Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 176 (2007) 47–60
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

A Rewriting Semantics for ABEL with Applications to Hardware/Software Co-Design and Analysis

Michael Katelman and José Meseguer [1]

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana , Illinois 61801, U.S.A.

Abstract

We present a rewriting logic semantics in Maude of the ABEL hardware description language. Based on this semantics, and on Maude's underlying LTL model checker, we propose a scalable formal analysis framework and tool for hardware/software co-design. The analysis method is based on trace checking of finite system behaviors against LTL temporal logic formulas. The formal properties of the hardware, the embedded software, and the interactions between both can all be analyzed this way. We present two case studies illustrating our method and tool.

Keywords: co-verification, co-design, rewriting logic, program semantics, hardware description languages, Maude

1 Introduction

The restricted class of functionality required of an embedded system suggests the possibility that its hardware and software components might be designed *concurrently*, commonly called *hardware/software co-design* [18]. The typical target for a co-design is much more limited than a general purpose microprocessor: it is usually an *embedded system* such as those found in consumer electronics or in control systems of various kinds, for example, those found in automobiles or power plants. One important recent trend with many applications is towards *distributed embedded systems* having their own communication resources and being connected through wireless networks. In sheer size, the number of such systems now dwarfs by several orders of magnitude the number of standard computers.

[1] Email: `{katelman, meseguer}@uiuc.edu`

doi:10.1016/j.entcs.2007.06.007

Co-design engineering is a complex process that goes well beyond simply delegating tasks to the appropriate engineer. A critical step in the co-design process, though not necessarily the first, is to partition the system into major functional blocks, and then to match these functional blocks with specific design technologies. In particular, a decision is made as to which components will be implemented as hardware and which will be implemented as software. The choices made during this step impact directly the cost and complexity of *implementing* the system, as well as the implementation's performance. A system that is required to compute Fourier transforms, for example, could choose to do so in software on a general purpose microprocessor, in software on a digital signal processor (DSP), or directly in hardware as part of an application-specific integrated circuit (ASIC).

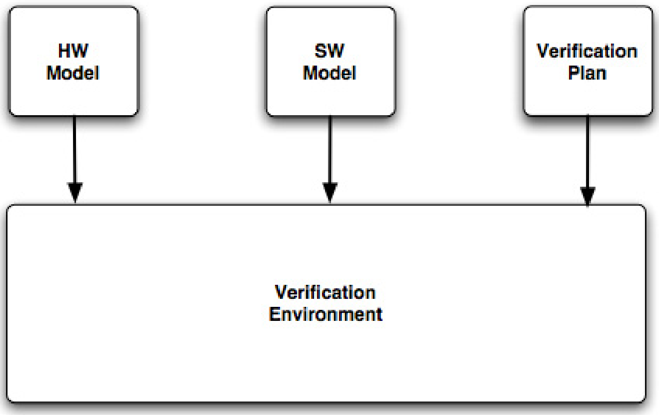
Contemporary research into embedded system design [22,13,7] has advocated a significant change to the process outlined above, whereby the major functionalities are first *modeled* at a high level before being mapped to specific technologies, such as a DSP or ASIC. Modeling is realized through a collection of formalisms covering a variety communication, concurrency, and computation schemes. In the Ptolemy II system design environment [13], for example, an engineer can describe computations using combinations of the many supported models, including data-flow network and discrete-event models. By separating functionality from implementation, tools such as Ptolemy II attempt to coordinate the system design process and produce verified designs more quickly by leveraging modularity and abstraction. In limited cases, automated tools have been able to produce implementations directly from these high-level models.

1.1 Embedded System Verification

Our view of *co-verification*, that is, the process and mechanisms through which a co-design is verified, is summarized by the generalized diagram presented as Fig. 1. Depending on the particular verification regime applied and the level at which the system's components are described, the individual boxes can be resolved more concretely. In this paper we focus on co-verification of *implementation level* designs, and in particular we focus on the hardware/software interactions at this level. We see this work as being part of a longer term and more ambitious project to formally specify and analyze embedded systems, especially the real-time and hybrid components of a system's operating environment.

