

Logic of Constraints: A Quantitative Performance and Functional Constraint Formalism

Xi Chen, Harry Hsieh, Felice Balarin, *Member, IEEE*, and Yosinori Watanabe, *Member, IEEE*

Abstract—In the era of billion-transistor design, it is critical to establish effective verification methodologies from the system level, all the way down to the implementations. In this paper, we introduce logic of constraints (LOC), a logic that is particularly suited to express quantitative performance constraints as well as functional constraints. We analyze the expressiveness of LOC and show that it is important and different from linear temporal logic, upon which traditional hardware assertion languages (e.g., PSL and OpenVera) are based. We propose an automatic simulation trace checking/runtime monitoring methodology that can be used to verify system designs very efficiently. Since a subset of LOC is decidable, we also discuss the formal verification approach for LOC formulas. Through several industrial case studies, we demonstrate the usefulness of the LOC formalism and the corresponding simulation and verification approach at the higher transaction level of abstraction.

Index Terms—Logic of constraints (LOC), performance constraint, simulation checker, trace analysis.

I. INTRODUCTION AND RELATED WORK

THE INCREASING complexity of embedded systems today demands more sophisticated design and verification methodologies. Systems are becoming more integrated as more and more functionality and features are required for the product to succeed in the market. Embedded-system architectures likewise have become more heterogeneous as it is becoming more economically feasible to have various computational resources (e.g., microprocessor, digital signal processor, reconfigurable logic) all utilized on a single chip. Designing at the register transfer level [1] or sequential C-code level is no longer efficient. More than ever, design and verification methodologies at higher levels of abstraction are required to minimize the design cost of an electronic product. The specification of the function and the architecture should be done at a high level of abstraction, and the design procedures should refine the abstract function, refine the abstract architecture, and map the function onto the architecture through automatic tools or manual means with tool support [2], [3]. High-level design procedures allow designers to tailor their architectures to the functions at hand, or to modify their functions to suit the available architectures (see Fig. 1).

Manuscript received July 13, 2003; revised November 26, 2003. This paper was recommended by Associate Editor J. H. Kukula.

X. Chen and H. Hsieh are with the University of California, Riverside, CA 92521 USA (e-mail: xichen@cs.ucr.edu; harry@cs.ucr.edu).

F. Balarin and Y. Watanabe are with Cadence Berkeley Laboratories, Berkeley, CA 94704 USA (e-mail: felice@cadence.com; watanabe@cadence.com).

Digital Object Identifier 10.1109/TCAD.2004.831575

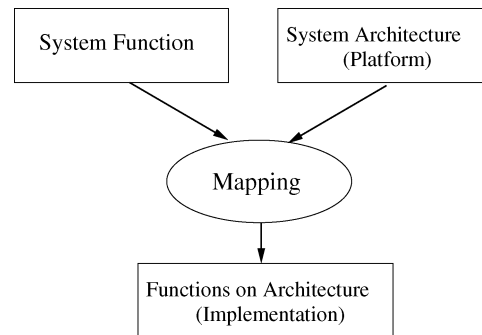


Fig. 1. System design methodology.

To make the practice of designing from high-level system specification a reality, verification methods must accompany every step in the design flow. Specification at the system level makes formal verification possible [4]. Designers can prove a property of a specification by writing down the property they want to check in some logic (e.g., linear temporal logic (LTL) [5]) and use a formal verification tool (e.g., the model checker SMV [6] and SPIN [7]) to run the verification. Formal verification checks the entire state space of a design to verify some specified property without any uncertainty. As the designs are refined, however, the complexity can quickly overwhelm the automatic tools, and simulation becomes the primary means for verification. The confidence of a simulation verification mainly depends on the design of test cases. Designers can insert embedded assertions into their hardware description language (HDL) descriptions to help uncover bugs of the designs during simulation. Today's embedded assertion languages capture those simple logics as language/platform-specific library blocks. A set of extended temporal logic is then used to operate on those blocks for expressing more complex assertions. Examples of assertion languages include PSL [8], SystemVerilog Assertions [9], and OpenVera [10].

