Early Validation of Computer Microarchitecture with Algorithm Level Models

Ziyad Hanna
(Ziyad.Hanna@comlab.ox.ac.uk)

Tom Melham
(Tom.Melham@comlab.ox.ac.uk)

Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford, OX1 3QD, England

Abstract. This research addresses the increasing challenges of functional validation for computer microarchitecture designs. We propose a new framework for algorithm-level modelling with a high level of specification and validation. The semantics of our models is based on Abstract State Machines. *Algorithm Level Models* with this semantics are written at a high level of representation in a hardware-oriented adaptation of AsmL, an executable object-oriented language with rich data types. The framework we describe provides a semantic infrastructure for both formal property verification and refinement verification. It also provides a link between Algorithm Level Models at the lowest level of abstraction and Design Models, the industry's golden models for circuit implementation.

This paper is an account of work in progress on this approach and describes some of the details of our proposed framework, illustrated through a case study. We believe our approach will make it easier to explore microarchitectural algorithms and to validate them using dynamic or formal techniques—yielding a more productive convergence to high quality implementations.

1 Introduction

This paper addresses the increasing challenges in functional validation of computer systems at the microarchitecture level. The growing size and the complexity of such systems is outstripping current validation methodologies, and validation has become the main challenge computer engineers face in producing correct and high quality systems [2].

An architecture's *Design Model* (DM) is expressed at Register Transfer Level and usually serves as the golden model for capturing the specification and driving the implementation. It is a detailed model, at a very low level of abstraction typically involving physical design aspects such as placement, timing, and power. Validating the DM is very time consuming, because it is entangled with so much non-functional detail.

Design abstraction is a promising strategy for coping with this complexity. The conventional approach is to write abstract *High Level Models* (HLMs) that exclude implementation details in order to focus on specification and validation. Typically, however, an HLM serves as a reference model for driving the downstream design steps *as well as* for validation. It has therefore tended to compromise the functional design and validation needs by including many details driven by implementation concerns. HLMs eventually become inefficient for validation. Another major drawback is the lack of a formal link from the HLM to the design model; validation work done on an HLM typically has to be repeated on the DM, so the return on investment in maintaining an HLM becomes suspect.

Our contribution is to propose a new framework for modelling and validation focussing solely on the *algorithms* embodied in our designs. We introduce the concept of an *Algorithm Level Model* (ALM) for *modelling and validating* the high level behavior of microarchitectural algorithms. An ALM is different from an HLM because it ignores the non-functional design implementation details and instead focuses *only* on the purely functional behavior.

Another contribution is the adaptation of an executable and abstract state machine specification language, AsmL [8], for algorithm level modelling and validation. AsmL has a precise semantics framed in terms of Abstract State Machine (ASM) theory [3, 7]. It is used mainly for model-based testing of software, but we have found it very suitable for describing and modelling microarchitectural algorithms. There is also a well-defined notion of refinement between AsmL programs, based on their ASM semantics; a major thread of our proposal is to use a sequence of model refinements to connect the ALM to models with more design detail, and ultimately to the DM itself.

In our framework we seek to provide a mathematically well-defined semantic infrastructure for a broad range of formal verification capabilities that complement dynamic validation of AsmL models. We aim to support both formal property verification and checking the correctness of refinement steps. Verification of the formal refinement relationship between the most detailed AsmL model in our framework and the DM

itself can be done using the *assertion program* approach [12] to GSTE model checking [18].

We believe our algorithm-level modelling approach will make it easier to explore microarchitectural algorithms and to validate them using dynamic or formal techniques—yielding a much more productive convergence toward high quality implementations.

2 Algorithm Level Modelling

We introduce Algorithm Level Models (ALMs) as objects that precisely describe the algorithmic behavior of computer systems at high levels of representation.

Several initiatives in the EDA industry have tried to capture systems at a high level of abstraction. A prominent concept has been the *High Level Model* (HLM). An HLM, however, typically serves at least two goals at the same time, dealing with circuit design as well as functional validation. The model's abstraction level then is gradually dragged down during the long design cycle, until at some point it is no longer justifiable to maintain the HLM in parallel with the DM. The HLM is usually abandoned, wasting the costly effort spent on its construction.

Our proposed ALMs, on the other hand, are intended to be the top *specification and validation model only*. They are kept algorithmic and separate from the DM. An ALM can then remain stable and free of implementation detail, and so continue to serve as the main vehicle for functional validation throughout the design cycle. In our approach, we separate validation and circuit design concerns by having two different (but logically correlated) design descriptions: the ALM for specification and validation, and the DM for circuit design needs and downstream convergence to physical representation.

