Verifying Compiler Based Refinement of Bluespec^{TM} Specifications using the SPIN Model Checker

Gaurav Singh and Sandeep K. Shukla

{gasingh, shukla}@vt.edu
FERMAT Lab, Deptt. of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA 24061, USA.

Abstract. The underlying model of computation for PROMELA is based on interacting processes with asynchronous communication, and hence SPIN has been mainly used as a verification engine for concurrent software systems. On the other hand, hardware verification has mostly focused on clock synchronous register-transfer level (RTL) models. As a result, verification tools such as SMV which are based on synchronous state machine models have been used more frequently for hardware verification. However, as levels of abstractions are being raised in hardware design and as high-level synthesis is being promoted for synthesizing RTL, hardware design verification problems are changing in nature. In this paper, we consider a specific high-level hardware description langauge, namely, Bluespec System Verilog (BSV). The programming model of BSV is based on concurrent guarded actions, which we also call as Concurrent Action Oriented Specification (CAOS). High-level synthesis from BSV models has been shown to produce efficient RTL designs. Given the industry traction of BSV-based high-level synthesis and associated design flow, we consider the following formal verification problems: (i) Given a BSV specification $\mathcal{S}$ of a hardware design, does it satisfy certain temporal properties? (ii) Given a BSV specification $\mathcal{S}$, and an implementation R synthesized from $\mathcal{S}$ using a BSV-based synthesis tool, does R conform to the behaviors specified by $\mathcal{S}$; that is, is R a refinement of $\mathcal{S}$? (iii) Given a different implementation $R^{'}$ synthesized from $\mathcal{S}$ using some other BSV-based synthesis tool, is $R^{'}$ a refinement of R as well? In this paper, we show how SPIN Model Checker can be used to solve these three problems related to the verification of BSV-based designs. Using a sample design, we illustrate how our approach can be used for verifying whether the designer intent in the BSV specification is accurately matched by its synthesized hardware implementation.

Keywords: Formal Verification, Hardware Designs, Bluespec System Verilog (BSV), SPIN Model Checker.

1 Introduction

The emphasis in PROMELA, which is the input specification language of SPIN, is on the modeling of process synchronization and coordination [1]. For this reason, SPIN is mainly targeted for the verification of concurrent software systems (described in terms of interacting processes) [2], rather than the verification of hardware designs. On the other hand, design of concurrent hardware

systems maps closely to clock-synchronous concurrency model as is primarily implemented by RTL HDLs (Hardware Description Languages) such as Verilog and VHDL. For describing such systems, synchronous state machines are better suited, making the use of tools like SMV [3] more prevalent for hardware verification.

However, for better handling of increasing design complexities and efficient hardware-software co-design, hardware design flows are changing in nature. Behavioral specifications of hardware designs are now being written in algorithmic style (similar to describing a software system) using C-like or other high-level HDLs at levels of abstraction above register-transfer level (RTL). Such high-level specifications can then be automatically converted into the corresponding RTL descriptions [4] using various high-level synthesis tools. Due to this changing trend, there is a need for new hardware verification techniques that are suitable for such high-level design flows. In this work, we show how SPIN, which is primarily a software verification tool, can be used for the verification of hardware designs generated from a particular type of high-level design specifications.

Recently, a new approach to high-level synthesis from Concurrent Action Oriented Specifications (CAOS) [5,6], as embodied in *Bluespec System Verilog (BSV)* [7,8], has been proposed. This is based on the idea that the functionality of any hardware design can be described in terms of high-level concurrent atomic actions. Note that since BSV semantics is based on guarded atomic actions, throughout this paper we use the term *action* instead of *rule* (as used in actual BSV syntax) in order to avoid any confusion between the two terms.

In BSV, each action consists of two parts - a guard and a body. Guard is a condition associated with an action which should evaluate to $True$ for that action to enable. Body of an action consists of a group of operations all of which are executed atomically when the action is enabled. For example, a BSV specification of GCD (Greatest Common Divisor) design can be written in terms of two actions *Swap* and *Diff* (Figure 1). g_1 and g_2 are the guards of actions $Swap$ and $Diff$ respectively (x and y are the registers). The swap of the values in the body of action $Swap$ occurs only when g_1 evaluates to *True*. The assignment $y <= y - x$ in the body of action $Diff$ occurs only when g_2 evaluates to *True*.

$$\textbf{Action Swap} : \mathbf{g_1} \equiv ((\mathbf{x} > \mathbf{y}) \;\&\&\; (\mathbf{y} \neq \mathbf{0}))$$
$$\mathbf{x} <= \mathbf{y}; \quad \mathbf{y} <= \mathbf{x};$$

$$\textbf{Action Diff} : \mathbf{g_2} \equiv ((\mathbf{x} \leq \mathbf{y}) \;\&\&\; (\mathbf{y} \neq \mathbf{0}))$$
$$\mathbf{y} <= \mathbf{y} - \mathbf{x};$$

Fig. 1. BSV Specification for GCD design.

Such a high-level BSV specification of a hardware design can be automatically converted into RTL code. High-level synthesis from BSV has been shown to produce hardware designs which are efficient in terms of area and latency [8]. Techniques for generating power efficient hardware designs from such specifications have also been proposed in the past [5,6,9].

However, the problem of automatic formal verification of hardware designs generated from BSV-based synthesis (which is the main focus of this paper) has not been looked at until now. In this respect, it can be argued that various techniques proposed in the past to formally verify RTL implementations of hardware designs can also be used to verify hardware generated from BSV-based

synthesis. However, as more and more functionality is being added to hardware designs, RTL is fast becoming too low level to efficiently handle the complexity of hardware designs, thus making it imperative to investigate techniques for the verification of high-level hardware descriptions earlier in the design cycle. High-level specifications do not contain details irrelevant to design's behavior, and hence formal verification of such specifications can aid in faster verification and architectural exploration leading to increase in the designer's productivity.

For a given BSV specification $\mathcal{S}$ of a design, its RTL implementation schedules its actions for execution in different clock cycles. A simple hardware schedule randomly selects just one action for execution in each clock cycle. Such a *sequential execution semantics* contains all possible hardware behaviors corresponding to the specification $\mathcal{S}$ but is undesirable from latency (total number of clock cycles elapsed until the execution halts) point of view. Thus, in RTL implementations generated using $\mathcal{S}$, multiple actions are scheduled for execution in a single clock cycle provided the set of behaviors shown by any such implementation is a subset of the set of behaviors of specification $\mathcal{S}$. In this sense, checking the schedule generated by such implementations against the behaviors of the specification $\mathcal{S}$ as well as figuring out relationships among the behaviors shown by two different implementations of a design are important verification issues.

A hardware design can be represented using a corresponding automaton $\mathcal{A}$ which encodes all the behaviors of the design in terms of its different states and transitions among those states. The language of automaton $\mathcal{A}$ contains all such behaviors of the design and can be denoted as $\mathcal{L}(\mathcal{A})$. Also, essential properties of the behaviors of the design can be expressed as a set of LTL (Linear Temporal Logic) formulae EP written in terms of its states and transitions. Using these notations, important verification problems associated with BSV-based design flow can be defined as follows -

1. Given a BSV-based specification $\mathcal{S}$, which corresponds to an automaton $\mathcal{A}_\mathcal{S}$ based on the *sequential execution semantics*, does $\mathcal{A}_\mathcal{S}$ satisfy all the essential properties of EP?
2. Given a BSV-based specification $\mathcal{S}$ and its implementation R (synthesized using a BSV-based high-level synthesis tool), which corresponds to an automaton $\mathcal{A}_\mathcal{R}$, does $\mathcal{A}_\mathcal{R}$ conform to the behaviors of $\mathcal{S}$; that is, does $\mathcal{L}(\mathcal{A}_\mathcal{R}) \subseteq \mathcal{L}(\mathcal{A}_\mathcal{S})$ hold? In other words, is R a refinement of $\mathcal{S}$?
3. Given automata $\mathcal{A}_\mathcal{R}$ and $\mathcal{A}_{\mathcal{R}'}$ for two different implementations R and $R^{'}$ of specification $\mathcal{S}$ (synthesized using two different BSV-based synthesis tools differing in their scheduling of actions) respectively, is $R^{'}$ a refinement of R; that is, does $\mathcal{L}(\mathcal{A}_{\mathcal{R}'}) \subseteq \mathcal{L}(\mathcal{A}_\mathcal{R})$ hold?

In order to solve these three problems efficiently, there is a need for performing automatic formal verification of BSV-based designs at a level of abstraction above RTL. In this paper, we show how SPIN Model Checker can be used for earlier and faster verification of BSV-based designs. The two major contributions of our work are: (1) We present algorithms for converting given action-based BSV specification of a hardware design and its implementations into corresponding process-based PROMELA models for verification of their essential properties

using SPIN. (2) We also present a technique which uses SPIN for proving strong language-containment results between two different models of a BSV-based design. Note that SPIN does not directly support proofs of language-containment between different but related PROMELA models. In this work, we present a technique which generates an LTL specification in the style of TLA (Temporal Logic of Actions) [10] for a given PROMELA model. Such an LTL specification can then be used for proving stronger language-containment results with respect to other related PROMELA models using SPIN's LTL Property Manager.

