International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:6, 2008

Development of A Meta Description Language for Software/Hardware Cooperative Design and Verification for Model-Checking Systems

Katsumi Wasaki, and Naoki Iwasaki

***Abstract*—**Model-checking tools such as Symbolic Model Verifier (SMV) and NuSMV are available for checking hardware designs. These tools can automatically check the formal legitimacy of a design. However, NuSMV is too low level for describing a complete hardware design. It is therefore necessary to translate the system definition, as designed in a language such as Verilog or VHDL, into a language such as NuSMV for validation. In this paper, we present a meta hardware description language, Melasy, that contains a code generator for existing hardware description languages (HDLs) and languages for model checking that solve this problem.

***Keywords*—**meta description language, software/hardware co-design, co-verification, formal verification, hardware compiler, model checking.

I. Introduction

ELECTRONIC devices operate in a variety of environments, especially in information technology, where their penetration and numbers are increasing yearly. Much social system infrastructure is dependent on this technology. The reliability of this technology, especially pertaining to its digital systems, is essential to maintaining the function and safety of the entire system. For this reason, a desirable design environment is one that enables users to design highly reliable systems, which operate accurately according to the specifications, at low cost and high efficiency [1], [2].

Compilers have been developed that generate objects and executable modules for a target system via code generation, by implementing the target system using a high-level language following its functional design [3], [4]. In particular, hardware compilers have been developed that generate the configuration information for a circuit. This information comes directly from the code written in a relatively high-level language to describe the design of the hardware. Such compilers are used in industry [5], [6], [7], [8].

There are model-checking tools [9] for hardware design, such as Symbolic Model Verifier (SMV) [10] and NuSMV [11]. These tools automatically check the formal legitimacy of a design. However, with NuSMV, it is necessary to use a very low-level language to describe the design of the actual hardware completely. Therefore, an additional process is required, which translates the system design from a language such as VHSIC Hardware Description Language (VHDL) [6] or Verilog [7] into a language suitable for validation, such as NuSMV. In addition, when considering the practical circumstances at a development site, it is difficult to implement the same design several times in different languages because of limitations such as cost and delivery time.

Katsumi Wasaki and Naoki Iwasaki are with Faculty of Engineering, Shinshu University, Nagano, Japan. E-mail: {wasaki,safii}@cs.shinshu-u.ac.jp

Codesign of software and hardware is possible during development, by using the SystemC description language [8]. There are formal design and verification methods in which the software and hardware areas of one system development can be separated in a coordinated manner, while they are being designed. This enables designers to design a system with a high-speed processing capability, so that the processes that are difficult to implement in hardware because of their algorithmic complexity can be handled in software. Alternatively, a processing area that requires a long calculation time when implemented using software (on a microprocessor), can be handled by hardware twists such as parallelization and the use of a pipeline. However, it is necessary to produce designs for both the software-processing areas and the hardware-processing areas. Then, if a change in the specification occurs in either the software area or the hardware area, the design of the other area must be modified, and the cost increases because changes in specification are now problematic.

In this study, we present the design of a Meta hardware description language system, Melasy, which has a code generator for existing hardware description languages, languages for model checking, and C language modules for simulations and codesign. Melasy uses a functional programming language, Haskell [12], and its higher-level parser library, Parsec [13], to implement its compiler-processing subsystem. A system described in the metalanguage can generate various target object codes by selecting a destination language for the code generation based on its description. Inter alia, we have succeeded in generating the code for a direct model-checking tool (SMV) by describing a system design, in conjunction with the specification to be met, using the metalanguage. To evaluate the syntactic merits and description capabilities via case studies, we have compared and evaluated a solvable range of problems, which are difficult to solve using conventional methods, by describing systems and using our compiler to create object code in various languages.

II. Related Studies

A. HDCaml

HD Categorical Abstract Machine Language (HDCaml) [14] is a language for hardware description and checking. This language is implemented as a library, using Objective Caml

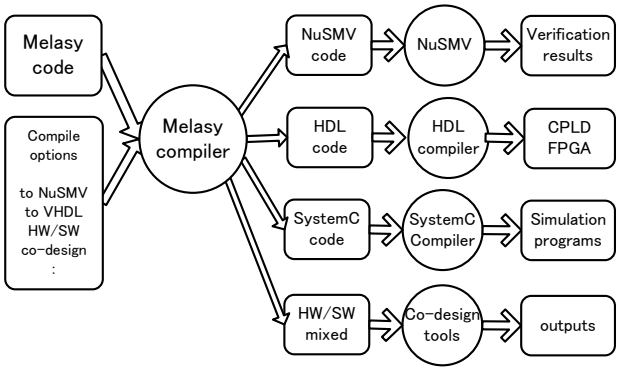

Fig. 1. The relationship of the Melasy compiler to various HDLs and the model checker.

