HDL Interoperability: A Compiler Technology Perspective

Serafin Olcoz
TGI S.A.
olcoz@tgi.es

Paul Menchini
Menchini & Associates
mench@mench.com

efforts towards HDL interoperability. Finally, section 6 presents the conclusions of this work.

2. Software and Hardware Compilers

The term "software compiler" emphasizes the action of this kind of compiler in translating a high-level computer programming language (HLL) description [2], written in a source language, into a lower-level (target) language [3-5]. The target language usually is an assembly language or machine code, depending on whether the compiler works together with an assembler to produce machine language or directly generates the machine code.

![Figure 1.- HLL Compiler architecture.](image)

A software compiler consists of five essential components: the scanner, the parser, the contextual constrainer, the code generator and the optimizer; see figure 1. The first three are part of the compiler front-end and are collectively referred to as the analysis phase. The job of the front end is to recognize a valid input source program. The compiler back end, comprising the remaining components, is concerned with generating and optimizing code for the target machine. This compiler phase is sometimes also called the synthesis phase.

The data structure passed between the analysis and synthesis phases is called intermediate data representation or format (IF). The IF contains all the information that the compiler is able to collect or infer during the analysis phase, as well as additional instructions inserted by the compiler to implement certain dynamic semantic checks. A well-designed IF contributes greatly to efficient maintenance, and continued development by facilitating a high degree of independence of the front- and back-end phases. In the case of an interpreter, the back-end is the part that produces the computed results, although the term is not generally applied to interpreters.

An emergent approach consists of producing an ANDF (Architecture Neutral Distribution Format) [6], as a
common IF representation. Such a common IF makes software written in any HLL portable to any computer architecture. These new compilers produce, after the analysis phase, ANDF from each particular IR. The ANDF is then the common input to a number of back ends, one for each target architecture.

The scanner, or lexical analyzer, reads the source text file in which the HLL program is written. The file is read as a string of characters, from which a stream of words (or lexemes) and symbols (called tokens) is created by the scanner. Rules to specify valid lexemes are called lexical rules, which are often expressed by a formalism called a regular expression grammar. The tokens generated by the scanner are passed to the parser, which recognizes the phrase structure of the source language and builds an abstract syntax tree (AST). Part of the meaning of a sentence in almost any language is the structure or order in which the lexemes occur in that sentence. Rules that specify lexeme order are called syntax rules, and the set of all syntax rules is called a grammar. Parsers are also named syntax analyzers.

Programming languages have many rules dealing with requirements far beyond lexical or syntax rules. These requirements are called static semantic rules or context-sensitive syntax. The contextual conrainer enforces type and declaration (scope) rules, leaving decorations on the AST.

The compiler designer defines the particular sequences of instructions in the target language that correctly translate each identifiable node in the IF. The execution of these sequences of instructions creates the behavior (meaning or semantics) of the program. The code generator is responsible for translating the IF representation of the source program into the native code of the target computer hardware. This hardware is designed according to a variety of performance and architectural specifications; some of them have profound effects on the efficiency and kind of code that must be generated for the same source program.

Different back ends produce target programs having differing performances for the same HLL program. It is possible to generalize this concept and to consider a compiler (cross-compiler) with several back ends, each of which is targeted for a specific computer architecture (including the one executing the own compiler).

Target code generated by a compiler can be very bad, especially if the code generator has been constructed in either a naive or an ad hoc manner. To improve the target code, modifications can be made to the IF before code generation begins. This task, carried out by the compiler optimizer, often results in greatly improved target code.

Computer architecture and compiler design should be tightly coupled tasks in order to develop a good trade-off between the cost and performance of a digital computer. A computer (Instruction Set Processor, ISP) is a special-purpose hardware that fetches, decodes, executes and possibly modifies instructions stored in memory. Applications can be implemented using ISPs and software from two different perspectives: a software approach and a hardware approach.

Programmers consider code as the program implementing an application. They can work without revisiting the decisions made during the design of the target computer. This simplification allows programmers to use an abstraction of the hardware and to approach the problemsolving task independently of the specifics of the target machine. Hence, software engineering has evolved to more abstract ways of thinking, ones that are more natural to human beings (e.g., object-oriented paradigms) than machine code, which corresponds to the register-transfer (RT) level of hardware descriptions.

