

Formal Hardware Specification Languages for Protocol Compliance Verification

ANNETTE BUNKER, GANESH GOPALAKRISHNAN

University of Utah

and

SALLY A. MCKEE

Cornell University

The advent of the system-on-chip and intellectual property hardware design paradigms makes protocol compliance verification increasingly important to the success of a project. One of the central tools in any verification project is the modeling language, and here we survey the field of candidate languages for protocol compliance verification, limiting our discussion to languages originally intended for hardware and software design and verification activities. We frame our comparison by first constructing a taxonomy of these languages, and then by discussing the applicability of each approach to the compliance verification problem. Each discussion includes a summary of the development of the language, an evaluation of the language's utility for our problem domain, and, where feasible, an example of how the language might be used to specify hardware protocols. Finally, we make some general observations regarding the languages considered.

Categories and Subject Descriptors: B.4.3 [**Input/Output and Data Communications**]: Interconnections—*Interfaces*; B.4.5 [**Input/Output and Data Communications**]: Interconnections—*Hardware reliability*; B.7.2 [**Integrated Circuits**]: Design aids—*Verification*; C.2.2 [**Computer-communication Networks**]: Network Protocols—*Protocol verification*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*data types and structures*

General Terms: languages, formal verification

Additional Key Words and Phrases: e, Esterel, Heterogeneous Hardware Logic, hardware monitors, Hierarchical Annotated Action Diagrams, Java, Lava, Live Sequence Charts, Message Sequence Charts, Objective VHDL, OpenVera, Property Specification Language, SpecC, Specification and Description Language, SystemC, SystemVerilog, Statecharts, timing diagrams, The Unified Modeling Language

This work was supported by National Science Foundation Grants CCR-9987516 and CCR-0219805.

Authors' addresses: Annette Bunker and Ganesh Gopalakrishnan, School of Computing, University of Utah, 50 S. Central Campus Drive, Rm 3190, Salt Lake City, UT 84112, USA, email: bunker@acm.org, ganesh@cs.utah.edu; Sally A. McKee, School of Electrical and Computer Engineering, Cornell University, 324 Phillips Hall, Ithaca, NY 14853, USA, email: sam@csl.cornell.edu.

A preliminary version of this paper appears as "An Overview of Formal Hardware Specification Languages" in the Grace Hopper Celebration of Women in Computing, October 2002.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 20Y ACM 1084-4309/20Y/0400-0001 \$5.00

1. INTRODUCTION

The emergence of the system-on-chip (SoC) development paradigm presents designers with new challenges in all aspects of the design process. Predictably, verifying a design consisting of intellectual property (IP) from several organizations is monumental among these challenges. Our research addresses this issue by investigating the possibilities of formally verifying that an interface complies with a published standard, and by investigating methods for performing such a verification. In this paper, we survey several formal specification languages that are candidates for interface standard compliance verification.

As in any engineering effort, the tools chosen for a verification project can greatly affect its 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 set to further investigate only those most appropriate for 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 similarities, differences and 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: hardware description languages (HDLs) versus traditional software programming languages versus temporal logic languages, in the textual case, and software engineering versus hardware engineering versus systems engineering, in the visual case.

Since we focus on tools for formal protocol compliance verification, we restrict our survey to languages that explicitly support concurrent interactions occurring at component interfaces. As this is not a sharp selection criterion, we supplement it with two secondary criteria. First, where we consider a language that emphasizes multiple features that enable protocol specification, we facilitate comparison by also including other languages emphasizing similar features. Second, we focus our discussion on recent developments in this field, avoiding, for instance, discussion of Scheme-based hardware description and many older logic languages.

We include Objective VHDL for its emphasis on message passing and its descent from a traditional hardware description language. Likewise, hardware monitor approaches and SystemVerilog grow directly from traditional simulation and emphasize protocol compliance verification. We include SpecC, SystemC, and Java because of their similarity to C, which should make them easy for most engineers to learn, and for their interface specification facilities. Industrial interest for verification applications of all sorts in e, OpenVera and Esterel merits their inclusion. Lava takes a radically different approach by using multiple interpretations of the same specification for various design and verification activities. Property Specification Language's foundations in formal temporal logics merit its inclusion. We consider Message Sequence Charts for their message passing model, and Statecharts for their expressive state machine notation. The Unified Modeling Language incorporates both of these, along with other diagrammatic notations. In addition, engineers find it useful in embedded and real-time systems. Explicit communication provided by a state machine model makes Specification and Description Language of interest to our survey. We include timing diagrams, Hierarchical Annotated Action Diagrams, and the Heterogeneous Hardware Logic for their formal foundations and for their hardware-specific specification capabilities. Finally, we include Live Sequence Charts for their strengths in specifying message passing and for other features that aid in formalizing specifications.

No single language is likely to match our application exactly, but we prefer to adapt an existing language, rather than to develop a new language. The ideal language for our purposes has four characteristics that we use to evaluate the candidate languages.

- Precise semantics.** Our application requires a language with a complete formalization. Because the specification provides a baseline for many implementations developed by different organizations, a precise semantics must be available.
- Short learning curve.** The language must be easy to learn and understand, even by those not highly trained in formal verification techniques. A well established, formal compliance verification system is likely to be used by engineers of all specialties, not just formal verification engineers.
- Connections with automatic verification techniques.** Because the same verification may be performed by both the IP designer and the IP integrator, the verification effort must be significantly less than the total design effort. Because the organizations involved may be contractually bound, the verification must also state definitively whether or not the IP block complies with the standard. Furthermore, the results of the verification must be reproducible. Languages that lend themselves to automatic verification techniques are more likely to satisfy these requirements.
- Integration with existing design practice.** The transfer of formal methods technologies to industrial practice depends on their ability to fit into existing or emerging design practices. The recent explosion of new design and reuse methods offers many opportunities for integrating formal compliance verification methods with accepted design practices.

The discussion of each language follows a simple outline. First, we summarize the syntax of the language and offer some basic background on its development. Next, where feasible, we present and discuss an example specified in the language. To facilitate comparison, we confine our examples to a single data-handshake protocol. In this protocol, an initiator process drives data on a data bus, then asserts a flag telling the target process that the data is ready. Upon receiving the validation signal, the target reads the data off the bus and asserts an acknowledgment signal, allowing the initiator to proceed with subsequent data transfers. Finally, we evaluate the language along the four axes listed above, highlighting any additional results particularly relevant to our problem domain.

2. TEXTUAL LANGUAGES

Textual notations are traditionally used as formal specification languages. They resemble hardware description languages and conventional programming languages, and therefore lend themselves to automated analysis. We divide textual approaches broadly into those inspired by hardware description languages, which tend to describe the design in low-level detail, and those arising from programming languages, which tend to specify behavior at a higher level of abstraction.

2.1 Hardware Description Languages

We first focus on three common hardware description language approaches, Objective VHDL, HDL monitors, and SystemVerilog. Objective VHDL was developed to support abstraction and reuse in hardware modeling. Hardware monitors have been used to specify the PCI 2.2 standard and the Intel® Itanium™ bus protocol. Accellera drives the recent additions to the IEEE Verilog standard known as SystemVerilog.

2.1.1 *Objective VHDL*. Several proposals for extending VHDL with object oriented elements have been developed [Swamy et al. 1995; Ashenden et al. 1997; Cabanis et al. 1996; Zippelius and Müller-Glaser 1992], but we focus our discussion on Objective VHDL (OVHDL) [Radetzki et al. 1997]. The language transforms VHDL into an object oriented language by adding data abstraction in the form of type classes, and structural abstraction in the form of entity classes. Type classes allow the user to create data types such as arrays or records, accompanied by the usual inheritance and data hiding associated with object oriented languages. VHDL entity-architecture pairs already implement data hiding, leaving only inheritance and message passing to be added by the OVHDL developers.

The message passing mechanism in OVHDL is designed to allow users maximum modeling flexibility. It consists of three portions: a communication structure, a protocol, and a dispatcher [Putzke-Röming et al. 1998]. A communication structure represents a communication pathway or channel between two objects. The protocol describes messages that may be passed along the channel and details of the signaling involved. The dispatcher listens for incoming messages and invokes the appropriate method call.

A case study reported by Allara et al. [1998] shows that the modularity of designs, and hence their reusability, can be enhanced by modeling in Objective VHDL. The original design of their ATM cell model consists of twenty processes in two architectures, while the OVHDL version contains eight entity classes, making maintenance easier and facilitating reuse of more abstract components. The case study indicates that designers require only minimal training in using the additional OVHDL syntax. Automatic formal verification tools for which VHDL is the modeling language also exist [Bell Labs Design Automation and Lucent Technologies 1998]. The Objective VHDL tool suite translate the model into VHDL, which can then be simulated by off-the-shelf tools. The potential for integration with existing design practice is quite good, although the semantics of an Objective VHDL model depend on the VHDL simulator chosen.

2.1.2 *Hardware Monitors*. The use of monitor modules as aids for simulation-based verification was proposed as early as 1996 [Monaco et al. 1996]. Shimizu et al. [2000] retarget monitors to formally specify hardware designs, showing that even the most complicated hardware protocols can be described in this manner [Shimizu et al. 2001]. 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 that must happen within a certain number of cycles after 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, including automatically creating testbenches for simulation [Shimizu and Dill 2002]. Finally, specifications written in the monitors style can be proved implementable, as demonstrated by Shimizu and colleagues.

Due to space limitations, we provide only a partial specification of our handshake protocol, presenting an example of how one property might be specified using monitors. Figure 1 shows how the counter machine might be described, specifying that the `ack` must happen within four cycles of the generation of the `valid` signal. While we use Verilog

```

module counter (timer, trigger, clear, max, clk);
  output timer;
  input trigger;
  input clear;
  input [BITS-1:0] max;
  reg [BITS-1:0] count;

  initial begin count = 0; end

  always @(clk) begin
    if (clear) count = 0;
    else if (trigger) count = count + 1;
    else if (count > 0) count = count + 1;
  end;
endmodule

module spec (correct, data, valid, ack, clk);
  output correct;
  input data;
  input valid;
  input ack;
  input clk;

  // ack must be received within 4 cycles of valid
  counter ack_4_clks (correct, valid, ack, 4, clk);
endmodule

```

Fig. 1. Data transfer example specified in the monitors style.

as the base language for this monitor specification, it could be written in any hardware description language. Typically, many different `correct` signals would be output from many different modules, each named to identify both the module the monitor watches and the line item in the specification that it represents. This way, when a correctness flag is raised during simulation, the source is easily identified, and the issue is easily diagnosed.

Monitor specifications do not require the designer to learn a new language to create the specification, since monitors are written in hardware description languages. Furthermore, using the same HDL as the implementation means that the specification is already fully integrated into the design cycle, complete with tool support. However, while portions of common HDLs have been formalized, we are not aware of a complete formalization of any HDL. As a result, a given monitor specification can mean different things when run on different HDL simulators. Also, HDL descriptions can be difficult for humans to parse, and related requirements may be located in physically remote portions of this type of specification.

2.1.3 SystemVerilog. SystemVerilog, adopted by Accellera, seeks to promote convergence of design and verification tasks. The language, an extension of the IEEE 1364 Verilog standard, adds facilities to Verilog for defining and verifying blocks at a high abstraction level, writing assertions, and creating test-benches [Sutherland 2003]. Based on language donations from several industrial leaders, the language includes a rich collection of programming language features [Sutherland 2002; Accellera Organization, Inc. 2002], including object oriented classes, dynamic class creation, garbage collection, abstract and user-defined datatypes, pass-by-reference variables, and semaphores, mailboxes and events for interprocess communication.