Regarding Fig. 1, we need a natural hardware model to capture implementation level designs. *Register-transfer level* (RTL) code is the most pervasive and natural choice, encompassing (subsets of) well known *hardware description languages* (HDL) such as Verilog and VHDL. For our work in this paper, however, we chose another language, ABEL [24], which is owned by Xilinx Inc. and is primarily used to program FPGAs, a widely utilized technology in embedded systems. For a piece of software to be able to 'run' on a hardware component designed in ABEL, it must be at a level that the hardware can understand, and so we have focused on assembly language programming for our software model. The exact format and set of instructions allowed as part of the assembly code must be tailored to the particular

Fig. 1. Co-verification Overview



microprocessor that it is targeting. Therefore, the verification environment must be updated to support a new instruction set when analyzing a new microprocessor component.

The final component in Fig. 1 is reserved for specifying what should be verified, which in our case will be properties of RTL microprocessors and of assembly language programs written to run on those processors. There are many interesting properties about these systems that can only be described with properties that span hardware, software, and meta-level properties of the software and its data structures. The goal of this paper is to demonstrate the natural way in which rewriting logic can be used to unify the varying levels at which the properties need to be described, so that they can be reasoned about together. For this purpose we present a Maude-based co-verification environment.

The particular verification methodology that our environment supports is a very practical kind of formal analysis, namely *trace checking*, a formal method widely used for verifying both hardware [4,1,6] and software [11,19]. In trace checking, formal properties expressed as temporal logic formulas are checked for finite traces, or runs, of the system. This has the practical advantage of saving the engineering effort that would be required to build a self checking test bench to do the same thing, and helps the validation engineer focus on the properties, rather than on how they are checked.

The rest of this paper is organized as follows. Section 2 presents, by way of ABEL, a high-level overview of how the syntax and semantics of a hardware description language are specified in Maude. Section 3 explains in detail our framework for specifying and trace checking co-designs and illustrates our proposed method through two in-depth case studies. Section 4 discusses related work, and Section 5 presents some conclusions.

```
module "DFFE"
declarations
   'D, 'EN input pin ;
   'Q node istype reg ;
   'OUT output pin ;
equations
   'Q := 'D ;
   when 'EN then
      'OUT = 'Q ;
   else
      'OUT = .Z. ;
end-module
```

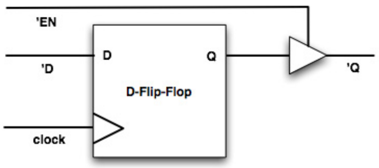


Fig. 2. Example Module and Circuit Diagram

2 Syntax and Semantics for ABEL

This section presents a high-level overview of our ABEL semantics focusing on major design decisions rather than on the particular details of ABEL. In places where concreteness can get a point across more quickly, we use ABEL syntax and explain the necessary details.

Rewriting semantics for many different types of programming languages has been described previously (e.g. see [15]). The goal of this section, in part, is to explain those areas of the RTL semantics that differ from software languages. Our second goal, which, in fact, falls out from the first, is to give enough semantic detail to support our discussion of trace checking in §3. Lastly, by way of ABEL specifically, we also show that the framework presented in §3 would permit a hardware designer to build a digital circuit in a way that is consistent with current engineering practice.

2.1 Circuit Semantics

Synchronous digital circuits need to respect two separate notions of execution order. First, during each clock cycle, the topology of the combinational (non-state carrying) network must be respected: each internal node value within the circuit is considered accurate only after its inputs become accurate, and some amount of time has elapsed to compute the value from the stabilized inputs. Second, latches should respect a *global clock* and should all be updated in parallel.

Although a synthesized circuit will mix combinational and sequential elements, it is convenient for the purposes of simulation to separate the two. Many HDL simulators do this, whereby the circuit is treated as a combinational network only. This is accomplished by changing the circuit's input/output interface to include new bits for each state element. The new inputs and outputs represent the current-state and next-state values, respectively. This transformation is used as the basis for our ABEL semantics.