Hardware designers consider software as a second level of control that can customize the hardware (existing data and control paths) underlying to the computer. From this perspective, a software compiler can be considered as a tool for automatically generating the programmed control part of the ISP. Software development can thus be considered as a form of digital hardware development at a higher level of abstraction than the RT level.

An intermediate perspective, between these two pure perspectives, appears when HLLs and compiler techniques are used together to describe and produce (synthesize) the hardware.

2.1 HLL-Based Hardware Synthesis

HLLs were originally developed for the purpose of programming computers. But they can also be used to produce application-specific hardware [7-8]. Such HLLs are often called Computer Hardware Description Languages (CHDLs or HDLs) [9-11]. The semantics of an HDL description is shown by its hardware implementation, where the semantics of an HLL program is its computer execution.

Hardware described by means of these languages can be processed to produce hardware in a similar way as HLL programs are compiled to produce the target code. CHDL compilers have a front end similar to software compilers. The IF and the back-end of the HDL compiler are generally quite different from their HLL counterparts. Instead of generating the target program in terms of a previously chosen machine code, HDL back ends produce an HDL description of the desired hardware at a low level of abstraction (e.g., from Register Transfer, RT, to gate level). This process can be iterative, passing through several abstraction levels or skipping some of them, and at the end of the chain the results are used to manufacture the specific hardware described by the HDL description.

Software compiler back ends need to define the target instruction set; they must also define the procedure used to translate the IF to the target language. Constraints are
used to guide the optimization task. Hardware synthesis tools need to define the HDL, or the format corresponding to the target hardware (i.e., the technology library of components to be used as hardware building blocks), and the constraints (area, timing, testability, power consumption, and other physical limitations) to be respected during the synthesis process. Constraints are used to guide the optimization and mapping of a hardware design towards feasible realizations.

VLSI technology offers the possibility of integrating a whole system, previously implemented in a PCB (Printed Circuit Board), on a chip. This additional capability allows the development and use of high-end (e.g., 32-bit) microprocessor-based controllers for industrial applications. It is possible to design and develop, using RT synthesis tools, such complex systems because 32-bit microprocessors, co-processors, cache memories, USARTs, timers and the like, which were previously available only as discrete components, are now available as HDL synthesizable supercells.

Developing systems on chip allows to deliver new ASSPs (Application-Specific Standard Products) as an alternative to ASIPs (Application-Specific Instruction Set Processors); both solutions are combined with application-specific software design to field a complete system.

Compilers and HDL synthesis tools are based on a similar technology, but HDL synthesis tools are as yet quite primitive when compared with industrial software compilers. In particular, they have yet to be used in the same, automatic way as software compilers are used today. However, the evolution of compiler technology allows us to consider hardware synthesis at a higher level of abstraction than at the RT level. Hardware applications can be described by means of an algorithmic HDL description, similar to an HLL program.

An algorithmic HDL description can be synthesized as an HLL program is compiled. However, instead of producing the program to control an ISP (consisting of both data path and control) they produce a specific data path and control for the application under development.

The higher abstraction levels introduced to manage application complexity, which moved from RT hardware descriptions to software designs, is coming back to the hardware world, appearing as a new hardware abstraction level: the algorithmic level of abstraction for describing digital hardware [12-13].

2.2 Hw/Sw Co-Synthesis

When an algorithmic HDL description is considered as the entry point for hardware synthesis, then it is possible to make a trade-off between realizing such an electronic system by means of dedicated hardware, or by means of software that is going to be executed by a (pre-designed) ISP, or a mixture of both. The problem is particularly acute in VLSI embedded systems, as a solution cannot easily be found either by increasing the computing capability of the ISP or by increasing the software complexity; the code density can be a big constrain for these systems [14].

Describing VLSI embedded systems at the algorithmic level allows the consideration of partitioning between, and integration of, hardware and software (plus an ISP). Hardware-software co-design [15], based on co-synthesis and co-simulation techniques, have opened a new world for developing embedded systems at algorithmic level of abstraction.

