A Symbolic Execution Framework for Algorithm-Level Modelling

Ziyad Hanna and Tom Melham
Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford, OX1 3QD, England
{zhanna,melham}@comlab.ox.ac.uk

***Abstract*—This work aims to address the well-known and acute challenge of functional validation for complex, contemporary microarchitectural circuit designs. We provide a new formal framework for *algorithm level modelling*—design modelling at a high abstraction level, focused exclusively on function and algorithms. The semantics of our models is based on Abstract State Machines with synchronous parallel execution, sequential execution, and nondeterminism. To express models we propose an executable, object-oriented *Architecture Specification Language* with rich data types and a well-defined formal semantics, based initially on Microsoft's AsmL. We describe an experimental framework for direct *symbolic execution* of models in this language, intended as a basis for both property and refinement verification, as well as design exploration.**

We explain and illustrate our approach through a case study, the modelling a simple μop scheduler and its refinement towards a design model for circuit implementation. We aim to show the utility of our language and symbolic execution framework for exploring microarchitectural algorithm and to validate designs using dynamic or formal techniques, yielding more productive convergence to high quality implementations.

I. Introduction

Functional validation has become an acute and expensive challenge for engineers designing high performance micro-electronic systems, especially in quickly evolving markets or where there are demanding time to market goals [1]. Most design activity, at least for complex microarchitecture, is still centred around low-level design models, encumbered with implementation detail. Numerous proposals have been made, in our view rightly, to make design exploration and analysis more tractable by raising the level abstraction at which designs are described—so called 'high-level' or 'transaction-level' modelling [2], [3], [4], [5]. The case is made forcefully and exceptionally clearly by Vardi in [6].

Our contribution in this paper is to propose a new framework for modelling and validation, focusing exclusively on *function* and *algorithms*, based around a language with a clean and obvious *formal semantics*, and providing *native symbolic execution* as its fundamental formal analysis tool. Our system integrates formal verification and dynamic validation into a common modelling and analysis framework, and leverages emerging technologies for reasoning above the bit level [7] to enable a genuinely 'high-level' approach.

The aim of this long-term research is to take an experimental step away from the more incremental, industrial efforts represented by SystemC and similar approaches. We wish to experiment with a more disruptive language and reasoning framework, and to assess how the particular ideas represented in our system might make it easier to explore microarchitectural algorithms and validate them using dynamic or formal techniques—yielding more productive convergence to high quality implementations.

In the sections that follow, we describe our *Architecture Specification Language* through a case study, the modelling of a simple μop scheduler and its refinement towards a design model for circuit implementation. Our language, still in the early stages of development, is based on Microsoft's open source AsmL [8], [9] extended with some hardware-oriented datatypes; we briefly sketch its semantic foundation in Abstract State Machines. We also describe an experimental interpreter that supports *symbolic execution* of models in this language, intended for design exploration, property checking, and refinement verification. We conclude by discussing related work, and outlining future research challenges and prospects.

II. Algorithm Level Modelling

We use the term *algorithm level model* (ALM) to mean a precise description of the functional and algorithmic behaviour of computer systems, framed in terms of abstract data types and granularities of time not necessarily tied to clock cycles. An ALM has the following main characteristics:

Abstract, yet sufficiently complete. All and only the algorithmically relevant features of the system should be represented. It need not be cycle-accurate or expressed at the bit level.

Simple and concise, written in a language with meaning transparent to both system architects and designers.

Precise, with a comprehensive and tractable formal semantics. This should be suitable to support a range of different formal verification technologies.

Hardware-oriented. The model should provide a semantics suitable for the *abstract* characteristics of hardware—correctly modelling concurrency, synchronisation, clocking, hierarchy, and modular composition.

Executable. The model can be run when encapsulated within a suitable test-bench and run-time environment.