Thus, we provide a complete automation flow for solving all three verification problems related to BSV-based hardware design flow. As confirmed by experiments, our approach can be successfully used for quick and early verification of hardware designs generated using BSV-based synthesis. Due to space constraints, throughout the paper we use a particular small example of a *Vending Machine Controller* to illustrate the usefulness of our approach.

To the best of our knowledge, no other work has been done in the area of formal verification of BSV-based hardware designs. This is can be attributed to the fact that BSV-based high-level synthesis has been recently proposed [7] and is relatively new. This paper is organized as follows. Section 2 presents formal description of BSV-based high-level synthesis and various scheduling semantics for BSV designs. Various correctness requirements for BSV designs are explained in Section 3. Algorithms for generating PROMELA models containing scheduling information corresponding to a BSV specification and its implementations are presented with sample models in Section 4. Section 5 presents algorithms for verification of essential properties of a BSV design and performing language-containment proofs between BSV models, and also discusses some sample experiments. Section 6 concludes the paper with a short discussion. *Finally, for the sake of proper reviewing, in this version of the paper, we have added an extensive Appendix containing various algorithms and listings of code. If the paper is accepted, in the final version, we will put the contents of the Appendix at a website in addition to some more examples.*

2 Background - High-level Synthesis from BSV

2.1 Hardware Description

Definition 1. *A BSV* ***specification*** $\mathcal{S}$ *of a hardware design consists of a set* $S = \{s_1, s_2, ..., s_k\}$ *of* k *state elements of the design and a set* $A = \{a_1, a_2, ..., a_n\}$ *of* n *actions of the design.*

Definition 2. *Each* ***state element*** $s_i \in S$ *is of the form* (t_i, n_i, in_i)*, where* t_i*,* n_i *and* in_i *represent the data-type, name and initial value of state element* s_i *respectively. The state elements of a BSV design can be in the form of registers, FIFOs or memories.*

Definition 3. *Each* ***action*** $a_i \in A$ *of specification* $\mathcal{S}$ *consists of two parts - a guard and a body. For example, an action* a_i *can be of the form,*

action $m_i()$ *if* $g_i(S_g^i)$ $\{$ $s_{i1} <= b_{i1}(S_b^i);$ $s_{i2} <= b_{i2}(S_b^i);$ $s_{i3} <= b_{i3}(S_b^i);$ $\}$ *end;*

Here, m_i is the name of action $a_i \in A$.
$S_g^i = \{s$: value of $s \in S$ is accessed in the guard of $a_i \in A\}$.
$S_b^i = \{s$: value of $s \in S$ is accessed in the body of $a_i \in A\}$.

Definition 4. *g_i denotes the **guard** of action a_i. It is a condition associated with a_i which evaluates to either $True$ or $False$ depending on current values of elements of S_g^i. Action a_i is said to be **enabled** if g_i evaluates to $True$.*

Definition 5. ***Body** of action a_i consists of set of assignment statements of the form, $s_{ij} <= b_{ij}(S_b^i)$, computing next state values of the design and, in general, it can be expressed as,*

$$B_i = \{(s_{ij}, b_{ij}(S_b^i)) : b_{ij}(S_b^i) \text{ computes the next state value for } s_{ij} \in S_u^i \text{ using current values of the elements of } S_b^i\}$$

where, $S_u^i = \{s$: value of $s \in S$ is updated in the body of $a_i \in A\}$.

Thus, B_i denotes the body of action $a_i \in A$. B_i consists of a group of operations all of which are executed atomically if a_i is enabled and selected for execution.

As an example, Listing 1.1 shows the BSV specification of a *Vending Machine Controller (VMC)*. Most of the description of Listing 1.1 is self-explanatory. For better understanding of BSV semantics and associated verification issues, we will explain our verification approach with respect to this design. The BSV specification of Listing 1.1 is composed of three parts-

Listing 1.1. BSV Specification of Vending Machine Controller.

```
1.  module mkVending()

2.  Reg(Int(7)) count = 0; Reg(Bool) moneyBack = False;
3.  Wire dispenseMoney, gumControl;

4.  // action that controls dispensing of money
5.  action doDispenseMoney() if(moneyBack)
6.    if (count == 0) moneyBack <= False;
7.    else { let newCount = count - 10; count <= newCount;
8.           dispenseMoney.send();
9.           if (newCount == 0) moneyBack <= False; }
10. end;
11. // action that controls dispensing of gum
12. action doDispenseGum() if(!moneyBack && count >= 50)
13.   count <= count - 50; gumControl.send();
14. end;

15. // input-handling methods
16. method tenCentIn() if(!moneyBack) count<=count + 10; end;
17. method fiftyCentIn() if(!moneyBack) count<=count + 50; end;
18. method moneyBackButton() if(!moneyBack) moneyBack<=True; end;

19. // wires connecting money and gum outputs
20. method dispenseTenCents() return dispenseMoney; end;
21. method dispenseGum() return gumControl; end;

22. endmodule;
```

1. **State elements**, *count* and *moneyBack* (Line 2), that correspond to the registers of the design.
2. **Atomic Actions**, *doDispenseMoney* (Line 4-10) and *doDispenseGum* (Line 11-14), that perform computations and update the state elements of the design when their guards (shown with *if* clause) evaluate to *True*. Action *doDispenseMoney* controls the dispensing of the money whereas *doDispenseGum* controls the dispensing of gum.
3. **Interface methods**, *tenCentIn*, *fiftyCentIn*, *moneyBackButton*, *dispenseTenCents* and *dispenseGum* (Lines 15-21), that interact with an external module such as a testbench or other hardware design. Interface methods also behave like atomic actions but their execution is controlled by the external module.

Listing 1.1 also contains some combinational wires (Line 3) which are used for controlling the outputs of VMC. Let $A_m \subseteq A$ be the set of actions corresponding to the interface methods of the BSV design. Thus, for the VMC specification, we have,

$S = \{count, moneyBack\}$.
$A = \{doDispenseMoney, doDispenseGum, tenCentIn, fiftyCentIn, moneyBackButton, dispenseTenCents, dispenseGum\}$.
$A_m = \{tenCentIn, fiftyCentIn, moneyBackButton, dispenseTenCents, dispenseGum\}$.

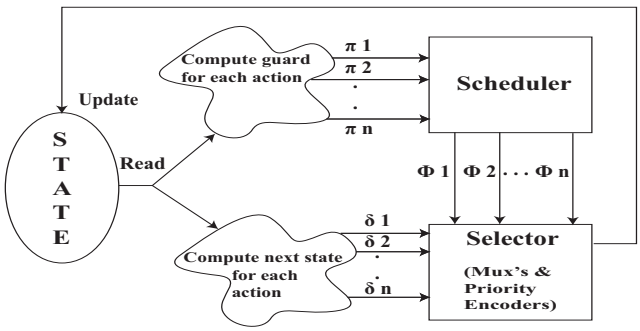

Fig. 2. Synthesis from BSV.

2.2 Synthesis

A given BSV specification $\mathcal{S}$ of a hardware design can be synthesized to generate efficient RTL code, as exemplified by *Bluespec Compiler (BSC)*, which is a commercial high-level synthesis tool based on BSV semantics [7,8]. Figure 2 shows the translation from such BSV-based description of a design to hardware. As shown in Figure 2, hardware synthesis from atomic actions can be achieved by implementing each *guard* and *body* as a combinational logic and synthesizing a control circuitry for appropriate scheduling and data-selection. BSV-based synthesis is behaviorally higher in abstraction than RTL-synthesis because it supports automatic handling of concurrency and synchronization issues (such as due to updates on the shared state elements) via atomic actions.

2.3 Scheduling of Actions

Let $d(s_i)$ denote the domain of state element $s_i \in S$ of the design. For hardware designs, usually $d(s_i)$ is the Boolean domain but for BSV specification, which is at a higher level of abstraction, domains such as 32-bit integer, n-bit bit-vector, etc. can be used. For example, if s_i is an 8-bit register, then $d(s_i) = \{0,1\}^8$, which is a set of all possible 8-bit strings of 0s and 1s.

Lets consider a vector $\hat{s} =< s_1, s_2, ..., s_k >$ of the state elements of the design. Note that $\hat{s}$ contains same elements as S and is used in this section in order to appropriately denote the state of the design for defining its behaviors. Let $\sigma_c(\hat{s}) =< d_1, d_2, ..., d_k > \in \Pi_{i=1}^{k} d(s_i)$ denote the state of the design at the end of clock cycle c. Thus, σ is the function which maps the state elements of the design to their respective values at some point during the design execution. A behavior of the design can be defined as a sequence of states (possibly infinite) given as,

$\beta = (\sigma_0(\hat{s}),\ \sigma_1(\hat{s}),\ \sigma_2(\hat{s}), ...,\ \sigma_c(\hat{s}), ...)$ such that $\sigma_{c+1}(\hat{s})$ results from executing actions in $A_c \subseteq A$ in clock cycle $(c+1)$ when the design is in state $\sigma_c(\hat{s})$.

