

Automatic Generation of Model Checking Properties and Constraints from Production Based Specification

M. S. Jahanpour
Cortina Systems, Inc.
sadeqh@cortina-systems.com

O. Ait Mohamed
Dept. of Electrical and Computer
Engineering, Concordia University
ait@ece.concordia.ca

ABSTRACT We propose a technique for automatic generation of environment constraints and component properties used in the constrained model checking (CMC). Beginning from a production based specification of the interface of two components, the tool generates a set of constraints for the interface signals. Industrial model checkers, like Formal Check can consequently use these constraints and properties to model check the components at each side of the interface separately. In this way, the methodology allows a compositional verification.

I. INTRODUCTION

Modeling environment behavior and developing correct properties from complex specifications are time-consuming activities in hardware verification. In conventional simulation, the environment model is often called test bench or bus functional model.

The environment model (or environment, for short) ensures that only acceptable sequence of values is applied to the design under verification. Environment is necessary in model checking as well. After developing environment, model checking is performed by verifying that the design holds its properties when it operates under the environment. This is known as assume-guarantee reasoning [1]. To ease the environment development, various researches have proposed to use constraints to represent the environment assumptions [6][13]. Nevertheless, hundreds of constraints may be necessary to model environment of a commercial design. In this paper, we present an algorithm to automate the generation of environment constraints from a high level language based on the Production Based Specifications (PBS) [8]. PBS language uses regular expressions and has originally been proposed for high-level hardware synthesis. We describe a linear-time procedure to compile PBS specification into a circuit, from which we (unlike the previous works) generate environment constraints. This is because the Model checking tools available as of today cannot handle large designs, we can use the interface monitors in formal verification to model check the components at each side of the interface separately. For this compositional verification to work, we have to generate a suitable environment model (or environment constraints) from the interface monitors to be able to properly constrain the individual components during the verification.

Related Work. Shimizu et al [10] proposed a BDD-based method to generate input stimulus from a set of given constraints. The constraints were developed manually from an informal specification. Oliveira et al [7] as well as CWL group [4] and Beer et al [3] used regular-expression-based languages to specify and synthesize monitors for relatively complex interface protocols. Our algorithm bridges the gap between these two lines of research by generating constraints from the monitors. We synthesize constraints from specifications written in regular expressions. These constraints can be used in simulation or in formal verification. The rest of the paper is organized as follows. Section 2

describes our constraint generation algorithm. Section 3 represents a case study. The conclusions are given in Section 4.

II. CONSTRAINT GENERATION FROM REGULAR EXPRESSION

PBS is a language based on regular expressions and its semantics has been proposed in [8]. Given a non-empty set of symbols, a string is a sequence of symbols constructed from the symbols of the set using three operators; (1) Concatenation operator “.” to join the words, (2) Kleene star operator “*” (“+”) to indicate 0 (1) or many repetitions of the symbols, and (3) choice operator “|” to choose between symbols. Our translation algorithm is similar to the algorithms proposed in the literature to convert regular expressions into data-flow circuits in a polynomial time[6][7]. After synthesizing the circuit, however, we generate a set of properties and constraints that can be used in model checking. We also modify the circuit initialization algorithm to avoid generation of conflicting constraints. Our translation begins by parsing the PBS specification and generating a single parse tree. Nodes of the parse tree contain regular expression operators while leaves of the tree contain Boolean formulas over interface signals (and protocol internal variables). A recursive function *BuildVerilog* constructs a circuit directly from the tree nodes¹. Each call of *BuildVerilog*(*n*, *PredCell*_{*i*}) over a tree node *n* constructs a sub-circuit and connects it to the predecessor circuit *PredCell*_{*i*} which is passed as an argument to this function. The function then returns the constructed circuit. The circuit is a sequence of cells where each cell has an activate_in input *a*_{in}, an activate_out output *a*_{out}, an action enable output *act_enb*_{out} and a proposition *p_i* (see Fig.1). The activate_in signal *a_n* indicates that this cell is enabled and is thus trying to check a Boolean function (i.e., the expression in the node) over the interface signals. If the cell agrees with the current values on these signals, *act_enb*_{out} becomes “1” to indicate that the action clauses attached to this node can be executed. Proposition *p_i* monitors the interface signals. When false, it warns that the cell is enabled and does not agree with the current values on the interface signals. Each call of *BuildVerilog*(*n*, *PredCell*_{*i*}) over a tree node *n* constructs a circuit and returns the last constructed cell as follows.