We believe that the hardware assertion languages are not natural to express more abstract properties such as transaction-level properties, where only the events observable from the system and their annotations are considered. Nor are they convenient to directly express performance constraints that are quantitative in nature (e.g., latency, throughput). To this end, we propose a constraint formalism: logic of constraints (LOC).¹ LOC is particularly suited for specification and simulation analysis of performance constraints at the transaction level, as will be shown later in this paper. A constraint formalism is not meaningful unless there exists a clear and efficient path to verification. We propose an efficient simulation-based approach for an-

¹Preliminary studies of LOCs have been presented in [11]–[13].

alyzing LOC formulas. C++ trace checkers are automatically generated from LOC formulas. The checkers analyze the simulation traces and report any constraint violations. In most cases, the traces are scanned only once and memory usage is very low. The automatic checker generation is parameterized, so it can be customized for fast analysis for specific verification environments (e.g., memory limitation). The choice of C++ for the checkers is a matter of convenience. It allows us to tightly integrate the checkers with the SystemC [14] simulator for runtime monitoring. No major difficulty exists to generate checkers in HDLs for integration with hardware simulators, or in Java for concurrent execution with the software simulators. To illustrate the concept and demonstrate the usefulness of our approaches, we conduct two separate case studies: a high-level description of a *picture-in-picture (PIP)* design [15] and an register transfer level design of a *finite impulse response (FIR)* filter.

A simulation-based approach can only disprove the LOC formula (if a violation is found), but it cannot prove it conclusively. However, for small but important designs or library modules that will be instantiated many times across different designs, it may be feasible to formally prove the desired properties. Formal verification is more expensive though the designers can be more confident about the result. It should be used only for small but important design modules [e.g., task transaction level (TTL) [16] channel in Section VI-A], possibly in concert with simulation verification of the entire system. An exact-verification algorithm exists for a broad class of LOC formulas [17]. However, due to the high complexity of this algorithm, we provide an alternative in this paper. We propose a formal verification approach, where LOC formulas are translated into verification models in Promela (SPIN's input language [7]) and LTL formulas. This approach is complete for a restricted subset of LOC. It can also be applied to a wider subset, but results might then be inconclusive, i.e., the verification is only partial. We illustrate the concept and demonstrate the usefulness of our approach through a case study on the formal verification of the TTL channel library module used in the PIP design.

While similar in spirit to the hardware-embedded assertion languages, our LOC formalism provides a unique combination of at least three fundamental aspects. First, LOC is designed for specifying all quantitative performance and functional constraints, not just functional ones. This means that one can easily specify requirements on timing or power consumption of the systems being designed, in addition to those on the functional correctness. Second, LOC can specify some properties that cannot be expressed with existing hardware assertion languages, as shown in Section III-B. Third, system-level functional and performance constraints written in LOC can be automatically and efficiently synthesized into static checkers, runtime monitors, or formal verification modules.

A. Related Work

Constraint definition is central to many methodologies. A general approach is taken by the Rosetta [18] 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 constraint definition in favor of a representation that is more natural to the designer and that is more computationally tractable.

Object constraint language (OCL) [19], part of the unified modeling language (UML), takes a more restricted approach. OCL supports invariants, pre- and postconditions, and guards, applied to classes, operations of classes, and states, respectively. Another related proposal is the design constraints description language (DCDL) [20] 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 preconditions, postconditions, 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.

Many constraint formalisms have been proposed that are at most as expressive as ω -regular languages (and in some case strictly less expressive). An incomplete list includes S1S [21], LTL [5], PSL [8], HAAD [22], and many variants of finite-state automata on infinite words, e.g., [23], [24]. MONA [25], on the other hand, is based on regular languages and finite-state automata on finite words. We believe that LOC is a good complement to all these approach, as there are certain natural properties (e.g., *data consistency* in Section III) that are not ω -regular, but can be expressed and verified (both formally and by simulation) using LOC.

Real-time logic (RTL) [26] is a formalism for expressing timing properties in real-time systems. With RTL, the properties are specified by means of timing relations on occurrences of events. RTL was primarily intended for formal reasoning, while LOC is more biased toward simulation monitoring. For example, RTL allows any number of index and time variables which can be arbitrarily quantified. This makes it very unsuitable for verification by simulation. In contrast, LOC allows only one index variable and no time variables or quantification. We made this choice precisely for the purpose of efficient simulation monitoring. Also, arithmetic in RTL is limited to Presburger arithmetic (i.e., linear inequalities), to ease formal reasoning, while LOC allows more complex expressions, because they can be handled quite easily in simulation. This separation of purposes is not total. When we consider a subset of LOC suitable for formal verification, we restrict LOC to Presburger arithmetic. Similarly, Mok and Liu have proposed a subset of RTL suitable for simulation monitoring [27], [28], and that subset indeed resembles LOC. However, they have not proposed any automatic formal verification technique for that subset. A subset of LOC suitable for formal verification can

be seen as a generalization of the subset of RTL suitable for simulation monitoring, as it allows specification of properties related to annotations other than time. In fact, data consistency, one of the properties that distinguish LOC from formalisms based on ω -regular languages, also distinguishes it from RTL, as it does not involve time at all.