An ALM has the following main characteristics:

Precise, at the appropriate level of detail. It should be flexible, to express just the level of precision or abstraction required. It should be easily modifiable, extensible for reuse, and adaptable to different formal verification activities (e.g. model checking or interactive theorem proving).

Simple and concise, to be understood by both system architects and designers. This will also make analysis of the model for consistency and completeness more manageable.

Abstract, yet sufficiently complete. All semantically relevant features of the system should be represented in the model, which should otherwise be abstract and minimal. It should include all the relevant interfaces to interact with the embedding environment, including modelling of environmental constraints. It need not be cycle-accurate.

Executable. The model can be run at a suitable level of abstraction using concrete test-bench and run-time environments.

Suitable for Hardware. The model should provide a semantics suitable for hardware—correctly modelling concurrency, synchronization, clocking, hierarchy and modular composition.

Checking the compliance or logical compatibility between the ALM and DM is a key challenging problem in our methodology. We adopt the approach of filling the gap betwen the ALM and the DM with a series of model refinements. Each step of refinement is validated, either dynamically, or with formal verification, or both.

2.1 ASM and AsmL

We believe that Abstract State Machines (ASM) [3] and a version of the Abstract State Machine Language (AsmL) [8] enriched with hardware data modelling are ideal for our Algorithm Level Modelling goals. They provide:

Computation. ASMs provide a model of computation structured into state and a transition relation. Some existing finite-state machine based techniques can be extended relatively naturally for ASMs.

Abstraction: Moving from Booleans to abstract and rich data types enables a great flexibility to specify systems at an abstract level. This will capture the specification in a more compact representation than what we have today in RTL languages.

Concurrency: Hardware components run in parallel in a series of computational steps, that typically align with each clock cycle (possibly with multiple clock domains). The AsmL operational semantics supports with high flexibility the interchange between serial and parallel execution.

Refinement: An ASM inherently supports the stepwise refinement concept. We use refinement to bridge the abstraction gap between the top level ALM and the DM.

Control and data separation: The update rules in a AsmL are created in such a way that the guard of an update is easy to identify and isolate from the update operation itself. It is possible to leverage this for applying various data abstraction techniques, reducing the verification complexity.

In the initial phases of our work, we have begun our modelling with AsmL and a formally-described interpreter for AsmL-S [8]. To support our approach, we need to assign a machine-represented semantics to AsmL that faithfully represents the underlying ASM model of the system and is general enough to allow mapping into more optimised representations for specific verification technologies. Although object-oriented in presentation, AsmL is quite naturally embedded in a typed functional setting, in which ab-

stract state is represented mathematically by the elements of types, or sets, and the model's specification of the transitions between states is represented by a function that computes collections of state updates.

We have therefore chosen to express the published semantics of AsmL within a mechanized, but mathematically-precise, language of functions and types—namely the *reFLect* functional programming language [6]. Abstract data types are represented by *reFLect* types, together with suitable predicates, relations and functions, over these types; the current state of a model is represented by a collection of (named) values drawn from these types; the transition system is represented by a function that computes a set of updates to the state; and a run of the system is represented by the sequence of successive abstract states.

By situating our functional language within a suitable, mechanized theory of sets and functions, we provide a default mathematical interpretation of our semantics that has several appealing features. First, the semantics of an AsmL model is just a functional program, so we can 'run' it on concrete states for testing (which, of course, AsmL also provides directly) or equivalence validation against the AsmL model from which it is derived. The semantic model is abstract and compact—of the same order of magnitude in size, we expect, as the original AsmL program. Because it exists within a well-defined mathematical theory, we can reason about the semantics directly using decision procedures, deductive theorem proving, or even symbolic simulation in *reFLect*.

Eventually we expect to extend our language with certain hardware-oriented features and data structures found in more conventional hardware description languages. An overriding principle, however, will be to keep the language clean and abstract.

3 Scheduler Case Study

A microoperation (μ-op) scheduler is an algorithmic microprocessor component that we use to demonstrate the ALM concepts in this work. It is an interesting test case because it is a typical μArch algorithm that represents those types of models to which algorithm level modelling and validation concepts will be applied.