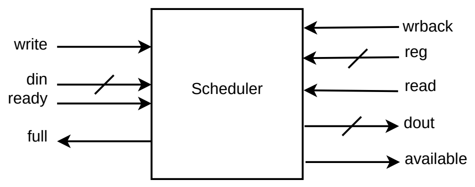

Fig. 1. Scheduler Top Level Interface

There have, of course, been many attempts to devise high-level models (HLMs) that share some of these desirable goals. Often, however, an HLM is expected to serve as a golden reference for downstream design steps *as well as* for validation. It has therefore tended to compromise validation needs by including information driven by circuit design concerns. Implementation details—such as power, timing and placement—inevitably become tangled up with the model. The model's abstraction level is dragged down to serve more design purposes, until at some point it is no longer has the merits of the clean models envisaged above.

We therefore speak of 'algorithm-level models' to emphasise that they are intended to model only data structures and algorithms, and to be used only for the purpose of algorithm exploration and validation. They are kept strictly separate from the RTL design model, and so can remain a stable algorithmic description and functional specification, free of implementation detail.

III. Algorithm Specification Language

Starting with an enlargment of the AsmL subset defined in [10], and adding some hardware-oriented data types and operations, we have devised an experimental language for algorithm level modelling and verification called the *Architecture Specification Language* (ASL). In this section, we provide a sketch of ASL through examples drawn from a case study.

A. Scheduler Case Study

A micro-operation (μop) scheduler is a microprocessor component that we have used to drive ASL development in this research. It implements functionality that is typical of the kinds of microarchitectural algorithms we wish to validate through ALMs, and to refine down to implementable designs.

The scheduler, shown in figure 1, receives a stream of μop 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 μops until the execution unit signals that their operands are available. When multiple μops are ready, the scheduler uses a FIFO policy for selection. For this study, a simple version of the scheduler is considered; the verification of a real RTL μop scheduler is described in [11].

Each μop has an opcode, a source register, and a destination register. Each μop comes to the scheduler on the *din* line accompanied by a *ready* bit, which is high exactly when all the instructions it must wait for have already been executed. In order to enqueue a μop into the scheduler (provided the *full* line is not high), the *write* line must be high. The scheduler should set the line *available* to high when it contains a waiting ready μop. When it sees this signal, the execution unit can request a μop on the *dout* lines by setting *read* to high. In the event that there is more than one ready μop, the scheduler provides the ready μop which entered it first.

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 μops it holds which might have been waiting for this write to happen. When a write-back takes place, any waiting instructions whose source which matches *reg* is tagged as ready for execution.

B. ASL Examples from the Scheduler Model

In ASL, we model the scheduler as a class with data fields for its internal state. This comprises the queue of μops, each accompanied by a status flag, the data output register, and a running index for the time of μop entry. Instructions and status tags are represented the using of the basic types

```
class UOP_BASE {
  opCode as Integer
  sReg as Integer
  dReg as Integer
}
class UOP extends UOP_BASE {
  status as Integer  // idle, wait, rdy, exec
}
```

and the scheduler's internal state is represented by

```
uopQueue as Map of Integer to UOP
dout as UOP_BASE
uopTag as Integer
```

At this level of abstraction, we represent the state of the queue by a map (in the natural mathematical sense) from arrival time to the μop and its status bit. The map is initially empty, and is populated during execution. The running time index is held in `uopTag`.

At each step of execution, a `schedule` method is invoked with the scheduler's inputs as arguments. This executes a group of parallel update statements to compute a modification of the scheduler's state. For example, when `write` is enabled and the queue is not full, the updates are as follows:

```
if write and not me.isfull() then (
 me.uopQueue{me.uopTag} := din;
 me.uopQueue{me.uopTag}.status :=
        if ready then rdy else wait;
 me.uopTag := me.uopTag + 1
)
```

The identifier `me` is similar to 'this' in C++. Used within a method of an object, it makes run-time reference to the object itself. The updates are scheduled in sequence here, but they could just as well be done in parallel.

During a read, the scheduler looks for the earliest ready μop in the map. In ASL, we express the updates as follows:

```
if read and me.isReady() then
 let tag = the w | w in keys me.uopQueue where
     me.uopQueue{w}.status == rdy and
     not (exists v in keys me.uopQueue where
          v<w and me.uopQueue{v}.status == rdy)
 do me.uopQueue{tag}.status := exec;
    me.dout := me.uopQueue{tag}
```

The **exists** and **the** constructs are part of a family of executable comprehension and quantification constructs that ASL (following AsmL) provides for compact specification. They specify values abstractly, instead of by explicit search.