Refinement of Behaviors - Consider two behaviors β and β' of the design. Let C_β and $C_{\beta'}$ denote the set of clock cycles of β and β' respectively. β' is said to be a refinement of β if $\exists\ r : C_\beta \rightarrow C_{\beta'}$, where r is an injective and monotonic function such that $\forall\ c \in C_\beta,\ \sigma_c^{\beta}(\hat{s}) = \sigma_{r(c)}^{\beta'}(\hat{s})$.

2.3.1 AOA Semantics During the execution of the design generated using BSV-based synthesis, multiple actions can get enabled in a clock cycle c. In a simple hardware schedule, one such enabled action can be randomly chosen for execution in c, thus proceeding the execution of the design in a sequential manner. The execution of the design halts when none of its actions are enabled in some clock cycle. We call such a sequential execution semantics where only one action is randomly chosen for execution in each clock cycle as *Any One Action (AOA) Semantics*.

Behavior in AOA Semantics - Behavior of the design in *AOA Semantics* can be given as,

$\beta^{AOA} = (\sigma_0(\hat{s}),\ \sigma_1(\hat{s}),\ \sigma_2(\hat{s}), ...,\ \sigma_c(\hat{s}), ...)$ such that $\sigma_{c+1}(\hat{s})$ results from executing actions in $A_c \subseteq A$, $|A_c| = 1$, in clock cycle $(c+1)$ when the design is in state $\sigma_c(\hat{s})$.

2.3.2 Concurrent Semantics In spite of being behaviorally correct, the sequential execution of just one action in each clock cycle as per *AOA Semantics* is undesirable from latency point of view, especially for designs containing large number of actions. Thus, in a hardware implementation R, synthesized from specification $\mathcal{S}$, multiple enabled set of actions $A_c \subseteq A$, $|A_c| \geq 1$, can be allowed to execute concurrently in a clock cycle c provided the atomicity of actions in A_c is maintained. This means that, in implementation R, behavior of the design resulting from concurrent execution of actions in A_c should be equivalent to at least one sequential behavior of actions in A_c based on *AOA Semantics*. We

call such a scheduling semantics where multiple actions are allowed to execute concurrently in a single hardware clock cycle as *Concurrent Semantics*.

Conflicting Actions - In hardware generated from BSV-based synthesis, maintaining such atomicity among various actions belonging to A_c may lead to complicated combinational circuit. To avoid this, a notion of **conflict** is introduced. An example of a conflict is two actions updating the same hardware register; that is, two actions $a_i, a_j \in A$ can be said to be conflicting with each other if $S_u^i \cap S_u^j \neq \phi$. Other kinds of conflicts can also exist within two actions of the design, thus forbidding the concurrent execution of those actions. In general, two actions are considered to be conflicting with each other if executing their operations in the same clock cycle is undesirable for pragmatic reasons (like long critical paths, write-write conflicts, complicated hardware analysis, etc.). In the synthesized circuit, such restrictions are enforced using small overhead logic.

For example, for VMC, actions *doDispenseGum*, *tenCentIn* and *fiftyCentIn* conflict with each other since they all update register *count*. In case two or more conflicting actions are enabled in the same clock cycle, a notion of **priority** is used to decide which of those actions should be executed in that cycle. A higher priority action is always chosen for execution over all the other lower priority conflicting actions. Let $C(i, j)$ represent a function which returns *True* if two actions $a_i, a_j \in A$ conflict with each other, and *False* otherwise. Then, $C_i = \{\ a_j : C(i,j) = True; a_j \in A \text{ has higher priority than } a_i \in A\ \}$ denotes the set of actions conflicting with a_i which are preferred for execution over a_i.

Sequential Ordering - In order to generate appropriate scheduling and control logic that maintains the atomicity of various actions of a design executing within the same clock cycle, BSV-based synthesis involves constructing (at compile time) a single sequential ordering S_{order} of all actions belonging to A of specification $\mathcal{S}$. Lets define a relation $<_s$ among any two actions $a_i, a_j \in A$, $C(i,j) = False$, such that $a_i <_s a_j$ holds if concurrent execution of a_i and a_j in a single clock cycle is equivalent to executing a_i followed by a_j in two consecutive clock cycles each executing just one action. For VMC, actions *doDispenseGum* and *moneyBackButton* can be executed concurrently since their concurrent execution is equivalent to the following sequential ordering- *doDispenseGum*, *moneyBackButton*. This can be denoted as: *doDispenseGum* $<_s$ *moneyBackButton*.

To construct S_{order}, transitivity property of relation $<_s$ is used and cycles are broken appropriately during the synthesis process. For VMC, one such possible ordering S_{order} of actions is given as - *tenCentIn*, *fiftyCentIn*, *doDispenseMoney*, *doDispenseGum*, *moneyBackButton*. Note that for simplification, we ignore actions *dispenseTenCents* and *dispenseGum* in this ordering since these actions neither perform any computations nor change the state of the design.

Behavior in Concurrent Semantics - For an implementation R, which is synthesized from specification $\mathcal{S}$ based on *Concurrent Semantics*, a behavior of the design can be defined as,

$\beta^R = (\sigma_0(\hat{s}),\ \sigma_1(\hat{s}),\ \sigma_2(\hat{s}), ...,\ \sigma_c(\hat{s}), ...)$, such that -

(1) $\sigma_{c+1}(\hat{s})$ results from executing actions belonging to $A_c \subseteq A$ in clock cycle $(c+1)$ when the design is in state $\sigma_c(\hat{s})$.

(2) If $|A_c| > 1$, then $C(i,j) = False\ \forall a_i, a_j \in A_c, i \neq j$; that is, A_c denotes a set of non-conflicting actions.

(3) β^R corresponds to an equivalent behavior β^R_{seq} generated using the sequential ordering S_{order} of R, with just one action being executed in each clock cycle in β^R_{seq}.

3 Correctness Requirements for BSV Designs

3.1 AOA Semantics

Depending on what actions are selected for execution in different clock cycles, specification $\mathcal{S}$ of the design can consist of multiple behaviors of the form β^{AOA} based on *AOA Semantics*. Let $\mathcal{A}_\mathcal{S}$ be the automaton encoding all such possible behaviors of specification $\mathcal{S}$. The language of automaton $\mathcal{A}_\mathcal{S}$ is denoted by $\mathcal{L}(\mathcal{A}_\mathcal{S})$, and is said to contain all behaviors of specification $\mathcal{S}$. Let EP represent the set of all essential properties of the design expressed as LTL (Linear Temporal Logic) formulae.

Correctness Requirement 1 (CR-1). The correctness constraint mandates that for $\mathcal{S}$ to be a valid specification of the hardware design, β^{AOA} should satisfy p, $\forall\ \beta^{AOA} \in \mathcal{L}(\mathcal{A}_\mathcal{S})$, $\forall\ p \in EP$.

3.2 Concurrent Semantics

For an implementation R generated from specification $\mathcal{S}$ based on *Concurrent Semantics*, any behavior of the form β^R shown by R corresponds to an equivalent behavior β^R_{seq} generated using S_{order}. Let A_R be an automaton encoding all possible behaviors of the form β^R_{seq} shown by R.

Correctness Requirement 2 (CR-2). The correctness constraint mandates that for R to be a valid implementation of $\mathcal{S}$, R should be a refinement of S. In other words, language-containment relation $\mathcal{L}(\mathcal{A}_\mathcal{R}) \subseteq \mathcal{L}(\mathcal{A}_\mathcal{S})$ should hold.

3.2.1 Maximal Concurrent Schedule (MCS)- For latency minimization, maximal set of actions of the design can be chosen for execution in each hardware clock cycle. Such a schedule of a design can be termed as a *Maximal Concurrent Schedule (MCS)* and an implementation R_{MCS} of the design generated based on this schedule contains multiple different behaviors of the form β^R such that $A_c = A^M_c$, where $A^M_c \subseteq A$ is a maximal set of non-conflicting actions scheduled for execution in clock cycle c. As mentioned earlier, each behavior β^R corresponds to an equivalent behavior β^R_{seq} generated using the sequential ordering S_{order} of R_{MCS}.

Let $\mathcal{A}^{\mathcal{MCS}}_\mathcal{R}$ be the automaton encoding all such possible behaviors $\mathcal{L}(\mathcal{A}^{\mathcal{MCS}}_\mathcal{R})$ of a hardware design under the maximal concurrent schedule. The correctness constraint requires that the language-containment relation $\mathcal{L}(\mathcal{A}^{\mathcal{MCS}}_\mathcal{R}) \subseteq \mathcal{L}(\mathcal{A}_\mathcal{S})$ should hold; that is, R_{MCS} should be a refinement of specification $\mathcal{S}$ of the design. BSC performs automatic concurrent scheduling of hardware designs, and

generates RTL code adhering to one such maximal concurrent refinement which satisfies $\mathcal{L}(\mathcal{A}_{\mathcal{R}}^{\mathcal{MCS}}) \subseteq \mathcal{L}(\mathcal{A}_{\mathcal{S}})$. (BSV is the CAOS-style input specification language of BSC.)

