Correct Safety Critical Hardware Descriptions via Static Analysis and Theorem Proving

Nicholas Moore, Mark Lawford
McMaster Centre for Software Certification, McMaster University,
1280 Main St. W, Hamilton, Ontario, L8S 4K1 Canada
{ moorenc, lawford }@mcmaster.ca

***Abstract*—We propose a new method for embedding Bluespec SystemVerilog descriptions in PVS's higher-order proof logic. In contrast to previous embeddings, our approach accepts a greater subset of BSV and provides a far greater degree of automation. Our custom software tool transforms the action-oriented semantics of BSV to a state-centric Kripke structure, enabling automated theorem proving. We demonstrate our verification technique by applying it to one of the function blocks of the IEC 61131-3 standard for PLCs.**

I. Introduction

Industrial processes everywhere are facing the challenges of aging equipment. Field Programmable Gate Array (FPGA) based control systems are being adopted to replace aging hardware, increasing performance, reducing power consumption, and increasing flexibility. However, safety critical applications must still meet regulatory requirements, including formal verification. We propose a methodology for the verification of FPGA designs in Bluespec SystemVerilog (BSV) [1] using the Prototype Verification System (PVS) [2]. We show that logical models mechanically extracted from BSV designs may demonstrate consistency between the original BSV description and formal requirements specifications. Model extraction is performed automatically by a translation tool. A case study of our proposed method is presented.

Preliminary information will be presented in §II. In §III we describe our translation algorithm. We describe the proof process in §IV, including the use of tabular specifications to automatically generate proof sub-goals. We present our translation tool BAPIP (**B**luespec **A**nd **P**VS **I**nterlanguage **P**rocessor) in §V, and discuss its structure, extensibility and applicability. We present our case study in §VI, examining a successfully verified FPGA based PLC function block. Finally, we discuss related research in §gVII and future work in §VIII.

II. Preliminaries

A. BSV

Bluespec SystemVerilog is a hardware description language (HDL) that originated as a Haskell library [3]. BSV designs can be produced more quickly and with fewer bugs than less abstract languages such as Verilog, which it compiles to [3]. Unlike other HDLs, BSV modules interact via atomic transactions, implemented as method calls.

Bluespec modules are composed of submodules and actions. Actions control how submodules interact, where memory elements such as registers are also submodules. BSV considers register read and write operations to be methods of register submodules, though syntactic sugar is still provided. There are two types of actions: *rules*, which are executed (or "fire") when permitted by the action arbitration mechanism, and *methods*, which are invoked by a module's parent module. Actions are composed of a guard and an set of statements, where a guard is a Boolean expressions that controls action execution. Actions may execute once per clock cycle, must execute completely, and all the statements in an action execute concurrently and independently. Each memory read returns the previous clock cycle's value, so race conditions are avoided.

Action arbitration is complex, and merits special attention. On any given clock cycle, all actions are either active or inactive. Rules have their guards evaluated every clock cycle to decide whether they are active. Methods are active only if invoked by the supermodule. Once invoked, methods are treated as actions of the highest precedence. Active actions must succeed in arbitration in order to fire. If there exist no conflicts between active actions, all active actions fire. Two actions are in conflict if the intersection of the sets of registers they write to is non-empty. If there are conflicts and action precedence has been specified, the action with higher precedence will pre-empt the action with lower precedence. If no precedence is specified, the action which fires will be selected arbitrarily.

Consider a module containing two rules (A and B) which write to the same register, and have guards which are not mutually exclusive. When scheduling these rules, if both rules are active, the BSV arbitrator will make a deterministic but unpredictable scheduling decision. However, if a `descending_urgency` attribute is added which states that A has higher priority than B, A will pre-empt B. B will now fire only if A cannot.