(OCaml) rather than using a new language. When programs described in OCaml are compiled, executable files to generate code for various hardware description languages (HDLs) and checking systems are generated. One of HDCaml's features is that OCaml's library can be included without modification, because HDCaml describes systems exactly as does OCaml.

B. Confluence

Confluence [15] is also a language for hardware description and checking. Although the compiler of this language is described using OCaml, it operates as a fully independent compiler. Confluence performs its code generation via an intermediate code, and the Confluence code is first compiled into intermediate code (Netlist) by its preprocessor. Then, the target codes for VHDL and Verilog are generated from the Netlist by its postprocessor.

III. Versatile Capabilities of the Melasy Compiler

Melasy has been designed as a metalanguage for a variety of existing languages. Hardware design descriptions in Melasy generate various HDL descriptions via the Melasy compiler. Figure 1 shows the relationships between the Melasy compiler, SMV, and HDL. By generating Verilog or VHDL hardware descriptions via the Melasy compiler, it is possible to produce hardware designs from highly abstract specifications. By changing the compiler options, it is also possible to obtain both verification code and executable code by generating a description for model-checking tools such as NuSMV. Furthermore, in the codesign of software and hardware, it is possible to use one Melasy description to generate various codes with different proportions of hardware and software processing, by specifying the ratio of hardware description to software description in the compiler options. The details of the syntax and description capability are discussed in Section V.

IV. Design Policy and Features of the Melasy Metalanguage

A. Language Design Policy

Since digital hardware often operates in parallel with other hardware, the repetitive processing that is often used in procedural-language software design is inefficient in many cases. In digital arithmetic circuits, design twists to enable parallel processing are frequently adopted. As a result, similar descriptions (modules) appear in the design, and this significantly impairs the readability and description capability of the code. Our proposed Melasy language generates multiple formulas from a single formula using a syntax-expressed repetition, with the aim of eliminating this problem. Furthermore, by allowing for recursive definition of objects as well as recursive definition of formulas, it enables a user to describe highly complex designs easily.

The advantages to be derived in future from enabling Melasy to describe the design of a software area using, for example, SystemC, as well as the advantages of its hardware description, have already been discussed. However, SystemC's language description capabilities were not adopted here, because the implementation of the handling of synchronization in the parallel processing area during output of the hardware codes would be complicated if Melasy were designed as a normal procedural language. Therefore, an object-oriented functional language has been adopted for Melasy. In Melasy, objects are circuits with inputs and outputs. The definition of an object input is handled as an argument when declaring an instance, and the output is handled similarly to a member variable. Each object corresponds to a hardware circuit and connects an input and an output when generating an object instance. Simple repetition of the same code, which is often seen in designs using existing HDLs, is described using a syntax that represents repetition rather than an actual repetition of code. This clearly expresses the meaning of a repetition structure while reducing the quantity of code required.

B. Type Inference

Type inference is a mechanism for obtaining a proper function definition, in which a compiler infers the type, even if the designer (user) does not specify the type of target object (variable) explicitly. The compiler infers the type via connections and/or substitution, starting from a point where its type is defined. Using type inference, a user can save on mental labor, and can check the type in any given situation. The user can generate a component with a determined type from an ambiguous definition during the type inference process, after defining a component using an ambiguous definition. Furthermore, a component of indeterminate type can be defined as a component independent of type by the output of various specific objects. This enables the user to define an object, such as a computing unit or a selector, that can be used for any type.

C. Comparison of Description Capability

The description capabilities of Melasy and SMV are compared in this subsection.

1) Ambiguous components.: Melasy can define various components from a single definition via ambiguous component definition using a template. For example, buffers with size = 2 and size = 4 are defined in Figure 2.

Using NuSMV, a buffer with size = 2 and a buffer with size = 4 must be defined separately (Figure 2(a)). However,

```
Module buffer2()
 VAR
   buf : array 0..1 of boolean;
Module buffer4()
 VAR
   buf : array 0..3 of boolean;
Module main
 VAR
   buf2 : buffer2;
   buf4 : buffer4;
```

(a) Example of a description using NuSMV.

```
component Buffer<N>()
 var
   buf[N] :: Bool;
component Main
 var
   buf2 :: Buffer<2>;
   buf4 :: Buffer<4>;
```

(b) Example of a description using Melasy.

