Noname manuscript No.
(will be inserted by the editor)

On Embedding a Hardware Description Language in Isabelle/HOL

Wilayat Khan[⋆,1] · **David Sanan**[2] · **Zhe Hao**[3] · **Liu Yang**[2]

Received: date / Accepted: date

Abstract In order to define executable hardware description language while at the same time be fit for formal proofs of properties, a hardware description language VeriFormal, embedded in Isabelle/HOL, was created. VeriFormal, together with a translator and Isabelle/HOL proof facility, provides a platform for designing, simulating and reasoning about hardware designs. Building such an environment is challenging due to the fact that the designer must have expertise in programming language design, the specific domain and theorem prover. It requires selection of a language design criteria, host language, grammar, embedding approach and techniques and mechanisms to address determinism and termination issues. When the language in question is a hardware description language, it requires specialized treatment of events, their scheduling, data types and assignments. In this paper, we report on our experience of embedding hardware description language VeriFormal in theorem prover Isabelle/HOL. In particular, the structure and execution of programs in the context of theorem provers and their impact on the overall language design are discussed. Among the main features of VeriFormal include formal semantics of the language, support for mechanical reasoning about designs and compiler and type checking of modules using Isabelle/HOL as well as VeriFormal type checkers.

Keywords Hardware description languages · domain-specific languages · Isabelle/HOL · VeriFormal · formal verification

[1]COMSATS University Islamabad, Wah Cantt, Pakistan
E-mail: wilayat@ciitwah.edu.pk
[2]Nanyang Technological University, Singapore
E-mail: yangliu@ntu.edu.sg
E-mail: sanan.baena@gmail.com
[3]Griffith University, Australia
E-mail: z.hou@griffith.edu.au ⋆Corresponding author

1 Introduction

To verify hardware circuits and systems, one proves the correctness of the implementation with respect to some formal specification. Due to the size and complexity of digital systems, they are described in terms of functions using higher-level mathematical notations such as Hardware Description Languages (HDLs). Most of the HDLs, such as Verilog [54] and VHDL [4], do not have formal semantics and such a *loosely defined semantics* [49] restricts them only to simulation. While simulation-based verification is common in the industry to show presence of errors, it fails to guarantee their absence [49]. Furthermore, a property P of the design under test D defined in Verilog can be tested by simulating only against limited number of inputs: it is computationally infeasible to simulate the design for all possible inputs (2^{256}) when each input is 256-bits wide.

There is a semantic gap [24] between such HDLs and the logic used in the formal verification and hence they cannot be used to mathematically prove properties of the designs described in them. Hardware circuits can directly be modelled using other mathematical logic notations [30,32], but this approach is not acceptable to many designers and computer tools such as simulators [10]. To fill this gap, a *semantic approach* [10] is taken: a formal model of hardware description language is built in a theorem proving environment such as Isabelle/HOL. The formal language, ideally defined in a proof assistant, enables semi-automated verification of hardware designs. The HDL VeriFormal [36] is such a language with formal semantics embedded in the proof assistant Isabelle/HOL [47].

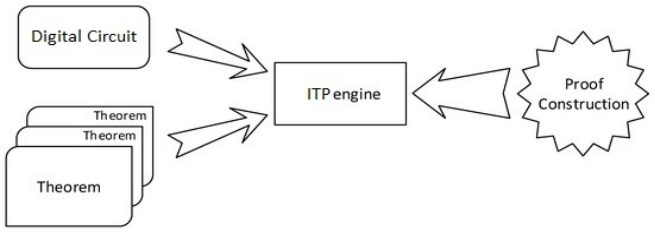


Fig. 1: Formal verification of digital circuits [28]

The formal approach to checking correctness of digital circuits is described in the Figure 1. Formal models of the digital circuit under verification and the properties of interest as theorems (both described in VeriFormal) are fed into an ITP engine (Isabelle/HOL in our case) and a mathematical proof that the (model of the) circuit holds the properties is carried interactively. Unlike simulation, the property P of the design D defined in an HDL with formal semantics can be mathematically proved, against all possible inputs, using the

logic behind the HDL (see an example of proof of property against all possible inputs to a program in Figure 2).

The semantic approach of embedding an HDL in a theorem prover is useful in many ways [10]. It gives formal, and thus precise, semantic definitions of various notations. Through embedding in Isabelle/HOL, mechanical support for HDL Verilog, including mechanical checking of proofs, is provided. The *type checker* of the theorem prover is used to statically check the syntax and types of the programs using a computer. Using the proof facility of the theorem prover, formal proofs about the classes of programs and constructs can be carried and checked mechanically. Furthermore, if the semantics of the language is developed in a functional style, designs can be executed thereby allowing designs simulation. For this later purpose, an executable simulator can be automatically extracted using code generation facility of Isabelle/HOL. Additionally, the semantic approach can be used as a step towards verification of compilers (e.g., Verilog simulators).

Defining an HDL with formal semantics has numerous benefits as described above, though, there are various challenges to address before availing these benefits. The first question that arise is whether to define a formal semantics for all or a carefully chosen subset of notations and constructs of the language. Considering the amount and frequency of work required to extend constructs and carry proofs, an approach for semantic embedding based on *deep* or *shallow* embedding [26,55] is chosen. HDLs consist of sequential as well as concurrent constructs with the later being more challenging to dealt with while defining the formal semantics. In particular, language constructs that either do not terminate or terminate in non-trivial fashion are difficult to formalize in languages that accepts only total functions. To ease programming in the new language, its syntax should be kept in a natural style with syntactical structure similar to other known languages (e.g., Verilog). In this research work, the challenges confronted while embedding the domain-specific language VeriFormal in a theorem prover and approaches taken to address them are discussed.

This paper is an extended version of our previous work on VeriFormal published as a conference paper [36]. It highlights the syntax and semantic challenges faced when embedding VeriFormal in Isabelle/HOL. In other words, the conference paper described syntax and semantics (*what* part) of the language while the existing paper focusses on *how* and *why* different approaches were taken to address the challenges encountered. The major contributions of this paper are the following:

- The checker predicate of VeriFormal is extended with an additional check on *always* blocks (Figure 17).
- The meta-programming construct, *generate loop* statement, is added to the VeriFormal translator to support parametric designs.
- Two proof examples, one interactive proof and one type checking, in Isabelle/HOL are added (Section 3). The examples are simple, though, they highlight the significance of embedding HDL in a theorem prover.

– An in-depth analyses and discussions on language design criteria, host language, context-free grammar, embedding approach, bit-vector assignment, event formation and scheduling are included (Sections 4 and 5).
– The peculiarities of formal description of (non)-deterministic behaviour, non-terminating and non-trivially terminating processes are presented with examples (Section 6).
– The impact of designing a separate type checker on the language design and correctness of programs are discussed (Section 7).

Readers are recommended to refer to the paper [36] and source codes at link `https://github.com/wilstef/veriformal` for detailed description of language syntax and operational semantics.

The rest of the paper is organized as follows. In next section, an introduction to HDL VeriFormal, formal specification and proving in Isabelle/HOL is given. The creation of VeriFormal is motivated in Section 3. The language design criteria and embedding approach are discussed in Section 4. The syntax challenges faced while embedding VeriFormal are discussed in Section 5. A detailed discussion on formalizing non-deterministic and non-terminating behaviours are given in Section 6. The syntactical complexity distribution among the embedded language, a separate type checker and translator is described in Section 7. Few examples of formal verification using VeriFormal are included in Section 8. A summary of the related work is given in the Section 10. The paper is concluded in Section 11.

2 Background

VeriFormal is a formal version of DSL Verilog, embedded in the higher order logic of theorem prover Isabelle/HOL as the host language. In this section, formal specification and verification using theorem proving approach is briefly introduced. Furthermore, the syntax of VeriFormal is described and compared with the syntax of Verilog. This section also highlights significance of formal methods in designing domain-specific languages.

2.1 Formal specification and verification using theorem proving

To formally reason about systems, a formal model of the system and the property of interest is built in the logic of a theorem prover (such as Coq [7] and Isabelle/HOL [47]). The proof facility of the theorem prover is used to carry a proof that the (model of the) system holds the property and the proof checker of the tool is used to mechanically check if the proof is valid. For a system (e.g., programming language) to be more 'prover friendly', it must be consistent to the logic of the prover. If it is not, the system is re-defined in the logic of theorem prover which enables one to carry proofs about the system using the logic of the tool. Theorem provers are similar to other programming languages, however, in addition to programming, they can be used to reason

about programs defined in the logic of theorem prover. Another approach to formal verification is based on model checking [16] and is out of scope of this paper.

```
1  theory natproof
2    imports Main
3  begin
4   datatype nat = zero | S nat
5   primrec add :: "nat ⇒ nat ⇒ nat" where
6      "add zero m = m"
7    | "add (S n) m = S (add n m)"
8   lemma add_m_o: "add m zero = m"
9     apply (induct m)
10    apply auto
11    done
12 end
```

Fig. 2: Interactive proof in Isabelle/HOL

To describe formal developments and proofs using theorem prover, a simple system of numbers is defined and reasoned about using the tool Isabelle/HOL. To begin with, numbers are inductively defined as data type `nat` using the keyword `datatype` with two constructors for generating elements of the type `nat` (line 4, Figure 2). The definition **nat** states that `zero` is `nat` and if `n` is `nat` then `S n` is also `nat`. The term `S (S (S zero))`, for example, is a number (3) in `nat`.

Next we define a recursive function `add` (lines 5–7) on the numbers just defined. The function returns the second argument `m` if first argument is `zero` and it returns `S (add n m)` if first argument is of the form `S n`. A lemma `add_m_o`, that `add m zero = m` holds for any value of `m`, is stated and proved in Figure 2 (lines 8–11). The lemma is proved using induction on the construction of `m`. During the proof process, the Isabelle/HOL tool is guided interactively by providing commands called *tactics* (lines 9–11).

