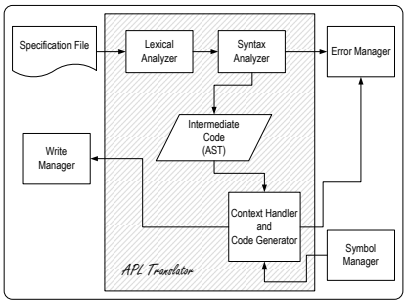

Fig. 3. APL Translator.

The lexical and syntax analyzers for the translator are generated using Flex [14] and Bison [15]. Bison provides a very easy way to construct abstract syntax tree from syntax rules. Whenever a sentence rule is matched, a corresponding C code segment can be executed. We use this feature to build our abstract syntax tree (AST). In order to use traversal algorithms to generate circuits in MDG-HDL, AST's are further annotated to indicate the number of 'next' operators in a given expression and the consequent number of registers required for the to-be-built components in monitor circuit [12].

Context checking is performed every time a function or a compare equation is encountered during code generation. In all cases, semantic mismatches are reported as semantic errors to the Error Manager.

C. Code Generation for Monitor Circuit

Building the AST with MDG-HDL in mind allows us to use simple traversal algorithms to perform both context handling and code generation. Context checking is performed every time a function or a compare equation is encountered during code generation. In all cases, semantic mismatches are reported as semantic errors to the Error Manager.

We traverse the AST in depth-first *post-order* fashion and look for components to build. Every time we encounter such an operator that corresponds to a component in MDG-HDL, we build the circuitry and replace the sub-tree of that node with the output of the component. This is done by storing an attribute called *output string* representing the output of the sub-tree. This string is further modified if registers are needed for that operator. Throughout the process, requests are made to the Write manager to write additional signals and circuits to the composite model files. In our depth-first *post-order* traversal, every time we visit a node, we replace the sub-tree with its output based on an algorithm.

V. Experimental Results

A. Performance Analysis

In this section, we compare the performance of APL-Tool and $\mathcal{L}_{MDG}$-Tools. We analyze the time factors for generating a monitor circuit given a specification. Thus, the design we

TABLE I

PERFORMANCE OF APL-TOOL COMPARED TO $\mathcal{L}_{MDG}$-TOOLS

Number of 'X' operator	APL Execution Time (E_1)	APL Performance (P_1=1/E_1)	APL CPU Usage (%)	Lmdg Execution Time (E_2)	Lmdg Performance (P_2=1/E_2)	Lmdg CPU Usage (%)
1	0.09	11.11	22.0	0.19	5.26	63.0
5	0.10	10.00	24.0	0.21	4.76	88.0
10	0.12	8.33	36.0	0.32	3.13	84.0
50	0.16	6.25	28.5	0.37	2.70	76.4
100	0.17	5.88	52.9	0.39	2.56	81.6
150	0.20	5.00	74.3	0.42	2.38	91.4
200	0.21	4.76	65.6	0.47	2.13	86.7
250	0.24	4.17	77.5	0.48	2.08	85.0
300	0.33	3.03	76.4	0.53	1.89	78.0
350	0.35	2.86	80.9	0.60	1.67	85.5
400	0.41	2.44	63.1	0.73	1.37	76.5
450	0.44	2.27	63.6	0.81	1.23	82.6
500	0.49	2.04	80.9	0.95	1.05	88.5

TABLE II

VERIFICATION RESULTS FOR READ-PORT SPECIFICATIONS

Spec. No.	APL runtime (seconds)	MDG Runtime (seconds)	Memory (MB)	MDG nodes
1	0.11	18.04	23.89	53837
2	0.09	19.46	20.19	54352
3	0.07	14.24	10.46	49638
4	0.12	55.82	92.61	173087
5	0.14	62.30	64.45	178321

choose for this experiment is not critical. For the results given in Table I, we use a design that has no functionality in real world but provides an interface for us to perform our experiments. For our specification, we choose a specification that has six comparators: four concrete and two abstract.

The inspection of CPU usage and execution time indicates that the APL-Tool is less processor intensive but produces the same result in less time. We calculate the average of Execution times. For APL-Tool it is 0.2544615385, and for $\mathcal{L}_{MDG}$-Tools it is 0.497692308. The relative performance on average is:

- Relative Performance = Avg.(E_2) / Avg.(E_1) = 1.9547

Where E_1 and E_2 represent the execution time for APL-Tool and $\mathcal{L}_{MDG}$-Tool, respectively. From this result, we can say that on average the APL-Tool performs about 1.95 times faster than the $\mathcal{L}_{MDG}$-Tools.