3.2.2 Alternative Concurrent Schedule (ACS)- As mentioned earlier, BSC schedules maximal set of actions in each clock cycle for latency minimization. However, concurrent execution of large number of actions for improving the latency of a hardware design is usually associated with a corresponding degradation of other attributes of the design, such as its area, peak-power, etc. This might not be desirable for a design having conflicting constraints on its latency and other attributes.

In such cases, instead of executing maximal set of actions A_c^M in clock cycle c, an alternative implementation R_{ACS} of the design needs to be derived which selects only a set of actions $A_c \subseteq A$ for execution in c such that all the constraints of the design are satisfied. Such a schedule of a design can be termed as a *Alternative Concurrent Schedule (ACS)*. Let $\mathcal{A}_{\mathcal{R}}^{\mathcal{ACS}}$ be the automaton encoding all possible behaviors $\mathcal{L}(\mathcal{A}_{\mathcal{R}}^{\mathcal{ACS}})$ of a hardware design corresponding to R_{ACS}. Again, the correctness constraint mandates that $\mathcal{L}(\mathcal{A}_{\mathcal{R}}^{\mathcal{ACS}}) \subseteq \mathcal{L}(\mathcal{A}_{\mathcal{S}})$; that is, any alternative implementation R_{ACS} of a design based on such a schedule is required to be a refinement of specification $\mathcal{S}$ of the design.

3.3 Comparing Two Implementations

Two different implementations of a BSV-based design differ in their scheduling of the actions of the design. In general, an implementation R of a BSV specification $\mathcal{S}$ may either enhance or restrict its set of behaviors as compared to some other implementation $R^{'}$. However, as mentioned earlier, all behaviors of R and $R^{'}$ should conform to the set of behaviors $\mathcal{L}(\mathcal{A}_{\mathcal{S}})$ of the specification $\mathcal{S}$, thus satisfying $\mathcal{L}(\mathcal{A}_{\mathcal{R}}) \subseteq \mathcal{L}(\mathcal{A}_{\mathcal{S}})$ and $\mathcal{L}(\mathcal{A}_{\mathcal{R}'}) \subseteq \mathcal{L}(\mathcal{A}_{\mathcal{S}})$.

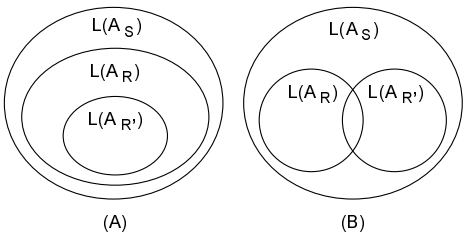

Fig. 3. Language-Containment Relationships.

Furthermore, depending on the design requirements, in some cases it might also be desirable to show that $R^{'}$ is a refinement of R; that is, $\mathcal{L}(\mathcal{A}_{\mathcal{R}'}) \subseteq \mathcal{L}(\mathcal{A}_{\mathcal{R}})$ holds as shown in Figure 3(A). For other cases, relation shown in Figure 3(B) may hold.

Correctness Requirement 3 (CR-3). For two different valid implementations R and $R^{'}$ of specification $\mathcal{S}$, $R^{'}$ is a refinement of R iff $\mathcal{L}(\mathcal{A}_{\mathcal{R}'}) \subseteq \mathcal{L}(\mathcal{A}_{\mathcal{R}})$.

4 Converting BSV Model to PROMELA model

Atomicity as well as priority of operations are important concepts in BSV which are also well supported in PROMELA. Moreover, as demonstrated by our approach, different hardware scheduling semantics can also be efficiently modeled in PROMELA using various constructs of the language [1]. Hence, PROMELA can be used for modeling the desired semantics of BSV designs, thus providing a path for verification of such designs using SPIN at a level of abstraction above RTL.

4.1 Generating PROMELA Variables and Processes

PROMELA model $\mathcal{M}$ of a system consists of a set V of variables and a set P of processes (P includes the 'init' process) used to describe the system. Given a BSV specification $\mathcal{S}$ of a hardware design, we present Algorithm $GenPROMELA$ (Figure 4) to generate PROMELA model $\mathcal{M}$ corresponding to $\mathcal{S}$. Algorithms $GenVARS$, $GenPROCS$ and $GenProcCycle$ which are used by Algorithm $GenPROMELA$ are shown in the Appendix.

ALGORITHM: $GenPROMELA$. **INPUT:** BSV Specification $\mathcal{S}$.
OUTPUT: PROMELA model $\mathcal{M}$.

1. Initialize $V = \phi$, $P = \phi$. (**Note:** V and P are sets of variables and processes of PROMELA model $\mathcal{M}$ respectively.)
2. Using Algorithm $GenVARS$ (Appendix - Figure 7), generate the set of variables V for $\mathcal{M}$ using $\mathcal{S}$.
3. Using Algorithm $GenPROCS$ (Appendix - Figure 8), generate the set of processes P for $\mathcal{M}$ using $\mathcal{S}$ and V. For each action of $\mathcal{S}$, a corresponding process is generated in $\mathcal{M}$.
4. In order to model the hardware behavior in PROMELA, use Algorithm $GenProcCycle$ (Appendix - Figure 9) to generate a process pr (whose execution will denote the start of a hardware cycle) using $\mathcal{S}$, V and P. Add pr to P.
5. Using V and P, generate PROMELA 'init' process which initializes all variables in V and instantiates all processes in P. Add 'init' to P
6. Output PROMELA model $\mathcal{M}$ whose sets of variables and processes are denoted by V and P respectively.

Fig. 4. Algorithm for generating PROMELA model from BSV specification.

4.2 Adding Scheduling Information to PROMELA Model

Algorithm $GenPROMELA$ (Figure 4) generates sets of variables and processes for PROMELA model $\mathcal{M}$ which corresponds to specification $\mathcal{S}$. For modeling a particular hardware execution semantics in PROMELA, a new model $\mathcal{M}_f$ needs to be generated by enhancing model $\mathcal{M}$ with the corresponding hardware scheduling information.

AOA Semantics In order to model a schedule based on *AOA Semantics* in PROMELA, we present Algorithm *AddSeqSched* (Appendix - Figure 10). The algorithm enhances PROMELA model $\mathcal{M}$ generated by Algorithm *GenPROMELA* such that during the execution of the model, 'start_of_cycle' process (whose execution denotes the start of a new hardware clock cycle) is executed after every execution of any other process of the model. The generated model $\mathcal{M}_f$ accurately models the behaviors $\mathcal{L}(\mathcal{A}_\mathcal{S})$ of specification $\mathcal{S}$ as per *AOA Semantics*.

Concurrent Semantics During the BSV-based synthesis, a sequential ordering S_{order} of the actions of the design is generated to which any concurrent execution of actions will correspond. Given such an ordering S_{order} for an implementation R, we present an Algorithm *AddConcSched* (Appendix - Figure 11), which generates model $\mathcal{M}_f$ by enhancing PROMELA model $\mathcal{M}$ (generated by Algorithm $GenPROMELA$ shown in Figure 4) with the scheduling information of implementation R. Note that behaviors of model $\mathcal{M}_f$ correspond to all possible behaviors $\mathcal{L}(\mathcal{A}_\mathcal{R})$ of R.

Algorithm *AddConcSched* is generic in the sense that it can model any particular schedule (MCS as well as ACS) of a design based on *Concurrent Semantics*. For this, it takes ordering S_{order} corresponding to R, and maximum number of actions allowed to execute concurrently in R as inputs. In order to model the hardware behavior, after every execution of 'start_of_cycle' process, Algorithm *AddConcSched* checks each process for execution based on S_{order} until maximum number of processes have executed.

4.3 Sample PROMELA Models

In the Appendix (Listings 1.2 and 1.3), we show generated PROMELA models corresponding to the implementations of VMC specification of Listing 1.1. Listings 1.2 and 1.3 show two PROMELA models - one corresponding to the implementation R_{MCS} which executes maximal set of actions in a single clock cycle, and another corresponding to implementation R_{ACS} based on an alternative schedule which executes only one action as per a sequential ordering.

These models are generated using Algorithm $GenPROMELA$ (Figure 4) and Algorithm *AddConcSched* (Appendix - Figure 11). Both the PROMELA models are shown in Listings 1.2 and 1.3 using appropriate markings for implementation-specific lines of code. These models consist of multiple processes including the PROMELA 'init' process and five other processes corresponding to different actions of the VMC specification. Not all processes could fit in Listing 1.2 so the remaining ones are shown in Listing 1.3. The main characteristics of such a conversion process that translates a given BSV model into corresponding PROMELA model are -

1. Each action of a BSV design with its corresponding guard and operations is modeled as a process in PROMELA model. Moreover, as shown in Listing 1.2, set of variables V (corresponding to the state elements of VMC)

are declared in the beginning of the PROMELA code. Variables *count_old*, *action_fired* and *one_action_fired* are used for verification purposes.
In order to model the atomicity of operations of an action, PROMELA construct 'atomic' is used [1]. This avoids the interleaving of operations of various processes, which is consistent with BSV semantics and aids in faster verification. Also, 'do', which is a repetition construct in *PROMELA* [1], is used for forwarding the execution of processes as in the real hardware (cycle by cycle).

