

Constraints Specification at Higher Levels of Abstraction

Felice Balarin, Jerry Burch, Luciano Lavagno, Yosinori Watanabe
Cadence Berkeley Laboratories
Berkeley, CA

Roberto Passerone, Alberto Sangiovanni-Vincentelli
Department of EECS
University of California at Berkeley
Berkeley, CA

Abstract

We are proposing a formalism to express performance constraints at a high level of abstraction. The formalism allows specifying design performance constraints even before all low level details necessary to evaluate them are known. It is based on a solid mathematical foundation, to remove any ambiguity in its interpretation, and yet it allows quite simple and natural specification of many typical constraints. Once the design details are known, the satisfaction of constraints can be checked either by simulation, or by formal techniques like theorem proving, and, in some cases, by automatic model checking.

1 Introduction

The value of any design can be divided into three broad categories: functionality, performance, and cost. All three are equally important, and a design can be successful only if it meets its goals in all three categories. However, the design process usually starts only with functional specification, because it is often not possible to evaluate performance and cost until much later in the design cycle. In other words, at higher levels of abstractions, performance and cost are constraints that have to be met, but they depend on quantities that will be defined only at lower levels of abstractions, when many implementation details are known. For example, an engine control system needs to meet hard real-time constraints, but its initial specification is typically in an untimed programming language.

We believe it is important that system specifications include performance and cost constraints starting at the highest level of abstraction, even if they are expressed in terms of yet undefined quantities. In this way, global constraints can be propagated to subsystems, possibly based on rough performance estimates. Inconsistent, or overly aggressive constraints can be recognized early, and appropriate adjustments can be made (e.g. relaxing some constraints, or adopting alternative designs). To gain wide acceptance, a constraint specification formalism must accomplish the following (somewhat conflicting) goals:

- it must be based on a solid mathematical foundation, to remove any ambiguity in its interpretation,
- it must feel natural to the designer, so that typical constraints are easy to specify,
- it must be compatible with existing functional specification formalisms, so that language extensions for constraint specification can be easily defined,
- it must be powerful enough to express a wide range of interesting constraints,

- it must be simple enough to be analyzable, at least by simulation, and ideally by automatic formal techniques.

In this paper, we propose a constraint specification formalism that addresses all of these goals. In case of conflicting goals (e.g. expressiveness vs. simplicity), we believe that the proposed formalism presents a reasonable compromise.

1.1 Related work

Constraint definition is central to many methodologies. A general approach is taken by the Rosetta [1] language: different domains of computation are described declaratively and constraints can be expressed as predicates on some defined quantities. Constraints are then applied by combining the different domains. In our work we restrict the scope of constraints definition in favor of a representation that is more natural to the designer and that is more computationally tractable.

A more restricted approach is taken in Object Constraint Language (OCL) [8], part of the Unified Modeling Language (UML). OCL supports invariants, pre- and post-conditions, and guards, applied to classes, operations of classes, and states respectively. Another related proposal is the Design Constraints Description Language (DCDL) [4] sponsored by Accellera, which is intended mostly for low-level (i.e. chip-level) constraints like clock slew, operating voltages and port capacitances. In both of these approaches, constraints are specified for a collection of entities that represent a system (classes and their operations and states in case of OCL, and physical objects in case of DCDL). This facilitates specifying constraints associated with the system as a whole, e.g. area, yield, testability, time to market. In contrast, we focus on specifying constraints for particular executions of the system, like response time, energy consumption and memory usage. OCL also supports this, to some extent, through pre-conditions, post-conditions, and guards. However, while these constructs naturally express constraints on a single transition, our approach makes it easy to express constraints that span several transitions. In fact, in our approach, it is easy to specify properties for which it is impossible to bound in advance the number of transitions needed to check them.

In the rest of this paper, we first present our constraint specification formalism in Section 2. We show how the proposed formalism can express some common constraints in Section 3. Simulation and formal verification of constraints are discussed in Section 4. Finally, in Section 5, we make conclusions and give some directions for future research.

2 Formalization

We aim at a constraint specification formalism that is compatible with a wide range of functional specification formalisms. Many of

these describe a system as a network of components communicating through fixed interconnections. The observed behavior of the system is usually characterized by sequences of values observed at the interconnections. In the rest of this section, we first define formal structures intended to model these sequences, and then propose the syntax and the semantics of the logic for specifying constraints over these structures.