The rest of the paper is organized as follows. In the next section, we introduce the quantitative constraint formalism, LOC, and its typical usage. In Section III, we discuss the expressiveness of LOC, and show that LOC can be used to express important constraints that cannot be expressed with LTL for the specification of system designs. In Section IV, we present the methodology for building a trace checker or runtime monitor from any given LOC formula. We demonstrate the usefulness and efficiency of the approach with two case studies in Section V. In Section VI, we discuss the formal verification approach for LOC formulas. We present a verification example and show how the approach works on checking important library modules. In Section VII, we summarize the contributions of the paper and provide several future research directions. Finally, in Appendix I, we present the formal syntax and semantics of LOC.

II. LOC

In this section, we introduce our quantitative constraint formalism, LOC. The constraint-specification formalism is compatible with a wide range of functional specification formalisms that 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. LOC is a formalism designed to reason about traces from the execution of a system. It consists of all the terms and operators allowed in sentential logic, with additions that make it possible to specify system level quantitative functional and performance constraints without compromising the ease of analysis.

LOC can be used to specify very common and useful real-time performance constraints

- rate, e.g., “Displays are produced every ten time units”

$$t(\text{Display}[i + 1]) - t(\text{Display}[i]) = 10 \quad (1)$$

- latency, e.g., “Display is generated no more than 25 time units after Stimuli”

$$t(\text{Display}[i]) - t(\text{Stimuli}[i]) \leq 25 \quad (2)$$

- jitter, e.g., “every Display is no more than four time units away from the corresponding tick of the real-time clock with period 10”

$$|t(\text{Display}[i]) - (i + 1) * 10| \leq 4 \quad (3)$$

- throughput, e.g., “at least 100 Display events will be produced in any period of 1001 time units”

$$t(\text{Display}[i + 100]) - t(\text{Display}[i]) \leq 1001 \quad (4)$$

- burstiness, e.g., “no more than 1000 Display events will arrive in any period of 9999 time units”

$$t(\text{Display}[i + 1000]) - t(\text{Display}[i]) > 9999. \quad (5)$$

In addition, LOC can also be used to specify quantitative functional constraints such as the data consistency, e.g., “the input data should be the same as the output data”

$$\text{data}(\text{input}[i]) = \text{data}(\text{output}[i]). \quad (6)$$

It should be emphasized that time is only one of the possible annotations. Any value that may be associated with an event (e.g., power, area, data value) can be used as an annotation. In the case of concurrent events, the values of time annotation should be the same. The indices of instances of the same event denote the strict order as they appear in the execution trace. There is no implied relationship between instances of different events. LOC can be used to express relationship between the annotations of the different instances of the same event (e.g., rate), or instances of different events (e.g., latency).

The latency constraint above is truly a latency constraint only if the Stimuli and Display are kept synchronized. Generally, we will need an additional annotation that denotes which instance of Display is “caused” by which instance of the Stimuli. If the cause annotation is available, the latency constraint can be more accurately written as

$$t(\text{Display}[i]) - t(\text{Stimuli}[\text{cause}(\text{Display}[i])]) \leq 25 \quad (7)$$

and such an LOC formula can easily be analyzed through the simulation checker presented in Section IV. However, it is the responsibility of the designer, the program, or the simulator to generate such an annotation.

A. LOC Syntax and Semantics

Here, we give an informal overview of LOC syntax and semantics. Full details are given in Appendix I. The basic blocks of LOC formulas are terms, which can be either:

- constants;
- integer variable i (the only index variable that can appear in an LOC formula);
- expressions of the form $a(e[n])$, where a is an annotation name, e is an event name, and *index expression* n is an integer-valued term;
- combination of simpler terms using usual arithmetic operators.

We interpret $a(e[n])$ as the value of annotation a of the n th occurrence of event e . All other terms are interpreted naturally. Terms can be combined using relational operators to create atomic LOC formulas. Finally, LOC formulas are standard Boolean expressions over atomic formulas.

LOC formulas may contain only one index variable, namely i . Having only one index 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 index variable. On the other hand, having only one index variable enables efficient simulation monitoring.

Models of LOC formulas contain a sequence of occurrences for each event name in the formula. We call such structures *annotated behaviors*. Each occurrence may be annotated with some annotation, but we do not require each annotation appearing in the formula to be defined. This feature is important for our design methodology, where performance requirements are specified early in the process, even though they can be evaluated much later, when many implementation details are set.

Given an annotated behavior, we evaluate the formula for each value of index variable i . This is done in quite a standard fashion, except that we need to consider the fact that some terms may not be defined (either because there are only finitely many occurrences of an event, or because an annotation is not defined for an existing event occurrence). To deal with this, we introduce the third logical value *undef*. In general, all operators (including Boolean) return *undef* if one of their operands are *undef*. The only exceptions are conjunction with false (which is false), and disjunction with true (which is true). Finally, we say that the annotated behavior satisfies the formula if it does not evaluate to false for any value of i .