2. In order to model the hardware behavior, an extra process named 'start_of_cycle' is generated as shown in Listing 1.3. This process denotes the start of a hardware clock cycle and reads inputs, if any, from environment external to the design (like a testbench or another hardware design). For VMC, such external inputs are read in variables *tenCentIn*, *fiftyCentIn* and *moneyBackButton*. These variables are used to signal if processes *IFC_tenCentIn*, *IFC_fiftyCentIn* and *IFC_moneyBackButton* which correspond to interface methods of the VMC specification are executed or not.
3. For processes which do not correspond to interface methods, the execution is dependent on a condition which contains logic related to the guard of the corresponding action of the design, as well as logic based on the conflicts with other higher priority actions. If this condition is *True*, then corresponding process is executed, otherwise next process in the ordering is considered for execution (as controlled by variable *action* and PROMELA's 'unless' construct in Listings 1.2 and 1.3).
4. For implementations based on *Concurrent Semantics*, variable named *action* is used to enforce a particular sequential ordering S_{order} of execution among the processes of the generated PROMELA model such that its behavior maps exactly to the concurrent hardware behavior. This is implemented using Algorithm *AddConcSched* (Appendix - Figure 11).

Maximal Concurrent Schedule (MCS) - As shown in Listings 1.2 and 1.3, in order to model an implementation based on MCS, each execution of a process in PROMELA code assigns a new value to variable *action*. The new value is assigned such that in the next step, next process in S_{order} is checked for execution. Such assignments are shown with lines of code marked as "FOR MAXIMAL CONCURRENT SCHEDULE" in Listing 1.2. In such a model, all the processes of the model are checked for execution in every clock cycle.

Alternative Concurrent Schedule (ACS) - For VMC, an implementation corresponding to MCS will execute actions *doDispenseGum* and *moneyBackButton* concurrently whenever *count* becomes greater than 50 cents, *moneyBack* is *False* and external environment signals the execution of action *moneyBackButton*. However, if the peak-power constraint of the design allows only one action to execute in a single clock cycle, then an alternative implementation of the VMC specification adhering to the peak-power constraint needs to be generated. Listings 1.2 and 1.3 also show generated PROMELA model corresponding to such an implementation of VMC. For that implementation, appropriate assignments to variable *action* are shown with lines of code marked

as "FOR ALTERNATIVE CONC SCHEDULE". The value of variable *action* is updated to six in each process, thus signifying the end of a hardware clock cycle after one process executes.

Note: Enforcing a particular sequential ordering (as done in Algorithm *AddConcSched*) suppresses non-determinism in the behavior of the PROMELA model but is needed to model the deterministic hardware behavior. However, note that no such sequential ordering is enforced in the PROMELA model generated by Algorithm *AddSeqSched* (Appendix - Figure 10). In that model, execution of a single process marks the end of a hardware clock cycle, and in the next cycle, a new process is non-deterministically (and not based on a sequential ordering) executed. Thus, such a model will contain all possible behaviors $\mathcal{L}(\mathcal{A}_\mathcal{S})$ of a BSV specification $\mathcal{S}$.

5 Formal Verification using SPIN

5.1 Verifying Correctness Requirement 1 (CR-1)

Proposition 1. *Given a set EP of essential properties, a BSV specification $\mathcal{S}$ satisfies property $p \in EP$ iff its corresponding PROMELA model $\mathcal{M}_f$ satisfies property p_m, where p_m is equivalent to p and is expressed with respect to $\mathcal{M}_f$.*

Based on Proposition 1, a BSV specification $\mathcal{S}$ can be verified for *Correctness Requirement 1 (CR-1)* mentioned in Section 3 using Algorithm $VerfCR1$ (Figure 5).

ALGORITHM: $VerfCR1$.
INPUT: 1. BSV specification $\mathcal{S}$, 2. Set EP of Essential Properties.
OUTPUT: Verify if $\mathcal{S}$ meets *Correctness Requirement 1 (CR-1)*?

---

1. Using Algorithm *GenPROMELA* (Figure 4) and Algorithm *AddSeqSched* (Appendix - Figure 10), generate a PROMELA model $\mathcal{M}_f$ based on *AOA Semantics* using $\mathcal{S}$.
2. Initialize $EP_m = \phi$.
3. For each property $p \in EP$
   (a) Convert p into p_m such that p_m is an LTL formula expressed with respect to $\mathcal{M}_f$ (using variable set V of $\mathcal{M}_f$).
   (b) Add p_m to EP_m.
4. $\forall\ p_m \in EP_m$, perform verification of $\mathcal{M}_f$ against p_m using SPIN.
5. If verification is successful $\forall\ p_m \in EP_m$, then $\mathcal{S}$ meets *Correctness Requirement 1*.

Fig. 5. Algorithm for Verifying Correctness Requirement 1.

ALGORITHM: $VerfLangCont$.
INPUT: 1. BSV specification $\mathcal{S}$, 2. Implementation R of $\mathcal{S}$.
OUTPUT: Verify if $\mathcal{L}(\mathcal{A}_\mathcal{R}) \subseteq \mathcal{L}(\mathcal{A}_\mathcal{S})$ holds or not?

1. Using Algorithm $GenPROMELA$ (Figure 4), generate PROMELA model $\mathcal{M}^\mathcal{R}$ for implementation R. Let V and P denote the set of variables and processes corresponding to $\mathcal{M}^\mathcal{R}$ respectively.
2. Initialize LTL_S = True.
3. For each process $pr \in P$, such that $pr \equiv (m, B, g)$ corresponds to an action $a \in A$,
   (a) Generate a set NSV of all next state values in $\mathcal{M}^\mathcal{R}$ that are possible according to the behaviors of $\mathcal{S}$ ($AOA\ Semantics$) when g becomes $True$ in $\mathcal{M}^\mathcal{R}$. (**Note:** Only next state values in $\mathcal{M}^\mathcal{R}$ corresponding to the execution of an action in $\mathcal{S}$ needs to be captured. For this, in order to signify the execution of a process in $\mathcal{M}^\mathcal{R}$, value of variable $action\_fired$ in V can be checked to be $True$.)
   (b) Generate an LTL property expression LTL_p of the form $LTL_p \equiv ([\,]\ (g -> X(\|\ NSV)))$, where $(\|\ NSV)$ is $True$ only when at least one of the elements of NSV is $True$, and $False$ otherwise.
   (c) $LTL_S = LTL_S$ && LTL_p.
4. Optimize LTL_S to reduce the number of different LTL expressions while retaining all its behaviors. (**Note:** At this stage, LTL_S contains all possible behaviors of $\mathcal{S}$ with respect to the state in $\mathcal{M}^\mathcal{R}$.)
5. Using Algorithm $AddConcSched$ (Appendix - Figure 11), generate PROMELA model $\mathcal{M}_f^\mathcal{R}$ using $\mathcal{S}$, $\mathcal{M}^\mathcal{R}$ and R.
6. Using SPIN's LTL Property Manager, perform verification of $\mathcal{M}_f^\mathcal{R}$ against LTL_S. If verification is successful, then $\mathcal{L}(\mathcal{A}_\mathcal{R}) \subseteq \mathcal{L}(\mathcal{A}_\mathcal{S})$ holds, otherwise not.

Fig. 6. Algorithm for Proof of Language Containment.

5.2 Verifying Correctness Requirement 2 (CR-2)

Proposition 2. *Given a BSV specification $\mathcal{S}$ and its implementation R, R is a refinement of $\mathcal{S}$ iff $\mathcal{M}_f^\mathcal{R}$ is a refinement of $\mathcal{M}_f^\mathcal{S}$, where $\mathcal{M}_f^\mathcal{R}$ and $\mathcal{M}_f^\mathcal{S}$ are corresponding PROMELA models of R and $\mathcal{S}$ respectively.*

Based on Proposition 2, an implementation R of a BSV specification $\mathcal{S}$ can be verified for *Correctness Requirement 2* (Section 3) using Algorithm *VerfLangCont* (Figure 6). Algorithm $VerfLangCont$ generates PROMELA model $\mathcal{M}_f^\mathcal{R}$ for implementation R. It also generates an LTL specification LTL_S encoding all the behaviors $\mathcal{L}(\mathcal{A}_\mathcal{S})$ of specification $\mathcal{S}$ (based on $AOA\ Semantics$) with respect to the state in $\mathcal{M}_f^\mathcal{R}$. Such LTL specifications are generated in the style of TLA [10] (as an example, see Appendix - Listing 1.4 which is explained later) because BSV specifications are written in terms of actions and not explicitly in terms of the state of the design. However, note that Algorithm $VerfLangCont$ only encodes the safety properties and not the liveness assumptions in the generated LTL specification LTL_S. This is because we are interested in showing that safety properties encoded in LTL_S also hold in implementation R. In other words, R is a refinement of $\mathcal{S}$.