Though the example in Figure 2 does not make use of the object oriented features of SystemVerilog, it does illustrate another major addition to the language, an interface. In its simplest form, an interface may be used to bundle groups of ports for reuse in the design. The `handshake` interface in our example, on the other hand, defines two tasks and a function for manipulating those ports, `send`, which outputs the data onto the wires and then sets the `valid` flag, `invalidate` which deasserts the signal passed into it as a parameter, and `receive`, which asserts `ack` and returns a copy of `data` to the calling module. The initiator module first includes the ports defined by the `hshake` interface in its port list, giving it access to all members of the interface. After initialization, it checks for a new data value at each clock cycle, sending it via the interface, when one arrives. Once the acknowledgment returns, it calls `h.invalidate()`, to deassert the `valid` signal. After

```

interface hshake
  bit data, valid, ack;

  task send (input d, output data, valid)
    data = d;
    valid = 1'b0;
  endtask

  function bit receive (input data, output ack)
    ack = 1'b1;
    return data;
  endfunction

  task invalidate (output sig)
    sig = 1'b0;
  endtask
endinterface

module init (hshake h, input clk, input rst)
  bit d;
  always @(negedge rst) h.invalidate(h.valid);
  always @(posedge clk) begin
    if (<new_data>) h.send(d, h.valid);
    else if (h.ack) h.invalidate(h.valid);
  end
endmodule

module targ (hshake h, input clk, input rst)
  bit my_data;
  always @(negedge rst) h.invalidate(h.ack);
  always @(posedge clk) begin
    if (h.valid) my_data = h.receive(h.data, h.ack);
    else h.invalidate(h.ack);
  end
endmodule

```

Fig. 2. Data transfer example specified in SystemVerilog

performing its own initialization, the target module awaits the `valid` signal, then reads the data and asserts `ack` via the interface. Once `valid` deasserts, `targ` deasserts the acknowledgment, also.

The unification of design, testbench creating and assertion-based verification requires fundamental changes to the Verilog simulation semantics [Moorby et al. 2003] to eliminate possible race conditions. The simulation timeslot now contains three new execution regions to allow non-blocking assignments to simulate with the correct semantics alongside assertions and testbench interactions. The new simulation algorithm evaluates events in specific, ordered sets: preponed, active, inactive, non-blocking assignments, observe, reactive, and postponed. The preponed region allows testbench code to access simulation data in the previous timeslot. The active, inactive, and non-blocking assignment regions remain unchanged from the IEEE standard. The observe region evaluates invariant assertions and clocked assertions not previously evaluated. The reactive region executes testbench code, including pass/fail statements included in assertions, and the postponed region allows testbench code to suspend until all other regions complete execution.

Similarities to older versions of the Verilog hardware description language and efforts by tool vendors to make SystemVerilog tools backward compatible, indicate that the learning the language and the accompanying assertion language and programming language interface should be simple. The wide belief that SystemVerilog will eventually become the IEEE Verilog Standard and the strong support of the language by major design and verification tool vendors [Goering 2003], even prior to its acceptance by the IEEE, imply that tools and methods for design will soon begin to appear.

2.2 Programming Languages

Next, we consider modeling techniques based on programming languages. First, we discuss several languages based in the C paradigm: SpecC, SystemCTM, and Sun Microsystems' [®] JavaTM. Then we consider two object oriented languages: e, which is marketed by Verisity, Inc.TM, and OpenVeraTM, which is sponsored by Synopsys Inc.[®], followed by another imperative language, Esterel. Finally, we consider the functional language Lava, which is based on Haskell.

2.2.1 SpecC. SpecC is an executable modeling language intended to facilitate hardware-software codesign [Gajski et al. 1999; Gajski et al. 2000; Dömer et al. 1998] currently supported by the SpecC Technology Open Consortium. The Consortium maintains a language reference manual [Dömer et al. 2002] and a reference compiler. The SpecC Methodology

```

interface HandShake {
  void Send(int i);
  int Receive(void);
};

channel HS implements HandShake {
  int Data;
  event Valid, Ack;
  void Send(int i) {
    Data = i;
    notify Valid;
    wait Ack; }
  int Receive(void) {
    int i;
    wait Valid;
    i = Data;
    notify Ack;
    return i; }
};

behavior Init(HandShake Port) {
  void main(void) {
    int d;
    d = <data assignment>;
    Port.Send(d);
  }
};

behavior Target(HandShake Port) {
  void main(void) {
    int d;
    d = Port.Receive();
    return d;
  }
};

behavior Main(void) {
  HS C;
  Init I(C);
  Target T(C);

  int main(void) {
    par {
      I.main();
      T.main();
    }
    return 0;
  }
};

```

Fig. 3. Data transfer example specified in SpecC. Adapted from Handshaking1.sc example by Rainer Dömer which accompanies the SpecC Reference Compiler.

consists of four activities, which occur at a level of abstraction higher than the register transfer level [Dömer and Gajski 1998]. First, an initial executable specification describes functionality, performance requirements, power goals, and cost constraints. Next, architectural exploration refines the specification by allocating resources, scheduling resources, and partitioning behavior. Communication synthesis, the third step, inserts communication protocols and synthesizes interfaces. Finally, software compilation and hardware synthesis generate the implementation models. Support tools generate models at each stage in the design cycle, and these models may be processed by typical simulation and debugging tools. Also, static analysis and estimation can be performed after each step to ensure that the design metrics remain within bounds established by the initial specification.

Central to the SpecC design methodology is the separation of communication from computation [Dömer and Gajski 2000], as shown in the data exchange example in Figure 3. The specification begins by declaring an interface `HandShake` which contains two methods, `Send` and `Receive`. Next, the keyword `channel` indicates the specification of a communication channel. In this case, the channel `HS` implements the `HandShake` interface by declaring a variable to contain the data transmitted as well as events for signaling the validity of the data and acknowledgment of receipt and then implementing methods for sending and receiving data. The method `Send` first places the data for transmission on the channel's port, then signals the `Valid` event, then waits for an `Ack` event from the receiver. Correspondingly, the `Receive` method copies the data off the port once it receives a `Valid` event then acknowledges receipt and returns the data to its caller.

The remainder of the code fragment specifies the behavior of the initiator process, the target process, and a parent process of both, `Main`. In this example, this specification style illustrates to some extent the intended structure of the system, but this is not always the case. The `Init` behavior sets the data for transmission, calls the `Send` method of the channel and then exits. Likewise, the `Target` behavior only reads the data element returned by the channel into the variable `d` and returns it to its caller. Finally, `Main` ties the system together by running `I.main` and `T.main` in parallel. This example illustrates the ease with which one might change communication protocols in the SpecC Methodology without changing the functionality of the communicating processes.

Because of its connections with the C programming language, we expect SpecC to be easy to learn. Support for turning various SpecC models into analogous models in con-

ventional simulation languages allows the SpecC Methodology to integrate easily into existing design practices. This same trait means that state-of-the-art verification tools and techniques exist for SpecC designs. The recent creation of a formal execution semantics for SpecC largely resolves formalization concerns [Mueller et al. 2002].

2.2.2 *SystemC*. Maintained and promoted by the Open SystemC Initiative (OSCI), the SystemCTM language provides an executable modeling environment intended for aiding hardware-software codesign, much like SpecC. The language and modeling platform are built in layers on top of C++. Architectural layers include the core language, a separate data type unit, an elementary channels unit (describing such as signals, mutexes, and FIFOs), and a second channels unit that includes more complex models, such as Kahn Process Networks [Kahn 1974] and master/slave libraries. SystemC supports other computational models, as well, such as Static Multi-rate Dataflow, Dynamic Multi-rate Dataflow and Communicating Sequential Processes.

SystemC modeling may take place at any of several levels: gate level, register transfer level, or the transaction level, which is higher than the register transfer level [Bhasker 2002]. Transaction level modeling (TLM) describes the system as a series of read and write transactions, simplifying the modeling effort and speeding simulation. After initial validation, the SystemC model can be partitioned into hardware and software components and passed to the appropriate teams for further development. The SystemC model is executable, providing software teams with a development platform as well as a simulation environment early in the hardware design cycle.

Five features represent the core circuit modeling language [Drechsler and Große 2002]. First, modules behave as the main structural entity. Modules may contain submodules, functions, data, and ports. Second, processes describe intended functionality. Next, ports provide a communication mechanism between modules. Ports may be unidirectional or bidirectional and they can send or receive data from other modules. Signals model wires and represent the interconnection of modules, carrying data between them. Finally, clocks, which are special kinds of signals, keep system time during simulation. Core language features likely to be added include dynamic thread creation, thread forks, thread joins, thread interrupts, thread aborts, timing constraint specification, support for real time modeling and support for scheduler modeling [Swan 2001].

The SystemC user's modeling view is similar to that provided by a hardware description language, even though the programming language constructs are based in C++. This makes hardware design akin to algorithm development. SystemC's relationship to C++ makes it easy to learn, and its object oriented encapsulation and inheritance capabilities make designs modular, which in turn facilitates their reuse. The language and simulation platform are accompanied by a well defined semantics in the form of a reference implementation, which is widely available [Open SystemC Initiative 1999]. Work in progress strives to complete a formal simulation semantics [Mueller et al. 2001]. A wide variety of commercial and non-commercial tools available for the language include simulators, translators which convert SystemC models into models written in other languages, as well as translators which convert other HDL models into SystemC models, synthesis tools, verification tools, hardware/software codesign environments, and environments designed specifically to aid the design and verification of system-on-chip systems in SystemC.

```

import JavaSpecification.ASL.*;
class initiator extends Thread {
public void run (Bool ack, Bool valid, Bool data)
{
  Bool my_data;
  data = my_data;
  valid = true;
  while(true) {
    if (ack == true)
      break;
  }
  valid = false;
}
}

import JavaSpecification.ASL.*;
class target extends Thread {
public void run (Bool valid, Bool data, Bool ack)
{
  Bool new_data;
  while (true) {
    if (valid == true) break;
    new_data = data;
    ack = true;
  }
  while (true) {
    if (valid == false) break;
    ack = false;
  }
}
}

```

Fig. 4. Data transfer example specified in Java.

2.2.3 *Java*. Java enters the hardware specification arena via three separate efforts. All three project teams observe a trend toward describing hardware at higher levels of abstraction, reasoning that the programming language level may be the abstraction level appropriate for next-generation design activities. They also argue that if programming languages are appropriate for use as hardware design languages, then using a single language for both hardware and software description makes sense. By using the same language as the hardware description language and the software programming language, the entire system can be simulated at once [Kuhn and Rosenstiel 1998].

In one project, class libraries support hardware simulation by implementing hardware design components such as ports and signals [Helaihel and Olukotun 1997]. This research inspires the Java specification of our data handshake example in Figure 4. The specification first imports the RTL simulation support library. The initiator class extends the Java Thread class and executes as an independent thread. The counter description contains only one method, `run`, which sets `data` equal to the contents of `my_data`, which may be previously set as the user chooses. `run` then asserts `valid` and awaits an `ack` signal, at which time it deasserts `valid`. The target class also extends the Java Thread class and runs as a separate thread, which busy-waits on the assertion of `valid`, reads the value of `data` into its internal variable, and then acknowledges receipt. Finally, the target again waits for `valid` to be deasserted, so it can deassert `ack`.

