An Overview of Formal Hardware Specification Languages

Annette Bunker*
School of Computing
University of Utah
Salt Lake City, Utah 84112
abunker@cs.utah.edu

Sally A. McKee
School of Electrical and Computer Engineering
Cornell University
Ithaca, New York 14853
smckee@aya.yale.edu

Ganesh Gopalakrishnan
School of Computing
University of Utah
Salt Lake City, Utah 84112
ganesh@cs.utah.edu

Abstract

Verification is widely recognized as one of the most difficult aspects of computer hardware design. The gap between design and verification capabilities grows, as does the cost of missed flaws. Many researchers investigate ways to formally verify processor designs, interconnects, and protocols, but creating verification methods and tools will remain a central problem for computer scientists for at least the next decade. We introduce this field by surveying formal specification languages. We construct a taxonomy of languages and discuss the applicability of each to standard compliance verification, or demonstrating that a hardware design complies to an interconnect standard.

1. Introduction

Verification is one of the most difficult aspects of computer hardware design. The cost of missing even a single bug can be staggering. For instance, the floating point division bug in the Intel® Pentium® processor cost the company $475 million [4]. Validation efforts for modern microprocessor designs can consume months of CPU time on large server farms, yet the total "runtime" for the simulated design during this period is equivalent to only a few minutes of wall-clock time on a 1 GHz CPU [2]. Furthermore, the gap between industrial design capabilities and verification capabilities is growing, rather than shrinking, as is the potential cost of bugs that escape verification teams.

In response to this problem, researchers and practitioners alike strive to create methods and technologies that can help close the design/verification gap. Current formal verification techniques target many aspects of computer systems, including microprocessor design [12, 15, 26], interconnect design, protocol design [11, 14], and specification validation. However, none of the techniques nor tools yet addresses the problem of verifying that register transfer language (RTL) designs comply with interconnect standards.

This paper surveys several formal specification languages that are candidates for our application of interest, interface standard compliance verification. As in any engineering effort, the tools chosen for a verification project can greatly affect its eventual success or failure. One of the most important tools of a formal verification engineer is the language used to specify the design. Having a broad view of the specification languages available allows the practitioner to refine the candidate language set to those most appropriate to the problem to be solved.

To that end, we present a taxonomy of the specification language space, and we use this taxonomy to structure our discussion of the relative merits of each approach. We classify specification languages first according to their form, i.e., textual versus graphical or visual. We make finer classifications according to origins: HDLs versus traditional software programming languages in the textual case, and software engineering versus hardware engineering versus system engineering in the visual case. This short survey is not intended to be comprehensive, but instead serves to familiarize the reader with the variety of languages used in practice and their applicability in our domain.

The ideal language for our purposes possesses four characteristics, as outlined below. While we realize it is unlikely that an existing language exactly matches our application, we prefer to make adjustments to a language that already exists rather than developing a new language.

- **Precise semantics**. A formal specification provides a precise baseline for many implementations developed by different organizations.
- **Short learning curve**. The specification is likely to be used by hardware engineers of all specialties, and thus the language must be easy to learn and understand, even by those not highly trained in formal verification techniques.
- **Connections with automatic verification techniques**. Because the same verification may be performed by multiple organizations (the intellectual property designer and integrator, for example), the verification effort must be significantly less than the total design effort. Languages that lend themselves to automatic verification techniques help mitigate verification effort.
- **Integration with existing design practice**. The transfer of formal methods technologies to industrial practice depends on their ability to fit neatly into existing or emerging design practices.

*This work was supported by National Science Foundation Grant CCR-9987516, and was conducted while the second author was at Utah.

2. Textual Languages

Textual languages constitute the majority of current specification languages. They resemble hardware description languages and conventional programming languages, and therefore lend themselves to automated analysis. We divide such approaches broadly into those inspired by hardware description languages, which tend to describe designs in low-level detail, and those arising from programming languages, which tend to specify behavior at a higher abstraction level.

2.1. Hardware Description Languages

We first focus on two HDL approaches, hardware monitors and Objective VHDL. Hardware monitors have been used to specify the PCI 2.2 standard and the Intel Itanium™ bus protocol. Objective VHDL was developed to support abstraction and reuse in hardware modeling.

Hardware Monitors. The use of monitor modules as aids for simulation-based verification was proposed as early as 1996 [19]. Shimizu, et al., retarget monitors for formally specifying hardware designs [25], showing that even the most complicated hardware protocols can be described in this manner [24]. Such a specification consists of a series of small monitor modules written in the same hardware description language as the implementation, where each monitor describes one requirement in the specification. Monitors used for hardware specification tend to be of two types: a small counter machine for an event which must happen within a certain number of cycles of a trigger event; or a flag machine that remembers one piece of information, such as whether the transaction is a read or a write.