To logically model BSV descriptions, we first consider memory-holding submodules to contain some value at every point in time, which we refer to as state. We also consider actions a system of transitioning between different states. Therefore, BSV descriptions may be described logically as Kripke structures as proposed by [4]. We formally define a Kripke structure $K = (S, S_0, T, L)$

978-1-5386-0422-9/17 $31.00 © 2017 IEEE
DOI 10.1109/FormaliSE.2017..11

where S is the set of program states, S_0 is the initial state, $T \subseteq S \times S$ is a left-total transition relation and $L : S \rightarrow 2^{AP}$ where AP is a set of atomic propositions.

In BSV it is mandatory to initialize declared state elements. This arrangement of memory comprises the initial state S_0. Rules and methods, along with the action arbitration semantic, create the state transition function. This logical approximation of BSV descriptions is of central importance to the PVS embedding presented in [4] and forms the logical framework for our own translation.

B. PVS

Prototype Verification System (PVS) is an open source emacs plugin developed by SRI International [5]. It consists of a specification language and interactive proof environment, providing both high mechanization and the expressive power of higher order logic. The proof environment automatically generates sub-goals and counter-examples, and produces highly legible proofs. In the past, PVS has been used to successfully verify safety critical embedded systems, such as the AAMP5 avionics microprocessor [6], and shutdown systems for the Darlington nuclear power plant [7]. Use of PVS to verify our BSV modules will be presented in §IV.

C. Comparison with Previous Approaches

The BSV training materials present two semantics for the execution of BSV modules, a timed semantic for hardware clock cycles, and an untimed semantic for execution steps [8], [9]. Under the untimed semantics one action executes atomically per execution step [8]. The action is selected arbitrarily, but deterministically. The timed semantics, discussed in §II.A, relate untimed steps to hardware clock cycles by iterating the untimed semantics until no actions remain which are eligible for execution.

Richards and Lester [4] first presented the logical model of BSV descriptions presented in §II.A. Their embedding of BSV in PVS attempted to minimize the syntactic difference between BSV and PVS, in order to facilitate manual translation between BSV and PVS. As a result, they present as transition predicates the disjunction of monadic representations of all actions in a module. This approach avoids modelling BSV's action arbitration semantics, allowing them to prove theorems for one transition during which any combination of rules can be invoked. While it is possible to prove certain types of theorems in this manner [4], this approach includes many possible execution patterns which are precluded by the original hardware description. The untimed semantics are not accurately modelled, because multiple actions may execute under universal disjunction, and the timed semantics are not accurately modelled, as no attempt is made to replicate the exclusionary properties of action arbitration. If the transition is a relation, a single pre-state may have many equally valid post-states. Our approach is to enforce the constraint of BSV descriptions until a single transition function may be obtained. As a result, our transition predicates map to hardware clock cycles, allowing us to assert claims regarding timing characteristics and the behaviour of the hardware synthesized from a BSV description.

III. The Semantic Encoding of BSV in PVS

We propose BAPIP, a translation tool encoding the translation presented in this section. The product of this encoding is a set of four PVS theories: type definition, state, transition, and top-level theories. The generation of these theories is described below. See [10] for full code examples of our case study.

A. Generating a Type Definitions Theory

BAPIP currently supports BSV type synonyms and enumeration types, and `Int`, `UInt`, and `Bit` types are included by default. Data types in BSV with declared bit-widths become sub-ranged integer types in PVS. Due to similar syntax, BSV type synonyms and enumerations require only minor syntactic alteration. While it is possible to include enumeration types in type classes in BSV, these distinctions are unnecessary for PVS's logical encoding.

BSV

$\langle$*type synonym*$\rangle$ typedef $\langle$*BSV type*$\rangle$ $\langle$*name*$\rangle$;

$\langle$*enumeration*$\rangle$ ::= typedef enum { $\langle$*e-list*$\rangle$ } $\langle$*name*$\rangle$;
| typedef enum { $\langle$*e-list*$\rangle$ } $\langle$*name*$\rangle$ deriving ($\langle$*typeclass list*$\rangle$) ;

PVS

$\langle$*type synonym*$\rangle$ ::= $\langle$*name*$\rangle$: type = $\langle$*PVS type*$\rangle$

$\langle$*enumeration*$\rangle$ ::= $\langle$*name*$\rangle$: type = { $\langle$*enumeration list*$\rangle$ }

B. Generating a State Theory

The state declared for each module in a BSV module creates a record type in PVS. Instantiated submodules are fields of the state record, typed with the state type of the submodule being instantiated. A module instantiation predicate is also provided to initialize modules (see §IV). In §II.A, interpretation of BSV descriptions as Kripke structures was discussed. The state record type corresponds to S in our Kripke model, and the instantiation predicate corresponds to S_0.