2.1 Representing system behaviors

We use the term *behavior* to denote the sequence of inputs and outputs that a system exhibits when excited by the input sequence. Formally, let E be a set of *event names*¹ and for each $e \in E$ let $V(e)$ be its *value domain*. Then, a *behavior* is a mapping $\beta : E \times \mathbb{N} \mapsto \bigcup_{e \in E} V(e)$ such that $\beta(e, n) \in V(e)$ for each $e \in E$, and each positive integer $n \in \mathbb{N}$. A *system* is specified by a set of event names, their value domains and a *set of behaviors*.

In a typical system, event names may represent interconnections, e.g. wires in a hardware system, or mailboxes in a software system. The behavior of the system is then characterized by sequences of values on wires, or sequences of messages to mailboxes.

Behaviors by themselves are not sufficient to evaluate constraints. For this, we need additional information regarding performance measures. We represent this information as annotations to behaviors. Formally, given an arbitrary set T , a T -valued *annotation* is a function from $E \times \mathbb{N}$ to T . Similarly to events, if f is a T -valued annotation, then we say that T is the value domain of f . An *annotated behavior* is a pair (β, A) where β is a behavior and A is a set of annotations.

In this paper we show a few uses of annotations, but make no proposal for their specification. We assume that they are part of functional specification, and thus specified with the same language as functional specification. In a way, they are an extension of an already common design practice, where comments and assertions are placed in the code to ease design understanding and debugging.

Annotated behaviors are structures for which we want to state constraints. We express these constraints in a subset of first-order logic called the *logic of constraints*, or LOC for short. In other words, annotated behaviors are models of LOC formulas.

2.2 LOC syntax

LOC formulas are defined relative to a multi-sorted algebra $(\mathcal{A}, \mathcal{O}, \mathcal{R})$, where \mathcal{A} is a set of sets (sorts), \mathcal{O} is a set of operators, and \mathcal{R} is a set of relations on sets in \mathcal{A} . More precisely, elements of \mathcal{O} are functions of the form $T_1 \times \dots \times T_n \mapsto T_{n+1}$, where n is a natural number, and T_1, \dots, T_{n+1} are (not necessarily distinct) elements of \mathcal{A} . If $o \in \mathcal{O}$ is such a function, then we say that o is n -ary and T_{n+1} -valued. Similarly, an n -ary relation in \mathcal{R} is a function of the form $T_1 \times \dots \times T_n \mapsto \{true, false\}$. We require that \mathcal{A} contains at least the set \mathbb{N} of natural numbers, and the value domains of all event names and annotations appearing in the formula. For example, if \mathcal{A} contains integers and reals, \mathcal{O} could contain standard addition and multiplication, and \mathcal{R} could contain usual relational operators ($=, <, >, \dots$).

LOC formulas may contain only one variable, namely i . The domain of i is \mathbb{N} . Having only one variable may seem very restrictive, but so far we have not found a natural constraint that required more than one. In effect, the ability of defining annotations allows one to specify formulas that otherwise require more than one variable, as we shall see later with examples. The advantages of a single variable are simpler syntax (fewer names), and simpler simulation monitoring.

¹In this paper, we assume that E is finite. However, the approach presented here could easily be extended to arbitrary sets of event names. This extension would allow us to consider networks with dynamic process and interconnection creation.

The basic building blocks of LOC formulas are *terms*. We distinguish terms by their value domains:

- i is an \mathbb{N} -valued term,
- for each value domain $T \in \mathcal{A}$, and each $c \in T$, c is a T -valued term,
- if τ is an \mathbb{N} -valued term, $e \in E$ is an event name, and f is a T -valued annotation, then $\text{val}(e[\tau])$ is a $V(e)$ -valued term, and $f(e[\tau])$ is a T -valued term,²
- if $o \in \mathcal{O}$ is a T -valued n -ary operator, and τ_1, \dots, τ_n are appropriately valued terms, then $o(\tau_1, \dots, \tau_n)$ is a T -valued term.

Terms are used to build *LOC formulas* in the standard way:

- if $r \in \mathcal{R}$ is an n -ary relation, and τ_1, \dots, τ_n are appropriately valued terms, then $r(\tau_1, \dots, \tau_n)$ is an LOC formula,
- if ϕ and ψ are LOC formulas, so are $\bar{\phi}$, $\phi \wedge \psi$, and $\phi \vee \psi$.