The second project targets Java specifically for embedded systems that must operate deterministically within bounded resources [Young et al. 1998]. Java programs cannot guarantee such bounded operation, and so restrictions must be made to the language. A method called successive formal refinement aids the user in turning an arbitrary Java program into one that can guarantee operating characteristics using the Abstractable Synchronous Reactive computation model.

The third project considers two separate interpretations of the Java description: the structural and the behavioral [Kuhn et al. 1999]. In the structural interpretation, objects can be mapped to structural components of the circuit, while the behavioral interpretation considers them to be connections between components. Again, to accurately represent hardware in this framework, limitations must be placed on the language. Dynamic data structures and floating point arithmetic are the most notable prohibitions.

Its wide acceptance in the software domain means that many engineers already know and use Java. Wide acceptance also means that tools are beginning to emerge, such as the Java PathFinder [Visser et al. 2000; Brat et al. 2000], which may be adapted for use in hardware verification. The difficulty in performing static dataflow and control analysis

presents a problem with respect to using Java for hardware verification. This, in turn, relates to the difficulty of assigning formal semantics to the language, making it difficult to determine the precise meaning of a specification written in Java. Without the ability to determine the exact meaning, the specification is of limited use for formal protocol compliance verification.

2.2.4 *e*. Created with functional verification specifically in mind, the *e* language from Verisity Design Inc.TM allows the user a great deal of flexibility in writing and extending functional specifications [Hollander et al. 2001]. The language incorporates concepts from several programming paradigms with the intent of allowing several users to extend the verification environment in different ways without affecting others' work. Its aspect oriented characteristics enable the user to modularize across structural objects in the system. The inclusion of subject oriented techniques make it amenable to unplanned extensions. Furthermore, *e*'s development draws from the lessons of the adaptive programming and object oriented programming communities.

As might be expected from an industrial language, tool support for *e* is substantial. Verisity's tool suite includes a testbench generator, a coverage measurement tool, and a testbench accelerator, which synthesizes commonly used portions of the testbench to the desired hardware platform. Furthermore, work is underway to link *e* with other industrial verification tools [Santarini 2001].

Test specification in *e* typically employs iterative refinement. Based on the system specification and the test plan, the test environment is developed, optionally linking to HDL or code written in other programming languages [Verisity Design 1999]. The test environment describes behaviors to check and defines the inputs to be generated. A synthesis system translates the environment and inputs to traditional HDL or other programming languages for simulation [Kuhn et al. 2001]. After simulation, coverage analysis feeds information back to the verification engineers, who can then bias the test generator to achieve desired coverage. One engineer might add constraints to the system to bias the test generator toward certain inputs while another engineer biases toward other inputs. Because of the aspect oriented and adaptive programming features of *e*, such multidirectional extensions can coexist without interfering with one another.

e is a powerful, flexible language with growing verification tool support. It is used in industrial verification practices. Having no first-hand experience with this language, we do not know how much training is required to learn it. For instance, specifications written in *e* may be difficult for a design engineer to use as an implementation guide, since the language is designed with back-end verification in mind. Also, we cannot ascertain whether or not the language has a formal basis, though we are aware of plans to integrate ForSpec Temporal Logic, a linear temporal logic created by Intel Corporation, which does have a rigorous formal semantics into the language [Armoni et al. 2002] (see Section 2.2.5 for more information on ForSpec).

2.2.5 *OpenVera*. Marketed commercially by Synopsys, Inc., OpenVeraTM responds to the need for high-level verification language that is easy to learn and use [Synopsys, Inc. 2001]. Syntactically, the language is similar to C++ or Java, including encapsulation and inheritance mechanisms. Executable and capable of modeling concurrent processes, the language and accompanying environment contain multiple facilities for test coverage analysis and augmentation, including user constraints placed on random tests, coverage statis-

```

function bit initiator_data ()
{
  bit data;
  <code initializing data>
  initiator_data = data;
}

function initiator_valid ()
{
  initiator_valid = HIGH;
}

function bit target_ack (bit data, bit valid)
{
  bit my_data;
  if (valid == HIGH)
    my_data = data;
  target_ack = HIGH;
}

program handshake
{
  bit data;
  bit valid;
  bit ack;

  data = initiator_data();
  valid = initiator_valid();
  ack = target_ack(data, valid);
}

```

Fig. 5. Data transfer example specified in OpenVera.

tics that may be queried during simulation, and generated tests that may be changed on-the-fly to enhance coverage. OpenVera also includes temporal property specification features.

Experience in using OpenVera offers several practical lessons [James and Dhamanwala 2000; Bergeron and Simmons 2000]. First, engineers with previous background in object oriented programming tend to learn the language much more quickly than those new to the paradigm. Second, the investment in education and foundation code can make the first OpenVera project proceed slowly, but these investments often pay off in subsequent projects. Third, the higher level of abstraction allows engineers to create more simulations and better design coverage with the same verification effort. Finally, advance planning of the verification method, class hierarchy, and support library is vital to the success of a verification project based on OpenVera.

Figure 5 shows how the handshake protocol example might be specified using OpenVera. Though we do not use any object oriented features in this specification, our naming convention is meant to suggest how this might be done. After some C-like compiler directives and macro definitions (not shown) [Synopsys, Inc. 2001], coverage analysis blocks, subroutines, and classes may be defined. OpenVera supports only two levels of code hierarchy, the top level and the subroutine or class level, with top-level code appearing last in the file. In our example, we first define two functions describing the behavior of the initiator. `initiator_data` specifies the generation of the `data` signal by creating a one-bit variable, `my_data`, initializing it, and then returning it as the value of the function by setting the function name equal to the value of `my_data`. `initiator_valid`, on the other hand, simply returns the value of `initiator_valid` as `HIGH`, a preconstructed OpenVera constant. The function describing the behavior of the target checks the value of `valid`, saves a copy of `data`, and returns the acknowledgment if `valid` is `HIGH`.

With the integration of the ForSpec Temporal Logic (FTL) [Synopsys, Inc. 2002] language from Intel Corporation, OpenVera now possesses the expressive power of a linear-time temporal logic language [Armoni et al. 2002], as well as several other features which make it amenable for the mechanical verification of hardware systems. ForSpec provides the user with a rich set of built-in hardware constructs, such as bit vectors, common logical operators, and common arithmetic operators. It allows the definition of regular sequences of Boolean events, the modeling of multiple clocks and reset mechanisms, and a wide variety of temporal connectives, including those that describe events occurring in the past.

Supported by a major electronics design automation (EDA) vendor, OpenVera is designed to augment the traditional design flow and the rest of the company's tool suite. In fact, commercial designers are using it to augment their verification methods for such projects as memory address controllers [Brunelli et al. 2001], PCI-X implementations [Wang and Wen 2002], and metro-area networks [Krishnamoorthy et al. 2002]. Further-

```

module Initiator;
output data : data_type;
output valid;
input ack;
loop
  var new_data: data_type in
    <compute new_data>
    emit data(new_data);
    emit valid;
    await ack;
  end var
end loop
end module

module target;
input data : data_type;
input valid;
output ack;
loop
  await valid;
  ?data;
  emit ack;
end loop
end module

```

Fig. 6. Data transfer example specified in Esterel.

more, OpenVera is being integrated with other verification languages, such ForSpec and the Unified Modeling Language (UML) [Thompson and Williamson 2002]. Since the level of effort required to learn the language depends on previous experience with object oriented programming languages, and since the language is designed as a simulation-based verification, we are unable to ascertain whether or not a formal semantics exists for OpenVera outside of its temporal logic subset.

2.2.6 Esterel. Esterel is primarily designed to describe reactive systems [2000; 1999]. Esterel models are based on the perfect synchrony hypothesis, which abstracts communication to a zero-delay broadcast action. The language includes a notational style suitable for dataflow applications such as signal processing, and it has an imperative notation style suitable for describing control-intensive applications such as bus interfaces. A diagrammatic notation called SyncCharts, derived from Esterel and Statecharts (see Section 3.1.2), provides a convenient way of expressing reactive behaviors [1996].

Figure 6 shows how the data handshake protocol might be specified using Esterel. Like its counterparts specified in other languages, the example defines two processes, initiator and target. The initiator creates output port `data` and handshake ports `valid` and `ack`. Inside an infinite loop, it creates a local variable `new_data`, computes its value for output, and then outputs it on the `data` port, using the `emit data(new_data)` construct. Finally, it emits the `valid` signal and then waits for an incoming `ack`. The corresponding target process defines its communication ports and then waits to receive a `valid` signal. When it does so, it reads the value off `data` and then outputs the acknowledgment.

The Esterel Studio™ tool suite connected with the language is marketed commercially by Esterel Technologies, Inc. The design methodology cycles through specification, simulation, verification, testing, and code generation [2000]. Esterel Technologies is developing language support specifically tailored for hardware design and verification [2000]. Among the proposed additions to the language, let statements model continuous assignment, and signal vectors aid in modeling situations in which data and control paths are not clearly distinguished. Encoder constructs enable easy exchange between various information encodings. Finally, nondeterminism allows for modeling design environments, while stronger boundaries between module interfaces more closely matches the hardware design being modeled.

The Esterel language is backed by a complete formal semantics, as well as a growing suite of design and verification tools targeted to the hardware design domain. The dual programming language/visual notation nature of the language and accompanying tool suite make it is easy to learn and utilize.

```

request_quiet => EF (data = <data_value>)
(data = <data_value>) => AX (valid = 1)
(valid = 1) => AF (ack = 1)
(ack = 1) => AF request_quiet
data = <data_value> until valid = 0

```

Fig. 7. Data transfer example specified in Property Specification Language.

2.2.7 *Lava*. Lava is a hardware specification and verification language developed at Chalmers University of Technology [Claessen and Sheeran 2000]. Researchers use the language in developing and verifying designs based on Field Programmable Gate Arrays (FPGAs) for applications such as high performance graphics, digital signal processing in high-speed networks, and circuits designed to draw Bezier curves. Implemented directly in the Haskell programming language, Lava consists of various Haskell modules that provide the user with hardware-design facilities.

Lava circuits can be interpreted in a number of ways. The simplest interpretation corresponds to direct simulation. More advanced interpretations allow users to simulate with symbolic data and to generate formulas that relate outputs to the inputs that generate them. Lava's roots in a general-purpose programming language facilitate connecting a circuit description with a theorem prover for deep analysis of the system. Another interpretation generates VHDL code that can be used in simulation, synthesis, or other verification activities. Lava provides this functionality through the use of monads, type classes, polymorphism, and higher order functions, all of which are standard features of functional programming languages [Bjesse et al. 1998].

In spite of efforts to simplify the language [Claessen and Sheeran 2000], we fear that its primary weakness is the steep learning curve. Few engineers have functional programming backgrounds, and they will likely have difficulty working with the advanced constructs required to fully utilize the hardware description features Lava embodies. Even though Lava models can be used in conjunction with theorem provers for verification purposes, using theorem provers requires extensive human intervention. Furthermore, semantics of Lava specifications rely fully on the Haskell compiler or interpreter on which they are executed. The lack of a formal semantics for the language weakens the link between the description and the formal verification tools. However, because the Lava tools can generate standard hardware description language models from Lava specifications, the language can easily be integrated into existing design flows.