Monitors possess capabilities especially useful for protocol design and verification. First, they can be checked for internal consistency using standard model checking tools, without the aid of a completed implementation. Second, they can be reused for many verification purposes. Finally, monitor-style specifications can be proven to be implementable, as demonstrated by Shimizu and colleagues.

Monitor specifications do not require the designer to learn a new language to create the specification, since they are written in hardware description languages. Furthermore, writing the specification in the same HDL as the implementation means that the specification is already fully integrated into the design cycle and is supported by appropriate tools. However, while portions of common HDLs have been formalized, we are aware of no complete formalization of any HDL. As a result, a given monitor specification may mean different things when run on different HDL simulators. Finally, hardware monitor specifications can be difficult to read, since HDL descriptions are often difficult for humans to parse, and since related requirements may be located in physically remote portions of the specification. Specifications used in the standard compliance domain must be amenable to human comprehension, as they must provide a baseline for many implementations.

Objective VHDL. Objective VHDL (OVHDL) raises the abstraction level at which design activities occur, enhancing design reuse [23]. This object-oriented version of VHDL adds type classes for data abstraction and entity classes for structural abstraction. Type classes allow the user to create class data types, such as arrays or records, with inheritance and data hiding. Since VHDL entity-architecture pairs already implement data hiding, OVHDL developers only add inheritance to complete the implementation of entity classes.

Allara, et al., point out that designers require only minimal training in using the additional OVHDL syntax [1], and automatic formal verification tools exist for which VHDL is the modeling language. The OVHDL tool suite translates models into VHDL that can be simulated by off-the-shelf tools, facilitating integration with existing design practice. As with hardware monitor specifications, the semantics of OVHDL models depend on the chosen VHDL simulators.

Figure 1(a) shows how a hypothetical counter module might be specified using OVHDL. The inputs `inc` and `reset` increment and reset the counter's value, respectively. In Figure 1(a), the counter class is declared with a single data attribute, `count`, of type integer. Procedures are then declared to handle the two inputs and implemented to handle the `inc` input and the `reset` input.

2.2. Programming Languages

Next we consider SpecC and Java™ for hardware specification. SpecC encourages design reuse by providing a single, executable language in which to document many design activities. Java, a high-level, object-oriented programming language, developed at Sun Microsystems®, enables hardware/software codesign by providing a single simulation environment for the entire system.

SpecC. SpecC is a formal, executable modeling language with an accompanying tool suite intended to facilitate hardware-software codesign [8]. The language retains much of the syntax of the C programming language. At each stage in the SpecC design cycle, tools generate models that can be analyzed by typical simulation and debugging tools. Other SpecC tools provide static analysis and estimation to ensure that design metrics remain within specified bounds. SpecC's formal semantics and familiar syntax combined with the complete design methodology surrounding the language make it a viable candidate for use in compliance verification. Automatic verification support, the language's one missing requirement, could be supplied.

Figure 1(b) specifies our counter in SpecC. The keyword `behavior` indicates a behavioral specification, rather than a structural model. A single variable holds the current value of the counter. When `inc` or `reset` is received, the counter value is updated appropriately.

Java. Java enters the hardware specification arena as a hardware/software codesign platform. By using the same language for hardware description and software programming, the entire system can be simulated at once [17]. Class

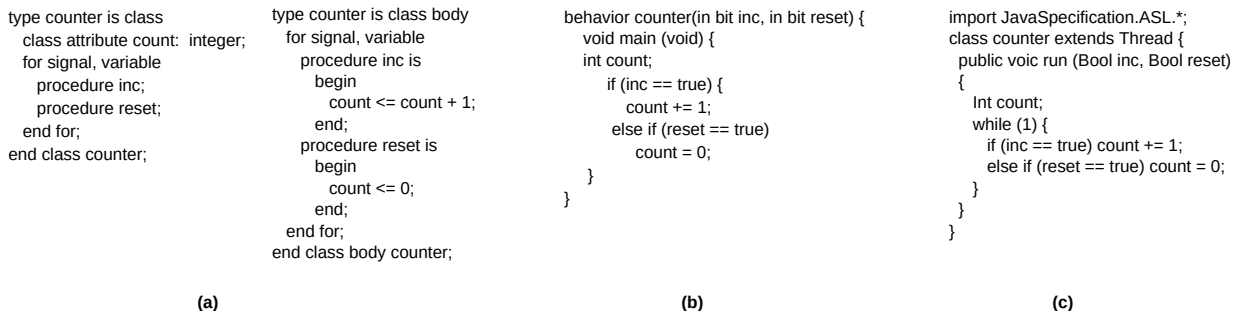

Figure 1. Counter specified in three textual languages: (a) Objective VHDL, (b) SpecC, and (c) Java.

libraries implement ports, signals, and other hardware design components needed for simulation.