Using SPIN's LTL Property Manager, model $\mathcal{M}_f^{\mathcal{R}}$ is then verified against LTL_S for ensuring the language-containment relation $\mathcal{L}(\mathcal{A}_{\mathcal{R}}) \subseteq \mathcal{L}(\mathcal{A}_{\mathcal{S}})$. This allows using SPIN for proving strong language-containment relationships for BSV designs.

5.3 Verifying Correctness Requirement 3 (CR-3)

Proposition 3. *Given two different implementations R and $R^{'}$ of a BSV specification $\mathcal{S}$, $R^{'}$ is a refinement of R iff $\mathcal{M}_f^{\mathcal{R}^{'}}$ is a refinement of $\mathcal{M}_f^{\mathcal{R}}$, where $\mathcal{M}_f^{\mathcal{R}^{'}}$ and $\mathcal{M}_f^{\mathcal{R}}$ are corresponding PROMELA models of $R^{'}$ and R respectively.*

Based on Proposition 3, Algorithm $VerfLangCont$ (Figure 6) can also be used to verify *Correctness Requirement 3* (Section 3). For this, all the steps of the Algorithm $VerfLangCont$ remain same except that the inputs to the algorithm in this case will be implementations R and $R^{'}$ (instead of specification $\mathcal{S}$ and R as shown in Figure 6). Consequently, the algorithm will verify PROMELA model of implementation $R^{'}$ against the LTL specification encoding all the behaviors of implementation R with respect to state in $R^{'}$.

5.4 Sample Experiments

I. We used Algorithm $VerfLangCont$ (Figure 6) to successfully verify the language-containment relations $\mathcal{L}(\mathcal{A}_{\mathcal{R}}^{\mathcal{MCS}}) \subseteq \mathcal{L}(\mathcal{A}_{\mathcal{S}})$ and $\mathcal{L}(\mathcal{A}_{\mathcal{R}}^{\mathcal{ACS}}) \subseteq \mathcal{L}(\mathcal{A}_{\mathcal{S}})$ among VMC specification of Listing 1.1 and its implementations R_{MCS} and R_{ACS}, shown in Appendix in Listings 1.2 and 1.3.

Listing 1.4 in Appendix shows the LTL specification LTL_S generated by Algorithm $VerfLangCont$, which corresponds to all behaviors $\mathcal{L}(\mathcal{A}_{\mathcal{S}})$ of the VMC specification. In Listing 1.4, variables *count_old*, *action_fired* and *one_action_fired* are used for expressing LTL_S with respect to the state in the PROMELA models of Listings 1.2 and 1.3 as follows -

1. *count_old* stores the old value of *count* at the start of every process and is used in LTL_S to compare any updates on the value of *count* (during the execution of the process) with the old value.
2. *action_fired* and *one_action_fired* are used in LTL_S to check the state of the PROMELA model at points (just after a process has executed its atomic block) which map to the state changes during the execution of the BSV design.

II. We also used Algorithm $VerfLangCont$ to verify if $\mathcal{L}(\mathcal{A}_{\mathcal{R}}^{\mathcal{MCS}}) \subseteq \mathcal{L}(\mathcal{A}_{\mathcal{R}}^{\mathcal{ACS}})$ holds for VMC.

Result - Verification done by SPIN proved that $\mathcal{L}(\mathcal{A}_{\mathcal{R}}^{\mathcal{MCS}}) \subseteq \mathcal{L}(\mathcal{A}_{\mathcal{R}}^{\mathcal{ACS}})$ does not hold for VMC, and pointed out a behavior shown by R_{MCS} which is not shown by R_{ACS}. This behavior corresponds to the case when $count \geq 100$ holds, *moneyBack* is *False* and external environment signals the execution of action *moneyBackButton* (assuming actions *tenCentIn* and *fiftyCentIn* are not signalled to execute). In such a state, the behaviors of the two implementations R_{MCS} and R_{ACS} differ as follows-

1. R_{MCS} (to which the generated PROMELA model corresponds) executes actions *doDispenseGum* as well as *moneyBackButton* in the first clock cycle. This is followed by the execution of *doDispenseMoney* in the next clock cycle.
2. R_{ACS} (to which the generated LTL specification corresponds) only executes *doDispenseGum* in the first clock cycle. This is because R_{ACS} allows only one action to execute in a single clock cycle, and based on the sequential ordering it just executes action *doDispenseGum*, thus dispensing gum and reducing *count* by 50 cents. In the next clock cycle, $count \geq 50$ holds and *doDispenseGum* will be again selected for execution (assuming actions *tenCentIn* and *fiftyCentIn* are not signalled to execute). Action *moneyBackButton* is not executed in such a behavior.

Thus, as successfully highlighted by SPIN, $\mathcal{L}(\mathcal{A}_{\mathcal{R}}^{\mathcal{MCS}}) \subseteq \mathcal{L}(\mathcal{A}_{\mathcal{R}}^{\mathcal{ACS}})$ does not hold for VMC.

III. Furthermore, we used SPIN to prove that $\mathcal{L}(\mathcal{A}_{\mathcal{R}}^{\mathcal{ACS}}) \subseteq \mathcal{L}(\mathcal{A}_{\mathcal{R}}^{\mathcal{MCS}})$ also does not hold for VMC. This implies that both implementations R_{MCS} and R_{ACS} of VMC conform to its specification but one is not a refinement of other. For some BSV designs, such relationships are acceptable because *Correctness Requirement 3 (CR-3)* (Section 3) needs to be satisfied only if required based on the design requirements. For VMC, R_{ACS} and R_{ACS} contain behaviors conforming to the specification and both implementations are acceptable.

These experiments demonstrate that the language-containment based verification approach of Algorithm $VerfLangCont$ (Figure 6) can be successfully used to compare behaviors of different implementations of BSV designs.

6 Summary

Verification of hardware designs at a level of abstraction above RTL aids in faster and efficient verification early in the design cycle. BSV-based high-level synthesis converts a BSV specification of a hardware design into its corresponding RTL code. However, verification of BSV designs at levels of abstraction above RTL has not been looked at until now. In this paper, we present a verification approach that provides such a verification path for BSV designs. We propose the conversion of BSV-based hardware designs into corresponding PROMELA models containing implementation-specific scheduling information. Such PROMELA models can then be verified using SPIN for their essential properties. Moreover, for stronger language-containment proofs, we propose a technique that uses SPIN to verify if a particular implementation of the BSV design is a refinement of its specification or some other implementation. We successfully used our verification techniques to check different BSV designs for correctness and language-containment based proofs.

References

1. Holzmann, G.J.: The SPIN Model Checker. Addison Wesley (2004)
2. Holzmann, G.J.: The model checker SPIN. Software Engineering **23**(5) (1997) 279–295
3. SMV: http://www-cad.eecs.berkeley.edu/~kenmcmil/
4. Raghunathan, A., K.Jha, N., Dey, S.: High-Level Power Analysis And Optimization. Kluwer Academic Publishers (1998)
5. Singh, G., Shukla, S.K.: Algorithms for Low Power Hardware Synthesis from CAOS - Concurrent Action Oriented Specifications. Special Issue of International Journal of Embedded Systems on Power/Energy/Thermal topics (IJES'07) (2007)
6. Singh, G., Shukla, S.K.: Low-Power Hardware Synthesis from TRS-based Specifications. International Conference on Formal Methods and Models for Codesign (MEMOCODE'06) (2006)
7. Hoe, J.C., Arvind: Hardware Synthesis from Term Rewriting Systems. Proceeding of VLSI'99 Lisbon, Portugal (December 1999)
8. Arvind, Nikhil, R., Rosenband, D., Dave, N.: High-level synthesis: An Essential Ingredient for Designing Complex ASICs. Proceedings of the International Conference on Computer Aided Design (ICCAD'04) (November 2004) 775–782
9. Singh, G., Schwartz, J.B., Ahuja, S., Shukla, S.K.: Techniques for Power-aware Hardware Synthesis from Concurrent Action Oriented Specifications. Journal of Low Power Electronics (JOLPE) **3**(2) (August 2007) 156–166
10. Lamport, L.: The temporal logic of actions. ACM Transactions on Programming Languages and Systems **16**(3) (May 1994) 872–923

Appendix - Algorithms and Code Listings

ALGORITHM: *GenVARS*. **IN:** BSV Specification $\mathcal{S}$. **OUT:** Set of variables V.

1. Initialize $V = \phi$. (**Note:** V is the set of variables of PROMELA model $\mathcal{M}$.)
2. For each $s \in S$ such that $s \equiv (t, n, in)$,
   (a) Construct a variable $v \equiv (t_p, n, in)$, where t_p is the PROMELA data-type corresponding to t.
   (b) Construct a variable $v_{old} \equiv (t_p, n_{old}, in)$. (**Note:** During execution of model, v_{old} will be used to store old value of v.)
   (c) Add v and v_{old} to V.
3. Add variables $v_{act} \equiv$ (byte, action, n+1) and $v_{act}^{Frd} \equiv$ (bool, action_fired, False) to V, where n is the total number of actions $|A|$ of specification $\mathcal{S}$.
4. For each action $a \in A_m$ (**Note:** A_m is the set of interface methods.)
   (a) Construct a variable $v \equiv (\text{bool}, m,$ False), where m is the name of action a. Add v to V.
5. Output V.