BSV

```
Reg#(Int#(16)) foo <- mkReg(5);
Reg#(Bool) bar <- mkReg(False);
```

PVS

```
MyModule : type =
  [# foo: Int(16)
  , bar: bool #]

MyModule_var : var MyModule

mkMyModule (MyModule_var) : bool
  =   MyModule_var‘foo = 5
  AND MyModule_var‘bar = False
```

The code segments above illustrate the conversion from BSV state declarations to a PVS state record and instantiation predicate.

C. Generating a State Transition Theory

To generate state transition predicates, BSV’s rule-centric semantics must be transformed into register-centric update expressions. While it is possible to organize BSV hardware descriptions such that each action is mutually exclusive, this can be an impractical limitation. Allowing the action arbitrator to resolve conflict arbitrarily is also undesirable in safety critical applications. BAPIP therefore exposes conflicts unaddressed by manual precedence assignment. The `descending_urgency` and `preempts` attributes manually assign precedence, and must be used to resolve conflicts, or BAPIP will not continue model extraction.

Algorithmically, for each module and submodule all actions are collected. Each action performs write operations on certain registers, the names of which are collected. Action precedence is applied to produce, for each action, a list of actions which pre-empt it, and a list of actions it preempts. Guard expressions for each action are augmented with pre-emption information. For each register, a list of actions writing to it is produced. A binary if-statement tree is generated that re-creates the pre-emption structure, including expressions being written to the register. From here, data is marshalled into “transition tables,” as defined by the following Haskell data structure:

```
data TransitionTable
  = TransSubModule Name [TransitionTable]
  | TransRegister Name TransTree deriving (Show)
data TransTree = Stem Guard TransTree TransTree
  | Leaf Expression deriving (Show)
```

Transition tables map to record update syntax, and sub-tables for submodules correspond to nested record updates. The transition predicate itself accepts at least two arguments, a pre state and a post state, as well as any arguments required by top-level method calls. The transition predicate is formulated as a test of equality between the post state and an updated pre state. If this predicate is used as an antecedent during theorem construction, it asserts the existence of a transition between the states used as arguments. The transition theory also contains functions which access state records in the same manner as the declared BSV access methods, which may be used to reference module output values.

BSV

```
(* descending_urgency =
       "auto_stop, inc" *)

  rule auto_stop (foo == 5);
    bar <= false;
  endrule

  rule inc (bar);
    foo <= foo + 1;
  endrule

  method Action start() if (!bar);
    bar <= true;
    foo <= 0;
  endmethod
```

PVS

```
MyModule_t (pre, post) : bool =
  ( post = pre with
    [ foo := if (bar AND
          (NOT (foo == 5)))
        then pre‘foo + 1
        else pre‘foo
      endif
    , bar := if (foo == 5)
        then False
        else pre‘bar
    ]
  )

MyModule_t_start
      (pre, post) : bool =
  ( post = pre with
    [ foo := if (NOT bar)
        then 0
        else if (bar AND
            (NOT (foo == 5)))
          then pre‘foo + 1
          else pre‘foo
        endif
    , bar := if (NOT bar)
        then True
        else if (foo == 5)
          then False
          else pre‘bar
    ]
  )
```

The above is an example transition, using the registers presented in §III.B. There are two rules, `auto_stop` and `inc`, as well as a method `start`. A descending urgency relationship is declared, specifying that `auto_stop` has higher priority than `inc`. This module will wait for `start()` to be invoked, at which point `foo` is reset. It will be incremented every subsequent clock cycle until reaching a value of 5. The `start()` method can only be invoked if `bar` is false.