Java is easy to learn, but is of limited use for compliance verification. First, static data flow and control analysis is extremely difficult [10], which makes it hard to assign a formal semantics. Second, lack of formalization makes developing formal tools based on the language of debatable value. Third, we are aware of no industrial design practices into which Java-based methods fit easily.

Figure 1(c) specifies our counter in Java, first importing the RTL simulation support library. The counter class extends the Java Thread class and executes as an independent thread. The counter description contains only on method `run`, which continuously waits for an input to fire and takes the corresponding action.

Hardware monitors, Objective VHDL, and Java are inadequate for verifying interface specification conformance, since they share the property that a specification may have different meanings when evaluated with different tools. SpecC therefore appears the most promising of the textual languages considered for formal compliance verification.

3. Visual Languages

Visual notations comprise a small number of available specification languages, but are typically easy to learn. We divide such approaches into those stemming from software engineering practices such as object-oriented design methods, traditional hardware description and documentation, and the newer field of system engineering.

3.1. Software Engineering

Three languages make up our discussion of specification languages developed to aid software engineering: Harel's Statecharts; Message Sequence Charts, as maintained by the International Telecommunication Union; and Specification and Description Language, a visual language maintained by the SDL Forum. We treat each in turn, below.

Statecharts. Statecharts [9] extend state machine notation by adding mechanisms for expressing hierarchy, concurrency, and communication, as shown in Figure 2(a). Rounded rectangles represent states; arrows denote transitions; and filled and banded pseudo-states represent start and final states, respectively. Statecharts support hierarchical models, keeping descriptions of large systems tractable for the user. Concrete states can be composed sequentially or concurrently to form abstract states. Hierarchical tool support makes navigating even large models manageable.

Since designers usually use state machines as their mental model of a design, Statecharts are easily learned and integrated into industrial practice. Tools exists for using Statecharts in the traditional design cycle and for automatic formal verification. Unfortunately, assigning semantics to Statecharts is difficult. Some methods require a global view of the description [22]. Lüttgen, et al., present a newer, compositional method that somewhat mitigates this problem, but is complicated to use [18].

Message Sequence Charts. The International Telecommunication Union (ITU) maintains a formal definition and semantics for Message Sequence Charts (MSCs) [13]. MSCs' strength lies in their ability to describe communication between cooperating processes. Figure 2(b) shows a single process, denoted by a rectangle containing a process identifier. The process life line extends downward. Arrows represent messages passed from a sending to a receiving process. Messages not starting or ending at a process life line are exchanged with users, be they human or mechanical.

MSCs represent hardware communication intuitively. Because of this they are very easy to understand and are already used in some informal protocol specifications [21]. To the discredit of the community, however, MSCs are often used in ways that contradict the standard semantics. The standard requires that all behaviors described by the MSC be present in all possible traces produced by the system, but users often specify behaviors that should be displayed by at least one system trace using MSCs. Krüger, et al., call these semantic interpretations universal and existential interpretations, respectively [16]. This issue serves to make the already difficult task of analyzing MSCs [20] even more problematic. We evaluate MSCs as a potential compliance verification modeling language in greater depth elsewhere [3].

Specification and Description Language. The Specification and Description Language (SDL) is an ITU standard promoted by the SDL Forum Society [6]. Similar to Statecharts in its notation and usage, SDL also extends simple state machines with communication notation, as shown in

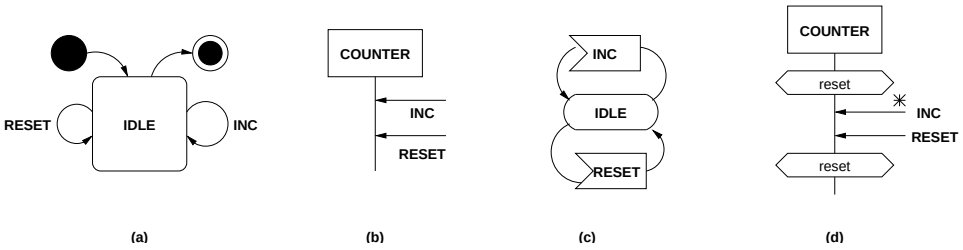

Figure 2. Counter specified in four visual languages: (a) Statecharts, (b) Message Sequence Charts, (c) Specification and Description Language, and (d) Live Sequence Charts.