2.2 Introduction to VeriFormal

VeriFormal is a formal version of Verilog, embedded in the proof assistant Isabelle/HOL. The syntax of VeriFormal include formal definitions of expressions, statements and top statements. Context-free grammars `exp`, `statement` and `top`, respectively, are used to define these constructs. An example VeriFormal module is listed in Figure 3 (right), side-by-side with equivalent description as Verilog module (left). Among the major syntactic differences between Verilog and VeriFormal constructs are the following. In VeriFormal,

– every term is preceded by a constructor name (directive) to help host language compiler in parsing (e.g., $_b$ t_1 [+] t_2 to model binary operation +

```
1  module addint(Xi, Yi);
2   input [7:0] Xi, Yi;
3   wire [7:0] X, Y;
4   reg [7:0] Z;
5
6   assign #0 X = Xi;
7   assign #0 Y = Yi;
8
9   always #2
10    begin
11        Z = X + Y;
12
13        $finish;
14    end
15 endmodule
```

```
1  module ([Xi, Yi])
2  [ input [7:0] [Xi,Yi],
3    wire [7:0] [X,Y],
4    reg [7:0] [Z],
5
6    assign #0 [nX] = nXi,
7    assign #0 [nY] = nYi,
8
9    always ( [#]2
10    BEGIN : [None]
11    ([nZ] [=] [#] -1
12    (b nX [+] nY));;
13    ($finish)
14    END) ]
15 endmod
```

Fig. 3: Hardware descriptions in Verilog (left) and VeriFormal (right)

on terms t_1 and t_2). The other constructor in the Figure 3 is $_n$ (for names when used as terms),

- related sequence of identifiers or terms are grouped in lists. Identifiers in the input port (line 1) and declarations (lines 2−4) and terms in assignments (lines 6, 7, 11) are combined into lists []),
- operators are enclosed in brackets to avoid conflict with Isabelle/HOL notations (e.g, [+]), and
- statements within a `BEGIN...END` block are separated by double semicolons ';;' (line 12, right, Figure 3) and enclosed in parenthesis to help Isabelle/HOL in disambiguating sentences during parsing.

A VeriFormal module is of the form `module (P) T endmod`, where `P` is a list of input ports and `T` is a list of top statements separated by comma ',' (right, Figure 3). The name of module is skipped in the VeriFormal module definition (line 1, right, Figure 3) and is later added as name of a definition (of type `program`) using the Isabelle/HOL keyword `datatype` (see below). All the top statements (declarations, continuous assignments and always blocks) in a module are grouped into a list separated by comma ',' which together with `module...endmod` and input ports form a `program` in VeriFormal. The `[None]` (line 10) is an optional code block name, `-1` (line 11) is delay (no delay) and parenthesis are used for disambiguating terms. Finally, the module is included in a definition of type `program` to build the design. The VeriFormal module in the Figure 3, for example, is defined as a program and simulated using `simulate` function as the following

```
definition addint :: program where "addint = module_definition"
value "simulate 2 (fedinput [(Xi, 2), (Yi, 5)] (state addint))"
```

The Isabelle/HOL command `value` evaluates the term following it. The function `state` creates a tuple of events from the design (program) and the `fedinput` initializes the two input port variables with values 2 and 5, respectively. The state, together with two values is simulated for two simulation cycles and the updated state (with updated values of variables) is returned. To simulate a design for all possible inputs, a separate interface program (written in C++) feeds all the inputs into the simulator, runs it and stores the result in a file. For more details about VeriFormal syntax, state creation, operational semantics and simulation, readers are recommended to refer to [36].

2.3 Formal methods based embedded DSLs

Internal DSLs, defining a DSL by embedding it in a host language, are equally popular as external DSLs among the research community (47.8% and 52.2%, respectively, as shown in the Table 1). Unlike general purpose programming languages, domain-specific languages constructs are simple, well-defined and offers feasible facilities for analysis and verification in terms of DSL constructs [45]. There are a number of widely used DSLs such as Verilog, VHDL, MATLAB and SQL, just to name a few, however, only limited number of DSLs uses formal approaches for describing syntax and semantics [37].

Table 1: Research distribution in DSL [37].

Approach	**Percentage**	
Internal	47.8	
External	52.2	
	Design	Domain analysis
Formal	16.8	5.7
Informal	83.2	94.3

The research community has widely studied both the design and domain of DSLs, though, very few has used formal approaches in the design and domain analysis (16.8% and 5.7%, respectively). According to the systematic analysis carried by Kosar et al. [37], many researchers (about 48%) have used the internal (embedded) approach in DSL implementation, however, the authors identified there is a clear lack of formal methods within domain analysis and semantic description of DSLs. This emphasise the adoption of formal methods to the design and domain analysis of DSLs. VeriFormal, is an embedded implementation of a DSL entirely in the higher-order logic of formal language Isabelle/HOL. For an in depth analysis and visual comparison of literature on the implementation approaches of DSLs, readers are advised to see references [37] and [45].

3 Yet Another HDL?

The first question that arise is, if there exists popular and well-established HDLs, why do we need yet another HDL VeriFormal? VeriFormal is an HDL with formal foundation that, a) includes type checkers to check hardware descriptions for type errors and supports mathematical reasoning, and b) the executable simulator is automatically created using trusted code generation facility.

3.1 An HDL with formal foundation

The main objective of VeriFormal is to enable hardware designers to design hardware, simulate and then carry mathematical proofs about the language itself, a particular design as well as a class of hardware designs. The first two facilities, namely hardware design and simulation, are available in existing HDL Verilog [54] and VHDL [4], however, the last facility requires the language to have a mathematical foundation. As none of these languages have formal foundation, it motivated the creation of HDL VeriFormal.

```
definition mulbyprod:: "int ⇒ program" where
 "mulbyprod x =
  module ([])
   [ wire [3:0] [R1],
     assign #-1 [nR1] = b v(x,4) [*] v(2,4) ]
  endmod"
```

Fig. 4: Multiplication using product operator

```
definition mulbyshift:: "int ⇒ program" where
        "mulbyshift x =
         module ([])
          [ wire [3:0] [R1],
            assign #-1 [nR1] = ((v(x,4)) [<<] 1) ]
         endmod"
```

Fig. 5: Multiplication using shift operator

```
1  lemma mul2equiv [simp]:
2    "simulate 0 (fedinput [] (state (mulbyshift 3)))
3      = simulate 0 (fedinput [] (state (mulbyprod 3)))"
4   apply (simp add: stepsim_def mulbyshift_def mulbyprod_def
5    mkconfig_def initconfig_def state_def processba_def
6    updateconfig_def nextcycleb_def nextpassb_def
7    updatevar_def slicebv_def maskn_def binopbv_def Let_def
8    shiftl_int_def bvlsn_def)
9   done
```

Fig. 6: Proof of equivalence of multiplication using product and shift

To highlight the prevailing difference between conventional HDLs and VeriFormal, a simple example is described in Figures 4–6. The module `mulbyprod` in Figure 4 multiplies a value (of type `integer`) by 2 using the product $\star$ operator and the same operation is implemented using left-shift operator in module `mulbyshift` (Figure 5). The lemma `mul2equiv` in Figure 6 states that both modules are functionally equivalent for a specific input value 3. The Isabelle/HOL tactics (commands) on lines 4–9 unfolds the functions involved and closes the mathematical proof. The lemma `mul2equiv` is specific but it can be extended to a general lemma for any integer. In addition to theorem proving, all the designs described in VeriFormal are type checked using Isabelle/HOL and VeriFormal type checkers. Any design written in VeriFormal is type checked by the Isabelle/HOL type checker (part of compiler) and only well-typed modules are accepted. The designs, however, are additionally checked for conformances to best practices. See Section 8 for other examples of type checking and reasoning about VeriFormal terms and modules.

3.2 Trusted simulator

The VeriFormal executable simulator in OCaml is automatically generated from Isabelle/HOL definitions using the `export_code` command. The correctness of the code generation process is required for the correctness of the simulator. The code generator in Isabelle/HOL has been formally verified correct [29,33] in a purely proof theoretic way using higher-order rewrite systems. The proof of correctness of the code generation facility in Isabelle/HOL ensures the generated simulator exactly corresponds to the Isabelle/HOL definitions.

4 Deep Embedding in Theorem Prover

Unlike general purpose programming languages, domain-specific languages are tailored towards more specific applications such as hardware description. Using a DSL for a particular domain, one can develop programs more quickly and effectively as compared to general purpose languages. Programs written in a DSL are concise, easier to maintain and reason about [31]. To implement a DSL, a natural question arise about the language implementation approach:

develop a separate DSL ecosystem (external) or embed the DSL in a general purpose programming language (external).

4.1 Internal or external embedding

According to the study carried by Cuadrado et al. [20], the internal (embedded) implementation approach is superior to the external approach. Cuadrado et al. compared internal and external implementations of two model transformation languages RubyTL and Gra2MoL. They found that internal approach requires almost 30% less effort in terms of lines of code as compared to external approach. Furthermore, their study suggests that internal language is easier to learn, cost effective, rich in features and easy to modify and experiment with. The most important motivation towards embedding approach is to avail the mechanical proof facilities available in the existing host languages, such as Isabelle/HOL. With these advantages in mind, we adopt the internal approach for the implementation of our DSL VeriFormal in Isabelle/HOL.

VeriFormal was embedded in Isabelle/HOL by writing 1232 lines of code (the translator additionally required 2148 lines of code). Had it been implemented as a standalone language, the study of Cuadrado et al. [20] suggests, it would require about 1600 lines of code (1232 + 0.30*1232). One might argue that writing these additional number of lines is not that much effort. An enormous amount of effort, though, would be again required to encode each description in the higher order logic of a theorem prover to enable mechanical reasoning.

4.2 Host language selection

In the external approach, a *standalone* DSL is developed with its own custom syntax and semantics and standard compiler design techniques are used to translate programs written in the DSL to a target language. This approach provides a separate entire language ecosystem including its own editor, compiler and debugger. This approach may be modified in several ways [31]: use computer tools Lex and Yacc to automatically generate lexer and parser or write an interpreter rather than a compiler, just to name few. This design approach requires significant work to implement the language from the scratch and document it [52, 26, 31]. To inherit the infrastructure and facilities of an existing language tailored towards a domain of interest, an embedded approach to DSL is taken.