Two transition predicates are produced: `MyModule_t_start` and `MyModule_t`. These are transitions during which `start()` is invoked and not invoked respectively. If `start()` is not invoked, the value of foo is incremented if the guard of `inc` is true and the guard of `auto_stop` is false. This is because `inc` is lower priority, and will not fire if it is pre-empted by the higher priority rule `auto_stop`. On the other hand, `bar` is written to only by `auto_stop`, the highest priority rule, and therefore the state of `inc`'s guard is irrelevant to the value `bar` receives. The transition in which `start()` is invoked is the same transition, but with one additional action entering arbitration. `start()` behaves like a rule, but enters arbitration with the highest priority, therefore potentially blocking the execution of both other rules.

D. Generating the Top-Level Theory

Some elements of the logical framework used by Pang et al. [11] have been adapted for BAPIP, such as the "tick" type. When parameterized by the hardware clock period, "tick" organizes inputs and states into a timeline. By making state and input variables functions from "tick" to some other type, a specific value of "tick" (i.e., a specific point in time) produces the value of the input or state at that time. This, in combination with functions allowing traversal of the timeline, allows us to relate state transitions to real time, a typical requirement of formal analysis of safety critical systems.

IV. Proving the Correctness of BSV Implementations

In order to prove the correctness of a BSV description, we must show that it conforms to its formal requirements, and we must demonstrate consistency. Consistency here being used in the same manner as [12]. In order to avoid the "False implies everything" problem, we must show that, for all pre-states, there exists some post state. If this is true, then the transition predicate we take as an antecedent can never be false by means of having no viable post state. Consistency is equivalent to the left-total property expressed in §II.A.

To construct a theorem determining functional correctness, we must invoke our transition predicate as an antecedent. It is also necessary to know what constitutes correct behaviour. In our case studies, the modules we implement have formal tabular requirements expressed in PVS. Requirements may be used to manually construct a set of pre and post condition pairs in the style of Hoare Logic. For each pair of pre and post conditions, a theorem (or sub-theorem) may be constructed, using pre conditions

```
                             HIGH_ALARM
                           +------------+
                           | HYSTERESIS |
X-----------------------+--|XIN1       Q|--+------------QH
                 +---+  |  |            |  |
H----------------| - |-----|XIN2        |  |
             +---|   |  |  |            |  |
             |   +---+  |  |            |  |
             +-------------|EPS         |  |  +-----+
      +---+  |          |  +------------+  +--| >=1 |
EPS---| / |--+          |                     |     |---Q
2.0---|   |  |          |    LOW_ALARM     +--|     |
      +---+  |          |  +------------+  |  +-----+
             |   +---+  |  | HYSTERESIS |  |
L----------------| + |-----|XIN1       Q|--+------------QL
             |   |   |  |  |            |
             +---|   |  +--|XIN2        |
             |   +---+     |            |
             +-------------|EPS         |
                           +------------+
```

Fig. 1: The LIMITS_ALARM function block, as described by the IEC61131-3 specification

as antecedents and post conditions as consequents. If a proof exists for each pair, we have demonstrated that the module under examination conforms to its requirements. However, this process may become tedious for large modules. We also present a method for using PVS to generate the necessary pre and post conditions automatically from tabular requirements expressed in PVS.

The theorems evaluating functionality have been proven by the `(grind :defs explicit)` proof strategy. This variant of `(grind)` limits redundant term rewriting. Specific proof times, are discussed in §V.

Proving consistency theorems requires minimal user interaction. In addition to generating the consistency theorems, BAPIP also generates proofs of these theorems using PVS's ProofLite proof scripting extension [13]. Consisting of the sequential skolemization, instantiation of the existentially quantified post state, and invocation of `(grind)`, these proofs need only be installed using the emacs command `install-prooflite-script`. They may be verified by invoking the interactive proof environment for the corresponding theorem and re-running the existing proof.