Figure 2(c). Again, the diagram specifies a counter with one state and two inputs. Rectangles with concave points represent messages received, and rectangles with convex points denote messages sent (not shown).

Since it has a formal definition, SDL is typically used for mission-critical systems, such as aerospace and complex real-time systems. An extension of traditional state machine notation, the language is easy to learn. Tool support and design methodology are available due to the strong European user base. The major drawback in using SDL for protocol specification, however,is that protocol specification descriptions tend to be large, and therefore difficult to understand and maintain.

Of the three languages originating in the software engineering discipline, none stands out as a particular candidate for our application. Statecharts and Message Sequence Charts both have semantics-related problems that are not easily overcome and SDL has shortcomings where human understanding is concerned.

3.2. Hardware Engineering

State machine diagrams, waveforms and circuit schematics are typical visual specification languages used in hardware design. The logic language we discuss here incorporates all three of these elements with a mathematical model of the physical implementation.

Heterogeneous Hardware Logic. Fisler presents a logic composed of hardware description diagrams [7]. This logic includes diagrammatic notations for circuit diagrams, algorithmic state machines (ASMs), and timing diagrams. Further, Fisler develops a rigorous mathematical model with which to represent physical device implementations of specifications, which no previous specification logic does. The verification engineer constructs proofs using the visual components in this logic, whereas other visual languages require translating the diagrammatic representations into a sentential form for manipulation. Including the logical representation of the physical implementation lets proofs of high-level properties extend to the lowest implementation level.

The heterogeneous hardware logic uses model representations taught to first-year engineering students, so no retraining is required and the models produced are already in use in industrial practice. The language is also fully formalized. Its only drawback is that it is better suited to human-intensive verification techniques than to automatic ones. For purposes of protocol compliance, proof tools that require little human intervention are preferable, since they facilitate quickly repeated verification by integrators of intellectual property (IP) components.

3.3. System Engineering

Finally, we consider the recently-proposed Live Sequence Charts language, which its developers use to specify an automatic rail-car system. The combination of formality with this richly expressive visual notation make the language an exciting prospect.

Live Sequence Charts. Harel and Damm's Live Sequence Charts (LSCs) extend Message Sequence Charts in several ways [5]. First, LSCs allow the user to specify behaviors as being required or optional, solving the universal/existential interpretation problems facing MSCs and giving users the equivalent of an if-then-else mechanism from a traditional programming language. When used in conjunction with the multiplicity symbol (an asterisk), optional behaviors represent loops. These features allow users to describe several alternative scenarios in a single chart, making LSCs scale better than MSCs. Furthermore, Live Sequence Charts contain a mechanism to enforce preconditions and postconditions on the charts, easing the computational complexity of analysis. These features, along with a complete formal LSC semantics, make LSCs attractive as a formal specification language for compliance verification.

Figure 2(d) demonstrates the Live Sequence Charts counter specification. We require that the counter be in a reset state prior to execution of the chart, as shown by the condition bar across the process life line. Likewise, the chart leaves the system in a reset state, as specified by the postcondition. Finally, we show that `inc` may occur zero or more times by applying the multiplicity operator to it.

The LSC language's main weakness, which lies outside our original evaluation criteria, is its lack of a timing model. This reduces the language's facility for describing hardware protocols. Like MSCs, LSCs rely on the partial order imposed by the order in which events occur along a life line

and in which messages are passed between processes. In order for the language to formally specify hardware protocol standards, a full timing model must be added. This could be accomplished by enhancing events to allow full, first-order assertions relating to a clock process. This addition or the timing model that Harel and Damm plan to create [5] would make LSCs the most promising candidate for compliance verification. We therefore expect to use one of these augmented forms of Live Sequence Charts for formally specifying hardware protocol standards and for verifying compliance of an RTL design in our research.

4. Conclusions

Choosing an appropriate specification language for a particular verification problem is important. Familiarity with many such languages and their strengths and weaknesses makes the choice more informed, if not more tractable. In the abbreviated survey presented here, we consider a variety of specification languages as candidates for protocol compliance verification. We are preparing a more comprehensive treatise addressing several additional languages. The textual language category includes SystemC, Haskel, Esterel, e, and Vera, and the visual category includes the UML, regular timing diagrams, and symbolic timing diagrams.