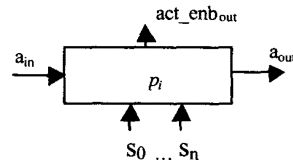


Fig. 1 Interface signals and internal variables

¹ The circuit is constructed in linear time with respect to the tree size.

Base case. If n is a leaf with a Boolean function b_i , then the function creates a new cell with $a_{in} := PredCell_i.a_{out}$, $act_enb_{out} = b_i \wedge a_{in}$, $a_{out} := b_i \wedge a_{in}$ delayed for one cycle, and, $p_i := (a_{in} \Rightarrow b_i)$, where $\Rightarrow$ represents the logical implication. While a_{out} introduces a register in the cell, both act_enb_{out} and p_i generate combinational circuits. *BuildVerilog* returns the newly created cell.

Choice operator. If n is a choice node with left node L and right node R , then the function calls *BuildVerilog*(L , *PredCell*₁) and *BuildVerilog*(R , *PredCell*₁) which return $cell_1$ and $cell_2$, respectively. Then, it creates a third cell *orCell* which contains an OR gate with output $orCell.a_{out} = cell_1.a_{out} \vee cell_2.a_{out}$, and returns *orCell* at the end. The choice operator creates branches in the circuit.

Concatenation operator. If n is a concatenation node with left node L and right node R , then the function calls $cell_1 = BuildVerilog(L, PredCell_1)$ and $cell_2 = BuildVerilog(R, cell_1)$. This way, the circuit generated by R follows the circuit generated from L . The function then return $cell_2$.

Kleene star. If n is a star node with a child node L , the function first creates an *orCell* with inputs $PredCell_i.a_{out}$ and $input_i$, where $input_i$ is a newly declared variable². Then, it calls $cell_1 = BuildVerilog(L, orCell)$ over node L . Upon returning $cell_1$, it assigns $input_i := cell_1.a_{out}$. Star nodes introduce feed back paths in the generated circuit. The function then returns *orCell*.

Initial Cell: To initialize the circuit, we generate a register (C_0) that has an output a_0 which becomes "1" for only one cycle. The circuit is completely synthesized by calling *BuildVerilog*(n_0, C_0) where n_0 denotes the syntax tree root node.

After generating the circuit by *BuildVerilog* program, the tool modifies propositions of the cells that have formed a branch (or fork) in the circuit. Propositions of cells at the branches must be disjuncted to remove conflicting propositions.

The general architecture of our tool is shown in Fig. 2. Firstly, the tool compiles PBS specification into monitor circuits. Secondly, it generates proposition and makes disjunctions. At this stage, we have used a BDD library to carry on disjunctions and Boolean simplifications. Finally, it splits the propositions into environment constraints and design properties based on each component input and output signal names.

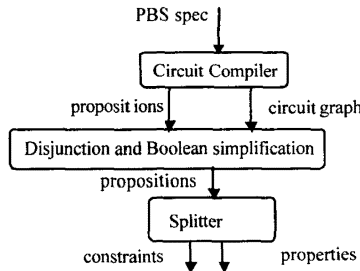


Fig. 2. The tool flow for the constraint generator

² In verilog, we declare a wire for this variable.

Example 1 Let M_1 and M_2 be two modules that are connected using Boolean signals *req* and *ack* (Fig. 3). M_1 asserts *req* and then M_2 asserts *ack*. Suppose that M_1 and M_2 interface specification is given by the regular expression: $Interface_spec \rightarrow (req, ack, \sim req \wedge \sim ack)$, which specifies that *req* is asserted in the first cycle, *ack* is asserted in the second cycle, and both *req* and *ack* are de-asserted in the third cycle.