III. EXPRESSIVENESS OF LOC

In this section, we discuss the expressiveness property of LOC especially in its relationship with the well-known LTL. It should be noted that LTL is defined on the state transition level where any change at the system state is accounted for, while LOC works on a higher abstraction level, in which only the events observable from the system and their annotations are considered. This apparent difference, however, is just a technicality, because it is not difficult to hide state transitions so that LTL and LOC are defined over the same kind of objects.

A. LTL

LTL is defined over *executions* of a system, i.e., linear sequences of *state transitions*. LTL formulas are constructed using terms, i.e., Boolean expressions on variables or system states, classical Boolean operators such as \neg (not), \vee (or), \wedge (and), \rightarrow (imply), and the linear temporal operators G (always), F (eventually), X (next) and U (strong until). For example, $G(A)$ is true if A is true for any state, $F(A)$ is true if A eventually becomes true in a future state, $X(A)$ is true if A is true in the following state, and $A \cup B$ is true if B eventually becomes true in a future state and A is true from the current state to that future state.

It has been proven that LTL formulas can be translated to equivalent Büchi automata [29]. Based on this theory, formal techniques like model checking are developed and utilized for verification of both digital designs (e.g., SMV [6]) and software protocols (e.g., Spin [7]). LTL is also used as the basis for the formal property specification for simulation-based assertion verification [10], [30], which is important to assure the integration and correctness of reusable intellectual property (IP) blocks.

B. LOC Versus LTL

Through several examples and claims, we will conclude that LOC and LTL are incomparable and have different domains of expressiveness.

Claim 1: There are LOC formulas that can be expressed with LTL.

Since both LOC and LTL contain basic Boolean expressions, a subset of LOC constraints that specify simple global Boolean conditions can be expressed in LTL also. For example, the constraint, “the annotation data of the event Display is always greater than 100,” is expressed in LOC as

$$\text{data}(\text{Display}[i]) > 100. \quad (8)$$

If we use a variable *Display_data* to store the value of data in the design, and use a flag *Display_occur* to indicate that an instance of the event Display occurs, this constraint can be easily expressed in LTL as

$$G(\text{Display_occur} \implies (\text{Display_data} > 100)). \quad (9)$$

Claim 2: There are LOC formulas that cannot be expressed with LTL.

Many quantitative constraints that can be easily expressed by LOC are not suitable for LTL. Specifically, when more than one events need to be compared in the same constraint (e.g., the latency constraint), LTL is not expressive enough to be used. For example, the data consistency constraint

$$\text{data}(\text{input}[i]) = \text{data}(\text{output}[i]) \quad (10)$$

requires comparing each instance of output with the instance of input with the same instance index. After the n th input occurs, it is unknown when the n th output will occur, i.e., the number of input instances that may occur before the n th instance of output is arbitrarily large. Therefore, this constraint cannot be modeled by a finite-state system, and it is impossible to express it using any formalism based on ω -regular languages, such as LTL or PSL.

It is interesting to note that there are simple LOC formulas that cannot be expressed by LTL even though they are ω -regular. For example, the property “the value of event A on every even occurrence is 1,” can be expressed by LOC formula $\text{data}(A[2i]) = 1$, as well as with a simple two-state automaton, but it is well known that it cannot be expressed by LTL [31].

To show that some LTL formulas cannot be expressed in LOC, we first recall that any property can be expressed as a conjunction of a *safety* and a *liveness* property. Safety properties are those which can always be shown violated by a finite trace. For example, any execution that does not satisfy the property “the value of A is never 1” must have a finite prefix which ends with the value of A being 1. On the other hand, liveness properties can ever be violated by a finite trace. For example, the property “for every request there is a response” can never be violated by a finite trace because there is always a chance that a response may come some time in the future.²

Claim 3: LOC can express only safety properties.

Indeed, if a trace does not satisfy an LOC formula, then there must exist an i for which the formula is false. We can evaluate all index expressions for that value of i . Since there can only be finitely many of these expressions, there must exist some point

²To disprove a liveness property, we need to show that the system can enter an infinite cycle in which there are unfulfilled requests.