Fig. 7. Algorithm for generating set of variables of PROMELA Model.

ALGORITHM: *GenPROCS*. **INPUT:** 1.BSV Specification $\mathcal{S}$, 2.Set of Variables V. **OUTPUT:** Set of processes P.

1. Initialize $P = \phi$. (**Note:** At the end, P will contain set of processes of PROMELA model $\mathcal{M}$.)
2. For each $a_i \in A$,
   (a) Using V, construct sets of variables V_g^i and V_b^i (with appropriate PROMELA data-types) corresponding to sets S_g^i and S_b^i of specification $\mathcal{S}$ respectively.
   (b) Construct function $g_i^P = g_i(V_g^i)$ corresponding to guard of a_i.
3. For each $a_i \in A$,
   (a) Initialize $B_i^P = \phi$, $B_i^{tmp} = \phi$.
   (b) For each $s \in S_u^i$ such that $s \equiv (t, n, in)$
      i. Find variables $v \equiv (t_p, n, in)$ and $v_{old} \equiv (t_p, n_{old}, in)$ corresponding to s in set V.
      ii. Construct a PROMELA statement $stmt \equiv (v_{old}, v)$ denoting an assignment of the form $v_{old} = v$.
      iii. Add $stmt$ to B_i^P.
   (c) For each $b \in B_i$ such that $b \equiv (s, b(S_b^i))$,
      i. Find a variable $v \equiv (t_p, n, in)$ corresponding to s in set V.
      ii. Construct a PROMELA statement $stmt \equiv (v, b(V_b^i))$ denoting an assignment of the form $v = b(V_b^i)$. (**Note:** $b(V_b^i)$ is constructed using types of PROMELA statements that retain corresponding meaning of $b(S_b^i)$.)
      iii. Add $stmt$ to B_i^{tmp}.
   (d) Order statements of B_i^{tmp} such that they model the concurrent hardware behavior of action a_i. This may require using variables $v_{old} \in V$ which are used to store old values of variables corresponding to state elements of $\mathcal{S}$.
   (e) $B_i^P = B_i^P \cup B_i^{tmp}$.
   (f) Construct a PROMELA statement $stmt \equiv$ (action_fired, True) denoting an assignment of the form, action_fired = True.
   (g) Add $stmt$ to B_i^P.
   (h) If $a_i \in A_m$ then,
      i. $g_i^P = g_i^P$ (**Note:** For interface methods, no need to update guards with conflict information as their execution is decided by external module.)
   (i) Else, for each $a_j \in C_i$,
      i. If $a_j \in A_m$ then,
         A. Find a corresponding variable $v \equiv (\text{bool}, m_j, in)$ in V, where m_j is name of action a_j.
         B. $g_i^P = g_i^P$ && !v.
      ii. Else, $g_i^P = g_i^P$ && !g_j^P.
   (j) Add (m, B_i^P, g_i^P) to P, where $m =$ IFC_m_i if $a_i \in A_m$, and $m = m_i$ otherwise (m_i is the name of action a_i).
4. Output P.

Fig. 8. Algorithm for generating set of processes of PROMELA Model.

ALGORITHM: *GenProcCycle*.
INPUT: 1. BSV Specification $\mathcal{S}$, 2. Set of Variables V, 3. Set of processes P.
OUTPUT: PROMELA process *start_of_cycle*.

1. Initialize $B^P_{cycle} = \phi$.
2. For each $pr \in P$ such that $pr \equiv (m, B, g)$,
   (a) **Comment:** Here, we generate statements corresponding to values read (inputs) from external modules of the hardware design. For verification purposes, all possible combinations of values are generated. Values are read at the start of every cycle.
   (b) If m corresponds to the name of an action a such that $a \in A_m$ then,
      i. Find a variable $v \equiv (\text{bool}, m, in)$ in set V.
      ii. Construct a PROMELA statement $stmt_F \equiv (g, v, \text{False})$ denoting an assignment of the form $v =$ False if g computes to *False* or if some other higher priority process is enabled.
      iii. Construct a PROMELA statement $stmt_T \equiv (g, v, (\text{False, True}))$ denoting a non-deterministic assignment $v =$ False or $v =$ True if g computes to *True* and no other higher priority process is enabled.
      iv. Add $stmt_F$ and $stmt_T$ to B^P_{cycle}.
3. Using V, construct a set $STMT$ of PROMELA statements, if any, corresponding to resetting of any state elements of $\mathcal{S}$ at the start of a hardware clock cycle.
4. $B^P_{cycle} = B^P_{cycle} \cup STMT$.
5. Construct a PROMELA statement (action_fired, False) and add it to B^P_{cycle}.
6. Output (start_of_cycle, $B^P_{cycle}, True$).

Fig. 9. Algorithm for generating process denoting start of hardware cycle in PROMELA Model.

ALGORITHM: *AddSeqSched.*
INPUT: 1. BSV Specification $\mathcal{S}$, 2. PROMELA Model $\mathcal{M}$ without scheduling information.
OUTPUT: PROMELA Model $\mathcal{M}_f$ executing processes based on *AOA Semantics.*

1. Initialize $P_s = \phi$.
2. Let V and P be the set of variables and processes of a PROMELA Model $\mathcal{M}$. Let $n = |P| - 1$.
3. For each $pr \in P$ such that $pr \equiv (m, B, g)$ and pr does not correspond to 'init' process.
   (a) If m corresponds to 'start_of_cycle' then,
      i. Construct a PROMELA statement *cond* corresponding to comparison operation (action== n).
      ii. Construct a conditional expression $cond_d$ using set S of state elements of specification $\mathcal{S}$ such that when $cond_d$ is *True*, no action of $\mathcal{S}$ is enabled. If it is not possible to construct any such statement, go to Step (c).
      iii. Construct $cong_d^p$ corresponding to $cond_p$ in terms of variables in V.
      iv. Construct a PROMELA statement *stmt* performing assignment, action=n, if expression $cond_d^p$ evaluates to *True*, or assignment, action=0, otherwise.
      v. Add *stmt* to B.
   (b) Else,
      i. If m corresponds to the name of an action a such that $a \in A_m$ then,
         A. Find a corresponding variable $v \equiv (\text{bool}, m, in)$ in V.
         B. Construct a PROMELA statement *cond* corresponding to comparison operation ((action$\neq n$) && v).
      ii. Else,
         A. Construct a PROMELA statement *cond* corresponding to comparison operation ((action$\neq n$) && g).
      iii. Construct a PROMELA statement $stmt \equiv (\text{action}, n)$ denoting an assignment action=n.
      iv. Add *stmt* to B.
   (c) Construct a PROMELA atomic block, $Block_{atomic} \equiv atomic\ \{cond -> \ B\}$.
   (d) Construct a PROMELA statement $Block_{do} \equiv \{do :: Block_{atomic};\ od\}$ using repetition construct 'do'.
   (e) Add $(m, Block_{do})$ to P_s.
4. Output PROMELA model $\mathcal{M}_f$ such that P_s is its set of processes and V is its set of variables.

Fig. 10. Algorithm for modeling AOA Execution Semantics in PROMELA Model.

ALGORITHM: *AddConcSched*.
INPUT: 1. BSV Specification $\mathcal{S}$, 2. PROMELA Model $\mathcal{M}$ without scheduling information, 3. Sequential Ordering S_{order} of an implementation R, 4. Maximum number of actions max allowed to execute concurrently in R.
OUTPUT: PROMELA Model $\mathcal{M}_f$ based on scheduling information of R.