For example, if a and b are names of \mathbb{N} -valued events, and f and g are \mathbb{N} -valued annotations, then the set of LOC formulas includes the following:

$$\begin{aligned} \text{val}(a[i]) = 5 \wedge \text{val}(a[i+1]) = 5 \\ f(a[i+4]) + f(b[g(a[i])]) < 20 \\ \overline{\text{val}(a[i]) = 0} \vee f(b[i]) = 0 \end{aligned}$$

When reading these formulas, it is helpful to think of i as being universally quantified, as clarified in the LOC semantics next.

2.3 LOC semantics

Informally, LOC formulas are evaluated at annotated behavior (β, A) as follows:

- the variable i evaluates to any positive integer,
- if τ evaluates to some positive integer n , then $e[\tau]$ evaluates to $\beta(e, n)$, and $f(e[\tau])$ evaluates to $f(e, n)$,
- all other operators, relations and Boolean functions are evaluated in the standard way, and
- an annotated behavior *satisfies* an LOC formula if the formula evaluates to *true* for all possible values of i .

More formally, we first define the *value* of formulas and terms with respect to an annotated behavior, and a value of variable i . We use $\mathcal{V}_{(\beta, A)}^n \llbracket \alpha \rrbracket$, where α is a term or a formula, to denote the value of α evaluated at the annotated behavior (β, A) and the value n of variable i . The value is defined recursively as follows:

- $\mathcal{V}_{(\beta, A)}^n \llbracket i \rrbracket = n$,
- $\mathcal{V}_{(\beta, A)}^n \llbracket c \rrbracket = c$ for each element c of each value domain T ,
- $\mathcal{V}_{(\beta, A)}^n \llbracket \text{val}(e[\tau]) \rrbracket = \beta(e, \mathcal{V}_{(\beta, A)}^n \llbracket \tau \rrbracket)$ for each event name e and each \mathbb{N} -valued term τ ,
- $\mathcal{V}_{(\beta, A)}^n \llbracket f(e[\tau]) \rrbracket = f(e, \mathcal{V}_{(\beta, A)}^n \llbracket \tau \rrbracket)$ for each annotation $f \in A$, each event name e , and each \mathbb{N} -valued term τ ,
- $\mathcal{V}_{(\beta, A)}^n \llbracket o(\tau_1, \dots, \tau_k) \rrbracket = o(\mathcal{V}_{(\beta, A)}^n \llbracket \tau_1 \rrbracket, \dots, \mathcal{V}_{(\beta, A)}^n \llbracket \tau_k \rrbracket)$ for each k -ary operator o ,

²It may appear that expression $f(e[\tau])$ is in conflict with the definition of a T -valued annotation as a function from $E \times \mathbb{N}$ to T . However, when we define the semantics of $f(e[\tau])$ it will become clear that there is no conflict.

- $\mathcal{V}_{(\beta,A)}^n \llbracket r(\tau_1, \dots, \tau_k) \rrbracket = r(\mathcal{V}_{(\beta,A)}^n \llbracket \tau_1 \rrbracket, \dots, \mathcal{V}_{(\beta,A)}^n \llbracket \tau_k \rrbracket)$ for each k -ary relation r ,
- $\mathcal{V}_{(\beta,A)}^n \llbracket \bar{\phi} \rrbracket = true$ if and only if $\mathcal{V}_{(\beta,A)}^n \llbracket \phi \rrbracket = false$,
- $\mathcal{V}_{(\beta,A)}^n \llbracket \phi \wedge \psi \rrbracket = true$ if and only if $\mathcal{V}_{(\beta,A)}^n \llbracket \phi \rrbracket = true$ and $\mathcal{V}_{(\beta,A)}^n \llbracket \psi \rrbracket = true$,
- $\mathcal{V}_{(\beta,A)}^n \llbracket \phi \vee \psi \rrbracket = true$ if and only if $\mathcal{V}_{(\beta,A)}^n \llbracket \phi \rrbracket = true$ or $\mathcal{V}_{(\beta,A)}^n \llbracket \psi \rrbracket = true$.

We say that an annotated behavior (β, A) satisfies a formula ϕ , if $\mathcal{V}_{(\beta,A)}^n \llbracket \phi \rrbracket = true$ for all $n \in \mathbb{N}$.