On an execution unit write-back, the scheduler sets each μop status to ready if its source register has been written to:

```
if wrBack then
  forall tag in keys me.uopQueue
   let uop = me.uopQueue{tag} do
    if uop.sReg = reg and uop.status = wait then
      uop.status := rdy
```

Updates to the all μop status bits in the queue are done in parallel, using a **forall** statement.

IV. Symbolic Execution

A distinguishing feature of our language framework is an implementation of direct *symbolic execution* [12] of ASL programs. Inspired by the success of symbolic simulation in, for example, Intel's Forte system [13], we provide this capability as a fundamental mechanism upon which we expect a range of formal verification methods to be built.

Ordinary execution of an ASL program computes an accumulated update to the initial program state—a collection of concrete values that all the state elements take on in the next state of the system. With our symbolic execution mechanism, we inject *meta-variables* into the computation to stand for the values of selected state elements. That is, instead of making the initial value of a field a concrete value, such as `0` or `true`, we make the initial value of the field a *variable*. Execution then produces *expressions* that give the final state of the system as a function of the variables that occur in the initial state. A key feature of our approach is that concrete and symbolic execution can be mixed, in almost arbitrarily flexible ways, under user control. You can write '**meta** "x" **as** *type*' anywhere a literal value can be written, and this will inject the meta-variable `x` at that point of the computation.

We illustrate the idea with the example below:

```
class counter {
   count as Integer,  // state
   inc () as Void (me.count := me.count + 1),
   dec () as Void (me.count := me.count - 1),
   run (up as Bool, dn as Bool) as Integer (
     (if up then me.inc()
         || // parallel composition
      if dn then me.dec())
     ; // sequential composition
     me.count)
}
// main program appears below
let c = new counter(meta "x" as Integer) do (
let u = meta "u" as Bool do
let d = meta "d" as Bool do
    c.run(u,d)
)
```

The class `counter` has a single state variable, `count`, and a `run` method that updates it by parallel, conditional invocations of the other two methods, `inc` and `dec`. In the main program, we create an object `c` of class `counter` and initialize the state to an arbitrary integer, represented by the typed meta-variable `x`. The `run` method is called with two Boolean meta-variables as parameters. When this program is executed, it computes the following expression as the next-state value of `count`:

$$\mathtt{u} \Rightarrow (\mathtt{d} \Rightarrow \text{error } \textit{“inconsistent update”} \mid \mathtt{x+1}) \\ \mid (\mathtt{d} \Rightarrow \mathtt{x-1} \mid \mathtt{x})$$

The notation '$P \Rightarrow A \mid B$' denotes a conditional choice of value: if P then A else B. The result encodes four possible values of `count` in the next state, according to the values of `u` and `d`. Note that when both `u` and `d` are true, the outcome is an inconsistant state. With expressions such as these, users can debug and avoid such inconsistent updates. ASL provides an **assume** construct to impose state constraints that restrict the computation to legal paths. State predicates can be checked using **assert**.

This example is simple straight-line code, but in real ASL programs there are loops and possibly rather sophisticated control flow. Symbolic execution of loops requires a termination condition to be established, otherwise the program never halts. An example is given below:

```
class A {a as Bit[3]}
let x = new A (meta "k" as Bit[3]) do (
   (while (x.a < 3'b011) do
           x.a := x.a + 3'b001)
   ; // sequential composition
   x.a
)
```

The state in this example is a bitvector of width 3, interpreted as a 2's complement integer. It is initialized to an arbitrary symbolic value using the meta-variable `k`. Each time through the loop, `x.a` is incremented by 1 until it becomes 3. In our implementation, the symbolic execution engine checks the loop condition at every step to decide whether to terminate or continue. Eventually, the condition is proven to be false, and the program terminates with an update value of 3 for `x.a`.