Each clock cycle, every identifier (specified in the ABEL modules, that is, in the ABEL syntax) gets associated with at least one value in the transformed circuit. These associations take the form of a triple of sort `NodeValue`, and individual triples are gathered together into a set.

```
op [_,_,_] : NodeType Identifier Value -> NodeValue .
```

The first argument is a token describing what *type* of node it is in the transformed circuit: primary input (`pi`), current-state input (`cs`), primary output (`po`), next-state output (`ns`), or internal node (`in`). Fig. 2 shows an ABEL module (suitably modified for Maude parsing) that stores a value from the current clock cycle, allowing it to be used during the next clock cycle. Optionally, this output can be suppressed. The single bit wires, or *pins*, represented by the identifiers `'D` and `'EN` are inputs to the module; `'Q` is an internal node; and `'OUT` is the only output. The fact that `'Q` takes its assigned value during the next clock cycle is specified with the `istype reg` qualifier. In the equations section for the circuit, `'Q` is assigned, for the following clock cycle, the current value of `'D`. `'OUT` gets the same value as `'Q` if the enable pin `'EN` is asserted (i.e. has logical value 1), and the special value `.Z.` otherwise. The current state of the circuit could look as follows:

```
[pi, 'D, 0] [pi, 'EN, 1] [cs, 'Q, 1] [ns, 'Q, 0] [po, 'OUT, 1]
```

Indicating that during the previous clock cycle `'D` was 1, and the `'EN` pin is asserted.

Our semantics has a single rewrite rule that transitions the current state (a set of triples of sort `NodeValue`) to the next clock tick by evaluating the ABEL constructs in the current state and with user-specified inputs. The equational rewrite rules necessary to facilitate the transition from one clock cycle to the next are very much like the ones used to specify the semantics for a simple imperative programming language in [15]. Equations are evaluated based on the order in which they are written down in the module, and, moreover, ABEL only supports what are essentially the equivalent of assignment, if-then-else, and switch statements, which are very easy to specify. In some cases the semantics proved tedious due to idiosyncrasies of ABEL, but were not technically difficult to handle. The major difference, then, from the semantics for a simple imperative programming language is that we explicitly evaluate the "program" over and over again, instead of only once.

An alternative semantics for digital circuits is described in [10], where instead of a synchronizing rewrite rule, each node value is tagged with the cycle number for which the node takes on the calculated value. In that scheme, it is not necessary to have any rewrite rules, but for use with the Maude model checker, it was cleaner and more useful to specify synchronization as explained above.

3 Trace Checking the Hardware/Software Interface

This section describes how we have designed an effective co-verification environment in Maude for use on implementation level system designs. A large component of our framework consists of the ABEL semantics presented in §2, so in what follows we mainly focus on how our framework exploits the ABEL semantics for *trace checking*. We describe both how the semantics allows us to use Maude's LTL model checker for trace checking, and the related subject of how to build an effective propositional language for writing meaningful LTL formulas. The language covers hardware, software, and meta-level properties of the software and its data structures, so that all of these entities can be reasoned about together and their interactions can be explored. §3.2 presents a case study.

3.1 Co-Verification Environment

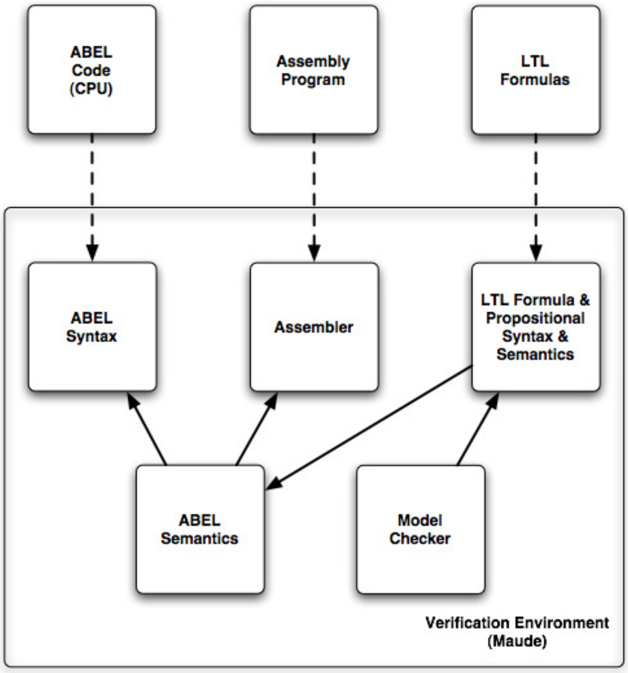
Hardware/software co-design is fundamentally about balance: deciding how the functionality of a system should be partitioned into hardware and software components. Understanding the purpose and goals of the project can help with defining metrics that guide the engineering team toward a suitable partition, one that strikes a balance among the various metrics. Common axes that are considered and influence design decisions include performance, power, size, programmability, and reduced complexity.