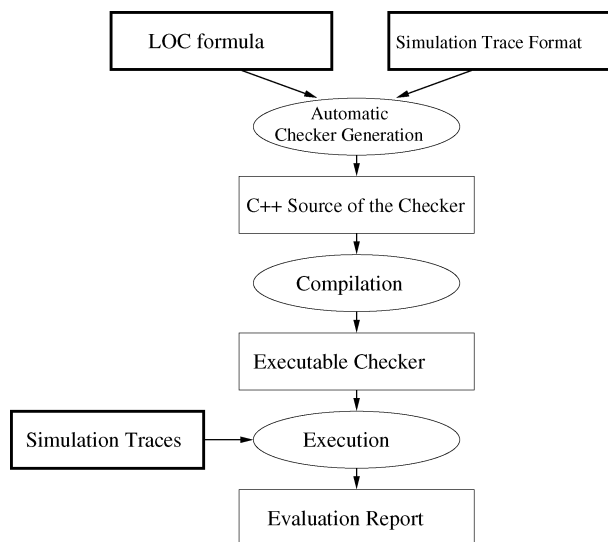


Fig. 2. Trace-analysis methodology.

in the execution such that, for that particular i , the formula does not refer to any event occurrence beyond that point. Clearly, the execution prefix up to that point is sufficient to disprove the property.

On the other hand, LTL is capable of expressing some liveness properties, for example $GF(A)$, i.e., “A occurs infinitely often.”

Conclusion: From Claims (2) and (3), we can conclude that LOC and LTL are incomparable.

Generally, LOC is designed for the specification of quantitative performance and functional constraints at the transaction level, where system events and their annotations are considered. Because of the use of index variable i , LOC is beyond the finite automata domain. On the other hand, LTL is suitable for the specification of functional constraints, and can effectively express the temporal patterns for system-state transitions. Because of this difference, LOC can express important properties that cannot be expressed with LTL, on which the traditional property specification languages are based.

In fact, we have shown that LOC is incomparable with any formalism capable of expressing ω -regular properties. From the theoretical point of view, it may be interesting to establish whether LOC can express all regular properties, i.e., whether LOC is more expressive than WS1S. However, for the methodology we propose here, that question is hardly relevant, because we propose LOC as a complement to and not a replacement for existing property languages capable of expressing regular properties.

IV. SIMULATION-BASED TRACE-ANALYSIS APPROACH FOR LOC FORMULAS

In this section, we present a simulation-based trace-analysis approach, and show that LOC constraints can be easily analyzed in an assertion-based simulation verification environment. The methodology for simulation-based verification with an automatically generated LOC checker is illustrated in Fig. 2. From the specification of LOC formulas and trace formats, an automatic checker generator is used to generate a C++ source of the

```

[LOC: rate]
formula: t(Disp[ay][i + 1])t(Disp[ay][i]) == 10
annotation: event value t
trace: "%s : %d at time %f"

[LOC: latency]
formula: t(Disp[ay][i])t(Stimuli[i]) <= 25
annotation: event value t
trace: "%s : %d at time %f"
  
```

Fig. 3. Definition of LOC formulas and trace formats.

```

username@chimera $ checker latency.trace
Reading from trace file "latency.trace" ...
Formula t(Disp[ay][i]) t(Stimuli[i]) <= 25 is violated
at trace line# 278: Disp[ay] : 6 at time 87
where i = 23
t(Disp[ay][i]) = 87
t(Stimuli[i]) = 60
:
  
```

Fig. 4. Example of error report.

checker. The source code is compiled into an executable that takes in simulation traces and reports any constraint violation.

An example of the definition file for the LOC formulas and trace formats is shown in Fig. 3. Each LOC formula is preceded by a label and followed by the format for extracting event names and their annotations out of the simulation traces. The format described in the figure is written to work with the trace shown in Fig. 7. It specifically looks for a line that starts with a string which ends in a “:”, followed by an integer, a string pattern “at time,” then followed by a floating point number. The string is taken as an event name, and each such line describes a particular instance of that event. The integer is taken as the value of that instance, and the floating point is taken as its “ t ” annotation. Which instance of an event a line is describing is naturally determined by the number of lines that precede it and match the same event name. For example, the n th line matching the pattern with event name “Display” describes the n th event instance of “Display.” Any line that does not match this format will be ignored. Multiple formulas may be checked at the same time with possibly different extraction formats.

The automatic checker generator parses the definition file to generate a C++ source for the checker in a straightforward manner, setting up the queue data structures for storing the annotations and translating the formula into C++ code. The detail of the algorithm inside the checker will be explained later in this section.

To help the designer find the point of error easily, the error report includes the value of index i , which violates the constraint and the value of each annotation in the formula. Fig. 4 shows the case where latency between the 23rd event instance of Display and 23rd event instance of Stimuli violate the given formula. The checker is designed to keep checking and reporting any violation until stopped by the user or if the trace terminates. We will discuss the LOC checker in three aspects: the algorithm of the LOC checking, the runtime monitoring, and how to deal with memory limitation.