2.3 Temporal Logic

In this section, we discuss formal hardware description languages stemming from traditional temporal logics, not all possible uses of temporal logics for hardware specification. One language makes up our discussion of such languages, namely Property Specification Language (PSL). PSL is the temporal assertion language developed by IBM® [Formal Methods Group 2000] and promoted by Accellera [Accellera Organization, Inc. 2003]. The original language extends Computational Tree Logic (CTL) by adding regular expressions and some syntactic sugar which makes the expression of many complicated properties easier and more readable [Beer et al. 2001]. Recent versions of the language combine CTL, Linear Temporal Logic (LTL) and Interval Temporal Logic (ITL) [Gordon 2002]. PSL includes a rich set of constructs including those for facilitating clocked and unclocked evaluation, describing safety and liveness properties, building linear and branching time assertions, and representing behaviors of finite and infinite length.

PSL properties consist of boolean expressions, which describe system state in a single cycle, sequences, which describe series of system states, and temporal operators, which describe the temporal relationships between expressions and sequences. The example properties specified in Property Specification Language shown in Figure 7 make use of the branching fragment of PSL, the portion of the language most like CTL. **EF**, **AX**, **AF**, and **until!** are temporal operators used in this specification. The first property states that after transactions on the interface become quiet, then at some point in the future it is possible that `data` be set to some data value for transmission. The second property then says that setting `data` means that at the next time instant, `valid` must be set, and the third property requires that at some future time after `valid` asserts that `ack` asserts. The next line asserts that at some time after `ack = 1`, the interface must become quiet, again. Finally, the last line specifies that the data value must remain valid until `valid` deasserts and that `valid` must eventually deassert.

The language builds on a well-founded semantic basis and it is amenable to use with formal verification tools. IBM's RuleBase [Beer et al. 1996; Beer et al. 1997] verifies properties specified in Property Specification Language and the FoCs tool generates checker modules for use with HDL-based simulation and verification [Abarbanel et al. 2000]. Furthermore, since the standard proposal names three different syntax styles, or flavors [Accellera Organization, Inc. 2003], which resemble Verilog, VHDL, and Environment Description Language (EDL) [Formal Methods Group 2001], the environmental description language for the IBM formal verification tools, other vendors are developing tools based on the language, as well. Unfortunately, despite attempts to make the syntax intuitive, the language still resembles the temporal logic from which it came and requires considerable background for use.

3. VISUAL LANGUAGES

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

3.1 Software Engineering

We consider four visual notations used for software engineering: Message Sequence Charts, Statecharts, the Unified Modeling Language (UML), and the Specification and Description Language. Message Sequence Charts are the message passing standard of the International Telecommunication Union (ITU). Message Sequence Charts and the Statecharts language are both included in the Unified Modeling Language. The Specification and Description Language is another standard maintained by the ITU.

3.1.1 Message Sequence Charts. The International Telecommunication Union maintains a formal definition and semantics for Message Sequence Charts (MSCs) [International Telecommunication Union 1999b; 1998]. The strength of MSCs lie in their ability to succinctly describe communication among several cooperating processes and the ease with which MSC descriptions may be understood. MSCs precisely show the division of labor between cooperating processes, as well as the ordering of events. While the ITU standard indicates that MSCs show the scenarios that a system may exhibit, in practice they are used to describe behaviors that the system must contain. This dichotomy typically

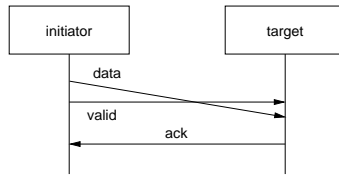


Fig. 8. Data transfer example specified in Message Sequence Charts.

results from the project lifecycle. During requirements-gathering and exploration phases, MSCs tend to show possible executions of the system. As project development continues, the interpretation of the MSCs generally shifts to describe execution segments that all executions of the system must exhibit. In the literature [Krüger et al. 1999], these possible MSC interpretations are called existential and universal.

Figure 8 shows two processes, each denoted by a rectangle containing a process identifier. A process life line extends downward from each process. An arrow represents a message passed from a sending process to a receiving process. In our example, the *initiator* first sends a message containing data to the *target*, after which it sends the message validating the data. The *target* waits until it receives the validation message before it reads the data. Finally, it sends acknowledgment of receipt back to the *initiator*. The time scale of each process is independent of the others', with the exception of the partial order imposed by passed messages on events occurring in the sending and receiving life-lines.

MSCs present four problems with respect to developing a formal protocol description. The first regards the formalization of the language. While the ITU maintains a formal definition, it is insufficient for verification purposes [Klose and Wittke 2001]. The common practice of interpreting MSCs two ways is inherently problematic. Formalization requires a mechanism for distinguishing MSCs for existential interpretation from those for universal interpretation. Note that universal MSCs can be written for an internally inconsistent system [Alur et al. 2000].

Second, MSCs lack the power of expression necessary for describing protocol behaviors. Gunter et al. [2001] shows a widely published protocol example that cannot be expressed using simple MSCs [Tanenbaum 1998], and so they propose High Level Compositional Message Sequence Charts, an MSC variant with the expressiveness required to describe the problematic example. MSCs also lack a mechanism by which the user may specify preconditions or postconditions for a chart. This complicates the task of formalizing MSCs, as there is no precise way to determine when MSCs may be legally concatenated to form a single execution of the system. This formalization challenge can be surmounted by introducing into the language a notation for conditions, and then by defining a semantics for these conditions. A closer study of these issues is presented elsewhere [Bunker and Gopalakrishnan 2001].

Third, MSCs do not exhibit the scalability required for today's complex protocols. Each system behavior requires that a separate chart be drawn for Protocols containing multiple user modes, and transaction options could take many MSCs to describe.

Finally, mechanically analyzing Message Sequence Charts is a difficult problem addressed by a large body of research. Problems considered include understanding why MSCs are harder to analyze than state machines [Muscholl and Peled 1999], the difficulty of embedding MSCs within one another [Muscholl et al. 1998], and determining whether or

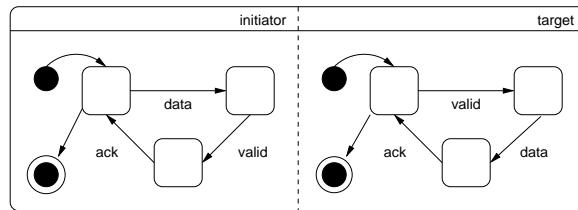


Fig. 9. Data transfer example specified in Statecharts.

not the language of a Büchi Automaton is in the language of an MSC [Muscholl and Peled 2001]. Muscholl and Peled [2000] summarize some of this work, simplifying the proofs. Further, Madhusudan shows that determining whether the set of all MSCs defined by a variant of MSCs, Message Sequence Graphs (MSGs), satisfies a monadic, second-order (MSO) formula is decidable [Madhusudan 2001], and Madhusudan and Meenakshi [2001] extend this result to define a model checking algorithm for MSGs and MSO. Henriksen et al. [2000] investigate an automata-theoretic characterization of regular MSC languages based on bounded message passing automata. They also use second-order logic to prove the problem decidable. Finally, [Alur and Yannakakis 1999] study verifying whether a Hierarchical MSC-graph model satisfies a temporal requirement given by an automaton, and they present algorithms for different cases of the problem.

3.1.2 Statecharts. Proposed by David Harel in the 1980s [Harel 1987], Statecharts extend state machine notation by adding mechanisms for expressing hierarchy, concurrency, and communication. Statecharts allow the user to build models hierarchically, permitting tractable descriptions of large systems. Concrete states can be composed sequentially or concurrently to form abstract states. Hierarchical tool support makes navigating even large models manageable. Like Message Sequence Charts, the Statecharts language has been integrated into the Unified Modeling Language.

Figure 9 specifies our data handshake in this visual formalism. Rectangles with rounded corners represent states which can be decomposed into parallel substates or sequential substates, as shown. The states labeled *initiator* and *target* operate in parallel, expressed as a dashed line dividing the superstate. Inside each, we see three sequentially composed states, which may be named if the user chooses. States may also specify actions to be performed on entry, exit, or during visit to the state. Arrows leading from one state to another represent state transitions, or events, in the naming convention of Statecharts. Events named *data*, *valid*, and *ack* represent the assertion of each signal. Though not shown in our example, the event description language is expressive, allowing users to specify event names, parameters (in case the event represents a procedure call), guard conditions, and actions to be performed on the transition [Rumbaugh et al. 1999]. The dark circle and banded circle represent start and end pseudo-states, respectively. Pseudo-states may not be decomposed, but merely represent the entrance and exit states of a chart.

The graphical notation and similarity to traditional state machines makes Statecharts easy to learn, use, and read. The Statecharts language is commonly used in software development practice, especially in object oriented design methods, and its popularity in the real-time and embedded systems communities is growing. As a result, multiple tools support various system-development activities and use Statecharts as input languages. In-

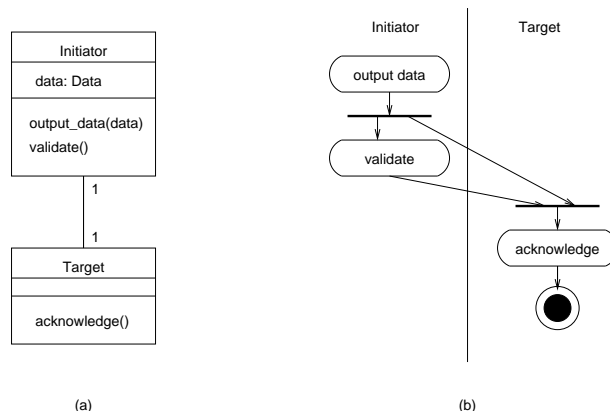


Fig. 10. Data transfer example specified using the Unified Modeling Language's (a) Class Diagram and (b) Activity Diagram.

dustrial tools have been developed to translate Statecharts specifications into executable programs or hardware descriptions suitable for simulation. Furthermore, attempts have been made to link Statecharts directly with ordered binary decision diagrams for use in model checkers [Helbig and Kelb 1994]. Unfortunately, assigning semantics to Statecharts is a difficult problem. Various semantics have been proposed in the literature, with efforts dating at least as far back as 1987 [Harel et al. 1987]. Until recently, assigning precise semantics to a Statecharts model required a global view of the description [Pnueli and Shalev 1991]. Lüthen, et al. develop a newer, compositional method that mitigates this problem, but is complicated to use [Lüttgen et al. 2000].

3.1.3 *The Unified Modeling Language.* The Unified Modeling Language (UML) is a combination of the Object Oriented Design Methods independently developed in the 1980s by Grady Booch, Ivar Jacobson, and Jim Rumbaugh [Booch et al. 1999; Rumbaugh et al. 1999; Fowler 1999], and the standard for the UML is maintained by the Object Management Group (OMG) [The Object Management Group 1999]. The language contains nine different diagram types, from which users may choose to model various aspects of a system. Use Case Diagrams target user requirements gathering. Class and Object Diagrams document the object oriented architecture of the system. Sequence Diagrams (see Figure 8 for one type of UML sequence Diagram, a message sequence Diagram), Collaboration Diagrams, Statechart Diagrams (see Figure 9) and Activity Diagrams all model behaviors. Component Diagrams and Deployment Diagrams specify the physical implementation of the system and the configuration of run-time processing, respectively.