V. Case Study: Limits Alarm

Limits Alarm is a variation of the Hysteresis block, which triggers an alarm if the value being tested exceeds either an upper or a lower limit. It is composed of two hysteresis blocks, one which tests the upper threshold, the other testing the lower threshold. The Limits Alarm block has three corresponding output signals for the high threshold (QH), the low threshold (QL), and the global alarm signal (Q). The FBD implementation of Limits Alarm presented in [14] is shown in Fig. 1, as well as the tabular specifications reproduced in Fig. 3, which originally appeared in [11]. This block is commonly used to make binary decisions based on analog data streams, and features prominently in control systems such as the classical thermostat example, and safety critical control applications.

The Limits Alarm function block is constructed from two Hysteresis blocks as shown in Fig. 1. Hysteresis is a common behaviour of threshold-style control systems,

Condition	**Q**
$XIN1 > (XIN2 + EPS)$	True
$(XIN2 - EPS) \leq XIN1 \leq (XIN2 + EPS)$	NC
$XIN1 < (XIN2 - EPS)$	False

assuming $EPS > 0$

Fig. 2: Hysteresis Tabular Specification

where the threshold value for activating some behaviour is higher than the threshold for deactivation by some "hysteresis". This stabilizes the response against noise, preventing rapid oscillations between active and inactive states that could damage physical systems and actuators. The IEC 61131-3 [14] specification describes a standard behaviour for hysteresis software components used in programmable logic controllers. This behaviour has been formalized in the tabular expressions in Fig. 2. These tabular expressions originally appeared in [11].

In this specification, $XIN1$ is the data being tested, $XIN2$ is the threshold value being tested for, and EPS is the hysteresis margin. Values for $XIN1$, $XIN2$, and EPS are passed into the module via an input method, which also calculates the resulting value of Q, and registers it in memory. The value is then available to be read via an access method on the subsequent clock cycle. Though the IEC 61131-3 [14] specifies Real data types for $XIN1$, $XIN2$, and EPS, both real and integer versions of the block were developed as separate BSV modules, and both were proven consistent with the tabular expression. The integer version is available at [10].

Here, X is the signal under test, H is the high threshold, L is the low threshold, and EPS is the hysteresis margin. This block, expressed in BSV, may be found at [10]. Some concessions to implementation were made during the development of the correctness proof for this block. The integer implementation of this block highlights a disparity between the way Limits Alarm and Hysteresis handle thresholding. In the case of Hysteresis, the deadband is $X \pm \frac{EPS}{2}$, with $X + \frac{EPS}{2}$ being the upper limit, and $X - \frac{EPS}{2}$ being the lower. In Limits Alarm, H is the upper limit of the upper threshold, and the lower limit of the upper threshold is $H - EPS$. In order to use a Hysteresis block to produce this effect, the threshold value it is given must be $H - \frac{EPS}{2}$, and the hysteresis margin must be $\frac{EPS}{2}$, critically introducing division operators. For real data types this poses no problem, as real division is not subject to round-off error. Integer division, on the other hand, does introduce round-off error. In order to prove our integer implementation, it was necessary to revise our requirements such that they reflected this. Some algebraic simplifications were reversed to re-introduce integer round-off errors to the specifications.

Another concession to practicality was the clock cycle on which the global alarm Q was available. The global alarm Q, from a black box perspective, must be available on the clock cycle following the setting of the module's inputs via the input method. As expressed by Pang et. al. in PVS, the Q tabular specification requires values of QH and QL as inputs. Since these are calculated values, and not inputs to the module, they do not become available until one clock cycle has elapsed. Up to this point, it has been typical to rewrite the tabular specifications slightly such that they apply to the output value on the following clock cycle. For Q, the tabular specification applies to the same clock cycle that it's inputs occur in. In the implementation, this is achieved by allowing the access method for Q to directly access QH and QL, calculate Q, and return that value. This bypasses the need to calculate and store the value of Q within the Limits Alarm module itself. During the developement of this case study, incorrect synchronization was the primary source of proof failure.