A. LOC Checker

The algorithm of LOC checking progresses based on the index variable i . Each LOC formula instance is checked sequentially with the value of i being $1, 2, \dots$, etc. A formula

instance is a formula with i evaluated to some fixed positive integer value, e.g., $\text{Display}[30] - \text{Display}[29] = 10$ is the 29th instance of the formula (1). Starting with i equal to 1, the LOC checker scans the trace sequentially. If any relevant data is read in, the checker stores it into a queue and checks the formula in the following manner:

```
check_formula {
  while (can evaluate instance i) {
    evaluate formula instance i;
    i++;
    memory recycling;
  }
}
```

The time complexity of the algorithm is linear in the size of the trace, since evaluating a particular Boolean expression takes constant time. The memory usage, however, may become prohibitively high if we try to keep the entire trace in the queue for analysis. As the trace file is scanned in, the checker attempts to store only the useful annotations, and in addition, evaluate as many formula instances as possible and remove from the memory parts of the annotations that are no longer needed (memory recycling).

For many LOC formulas [e.g., constraints (1), (3)–(5) in Section II], the algorithm uses a fixed amount of memory, no matter how long the traces are (see Table I).³ Memory efficiency of the algorithm comes from being able to free stored annotations as their associated formula instances are evaluated. This ability is directly related to the choice made in designing LOC. From the LOC formula, we can conservatively identify what annotation data will not be useful anymore once all the formula instances with indices less than a certain number are all evaluated. For example, consider an LOC formula:

$$t(\text{Display}[i + 10]) - t(\text{Stimuli}[i + 5]) < 300 \quad (11)$$

and let the current value of i be 100. Because the value of i increases monotonically, we know that event *Display*'s annotation t with an index less than 111 and event *Stimuli*'s annotation t with an index less than 106 will not be useful in the future and their memory space can be released safely. Each time the LOC formula is evaluated with a new value of i , the memory recycling procedure is invoked, which ensures minimum memory usage.

As described in Section II, the LOC semantics allow us to evaluate an LOC formula even if some of its expressions are not defined. When an annotation with a particular index value is not yet available from the trace, or when the index has an invalid value (e.g., negative value), the Boolean expression that contains this annotation is evaluated to *undef*. The entire LOC formula could then be evaluated according to the standard three-value logic [32] evaluation. For example, given the following LOC formula:

$$t(A[i + 10]) > 100 \vee t(B[i - 5]) < 300 \quad (12)$$

³The verification of the constraint (2) may also have constant memory usage if the given trace has a certain regular structure.

let the current value of i be 10. If we know, from the trace, that the value of $t(A[20])$ is 200, the formula can already be evaluated to be true even if the value of $t(B[5])$ is still not available at this point in the simulation (trace). Thus, the LOC formula instances can be evaluated as soon as possible, which further minimizes the memory usage. Also, if we let the current value of i be 4, -1 is then an invalid index for annotation t of event *B*. The expression $t(B[-1]) < 300$ is evaluated to *undef* and the whole formula can be evaluated to true if the evaluation of $t(A[14]) > 100$ is true, and *undef* otherwise.

B. Runtime Monitoring

The static-trace checking technique, as described above, assumes that a simulation trace is first generated and the subsequent LOC checking parses the trace and looks for constraint violation. How the trace is generated is immaterial as long as the format is correctly specified in the definition file. The trace file for a realistic design, however, can frequently occupy several gigabytes of disk space. It may be desirable to compile the checker as a runtime monitor to run concurrently with the simulator through a Unix pipe. Alternatively, the checker can be compiled into the compiled-code simulator for higher efficiency and tighter integration. As an example of such tight integration, the checker generator has been extended to generate LOC checkers as SystemC modules [14]. During the simulation, other SystemC modules (representing the design) can pass the events and annotations directly to the monitor modules through channels. A case study of this approach is reported in Section V-B. Runtime monitoring is more efficient than static checking, but then, obviously, the simulation need to be repeated if some new formula need to be checked later. Furthermore, the trace is no longer kept so any debugging has to rely solely on the error report.

C. Dealing With Memory Limitation

Despite the memory efficiency for most LOC formulas, some LOC formulas may require high memory usage that the verification environment cannot support. To deal with the case of preset memory limitation, another extension has been added to the checker generator. Generally, the checker tries to read the trace and store the annotations only once. However, if the preset memory limit has been reached, it stops storing the annotation and instead, scans the rest of the trace looking for needed events and annotations for evaluating the current formula instance (with the current value of i). After freeing some memory space, the algorithm resumes storing annotations and reading the trace again from the same location. The analysis time can certainly be impacted (see the case study in Section V-B) and may no longer be of linear complexity. However, the verification can continue and the constraint violations can be checked under the memory limitation of the verification environment.