In addition to the previous examples of Message Sequence Diagrams and Statecharts Diagrams, we show two more types of UML diagrams here. Figure 10 illustrates how we might use Class Diagrams and Activity Diagrams to describe our handshake protocol. The Class Diagram in Figure 10(a) describes two class entities, an `Initiator` and a `Target`. For every `Initiator`, there must be exactly one matching `Target`, denoted here by the solid line (class association) drawn between the classes and annotated by singular multiplicities (1s). The `Initiator` class contains a single attribute, `data` of class `Data`, shown in the first subsection of the class. It also contains two operations, `output_data(data)`

and `validate()`, shown in the lower compartment. Similarly, the `Target` class contains no attributes (upper compartment) and only one operation, `acknowledge()`.

The Activity Diagram in Figure 10(b) shows that the `Initiator` is responsible for the `output data` activity and the `validate` activity, while the `Target` is responsible for the `acknowledge`. The banded circle at the bottom of the diagram indicates the end of a single process flow. Note that Activity Diagrams and Class Diagrams generally lack the close correspondence between activities and operations exhibited in this particular example.

The UML's acceptance for embedded system and realtime domains grows [Bianco et al. 2002], making it an attractive candidate for system modeling tasks. User interaction, behavior, architecture and communication patterns can all be expressed, depending on the diagram types chosen to describe the system. Many tools exist to support object oriented design using the UML, and new tools and methods are being developed for the real time and embedded system domains. These tools allow the user to animate the design, cross check different charts for internal consistency, and automatically generate code.

On the other hand, there is no formal definition of the UML, making it difficult to assign meaning to a specification. Efforts to disambiguate UML specifications fall along two axes. First, the Object Constraint Language (OCL) [Warmer and Kleppe 2000], now included in The UML standard, is a first-order logical language that uses entities from UML diagrams as variable declarations. OCL annotations can be added to UML specifications for clarity. Second, a large body of research focuses on fully formalizing the language. These approaches generally fall into one of three categories: those using a set-theoretic semantics [Richters and Gogolla 1998; 2001]; those that translate the specification into formal languages such as Z [France et al. 1997], Object-Z [Kim and Carrington 1999], Larch [Brickford and Guaspari 1998], or other formalizations such as Abstract State Machines [Böger et al. 2000] or Petri Nets [Baresi 2002]; and those that bootstrap a semantics via metamodeling [Clark et al. 2001]. Hussmann [2002] considers these approaches in more detail and presents an attempt to capitalize on their strengths by using object algebras to unify the theories.

3.1.4 Specification and Description Language. The Specification and Description Language (SDL) is an ITU standard [International Telecommunication Union 1999a] promoted by the SDL Forum Society [Ellsberger et al. 1997]. A diagrammatic syntax and a textual syntax are both maintained as part of the standard. SDL may be used to specify systems at four levels of hierarchy: system, block, process, and procedure. System specifications define the boundaries between the specified system and its environment, and contain at least one block. Block specifications group sets of processes and show the communication that takes place among them. Similar to Statecharts in notation and usage, process specifications also extend simple state machines with communication notation. Finally, procedure specifications break processes into task units that can be reused easily.

SDL specifications often consist of diagrams representing each of four levels of hierarchy. In Figure 11 we specify a data transfer handshake consisting of two blocks and showing only the process level. Each process begins execution in the state uppermost in the figure. Rectangles with concave points represent messages received, and rectangles with convex points denote messages sent. States with dashes denote a loop back to the start state. The initiator process sends a `data` message followed by the `valid` message. It waits for the `ack` from the target and loops. The target, on the other hand, awaits both the `valid` and the `data` messages before it sends the `ack` message and loops to the start state.

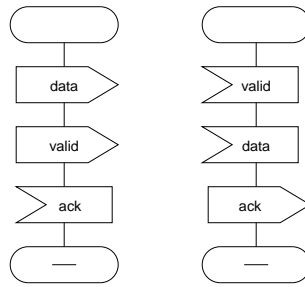


Fig. 11. Data transfer example specified in Specification and Description Language.

Since it has a formal definition [International Telecommunication Union 2000a; 2000b], SDL is often used for mission-critical systems, such as complex real-time [Bozga et al. 2001; Bozga et al. 2000], aerospace, or distributed systems. SDL has a large software development user base in Europe, and a wide variety of support tools are available, including automatic code generators for C, C++, and Java and model checkers which input the implementation model as an SDL program [Levin and Yenigün 2001; Sharygina et al. 2001]. Its clear interfaces and object oriented nature aid design reuse. Unfortunately, SDL is verbose: even small specifications can take pages to document. Since hardware protocol specifications usually include descriptions of several communicating state machines, SDL specifications are likely to grow cumbersome.

3.2 Hardware Engineering

Timing diagrams are often used in hardware design both to express the intended behavior of a set of signals and to visualize their actual behavior. In this section, we first discuss efforts to formalize timing diagrams for use in hardware specification. Then we present Hierarchical Annotated Action Diagrams, a language that combines timing diagrams with a notation for composing their behaviors. Finally, we complete our presentation of languages originating with the hardware design profession by considering the Heterogeneous Hardware Logic, which incorporates waveform diagrams, circuit schematics, state machine notations, and several logics into a single, coherent specification system.

3.2.1 Timing Diagrams. Because of their widespread use in industrial design practice, timing diagrams have been investigated as a specification language since at least the early 1990s [Walkup and Borriello 1994]. Researchers pursue at least two major approaches, one aimed at hardware/software cosynthesis and timing verification, and the other intended to alleviate the functional verification problem.

The first approach translates the timing diagrams to temporal logic formulas, which then determine the exact semantics of the diagrams [Lüth et al. 1998]. Timing diagrams express constraints on the operating environment, causal relations on events, and constraints on the design outputs. Tools can then synthesize the timing diagram specification into either a VHDL model or a set of assertions for formal verification [Amon et al. 1997].

In a second approach, Amla et al. [1999; Amla et al. [2001] define Regular Timing Diagrams (RTDs), a class of timing diagrams for which both a precise semantics and a decompositional model-checking algorithm exist. The language includes mechanisms for iterating and overlapping the execution of the specification diagrams. The user may also

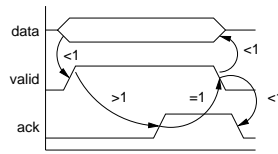


Fig. 12. Data transfer example specified in Regular Timing Diagrams.

express signal transitions, sequential and concurrent dependencies between events, and timing constraints. The model checking algorithm translates the diagram into a universal finite automaton, and then checks that the language of the implementation model is included in the language of the RTDs.

Given the similarity in syntax between the two timing diagram languages, we present only Regular Timing Diagrams in Figure 12. Arrows leading from one waveform to another indicate sequential dependencies; these arrows are labeled with the interval notations attached to each state’s timing constraints. For instance, < 1 is shorthand for $[0, 1]$, which indicates that the dependent event must occur between zero and one time units after the independent event. In our example, the `valid` bit must be set less than one time step after the `data` bit is set. One or more time units later, the `ack` may be asserted. The assertion of `ack` triggers the deassertion of `valid` and the invalidation of the value on `data`, both of which happen within a single time step. Finally, the deassertion of `valid` forces the deassertion of `ack` within one unit of time.

Timing diagram notations have several characteristics desirable for protocol compliance verification. First, their formal semantics makes specifications precise. Second, their usage as specification languages [Amla et al. 2000; 2001] as well as that of their successor language, Modular Timing Diagrams [Amla et al. 2002] comes directly from the verification tool community, and so many tools and automatic verification techniques can be leveraged. Finally, since timing diagrams are commonly used as design teaching tools, they are already fully integrated into industrial design and verification practice.

3.2.2 Hierarchical Annotated Action Diagrams. Though not included in our list of evaluation criteria, another drawback to using timing diagrams for hierarchical specification and verification is the absence of a way to determine whether or not the specification is realizable via a series of independently developed modules [Khordoc and Cerny 1998; Khordoc 1996]. Action Diagrams and Hierarchical Annotated Action Diagrams (HAAD) extended timing diagrams to include methods for determining causality and realizability. A HAAD specification consists of two types of diagrams: Leaf Diagrams and Hierarchical Diagrams. As their names imply, Leaf Diagrams describe the behavior of ports of the design, or leaf behaviors. They are similar to the timing diagrams discussed earlier. Hierarchical Diagrams compose leaf behaviors in sequence, in parallel, in a looping construct, in a choice construct, or in an exception handling construct.

Figure 13 shows how the data transfer example might be specified in HAAD. Figure 13a shows a timing diagram describing the behavior of the four ports. `clk` represents the system clock, which is an input to the hypothetical module. `data` and `valid` are both outputs, and `ack` is an input connected to a complementary module. As in other timing diagrams, minimum and maximum reaction constraints may be specified. HAAD specifications may also be annotated with parameters, variable names, and Boolean functions, which are not

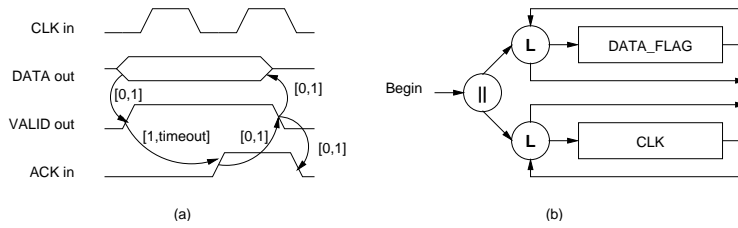


Fig. 13. Data transfer example specified in Hierarchical Annotated Action Diagrams: (a) Leaf Diagram, (b) Hierarchical Diagram.

shown in our example. Figure 13b shows a Hierarchical Diagram that composes the Leaf Diagram on the left with two hypothetical Leaf Diagrams in parallel ($| |$ operator) to complete the example specification. Each of the component modules may operate zero or more times, indicated by the L (loop) operator.

HAAD is designed with interface specification, verification and synthesis specifically in mind. It allows the user to specify behavior and timing constraints separately. The HAAD specification has a formal semantics and can be translated into a textual format suitable for simulation and formal verification purposes. The textual format can also be translated into fully synthesizable VHDL code [Khordoc et al. 1991]. However, HAAD has an expressive drawback that makes its syntax somewhat unnatural for specifying interfaces, although our example does not expose this. Each Leaf Diagram contains an implicit begin-end pair corresponding to the left and right boundaries of the diagram. Because begin and end are treated as actions, they can alter the solution space of the annotations and the timing constraints of the diagram. Eliminating ill effects from this characteristic commonly requires that diagrams begin or end at times other than the natural beginning or ending of transactions. Consequently, describing pipelined protocols can be particularly difficult in Hierarchical Annotated Action Diagrams.