ASL supports set and list comprehensions, as well as nondeterministic choice. We illustrate the symbolic execution of these with the following example:

```
class A {a as Integer}
let x = new A (0) do
let n = meta "n" as Integer do
let c = meta "c" as Bool do
 assume n==2 do (
  if c then (
   let res = any i | i in {1,2,3,4} where i<=n do
       x.a := res
  )
)
```

The ASL **any** construct (following AsmL) makes a nondeterministic choice of value drawn from a given set and which satisfies a stated condition. The integer `res` will be a value between 1 and 4 that does not exceed `n`, which itself is known to be 2. In other words, `res` will be 1 or 2.

When we run this program, our system will compute the following symbolic update for `x.a`:

$$(\texttt{c} \wedge \texttt{n=2}) \Rightarrow (i_0 \Rightarrow \texttt{1} \mid \texttt{2}) \mid \texttt{0}$$

In normal execution, a random choice between 1 and 2 would be made for the next value of `x.a`. In symbolic execution, we index all possible outcomes by generating index variables, in this case i_0, and constructing a decision tree to encode the outcome. It is, of course, enough to have $\lceil \log_2 k \rceil$ Boolean variables for k possible choices.

V. Refinement of the Scheduler

Establishing and maintaining a formal link between an ALM specification in ASL and an RTL design model is central to realising the long-term value of algorithm level models. The size of the abstraction gap is a significant challenge, which we propose should be bridged through a series of ALM refinements. Ultimately, these should be semi-automated, or at least machine assisted—tackling the other main concern of the cost of maintaining an ALM—but this is future work. Martin explores similar ideas in System-ML [14], as does Seger in IDV [15].

Our initial focus is on *data* refinement, *algorithm* refinement, and *hybrid* refinement that combines both. In data refinement, the state representation is replaced in the refined model with a more implementation-oriented and efficient one, but the computation method is the same. *Algorithm* refinement, on the other hand, replaces the algorithm by another, more efficient, algorithm, but retains the data representation. In *hybrid* refinement, the most common form, both data and algorithm refinements are involved.

A. Refined Scheduler ALM

In this section, we sketch a hybrid refinement of the scheduler. In our initial ALM, we used an infinite queue to store incoming μops according to arrival time, maintained by an integer index. This is an abstract model, but not close to hardware implementation. The main problem is that the initial model uses a mathematically perspicuous but expensive operation to find the earliest ready μop. To move closer to high-performance circuit implementation, we add a scoreboard that indicates if μop_i is earlier than μop_j across a certain range of i and j. Each μop is compared to all the others in parallel.

The refined scheduler is shown in Figure 2. Its state is modelled in ASL by

```
uopQueue as Map of Integer to UOP,
valid as Map of Integer to Bool,
scoreBoard as Map of Integer to Bool
```

The array `valid` is used to indicate the occupied slots in the μop queue. The scoreboard is a one-dimensional array of Booleans, but will be used as a two-dimensional matrix.

This refinement also replaces the algorithms for reading and writing. We show the read code below:

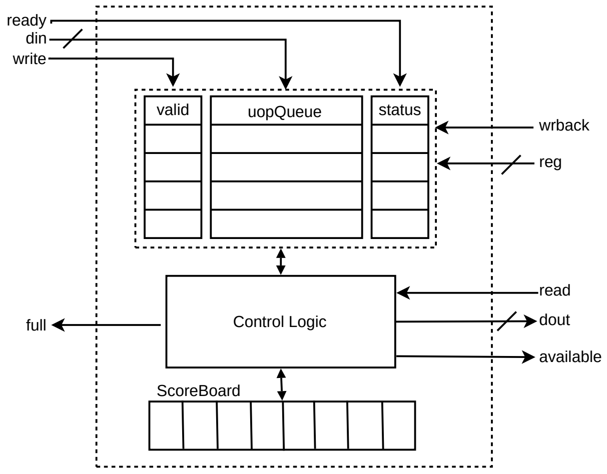

Fig. 2. Refined Scheduler

```
if read and isReady() then
 let slot = the s | s in keys valid where
     valid{s} and uopQueue{s}.status == ready and
     not (exists t in keys valid where t <> s and
            scoreBoard{t*qsize+s} and valid{t})
 do (valid{slot} := false;
     forall v in keys valid where v<>slot
     do (if scoreBoard{slot*qsize+v} then
            scoreBoard{slot*qsize+v} := false;
         scoreBoard{v*qsize+slot} := true);
     uopQueue{slot}.status := exec;
     dout := uopQueue{slot})
```

In the refined read operation, the earliest ready μop is found using the one-dimensional `scoreBoard` as a priority matrix. After reading, the corresponding slot in the *valid* array is cleared and the relation in `scoreBoard` updated.