V. CASE STUDIES

We apply our LOC-based simulation-verification methodologies to two design examples. The first is a system-level design for set-top video processing, PIP, which is originally specified with Y-chart application programming interface (YAPI) [33]. PIP is partially respecified and simulated with Metropolis envi-

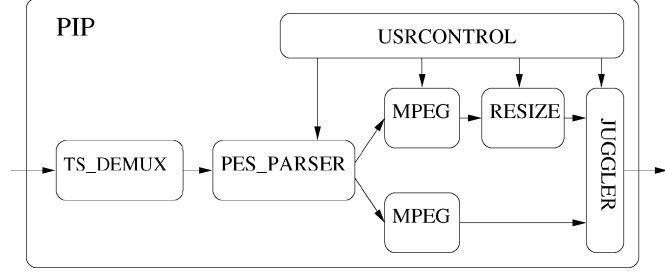


Fig. 5. PIP design.

ronment [3]. The other one is an register transfer level model of an FIR filter written in SystemC and is actually part of the standard SystemC distribution. We use the generated trace checkers to verify a wide variety of functional and performance constraints.

A. PIP

Fig. 5 shows the PIP design. TS_DEMUX demultiplexes the single-input transport stream (TS) into multiple packetized elementary streams (PES). PES_PARSER parses the PES to obtain MPEG video streams. Under the control of the user (USRCONTROL), decoded video streams can either be resized (through RESIZE) or directly feed to JUGGLER, which combines the images to produce the PIP videos. The entire description consists of approximately 19 000 lines of Metropolis [3] and YAPI code. With the sample input stream we used, it produced 120 000 lines of output representing header information for the processed frames.

In the transaction-level design of PIP, where time is still not available, we can check both functional and performance constraints with proper annotations output from the simulation. In the component RESIZE of PIP, the images processed are in interlaced format with alternating fields of all odd lines, then all even. The image size should only change after a complete frame, each of which has two fields, is produced. Therefore, the field sizes of paired even and odd fields should be the same. This property can be expressed as an LOC formula

$$\begin{aligned} & \text{size}(\text{field_start}[2i + 2]) - \text{size}(\text{field_start}[2i + 1]) \\ &= \text{size}(\text{field_start}[2i + 1]) - \text{size}(\text{field_start}[2i]) \end{aligned} \quad (13)$$

where *field_start* is an event at which RESIZE starts to output a new image field. The annotation *size* is the cumulative number of pixels processed by RESIZE. Fig. 6 shows snapshots of the PIP trace. The generation of the checker for this LOC formula and the actual checking on the simulation trace take less than 1 min of CPU time.

Another functional property we are interested in is that the number of the fields the RESIZE component reads in should be equal to the number of fields it produces. Two local counters, one at RESIZE's input part and one at its output part, provide these annotations. After a piece of video is processed, these two counters need to be compared to see if the property holds. The LOC used to check this property is

$$\text{field_cnt}(\text{in}[i]) = \text{field_cnt}(\text{out}[i]), \quad (14)$$

```

:
WINDOW_DATA_OUT 23483 87000
WINDOW win_params_update x_begin: 12 y_begin: 6
RESIZE field_start field_count: 2 size: 6720
:
WINDOW win_params_update x_begin: 12 y_begin: 6
USRCONTROL write pixels_out: 144
RESIZE field_start field_count: 3 size: 10368
:
USRCONTROL write lines_out: 64
THSRC_CTL_OUT finfo_write value: 12876
RESIZE field_start field_count: 4 size: 14016
:

```

Fig. 6. PIP simulation trace.

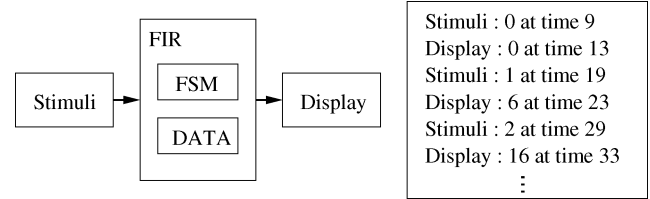


Fig. 7. FIR design and simulation trace.

The events in and out are generated by the input and output parts of RESIZE respectively whenever they finish processing a whole piece of video. The annotation *field_cnt* represents the number of fields processed by the input and output parts of RESIZE. The generation of the checker for this formula and the actual trace checking take less than 1 min of CPU time.