3.2.3 Heterogeneous Hardware Logic. The Heterogeneous Hardware Logic (HHL) is composed of six hardware description techniques, including circuit diagrams, algorithmic state machines (ASMs), timing diagrams, computational tree logic (CTL), second-order logic, and regular expressions over timing diagrams and linear temporal logic (LTL) [Fisler 1996]. HHL circuit diagrams represent a collection of components built from `and` gates, unit delay elements, and inverters. Algorithmic state machines combine Mealy and Moore machines into a single notation resembling flow charts. Rectangles represent states, bars with pointed, convex ends denote decisions, and ovals denote Mealy outputs. As one of the diagram’s first formalizations, timing diagrams in HHL closely resemble those presented in Section 3.2.1. Since timing diagrams take the view that there is only one possible computational future, CTL is included to represent the view that many future possibilities exist. Likewise, second-order logic adds the flexibility of quantification over functions from discrete time values to the logical values 1 and 0 for modeling signals. The sixth and final component of the logic, the Calculus of Temporal Representations allows automata to be used as operators within temporal logics, connecting and unifying various aspects of HHL.

The verification engineer constructs proofs using the visual components of this logic directly, whereas most other visual languages require translating the diagrammatic representations into a sentential form for manipulation. Notations in the HHL each have infer-

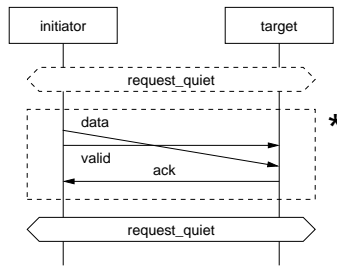


Fig. 14. Data transfer example specified in Live Sequence Charts.

ence rules for doing logical manipulation within a single notation, along with inference rules for formalizing the translation between notations. This allows the user to show that a particular circuit schematic implements the algorithm shown in an ASM, for example. Some of the HHL sublanguages have support for algorithmic verification, as well [Fisler 1999]. Including the logical representation of the physical implementation lets proofs of high-level properties extend to the lowest implementation level.

The Heterogeneous Hardware Logic’s flexibility is its greatest strength. It also fully formalizes several diagrammatic notations commonly used in engineering practice. However, both these advantages come at the price of ease of use and notational intuitiveness. Most engineers require training in specialized logics before they can fully utilize the language. Though the HHL includes some automated tool support, we fear that verification using the logic would require prohibitively large amounts of time and expertise. The HHL is better suited for use with formal verification tools that require intensive human interaction.

3.3 Systems Engineering

The only language comprising our discussion of languages originating from the system engineering discipline, Harel and Damm’s Live Sequence Charts (LSCs) [Damm and Harel 2001] extend Message Sequence Charts (see Section 3.1.1) in several ways. First, LSCs contain a mechanism for testing conditions. In fact, when used in conjunction with the second language extension, the ability to differentiate between optional and required behaviors, conditions can model if-then-else constructs from traditional programming languages. Finally, multiplicities (represented as asterisks) specify that the associated behavior may occur multiple times. When combined with conditions and optional behaviors, they can represent loops. These features allow users to describe several alternative scenarios in a single chart, unlike MSCs, which can only express one series of events per chart.

Figure 14 demonstrates the data transfer example specified in Live Sequence Charts. Note the MSC-like features. We require that the interface be inactive prior to execution of the chart, as shown by the optional (dashed) condition bar across the process life lines. If the system is actively transmitting, the chart simply exits. Likewise, the chart leaves the interface idle, as specified by the required postcondition. The required condition indicates that leaving the interface in any state but `request_quiet` is a design flaw. Finally, applying the multiplicity operator to a subchart (dashed box) containing the series of repeatable behaviors indicates that zero or more transfers may occur.

The intuitive, graphic nature of the language makes it easy to learn, and specifications are easy to read and to understand. Current work seeks to develop a formal semantics

for Live Sequence Charts [Damm and Harel 2001; Klose and Wittke 2001]. Tool support for LSCs emerges rapidly as the language is integrated into the Statemate verification system [Damm and Klose 2001; Bohn et al. 2002]. Also, tools allowing the user to create and LSC specification of a system by describing interaction scenarios, called play-engines are in development [Harel 2001; Harel et al. 2002; Harel and Marelly 2002]. Finally, Bunker and Gopalakrishnan [2002] show how LSCs might be used in an automatic protocol compliance verification system. These qualities make the Live Sequence Charts language a promising candidate for protocol compliance verification.

4. CONCLUSIONS

Choosing an appropriate specification language for a particular verification problem is an important task, as the specification language is the main tool of the verification engineer. Familiarity with many such languages, their strengths, and their weaknesses makes the choice more informed, if not more tractable. In the survey presented here, we consider a variety of specification languages as candidates for protocol compliance verification and evaluate their potential for specifying hardware communication protocols in a formal verification context. Our evaluation, based on four criteria, yields the following general observations about the four required criteria listed in Section 1.

- Precise semantics.** Languages of all types—visual or sentential, intended for hardware or software engineering—are equally likely to be formalized. That is, few languages in any category have a formal semantics, and existing semantics often leave questions unanswered, as in the case of Message Sequence Charts (see Section 3.1.1). We believe this to be a result of the difficulty in writing a formal semantics, and not a characteristic inherent in any particular type of language.
- Short learning curve.** Hardware design engineers typically shy away from logical languages such as temporal logics. They also show some resistance to learning new programming paradigms like object-oriented techniques. However, if the syntax resembles that of a language with which the practitioner is familiar, she can often learn a new modeling language (of any type) with ease. Furthermore, becoming proficient in using visual languages seems to require less background in similar notations.
- Connections with automatic verification techniques.** Textual languages are particularly amenable to linkage with automatic verification tools and techniques. This may be due to the historical development of the verification discipline. Also, it appears that the presence or absence of specific language constructs greatly impacts visual languages' potential for integration with automatic tools.
- Integration with existing design practice.** Maturity of the language or paradigm seems to determine whether or not practitioners use that language in daily tasks. This is not surprising, given the time required to develop tool support and to migrate new techniques into the workplace and classroom.

The growing complexity of the verification problem outpaces the growing power of our tools, and current approaches represent but partial solutions. As more designers include others' intellectual property (IP) in their own designs, the availability of decisive standard compliance verification techniques becomes critical. We feel that 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.

REFERENCES

- ABARBANEL, Y., BEER, I., GLUHOVSKY, L., KEIDAR, S., AND WOLFSTHAL, Y. 2000. FoCs—Automatic Generation of Simulation Checkers from Formal Specifications. In *Computer Aided Verification*, E. A. Emerson and A. P. Sistla, Eds. Vol. 1855. Springer-Verlag, Lecture Notes in Computer Science, 538–542.
- Accellera Organization, Inc. 2002. *SystemVerilog 3.0: Accellera's Extensions to VerilogTM*. Accellera Organization, Inc.
- Accellera Organization, Inc. 2003. *Property Specification Language Reference Manual*. Accellera Organization, Inc.
- ALLARA, A., BOMBANA, M., CAVALLORO, P., NEVEL, W., PUTZKE, W., AND RADETZKI, M. 1998. ATM Cell Modelling Using Objective VHDL. In *Proceedings of Asia and South Pacific Design Automation Conference*. 261–264.
- ALUR, R., ETESSAMI, K., AND YANNAKAKIS, M. 2000. Inference of Message Sequence Charts. In *22nd International Conference on Software Engineering*. 304–313.
- ALUR, R. AND YANNAKAKIS, M. 1999. Model Checking of Message Sequence Charts. In *Proceedings of the Tenth International Conference on Concurrency Theory*. Lecture Notes in Computer Science, vol. 1664. Springer-Verlag, 114–129.
- AMLA, N., EMERSON, E. A., KURSHAN, R. P., AND NAMJOSHI, K. 2001. RTDT: A Front-End for Efficient Model Checking of Synchronous Timing Diagrams. In *Proceedings of Computer-Aided Verification*. Lecture Notes in Computer Science, vol. 2102. Springer-Verlag, 387–390.
- AMLA, N., EMERSON, E. A., KURSHAN, R. P., AND NAMJOSHI, K. S. 2000. Model Checking Synchronous Timing Diagrams. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, W. A. Hunt Jr. and S. D. Johnson, Eds. Lecture Notes in Computer Science, vol. 1954. Springer Verlag, 283–298.
- AMLA, N., EMERSON, E. A., NAMJOSHI, K., AND TREFLER, R. 2001. Assume-Guarantee Based Compositional Reasoning for Synchronous Timing Diagrams. In *Tools and Algorithms for the Construction and Analysis of Systems*, T. Margaria and W. Yi, Eds. Lecture Notes in Computer Science, vol. 2031. Springer-Verlag, 465–479.
- AMLA, N., EMERSON, E. A., AND NAMJOSHI, K. S. 1999. Efficient Decompositional Model Checking for Regular Timing Diagrams. In *Correct Hardware Design and Verification Methods: 10th IFIP WG10.5 Advanced Research Working Conference*. Lecture Notes in Computer Science, vol. 1703. Springer-Verlag, 465–479.
- AMLA, N., EMERSON, E. A., NAMJOSHI, K. S., AND TREFLER, R. J. 2002. Visual Specifications for Modular Reasoning about Asynchronous Systems. In *Formal Techniques for Networked and Distributed Systems*. Lecture Notes in Computer Science, vol. 2529. Springer-Verlag, 226–242.
- AMON, T., BORRIELLO, G., HU, T., AND LIU, J. 1997. Symbolic Timing Verification of Timing Diagrams using Presburger Formulas. In *34th Design Automation Conference Proceedings*. Association for Computing Machinery, 226–231.
- ANDRÉ, C. 1996. Representation and Analysis of Reactive Behaviors: A Synchronous Approach. Tech. rep., Laboratoire Informatique Signaux, Systèmes.
- ARMONI, R., FIX, L., FLAISHER, A., GERTH, R., GINSBURG, B., KANZA, T., LANDVER, A., MADORHAIM, S., SINGERMAN, E., TIEMEYER, A., VARDI, M. Y., AND ZBAR, Y. 2002. The ForSpec Temporal Logic: A New Temporal Property-Specification Language. In *Tools and Algorithms for Construction and Analysis of Systems*. Lecture Notes in Computer Science, vol. 2280. Springer-Verlag, 296–211.
- ASHENDEN, P. J., WILSEY, P. A., AND MARTIN, D. E. 1997. SUAVE: Painless Extensions for an Object-Oriented VHDL. In *VHDL International Users Forum Conference Proceedings*.
- BARESI, L. 2002. Some Preliminary Hints on Formalizing UML with Object Petri Nets. In *Proceedings of the Sixth Biennial World Conference on Integrated Design and Process Technology*, H. Ehrig, B. J. Krämer, and A. Ertas, Eds. Society of Design and Process Science, 17.
- BEER, I., BEN-DAVID, S., EISNER, C., FISMAN, D., GRINGAUZE, A., AND RODEH, Y. 2001. The Temporal Logic Sugar. In *Computer Aided Verification*, G. Berry, H. Comon, and A. Finkel, Eds. Lecture Notes in Computer Science, vol. 2102. Springer-Verlag, 363–367.
- BEER, I., BEN-DAVID, S., EISNER, C., GEIST, D., GLUHOVSKY, L., HEYMAN, T., LANDVER, A., PAANAH, P., RODEH, Y., RONIN, G., AND WOLFSTHAL, Y. 1997. RuleBase: Model Checking at IBM. In *Computer Aided Verification*, O. Grumberg, Ed. Lecture Notes in Computer Science, vol. 1254. Springer-Verlag.