Fig. 2. Example of a description for a buffer with sizes = 2 and 4.

```
MODULE Hoe
 VAR
   hoe : array 0..7 of boolean;
 ASSIGN
   init(hoe[0]):=1;
   init(hoe[1]):=0;
   init(hoe[2]):=0;
   init(hoe[3]):=0;
   init(hoe[4]):=0;
   init(hoe[5]):=0;
   init(hoe[6]):=0;
   init(hoe[7]):=1;
```

(a) Example of a description using NuSMV.

```
component Hoe()
  var
    hoe[N=8] :: Bool | N==0 | N==7  =1,
                     | otherwise    =0;
```

(b) Example of a description using Melasy.

Fig. 3. Example of a description for the initialization of an assignment array, hoe[].

using Melasy, buffers of various sizes can be generated from a single definition by initially defining the buffer to have an ambiguous size and then specifying the size when declaring an instance (Figure 2(b)).

2) Initialization of an array.: The methods for the initialization of an assignment array and for substitution are compared in this subsection. As an example, let us compare the methods for performing a special initialization of the first and last elements of an array. Instead of using a special initialization routine, let us initialize the first and last elements with a value of "1" and the other elements with a value of "0". Figure 3 shows examples of each definition.

Because NuSMV offers a choice of syntax for handling arrays, it is necessary to initialize all the elements (Figure 3(a)). Using Melasy, this can be written concisely by using a guard area and an implicit `foreach`. In addition, the conciseness contributes to a clearer description when reading the code, and a reduction in the user's workload (Figure 3(b)).

V. Syntax of the Melasy Language

A. Syntax Rule Definition

Figure 4 shows the Melasy syntax described in Extended Backus–Naur Form (EBNF). Melasy code comprises "columns of component definitions" that compose a circuit. Each component definition contains a `var` block that defines the variables in the component, and an `assign` block that defines the update of the status. In both blocks, the description of a formula uses an implicit `foreach` with a guard area, as will be discussed in detail in Section V-B.

```
<idunit>  = <lower> | <upper> | <digit>
<idunits> = <idunit>|<idunit> <idunits>
<id>      = <lower> | <lower> <idunits>
<Id>      = <upper> | <upper> <idunits>
<number>  = <digit> | <digit> <number>
<varId>   = <id> *<arraySize> ["." <varId>]
<suffixReferer> = "@" <number>
<substitution>  = <substitution_>
                | <connection> | 1#<guard>
<substitution_> = "=" <expr>
<connection>    = "(" #<expr> ")"
<if>         = "if"<expr_b> "then" <expr>
                       "else" <expr>
<guard>      = "|" <expr_b> <substitution>
<assign>     = <varId> <substitution>
<program>    = <components>
<components> = <component>
             | <component> <components>
<component>  = "component" <Id> [<templateVar>]
               "(" [<arguments>] ")" *<block>
<templateVar> = "<" #<Id> ">"
<templateArg> = "<" #<Expr> ">"
<arguments>   = #<argument>
<argument>    = <varId> [<varType>]
<arraySize>   = "[" [[<id> "="] <expr>]"]"
<block>    = <varBlock> | <assignBlock>
           | <defineBlock>
<varBlock> = "var" *(<var> ";")
<var>      = <varId> [<varType> [<templateArg>]
             [<substitution>]
<varType>  = "::" ("Bool" | <Id>)
<assignBlock> = "assign" *(<assign> ";")
```

Fig. 4. Melasy syntax rules (EBNF).

B. Implicit `foreach` with Guard Area

An implicit `foreach` is a mechanism that helps to define the values of an array. It allows a user to define all the elements of an array using a single formula, by making the formula act on all the elements. For example, since the information regarding the index of an element in the array is given to the formula via a special invariant, the value of the formula can be changed using the partial information for the array. It is also possible to change the shape of a formula according to the conditions by setting a guard area.

Since many existing HDLs do not have a function that expresses statement repetition, they cannot describe simple repetition clearly. Therefore, users have to write similar expressions using copy-and-paste, or they have to create a simple preprocessor themselves. This makes the process troublesome, even in cases where a description is repeated by slightly changing part of an identifier string, such as a constant identifier or a variable identifier. Using Melasy, such operations can be

reduced by using the implicit `foreach` and guard area. The description of an array can be completed in a single row, and processing, such as the input of special values to a part of an array, can also be performed easily via the guard area. An example of this, using a Melasy description, is shown in Figure 5.