The Scheduler receives a stream of μ-op instructions to be executed and is responsible for delivering each of these to an execution unit at the appropriate time. The scheduler must hold back some μ-ops until the execution unit signals that their operands are available. When multiple μ-ops are ready, the scheduler must follow the in-order execution—FIFO policy for selection. For this study, a simple version of the scheduler is considered. Details of a real verification of a μ-op scheduler can be found in [19].

Each μ-op consists of an opcode, a source register and a destination register. During a run of the scheduler, each μ-op carries with it a *ready* bit. An instruction's *ready* bit is set to high when all the instructions on which this one must wait have already been executed. In order to enqueue a μ-op into the scheduler (provided the *full* line is not high), the *write* line of the scheduler must be set high, the μ-op presented on the *din* lines and the *ready* bit for this μ-op presented at the *ready* line. The *wrback* line from the execution unit is used to signal when the execution of an instruction has resulted in the writing of data to a register. The index of this register is supplied on the *reg* lines. This allows the scheduler to select any μ-ops it holds which might have been waiting for this write to happen (for example if they must read from this register themselves).

The design makes the environmental assumption that any μ-op which is enqueued in the scheduler is waiting for at most one other instruction to execute. With this assumption, when a write-back takes place, if any waiting instructions have a source register which matches *reg* then their *ready* bits can safely be set to high. The scheduler should set the line *available* to high when it contains a waiting ready μ-op. When it receives this signal, the execution unit can request a μ-op on the *dout* lines by setting *read* to high. In the event that there is more than one ready μ-op, the scheduler provides the ready μ-op which entered the scheduler first.

In our case study, the scheduler is a class type that holds the global state of the ALM with local data fields for holding the μ-op instruction queue. Instructions are represented as a class using constrained basic types as shown below:

```
type opcodeType = Integer
       where value in 0..maxOpcode
structure uOp
  opCode as opcodeType
  sReg as regType
  dReg as regType
  ready as Boolean
class uOpState
  u as uOp
  var time as Integer
  var status as uopStatus
```

The transition behavior of the scheduler is denoted by the *scheduler()* method. At the top level model of abstraction, we use a set (in the natural mathematical sense), *uopQueue*, to represent the global state of the ALM. The set is initially empty, and is populated during execution. The arrival time of each μ-op is tagged by *time* variable so it will be possible to associate a relative order among the μ-ops in the set. The *dout* variable represents the scheduled μ-op output and initially has the value *nop* (no operation).

```
var uopQueue as Set of uOpState = {}
var dout as uOp = nop
var time as Integer = 0
```

At each *step* of execution, parallel synchronized statements run and update the global state. For example, when *write* is enabled a new object is created, that is stored in the next step of execution.

During *read*, assuming in-order execution, the scheduler looks for the earliest ready instruction in the set, using quantified logic.

```
if read then
  let uop = the u | u in uopQueue
    where u.status = rdy and
      not (exists v in uopQueue
           where v.time < u.time and
                       v.status = rdy)
  uop.status := exec
  dout := uop.u
  remove uop from uopQueue
```

During write-back from the execution unit, the scheduler sets each μ-op state to ready if its source register has been written to. The update of the μ-op status is done in parallel using the *forall* statement.

We recall that one of the main advantages of an ALM is being executable, and through dynamic testing we can examine the behavior of ALM in SpecExplorer [16] and easily find run-time bugs. For this example, we built a test bench that lived together with the ALM scheduler and sampled and drove the scheduler with control and data. In this framework, tests can be generated either manually or automatically. The environment and the ALM execute in parallel, and during each step the environment chooses how to drive the scheduler with new data and the scheduler responds accordingly. The validation of this example was done using the SpecExplorer.

4 Refinement

The ALM concept is valuable when talking about modelling and validation at high levels of abstraction, however linking it to design models is a great challenge because of the large abstraction gap between ALMs and design models. For bridging the gap, we can apply known refinement ideas on an ALM to provide a crucial capability for supporting the system design and validation environments.