- BEER, I., BEN-DAVID, S., EISNER, C., AND LANDVER, A. 1996. RuleBase: an Industry-Oriented Formal Verification Tool. In *Proceeding of the 33rd Annual Conference on Design Automation*. ACM Press, 655–660.
- Bell Labs Design Automation and Lucent Technologies 1998. *FormalCheck User's Guide*, v2.1 ed. Bell Labs Design Automation and Lucent Technologies.
- BERGERON, J. AND SIMMONS, D. 2000. Exploiting the Power of Vera: Creating Useful Class Libraries. In *Proceedings of Synopsys Users Group*.
- BERRY, G. 1999. The Esterel v5 Language Primer. Tech. rep., Centre de Mathématiques Appliquées, Ecole des Mines and INRIA.
- BERRY, G. 2000. The Foundations of Esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, Eds. MIT Press.
- BERRY, G. AND KISHINEVSKY, M. 2000. Hardware Esterel Language Extension Proposal. Tech. rep., Esterel Technologies.
- BHASKER, J. 2002. *A SystemC Primer*. Star Galaxy Publishing.
- BIANCO, V. D., LAVAZZA, L., AND MAURI, M. 2002. A Formalization of UML Statecharts for Real-time Software Modeling. In *Proceedings of the Sixth Biennial World Conference on Integrated Design and Process Technology*, H. Ehrig, B. J. Krämer, and A. Ertas, Eds. Society of Design and Process Science, 16.
- BJESSE, P., CLAESSEN, K., SHEERAN, M., AND SINGH, S. 1998. Lava: Hardware Design in Haskell. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 174–184.
- BLANC, L. AND DISSOUBRAY, S. 2000. Esterel Methodology for Complex System Design. In *International Summer School on Advance Microelectronics*.
- BÖGER, E., CAVARRA, A., AND RICCOBENE, E. 2000. An ASM Semantics for UML Activity Diagrams. In *Algebraic Methodology and Software Technology*, T. Rus, Ed. Lecture Notes in Computer Science, vol. 1816. Springer-Verlag, 298–308.
- BOHN, J., DAMM, W., WITTKÉ, H., KLOSE, J., AND MOIK, A. 2002. Modeling and Validating Train System Applications Using StateMATE and Live Sequence Charts. In *Proceedings of the Sixth Biennial World Conference on Integrated Design and Process Technology*, H. Ehrig, B. J. Krämer, and A. Ertas, Eds. Society for Design and Process Science, 34.
- BOOCH, G., RUMBAUGH, J., AND JACOBSON, I. 1999. *The Unified Modeling Language User Guide*. Object Technology Series. Addison Wesley.
- BOZGA, B., GRAF, S., MUNIER, L., OBER, I., ROUX, J.-L., AND VINCENT, D. 2001. Timed Extensions for SDL. In *Proceedings of the Tenth SDL Forum*. Lecture Notes in Computer Science, vol. 2078. Springer-Verlag, 223–240.
- BOZGA, M., GRAF, S., KERBRAT, A., VINCENT, D., MOUNIER, L., AND OBER, I. 2000. SDL for Real-Time: What Is Missing? In *Proceedings of SAM: 2nd Workshop on SDL and MSC*. 108–122.
- BRAT, G., HAVELUND, K., PARK, S., AND VISSER, W. 2000. Java PathFinder—A Second Generation of a Java Model Checker. In *Workshop on Advances in Verification*.
- BRICKFORD, M. AND GUASPARI, D. 1998. Lightweight Analysis of UML. Tech. rep., Odyssey Research Associates.
- BRUNELLI, M., BATTÚ, L., CASTELNUOVO, A., AND SFORZA, F. 2001. Functional Verification of a HW Block Using VERA. In *Synopsys Users Group Proceedings*.
- BUNKER, A. AND GOPALAKRISHNAN, G. 2001. Using Live Sequence Charts for Hardware Protocol Specification and Compliance Verification. In *IEEE International High Level Design Validation and Test Workshop*. IEEE Computer Society Press, 95–100.
- BUNKER, A. AND GOPALAKRISHNAN, G. 2002. Verifying a VCI Bus Interface Model Using an LSC-based Specification. In *Proceedings of the Sixth Biennial World Conference on Integrated Design and Process Technology*, H. Ehrig, B. J. Krämer, and A. Ertas, Eds. Society of Design and Process Science, 48.
- CABANIS, D., MEDHAT, S., AND WEAVERS, N. 1996. Object-Oriented Extensions of VHDL: The Classification Orientation. In *VDHL User Forum. SIG-VHDL*, 265–274.
- CLAESSEN, K. AND SHEERAN, M. 2000. A Tutorial on Lava: A Hardware Description and Verification Language. Tech. rep., School of Computer Science and Engineering, Chalmers University of Technology and Göteborg University.

- CLARK, T., EVANS, A., KENT, S., AND SAMMUT, P. 2001. The MMF Approach to Engineering Object-Oriented Design Languages. In *Proceedings of the Workshop on Language Descriptions, Tools and Applications*.
- DAMM, W. AND HAREL, D. 2001. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 45–80.
- DAMM, W. AND KLOSE, J. 2001. Verification of a Radio-Based Signaling System Using the StateMate Verification Environment. *Formal Methods in System Design* 19, 121–141.
- DÖMER, R. AND GAJSKI, D. D. 1998. Comparison of the Scenic Design Environment and the SpecC System. Tech. rep., Department of Information and Computer Science, University of California, Irvine.
- DÖMER, R. AND GAJSKI, D. D. 2000. Reuse and Protection of Intellectual Property in the SpecC System. In *Proceedings of Asia South Pacific Design Automation Conference*. 49–54.
- DÖMER, R., GERSTLAUER, A., AND GAJSKI, D. 2002. *SpecC Language Reference Manual: Version 2.0*. SpecC Technology Open Consortium.
- DÖMER, R., ZHU, J., AND GAJSKI, D. D. 1998. The SpecC Language Reference Manual. Tech. rep., Department of Information and Computer Science, University of California, Irvine.
- DRECHSLER, R. AND GROßE, D. 2002. Reachability Analysis for Formal Verification of SystemC. In *Proceedings of the Euromicro Symposium on Digital System Design*. IEEE Computer Society, 337–340.
- ELLSBERGER, J., HOGREFE, D., AND SARMA, A. 1997. *SDL: Formal Object-oriented Language for Communicating Systems*. Prentice Hall.
- FISLER, K. 1996. A Unified Approach to Hardware Verification Through a Heterogeneous Logic of Design Diagrams. Ph.D. thesis, Indiana University.
- FISLER, K. 1999. Timing Diagrams: Formalization and Algorithmic Verification. *Journal of Logic, Language, and Information* 8, 3.
- FORMAL METHODS GROUP. 2000. *Guide to Sugar Formal Specification Language*. IBM Haifa Research Laboratory.
- FORMAL METHODS GROUP. 2001. *EDL*. IBM Haifa Research Laboratory.
- FOWLER, M. 1999. *UML Distilled*. Object Technology Series. Addison-Wesley.
- FRANCE, R., BRUEL, J.-M., LARRONDO-PETRIE, M. M., AND SHROFF, M. 1997. Exploring the Semantics of UML Type Structures with Z. In *Proceedings of the Second IFIP Formal Methods in Object-Oriented and Distributed Systems (FMOODS)*. 247–260.
- GAJSKI, D. D., ZHU, J., DÖMER, R., GERSTLAUER, A., AND ZHAO, S. 1999. The SpecC Methodology. Tech. Rep. ICS-99-56, Department of Information and Computer Science, University of California, Irvine.
- GAJSKI, D. D., ZHU, J., DÖMER, R., GERSTLAUER, A., AND ZHAO, S. 2000. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers.
- GOERING, R. 2003. EDA divided on SystemVerilog. In *EE Times*.
- GORDON, M. J. C. 2002. Using HOL to Study Sugar 2.0 Semantics. In *Track B Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics*. Number CP-2002-211736. National Aeronautics and Space Administration, 87–100.
- GUNTER, E. L., MUSCHOLL, A., AND PELED, D. A. 2001. Compositional Message Sequence Charts. In *Tools and Algorithms for the Construction and Analysis of Systems*, T. Margaria and W. Yi, Eds. Lecture Notes in Computer Science, vol. 2031. Springer-Verlag, 496–511.
- HAREL, D. 1987. StateCharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8, 231–274.
- HAREL, D. 2001. From Play-In Scenarios to Code: An Achievable Dream. *IEEE Computer* 34, 1 (January), 53–60.
- HAREL, D., KUGLER, H., MARELLY, R., AND PNUELI, A. 2002. Smart Play-Out of Behavioral Requirements. In *Formal Methods in Computer-aided Design*. Lecture Notes in Computer Science, vol. 2517. Springer-Verlag, 378–398.
- HAREL, D. AND MARELLY, R. 2002. Playing with Time: On the Specification and Execution of Time-Enriched LSCs. In *Proceedings of the 10th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE Computer Society, 193–202.
- HAREL, D., PNUELI, A., SCHMIDT, J. P., AND SHERMAN, R. 1987. On the Formal Semantics of Statecharts. In *IEEE Symposium On Logic In Computer Science*. IEEE Computer Society Press, 54–64.