The standalone approach is popular among the object-oriented community [23, 26] while the *embedding* approach is common among functional programmers [31, 26]. It appears that the functional programming features such as higher order functions extremely facilitate the embedding approach [26, 25]. Embedding a DSL in a conventional language enables one to execute programs written in the embedded language, however, it does not provide any

facility to reason about the programs. The HDLs Chisel [6], PyMTL [41], MyHDL [22] are the examples of such languages embedded in Scala and Python. The execution facility can be used to simulate programs (e.g., circuit designs) by providing input vector, however, this feature alone may hinder run time limitations [40], in particular when the simulation is exhaustive. Embedding HDL in a theorem prover enables one to reason about circuits described in the embedded DSL. Furthermore, if DSL is embedded in a functional style, in addition, it would provide the execution facility. Keeping in mind these prevailing benefits, the embedding approach with theorem prover Isabelle/HOL as the host language is chosen for embedding VeriFormal. The higher order logic of Isabelle/HOL can be used to prove theorems about the programs (see Figure 6) and as VeriFormal has been defined in functional style, programs can be simulated [36].

4.3 Deep or shallow embedding

When embedding a DSL in a theorem prover such as Isabelle/HOL, the designer has to choose from *deep* or *shallow* embedding. When deep embedding a DSL, the logical formulas in the language are defined as data types in the (higher order) logic of theorem prover (Isabelle/HOL in this case) and interpretation functions are defined to assign semantics to the algebraic data types. On the contrary to deep embedding, in shallow embedding, the language is represented directly with semantics in the logic of theorem prover [26]. In this latter embedding style, VeriFormal formulas would become predicates (on states) in Isabelle/HOL. Examples of shallow and deep embeddings in the higher order logic of HOL theorem prover are given in [11] and [50], respectively. Both, deep and shallow, embeddings have pros and cons. In deep embedding, abstract syntax trees represented as data types complicate language extension: adding a construct would require changes to both the syntax tree and all functions manipulating the tree. In particular, the interpretation functions in the semantics need to be updated to interpret the semantic meaning of the type constructors added or updated. On the other hand, shallow embedded language is easily extensible as long as language constructs are represented in the semantic domain [53]. VeriFormal is a replica of DSL Verilog and rarely requires syntax extension, making the extensibility in deep embedding less challenging. Furthermore, to reduce or more importantly modularize the effort required to extend the language constructs and interpretation functions, VeriFormal is augmented with a type checker predicate (Section 7) and a translator [36]. The grammar productions are grouped by introducing additional non-terminals making the constructor extension more organized and easy (Section 5).

The proof script in shallow embedding is about twice as the size of proof script in deep embedding [55, 2]. On the other hand, deep embedding allows one to quantify over syntactic structures thereby allowing to reason about classes of programs [43]. The Isabelle/HOL proof facility is aimed to prove

correctness of different designs described in VeriFormal. Considering the fact that formal proofs carried in theorem prover requires huge effort [14,9], it is important to focus on reducing the more frequently-needed proof effort than the work required for rare language extension. Furthermore, as Verilog tools are normally based on event-driven simulators, an operational-style semantics is more suitable for Verilog [10] thereby advocating deep embedding approach for VeriFormal in the higher order logic of Isabelle/HOL.

The syntactic structure of VeriFormal consists of expressions, statements, top statements, events and processes. All these major constructs are defined as data types in Isabelle/HOL using keyword `datatype` followed by functions to interpret these constructs. A function `mkconfig` gets a list of constructs (module) and creates a tuple called program state or configuration. A number of other functions are defined to evaluate the state of the language created by `mkconfig`. All these functions are combined in a single function `simulate` which gets a module (program), creates an initial state for the module and simulates it for a given number of simulation cycles.

5 Syntax Challenges

The first step in embedding a language is to precisely define the syntactical structure of the programs. The language designer has to consider among different forms of context-free grammar, data types and keep the embedded language as similar as possible to an existing language. Furthermore, assignment to (tuple of) variables in HDLs is different than in general purpose languages and is explained in this section.

5.1 On the choice of grammar

VeriFormal constructs are defined using the most widely used [45] formal notation called context-free grammar: shorthand notations for inductive definitions of non-terminals representing sets of strings. The grammar of the language must neither be ambiguous nor left-recursive. Ambiguous grammars result different interpretations of the same program for the same input while left-recursion causes the parser to get into looping. Both of these properties of the grammar are more prominent in the standalone languages, however, they are not required in languages embedded in theorem prover. The host language compiler deals with ambiguity and left-recursion in the grammar and hence are not considered while embedding VeriFormal in Isabelle/HOL.

This is further elaborated with few examples. Consider the grammar for language of expressions in Figure 7 (left).

```
exp → exp + exp
      | exp - exp
      ...
```

```
datatype exp =
  exp_sum exp exp
| exp_sub exp exp
...
```

Fig. 7: Grammar and its implementation as datatype

```
exp ⇒ exp + exp
    ⇒ exp - exp + exp
    ⇒ 3 - exp + exp
    ⇒ 3 - 2 + exp
    ⇒ 3 - 2 + 1
```

```
exp ⇒ exp - exp
    ⇒ exp - exp + exp
    ⇒ exp - exp + 1
    ⇒ exp - 2 + 1
    ⇒ 3 - 2 + 1
```

Fig. 8: Left-most and right-most derivations

This grammar is ambiguous: the string (program) "3 - 2 + 1" has two (left-most and right-most) derivations (syntax trees) and hence is evaluated to two different values 2 (left) and 0 (right), respectively (Figure 8). To remove ambiguity, disambiguation rules [3], such as operator precedence may be applied to the grammar. Domain-specific language embedded in a theorem prover has an advantage: the syntax of the program is defined using unique constructors (right, Figure 7) checked using pattern matching which is sufficient to enforce determinacy [8].

The program "3 - 2 + 1" is encoded as `exp_sum (exp_sub 3 2) 1` if it is meant to evaluate the subtraction first, otherwise, it is encoded as `exp_sub 3 (exp_sum 2 1)`[1]. The program `exp_sum exp_sub 3 2 1` is not accepted as it results a unification error: the first argument of constructor `exp_sum` must be of type `exp`. In such cases, Isabelle/HOL compiler require the programmer to insert parenthesis in the programs to disambiguate them. In any case, Isabelle/HOL compiler/type checker accepts only unambiguous programs with only one interpretation, thereby relieving the designer of the DSL (in theorem prover) to rewrite an unambiguous version of the grammar by introducing a new non-terminal (see [3] for examples). Such a refinement in the grammar is enforced in Isabelle/HOL using mutually (depended) recursive types which are relatively difficult to dealt with in proofs. Another issue with the grammar in the example is left-recursion: a grammar is left-recursive if the non-terminal on the left of arrow also precedes the body on the right. In standalone language, left-recursion is a critical issue: the parser may not return. In language embedded in theorem prover, it is not: there is a unique constructor name for each constructor (such as `exp_sum` and `exp_sub`) which helps the theorem prover compiler selecting the proper constructor (production) using pattern matching. The host language compiler does not get into looping even if the

[1] Numbers are expressions and its constructor is skipped for simplicity.

grammar is left-recursive. Unique constructors play a vital role as described in previous sections, however, they are tedious to dealt with when writing a translator for Verilog programs (Section 5.2).

5.2 Similarity with existing language

To write programs in the new DSL designed, a programmer has to learn the syntax of the language to write programs. Another option would be to use computer tools to automatically translate programs from existing known language to the embedded language. This later approach is crucial when translation of the existing programs to the new language is needed: to reason about existing Verilog designs mathematically, they need to be translated to VeriFormal. Both, translation to and programming in the embedded language (VeriFormal) is easy provided the embedded language is designed with syntax similar to an existing language (Verilog in this case). Isabelle/HOL allows to use symbolic notations for non-terminals and constructors which enables the new DSL similar in syntax to existing language. After notations are used, VeriFormal declarations are the same as in Verilog with minor differences: in VeriFormal, identifiers are enclosed in the brackets and size `[0:0]` is included for single bit variables. In general, Verilog declarations of type *var_type* $[n_1{:}n_0]$ v_m, ..., v_2, v_1, v_0 are translated to VeriFormal declarations *var_type* $[n_1{:}n_0]$ $[v_m$, ..., v_2, v_1, $v_0]$. To disambiguate VeriFormal expressions while at the same time keep its syntax similar to Verilog, constructors with symbolic notations are used: the short notation for VeriFormal term `exp_bop (exp_name x) bvsPLUS (exp_name y)` is $_b$ $_n$`x [+]` $_n$`y` which resembles the Verilog term $x + y$.

As observed, constructor notations help in disambiguating implementation of the ambiguous grammars, however, such notations must be inserted in proper places to create equivalent terms in the target language making direct programming and translation into VeriFormal tricky. To translate the Verilog expression $x + y$ to an equivalent VeriFormal expression $_b$ $_n$`x [+]` $_n$`y`, it requires enclosing + in brackets and insertion of constructor subscripts $_b$ and $_n$ in proper places.

5.3 Data type of values

In HDLs, the most common data type is the sequence of logic values representing signals on wires (e.g., buses) or storage units (e.g., registers). To represent these sequences of logic values, Isabelle theories provide the facility to define word type of specific word length [21]. Through these theories, the embedded language can use the host language (Isabelle/HOL) built-in operations on words (bits) and all the existing theorems and lemmas defined on words can be exploited while reasoning about the language. Unfortunately, word type definitions with arbitrary length has to be explicitly defined. In HDLs such as Verilog, it is common to design circuits with various word sizes (e.g., buses

with width 8, 16, ...) and defining word types of each size is tedious. An alternative is to use single but general type bit-vector [44] for all words of any length. A bit-vector is a pair of the form (v, s) where the value v is an integer and the size s is a natural number. There are downsides of bit-vectors, though, that the Isabelle/HOL theorems and lemmas defined on words can not be applied to bit-vectors and the operations on these vectors need to be re-defined. Luckily, a partial support exists in Isabelle theories: theories such as `Groups`, `Divides`, `Bits` and `Orderings` [47] support arithmetic, bit-wise and relational operations directly on integers, which are extended with some effort to operations on bit-vectors.

As an example, the relational operation `less-than` on bit-vectors $\mathtt{v}_1$ and $\mathtt{v}_2$, is defined as the following: $\mathtt{booltobv(fst(v_1)} < \mathtt{fst(v_2))}$. The functions `fst` returns the first (integer) value of the vector pair and `booltobv` converts the boolean result to a bit-vector of length one. The symbol $<$ is a shorthand for calling `less` function defined in Isabelle/HOL theory `Orderings`.