The model of our BSV implementation of the Limits Alarm block, as extracted by BAPIP, was proven consistent with the tabular specification presented in Fig. 3. Translation of the module took an average of 26.4ms. The proof strategy employed was PVS's "grind," and the proof took an average of 1.51s on an ASUS G46V (quad-core Intel i5-3230M at 2.60GHz, 8GB RAM). Our proof of consistency for the Hysteresis block took an average of 0.088s to discharge. The reason for the discrepancy between the Hysteresis time of 0.422s and the Limits Alarm time of 0.088s is that, while more complex, automatic proof strategies were employed earlier in the Limits Alarm proof, this as opposed to the Hysteresis proof, which was done more completely by hand, and as a result is a longer proof. All PVS files, including translator output, theorems, and proofs are available at [10].

VI. Tool Support: A Monadic BSV-to-PVS Translator

The semantic transformation of BSV into PVS, described in §III, has been implemented in the BAPIP software tool. The tool is written in Haskell, using the Parsec parser-combinator library. BAPIP is a command-line tool developed for use in a Linux/Unix environment.

Parsec was selected over Happy [15] as a parsing library for this project for a number of reasons. Unlike Happy, Parsec allows seamless integration with Haskell functions, making it more flexible. This increased flexibility allowed for more advanced parsing algorithms, such as custom permutation parsing. Fig. 4 is a structural overview of the current BAPIP translator.

BAPIP has two modes, "BSV2PVS" and "BSV2BSV." The primary purpose of "BSV2BSV" mode is to test the BSV parser, but could be used for future expansion of the translation algorithm.

A. Applicability of Results

While we further the goal of automatic verification of BSV descriptions, it is important to discuss the limitations of this methodology. For example, the module we wish to

Condition	**QH**
$X > H$	True
$(H - EPS) \leq X \leq H$	NC
$X < (H - EPS)$	False

Condition	**QL**
$X < L$	True
$L \leq X \leq (L + EPS)$	NC
$X > (L + EPS)$	False

Condition	**Q**
$QL \vee QH$	True
$\neg(QL \vee QH)$	False

assuming $(EPS/2) > 0$

Fig. 3: Limits Alarm Tabular Specification for QH, QL, and Q

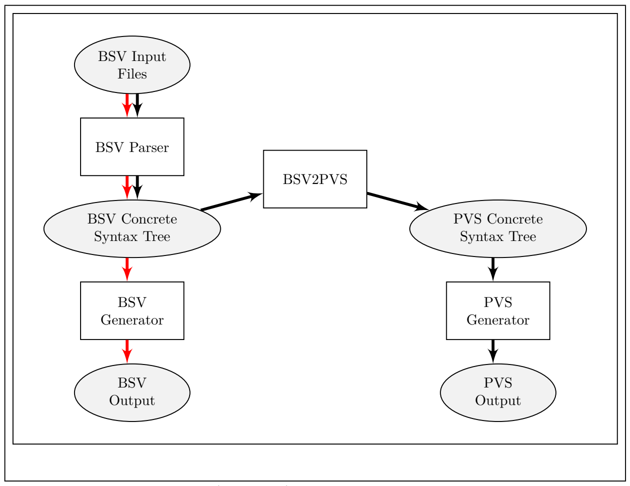

Fig. 4: BAPIP Architectural Overview

verify must be implemented in Bluespec SystemVerilog. Fortunately, Logical Equivalence Checkers (LECs) can address this deficiency. LECs are off-the-shelf tools which analyze two hardware descriptions for black-box logical equivalence. Our verified implementation of the Hysteresis function block, a submodule of the Limits Alarm function block discussed in our case study, has been successfully LEC checked against an independent implementation in VHDL. Our proof of correctness of the BSV block is transferable to the VHDL implementation via the transitivity of logical equivalence.