Clearly, the verification of a co-design project should include the interactions between hardware and software; both to ensure system wide correctness and to validate the engineering choices that led to the partitioning scheme. If, for example, the instruction throughput is never high enough to take advantage of all of the functional units, then it might be worthwhile to use this information to reevaluate the hardware partition and scale back the design. If the software is complicated and depends on many subtle interactions with the available hardware, then this inefficiency might be very hard to uncover without an implementation to test on, or without formal analysis. If, as a second example, the co-design effort tried to minimize power consumption, then it would be useful to formally validate how the power saving features perform when the software is executing on the chip. Hence, validation of the co-design decisions after implementation may be necessary to guide redesign or other efforts in the future.

Creating a Maude specification for the semantics of an HDL such as ABEL represents our first step in an effort to facilitate co-verification. The strong modeling properties of rewriting logic, and Maude in particular, allow a user to take both the hardware and software components of a design and instantly embed them into a unified mathematical framework. In addition, meta-level properties of the system (e.g. software data structures) can also be conveniently modeled and related to the rest of the design. This allows the engineer to specify properties of his/her system in the most appropriate and natural language, hence promoting a modular verification effort that abstracts away unnecessary details and helps avoid error-prone encodings.

Our current verification framework is built around a combination of simulation and LTL model checking, called *trace checking*. For trace checking to be applicable, the state of the co-designed system must evolve in exactly one way, corresponding to a concrete simulation run on fixed inputs. Our ABEL semantics supports this by design: there is only one rewrite sequence, modulo ACI axioms. When using a propositional model checker such as the one in Maude, it is necessary to develop a strong propositional language; one that makes it easy to write properties of the circuit at a high level of abstraction. For example, a circuit design used to control a traffic light should allow the current state of the light to be specified as `red`, `yellow`, or `green`, as opposed to the signal values that each color corresponds to in

Fig. 3. Co-verification Environment



the circuit. The same remains true when the model checker is being used for the more restricted case of trace checking, and so the propositional language that we have developed for trace checking co-designs has many of these features. On the hardware side, node values associated with an identifier can be checked using the propositions

```
op _asserted : Identifier -> Proposition .
op _deasserted : Identifier -> Proposition .
op _::=_ : Identifier Int -> Proposition .
op _::=_ : Identifier Identifier -> Proposition .
op p : Proposition -> Prop .
```

For identifiers associated with multiple node values in the transformed circuit, the value used is the one associated with the current-state. Single bit nodes are *asserted* if they have the value 1 in the current state, and *de-asserted* if they have the value 0. Multi-bit values can be checked for equality with an integer constant or with the value of another multi-bit node, again by checking the current state. The `::=` symbol is used for this purpose, to avoid conflicts with other sorts defined in Maude. In addition, it is convenient and easy to name constant values, such as the colors in the traffic light example. If the module declares each color as a constant value, then the identifiers associated with each constant value can be used directly by writing

```
'LIGHT ::= 'GREEN
```

As usual for LTL model checking in Maude, the semantics of state propositions is

Table 1
Simple Processor ISA:

add (addition)	sub (subtraction)	mul (multiply)
nand (bitwise nand)	lw (load)	sw (store)
beq (branch on equal)	bgt (branch on greater than)	halt

defined equationally, with equations that query the current state values to decide the truth or falsity of the proposition. For assembly programs we have added other propositions : for example we have propositions that check for when an instruction is executed, when an instruction is committed to programmer visible state, and so on. In the case study we present other abstractions that let us reason about specific programs and data structures. Fig. 3 summarizes the components of the verification environment and their dependencies, the solid arrows can be read from tail to head as 'is dependent on'.