5.4 Assignment to tuple

Verilog allows updating more than one variables in a single assignment statement and the same has been formalized in VeriFormal. Assignment with one identifier on the left is straight forward: the identifier name and bit-vector pair is registered as an update event and added to the events store. Formal definition of assignment to a tuple of identifiers is tricky. A simple Verilog assignment statement involving tuple of identifiers on the left is given below followed by equivalent VeriFormal statement and described pictorially in Figure 9.

assign $\{id_m,...id_2,id_1,id_0\}$ = exp_{bv}
`assign #-1 [`$_n\mathtt{id}_0$`, `$_n\mathtt{id}_1$`, `$_n\mathtt{id}_2$`,...,`$_n\mathtt{id}_m$`] = `*exp_{bv}*

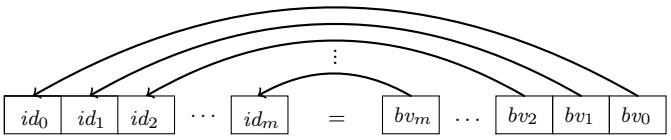


Fig. 9: Bit-vector assignment to tuple

The body of assignment exp_{bv} evaluates to a bit-vector bv and is assigned to $m + 1$ identifiers on the left. Part of the bit-vector, equal to the width of identifier, is assigned to each identifier in the tuple. Assignment of these sub-vectors happens in a natural style in Verilog: right-most part of the vector bv is assigned to right-most identifier in the tuple. In VeriFormal, though, things are

different: the tuple is modelled as a list. In this setting, the right-most identifier in the Verilog statement is the left-most in VeriFormal statement as shown in the example statement. In VeriFormal assignment statement, the assignment occurs as the following: the right-hand side of the statement is evaluated to a bit-vector bv which is split into $m-1$ slices where slice bv_i is equal to the width of identifier id_i. In other words, the assignment is divided into $m-1$ sub-assignments of the form $id_i = bv_i$ as shown in the Figure 9. Finally, each assignment is executed by registering an update event which upon execution adds (or updates) a binding (id_i, bv_i) to the environment in the store.

5.5 Inductive or recursive definitions

While defining the semantics of a language in Isabelle/HOL, many formal definitions such as interpretation functions can be expressed as a recursive function as well as inductive definition in Isabelle/HOL. Recursive functions support evaluation and hence are ideal for executable semantics while inductive definitions are suitable for conducting proofs, in particular, when case analysis is needed. The interpretation functions of VeriFormal have been defined as recursive functions to ease executing (simulating) VeriFormal designs while at the same time can be used for conducting proofs. In Isabelle/HOL, there is an automatic tool [39] to extract recursive function from inductive definition. This motivates inductive definition (for conducting proofs) of the interpretation functions and then automatically extracting recursive definitions for evaluating programs. This is an interesting domain and worth studying and is left as a future work.

6 Semantic Challenges

To assign semantic meaning to constructs as defined by the syntax of the language, an operational semantic is defined to execute VeriFormal modules. VeriFormal allows the programmer to choose the order of execution (in a deterministic way) of concurrent processes and define non-terminating constructs using guard conditions. Starting from events scheduling, formal definitions of (non-) deterministic and non-terminating behaviours are described in this section.

6.1 Scheduling VeriFormal events

To execute programs written in VeriFormal, an interpreter function `simulate` [36] is defined in Isabelle/HOL. A function `mkconfig` of the interpretor takes a VeriFormal module (list of top statements) and creates a state: tuple of event stores, an environment, simulation cycle, finish flag and set of disabled names. Executions starts with an empty state where event stores, environment and set of disable names are empty, initial cycle is 0 and the finish flag value is `false`.

The environment of the initial state is populated from the declarations where a name-value pair entry, with initial values 0, is added to the environment for each variable in the declaration.

All other top statements in the module are mapped to a list of events (or processes) of six types scheduled for execution as shown in the Figure 10. The dotted and dashed arrows represent the effect of an event (e.g., update event triggers listening event), the solid arrows show manual transformation (e.g., converting inactive event to active event) and the loosely dashdotted arrows (after the vertical dotted line) show transformation (e.g., future event conversion to update event) in the next cycle. Simulation cycles are separated by the vertical dotted line. The dotted arrows in the Figure 10 highlight the looping and are discussed in detail in Section 6.3. A simulation cycle completes when all the events of any type (except future events) are executed.

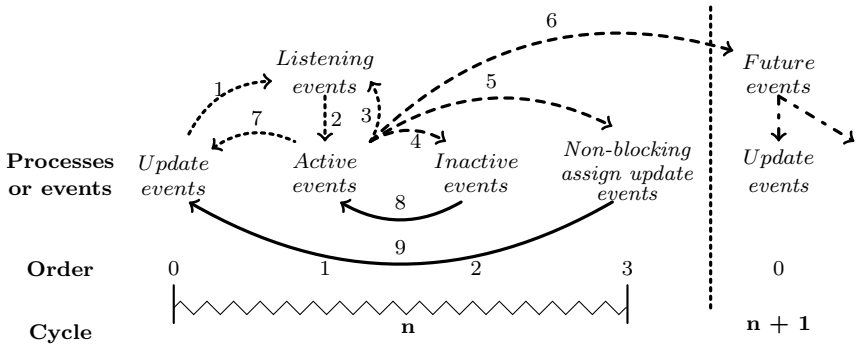


Fig. 10: Events schedule of execution

Continuous assignments without delay and without identifier in the right hand body are executed by registering update event (events if tuple on the left side). Continuous assignments without delay and at least an identifier in the body is registered as listening event. Initial block is converted to active (event) process which is executed after the update events. Similarly, statements with zero delay are converted to in-active events, non-blocking assignment statements are converted to non-blocking assign update events and statements with non-zero delay are converted to future events. Update event may trigger listening events into active events (arrows 1–2) and active event may in turn add to listening, inactive, non-blocking assign update, update or future events (arrows 3, 4, 5, 7 and 6, respectively). Inactive events are *activated* to active events (arrow 8) and then executed. Non-blocking assign update events are converted to update events (arrow 9). When all the events, scheduled for execution in the current cycle, are executed, simulation enters next simulation cycle to execute future events scheduled for execution in the new cycle (loosely dashdotted arrows).

The list of events in VeriFormal is exactly the same as found in the literature [40, 44] with minor differences in the way they are implemented. Events that are waiting for a trigger are modelled as a separate listening event in VeriFormal and [44] and then converted to active event while it is directly modelled as active event in [40]. Similarly, monitor event is explicitly included in [40] and [44] but it is implicit in VeriFormal: the `simulate` function returns the final state of execution with updated values of all the variables.

6.2 Formalizing non-deterministic behaviour

In Verilog, concurrency of hardware circuits can be modelled: there can be multiple processes scheduled for execution at the same time. When multiple processes are scheduled for execution at the same time, the order of execution is not specified by the IEEE standard [5] and hence an arbitrary (non-deterministic) order is chosen by simulators. For robust design, *linter* [34] tools may be used to detect and flag race condition cases and correct before simulation. In VHDL [4], the language definition inherently guarantees determinism by preventing the race conditions created by blocking assignments.

To model two components running in parallel (concurrent), Verilog is using continuous assignments and procedural blocks (e.g., always block). One form of non-deterministic behaviour can arise when two always blocks are scheduled for execution in parallel. The two always blocks in Figure 11 have the same sensitivity levels (both are triggered on the positive edge of x) and hence the blocking statements inside the blocks are scheduled to be executed at the same time. As software programs execute one statement at a time in serial fashion, a simulator execute the two concurrent blocks serially in an arbitrary order. This phenomenon of non-deterministic execution is called the *race condition*. Depending on the order of execution chosen by a simulator, the value of variable y will be different: for a specific value of d, the final value of y will be d if the first block is executed last and complement of d if the second block is executed last. The order of execution is not specified by the IEEE standard [5] and both results are valid.

```
1  ...
2  always @(posedge x)
3        y = d;
4
5  always @(posedge x)
6        y = ~d;
```

Fig. 11: Non-deterministic behaviour: two always blocks

To avoid the race problem, blocking assignments are replaced with non-blocking assignments [40]. Unlike blocking assignments, which are executed at the time when encountered, non-blocking assignments evaluates the value of

right-hand expression when encountered and the assignment to the left-hand side variables is performed at the end of the current simulation cycle. Non-blocking assignments can not resolve race condition in all cases: for example, the race-condition when non-blocking assignments are used inside an always block (Figure 12). As both non-blocking assignments are triggered at the same time on the positive edge of the common sensitivity, both assignments are scheduled for execution at the same time. The order at which the value is assigned to variable y is not specified and hence can result different value of y depending on the order of execution.

```
1 ...
2 always @(posedge x)
3       y <= 1'b0;
4       y <= 1'b1;
```

Fig. 12: Non-deterministic behaviour: assignments within an always block

Another unavoidable race condition arises when two blocks, with at least one has more than one assignment, are scheduled to be executed at the same time. The assignment statements from both blocks may be interleaved with one another. That is, the simulator may execute statement of the other block before finishing execution of all the statements in the current block [40]. This unavoidable non-deterministic behaviour is shown Figure 13.

```
1 ...
2 always @(posedge x)
3       y = 1'b0;
4       z = y;
5
6 always @(posedge x)
7       y = 1'b1;
```

Fig. 13: Non-deterministic behaviour: interleaving statements

The two orders of executions are given in the listings below. In both cases, the value of z will be valid but different: $z = 0$ in the first (left) case and $z = 1$ in the second (right).

```
1       y = 1'b0;
2       z = y;
3       y = 1'b1;
```

```
1       y = 1'b0;
2       y = 1'b1;
3       z = y;
```

Any order of execution is valid [5] but the results are different and depend on the order of execution chosen by a simulator. When embedding VeriFormal,

there is a choice either to keep the non-deterministic behaviour as allowed by the standard or execute concurrent constructs in the order specified by the designer. To execute concurrent constructs in a specific order, the design community often follows the best practices guidelines. Such best practices required by a language, though, are considered as a shortcoming of the language. Unlike Verilog, determinism is inherently defined in the VeriFormal interpreter which execute events in the order they happen in the code. This approach has been used in VHDL [4] and conforms to Verilog standard: the standard allows any order of execution. The difference is, in Verilog, the order of execution is chosen by the simulator while in VeriFormal, it is chosen by the programmer.