3 Examples of constraints expressed in LOC

In the following examples, we assume that the set of event names is $E = \{in, out\}$, and that a real-valued annotation t is defined. Intuitively, we assume that $t(e[i])$ corresponds to the time of the i -th occurrence of an event e . The following common constraints are now easy to express:

rate: “a new *out* will be produced every P time units”:

$$t(out[i+1]) - t(out[i]) = P ,$$

latency: “*out* is generated no more than L time units after *in*”:

$$t(out[i]) - t(in[i]) \leq L ,$$

jitter: “every *out* is no more than J time units away from the corresponding tick of the real-time clock with period P ”:

$$|t(out[i]) - i * P| \leq J ,$$

throughput: “at least X *out* events will be produced in any period of T time units”:

$$t(out[i+X]) - t(out[i]) \leq T ,$$

burstiness: “no more than X *in* events will arrive in any period of T time units”:

$$t(in[i+X]) - t(in[i]) > T .$$

Consider the latency constraint above. It is valid assuming that *in* and *out* are kept synchronized, i.e. the i -th occurrence of *in* causes the i -th occurrence of *out*. However, this is not true for all systems. In some systems, input events may be lost or intentionally ignored. In these cases, it is not obvious which occurrences of *in* and *out* should be compared for latency. Annotations provide a convenient way for a designer to add this information to the system specification. For example, assume that an \mathbb{N} -valued annotation named *cause* is defined for each *out* event, and that $cause(out[i])$ represents the index of the occurrence of *in* which caused $out[i]$. In this case, the latency constraint can be specified as follows:

$$t(out[i]) - t(in[cause(out[i])]) \leq L . \quad (1)$$

This example illustrates how annotations can be used to augment the design with information which is not strictly necessary for its functionality, but which helps in understanding design intentions. In general, such information could be also used by formal techniques to help prove (or disprove) the correctness of the design.

This example also illustrates how annotations can sometimes replace additional variables. Assume that *out* should be compared

with the most recent *in* for latency. If LOC allowed variable j in addition to i , then this property could be expressed by:

$$(t(in[j]) < t(out[i]) \wedge t(out[i]) \leq t(in[j+1])) \implies (t(out[i]) - t(in[j]) \leq L) . \quad (2)$$

If *cause* is properly defined, then (1) and (2) are equivalent. We can check that *cause* is properly defined by verifying the following LOC formula:

$$t(in[cause(out[i]) < t(out[i]) \wedge t(out[i]) \leq t(in[cause(out[i]) + 1]) .$$

4 Verifying LOC constraints

At the beginning of a design cycle most annotations will likely not be defined, and LOC formula cannot be evaluated. Later on, as design details are filled in, annotations will become defined and constraints can be verified. In general, this can be done either informally by simulation, or by formal techniques like theorem proving or automatic model checking.

4.1 Simulation

Annotated behaviors are intended to be a formalization of simulation traces. However, annotated behaviors are infinite structures ($\beta(e, n)$ is defined for all positive integers n), while in reality, any simulation trace is only a final prefix of a conceptually infinite trace. By letting a simulation run longer, the trace can be extended, but it can never be completed. To check that a formula is satisfied by a trace, one needs to evaluate it for all possible values of variable i . Unfortunately, given only a finite prefix one can typically evaluate a formula only for some values of i . If any of these evaluations produces *false*, we can conclude that the formula is not satisfied. However, we can never conclude that an arbitrary formula is satisfied just from a finite simulation trace.

A simple algorithm that checks whether a given simulation trace satisfies a given LOC formula is:

1. Let the current value of i be 0.
2. Try to evaluate the formula for the current value of i .
3. If the formula evaluates to *true*, or if the given trace is too short to evaluate the formula for the current value of i , then increase i by one and go to step 2. Otherwise (i.e. if the formula evaluates to *false*), report that the formula is not satisfied and stop.

This algorithm will terminate only if the formula is not satisfied. To ensure that the algorithm terminate, we need to enable it to either:

- conclude that the formula is satisfied for all values of i , or
- conclude that the trace is not long enough to evaluate the formula for any additional values of i .

Both of this analyses can be automated, but, unfortunately, they require complex capabilities typically found in theorem proving tools.

For example, the formula:

$$i > 5 \vee \text{val}(a[i]) = 1$$

obviously evaluates to *true* for all values of i greater than 5, so if it is not violated by a finite trace with at least five occurrences of a , we can conclude that it is satisfied by all infinite extensions of that finite trace.

Similarly, given a trace in which event a occurs 18 times, the formula

$$\text{val}(a[i^3 - 10i^2 + 26]) = 1$$

can be evaluated only for five values of i (1, 3, 4, 5, 6). Unfortunately, finding this fact requires solving the following non-linear integer inequality:

$$i^3 - 10i^2 + 26 \leq 18 .$$

In any case, incomplete verification results are intrinsic to simulation. Systems can typically exhibit infinitely many different behaviors (prompted by different input stimuli), only few of which can be simulated with finite resources. Nevertheless, simulation is still the cornerstone of all practical verification methodologies. Therefore, it is reasonable to expect that checking LOC constraints by simulation will be valuable in increasing confidence in design performance.