3.2 A HW/SW Case Study

Our case study has three major components: a simple microprocessor designed in ABEL, a set of programs for the processor written in assembly, and several properties to be verified. All of these components are processed by Maude using internally defined sorts and operators; the dashed arrows in Fig. 3 show which modules are responsible for defining these external representations (i.e. RTL and assembly) that can be processed as terms in Maude.

3.2.1 Simple Microprocessor

Our microprocessor design is based on the classic five-stage, in-order pipeline example from [12] and supports the instruction set architecture (ISA) presented in Table 1. Notable features that are missing from our processor design, but would be expected in a full featured chip, include I/O subsystems, caches, and interrupt handling. All are left for future work. Programs are written under the assumption of unit latencies and are converted into machine code by the Assembler in Fig. 3. Unit assumed latency means that each instruction dynamically executed on the processor should be processed as if all preceding instructions had finished.

Each instruction flows through the five pipeline stages, eventually finishing and updating the programmer-visible state. An instruction is first fetched (IF), then decoded (ID), executed (EX), allowed to access memory (MEM), and, finally, allowed to write the register file (WB). To improve performance, a value that has already been computed, but has not yet made it back to the register file, is *forwarded* to any instruction that consumes that value. In addition, branches are statically predicted to be not-taken, and instruction fetch continues speculatively until such a prediction is proved wrong, at which point the bad instructions are invalidated. Because memory accesses happen after the execution stage, instructions fetched during the next cycle after a load and dependent on its result are stalled for one cycle in the fetch stage. All of these are interesting features that can be monitored during simulation. If there is a violation of one of these properties, the executing software can help to put the trace into a more understandable context. On the other hand, if a piece of software is producing incorrect results, but the property passes the test, then the programmer might want to look at the software itself.

3.2.2 Software and Verification Examples

The first set of formulas that we specified focus on *internal properties of the microprocessor*. The major components that we targeted were the forwarding, branching, and stall logic. We checked that each instruction gets the appropriate operand values in the execution stage, that no speculative instructions are allowed to update the state when a branch is taken, and that only certain conditions cause stalls, and these stalls are of a fixed number of cycles. Table 2 lists, and gives a short description of, each of the processor-specific formulas that we verified. The main property we use to verify the forwarding logic is `hw-02`, which is specified by the LTL formula

```
op hw-02 : -> Formula .
eq hw-02 =
[] p((('MEM-WB-valid : 0 asserted)
   /\ ('IF-ID-reg-A-used : 3 asserted))
   ->
   (('alu-input-A : 2 ::=
     register-value('IF-ID-reg-A : 3, 0))))
...same for second register source operand.
```

The propositions above show an addition to the language not described previously, and which is used to reason about the pipeline stages. The proposition

```
'alu-input-A : 2 ::= register-value('IF-ID-reg-A : 3, 0)
```

evaluates to true when the internal multi-bit node named `alu-input-A`, which feeds into the `EX` stage ALU, carried a value 2 cycles ago that equals the current state of the register specified in the 'A' position of the instruction word fetched 3 cycles ago. So the LTL formula basically states that for any instruction which completes, the register operands it got in the `EX` stage of the pipeline was equal to the register file contents when it committed. Of course, a more complete formula would verify that the internal signals referenced actually have their intended meaning. However, as given, the formula would be able to catch most of the common errors with respect to operand forwarding.