To implement concurrent behaviour in a deterministic way, VeriFormal stores listening events in a list. The data structure used for events is crucial: when a set is used, events are chosen non-deterministically and when a list is used, events are chosen in order. For the concurrent scenario described in Figure 11, VeriFormal registers two listening events and store them in a list in the store. When data type list is used, the order of execution is deterministic and is chosen by the language designer: the list is extended at tail and the triggered (listening) events are picked up in order from head to tail. It ensures execution of concurrent processes (listening events) on first come first served basis. This *deterministic by compilation* approach allows the programmers to choose the order at which they want to execute the concurrent constructs. The designer implicitly chooses the order as the constructs (processes) are executed in order they are written in the design. In the example above (Figure 11), on the positive edge of x, the first always block is executed first followed by the second block. Using list has an additional advantage: the semantics based on lists is directly executable while the semantics with sets is not. To make semantics with sets executable, a number of proofs must be carried about the functions operating on sets.

6.3 Formalizing termination

In this sub-section, different scenarios of non-terminating and non-trivially terminating processes are considered and the approaches taken while formalizing them are discussed in detail. The dotted arrows 1–3 and 7 in Figure 10 are loops: an update event (e.g. registered by continuous assignment) triggers a listening event (registered by an always block) which triggers the listening event registered by the continuous assignment (path 1→2→7). This scenario is described by a VeriFormal code snippet in the Figure 14: the continuous assignment changes the value of wire y which is in the sensitivity list of always block. This triggers the listening event registered by the always block which include the assignment that updates the variable x. As variable x is in the right-hand expression of continuous assignment, it triggers the listening event for continuous assignment and it repeats. Similarly, there are other scenarios which indeed terminate, but Isabelle/HOL compiler cannot determine it (see below).

```
1  ...
2  assign #-1 [ny] = nx,
3
4  always [@](trgexp ny)
5        nx = exp
```

Fig. 14: Non-terminating module: continuous assignment and always block

The higher order logic underneath Isabelle/HOL is a logic of total functions where termination of the functions is a fundamental requirement to maintain consistency [38]. The totality requirement is not specific to Isabelle/HOL and is equally significant in other theorem provers as well [8]. When a recursive function is defined in Isabelle/HOL, it automatically verifies if the function terminates and accepts it if it does. However, in some cases (see example below) it fails and the termination must be proved manually. A custom *well-founded* relation is provided and proved that a *termination argument* decreases in every iteration [38]. This approach requires proof effort, in particular, when the recursive functions are complex such as those defined in VeriFormal. To avoid tedious proofs of termination, a termination argument (guard condition) is added to the arguments of recursive functions whose termination cant be verified automatically by Isabelle/HOL. The recursive call to the function from within the body is subject to decrease in the argument value (by checking it against the maximum number of executions using *less-than* relation). This approach is similar to the proof of termination [38] with the major difference is that a manual proof of termination is not required here. The automatic termination verification is, instead, *assisted* by providing extra information in terms of termination condition checked inside the function. The termination issues in recursive functions over, both, terminating and non-terminating constructs and the way they are addressed are discussed in more detail in the following paragraphs.

The recursive function `execproc` executes individual processes and is the most tricky one to formalize in Isabelle/HOL. The approach taken for termination is a guard condition with a value (passed as argument) of type `nat` is used in the *less-than* [38] relation inside the body of the function. The condition $c' < c$ is checked before every recursive call: c' and c are the number of computations in the process after and before the current iteration, respectively.

Terminating constructs For recursive functions to terminate, the value of at least an argument must decrease in every iteration. If an argument reduces in every recursive call, the compiler is happy with it. However, there are scenarios when the value of an argument (e.g., the number of sub-computations) increases, remains the same or decreases but arbitrarily. In other words, the count of the argument used for termination may increase in some iteration and decrease in others and eventually reduces to zero causing termination. This phenomenon is non-deterministic and hence the compiler can not determine if the execution will ever terminate. In VeriFormal, this is either caused

by a sequence statement, which initially is counted as a single process but consists of more than one statements separated by double semicolons ;; or a process that triggers other processes (e.g., listening events). In both cases, the Isabelle/HOL compiler cannot determine if the recursive function, executing the process, will ever terminate and hence complains. Both scenarios are demonstrated with examples following with a discussion on the way they are executed.

Consider an always process `cpt_alw (s1;;s2)`, where s_1 and s_2 are two blocking assignment statements (sub-computations). This is a three-computation process as the sequence statement consist of two blocking assignments and a no operation process `;;`. Each assignment is executed separately by converting it to an active process (and then to update event) and the semicolon pair is a no-operation process (computation). In other words, when this (active) process is executed, it generates three more processes with two consisting of blocking assignments. To completely execute a process, the process itself and all of its side-effects (sub-processes triggered) must also be executed. In this example, to execute the always process, the two blocking assignments in the process body and `;;` must also be executed. Assuming the blocking assignments do not trigger listening events, the compiler will still complain. The reason is that in the first iteration, there was one process (just always process) which was unfolded into three sub-processes. The value of argument (count of processes) was increased during the first iteration while it should have been decreased and hence the compiler will not accept it.

The example for the second case is a blocking assignment ${}_nx = exp$ and a continuous assignment `assign #-1 [`${}_n$`y] =` ${}_n$`x` and assuming y is not listed in any sensitivity list. Without the guard condition, the process registered for the first assignment is executed by the function `execproc` and updates the value of x which triggers a listening event registered for the second assignment. The function `execproc` has to continue its second iteration to execute the triggered event, however, nothing reduces during the first iteration. In theory, the execution of the initial process terminates after this second event is executed, however, the compiler cannot find it and hence complains: a process was being executed and it is still executing a process.

The guard condition, asserts that the number of computations (sub-processes) reduces in every iteration, is added to the function `execproc`. The sum of number of computations in a process and listening events in the store are calculated in the caller function `execprocs` using functions, `countcpt` and `countcptel`, and passed as argument (of type `nat`) to `execproc`. The reason listening events are taken into account is that a process may trigger some of them as described in the previous paragraphs. To show how this treatment assist Isabelle/HOL to verify termination automatically, we re-visit the example `cpt_alw (s1;;s2)` given above. Assume listening event store has n events and statements s_1 and s_2 do not trigger[2] any event. Initially, the total count of computations passed as argument to `execproc` is $c = n+3$: there are n events in listening event store

[2] This latter assumption is used to keep the scenario simple.

and 3 sub-processes s_1, s_2 and ;; in the current process `cpt_alw (s1;;s2)`. During the first iteration, the body of sequence statement is unfolded, s_1 is executed and the no-operation computation ;; is discharged. The total count is now $c' = n + 1$: only s_2 is left to execute. As the condition $c' < c$ satisfies, the execution proceeds to the next iteration with new termination argument $c = n + 1$. In the second iteration, the statement s_2 is executed which reduces the count to $c' = n + 0$. The condition for recursive call still holds, however, as the body of the initial process has been completely executed, it causes the function to return. A similar explanation holds for the second example given in the previous paragraph.

Non-terminating constructs The example in Figure 14 does not terminate: the processes registered for continuous assignment and always block are mutually dependent and triggers each other. The termination condition approach discussed above is still helpful to terminate the function. Both, the continuous assignment and always block are registered as listening events in the store. Assume the listening event store only has these two events and a single-computation process (e.g., process registered for ${}_nx = exp$) that triggers the first listening event is being executed. When `execproc` function is called to execute the process, the termination count argument is $c = 1 + 2$: a single-computation process + 2 listening events in the store. During the first iteration, the listening event for continuous assignment is triggered which is converted to active event to execute next. As the triggered listening event is removed from the store, the count is $c' = 1 + 1$: the activated event + a listening event in the store. As the termination condition holds, $c' < c$, the execution proceeds to next call with new $c = 2$ is passed as argument. In next iteration, the activated event is executed which triggers the listening event registered for the always block. The new count $c' = 1$: just the newly activated event (listening event store is empty). The termination condition still holds, $1 < 2$, and recursive calls continue with updated argument $c = 1$. In the next iteration, the triggered event for always block is executed which completes the execution of the first process: the process itself and all the side-effects have been completely executed. The termination condition still holds: $0 < 1$, however, as there is no event to continue with, the function returns. This last event should have triggered the listening event for continuous assignment but as it has been removed from the listening event store, the execution terminates. The main trick that causes the function to terminate is the removal of listening event from the store once triggered.

```
1  ...
2  assign #-1 [ny] = nx,
3
4  always [@](trgexp ny)
5        nx = exp
```

Fig. 15: Non-terminating module: continuous assignment and always block

Careful readers must have observed that, the listening event is triggered only once, which is not in line with the actual behaviour of the circuit: a green traffic signal must always (not just once) be turned green after a certain interval. To incorporate such a non-terminating behaviour while at the same time be acceptable to Isabelle/HOL compiler, the listening event is temporarily removed (converted to an active process) from listening events and is restored (converted) back to listening event store *after* the activated process is *completely* exhausted. This treatment will allow other processes to activate the same listening event later on.

The timing of re-creation of a listening event is crucial. If it is not converted back to listening event store, the real essence of listening event is lost (as discussed above) and if it is converted back soon after the activated listening event is executed, then it gets into an infinite loop (e.g., case in Figure 14). To understand the termination issue in non-terminating scenario, the Figure 14 is revisited again as Figure 15. Assume the variable x (line 2) is updated by an assignment in the module which triggers the listening event registered for the continuous assignment (line 2). When the body of the activated event is executed, it will update the value of variable y which is listed in the sensitivity list of the always block (lines 4–5). This will trigger the listening event registered for the always block which updates the value of variable x. If the listening event registered for the continuous assignment is not removed from the listening event store, the always block will trigger it. Similarly, if the listening event registered for the always block is not removed from the store after it is once triggered, it will be triggered again by the update in the continuous assignment and hence the execution will enter into an infinite loop.