2.3 Merging Software Compilers and Hardware Synthesis Tools

Software compilers and HDL synthesizers are evolving very rapidly. A possible realization of the co-synthesis approach is a cross-compiler with several back ends for different target architectures. However, a step further in this direction could be possible by mixing both technologies in order to have a unique synthesis tool that generates both hardware descriptions and the necessary software. This new synthesis tool would be able to synthesize an electronic system described in some (possibly new) CHDL into dedicated hardware and corresponding software, all of which generated by the same compilation. It may be, that a new abstraction level, higher than the algorithmic level, is needed for this new hardware-software compiler technology. A possible solution could come from the software world, where object-oriented techniques [16], which are proving to be very powerful and more natural for the human being, are already being applied.

3. Computer-Based Simulation

HLL programs can be compiled to produce either a specific hardware implementation or a machine-code program to control an ISP. Both approaches are used to provide hardware-software solutions to given applications. HDLs are used to describe hardware, but this statement does not necessarily mean that a HDL description must be compiled into just hardware. A different use of HDLs is in describing an electronic system that is going to be transformed into a software model, see figure 2. When the resulting software is executed under the control of a software application, called an HDL simulator, the software model exhibits the intended behavior of the HDL description. The semantics of HDLs used for simulation purposes are generally not identical to the semantic of HDL descriptions used for synthesis purposes.
Figure 2.- VHDL-based model generation and testing.

The front end of an HDL compiler can be common to both purposes, synthesis and simulation; however, these applications require different back ends. The back end of a simulation model compiler must transform the HDL description into the language used by the HDL simulator. The result may be a program to be interpreted by a simulator, which in turn is executed by a computer; this approach is called interpreted simulation. Alternatively, the HDL description may be compiled into object code and linked to the HDL simulation kernel, with the result being directly executed; this approach is called compiled simulation. The translation from HDL to object code may take place either in one step (in which case the process is called native compiled simulation) or in two steps (from HDL to some intermediate HLL, and finally to machine code).

3.1 Simulation: An experimental technique

Originally, the word simulate meant to feign or sham and had a rather unpleasant implication of counterfeit or duplicitous behavior. In its technical meaning which has evolved within Operations Research, it has come to mean a specific kind of modeling in a computer-implementable form. Taking a broad view, all programming can be seen as modeling of some sort. But to be considered a simulation, there must be some modeling of how the system will develop over time [17]. This requirement can make infeasible the use of the same HDL description for simulation and for synthesis purposes. The precision of the HDL simulation model cannot match that of the hardware resulting from its synthesis. This difference is referred as a semantic gap between the simulation and synthesis semantics of the HDL. Simulation, in the sense the word is used here, is a modeling process where a dynamic reality, either actual or projected, is imitated in terms of computer actions. Simulations are dynamic software models implemented on a computer. However, the hardware produced by the synthesizer is not an imitation of the system described by the HDL, but its implementation. Simulation serves as a direct means of observing the hypothetical system's behavior under a variety of conditions, with a chosen initial state. A simulation run can be repeated as often as required to establish confidence in its predictions. Simulations generally require fewer abstractions and assumptions than are required by a mathematical model. In simulation, there is no solution in the mathematical or formal sense; what is gained is an understanding of the relationship between the components of the system and some feel for selected aspects of its average behavior. The practice of simulation is thus an experimental technique: a completely rigorously defined, predictive calculus is abandoned for a trial-and-error methodology in which policies and capabilities can be assessed.

3.2 Time Advance

Having established that the essence of simulation is the modeling of a dynamic system, i.e., one that evolves in time, the first consideration must be how to represent the forward movement of time, as a backdrop to the changes of state in the representation of the object system. For all types of simulation, the mechanism of time advance is the prime mover for all other effects.

To represent time flow in a digital computer, it is necessary to use a numeric variable to store successive values. Whether it is an integer or a real value for time depends on the type of time advance envisaged for the simulation. Basically, it depends on whether the time is constrained to a discrete set of points in a temporal continuum, or whether one admits the possibility that a state change could occur at any point in the continuum.

In a discrete time system the model advances to each successive stage in a series of jumps, the time being incremented by fixed amounts. After every clock tick the value of time is increased by a constant, leading to a synchronous time advance. One problem in implementing a simulation of a synchronous time advances is that all time-dependent happenings occur at instants when other happenings are also taking place, so an intermediate and fictitious state space must be implemented to contain the changes before committing the changes to actuality. The states of the model are then necessarily discrete, persisting between instants, although if they are close enough together they may approximate a continuum.