Turning to the software component, we started by writing a *bubble sort program* in the assembly language of our microprocessor. In addition to the properties from Table 2, we added four new ones specific to the software component and its functionality at the meta-level. Meta-level reasoning is facilitated by defining a sort function for integer arrays directly in Maude, together with a map that extracts an array from a software pointer value and length. This makes it possible to compare the mathematical definition of a sorted array with the assembly code sorting program and its data structures. Getting the pointer and length values for an array

Table 2
LTL Formulas for Processor Verification

Formula Name	Description
`hw-01`	Instructions only stall in IF.
`hw-02`	ALU operands reflect register file at commit.
`hw-03`	No stalls longer than 3 cycles.
`hw-04`	No state change after halt.
`hw-05`	On branch, next committed instruction is target.
`hw-06`	`r0` is never overwritten.
`hw-07`	Taken branch flushes IF, ID, and EX.
`hw-08`	Stalls only occur on branch taken and load-use.

would generally depend on the application binary interface (ABI) for argument and stack conventions, and also on the array implementation assumed. It would also rely on the specific register allocation map used. The ABI would be defined before any circuit design work is started, and the register map would be known by the engineer who wrote the assembly code or the compiler that generated it.

Correctness of the the bubble sort program is checked for by an LTL formula that ensures that whenever there is an inversion in the array, the swap code in the inner loop is executed at some point in the future. The proposition for checking for an inversion in an ordered integer array is given by

```
op inversion : IntArray -> Prop .
eq inversion(IX:Int) = false .
eq inversion(IX:Int, IY:Int, ILX:IntList) =
   IX > IY or inversion(IY, ILX) .
```

In addition to verifying correctness using the inversion check, the *running time* of the function can also be monitored. Run time properties can be stated in terms of the number of cycles, relative to the size of the input array, until the halt instruction is executed. In the case of bubble sort, this should be some constant multiple of the square of the array size. For a more sophisticated program it would probably be convenient to track the running time by inspecting the data structures using an appropriate meta-level abstraction, like the one for integer arrays. In the case of sorting, the running time could be specified in terms of the number of inversions in the input array, or in the case of matrices some measure of sparsity might be useful.

We also checked that *the program is not self-modifying* and *doesn't read or write outside the array boundaries*. Table 3 lists all of the properties specific to the sorting program, along with a short description for each one. Checking all of the properties from Tables 2 and 3 for a run of the sorting program with an array of size 13 took about 4 minutes and 5000 simulation cycles on a 2.5GHz/2GB PowerPC machine.

Table 3
Formulas for Bubble Sort

Formula Name	Description
`bs-01`	Array inversion leads to swap.
`bs-02`	Run time is $< 10n^2$ cycles.
`bs-03`	No self-modifying code.
`bs-04`	No reads or writes outside the array bounds.

The second program we studied was taken from graphics programming. It is part of a *shader routine* that calculates the normal to a surface. Short assembly code fragments that need to obtain a high percentage of the chip's theoretical peak performance are routinely written for low level graphics operations. Using this reasoning we set about writing the corresponding code for our microprocessor with the intent that the instructions should be scheduled for highest performance. Therefore, we specified that the code should not have any dynamic *pipeline stalls* and that it should *complete in less than 100 cycles*. All of the properties we checked are given in Table 4.

Table 4
Formulas for Phong Shader

Formula Name	Description
`ps-01`	No pipeline stalls in the main code body.
`ps-02`	Computes the correct value.
`ps-03`	No self-modifying code.
`ps-04`	No reads or writes outside of array bounds.
`ps-05`	Finishes in less than 100 cycles.

In trying to schedule the instructions for high performance, we initially made a mistake that caused the program to compute incorrect results. This was caught by `ps-02` and we updated the code. When we fixed the code, a load-use dependency was unintentionally created and caused a pipeline stall, this was immediately found by `ps-01` and fixed.

4 Related Work

IBM's FoCs program [1] turns RCTL [4] assertions into Verilog modules that hook directly into the simulation infrastructure and monitor execution. When the circuit is simulated, each of the monitors watches how the system state and internal signals change over time and reports any problems. SHERLOCK [6] is a second trace checking tool for digital circuits. It can check simulation traces for violations of properties specified in a linear temporal logic augmented with first-order variables, arrays, and queues. The data structures permit property specification at a higher level of abstraction, thus improving usability and reducing low-level specification errors. However, neither of these tools can easily be used to understand hardware/software interactions.