4.2 Theorem proving

LOC is just a subset for first order logic, so any theorem prover that can handle first order logic can also handle LOC, e.g. [3]. If a system behavior is also represented inside the prover, then satisfaction of LOC constraints can be formulated as theorems, and checked by the tool. How to represent system behaviors within a theorem prover is beyond the scope of this paper, but it is a topic that has attracted a lot of interest in the research community (e.g. [6]).

Even without the system specification, theorem provers can be useful in analyzing constraints. Consider for example the latency constraint:

$$t(\text{out}[i]) - t(\text{in}[i]) \leq 10 .$$

If we architect the system such that it consists of two subsystems, the first one of which takes in as input and produces an intermediate event x , and the second takes x as input and produces out , we may decide to decompose the global latency constraint into the following two local constraints:

$$\begin{aligned} t(\text{out}[i]) - t(x[i]) &\leq 5 \\ t(x[i]) - t(\text{in}[i]) &\leq 5 , \end{aligned}$$

and proceed with the design of subsystems based on the local constraints. Theorem provers can help ensure that local constraints indeed imply the global constraint, even before the design of subsystems is done.

4.3 Model Checking

In order to use automatic model checking [7, 5] to verify LOC constraints, the following needs to be done:

1. system behaviors (including annotations) need to be represented as a finite-state automaton over some finite alphabet,
2. an LOC formula needs to be converted into a temporal logic formula over the same alphabet (or, equivalently, into a finite state automaton over the same alphabet).

Both of these steps are non-trivial, and each one may be impossible. Both are clearly impossible if value domains of some events or annotations are infinite. But even if all value domains are finite, problems may arise.

Consider the system described by the pseudo code in Figure 1. It has three event names (input , a , b), all with the value domain $\{\text{T}, \text{F}\}$. It has many behaviors, determined by different input streams. One of its behavior is shown in Figure 2.

Note that the behaviors are well defined only if the input event takes both value T and F infinitely often. Otherwise, there are only

bool a_val, b_val

```

 $a\_val := b\_val := \text{T}$ 
repeat {
  await ( new  $\text{input}$  event )
  if ( value of  $\text{input}$  event is T )
    emit event  $a$  with value  $a\_val$ 
     $a\_val := \overline{a\_val}$ 
  } else {
    emit event  $b$  with value  $b\_val$ 
     $b\_val := \overline{b\_val}$ 
  }
}

```

Figure 1: A simple system.

| β | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |
|----------------|---|---|---|-----|---|-----|---|---|-----|
| input | T | F | F | F | T | F | F | T | ... |
| a | T | F | T | ... | | | | | |
| b | T | F | F | F | T | ... | | | |

Figure 2: A behavior of the system in Figure 1.

finitely many occurrences of a or b , and our definition of a behaviors requires that each event occurs infinitely many times. Not being able to consider events which occur only finitely many time might be considered a shortcoming of our approach. This shortcoming could be fixed by allowing behaviors to contain finite sequences for some events. While this modification is conceptually simple, it does introduce significant technical difficulty. In particular, LOC syntax needs to deal with the case when terms like $\text{val}(e[i])$ are not defined. To avoid these tedious details, we have decided to stick with infinite behaviors in this paper.

Getting back to the example in Figure 1, if we want to represent it as a finite-state automaton, we first need to choose an alphabet. It may seem that $\{\text{T}, \text{F}\}^3$ is a good choice. In this alphabet, the behavior in Figure 2 would be represented as the sequence:

$(\text{T}, \text{T}, \text{T}), (\text{F}, \text{F}, \text{F}), (\text{F}, \text{T}, \text{F}), \dots$

Unfortunately, in this representation, the system cannot be modeled with a finite number of states, because it needs to make and remember a potentially unbounded number of guesses about future input values in order to determine the value of b in the current “step”.

A better choice of alphabet is $\{\perp, \text{T}, \text{F}\}^3$, where \perp indicates that the corresponding event is not emitted in that step. In this alphabet, the behavior in Figure 2 would be represented as the following sequence:

$(\text{T}, \text{T}, \perp), (\text{F}, \perp, \text{T}), (\text{F}, \perp, \text{F}), (\text{F}, \perp, \text{F}), (\text{T}, \text{F}, \perp), \dots$

A finite state representation of the system is shown in Figure 3.