3.3 Discrete-Event Simulation

Discrete-event simulation (DES) does not stipulate in advance the value of the time increment; on the contrary, this value is determined individually for each time step, based on the component actions of the model.

The components of a model are usually known as entities. These entities are all discrete objects, each being separate from all the others. Entities may be in two kinds of state: either they are busy, engaged in some activity, or they are
idle, doing nothing but waiting in a queue. A queue in this
context does not literally mean a number of entities lined
up one behind another. It means a common state in which
a number of entities find themselves when they have
finished one activity, but before moving on to the next.
When an entity moves from an activity to a queue, or vice
versa, the state of the system changes. This instant of
change is known as an event. Discretisation of time is thus
implicit in the system itself, rather than being explicitly
imposed by the simulator. The simulation program is thus
concentrated on the events interleaved between activities,
and is relatively unconcerned about the periods of active
work.

A DES is a model of a dynamic system which is subject to
a series of instantaneous happenings, or events. The
events themselves arise out of actions within the
simulation carried out at previous events, and are used as
the basic elements to drive the simulation through a
developing sequence of state-changes. The future events
which have yet to occur are stored in the Future Event Set
(FES). Time is advanced by two primitive operations on
the FES: first, the set is assessed to provide the next event
(i.e., the future event with the minimum occurrence time);
second, newly derived events may be stored away
(scheduled) to await future occurrence. DES is thus a kind
of event-driven programming. The DES simulator is a
program that performs the following simulation cycle.
(The cycle is repeatedly executed after the simulation
model has been initialized.) The simulation cycle consists
of: time advancement, event computation, and execution
of processes. Most HDLs simulation semantics, particularly
those specialized on digital hardware
descriptions, are based on DES.

3.4 Strategies of Event-Driven Simulation

Simulations are often quite intricate programs and rarely
can one go straight from a conception of a system to its
programmatic representation. Each simulation language
adopt a particular simulation strategy; with the chosen
strategy unavoidably affecting the methodology and
implementation of the simulation program. The three main
strategies developed for event-driven simulation are: event
scheduling, activity scanning and process interaction. The
last strategy is the one most used by HDLs, e.g., VHDL
[1], and Verilog [18].

4. A VHDL Compiler

The abstract of the VHDL Language Reference Manual
(LRM) states that VHDL is a formal notation for use in all
phases of the creation of electronic systems. Because it is
both machine readable and human readable, it supports
the development, verification, synthesis, and testing of
hardware designs; the communication of hardware
design data; and the maintenance, modification, and
procurement of hardware. VHDL is a CHDL, and to be
used for any hardware description purpose, it has to be
compiled in the same way as any other HDL.

4.1 VHDL Descriptions

VHDL allows one to describe an electronic system by
means of a design hierarchy. The root and internal nodes
of the hierarchy structural descriptions; the leaves of the
hierarchy are behavioral descriptions. VHDL descriptions
are independent of the abstraction level and the design
chosen by the VHDL designer.

The design entity is the basic abstraction in VHDL: it
represents a part of a system that has a well-defined
interface by means of its entity declaration. A design
entity also performs a well-defined function described by
means of its architecture. A design entity always
comprises an entity declaration and exactly one
architecture; however, a given entity interface may have
multiple architectures corresponding to it, as is later
discussed.

A VHDL design hierarchy is composed from design
entities by means of component instantiation. Each
component may be bound to a lower-level design entity in
order to define the structure or the final behavior of that
component. A design entity can also be considered as a
block hierarchy containing concurrent statements. A
special concurrent statement is the process statement,
which in turn contains sequential statements describing
the desired behavior by means of an algorithm. Most of
VHDL's sequential statements are quite similar to any
HLL, but there are two special statements, the signal
assignment and the wait statement, used to communicate
between processes by passing messages. Messages are
named transactions in VHDL terms; each transaction is a
tuple whose first component is a value and whose second
component is a time (projected time). When the new value
of a signal, given by a transaction at its corresponding
time, is different from the old one, then it is said that there
is an event on this signal.

The signal assignment statement introduces new
transactions into a queue of the targeted signal. The wait
statement suspends the execution of a process until certain
conditions occur. Such conditions may be any
combination of events on signals, specified conditions, or
until a given time has elapsed. These features define the
simulation semantics of VHDL. Message passing was the
only processes interaction mechanism allowed in
VHDL'87. However, VHDL'93 has included the
concurrent processes communication mechanism
alternative to message passing, i.e., the use of shared
variables.