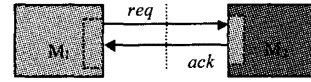


Fig. 3. Two modules M_1 and M_2

The circuit obtained by our synthesis algorithm is given in Fig. 4 where each block (except the *init* block) has an *activate_in* input a_i , an *activate_out* output a_{i+1} . The blocks also receive all interface signals, e.g., *req* and *ack*. The *init* block output a_0 is asserted in the initial state for one cycle to activate the monitoring process (as described below). Suppose that $d_{out} := Delay(d_{in})$ denotes a register with input d_{in} and output d_{out} . Activate_out signals a_1, a_2 , and a_3 are given in (2)-(4).

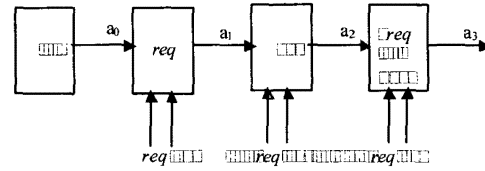


Fig. 4. A monitor I for M_1 - M_2 interface specification

$$a_1 := Delay(a_0 \wedge req) \quad (2)$$

$$a_2 := Delay(a_1 \wedge ack) \quad (3)$$

$$a_3 := Delay(a_2 \wedge \sim req \wedge \sim ack) \quad (4)$$

Suppose that at cycle 0 (i.e., at the initial state), we have $a_0 = 1$ by the *init* block. At this cycle, if the interface signal *req* is 1, then, the *req* block will assert a_1 at cycle 1. At the cycle 1, if the interface signal *ack* is 1, then the *ack* block will assert a_2 at cycle 2. Similarly, the $(\sim req \wedge \sim ack)$ block will assert a_3 at cycle 3 if the relation $(\sim req \wedge \sim ack)$ holds on interface signals at cycle 2. With $(a_3 = 1)$ in cycle 3, the monitor recognizes (or accepts) the specification (1). Note that only one block is activated at any cycle in this monitor and that the same block defines what value can happen on interface signals at that cycle.

The environment constraints were automatically generated from the monitors. We then split them into environment constraints and design properties. For the example shown in Fig. 4, we obtain the following propositions.

$$p_1 := (a_0 \Rightarrow req) \quad (5)$$

$$p_2 := (a_1 \Rightarrow ack) \quad (6)$$

$$p_3 := (a_2 \Rightarrow (\sim req \wedge \sim ack)) \quad (7)$$

A proposition here, is an implication that specifies that if a block is activated, (i.e., the proposition antecedent becomes true) then the combinational relation given in the proposition consequent should

hold true on the interface signals. If the block is not activated, then the proposition is trivially true.

p_1 given by (5) asserts that req should be asserted at cycle 0. p_1 is an *output proposition* (or a property) of M_1 since it involves only M_1 output req . p_2 asserts that ack should be asserted at cycle 1. p_2 is an *input proposition* (or environment constraint) for M_1 since it involves only M_1 input ack . p_3 is an *inout proposition* since it involves both inputs and outputs of M_1 . An inout proposition introduces one property and one constraint. One constraint to compute the input values combinationaly from the design output values. And one property to show that the proposition is forcible as a constraint, i.e., there exists a possible valuation for the design inputs and outputs that satisfies the constraint³. Forcibility of propositions can be checked by a CTL (computation tree logic) property [11]. Let $p = p_1 \wedge p_3$ denote the conjunction of all input and inout propositions.