The automated model extraction performed by BAPIP is not total over Bluespec SystemVerilog. It is our position that the subset of Bluespec SystemVerilog for which our software operates can express meaningful hardware descriptions. Improvements are also planned which will expand the expressivity of the accepted subset of BSV. It is also important to note that the translation is not injective into PVS, so a future reverse translation for the purposes of round-trip engineering would not be able to exactly reconstruct the original BSV file. This could be remediated by using the original BSV source to supplement deficiencies. The reverse translation would also only operate on the set of PVS files that are generatable by the translator, not general PVS specifications.

Bluespec cannot exactly duplicate all functionality expressible in lower-level hardware description languages. The Bluespec compiler adds hardware elements to create properties such as rule scheduling and atomicity, but lower level descriptions do not have these restrictions. It is therefore possible for a hardware description in a lower-level language to conform to the requirements set out by Pang et. al. [11], and contain implementation details that can not be replicated in BSV.

For the purposes of deciding formal logical equivalence, however, implementation details of this kind are not a problem, as logical equivalence is defined by two modules producing the same output for the same input (i.e., black-box equivalency). LEC checking is not perfect, however, and can produce false negatives [16]. In such cases, constraints can be placed on the design to ignore failing states, but always at the risk of introducing a false positive test of equivalence.

VII. Related Works

FPGA and ASIC designs are increasing being used in safety-critical applications. Significant scientific effort has been directed toward the mathematical verification of these design languages.

Blech and Biha [17], [18] propose formal semantics for SFC, FBD, LAD and IL, and implement a partial automation tool for translation into COQ, with the goal of certified compilation/interpretation. While this technique would only require a certified PLC language to HDL compiler to complete the tool-chain, the actual proving of formalized properties in COQ is non-trivial, and best performed by someone familiar with the system, despite a degree of automatic tactic generation. In contrast, the BAPIP process is intended to require no more training in formal methods than is required to create formal requirements for the modules under examination. Another hardware verification strategy targeting COQ, though not PLC languages, is Featherweight Synthesis (Fe-Si) [19]. Similarly to BAPIP, this tool translates a subset of Bluespec SystemVerilog to a proof environment, in this case COQ. Similarly to [17], verification is dependant on the advanced skills required to operate in a dependently typed proof environment. Fe-Si is presented as a proof-of-concept, and expansion to a more practical subset of BSV is cited as a goal of the project. Vijayaraghavin et. al. present a prototype embedding of Bluespec in COQ [20], similarly to the Richards and Lester paper on which our embedding in PVS is based. They present a far more complex example than Richards and Lester, a multi-core shared memory system, and indicate that they are working on a software tool for automating this translation process. The example they present is beyond the current capability of our technique, but automatic verification of such complex systems is a long-term goal of the project.

A similar scheme to [4] has been proposed by Oliver [21], with one interesting inversion. The proposed syntactic transformation originates in the proof language B and terminates in Bluespec SystemVerilog. Similarly to [4] this is a proposed transformation only, and no translation tool has yet been implemented based on this specification. Singh and Shukla also present a verification framework for BSV [22]. Focusing on the satisfaction of temporal properties, the SPIN model checker is employed to determine whether a BSV implementation faithfully implements a corresponding semantic.

Efforts by the McMaster Centre for Software Certification have also contributed to the production of pre-verified PLC function block libraries for FPGA-based platforms. Specifically, Pang et. al. present a formalization of the requirements of the PLC function blocks of the IEC61131-3 specification [14], using tabular expressions in PVS [11]. Certain ambiguities, assumptions and inconsistent implementations within the IEC61131-3 specification are also discussed.

VIII. Conclusion

We have demonstrated a semi-automated toolchain for the verification of BSV hardware descriptions against function requirements in the form of tabular expressions, as well as associated consistency theorems. The BAPIP software tool allows the automated translation of BSV programs into the specification logic of PVS, and a technique for using the generated PVS theories is clearly elaborated. This application of traditionally software-oriented techniques to hardware description also highlights a growing grey-area between software and hardware.

By leveraging the methodologies presented here, the application of formal methods to modern hardware design becomes more feasible. Future work will focus the creation of a formally verified library of IEC-61131-3 compliant function blocks targeting FPGAs, as well as verifying more complex examples.