In the embedded design space, there are industrial tools that allow a certain degree of hardware/software debugging. The Xilinx EDK [25], for example, allows the user to debug his/her software with GDB and to scope the internal hardware signals when a breakpoint is triggered. However, it does not work in the other direction: hardware events cannot be used to stop the software. Furthermore, formal analysis is not supported by these debugging tools.

Ptolemy II [13] is a project focusing on embedded system modeling and design. An entire system, including both hardware and software, can be modeled at varying levels of abstraction using the supported 'models of computation'. The intent is for the modeling of the system to guide its implementation, which in some cases can even be done automatically from the abstracted view. Ptolemy II captures both hardware and software components of an embedded system but is not, in particular, focused on validating a specific implementation of the system.

Trace checking has also been used successfully in software monitoring. For example, see [11,9]. In fact, this has developed into a new software engineering research area called *runtime verification* [19].

5 Conclusions and Future Work

We have presented a formal semantics in Maude of the ABEL HDL, and have explained how this executable semantics can be used as the basis of a co-verification framework. To the best of our knowledge, both the ABEL semantics, and the capabilities for trace checking ABEL co-designs are new contributions. Our experience so far has been quite encouraging, in that rewriting logic and Maude have given us a flexible framework in which to specify and analyze sophisticated properties of both the hardware and the software and how they interact. However, this is still work in progress open to many improvements and new developments. For example, we should improve the efficiency of the HDL simulator to enable larger designs and also add support for other HDLs, or synthesizable subsets of them. Similarly, we should incorporate known techniques for optimizing trace checking, so as to help performance. Also, adding first order variables to the specification logic would be extremely useful, and shouldn't pose a problem for trace checking. Furthermore, we should develop a larger and more ambitions suite of real-life case studies.

Our longer-term goal is to develop new methods and tools to formally specify and analyze embedded systems. This paper advances that goal but does not address a number of important issues that, besides the improvements and extensions mentioned above, will be topics of future research. We have pointed out that for many embedded systems, specifying the environments in which they operate and verifying the properties related to their interactions with such environments, is as crucial as specifying and verifying the hardware/software system itself: both tasks should be done together. Although we have addressed some real-time and performance issues in our case studies, a full modeling of environments, though very important, is beyond the scope of this paper. The natural approach for modeling such environments is viewing embedded systems as *real-time* systems that can be *hybrid*, and can even be both *stochastic and hybrid*. Therefore, from a rewriting logic perspective the natural techniques and tools to use will include real-time rewrite theories [20], probabilistic rewrite theories [3], the modeling of stochastic hybrid systems [17], and tools such as Real-Time Maude [21] and the upcoming PMaude [3] and SHYMaude [17]. This should enable us to handle mechanical and sensing interfaces, such as those in many control systems, for example, anti-lock brakes systems. In the terminology of this paper it will also provide a natural extension of the meta-level properties that can be formally specified and analyzed.

Acknowledgement

We thank the referees for their detailed comments on a previous version that have helped us improve the presentation. In addition, we thank Sean Keller for reviewing a draft of the paper, and Andrew Colombi for a version of the graphics program. This research has been supported in part by ONR grant N00014–02-1-0715.

References

[1] Abarbanel, Y., I. Beer, L. Glushovsky, S. Keidar and Y. Wolfsthal, *FoCs: Automatic Generation of Simulation Checkers from Formal Specifications*, in: *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification* (2000), pp. 538–542.

[2] Abarbanel-Vinov, Y., N. Aizenbud-Reshef, I. Beer, C. Eisner, D. Geist, T. Heyman, I. Reuveni, E. Rippel, I. Shitsevalov, Y. Wolfsthal and T. Yatzkar-Haham, *On the effective deployment of functional formal verification*, Form. Methods Syst. Des. **19** (2001), pp. 35–44.

[3] Agha, G., J. Meseguer and K. Sen, *PMaude: Rewrite-Based Specification Language for Probabilistic Object Systems*, in: *3rd Workshop on Quantitative Aspects of Programming Languages (QAPL'05)*, 2005.

[4] Beer, I., S. Ben-David and A. Landver, *On-the-Fly Model Checking of RCTL Formulas*, in: *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification* (1998), pp. 184–194.