B. Validation of the Refinement

Validating ASL refinements can be done initially using intensive simulation. The two models are exercised by a testbench, also written in ASL, that runs them on the same inputs. The results are compared at the end of each computation step. For the scheduler case study, we did random simulation.

Test data can be a mixture of concrete and symbolic values. In the following testbench, which runs only one cycle of each model, the μop is symbolic and the control inputs are concrete:

```
let din = new UOP(idle,
                  meta "opc0" as Integer,
                  meta "src0" as Integer,
                  meta "des0" as Integer) do
let inp =
    new INTERFACE(true,false,false,din,0,true) do
queue.schedule (inp);      // abstract scheduler
scoreBoard.schedule (inp); // refined scheduler
let res1 = queue.read_dout() do
let res2 = scoreBoard.read_dout() do
if res1 == res2 then writeln "passed"
                else writeln "failed"
```

An `INTERFACE` is just a record packaging up the inputs.

Verification of the refinement can also be done completely symbolically. We execute the two models with the same symbolic inputs and compare the expressions generated for dout. The symbolic input generator shown below:

```
class SYMBOLIC { // build symbolic transaction
inp (i as Integer) as INTERFACE (
  let write = meta "wr"^(int2str i) as Bool do
  let read = meta "rd"^(int2str i) as Bool do
  let wb = meta "wb"^(int2str i) as Bool do
  let ready = meta "rdy"^(int2str i) as Bool do
  let din = new UOP( idle
   meta "opc"^(int2str i) as Integer,
   meta "src"^(int2str i) as Integer,
   meta "des"^(int2str i) as Integer) do
  let reg = meta "reg"^(int2str i) as Integer do
  new INTERFACE (write, read, wb, din, reg, ready)
 )
}
```

The method inp takes a cycle number and generates meta-variables to represent the inputs values at the scheduler interface. The result is a high-level 'symbolic transaction' defined over the abstract data types of ASL. For our case study, we were able to execute both ASL models in this completely symbolic manner and prove that the outputs agree.

C. Semantics of ASL

The state of an ASL program is encapsulated within classes. Evaluating **new** c(e) allocates a new object of class c, initializing its state to the value of e. The result is a unique *object identifier* for the object created. For example,

```
class A {f as Integer} new A(7).f
```

evaluates to 7, the initial value stored in field f of the allocated object of class A.

ASL is in essence a language for describing synchronous parallel updates to state. The fundamental way to generate updates is with an update expression e1.f := e2, where e1 evaluates to an object identifier and e2 to a value of the correct type for field f of this object. Evaluating the update expression itself does not immediately change the state, but simply generates a record of the update for later application to the state. Evaluating update expressions in parallel produces the union of the updates they generate. For a sequential composition e1;e2, the updates generated by e1 are temporarily applied to the state when generating updates from e2—i.e. sequential execution does 'update composition'.

Our interpreter is essentially an operational semantics of ASL programmed in Intel's *reFLect* [16] functional language. For example, the ASL expression above translates into the *reFLect* code below:

```
let A = CLASSID "A";
let x = FIELDID "x";
asl [class A x] (dot (new A(lit 7)) x);
```

Each of the functions asl, class, dot, new and lit corresponds to one of the syntactic categories of ASL abstract syntax, and computes the changes to state and updates expected by the operational semantics [10].

To give a flavour of the technical details of our semantics, we sketch the definitions of object allocation and field update. We first introduce the *reFLect* data types that represent identifiers:

```
lettype classid = CLASSID string;
lettype objectid = OBJECTID string;
lettype fieldid = FIELDID string;
```

Identifiers are just tagged strings.