The choice of alphabet also affects how LOC formulas are translated into automata. For example, in the alphabet $\{\text{T}, \text{F}\}^3$, the formula

$$a[i] = b[i]$$

is represented by a trivial one-state automaton. However, the same formula in the alphabet $\{\perp, \text{T}, \text{F}\}^3$ requires infinitely many states, because i -th occurrences of a and b can be separated by arbitrary many steps. On the other hand, some formulas are represented by the same automaton in both alphabets. An example is the constraint:

$$a[i + 2] = a[i]$$

which states that a is periodic with period 2. The automaton for this formula is shown in Figure 4. Automatic model checking tools

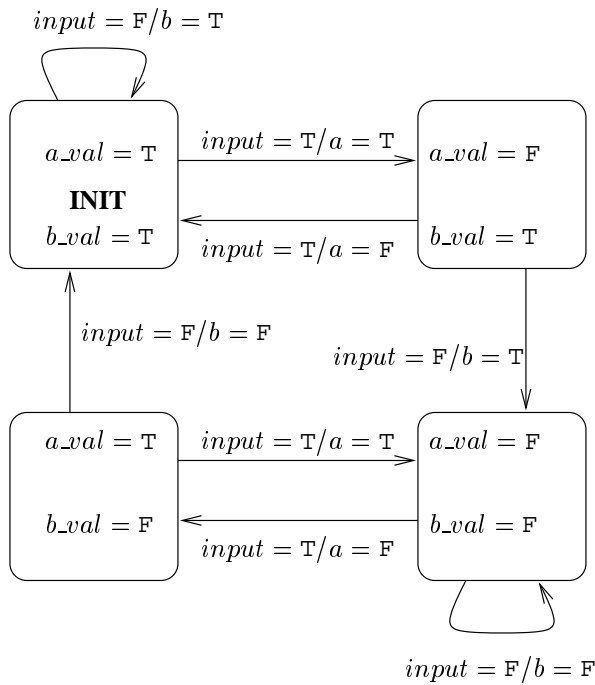


Figure 3: A finite state automaton for the system in Figure 1.

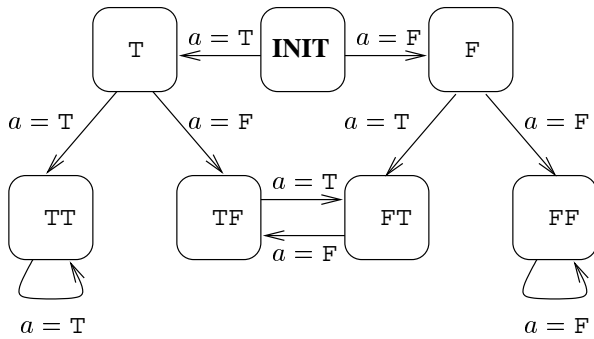


Figure 4: A finite state automaton for the constraint $a[i + 2] = a[i]$.

can easily check that all sequences generated by the automaton in Figure 3 are accepted by the automaton in Figure 4, proving that a is indeed periodic with period 2.

5 Conclusions

The logic of constraints is a simple subset of first order logic that is still sufficient to express many interesting constraints. It is interpreted over structures that closely resemble simulation traces. Therefore, it can be analyzed both by formal techniques and simulation. We intend to use the LOC as a part of a comprehensive system-level design environment called Metropolis, a successor to POLIS design environment [2].

References

- [1] P. Alexander, C. Kong, and D. Barton. Rosetta usage guide. available at <http://www.sldl.org>, 2001.
- [2] Felice Balarin et al. *Hardware-software co-design of embedded*

systems: the POLIS approach. Kluwer Academic Publishers, 1997.

- [3] R.S. Boyer, M. Kaufmann, and J.S. Moore. The boyer-moore theorem prover and its interactive enhancement. *Computers & Mathematics with Applications*, pages 27–62, January 1995.
- [4] Quick reference guide for the Design Constraints Description Language (DCDL). available at <http://www.eda.org/dcwg>, 2000.
- [5] Z. Har’El and R. P. Kurshan. Software for analysis of coordination. In *Proceedings of the International Conference on System Science*, pages 382–385, 1988.
- [6] Warren A. Hunt Jr. and Steven D. Johnson, editors. *Formal methods in computer-aided design: third international conference FMCAD 2000*. Springer-Verlag, 2000. LNCS vol. 1954.
- [7] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
- [8] Object Constraint Language specification. available at <http://www.omg.org>, 1997.