[5] Ben-David, S., C. Eisner, D. Geist and Y. Wolfsthal, *Model Checking at IBM*, Form. Methods Syst. Des. **22** (2003), pp. 101–108.

[6] Canfield, W., E. A. Emerson and A. Saha, *Checking Formal Specifications under Simulation*, in: *ICCD'97: Proceedings of the 1997 International Conference on Computer Design*, 1997.

[7] Edwards, S., L. Lavagno, E. A. Lee and A. Sangiovanni-Vincentelli, *Design of Embedded Systems: Formal Models, Validation, and Synthesis*, Proc. of the IEEE **85** (1997).

[8] Eker, S., J. Meseguer and A. Sridharanarayanan, *The Maude LTL Model Checker*, in: F. Gadducci and U. Montanari, editors, *Fourth Workshop on Rewriting Logic and its Applications, WRLA '02*, Electronic Notes in Theoretical Computer Science **71** (2002).

[9] Feng Chen and Grigore Roşu, *Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation*, in: *Third International Workshop on Runtime Verification*, ENTCS **2003**, pp. 106–125.

[10] Harman, N. A., *Verifying a Simple Pipelined Microprocessor Using Maude*, in: *WADT '01: Selected papers from the 15th International Workshop on Recent Trends in Algebraic Development Techniques* (2001), pp. 128–151.

[11] Havelund, K. and G. Rosu, *Monitoring Programs Using Rewriting*, in: *ASE '01: Proceedings of the 16th IEEE International Conference on Automated Software Engineering* (2001), p. 135.

[12] Hennessy, J. L. and D. A. Patterson, "Computer Organization and Design," Morgan Kaufmann, 1998, 3rd edition.

[13] Hylands, C., E. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, Y. Zhao and H. Zheng, *Overview of the Ptolemy Project*, Technical report, Department of Electrical Engineering and Computer Science, University of California Berkeley (2003).

[14] Janick Bergeron, "Writing Testbenches: Functional Verification of HDL Models," Kluwer Academic Publishers, Norwell, MA, USA, 2003, Second edition.

[15] José Meseguer and Grigore Roşu, *The Rewriting Logic Semantics Project* , in: *Structural Operational Semantics 2005*, ENTCS, 2005.

[16] Manuel Clavel and Francisco Durán and Steven Eker and Patrick Lincoln and Narciso Martí-Oliet and José Meseguer and Carolyn Talcott, "Maude Manual (Version 2.2)," .
URL http://maude.cs.uiuc.edu/

[17] Meseguer, J. and R. Sharykin, *Specification and Analysis of Distributed Object-Based Stochastic Hybrid Systems*, Technical report, University of Illinois at Urbana-Champaign, CS Department (2005), to appear in *P*roc. Hybrid Systems 2006, Springer LNCS.

[18] Micheli, G. D. and R. K. Gupta, *Hardware/Software Co-Design*, Proc. of the IEEE **85** (1997).

[19] Oleg Sokolsky and Mahesh Viswanathan, editor, "RV'03, Workshop on Runtime Verification," Electronic Notes in Theoretical Computer Science **89**, Elsevier, 2003.

[20] Ölveczky, P. C. and J. Meseguer, *Specification of Real-Time and Hybrid systems in Rewriting Logic*, Theoretical Computer Science **285** (2002), pp. 359–405.

[21] Ölveczky, P. C. and J. Meseguer, *Real-Time Maude 2.1* (2004), proc. 5th Intl. Workshop on Rewriting Logic and its Applications.

[22] Sgroi, M., L. Lavagno and A. Sangiovanni-Vincentelli, *Formal Models for Embedded System Design*, IEEE Design and Test of Computers **17** (2000).

[23] Vardi, M., *Designing a Property Specification Language* (2005), presentation at UIUC Formal Methods Seminar.

[24] Xilinx, *Online ABEL Reference*, http://toolbox.xilinx.com/docsan/xilinx7/help/iseguide/mergedProjects/abelref/abelref.htm (2005).

[25] Xilinx, *Platform Studio and the EDK*, http://www.xilinx.com/ise/embedded_design_prod/platform_studio.htm (2005).