1. Initialize $P_s = \phi$, $i = 1$. Let V and P be the set of variables and processes of a PROMELA Model $\mathcal{M}$. Let $n = |P| - 1$.
2. Construct a variable $v \equiv$ (byte,num, 0) (**Note:** num will be used to denote the number of processes executed so far in the model). Add v to V.
3. While $(i < n)$
   (a) Find a process $pr \in P$ such that $pr \equiv (m, B, g)$ corresponds to i^{th} element of S_{order}.
      i. If m corresponds to the name of an action a such that $a \in A_m$ then,
         A. Find a corresponding variable $v \equiv$ (bool, m, in) in V.
         B. Construct a PROMELA statement $cond_B$ corresponding to comparison operation ((action== i) && v).
      ii. Else,
         A. Construct a PROMELA statement $cond_B$ corresponding to comparison operation ((action== i) && g).
      iii. Construct a PROMELA statement $stmt1 \equiv$ (num, num+1) denoting an assignment of the form, num=num+1.
      iv. Construct a PROMELA statement $stmt2 \equiv$ ((num== max), (action= n), (action= $i + 1$)) denoting an assignment of the form action=n if (num==max) is $True$, and assignment of the form, action=i+1 otherwise. Add $stmt1$ and $stmt2$ to B.
      v. Construct a PROMELA statement block, $Block_B \equiv \{cond_B -> B\}$.
      vi. Construct a PROMELA statement $cond$ corresponding to comparison operation (action == i).
      vii. Construct a PROMELA statement $stmt3 \equiv$ (action, $i + 1$) denoting an assignment of the form, action = i+1.
      viii. Construct a PROMELA statement $stmt4 \equiv$ (action_fired, False) denoting an assignment of the form, action_fired = False. Let $B^{'} = \{stmt3, stmt4\}$
      ix. Construct a PROMELA statement block, $Block_{B'} \equiv \{cond -> B^{'}\}$.
      x. Construct a PROMELA atomic block, $Block_{atomic} \equiv atomic\ \{Block_{B'}\ unless\ Block_B\}$.
      xi. Construct a PROMELA statement $Block_{do} \equiv \{do :: Block_{atomic};\ od\}$ using repetition construct 'do'.
      xii. Add $(m, Block_{do})$ to P_s. $i = i + 1$.
4. For process $pr \in P$ such that $pr \equiv (m, B, True)$ corresponds to 'start_of_cycle' process,
   (a) Construct a PROMELA statement $cond$ corresponding to comparison operation (action== n).
   (b) Construct a PROMELA statement $stmt1 \equiv$ (action, 1) denoting an assignment of the form, action=1.
   (c) Construct a PROMELA statement $stmt2 \equiv$ (num, 0) denoting an assignment of the form num=0. Add $stmt1$ and $stmt2$ to B.
   (d) Construct a PROMELA atomic block, $Block_{atomic} \equiv atomic\ \{cond -> B\}$.
   (e) Construct a PROMELA statement $Block_{do} \equiv \{do :: Block_{atomic};\ od\}$ using repetition construct 'do'. Add $(m, Block_{do})$ to P_s.
5. Output PROMELA model $\mathcal{M}_f$ such that P_s is its set of processes and V is its set of variables.

Fig. 11. Algorithm for modeling Concurrent Execution Semantics in PROMELA Model.

Listing 1.2. PROMELA Models corresponding to Maximal and Alternative Concurrent Schedules of VMC.

```
int count, count_old; byte action;
bool moneyBack, gumControl, tenCentIn, fiftyCentIn,
    moneyBackButton, action_fired, one_action_fired;

proctype IFC_tenCentIn() {
  do
  :: atomic { { (action==1) -> {action = 2; action_fired = 0;} }
               unless
               { (action==1 && tenCentIn) ->
                    { count_old = count;
                      count = count + 10;
                      one_action_fired = 1; action_fired = 1;
                      action=2; –FOR MAXIMAL CONCURRENT SCHEDULE
                      action=6; –FOR ALTERNATIVE CONC SCHEDULE
                    }
               } }
  od
}
proctype IFC_fiftyCentIn() {
  do
  :: atomic { { (action==2) -> {action = 3; action_fired = 0;} }
               unless
               { (action==2 && fiftyCentIn) ->
                    { count_old = count;
                      count = count + 50;
                      one_action_fired = 1; action_fired = 1;
                      action=3; –FOR MAXIMAL CONCURRENT SCHEDULE
                      action=6; –FOR ALTERNATIVE CONC SCHEDULE
                    }
               } }
  od
}
proctype doDispenseMoney() {
  do
  :: atomic { { (action==3) -> {action = 4; action_fired = 0;} }
               unless
               { (action==3 && moneyBack) ->
                    { count_old = count;
                      if
                      :: (count == 0) -> moneyBack = 0;
                      :: else -> { count = count - 10;
                                   if
                                   :: (count == 0) ->
                                       moneyBack = 0;
                                   :: else -> skip;
                                   fi }
                      fi
                      one_action_fired = 1; action_fired = 1;
                      action=4; –FOR MAXIMAL CONCURRENT SCHEDULE
                      action=6; –FOR ALTERNATIVE CONC SCHEDULE
                    }
               } }
  od
}
proctype doDispenseGum() {
  do
  :: atomic { { (action==4) -> {action=5; action_fired = 0;} }
               unless
               { (action==4 && !tenCentIn && !fiftyCentIn
                  && !moneyBack && count >= 50) ->
                    { count_old = count;
                      count = count - 50;
                      gumControl = 1;
                      one_action_fired = 1; action_fired = 1;
                      action=5; –FOR MAXIMAL CONCURRENT SCHEDULE
                      action=6; –FOR ALTERNATIVE CONC SCHEDULE
                    }
               } }
  od
}
```

Listing 1.3. PROMELA Models corresponding to Maximal and Alternative Concurrent Schedules of VMC (Continued).

```
proctype IFC_moneyBackButton() {
  do
  :: atomic { { (action==5) -> {action=6; action_fired = 0;} }
               unless
               { (action==5 && moneyBackButton) ->
                    { count_old = count;
                      moneyBack = 1;
                      one_action_fired = 1; action_fired = 1;
                      action = 6;
                    }
               } }
  od
}

proctype start_of_cycle() { /*Denotes end of a clock cycle.*/
  do
  :: atomic { if
               :: (action==6) ->
                   { if /* Read external stimulus*/
                     :: moneyBack -> {tenCentIn=0; fiftyCentIn=0;
                          moneyBackButton=0;}
                     :: else -> if
                                ::{tenCentIn=1; fiftyCentIn=0;
                                    moneyBackButton=1;}
                                ::{tenCentIn=1; fiftyCentIn=0;
                                    moneyBackButton=0;}
                                ::{tenCentIn=0; fiftyCentIn=1;
                                    moneyBackButton=1;}
                                ::{tenCentIn=0; fiftyCentIn=1;
                                    moneyBackButton=0;}
                                ::{tenCentIn=0; fiftyCentIn=0;
                                    moneyBackButton=1;}
                                ::{tenCentIn=0; fiftyCentIn=0;
                                    moneyBackButton=0;}
                                fi
                     fi
                     gumControl = 0;
                     one_action_fired = 0; action_fired = 0;
                     action = 1; }
               fi }
  od
}

init {
  atomic {
     count = 0; count_old = 0;
     moneyBack = 0; gumControl = 0;
     tenCentIn = 0; fiftyCentIn= 0; moneyBackButton = 0;
     action = 6;

     run IFC_tenCentIn(); run IFC_fiftyCentIn();
     run doDispenseMoney(); run doDispenseGum();
     run IFC_moneyBackButton(); run start_of_cycle(); }
}
```

Listing 1.4. LTL Specification (in TLA style) encoding all behaviors of VMC Specification w.r.t. PROMELA models for Maximal and Alternative Concurrent Schedules.

```
#define eqTo0 count == 0
#define gEq50 count >= 50
#define gt10  count > 10

#define inc10 count_old == (count - 10)
#define inc50 count_old == (count - 50)
#define dec10 count_old == (count + 10)
#define dec50 count_old == (count + 50)

#define ins10 tenCentIn
#define ins50 fiftyCentIn
#define mbbtn moneyBackButton

#define dspGm (dec50 && gumControl)


[] ( ( (moneyBack &&  gt10) -> X (!action_fired U (dec10 &&
    moneyBack))
    ) &&
    ( (moneyBack && !gt10) -> X (!action_fired U (eqTo0 && !
        moneyBack))
    ) &&
    ( (( one_action_fired && !moneyBack && gEq50) -> X (!
        action_fired U (inc10 || inc50 || moneyBack || dspGm)))
        &&
      ((!one_action_fired && !moneyBack && gEq50 && !ins10 && !
          ins50 && mbbtn) -> X (!action_fired U (moneyBack ||
          dspGm))) &&
      ((!one_action_fired && !moneyBack && gEq50 && !ins10 && !
          ins50 && !mbbtn) -> X (!action_fired U dspGm))
    ) &&
    ( (( one_action_fired && !moneyBack && !gEq50) -> X (!
        action_fired U (inc10 || inc50 || moneyBack))) &&
      ((!one_action_fired && !moneyBack && !gEq50 && !ins10 && !
          ins50 && mbbtn) -> X (!action_fired U moneyBack)) &&
      ((!one_action_fired && !moneyBack && !gEq50 && !ins10 && !
          ins50 && !mbbtn) -> X (!action_fired U (inc10 || inc50
           || moneyBack)))
    ) &&
    ( ((!one_action_fired && !moneyBack &&  ins10 && !ins50 &&
        mbbtn) -> X (!action_fired U (inc10 || moneyBack))) &&
      ((!one_action_fired && !moneyBack &&  ins10 && !ins50 && !
          mbbtn) -> X (!action_fired U inc10)) &&

      ((!one_action_fired && !moneyBack && !ins10 && ins50 &&
          mbbtn) -> X (!action_fired U (inc50 || moneyBack))) &&
      ((!one_action_fired && !moneyBack && !ins10 && ins50 && !
          mbbtn) -> X (!action_fired U inc50))
    )
)
```