There are several types of refinements, and in particular we focus on *data* refinement, *algorithm* refinement, and *hybrid* refinement—which combines both. In this this case study we demonstrate the *hybrid* refinement in two steps. First we change the μ-op container data structure from a set to a sequence, and add an another *valid* sequence to mark the occupied entries in the μ-op sequence. The μ-ops are stored in the *uopQueue* as decided by the environment according to the empty entries in the *valid* sequence. The arrival times are still tagged to each μ-op by a representation of time. An algorithm refinement is needed to comply with the data refinement in the new state structure. For this, various methods of the refined scheduler need to be changed. For example the read method is changed as following:

```
if read then
  let slot = the i | i in queueRange
    where valid(i) and
      uopQueue(i).status = rdy and not
     (exists j in queueRange where
        uopQueue(j).time <
         uopQueue(i).time and valid(j)
       and uopQueue(j).status = rdy)
  uopQueue(slot).status := exec
  dout := uopQueue(slot).u
  valid(slot) := false
```

This refinement is indeed a small step as it yields a new specification that is similar to the previous specification, e.g. we still use a time-stamp, which from the implementation side has two problems. First the number of incoming μ-ops is unlimited or not known up front, so we need to reset the time representation occasionally (e.g. when *uopQueue* is empty). The second problem is that we need to apply an expensive hardware operation to compare the integer numbers for finding the earliest ready μ-op for execution. To make it close to circuit implementation, which is more efficient, we are going to do another refinement step by adding a priority matrix, *pm*, that indicates if uop_i is earlier than uop_j if the $pm[i][j]$ is true. While doing this the scheduler can rapidly calculate the relative order of ready μ-ops. Powered with this technique we can get rid of the *time* variable.

Validating the refinement steps can be done using intensive simulation, where the refinement models under the same tests can independently compute the μ-op sequence for execution (at the *dout* line). The results are compared at the end of each computation step. For this example we did an extensive simulation where tests were generated randomly and fed in parallel into the three scheduler versions. The three results are compared at each execution step.

5 Framework Summary

Now that we have described all the major components and technologies of ALM, we sketch in Figure 1 the entire flow and framework. The user defines the top level model in an abstract language, in our case AsmL, and then gradually refines it by increasing the level of detail according to certain implementation aspects. At each abstraction level, the AsmL model can be read into the validation environment (i.e. SpecExplorer), and can be quickly checked for correctness. The user can run the executable specification and test it on a large regression suite generated either automatically or manually. During this phase, the dynamic validation environment provides a quick and productive way to examine the design, flush out easy logic errors and provides a fast turn around to fix them. After gaining some confidence about the correctness of the design, the AsmL model can be loaded into the formal verification environment, preserving its semantics, for building the internal ALM representation in *ReFLect*. Then formal verification tasks,