The execution of the process, that triggered this loop by updating the value of variable x, completes when all the triggered processes are *completely* executed. In this example, the process that triggered the loop completes when the events registered for continuous assignment and always block are completely executed. To break the loop while at the same time keep the triggered events for latter calls, all the triggered listening events are removed from the store until the process (that triggered them) is completely executed. In the above example, both the listening events are restored back to the store after their activated versions are completely executed.

```
1 | ...
2 | always [@](trgexp ₙx)
3 |       ₙx = exp
```

Fig. 16: Non-terminating module: self-triggering always block

This treatment can also break the loop in self-triggering always block scenario as described in the Figure 16. In a self-triggering always block, a non-recommended coding style, the sensitivity variable x is driven by the assignment in the always body. When the value of x is changed first time, it will

trigger the listening event registered for the always block which executes the assignment statement in the body of the always block. The listening event is removed from the store while its activated version is being executed. When x is updated by the assignment inside the always block, at that time the corresponding listening event is not in the store and hence can not be triggered which terminates the loop. The listening event is restored soon after the body of triggered always block is executed.

Terminating simulation A module simulation can get into looping even if there is no non-terminating construct in the module. Nested future events, for example, add other future events which can cause the simulation to run in loop. This can happen when the delay changes dynamically. Furthermore, a simulation may be desired to run until there is no future event left in the store. This may not terminate either because of the reason above or it may not be possible for the compiler to determine it will ever terminate. To resolve such issues, the simulation is bounded above by the number of simulation cycles. The intended max number of simulation cycles `m` is passed as argument to the main simulation function `simulate` which decrements `m` after each simulation cycle. The `simulate` function returns either if there is no future event to execute (no event scheduled for execution at the next cycle) or the maximum number of simulation cycles is reached.

7 Syntax Complexity Balancing

When embedding an HDL with the aim to enable mechanical reasoning using proof assistant, a *simplicity first* approach should be adopted. Such an approach is needed to reduce proof effort by reducing number of language constructs in the core of DSL while at the same time including most of the language features. The DSL size can be reduced by considering more general constructs with different interpretations implemented in other modules separated from the core of the language. This would keep the core of the language simple and elegant. In our case, the complexity of the core of DSL VeriFormal is reduced by moving some features to the type checker and translator modules.

7.1 Simplicity through separate type checker

The syntactic structure of the embedded language is defined using context-free grammars that precisely define language constructs. To prove that a grammar `G` generates a language `L`, one must show that every string generated by G is in L and conversely prove that every string in language L can indeed be generated by the grammar G [3]. Such a proof for a real life language is extremely tedious, though, careful analyses and scrutiny of the grammars of the language is necessary. As a result of such an investigation, the productions

that generate strings that should not be included in the language are removed or replaced.

Luckily, embedding language in a theorem prover does not need to prove that a language generates valid strings. In fact, the type checker of the theorem prover checks the terms of the language and only accept programs (sentences) that follow the grammar. It ensures that an accepted sentence conforms to the language syntax, however, it does not guarantee that the sentence is valid in the intended language. In Verilog, for example, an `always` block must include procedural timing control with time delay `#` or sensitivity list `@`. The corresponding formal definition of `always` block has the form: `always` *`statement`*, where a statement can be with or without timing control. That is the definition does not enforce the timing control in always block while it should. For example, the grammar accepts the (syntactically correct but) invalid (according to the specification of Verilog) always statement: `always ([`$_n$`x] [=] [#]-1 exp)`.

It must be ensured that all the productions of the grammar must only generate programs allowed by the specification. This property holds if no production of the grammar generates any sentence against the language specification. It can be achieved by rewriting the grammar for statement: write two grammars for statements, one with and the other without time control (time delay and delay). This approach would require additional non-terminal in the grammar and nested pattern-matching in functions operating on statements. A different approach based on *simplicity first* is adopted here: keep the grammar simple even if it produces ill-sentences. To rule out invalid sentences, instead, rules are added in a separate checker program to eliminate sentences that should not be part of the language. This, though, would require additional work, however, would result a modular and simpler language design.

The checker (predicate) program `wfprog` [36] checks the correctness of the VeriFormal module, such as checking validity of the always top statement. The predicate `wfprog` is a conjunction of various other predicates (boolean functions) representing different properties of the language. To check always statement is defined with a timing control, a function `hastimectrl` (Figure 17) is defined. Using pattern-matching on the structure of statement, the function checks the existence of timing control in the statement. An `always` block is well-formed (acceptable in the language) if the statement that follows has a timing control and satisfy few other conditions.

```
1  fun hastimectrl :: "statement ⇒ bool" where
2    "hastimectrl (stm_sensl _ _) = True"
3  | "hastimectrl (stm_delay _ _) = True"
4  | "hastimectrl _ = False"
```

Fig. 17: Checking timing control in statement

The checker predicate can be extended easily without affecting the syntax of the embedded language. If a VeriFormal module (program) is well-formed (passes the test), it indicates the module holds a number of properties: all always statements include at least a timing control. Adding boolean functions to the checker program can be seen as adding a set of hypothesis in the proof context. For example, the predicate `wfprog p` over a program `p` is a conjunction of predicates including `hastimectrl` over statement. In general, the checker predicates rule out non-recommending or invalid programming styles, leads to modular language design and simplifies proofs.

7.2 Simplicity through the translator

Manual encoding of existing Verilog designs in the syntax of VeriFormal is tedious and error prone. To automate translation from Verilog designs into VeriFormal, a prototype translator has been written. The translator gets a Verilog module as input and returns a VeriFormal module as an Isabelle/HOL theory. One of the challenges in such a translation is the creation of unique identifiers as names in VeriFormal and updating the type definitions accordingly. The translator collects the list of all valid Verilog identifiers in the input module and creates an Isabelle/HOL type definition for all VeriFormal names (Verilog identifiers). It updates the type definition `datatype name = list_of_names` in the theory `Syntax.thy` by replacing the existing names *list_of_names* with new names separated by the Isabelle/HOL symbol |. The generated output module includes all the required Isabelle/HOL theories and can be type checked by the Isabelle/HOL compiler for type correctness. The main objective of the translator available with VeriFormal is to translate existing Verilog modules into equivalent modules in VeriFormal, though, it can be used to keep the core of VeriFormal simple. Two Verilog constructs, the generate loop statement and module instantiation, have not been directly formalized in the VeriFormal syntax but included in its translator.

The Verilog meta-programming construct, generate loop statement, enables parametric designs and circuits. One way to add the support for generate statement is to directly formalize it in Isabelle/HOL, however, this would complicate the syntax of VeriFormal. Instead, the support for the meta-programming generate loop construct is added to the VeriFormal translator. An example of a Verilog module with meta-programming construct, its elaboration, RTL schematic and translation to VeriFormal are shown in Figure 18. The Verilog module in Figure 18-a is a description of 2×1 multiplexer using the generate loop statement. The VeriFormal translator automatically *elaborates* it to a Verilog module (Figure 18-b) without generate statement. Both versions of the Verilog modules in Figure 18-a and b result exactly the same schematic (Figure 18-c). The elaborated version of the Verilog module is automatically translated to VeriFormal (Figure 18-d) which is now fit for formal verification (see Section 8) using Isabelle/HOL theorem prover.

```
module mux2x1(bus_out, bus0, bus1, sel);
 output [31:0] bus_out;
 input [31:0] bus0;
 input [31:0] bus1;
 input sel;
 wire [31:0] inpbus [1:0];
 wire [31:0] temp [1:0];

 assign inpbus[0] = bus0;
 assign inpbus[1] = bus1;

 genvar i;
 generate
  for (i=0; i< 2 ; i++)
   begin
    assign temp[i] = (sel == i) ? inpbus[i] : 0;
   end
 endgenerate

 assign bus_out = temp[0] | temp[1];
endmodule
```

(a)

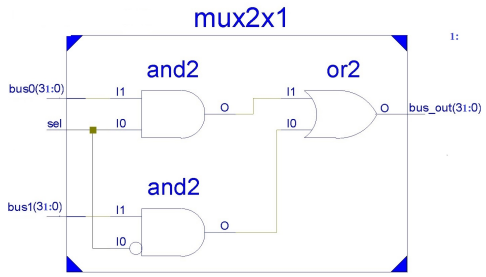
```
module mux2x1(bus_out, bus0, bus1, sel);
 output [31:0] bus_out;
 input [31:0] bus0;
 input [31:0] bus1;
 input sel;
 wire [31:0] inpbus0;
 wire [31:0] inpbus1;
 wire [31:0] temp0;
 wire [31:0] temp1;

 assign inpbus0 = bus0;
 assign inpbus1 = bus1;

 assign temp0 = (sel == 0) ? inpbus0 : 0;
 assign temp1 = (sel == 1) ? inpbus1 : 0;

 assign bus_out = temp0 | temp1;
endmodule
```

(b)



(c)

```
module ([bus_out, bus0, bus1, sel])
 [output [31:0] [bus_out],
 input [31:0] [bus0],
 input [31:0] [bus1],
 input [0:0] [sel],
 wire [31:0] [inpbus0],
 wire [31:0] [inpbus1],
 wire [31:0] [temp0],
 wire [31:0] [temp1],

 assign #−1 [n inpbus0]=n bus0,
 assign #−1 [n inpbus1]=n bus1,

 assign #−1 [n temp0]=
   c(l n sel[==]v (0,1)) n inpbus0 v (0,32),
 assign #−1 [n temp1]=
   c(l n sel[==]v (1,1)) n inpbus1 v (0,32),

 assign #−1 [n bus_out]=b n temp0 [|] n temp1]
endmod
```

(d)

Fig. 18: Automatic elaboration and translation of Verilog module - (a) with and b) without generate statement, (c) schematic diagram of (a) and (b), (d) VeriFormal module

Like many other programming languages, such as C++ and Java, Verilog includes modules to implement code-reuse. A module is defined once and re-used many times by instantiating it inside another module. The module instantiation feature of Verilog is not directly formalized in VeriFormal but implemented through the translator. VeriFormal, instead of formalizing module substantiation, compresses the entire Verilog design into one VeriFormal module using *flattening* [19,18,27]. A top level module, a module that instantiates other modules, is flattened [27] by renaming all local variables in the module instances to avoid clashes, replace all module instances by the sequence of statements contained in the instantiated modules and declare all local variables at top level. A simple example of flattening performed by the VeriFormal translator is available at the Github repository at `https://github.com/wilstef/veriformal`. As suggested by Gordon [27], other transformations may also be included in the translator. One such trans-

formation would be to replace continuous assignments with `always` blocks and remove the definition of continuous assignment from VeriFormal syntax.