VHDL syntax, similar to other HLLs, is quite similar to
Ada. VHDL also has other features common to other
HLLs. For instance, it is a strongly typed language, with
an extensive user-definable typing capability; its packages
provide a means to define resources (e.g., types and subprograms) in a way that allows different design units to share the same declarations. For example, the packages STD.STANDARD, STD.TextIO, and IEEE std_logic_1164 provide predefined sets of types and operations on those types.

4.2 VHDL Analysis

A VHDL description is a textual representation that follows the syntax rules of the LRM. The textual description is stored in one or more text files (design files). The content of a design file is sequentially organized in one or more design units (DUs). A DU can be an entity declaration, an architecture body, a configuration declaration, a package declaration, or a package body.

The DUs of a given design file are sequentially and independently analyzed in order to check the correctness of its syntax and static semantics. The DU context is defined by as all the DUs referenced by it. The result of a DU analysis is essentially an AST, where all name resolution and type matching has been done and where a number of attributes have been set. The resulting AST is stored as a Library Unit (LU). This AST is usually called a VHDL Intermediate Format (VIF) description.

A Design Library System (DLS) is an implementation-dependent storage facility for LUs. The LUs are organized in several Design Libraries (DLs), but the VHDL analyzer always inserts the resulting LUs, of a given design file, into a distinct DL called WORK.

DUs can use information belonging to other DUs, if so, then referenced DU must be analyzed in advance. If a DU is reanalyzed all LUs are potentially affected by such a change and become obsolete. Such LUs must be reanalyzed before they can be used again. Thus, VHDL analysis provides the "built-in" basic support for applying configuration management (in the software sense) to the VHDL descriptions stored in the DLS. This facility enhances design reusability of VHDL descriptions.

VHDL analysis is defined by the LRM as an intermediate step of VHDL compilation, although the access to the VIF is not defined by the LRM. VHDL analysis is not just an implementation-dependent decision taken by the VHDL tool developer in order to carry out an incremental compiler construction. Such a decision could also be applied to a Verilog front end, but it has no language or description level impact.

VHDL analysis can be considered as the minimum common analysis, in the sense of a compiler front end, to be performed to a VHDL description. This VHDL-specific analysis provides a set of LUs useful for simulation, synthesis or any other purpose for which a specific back end is available, see figure 3.

![Figure 3.- VHDL Analysis and Elaboration.](image)

The VIF cannot be used as the IF to interface with the VHDL compiler back ends for simulation, synthesis and other purposes. The reason is that there is not a common IF useful for any purpose. Each given back end needs a specific IF, and an elaboration step from VIF to this specific purpose IF is needed. However, VHDL analysis defines an intermediate compilation step to allow all VHDL-based tools to share the same front end independently of it purpose. The lack of this step in Verilog and other HDLs makes this VIF the wrong place to establish an interoperable platform among HDLs.

4.3 VHDL Elaboration

The only VHDL elaboration considered by the LRM is the one used to produce a simulation model. This model does not depend on the description style or on the abstraction level chosen by the designer.

To elaborate this simulatable model, the VHDL designer must choose the design entity (i.e., the LU of a given configuration declaration, or the LUs corresponding to an entity declaration and one of its architecture bodies) representing the root of the design hierarchy to be simulated in order to complete the contextual constraining in the closed context provided by the chosen root DU. However, VHDL simulators can support implementation-dependent mechanisms to communicate the simulation model with some external system's description.

The design entity to be elaborated represents the complete electronic system to be simulated. The elaboration transforms the design hierarchy into a flat model consisting of an hierarchy of VHDL processes interconnected by a network of signals [19-20]. Although it is a flat model, the information of the design hierarchy can be recovered through the information attached to the elaborated signals.

The elaborated model can be considered as the program of an event-driven simulator based on the process interaction simulation strategy. From the compiler point of view, the result of the elaboration is the IF of an event-driven simulation model. This IF can be interpreted by a VHDL simulator, or can be translated into native code and then linked to the program corresponding to the VHDL simulator, thus performing a VHDL compiled simulation.
A common IF could be defined to be shared by VHDL and other HDLs based on the same simulation strategy, i.e., Verilog; see figure 4. The extended VHDL elaborated IF can be considered as a particular ANDF-like description applicable to any HDL with an event-driven simulation semantics based on the interactive process simulation strategy. This approach solves the simulation interoperability problem that exists between HDLs; it also enhances the portability of generated software models between simulators, operating systems, and computer architectures.