Forcibility of these propositions can be checked by verifying $\exists s_0. M_1. s_0 \models p$ and $M_1 \models AG (p \Rightarrow EX p)$ [6][10]. This means that there exists an initial state that satisfies the constraints and for any state that satisfies them, there exists a next state that does so. Forcibility check for inout propositions can reveal bugs in the design. For instance, if M_1 assigns “1” to req in cycle 2, then forcibility check for p_3 will fail since the requirement ($req = 0$) of p_3 could not be satisfied by M_1 . The forcibility check proves (or disproves) that the current values on the design outputs agree with the values assumed for them in the inout proposition. After checking forcibility, inout propositions can safely be used as constraints.

Suppose that we want to verify that M_1 respects the specification defined by the monitor. We constrain M_1 input using input and inout propositions p_2 and p_3 to check M_1 output against output proposition p_1 . This kind of model checking is known as constrained model checking (CMC) [6]. Practically, model checker assigns 1 and 0 to ack at cycles 1 and 2 to check that req is 1 in cycle 0. We have also verified by forcibility check that req is 0 at cycle 2. From these verifications, once successful, we can conclude that req satisfies interface protocol represented by the monitor. M_2 is verified similarly using p_1 and p_3 as constraints and p_2 and p_3 as forcible-in- M_2 properties⁴.

Example 2 The monitor in Fig. 5 shows another example of a circuit synthesized from the regular expression, $spec \rightarrow (req, req^*, ack)$, using our algorithm. Block propositions are defined by:

$$p_1 := (a_0 \Rightarrow req), p_2 := (a_2 \Rightarrow req), \text{ and } p_3 := (a_2 \Rightarrow ack)$$

p_2 and p_3 are re-written as one proposition as :

$$p_2 := (a_2 \Rightarrow (req \vee ack)).$$

Making disjunction is important since it may simplify the proposition or change its type from inout to input or output or from input or output to inout proposition.

³ We assume Moore machines and as such these machines do not contain combinational relation between their inputs and outputs. The environment constraints, however, may introduce such combinational dependency.

⁴ Note that p_3 -is-forcible is proven using traditional CTL model checking without using p_3 as a constraint.

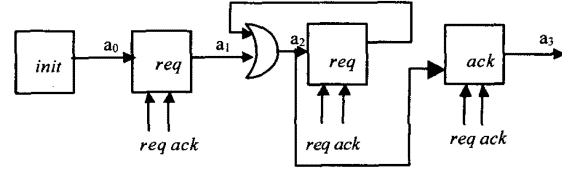


Fig. 5. A monitor for $spec \rightarrow (req, req^*, ack)$

III. CASE STUDY: ATM SWITCH FABRIC

Oliveira et al [7] used PBS language to generate monitors for ARM AMBA protocol. In our case study, we show how to use the same language to specify an ATM switch properties.

A fairisle ATM switch [12] consists of a 4x4 switch fabric that performs the actual switching, and four port controllers that handle cell queueing, prioritizing and transfer to the fabric. (A 2x2 switch fabric without the port controllers is shown in Fig. 6) The in port controllers append two routing tags $h1$ and $h2$ in front of each cell, and send them to the fabric. The complete definition of the switch fabric subsystems and their properties is too large to be included in this paper, but we present cell order preservation property as an example. Consider three arbitrary cells: $cell0 := \langle h1, h2, 1, 2, 3, 4 \rangle$, $cell1 := \langle h1, h2, 5, 6, 7, 8 \rangle$, and $cell2 := \langle h1, h2, 9, 10, 11, 12 \rangle$. We shall verify that if $cell1$ and $cell2$ enter the fabric exactly once, non-deterministically (among a stream of $cell0$), and in order (e.g., $cell1$ enters before $cell2$), then they reach out as such. Using regular expressions, the language given by (8) where $cell0_sent$ indicates that $cell0$ is entirely sent to the fabric should be supplied on I_1 and the similar language should be recognized on I_2 . I_1 reads the fabric input $dIn0$ and the fabric acknowledgment output $ackOut0$ and supplies (environment) constraints over $dIn0$ that should be respected by the model checker while generating values for $dIn0$. Using the timing diagram specification of the fabric in port interface, monitor I_1 is specified as follows.