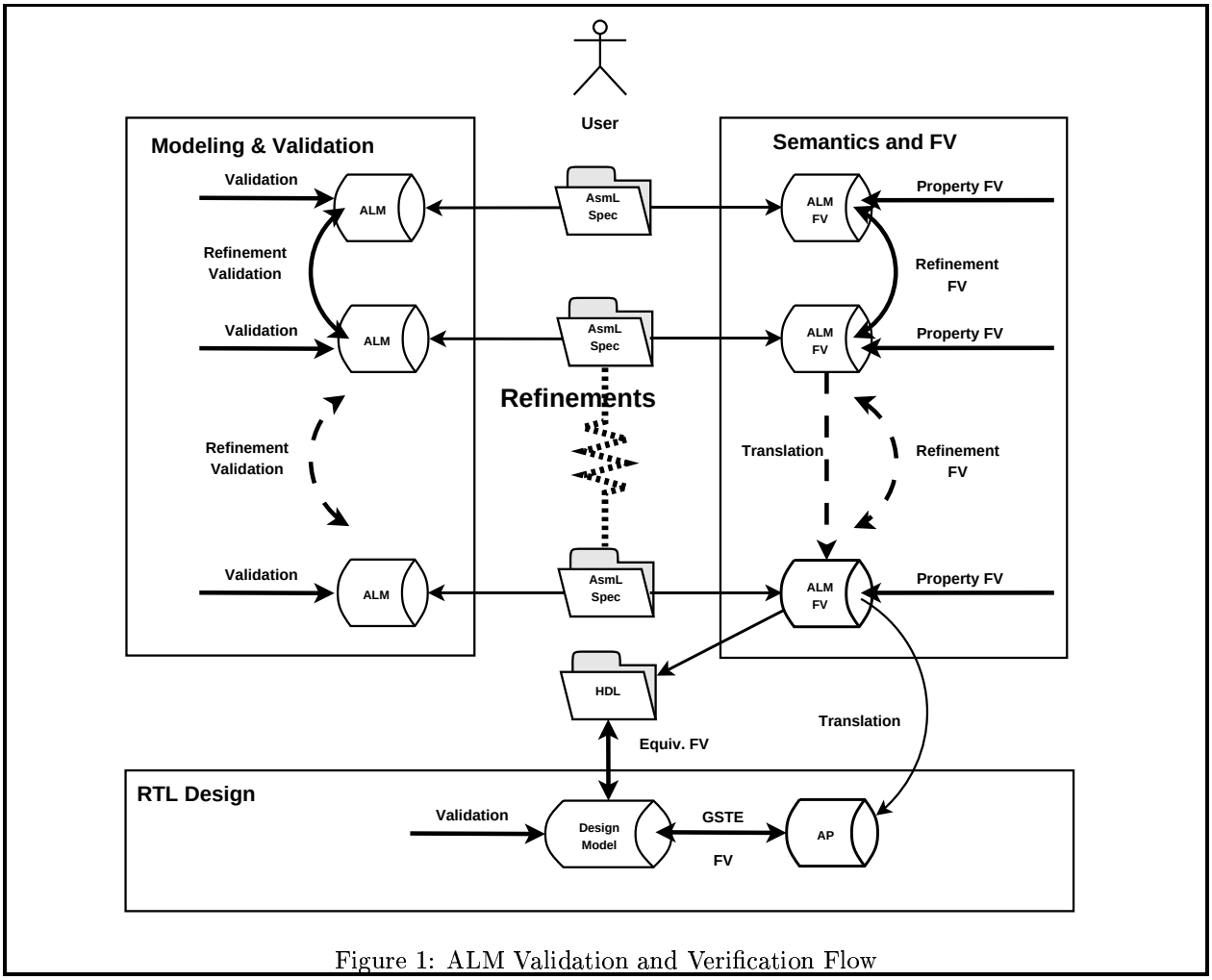

Figure 1: ALM Validation and Verification Flow

like formal property specification and verification can take place on the ALM. The correctness of the user generated refinement can be checked either dynamically in the validation environment or formally in the FV environment.

At the lowest level of abstraction of the AsmL, the verification environment can support translation of the AML to an assertion program (shown in the figure as AP). The specification is then in a suitable form for driving symbolic circuit simulation using GSTE, providing the final refinement step to the gate-level design. We can also translate the ALM into a hardware description language such as *SystemVerilog* and apply standard sequential equivalence verification methods (e.g. [11]) to prove the functional equivalence between the ALM and design.

Within the FV framework, we can take advantage of the precise representation and semantics of the ALM and apply design or program transformation methods to further refine the ALM. Theoretically, we believe that we can apply sequences of transformations to get to the lowest level of abstraction, however this part is out of our research scope for now.

6 Related Work

SystemC [4] mathematical semantics and its simulation kernel are not fully clear, so attempts to perform formal property verification or other static analysis are usually not well supported. Tahar and Habibi [9] address this with a verification methodology for SystemC designs based on a combination of static code analysis and semantics described in AsmL. From our perspective, this work has shown that SystemC semantics can be expressed in ASM and therefore confirms that ASM is a suitable focus for our research in model validation.

SystemVerilog [1] simulation kernel semantics can be mapped to ASM. It seems reasonable to expect that the semantics of the SystemVerilog language as a whole can also be mapped to ASMs in away similar to the work by Tahar and Habibi on SystemC.

Bluespec [10] computational model is very similar to ASM, however, it is mainly aimed at modelling and synthesis of designs, rather than formal verification of algorithms. It supports only a limited set of data types aimed at easing synthesis, whereas our language has higher level data types.

Work on verification and ASM includes the lim-

ited study of the formal link from ASM to symbolic model checking in SMV employed by Winter [17]. An extension of this work was done in collaboration with Tahar for translating ASM models to Multiway Decision Graphs (MDGs [13]), a data structure for representing transition systems over abstract data types.

Ternovska [15] approached formal verification for ASM using on bounded model checking and Answer Set Programming (ASP) techniques. This work used only a subset of ASM with Boolean or enumerated datatypes, and so encoding into bits and Boolean solving is a natural choice for verification technology.

Research on bridging the gap between ESL and RTL models is still at early stage. We do not yet see major achievements in this area, except perhaps for the some attempts to verify equivalence of SystemC and Verilog designs in the EDA industry [14], and a by Clarke and Kroening [5] on C/SystemC to Verilog verification.

The key difference between our work and these is that we focus exclusively on *algorithmic* with a clean, semantically well-defined, and simple core language. We see modelling languages such as SystemC, SystemVerilog or Bluespec as complements to our research that offer a richer front end and dynamic validation environment. A semantics of such models can be given by translation into our ALM language and ASM semantics.