Different elaborations could be defined taking into account a different semantics (e.g., for synthesis) than the one defined by the LRM. These elaborations would produce particular IFs suitable for the specific back ends. However, implied in this approach is the definition of CHDL synthesis and the other semantics. Then and only then, a common IF could be defined for each purpose (e.g., synthesis).

5. Current standardization approaches to HDL interoperability

Currently, there are two standardization approaches to HDLs interoperability:

- **AIRE (Advanced Intermediate Representation with Extensibility)** [21], supported by DARPA (Defense Advanced Research Projects Agency) and the U.S. Air Force’s Wright Laboratories; and

- **OMI (Open Mode Interface)** [22], developed and supported by OMF (the Open Model Forum).

AIRE provides a C++ object-oriented description of an AST-based representation of VHDL grammar that claims, by means of its extensibility, to support all compilation steps for simulation and synthesis purposes of VHDL (digital, 87, 93, and the upcoming 98, and analog versions), Verilog and other HDLs. AIRE consists of two coordinated specifications: The Internal [21] (IIR), and File Intermediate Representation (FIR), for AST representation in memory and in file respectively. It can be considered as the second effort to define a standard VIF, although with a scope much larger than the first effort’s [23], which only considered the AST resulting from VHDL analysis.

In principle, IIR’s extensible approach is so general that it could be applied to process any HLL, including those not based on event-driven simulation semantics. It could also include different semantics for the same HDL.

The definition of FIR, when available, will clarify AIRE’s suitability to manage the differences between VHDL analysis and elaboration with respect to Verilog and other HDLs. However, the existence of FIR means that the advantages offered by persistent object-oriented languages, such as Eiffel [24] has not been considered.

AIRE’s suitability for dealing with simulation and synthesis is quite difficult to predict. This standardization approach is still in its first stages, and there is a long way to go before it is completely defined and sponsored by an recognized standards body (e.g., the Institute of Electrical and Electronics Engineers, IEEE, or the Electronics Industrial Association, EIA) and adopted by the major ESDA (Electronic Systems Design Automation) tool vendors.

OMI defines two ANSI C interfaces to previously compiled simulation models coming from several HDLs, in particular VHDL and Verilog. It is also possible to write OMI-compliant models in C or C++. The intention of the OMF group, under the CFI (CAD Framework Initiative) umbrella, is to provide only a common simulation interface between event-driven simulation models and HDL simulators or other applications dealing with these simulation models. OMF is driven by major ESDA and silicon vendors, and offers a clear channel to its industrial acceptance. To do so, OMF plans to submit the results of its work to the IEEE in order to become an IEEE standard.

OMF/OMI interfaces will protect the IPR (Intellectual Property Rights) of HDL descriptions in the same way as executable software. However, AIRE does not address the IPR issue, although the APIs to the representations can be as closed as desired.

From a technical point of view, it could be possible to extend AIRE to support OMI-compliant tools and models, considering AIRE as really extensible specification of ASTs able to deal with HLL grammars. It is too early to guess the future of both standardization initiatives as they have been working for just a bit more than a year. The complete process can still take one or more additional years.

---

1 Based on this IEEE initiative, TGI has developed a VHDL’93 Intermediate Format in Eiffel language, as part of the development of a VHDL Integrated Common Environment (VHDL-ICE). This VIF has been submitted to AIRE and CEN/CENELEC standardization bodies to contribute to the definition of an Open Standard VIF.
6. Conclusions

This paper has explored the common foundations between HLLs and HDLs and has proposed additional avenues for exploitation of these similarities. Particularly, the work has been focused on the interoperability between HDLs from a compiler technology perspective. Stating the need of providing IFs for HDLs sharing the same semantics or purpose (i.e. simulation and synthesis) as a solution for having different syntax and description capabilities. Possible future papers will describe additional work being done in this area.

References

[22] Open Model forum, "Open Model Interface Draft Standard", Version 0.9, April 1996.