IX. References

[1] "bluespec.com :: View forum - software releases," 2017, http://bluespec.com/forum/viewforum.php?f=17&sid=93b12aca2328a68dcc72c17ec094ca27.

[2] "Pvs specification and verification system," 2017, http://pvs.csl.sri.com.

[3] R. Nikhil, "Bluespec system verilog: efficient, correct rtl from high level specifications," in *Proceedings of the Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004*. IEEE, 2004, pp. 69–70.

[4] D. Richards and D. Lester, "A monadic approach to automated reasoning for bluespec systemverilog," *Innov. Syst. Softw. Eng.*, vol. 7, no. 2, pp. 85–95, Jun. 2011. [Online]. Available: http://dx.doi.org/10.1007/s11334-011-0149-0

[5] S. Owre, J. M. Rushby, and N. Shankar, "Pvs: A prototype verification system," in *Automated Deduction-CADE-11*. Springer, 1992, pp. 748–752.

[6] S. P. Miller and M. Srivas, "Formal verification of the aamp5 microprocessor: A case study in the industrial use of formal methods," in *Industrial-Strength Formal Specification Techniques, 1995. Proceedings., Workshop on.* IEEE, 1995, pp. 2–16.

[7] A. Wassyng, M. S. Lawford, and T. S. Maibaum, "Software certification experience in the canadian nuclear industry: lessons for the future," in *Proceedings of the ninth ACM international conference on Embedded software.* ACM, 2011, pp. 219–226.

[8] "Training resources - learning bluespec," 2017, http://wiki.bluespec.com/Home/Training-Resources.

[9] *Bluespec^{TM}SystemVerilog Reference Guide*, Bluespec Inc., 2012.

[10] N. Moore, "BAPIP project homepage," 2017, http://www.cas.mcmaster.ca/~moorenc/bapip.html.

[11] L. Pang, C.-W. Wang, M. Lawford, and A. Wassyng, "Formalizing and verifying function blocks using tabular expressions and pvs," in *Formal Techniques for Safety-Critical Systems.* Springer, 2013, vol. 419, pp. 125–141.

[12] A. Camilleri, M. Gordon, and T. F. Melham, *Hardware verification using higher-order logic.* University of Cambridge, Computer Laboratory, 1986.

[13] C. A. Muñoz, "Batch Proving and Proof Scripting in PVS," 2007. [Online]. Available: http://shemesh.larc.nasa.gov/people/cam/publications/NASA-CR-2007-214546.pdf

[14] IEC, *61131-3 Ed. 3.0 en:2013: Programmable Controllers — Part 3: Programming Languages.* International Electrotechnical Commission, 2013.

[15] "Happy: The parser generator for haskell," 2017, http://www.haskell.org/happy/.

[16] M. Turpin, "Solving verilog x-issues by sequentially comparing a design with itself. you ll never trust unix diff again!" SNUG, 2005.

[17] J. O. Blech and S. O. Biha, "On formal reasoning on the semantics of plc using coq," *arXiv preprint arXiv:1301.3047*, 2013.

[18] ——, "Verification of plc properties based on formal semantics in coq," in *Software Engineering and Formal Methods.* Springer, 2011, vol. 7041, pp. 58–73.

[19] T. Braibant and A. Chlipala, "Formal verification of hardware synthesis," in *Computer Aided Verification.* Springer, 2013, vol. 8044, pp. 213–228.

[20] M. Vijayaraghavan, A. Chlipala, N. Dave *et al.*, "Modular deductive verification of multiprocessor hardware designs," in *International Conference on Computer Aided Verification.* Springer, 2015, pp. 109–127.

[21] I. Oliver, "A demonstration of specifying and synthesising hardware using b and bluespec," in *Forum on Design Languages FDL'06*, 2006.

[22] G. Singh and S. K. Shukla, "Verifying compiler based refinement of bluespec^{TM}specifications using the spin model checker," in *Model Checking Software.* Springer, 2008, pp. 250–269.