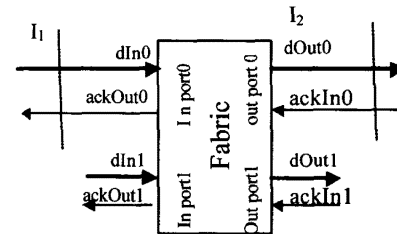


Fig. 6. Fabric surrounded by interface monitors

Monitor $I_1 \rightarrow (cell0_sent)^*, (cell1_sent), (cell0_sent)^*, (cell2_sent), (cell0_sent)^*$; (8)

$cell0_sent \rightarrow (dIn0 = h_1), (dIn0 = h_2), (dIn0 = 1),$

$(dIn0 = 2 \wedge ackOut0 = 1), (dIn0 = 3 \wedge ackOut0 = 1),$
 $(dIn0 = 4 \wedge ackOut0 = 1); \quad (9)$

$cell1_sent \rightarrow (dIn0 = h_1), (dIn0 = h_2), (dIn0 = 5),$
 $(dIn0 = 6 \wedge ackOut0 = 1), (dIn0 = 7 \wedge ackOut0 = 1),$
 $(dIn0 = 8 \wedge ackOut0 = 1); \quad (10)$

$cell2_sent$ is defined similarly. Line (9), for instance, asserts that if positive acknowledgement is received on $ackOut0$ during the cell transmission, entire $cell0$ is forwarded to the fabric. Each line (8)-(10) in this specification is called a production [8]. A production is a name given to a sub-expression so that the expression can be referenced within another production. If all intermediate productions are replaced by their right hand side expressions a single non-recursive regular expression is obtained from PBS specification [8].

The following propositions have been generated from $Monitor_I_1$:

$p_1 := (a_1 \Rightarrow (dIn0 = h_1))$
 $p_2 := (a_2 \Rightarrow (dIn0 = h_2))$
 $p_3 := (a_3 \Rightarrow (dIn0 = 1))$
 $p_4 := (a_4 \Rightarrow ((dIn0 = 2) \wedge ackOut0 = 1)),$
 ...

where $a_1 \dots a_4$ are monitor internal signals. Monitor I_1 has a total of 15 input propositions and 15 inout propositions.

If we force constraints of monitor I_1 to be always *true* during verification, the model checker should only supply the sequence of inputs on $dIn0$ that satisfy I_1 constraints. It assigns h_1 to $dIn0$ in the first cycle to respect constraints p_1 . Then, it assigns h_2 in the second cycle to satisfy p_2 and so on. The net result is that language (8) is supplied to the fabric input. Suppose the cells transmitted from $dIn0$ (i.e., the fabric in port0) are destined to $dOut0$, (i.e., the fabric out port0). Then, Similar to I_1 , we develop a monitor I_2 to recognize the successful transition of cells and acknowledgements over the fabric out port0 signals $dOut0$ and $ackIn0$. I_2 has 15 output propositions and 10 inout propositions. We found a bug in fabric when we were checking for forcibility of I_1 inout proposition $p_4 := (a_4 \Rightarrow ((dIn0 = 2) \wedge ackOut0 = 1))$. Fabric were not correctly transferring acknowledgment from $ackIn0$ to $ackOut0$ and therefore it could not satisfy the condition $ackOut0 = 1$ assumed in p_4 . The bug was fixed and all forcibility checks passed.

After proving that all I_1 and I_2 input and inout propositions are forcible, we verify the fabric implementation against I_2 output propositions. Consider the subsystem $I_1 \parallel fabric \parallel I_2$ shown in Fig. 5. We set all I_1 and I_2 input and inout propositions to be always true during verification to verify that all I_2 output propositions remain always true in the fabric subsystem. This verification proved that if the sequence of $cell0$, $cell1$, and $cell2$ are supplied on fabric input as prescribed by I_1 (i.e., by I_1 input and inout propositions), and if acknowledgment is supplied on fabric acknowledgment input as prescribed by I_2 (i.e., by I_2 input and inout propositions), then the correct sequence of cells are observed on fabric outputs by I_2 (i.e., by I_2 output propositions). The verification run took 39 seconds on a Ultra Sparc workstation and used 16.5 M bytes of memory.