8 Circuit Verification in VeriFormal

The main motivation for creating VeriFormal was to support formal verification of hardware descriptions described in HDL Verilog. The HDL VeriFormal together with its translator, type checker and Isabelle/HOL compiler provides a platform to formally type-check Verilog modules in an automatic way. An existing Verilog module can be automatically translated to VeriFormal module and then type-checked using VeriFormal and Isabelle/HOL type-checkers.

To demonstrate this feature with an example, we designed a 32-bit 8×1 multiplexer circuit in Verilog and automatically translated it to an Isabelle/HOL theory. The generated theory contains definition of VeriFormal module `mux8x1` and includes the required theories. All the identifiers in the VeriFormal module have type `name` and must be defined, otherwise, the Isabelle/HOL compiler will complain at run time. To resolve this issue, the translator additionally collects the list of Verilog identifiers in the module and creates a type definition `name` for all the identifiers. The type definition `name` in Figure 19 includes as its members all the identifiers in the VeriFormal module described in Figure 20.

```
1 datatype name =
2    bus_out | bus0 | bus1 | bus2 | bus3 | bus4 | bus5 | bus6 |
3    bus7 | sel | inpbus0 | inpbus1 | inpbus2 | inpbus3 | inpbus4 |
4    inpbus5 | inpbus6 | inpbus7 | temp0 | temp1 | temp2 | temp3 |
5    temp4 | temp5 | temp6 | temp7
```

Fig. 19: VeriFormal identifiers as names

```
module ([bus_out,bus0,bus1,bus2,bus3,bus4,bus5,bus6,bus7,sel])
 [output [31:0] [bus_out],
 input [31:0] [bus0],
 input [31:0] [bus1],
 input [31:0] [bus2],
 input [31:0] [bus3],
 input [31:0] [bus4],
 input [31:0] [bus5],
 input [31:0] [bus6],
 input [31:0] [bus7],
 input [2:0] [sel],

 reg [31:0] [inpbus0],
 wire [31:0] [inpbus1],
 wire [31:0] [inpbus2],
 wire [31:0] [inpbus3],
 wire [31:0] [inpbus4],
 wire [31:0] [inpbus5],
 wire [31:0] [inpbus6],
 wire [31:0] [inpbus7],

 wire [31:0] [temp0],
 wire [31:0] [temp1],
 wire [31:0] [temp2],
 wire [31:0] [temp3],
 wire [31:0] [temp4],
 wire [31:0] [temp5],
 wire [31:0] [temp6],
 wire [31:0] [temp7],

 assign #-1 [n inpbus0]=n bus0,
 assign #-1 [n inpbus1]=n bus1,
 assign #-1 [n inpbus2]=n bus2,
 assign #-1 [n inpbus3]=n bus3,
 assign #-1 [n inpbus4]=n bus4,
 assign #-1 [n inpbus5]=n bus5,
 assign #-1 [n inpbus6]=n bus6,
 assign #-1 [n inpbus7]=n bus7,

 assign #-1 [n temp0]=c( l n sel[==]v (0,3)) n inpbus0 v (0,32),
 assign #-1 [n temp1]=c( l n sel[==]v (1,3)) n inpbus1 v (0,32),
 assign #-1 [n temp2]=c( l n sel[==]v (2,3)) n inpbus2 v (0,32),
 assign #-1 [n temp3]=c( l n sel[==]v (3,3)) n inpbus3 v (0,32),
 assign #-1 [n temp4]=c( l n sel[==]v (4,3)) n inpbus4 v (0,32),
 assign #-1 [n temp5]=c( l n sel[==]v (5,3)) n inpbus5 v (0,32),
 assign #-1 [n temp6]=c( l n sel[==]v (6,3)) n inpbus6 v (0,32),
 assign #-1 [n temp7]=c( l n sel[==]v (7,3)) n inpbus7 v (0,32),

 assign #-1 [n bus_out]=
   b n temp0 [|] b n temp1 [|] b n temp2 [|] b n temp3 [|]
   b n temp4 [|] b n temp5 [|] b n temp6 [|] n temp7]

endmod
```

Fig. 20: Type checking description of 8×1 MUX using VeriFormal type checker

The VeriFormal description `mux8x1` was type checked for type errors using the command `value "wfprog mux8x1"`. The predicate `wfprog` in VeriFormal returns `True` if the module conforms to certain typing rules defined in `Correctness` library, otherwise, it returns `False`. We intentionally introduced a type error by changing the type `wire` on line 13 to `reg`. The type checker predicate `wfprog` returned `False`. The Verilog version of the same erroneous module was not accepted by the Icarus Verilog simulator but accepted by Synopsys VCS 2014. 10 (online) simulator[3]. The reason, VeriFormal type checker and Icarus Verilog simulator did not accept it, was the type `reg` (line 13) of identifier `inpbus0`: an identifier of type `reg` cannot be driven by a continuous assignment. The Icarus Verilog 0.9.7 simulator captures this type error and complains as *error: reg inpbus0; cannot be driven by primitives or continuous assignment.* When the type of identifier `inpbus0` is changed to `wire` in Verilog and VeriFormal modules, the Icarus Verilog, Synopsys VCS and VeriFormal simulators accept them.

Additionally, formal setting underneath VeriFormal and theorem proving facility of Isabelle/HOL may also be used to interactively prove correctness of VeriFormal expressions and modules. A simple example of such a verification is given in Figure 21. The well-formedness of the expression `exp_name q` is checked interactively. The lemma `well-formed-exp` states that in the environment where identifier `q` is defined, the expression `exp_name q` is well-formed.

```
1 | lemma well-formed-exp: "wfexp [top_in n2 n1 [q]] (exp_name q)"
2 |   apply (simp add: wfexp_def env_def)
3 | done
```

Fig. 21: Proving well-formedness of VeriFormal expression

Furthermore, the type checker module of VeriFormal has been used to statically type check many other descriptions for type correctness [36]. VeriFormal has been successfully used to describe and verify hardware circuits [35]. A full adder circuit was implemented in VeriFormal using two half adders and a 3-to-8 lines decoder. These two different implementations of full adder circuit were checked equivalent in the executable VeriFormal simulator. Similarly, a 4-bit comparator circuit was described in VeriFormal and Verilog and the equivalence of both designs was checked by running them in VeriFormal and Icarus simulators. This kind of verification checks the correction of Verilog description against a formal specification of the same design in VeriFormal. The source codes of aforementioned designs are available at our Github repository at link `https://github.com/wilstef/veriformal-fyp2017`.

[3] The online Synopsys VCS and Icarus Verilog simulators were accessed from URL `https://www.edaplayground.com/`.

9 Lessons Learned

Embedding domain-specific HDL in theorem provers is challenging but fun. There are very few HDL embedded in the logic of theorem prover and hence a detailed discussion on how the syntax of language constructs were defined and interpreted is important for programming language design and hardware verification community. Following is a non-exhaustive list of lessons learned during this work.

- Embedding domain-specific language in a theorem prover requires multidisciplinary expertise, namely in programming language design, the specific domain (e.g., hardware descriptions), the theorem prover itself and the logic behind it.
- Using a theorem prover as a host language provides many additional benefits: proof facility in the logic of theorem prover and static type checking of the programs.
- The type checker of theorem prover enforces the designer to choose unique constructor names for each production of the grammar which the compiler uses for disambiguation. Using pattern-matching on these constructors, the theorem prover allows one to formalize ambiguous and left-recursive grammars.
- Constructor names complicates translation from existing languages to the new one: these names must be put in proper places to create equivalent constructs in the new language.
- The notation feature in theorem prover facilitates making the new language similar to an existing known language, but the requirement of unique constructors restricts this facility.
- While defining HDL in theorem prover, formalizing terminating constructs can be as tricky as non-terminating constructs.

10 Related Work

There is a huge body of work on domain-specific languages in general [53, 52] and on HDLs in particular. Domain-specific HDLs defined in [6, 41, 22, 46] do not have formal semantics with logic suitable for formal reasoning and hence are orthogonal to our work on VeriFormal. There are others HDLs with formal semantics, however, they are either not widely used [10, 12] as Verilog or include only a subset of constructs [42] or the underlying logic is not powerful and expressive [44] as the higher-order logic of Isabelle/HOL.

Programming languages tailored towards targeted domain, domain-specific, are more expressive and easy to learn as compared to general purpose programming languages. To define a DSL, it is either implemented as a standalone language with its own tools (external DSLs) or embedded in a host language (internal DSLs) [23]. The later approach is often preferred for DSLs [52] as it allows designers to leverage the host language tools. It is evident that functional programming features facilitate the embedded approach [26,

25]. Tony Sloane [52] explored the trade offs between embedding and standalone approaches to language implementation. The author went on to sharing his experience of implementing DSLs, by embedding a simple imperative language in Scala [48]. In particular, the Scala language features that assisted in the embedding approach were investigated. Svenningsson and Axelsson [53] presented a technique for combining deep and shallow embedding approaches and demonstrated their technique by embedding a small functional language FunC. According to the authors, their technique keeps the deep embedding small, makes the embedded language extension easier and provides a natural programming interface to the embedded language. Mernik et al. [45] defined DSL development methodologies and guided language designers about when and how to to define a DSL. They also discussed systems for language development and tools to analyse the target domain of the language with the aim to speed up the language designs.

VeriFormal is not the only embedded HDL. Among many others are Chisel [6], PyMTL [41], MyHDL [22] and Bluespec [46]. The first one was embedded in Scala, the second and third were embedded in Python and the last one is a standalone language with strong type checking facility. None of these languages have formal semantics and can not be used for formal reasoning. Braibant et al. [13] defined a deep embedding of Fe-Si, a simplified version of the higher-level language Bluespec, in the theorem prover Coq. In a recent similar work, Choi et al. [15] developed a platform Kami for high-level parametric hardware specification. Kami is a formalized version of a sub-set of Bluespec in Coq. Both, Fe-Si [13] and Kami [15] differs VeriFormal in the level of hardware specification: Bluespec is used for high-level hardware specification while VeriFormal is a low-level language at RTL.