The growing complexity of the verification problem presently outpaces the growing power of our tools, and current approaches represent but partial solutions. As more designers include intellectual property developed by others in their own designs, the availability of decisive standard compliance verification techniques becomes critical. Specification languages must adapt to meet the demands of this burgeoning problem domain, and one part of our broader research agenda attempts to determine the specifics of how to effect these changes. At present, we find that Live Sequence Charts augmented with a concrete timing model provide the most promising approach to verifying standard compliance; the heterogeneous hardware logic and SpecC show potential for this application, as well.

References

[1] A. Allara, M. Bombana, P. Cavalloro, W. Nebel, W. Putzke, and M. Radetzki. ATM Cell Modelling Using Objective VHDL. In *Proceedgings of Asia and South Pacific Design Automation Conference*, pages 261–264, 1998.

[2] B. Bentley. Validating the Intel® Pentium® 4 Microprocessor. In *Proceedings of the 38th Design Automation Conference*, pages 244–248, 2001.

[3] A. Bunker and G. Gopalakrishnan. Using Live Sequence Charts for Hardware Protocol Specification and Compliance Verification. In *IEEE International High Level Design Validation and Test Workshop*, 2001.

[4] B. Colwell and B. Brennan. Intel's Formal Verification Experience on the Willamette Development. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1869 of *LNCS*, pages 106–107, 2000.

[5] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, pages 45–80, 2001.

[6] J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL: Formal Object-oriented Language for Communicating Systems*. Prentice Hall, 1997.

[7] K. Fisler. A Logical Formalization of Hardware Design Diagrams. Technical Report 416, Indiana University, 1994.

[8] D. D. Gajski, J. Zhu, R. Doemer, A. Gerstlauer, and S. Zhao. The SpecC Methodology. Technical Report ICS-99-56, Department of Information and Computer Science, University of California, Irvine, 1999.

[9] D. Harel. StateCharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[10] R. Helaihel and K. Olukotun. Java as a Specification Language for Hardware-Software Systems. In *Proceedings of the 1997 International Conference on Computer-Aided Design*, pages 690–697, 1997.

[11] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1990.

[12] R. Hosabettu. *Systematic Verification of Pipelined Microprocessors*. PhD thesis, University of Utah, 2000.

[13] International Telecommunication Union. *ITU-T Recommendation Z.120: Message Sequence Chart (MSC)*, 1999.

[14] M. D. Jones. *Formal Verification of Parameterized Protocols on Branching Networks*. PhD thesis, University of Utah, 2001.

[15] R. B. Jones. *Applications of Symbolic Simulation to the Formal Verification of Microprocessors*. PhD thesis, Stanford University, 1999.

[16] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to Statecharts. In *Distributed and Parallel Embedded Systems*. Kluwer Academic Publishers, 1999.

[17] T. Kuhn and W. Rosenstiel. Java Based Modeling And Simulation Of Digital Systems On Register Transfer Level. In *Workshop on System Design Automation*, 1998.

[18] G. Lüttgen, M. von der Beeck, and R. Cleaveland. A Compositional Approach to Statecharts Semantics. In *Technical Report of ICASE, NASA Langely Research Center, Hampton, VA*, 2000.

[19] J. Monaco, D. Holloway, and R. Raina. Functional Verification Methodology for the PowerPC microprocessor. In *Proceedings of the 33rd Design Automation Conference*, pages 319–324, 1996.

[20] A. Muscholl and D. Peled. Analyzing Message Sequence Charts. In *2nd Workshop on SDL and MSC*, 2000.

[21] OCB Design Working Group. *Virtual Component Interface Standard*. Virtual Sockets Interface Alliance, 2000.

[22] A. Pnueli and M. Shalev. What is in a Step: On the Semantics of Statecharts. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *LNCS*, pages 244–264, 1991.

[23] M. Radetzki, W. Putzke, W. Nebel, S. Maginot, J.-M. Berge, and A.-M. Tagant. VHDL Language Extensions to Support Abstraction and Re-use. In *Workshop on Libraries, Component Modeling and Quality Assurance*, 1997.

[24] K. Shimizu, D. L. Dill, and C.-T. Chou. A specification methodology by a collection of compact properties as applied to the intel itanium processor bus protocol. In *Correct Hardware Design and Verification Methods: 11th IFIP WG 10.5 Advanced Research Proceedings*, volume 2144 of *LNCS*, Livingston, Scotland, September 2001. Springer-Verlag.

[25] K. Shimizu, D. L. Dill, and A. J. Hu. Monitor-Based Formal Specification of PCI. In W. A. Hunt and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design*, volume 1954 of *LNCS*, pages 335–352, 2000.

[26] P. J. Windley. Formal Modeling and Verification of Microprocessors. Technical Report LAL-91-03, University of Idaho, 1991.