IV. CONCLUSIONS AND FUTURE WORKS

We have presented an algorithm to convert production based specifications (PBS) of an interface into a set of propositions. These propositions are first simplified using BDD and are then automatically partitioned into input, output and inout propositions to be used in the constrained model checking of the components.

Input and inout propositions are used to constrain the component inputs and the output propositions are used to verify the component output conformance to the interface specification.

Current implementation of the tool does not handle pipeline specifications. More experiments have to be carried out to fully understand problems related to automatic generation of propositions for pipeline specifications. And finally, it worth to investigate how one can generate simulation stimulus from the set of propositions generated from the PBS specification.

REFERENCES

- [1] R. Alur, T. A. Henzinger, "Reactive Modules," *proceedings of the 11th IEEE Symposium on Logic in Computer Science (LICS)*, pages 207-218, 1996.
- [2] R. Armoni, L. Fix, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, A. Tiemeyer, E. Singerman, M.Y. Vardi, and Y. Zbar. The ForSpec temporal language: A new temporal property-specification language. In *Proc. 8th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, *Lecture Notes in Computer Science* 2280, pages 296-311. Springer-Verlag, 2002.
- [3] Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. *The temporal logic Sugar*. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. 13 th International Conference on Computer Aided Verification (CAV)*, LNCS 2102. Springer-Verlag, 2001.
- [4] Fujitsu Ltd, Fujitsu Laboratories Ltd., *Component Wrapper Language User's Manual*, Rev. 1.0 Jan. 27 2003 http://www.labs.fujitsu.com/techinfo/cwl/documents/cwl2hdl_1_0e.pdf
- [5] M.S. Jahanpour, E. Cerny, "Compositional Verification of an ATM Switch Using Interface Recognizers/Suppliers (IRS) ", in *IEEE int'l High Level Design Validation and Test Workshop*, pp. 71-76, November 2000, Berkley, California
- [6] M. Kaufmann, A. Martin, and C. Pixley. "Design constraints in Symbolic Model Checking", In A. Hu and M.Vardi, editors, *CAV98: Computer Aided Verification, Lecture Notes in Computer Science*, Springer-Verlag, pages 477-487, 1998.
- [7] M. T. Oliveira, A. J. Hu, "High-Level Specification and Automatic Generation of IP Interface Monitors", *DAC 2002*.
- [8] Seawright, F. Brewer, "Clairvoyant, A Synthesis System for Production-Based Specification", *Proc. IEEE Trans. on VLSI Systems*, vol. 2, no. 2, pp. 172-185, June 1994.
- [9] P. Raymond, recognizing Regular Expressions by means of Dataflows Network, *23rd International Colloquium on Automata, Languages, and Programming, (ICALP'96)*, LNCS 1099, Springer Verlag, Paderborn (Germany), July 1996.
- [10] K. Shimizu and D. L. Dill. "Deriving a Simulation Input Generator and a Coverage Metric From a Formal Specification", In *Proceedings of DAC 2002*, New Orleans, LA, June 2002.
- [11] E. Clarke and E. Emerson, Design and synthesis of synchronization skeletons using branchingtime temporal logic, in *Proceedings of the Workshop on Logic of Programs, Yorktown Heights, D. Kozen, ed., vol. 131 of Lecture Notes in Computer Science*, Springer-Verlag, 1981, pp. 52-7
- [12] P. Curzon, "The Formal Verification of the Fairisle ATM Switching Element," *Technical Report Number 329*, Computer laboratory, University of Cambridge, March 1994.
- [13] J. Yuan, K. Albin, A. Aziz, and C. Pixley. Constraint Synthesis for Environment Modeling in Functional Verification. *ACM Design Automation Conference*. June 2003.