In this way, Melasy allows users to explicitly express information that has structural meaning such as, in this case, that the even value is "1" and the odd value is "0". This is in addition to enabling a user to describe codes easily (Figure 5).

```
even[N] | (N%2)==0  = 1,
        | otherwise = 0;
```

Fig. 5. Example of a Melasy description that uses implicit `foreach` for the initialization of a register.

C. Component with a Template

A template is a mechanism by which a user freely defines invariants that are processed during compiling when declaring instances. For example, by defining an N-bit buffer using a template, rather than by defining a buffer with a width of four bits and a buffer with a width of eight bits separately, and then specifying a value for N when generating an instance, it is possible to create designs with buffers of various widths from a single design (Figure 6).

```
component Buffer<N>()
 var
   buf[N] :: Bool;
component Main
 var
   buf2 :: Buffer<2>;
   buf4 :: Buffer<4>;
```

Fig. 6. Example of a Melasy description with ambiguous components for buffers of size = 2 and 4.

VI. Code Generation for a Model-Checking Tool

There is a significant difference in the description capability between the codes that contain Melasy's implicit `foreach` and/or templates and the codes that can be described by the SMV model-checking tool. For this reason, we decided to expand the implicit `foreach` and template first, and then generate intermediate code that does not contain these functions, when generating SMV code from Melasy code.

A. Generation Procedure via Intermediate Code

1) Expansion of Implicit `foreach`*:* In the expansion of an implicit `foreach`, the suffixes in an array are substituted by special invariants, and the invariants and suffixes are substituted by numeric values that represent these values in an invariant list.

2) Expansion of a Template: In the expansion of a template, the template name is expanded (mangled) using the values passed to the template arguments, and all the template arguments in the expanded component are substituted by invariants. In the compiler we have developed so far, a name mangle is the value input into a template argument and is added after the name using an underscore delimiter "_". Because a template argument may be used in the declaration of an instance defined for a component that includes a template, it is necessary to repeat the same processing steps until the end of the template expansion. At this stage, an ambiguous component will be concretized, and an implicit `foreach` will have been substituted by an assignment for a common array. However, other syntactic elements will require some simple modifications.

3) Expansion of "If-then-else" Syntax: SMV has no "if-then-else" syntax. Instead, it has the function of case separation for values via `case-esac`, and substitutes for the condition by multiple `case` statements (Figure 7).

```
if expr then a else b;
```

(a) Conditional statement in Melasy.

```
case
  expr : a;
  1    : b;
esac;
```

(b) Expansion in SMV to a case statement.

Fig. 7. Example of expansion of an "if-then-else" description.

4) Expansion of an Array: The current version of the Melasy compiler cannot handle multidimensional arrays. We are developing the next version, which will have this ability. Because SMV cannot handle multidimensional arrays, they must be expanded into one-dimensional arrays. For example, SMV cannot handle this description:

```
next(value[0]) = value[n].
```

In SMV, access using a variable for the suffix of an array is not allowed, but there is no problem if "value[0]" is treated as an identifier such as "value_0". Therefore, the compiler handles multidimensional arrays by treating all the elements of an array as a set of identifiers whose suffixes are given.

B. An Example of SMV Code Generation and Verification

An example of SMV code being generated from Melasy code is shown in Figure 8. In this example, an N-bit counter is defined. The Melasy code has three components. The "CounterCell" component oscillates between 0 and 1 as a simple flipflop. The "Counter" component generates N CounterCell instances, namely "values[N]", with guard areas. The first cell connects with a clock, and the following cells connect with the previous cell's carry output. Then, in the "Main" component, the instance of a "Counter" component is generated for "N = 4", and oscillates as a clock (Figure 8(a)).

The SMV code generated by Melasy is also shown, in Figure 8(b). This SMV code is executable by NuSMV-2.4.

Figure 8(c) shows the result of 10 simulated steps. At this stage, the counter state has become "0101" in binary. As the clock cycle takes two steps, this result is correct.

```
component CounterCell(carryIn, reset)
 var
   value :: Bool = 0;
 assign
   value = if reset then 0 else value^carryIn;
 define
   carryOut = value & carryIn;

component Counter<N>(clk)
 var
   values[N] :: CounterCell
          | N==0      (clk,0),
          | otherwise (values[N-1].carryOut,0);

component Main()
 var
   clk   :: Bool = 0;
   cnt10 :: Counter<4>(clk);
 assign
   clk = if clk then 0 else 1;
```