Boolean and integer literals are inherited from built-in *reFLect* types. The literal VOID stands for the element of a certain singleton type used for expressions that do not return some other value. Values are either literals or object identifiers:

```
lettype lit = VOID | BOOL bool | INT int;
lettype value = OBJ objectid | LIT lit;
```

The state of an ASL program, called a store, is represented by a content map (cmap) from value-holding *locations* (loc) to values, together with an update set (uset). A location is uniquely identified by an object identifier paired with a field identifier. An update is just a location paired with a value:

```
type loc = objectid × fieldid;
type cmap = loc → value;
type update = loc × value;
type store = cmap × update set;
```

We can now represent ASL expressions as *reFLect* functions that map a declaration context, represented by type dcxt, and a current store to a new store together with the resulting value:

```
type exp = dcxt → store → (store × value);
```

The declaration context, details of which are unimportant here, is just a static table of class information.

We can now express the semantics of new as follows:

```
let new {c::classid} {e::exp} {d::dcxt} {s::store} =
    let o = freshid c in
    val ((cm,us),v) = e d s in
    val f = lookup d c in
        (((update (o,f) v cm),us),OBJ o);
```

The function new takes a class identifier c and an expression e and returns a function from the declaration context and store to a new store and a value. The function freshid allocates a new object identifier o. The expression e is then evaluated to produce a new store, and the single field identifier for class c is obtained from the declaration context. (Our system in fact supports multiple fields.) Finally, the function returns a new state consisting of the updated content map and the unchanged update set, together with the allocated object identifier o as the resulting value.

The definition of the semantics of updates is equally straightforward. An update e1.f := e2 has semantics assign e1 f e2, where

```
let assign {e1::exp} {f:fieldid} {e2::exp}
           {d::dcxt} {s::store} =
    val ((cm1,us1),(OBJ o)) = e1 d s in
    val ((cm2,us2),v) = e2 d s in
        ((cm1 ∪ cm2, {(o,f),v} ∪ (us1 ∪ us2)),
         VOID);
```

The resulting content map is just the union of content maps arising from evaluation of e1 and e2, which may of course have allocated objects. The resulting update map contains the new update. The return value is just VOID.

VI. Related Work

Research addressing the challenges of functional validation using high level models has been active for decades, in both academia and industry. SystemC [2], designed for high level modelling of systems, extends C/C++ standards with features needed for hardware design and verification. The semantics of SystemC are, however, not suitable for formal analysis [17], because its design was not driven by semantic clarity, and it has evolved from other languages that either were not fully semantically defined or not well suited to hardware modelling.

TLA [5], the Temporal Logic of Actions by Leslie Lamport, does have a precise semantics suitable for formal verification of small to medium size problems. Models in TLA are not aimed for dynamic simulation or creation of test bench environment, limiting TLA to formal verification only. Formal analysis in TLA is based on explicit model checking, which of course suffers the state explosion problem. In Murphy [18] the basic concepts are similar to ASL, but its semantics does not allow native sequential composition of rules, which we view as helpful for modelling hardware. Like TLA, it has an explicit state model checker for formal analysis only. Esterel [3] is an evolving language and system used mostly to model control-oriented reactive systems, and so it less suitable for ALM.

Bluespec [4] is a language with a term rewriting semantics aimed at capturing model behaviour and synthesizing it to hardware design. Bluespec is similar in many ways to ASL. But the focus in Bluespec has very much been automatic scheduling and high level synthesis to RTL, while we are interested in modelling and verification methods. Bluespec took recently a new direction into the RTL domain by aiming to be interoperable with SystemVerilog [19].

Intel's Forte [13] is a powerful symbolic simulation system, but is targeted at gate level designs. Theorem provers (e.g. HOL [20]) are dedicated to interactive proof development and are (with effort) scalable, but the models in such systems are not executable. Symbolic execution with term rewriting similar to what we propose for ASL was shown to be an effective combination for verifying sequential programs in C. ASL, by contrast, supports the interleaved sequential and parallel execution we believe is essential for hardware modelling.