When programs in the language are desired to, in addition to execution, reason about them, the best choice is to embed the DSL in a language with both facilities: functional programming and theorem proving. Languages with both of these features are called interactive theorem provers, such as Isabelle/HOL [47] and Coq [7], to name a few. Boulton et al. [10] embedded three HDLs ELLA, SILAGE and VHDL in theorem prover HOL [1]. The semantics of the first two languages were represented denotationally in the logic of HOL (shallow embedding) whereas an interpreter was defined for VHDL to interpret program texts in an operational style (deep embedding). The embedded approach to ELLA and SILAGE differs in the type of data structure chosen for denotations: for the former, the denotations of programs are functions while they are relations for the later.

Gordon [27] embedded a non-executable simplified version of Verilog using mathematical notations. With the aim to demonstrate the semantic challenges of Verilog, the author included only a small subset of Verilog features and provided an informal operational semantics in English. Meredith et al. [44] defined an executable semantics for Verilog by embedding it in the tool Maude [17] with rewriting logic as the underlying logic. The formal semantics in Maude motivates the work in VeriFormal, however, the rewriting logic underneath Maude is inadequate in various reasoning scenarios [36]. Building

upon the concepts behind proof-carrying code, Love et al. [42] implemented a framework by formalizing a synthesizable subset of Verilog in proof assistant Coq. Another closely related work is the HDL E [12], deeply embedded in theorem prover ACL2 [51], which models a subset of Verilog. The hardware units, designed as E modules using And-Inverter-Graph and Boolean function representations, can be symbolically simulated.

11 Conclusions

This work highlights the significance of embedding DSL in theorem prover and provides a basis for further research and developments towards compiler verification, verification of Verilog simulators and formal tools for hardware description language Verilog. To implement a domain-specific language, the designer either implements it as a standalone language or embed it using a general programming language. Embedding a DSL in a theorem prover results many additional benefits over other general purpose programming languages as the host languages: the tools of theorem prover can be used to statically type check and reason about programs defined in the embedded language. A domain-specific language, VeriFormal, embedded using the higher-order logic of theorem prover Isabelle/HOL was investigated in this research work. While embedding VeriFormal, a number of choices in language design were investigated. In particular, the challenges inherent to hardware description languages, were highlighted in the domain of theorem prover. Embedding a DSL in theorem prover, though, comes with a cost of confronting with defining total functions and understanding and programming in the syntax of the embedded language. VeriFormal currently supports simulation and formal reasoning and an extension towards synthesis is in future plans.

References

1. HOL Interactive Theorem Prover. `https://hol-theorem-prover.org/`. Accessed: 2018-01-08
2. VeryPCC project. `http://isabelle.in.tum.de/verypcc/`. Accessed: 2017-11-22
3. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools, vol. 2. Addison-wesley Reading (2007)
4. Ashenden, P.J.: The designer's guide to VHDL, vol. 3. Morgan Kaufmann (2010)
5. Association, I.S., et al.: IEEE Standard for Verilog Hardware Description Language. Design Automation Standards Committee, 2005. IEEE Std 1364TM-2005
6. Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., Asanović, K.: Chisel: constructing hardware in a scala embedded language. In: Proceedings of the 49th Annual Design Automation Conference, pp. 1216–1225. ACM (2012)
7. Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J.C., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C., et al.: The Coq proof assistant reference manual: Version 6.1. Ph.D. thesis, Inria (1997)
8. Barthe, G., Forest, J., Pichardie, D., Rusu, V.: Defining and reasoning about recursive functions: a practical tool for the Coq proof assistant. In: International Symposium on Functional and Logic Programming, pp. 114–129. Springer (2006)

9. Bohannon, A.: Foundations of web script security. University of Pennsylvania (2012)
10. Boulton, R.J., Gordon, A.D., Gordon, M.J., Harrison, J., Herbert, J., Van Tassel, J.: Experience with Embedding Hardware Description Languages in HOL. In: TPCD, vol. 10, pp. 129–156 (1992)
11. Bowen, J., Gordon, M.: A shallow embedding of Z in HOL. Information and software technology **37**(5-6), 269–276 (1995)
12. Boyer, R.S., Hunt Jr, W.A.: The e language. In: Proceedings of the International Workshop on Hardware Design and Functional Languages. March (2007)
13. Braibant, T., Chlipala, A.: Formal verification of hardware synthesis. In: Computer Aided Verification, pp. 213–228. Springer (2013)
14. Bugliesi, M., Calzavara, S., Focardi, R., Khan, W.: CookiExt: Patching the browser against session hijacking attacks. Journal of Computer Security **23**(4), 509–537 (2015)
15. Choi, J., Vijayaraghavan, M., Sherman, B., Chlipala, A., et al.: Kami: a platform for high-level parametric hardware specification and its modular verification. Proceedings of the ACM on Programming Languages **1**(ICFP), 24 (2017)
16. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT press (1999)
17. Clavel, M., Durán, F., Hendrix, J., Lucas, S., Meseguer, J., Ölveczky, P.: The Maude formal tool environment. In: Algebra and Coalgebra in Computer Science, pp. 173–178. Springer (2007)
18. Compiler II, F.: Fpga express verilog reference manual, synopsys. Inc.-Ver (1999)
19. Compiler II, F.: Fpga express vhdl reference manual, synopsys. Inc.-Ver (1999)
20. Cuadrado, J.S., Izquierdo, J.L.C., Molina, J.G.: Comparison between internal and external dsls via rubytl and gra2mol. In: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, pp. 109–131. IGI Global (2013)
21. Dawson, J.: Isabelle theories for machine words. Electronic Notes in Theoretical Computer Science **250**(1), 55–70 (2009)
22. Decaluwe, J.: MyHDL: a python-based hardware description language. Linux journal **2004**(127), 5 (2004)
23. Fowler, M.: Domain-specific languages. Pearson Education (2010)
24. Ghosh, D.: DSL for the uninitiated. Communications of the ACM **54**(7), 44–50 (2011)
25. Gibbons, J.: Functional programming for domain-specific languages. In: Central European Functional Programming School, pp. 1–28. Springer (2013)
26. Gibbons, J., Wu, N.: Folding domain-specific languages: Deep and shallow embeddings (functional pearl). In: ACM SIGPLAN Notices, vol. 49, pp. 339–347. ACM (2014)
27. Gordon, M.: The semantic challenge of Verilog HDL. In: Logic in Computer Science, 1995. LICS'95. Proceedings., Tenth Annual IEEE Symposium on, pp. 136–145. IEEE (1995)
28. Grimm, T., Lettnin, D., Hübner, M.: A survey on formal verification techniques for safety-critical systems-on-chip. Electronics **7**(6), 81 (2018)
29. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: International Symposium on Functional and Logic Programming, pp. 103–117. Springer (2010)
30. Hanna, K., Daeche, N.: Specification and verification using higher-order logic. Computer hardware description languages and their applications pp. 418–433 (1985)
31. Hudak, P.: Building domain-specific embedded languages. ACM Computing Surveys (CSUR) **28**(4es), 196 (1996)
32. Hunt Jr, W.A.: The mechanical verification of a microprocessor design. HDL Descriptions to Guaranteed Correct Circuit Designs pp. 89–129 (1987)
33. Hupel, L., Nipkow, T.: A verified compiler from isabelle/hol to cakeml. In: European Symposium on Programming, pp. 999–1026. Springer (2018)
34. Johnson, S.C.: Lint, a C program checker. Bell Telephone Laboratories Murray Hill (1977)
35. Khan, W., Azam, B., Shahid, N., Khan Abdul, M., Shaheen, A.: Formal verification of digital circuits using simulator with mathematical foundation. In: Materials Science Forum, vol. x, pp. x–x. Trans Tech Publ (2018)
36. Khan, W., Tiu, A., Sanán, D.: VeriFormal: An Executable Formal Model of a Hardware Description Language. In: SG-CRC, pp. 19–36 (2017)
37. Kosar, T., Bohra, S., Mernik, M.: Domain-specific languages: A systematic mapping study. Information and Software Technology **71**, 77–91 (2016)

38. Krauss, A.: Defining recursive functions in Isabelle/HOL (2008)
39. Krauss, A.: Partial and nested recursive function definitions in higher-order logic. Journal of Automated Reasoning **44**(4), 303–336 (2010)
40. Lam, W.K.: Hardware Design Verification: Simulation and Formal Method-Based Approaches (Prentice Hall Modern Semiconductor Design Series). Prentice Hall PTR (2005)
41. Lockhart, D., Zibrat, G., Batten, C.: PyMTL: A unified framework for vertically integrated computer architecture research. In: Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on, pp. 280–292. IEEE (2014)
42. Love, E., Jin, Y., Makris, Y.: Proof-carrying hardware intellectual property: A pathway to trusted module acquisition. IEEE Transactions on Information Forensics and Security **7**(1), 25–40 (2012)
43. Melham, T.F., et al.: Using recursive types to reason about hardware in higher order logic, vol. 135. University of Cambridge, Computer Laboratory (1988)
44. Meredith, P., Katelman, M., Meseguer, J., Rosu, G.: A formal executable semantics of Verilog. In: Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on, pp. 179–188. IEEE (2010)
45. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM computing surveys (CSUR) **37**(4), 316–344 (2005)
46. Nikhil, R.S.: Bluespec: A general-purpose approach to high-level synthesis based on parallel atomic transactions. In: High-Level Synthesis, pp. 129–146. Springer (2008)
47. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer Science & Business Media (2002)
48. Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: An overview of the Scala programming language. Tech. rep. (2004)
49. Pace, G.J., He, J.: Formal reasoning with Verilog HDL. In: Workshop on Formal Techniques for Hardware and Hardware-like Systems (1998)
50. Reetz, R.: Deep embedding VHDL. In: International Conference on Theorem Proving in Higher Order Logics, pp. 277–292. Springer (1995)
51. Shelley, G., Forrest, S.: ACL2 Theorem Prover
52. Sloane, T.: Experiences with domain-specific language embedding in Scala. In: Domain-Specific Program Development, p. 7 (2008)
53. Svenningsson, J., Axelsson, E.: Combining deep and shallow embedding for EDSL. In: International Symposium on Trends in Functional Programming, pp. 21–36. Springer (2012)
54. Thomas, D., Moorby, P.: The Verilog® Hardware Description Language. Springer Science & Business Media (2008)
55. Wildmoser, M., Nipkow, T.: Certifying machine code safety: Shallow versus deep embedding. Theorem Proving in Higher Order Logics pp. 133–142 (2004)