(a) Melasy source code.

```
MODULE CounterCell(carryIn, reset)
VAR
        value : boolean;

ASSIGN
        init(value) := 0;
        next(value) :=
  case
    reset : 0;
    1 : value xor carryIn;
  esac;

DEFINE
        carryOut := value&carryIn;

MODULE main()
VAR
        clk : boolean;
        cnt10 : Counter_4(clk);

ASSIGN
        init(clk) := 0;
        next(clk) :=
  case
    clk : 0;
    1 : 1;
  esac;

MODULE Counter_4(clk)
VAR
        values_0 : CounterCell(clk, 0);
        values_1 : CounterCell(values_0.carryOut, 0);
        values_2 : CounterCell(values_1.carryOut, 0);
        values_3 : CounterCell(values_2.carryOut, 0);
```

(b) Automatically generated SMV code.

```
NuSMV > simulate 10
*******  Simulation Starting From State 1.1   *******

NuSMV > print_current_state -v
Current state is 1.11
clk = 0
cnt10.values_0.value = 1
cnt10.values_1.value = 0
cnt10.values_2.value = 1
cnt10.values_3.value = 0
```

(c) Verification result using NuSMV-2.4.

Fig. 8. Example of SMV code generation and verification.

VII. Conclusions

It is possible to define components that can be used flexibly via the use of templates. With an implicit `foreach` and guard area, it is possible to substitute for a complicated initialization a description that uses case separation rather than an enumeration of assignment statements. We believe that in the future, libraries that will improve the descriptive power of Melasy could be developed easily, by devising methods to facilitate the design of user-defined libraries and types. The Melasy compiler can output SMV code at present, and we plan to enable it to generate additional HDL codes in the future. We believe that a metalanguage that contains only functions similar to those of existing languages can be created. This will be achieved by limiting the codes directly generated from the Melasy code to just the Melasy intermediate code, and then expanding an implicit `foreach` and/or template during the code generation step. In this way, we plan to enable the Melasy compiler to generate a variety of codes.

References

[1] N. Wirth. *Digital Circuit Design*. Springer, New York, NY, USA, 1985.
[2] Y. N. Patt, S. J. Patel, M. Evers, D. H. Friendly, and J. Stark. One billion transistors, one uniprocessor, one chip. *IEEE Computer*, 30(9):51–57, 1997.
[3] A. V. Aho and J. D. Ullman. *Principles of Compiler Design*. Addison Wesley, Boston, MA, USA, 1977.
[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Boston, MA, USA, 1986.
[5] H. Trickey. Flamel: A high-level hardware compiler. *IEEE Trans. on Comp.-Aided Design of Int. Circ. and Sys.*, 6(2):259–269, 1987.
[6] IEEE. *VHDL (VHSIC Hardware Description Language)*. IEEE Design Automation Standards Committee, 1076, 2002.
[7] D. E. Thomas and P. R. Moorby. *The Verilog(r) Hardware Description Language*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.
[8] T. Grotker. *System Design with System-C*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.
[9] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Boston, MA, USA, 2000.
[10] K. L. McMillan. *The SMV system (Symbolic Model Verifier)*. Cadence Berkeley Labs, Berkeley, CA, USA, 1999.
[11] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: A new symbolic model verifier. In *Proc. of 11th Conference on Computer-Aided Verification (CAV'99)*, pages 495–499, 1999.
[12] S. P. Jones, C. V. Hall, K. Hammond, and W. Partain. The glasgow haskell compiler: A technical overview. In *Proc. of the Joint Framework for Information Technology Technical Conference*, 1993.
[13] D. Leaden and E. Meier. Parsec: Direct style monadic parser combinatory for the real world. Technical report, EU-CS-2001-35, Utrecht University, Netherlands, 2001.
[14] *HDCaml*. http://www.confluent.org/wiki/doku.php/hdcaml.
[15] *Confluence*. http://www.confluent.org/wiki/doku.php/confluence.

Katsumi Wasaki received the B.E. and M.E. degrees in information engineering from the faculty of engineering, Shinshu University in 1991 and 1993, and the Ph.D. degrees in engineering from Shinshu University in 1997, respectively. He is an associate professor in the department of information engineering, faculty of engineering, Shinshu University since 2001. His current research interests include fault tolerant, system stability and formal verification of parallel systems. He is a member of IEEE, IEICE, IPSJ and IIEEJ.

Naoki Iwasaki received the B.E. degrees in information engineering from the faculty of engineering, Shinshu University in 2007. He is a student in the graduate school of science and technology, Shinshu University since 2007. His current research interests includes software engineering, hardware compiler, generic programming, formal verification and model checking systems. He is a member of ACM and IEICE.