VII. Prospects

The ASL work presented here represents some significant first steps in a long-term research on algorithm level modelling. Using AsmL as a starting point, we have designed and implemented a protoype ASL environment that supports native symbolic execution and a rich collection of language constructs. We have exercised our system, and driven its design through a series of case studies, including the μop scheduler, a model of the AMBA protocol [21], and Lamport's bakery mutex protocol. We are planning to scale up our case studies and focus our efforts on methodology for ALM refinement down to design models. Tuning the performance and enhancing the capacity of our system is work in progress. We also intend to connect ALM models specified in ASL to design models written in SystemVerilog or SystemC, providing a path to downstream RTL design and validation flows.

Acknowledgments

We are grateful to the Microsoft Research team, including Yuri Gurevich and Wolfram Schulte, for their initial support in AsmL and Spec Explorer.

References

[1] B. Bentley, "Validating a modern microprocessor," in *Computer Aided Verification: 17th International Conference: Proceedings*, ser. LNCS, vol. 3576. Springer-Verlag, 2005.
[2] OSCI, "IEEE - the open systemc initiative," 2008. [Online]. Available: http://www.systemc.org/downloads/lrm
[3] G. Berry, P. Couronn, and G. Gonhier, "Synchronous programming of reactive systems: an introduction to ESTEREL," *INRIA report 647 (987)*.
[4] J. C. Hoe and Arvind, "Synthesis of operation-centric hardware descriptions." in *ICCAD*, 2000, pp. 511–518.
[5] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
[6] M. Y. Vardi, "Formal techniques for SystemC verification," in *Design Automation Conference*. ACM, 2007, pp. 188–192.
[7] D. Kroening and O. Strichman, *Decision Procedures: An Algorithmic Point of View*. Springer-Verlag, 2008.
[8] "AsmL: The abstract state machine language," 2006.
[9] AsmL. [Online]. Available: http://www.codeplex.com/AsmL/
[10] Y. Gurevich, B. Rossman, and W. Schulte, "Semantic essence of AsmL." *Theoretical Computer Science*, vol. 343, no. 3, pp. 370–412, 2005.
[11] J. Yang and C.-J. H. Seger, "Compositional specification and verification in GSTE," in *16th International Conference on Computer Aided Verification (CAV)*, ser. LNCS, R. Alur and D. A. Peled, Eds., vol. 3114. Springer-Verlag, 2004, pp. 216–228.
[12] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.
[13] C.-J. H. Seger, R. B. Jones, J. W. O'Leary, T. Melham, M. D. Aagaard, C. Barrett, and D. Syme, "An industrially effective environment for formal hardware verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 9, pp. 1381–1405, September 2005.
[14] A. K. Martin, "Bridging the gap between abstract RTL and bit-level designs," in *Seventh International Workshop on Designing Correct Circuits: Participants' Proceedings*, G. J. Pace and S. Singh, Eds. ETAPS 2008, March 2008, p. 72.
[15] C. Seger, "The design of a floating point unit using the integrated design and verification (IDV) system," in *DCC'06: Participants' Proceedings*, M. Sheeran and T. Melham, Eds., March 2006.
[16] J. Grundy, T. Melham, and J. O'Leary, "A reflective functional language for hardware design and theorem proving," *Journal of Functional Programming*, vol. 16, no. 2, pp. 157–196, March 2006.
[17] Y. Mahajan, C. Chan, A. Bayazit, S. Malik, and W. Qin, "Verification driven formal architecture and microarchitecture modeling," in *MEMOCODE'07*, April 2007.
[18] C. W. Murphy, "An overview of the murphy model," *Australian Economic Papers*, vol. 27, no. 0, pp. 175–99, Supplemen 1988. [Online]. Available: http://ideas.repec.org/a/bla/ausecp/v27y1988i0p175-99.html
[19] S. D. Stuart Sutherland and P. Flake, *SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling*. Kluwer Academic Publishers.
[20] M. J. C. Gordon and T. F. Melham, Eds., *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
[21] P. Böhm and T. Melham, "A refinement approach to design and verification of on-chip communication protocols," in *2008 Formal Methods in Computer Aided Design*, A. Cimatti and R. B. Jones, Eds. IEEE, 2008, pp. 136–143.