B. Bounded Model-Checking of LA-1 Interface

In this section, we verify a part of a data transfer protocol, called Look-Aside Interface (LA1) [16]. This design was originally proposed by the Network Processor Forum. The design was implemented and verified by Li et al. [17] using MDG based model-checking methodology.

We present Read-port specifications of this design, process them using APL-Tool and discuss the experimental results of MDG based model-checking. The tool gives us the composite model with its condition file based on the specification. We then proceed to run the MDG model-checker to verify the specification. Table II gives runtime, memory usage and the number of MDG nodes used for verifying each specification. It also gives time (in seconds) it takes for APL-Tool to generate the composite models.

VI. CONCLUSIONS

We have presented the specification of the new language, called APL. We have eliminated all the lexical restrictions that were present in the $\mathcal{L}_{MDG}$ specification language. While replacing $\mathcal{L}_{MDG}$ with APL, we have added standardized operators and introduced new operators borrowed from PSL to improve expressiveness. We have presented how we generated the verification-ready composite models and their condition files, conforming to the requirements of the MDG model-checking algorithms. We have implemented our APL-tool as a single executable, capable of handling first-order temporal logic. As a result, there is no need for employing multiple tools to process the specification (properties). The obtained performance is promising such that the tool produces the result in almost half the time compared to the existing tools. Future work includes design of a single user-interface representing both front-end and the back-end of the MDG model-checking process and develop a translator that can process a design represented in SystemVerilog or in other traditional HDL's.

REFERENCES

[1] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, 1986.

[2] C. E., C. F., L. M., S. X., T. S., and Z. Z., *Automated Verification with Abstract State Machines using Multiway Decision Graphs*, ser. Formal Hardware Verification: Methods and Systems in Comparison. Springer Verlag, 1997.

[3] E. Gascard, "From sequential extended regular expressions to determinstic finite automata," in *ITI 3rd International Conference on Information and Communications Technology*, 2005, pp. 145–157.

[4] "IEEE Std 1850: IEEE Standard for Property Specification Language (PSL)," 2005.

[5] M. Boule and Z. Zilic, "Incorporating efficient assertion checkers into hardware emulation," in *Computer Design: VLSI in Computers and Processors.*, 2005, pp. 221–228.

[6] K. Morin-Allory and D. Borrione, "Proven correct monitors from psl specifications," in *DATE '06. Proceedings on Design, Automation and Test in Europe*, vol. 1, 2006, pp. 1–6.

[7] O. S., R. J.M., and S. N., "Pvs: a prototype verification system," in *International Conference on Automated Deduction*, 1992, pp. 748–752.

[8] X. Y., "Model Checking for a first-order temporal logic using multiway decision graphs," Phd. Thesis, University of Montreal, Quebec, Canada, 1999.

[9] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Language and Systems*, vol. 8, no. 2, pp. 244–263, 1986.

[10] O. Coudert, C. Berthet, and J. C. Madre, "Verification of synchronous sequential machines based on symbolic execution," in *Proceedings of the international workshop on Automatic verification methods for finite state systems*. New York, NY, USA: Springer-Verlag New York, Inc, 1990, pp. 365–373.

[11] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill, "Sequential circuit verification using symbolic model checking," in *27th Proceedings on Design Automation.*, 1990, pp. 46–51.

[12] H. K., "Abstract property language for mdg model-checking methodology," Masters Thesis, Concordia University, Montreal, Quebec, Canada, 2007.

[13] Y. Xu, E. Cerny, X. Song, F. Corella, and O. Ait-Mohamed, "Model checking for a first-order temporal logic using multiway decision graphs," in *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*. London, UK: Springer-Verlag, 1998, pp. 219–231.

[14] GNU, "Flex: The fast lexical analyzer," http://flex.sourceforge.net/.

[15] Bison, "Bison: Gnu parser generator," http://www.gnu.org/software/bison/.

[16] N. P. Forum, *Look-Aside (LA-1) Interface, Implementation Agreement, Revision 1.1*. Kluwer Academic Publishers, 2004.

[17] D. Li and O. Ait-Mohamed, "Mdg-based verification of the look-aside interface," in *Canadian Conference on Electrical and Computer Engineering*, 2006, pp. 1064–1068.