Our ALMs are intended to be abstract enough to express algorithms in natural terms, which should allow reasoning at a higher level than cycle-accurate models at a bit level of representation. The expectation is that this will give greater capacity than that which exists today in bit-level models. We also plan to explore and develop scalable verification techniques to formally link the ALM with the detailed DM, keeping our ALM verifications relevant to design implementations.

7 Status of Our Work

We have completed the shallow embedding of the core fragment of AsmL-S, published in [8], into *reFLect*. A front end parser was also developed in C/C++ to support AsmL-S source specifications. This provides us with an open research platform in which we can easily customize the language and its semantics in reFLect to suit our needs. We want to extend the language to include hardware related primitive data types, such as bitvectors, arrays, and sets so it will be easier to specify microarchitectural algorithms. In addition, we are currently looking into the verification methods that we want to apply on ALM, including in particular symbolic simulation and bounded model checking technologies.

8 Acknowledgments

The authors thank Steve McKeever and Joel Ouaknine for helpful comments on a draft of this paper. We are grateful to the ASM research team at Microsoft, including Yuri Gurevich, Wolfram Schulte and Davor Runje, for their generous support with AsmL and SpecExplorer.

References

[1] IEEE Accellera. *http://www.accellera.org.*

[2] Bob Bentley. Validating a modern microprocessor. In *Computer Aided Verification: 17th International Conference: Proceedings*, volume 3576 of *LNCS*. Springer Verlag, 2005.

[3] Egon Börger and Robert Stark. *Abstract State Machines.* Springer-Verlag Berlin Heidelberg, 2003.

[4] Calypto. *http://www.calypto.org.*

[5] Edmund M. Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of c and verilog programs using bounded model checking. In *DAC*, pages 368–371, 2003.

[6] Jim Grundy, Tom Melham, and John O'Leary. A reflective functional language for hardware design and theorem proving. *Journal of Functional Programming*, 16(2):157–196, March 2006.

[7] Yuri Gurevich. Evolving algebras: An attempt to discover semantics. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 266–292. World Scientific, River Edge, NJ, 1993.

[8] Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte. Semantic essence of AsmL. *Theor. Comput. Sci.*, 343(3):370–412, 2005.

[9] Ali Habibi and Sofiène Tahar. An approach for the verification of systemc designs using asml. In *ATVA*, pages 69–83, 2005.

[10] James C. Hoe and Arvind. Synthesis of operation-centric hardware descriptions. In *ICCAD*, pages 511–518, 2000.

[11] Zurab Khasidashvili, Marcelo Skaba, Daher Kaiss, and Ziyad Hanna. Theoretical framework for compositional sequential hardware equivalence verification in presence of design constraints. In *ICCAD*, pages 58–65. IEEE Computer Society / ACM, 2004.

[12] Edward Smith. A method for generation of GSTE assertion graphs. In Robert Jones and Ian Harris, editors, *High-Level Design Validation and Test Workshop, 2005. Tenth IEEE International*, volume 10, pages 160–167. IEEE Computer Society, 2005.

[13] Mandayam K. Srivas and Albert John Camilleri, editors. *Formal Methods in Computer-Aided Design, First International Conference, FMCAD '96, Palo Alto, California, USA, November 6-8, 1996, Proceedings*, volume 1166 of *Lecture Notes in Computer Science*. Springer, 1996.

[14] SystemC. *http://www.systemc.org*.

[15] Calvin Kai Fan Tang and Eugenia Ternovska. Model checking abstract state machines with answer set programming. In *LPAR*, pages 443–458, 2005.

[16] Margus Veanes, Colin Campbell, Wolfram Schulte, and Nikolai Tillmann. Online testing with model programs. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 273–282, New York, NY, USA, 2005. ACM Press.

[17] Kirsten Winter. Towards a methodology for model checking asm: Lessons learned from the flash case study. In *Abstract State Machines*, pages 341–360, 2000.

[18] Jin Yang and Carl-Johan H. Seger. Generalized symbolic trajectory evaluation - abstraction in action. In Mark Aagaard and John W. O'Leary, editors, *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*, volume 2517 of *LNCS*, pages 70–87. Springer, 2002.

[19] Jin Yang and Carl-Johan H. Seger. Compositional specification and verification in GSTE. In Rajeev Alur and Doron Al. Peled, editors, *16th International Conference on Computer Aided Verification (CAV)*, volume 3114 of *Lecture Notes in Computer Science*, pages 216–228. Springer, 2004. Boston, MA, USA, July 13–17.