- HELAIHEL, R. AND OLUKOTUN, K. 1997. Java as a Specification Language for Hardware-Software Systems. In *Proceedings of the 1997 International Conference on Computer-Aided Design*. 690–697.
- HELBIG, J. AND KELB, P. 1994. An OBDD-Representation of Statecharts. In *The European Conference on Design Automation*. IEEE Computer Society Press, 142–149.
- HENRIKSEN, J. G., MUKUN, M., KUMAR, K. N., AND THIAGARAJAN, P. 2000. On Message Sequence Graphs and Finitely Generated Regular MSC Languages. In *International Symposium on Mathematical Foundations of Computer Science*. Lecture Notes in Computer Science, vol. 1853. Springer-Verlag, 675–686.
- HOLLANDER, Y., NOY, A., AND MORLEY, M. 2001. The e Language: A Fresh Separation of Concerns. In *Proceedings Technology of Object-Oriented Languages and Systems (TOOLS 38 '01)*. IEEE Computer Society, 41–50.
- HUSSMANN, H. 2002. Loose Semantics for UML/OCL. In *Proceedings of the Sixth Biennial World Conference on Integrated Design and Process Technology*, H. Ehrig, B. J. Krämer, and A. Ertas, Eds. Society of Design and Process Science, 15.
- International Telecommunication Union 1998. *ITU-T Recommendation Z.120: Message Sequence Chart (MSC) – Annex B: Formal Semantics of Message Sequence Charts*. International Telecommunication Union.
- International Telecommunication Union 1999a. *ITU-T Recommendation Z.100: Specification and Description Language (SDL)*. International Telecommunication Union.
- International Telecommunication Union 1999b. *ITU-T Recommendation Z.120: Message Sequence Chart (MSC)*. International Telecommunication Union.
- International Telecommunication Union 2000a. *ITU-T Recommendation Z.100: Specification and Description Language (SDL): Annex F1: SDL Formal Definition: General Overview*. International Telecommunication Union.
- International Telecommunication Union 2000b. *ITU-T Recommendation Z.100: Specification and Description Language (SDL): Annex F3: SDL Formal Definition: Dynamic Semantics*. International Telecommunication Union.
- JAMES, P. AND DHAMANWALA, S. 2000. Vera, Vera on the Wall: Useful Lessons for First-Time Vera Users. In *Proceedings of Synopsys Users Group*.
- KAHN, G. 1974. The Semantics of Simple Language for Parallel Programming. In *IFIP Congress 1974*. 471–475.
- KHORDOC, K. 1996. Action Diagrams: A Methodology for the Specification and Verification of Real-time Systems. Ph.D. thesis, McGill University, Montreal, Canada.
- KHORDOC, K. AND CERNY, E. 1998. Semantics and Verification of Action Diagrams with Linear Timing Constraints. *ACM Transactions on Design Automation of Electronic Systems* 3, 1, 21–60.
- KHORDOC, K., DUFRESNE, M., AND CERNY, E. 1991. A Stimulus/Response System Based on Hierarchical Timing Diagrams. In *Proceedings of IEEE International Conference on Computer-Aided Design*. 358–361.
- KIM, S.-K. AND CARRINGTON, D. 1999. Formalizing the UML Class Diagram Using Object-Z. In *<<UML>>'99–The Unified Modeling Language: Beyond the Standard*, France and Rumpe, Eds. Lecture Notes in Computer Science, vol. 1723. Springer-Verlag, 83–98.
- KLOSE, J. AND WITTKE, H. 2001. An Automata Based Interpretation of Live Sequence Charts. In *Tools and Algorithms for the Construction and Analysis of Systems*, T. Margaria and W. Yi, Eds. Lecture Notes in Computer Science, vol. 2031. Springer-Verlag, 512–527.
- KRISHNAMOORTHY, S., ARORA, G., AND GURAVANNAVAR, R. 2002. Network System Verification with VERA. In *Proceedings of Synopsys Users Group*.
- KRÜGER, I., GROSU, R., SCHOLZ, P., AND BROY, M. 1999. From MSCs to Statecharts. In *Distributed and Parallel Embedded Systems*. Kluwer Academic Publishers.
- KUHN, T., OPPOLD, T., WINTERHOLER, M., ROSENSTIEL, W., EDWARDS, M., AND KASHAI, Y. 2001. A Framework for Object Oriented Hardware Specification, Verification, and Synthesis. In *Proceedings of the 38th Design Automation Conference*. Association for Computing Machinery, 413–418.
- KUHN, T. AND ROSENSTIEL, W. 1998. Java Based Modeling And Simulation Of Digital Systems On Register Transfer Level. In *Workshop on System Design Automation*.
- KUHN, T., ROSENSTIEL, W., AND KEBSCHULL, U. 1999. Description and Simulation of Hardware/Software Systems with Java. In *Proceedings of the 36th Design Automation Conference*. Association for Computing Machinery, 790–793.

- LEVIN, V. AND YENIGÜN, H. 2001. SDLcheck: a Model Checking Tool. In *Computer Aided Verification*, G. Berry, H. Comon, and A. Finkel, Eds. Lecture Notes in Computer Science, vol. 2102. Springer-Verlag, 378–381.
- LÜTH, K., NEIHAUS, J., AND PEIKENKAMP, T. 1998. HW/SW Cosynthesis using Statecharts and Symbolic Timing Diagrams. In *International Workshop on Rapid System Prototyping*. IEEE Computer Society, 212–217.
- LÜTTGEN, G., VON DER BEECK, M., AND CLEAVELAND, R. 2000. A Compositional Approach to Statecharts Semantics. In *Technical Report of ICASE, NASA Langley Research Center, Hampton, VA*. ICASE Report No. 2000-12, NASA/CR-2000-210086.
- MADHUSUDAN, P. 2001. Reasoning about Sequential and Branching Behaviors of Message Sequence Graphs. In *International Colloquium on Automata, Languages, and Programming*, F. Orejas, P. Spirakis, and J. van Leeuwen, Eds. Lecture Notes in Computer Science, vol. 2076. Springer-Verlag, 809–820.
- MADHUSUDAN, P. AND MEENAKSHI, B. 2001. Beyond Message Sequence Graphs. In *Foundations of Software Technology and Theoretical Computer Science*, R. Hariharan, M. Mukund, and V. Vinay, Eds. Lecture Notes in Computer Science, vol. 2245. Springer-Verlag, 256–267.
- MONACO, J., HOLLOWAY, D., AND RAINA, R. 1996. Functional Verification Methodology for the PowerPC Microprocessor. In *Proceedings of the 33rd Design Automation Conference*. 319–324.
- MOORBY, P., SALZ, A., FLAKE, P., DUDANI, S., AND FITZPATRICK, T. 2003. Achieving Determinism in SystemVerilog 3.1 Scheduling Semantics. In *Proceedings of the Design and Verification Conference*.
- MUELLER, W., DÖMER, R., AND GERSTLAUER, A. 2002. The Formal Execution Semantics of SpecC. In *Proceedings of the 15th International Symposium on Systems Synthesis*. 150–155.
- MUELLER, W., RUF, J., HOFFMANN, D., GERLACH, J., KROPF, T., AND ROSENSTIEHL, W. 2001. The Simulation Semantics of SystemC. In *Design Automation and Test, Europe*. 64–70.
- MUSCHOLL, A. AND PELED, D. 1999. Message Sequence Graphs and Decision Problems on Mazurkiewicz Traces. In *International Symposium on Mathematical Foundations of Computer Science*. Lecture Notes in Computer Science, vol. 1672. Springer-Verlag, 81–91.
- MUSCHOLL, A. AND PELED, D. 2000. Analyzing Message Sequence Charts. In *Proceedings of SDL and MSC Workshop*.
- MUSCHOLL, A. AND PELED, D. 2001. From Finite State Communication Protocols to High-Level Message Sequence Charts. In *International Symposium on Mathematical Foundations of Computer Science*. Lecture Notes in Computer Science, vol. 2076. Springer-Verlag, 720–731.
- MUSCHOLL, A., PELED, D., AND SU, Z. 1998. Deciding Properties for Message Sequence Charts. In *Foundations of Software Science and Computation Structures*. Lecture Notes in Computer Science, vol. 1378. Springer-Verlag, 226–242.
- OPEN SYSTEMC INITIATIVE. 1999. www.systemc.org.
- PNUELI, A. AND SHALEV, M. 1991. What is in a Step: On the Semantics of Statecharts. In *Theoretical Aspects of Computer Software*, T. Ito and A. R. Meyer, Eds. Lecture Notes in Computer Science, vol. 526. Springer-Verlag, 244–264.
- PUTZKE-RÖMING, W., RADETZKI, M., AND NEBEL, W. 1998. A Flexible Message Passing Mechanism for Objective VHDL. In *Proceedings of Design Automation and Test, Europe*. 242–249.
- RADETZKI, M., PUTZKE, W., NEBEL, W., MAGINOT, S., BERGE, J.-M., AND TAGANT, A.-M. 1997. VHDL Language Extensions to Support Abstraction and Re-use. In *Workshop on Libraries, Component Modeling and Quality Assurance*.
- RICHTERS, M. AND GOGOLLA, M. 1998. On Formalizing the UML Object Constraint Language OCL. In *Proceedings of the 17th International Conference on Conceptual Modeling*, T.-W. Ling, S. Ram, and M. L. Lee, Eds. Lecture Notes in Computer Science, vol. 1507. Springer-Verlag, 449–464.
- RICHTERS, M. AND GOGOLLA, M. 2001. OCL–Syntax, Semantics and Tools. In *Advances in Object Modelling with the OCL*, T. Clark and J. Warmer, Eds. Lecture Notes in Computer Science, vol. 2263. Springer-Verlag, 43–69.
- RUMBAUGH, J., JACOBSON, I., AND BOOCH, G. 1999. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley Longman, Inc., Reading, MA.
- SANTARINI, M. 2001. Deal Links Formal Verification to Testbench Generation. In *EETimes*. CMP Media.
- ACM Transactions on Design Automation of Electronic Systems, Vol. V, No. N, M 20Y.

- SHARYGINA, N., BROWNE, J. C., AND KURSHAN, R. P. 2001. A Formal Object-Oriented Analysis for Software Reliability: Design for Verification. In *Fundamental Approaches to Software Engineering*, H. Hussmann, Ed. Lecture Notes in Computer Science, vol. 2029. Springer-Verlag, 318–333.
- SHIMIZU, K. AND DILL, D. L. 2002. Deriving a Simulation Input Generator and a Coverage Metric From a Formal Specification. In *Proceedings of the 39th Design Automation Conference*. Association for Computing Machinery, 801–806.
- SHIMIZU, K., DILL, D. L., AND CHOU, C.-T. 2001. 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 WG10.5 Advanced Research Working Conference*. Lecture Notes in Computer Science, vol. 2114. Springer-Verlag, 340–354.
- SHIMIZU, K., DILL, D. L., AND HU, A. J. 2000. Monitor-Based Formal Specification of PCI. In *Formal Methods in Computer-Aided Design*, W. A. Hunt Jr. and S. D. Johnson, Eds. Lecture Notes in Computer Science, vol. 1954. Springer-Verlag, 335–352.
- SUTHERLAND, S. 2002. Verilog, The Next Generation: Accellera’s SystemVerilog. In *Proceedings of the HDL Conference*.
- SUTHERLAND, S. 2003. SystemVerilog 3.1: It’s What the DAVES In Your Company Asked For. In *Proceedings of the Design and Verification Conference*.
- SWAMY, S., MOLIN, A., AND COVNOT, B. 1995. OO-VHDL: Object-Oriented Extensions to VHDL. *IEEE Computer* 28, 10, 18–26.
- SWAN, S. 2001. An Introduction to System level Modeling in SystemC 2.0. Tech. rep., Open SystemC Initiative.
- Synopsys, Inc. 2001. *OpenVera 1.01: Language Reference Manual*. Synopsys, Inc.
- SYNOPTSYS, INC. 2001. OpenVera Technology Backgrounder.
- Synopsys, Inc. 2002. *OpenVera™ Assertions (OVA) and ForSpec*. Synopsys, Inc.
- TANENBAUM, A. 1998. *Computer Networks*. Prentice Hall.
- The Object Management Group 1999. *OMG Unified Modeling Language Specification*. The Object Management Group, Needham, MA.
- THOMPSON, K. AND WILLIAMSON, L. 2002. Hardware Verification with the Unified Modeling Language and Vera. In *Proceedings of Synopsys Users Group*.
- VERISITY DESIGN, I. 1999. Spec-based Verification.
- VISSER, W., HAVELUND, K., BRAT, G., AND PARK, S. 2000. Model Checking Programs. In *International Conference on Automated Software Engineering*.
- WALKUP, E. A. AND BORRIELLO, G. 1994. Interface Timing Verification with Application to Synthesis. In *Proceedings of the 31st Design Automation Conference*. Association for Computing Machinery, 106–112.
- WANG, R. AND WEN, Z. 2002. A Verification Environment for PCI-X BFM in VERA. In *Proceedings of Synopsys Users Group*.
- WARMER, J. AND KLEPPE, A. 2000. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley Longman, Inc., Boston, MA.
- YOUNG, J., MACDONALD, J., SHILMAN, M., TABBARA, P., AND NEWTON, A. 1998. Design and Specification of Embedded Systems in Java Using Successive, Formal Refinement. In *Proceedings of the 35th Design Automation Conference*. Association for Computing Machinery, 70–75.
- ZIPPELIUS, R. AND MÜLLER-GLASER, K. D. 1992. An Object-oriented Extension of VHDL. In *VHDL Forum for Computer Aided Design in Europe*. 155–163.

Received October 2002; revised May 2003, accepted month 2003