We can also check performance properties such as latency. The latency issue in RESIZE relates to the timely response to user-size specification. Since PIP is specified at the behavior level, no detail timing information is available. We, therefore, specify a bound (e.g., 5) on the number of fields processed between reading a new size specification (*read_size*) and the actual change in output image size (*change_size*)

$$\text{field_cnt}(\text{change_size}[i]) - \text{field_cnt}(\text{read_size}[i]) \leq 5 \quad (15)$$

where *read_size* is generated whenever RESIZE reads a new size specification from USRCONTROL, and *change_size* is generated whenever the size of the output image is actually changed. The annotation *field_cnt* is the value of a global counter that is incremented by one whenever RESIZE processes a new image field. The generation of the checker for this LOC formula and the actual trace checking also take less than 1 min of CPU time.

B. FIR Filter

While the PIP example illustrates the power of LOC in dealing with transaction-level functional and performance properties, we use the FIR example to show how LOC can be used to efficiently check real time performance properties. Fig. 7 shows a 16-tap FIR filter which reads in samples when the input is valid and writes out the result when output is ready. The filter design is divided into a control FSM and a data path. The test bench feeds sampled data of arbitrary length and the output is displayed with the simulator.

We use our automatic trace checker generator to verify the properties specified in constraints (1)–(5) (of Section II). The

TABLE I
COSTS OF CHECKING CONSTRAINTS (1)–(5) ON FIR

Lines of Trace		10^5	10^6	10^7	10^8
C1	Time(s)	1	8	89	794
	Memory	28B	28B	28B	28B
C2	Time(s)	1	12	120	1229
	Memory	28B	28B	28B	28B
C3	Time(s)	1	7	80	799
	Memory	24B	24B	24B	24B
C4	Time(s)	1	7	77	803
	Memory	0.4KB	0.4KB	0.4KB	0.4KB
C5	Time(s)	1	7	79	810
	Memory	4KB	4KB	4KB	4KB

TABLE II
TIME USAGE OF SIMULATION AND CHECKING FOR CONSTRAINT (2) ON FIR

Lines of Trace		10^5	10^6	10^7	10^8
Simulation w/o Monitoring (s)		1	14	148	1404
Static Trace Checking Only (s)		1	12	120	1229
Simulation w/ Monitoring (s)		2	14	145	1420

same trace files are used for all the analyzes and each constraint is checked one at a time. The time and maximum memory usage are shown in Table I. We can see that the time required for analysis grows linearly with the size of the trace file, and the maximum memory requirement is formula dependent, but stays fairly constant. Using LOC for common real-time constraint verification is indeed very efficient.

Given the large file size, runtime monitoring (see Section IV-B) may reduce the total verification time (simulation and checking), since no trace file needs to be actually generated. For the latency constraint [the formula (2)], we implement the checker as a SystemC module and the simulation trace is no longer written to a file, but passed to the monitoring module directly. Table II lists CPU times used for simulation, trace checking, and simulation with runtime monitoring for the formula (2) on the traces with different lengths. For the trace size of 100 million lines, the static checking approach requires 1404 s of simulation time and 1229 s of checking time for a total of 2633 s. The runtime-monitoring technique requires only 1420 s for both simulation and monitoring. If the simulation trace is really long (e.g., hundreds of gigabytes), the runtime monitoring can significantly save CPU time compared to the off-line trace checking.

We also verify constraint (7) to illustrate verification with memory limitation since this constraint is particularly expensive in terms of memory usage. Table III shows that the simulation time grows linearly with the size of the trace file. However, due to the use of an annotation in an index expression, memory can no longer be recycled and we see that it also grows linearly with the size of the trace file. Indeed, since we will not know what annotation will be needed in the future, we can never remove any information from the queue. If the memory is a limiting factor in the simulation environment, the analysis speed must be sacrificed to allow the verification to continue, as discussed in Section IV-C. The result is shown in Table III, where the memory usage is limited to 50 KB. We see that the analysis takes more time when the memory limit has been reached. Information about trace pattern can be used to dramatically reduce the

TABLE III
COSTS OF CHECKING CONSTRAINT (7) ON FIR

Lines of Trace ($\times 10^4$)		2	3	4	5
Unlimited	Time (s)	<1	<1	<1	1
	Memory	Mem(KB)	40	60	80
Mem Limit (50KB)	Time (s)	<1	61	656	1869
	Memory	Mem(KB)	40	50	50

running time under memory constraints. Aggressive memory-minimization techniques and data structures can also be used to further reduce time and memory requirements. For most LOC formulas and simulation traces, however, the memory space can be recycled and the memory requirements are small.

VI. FORMAL VERIFICATION OF LOC FORMULAS

Although our trace analysis enables efficient verification of LOC formulas in a simulation environment, formal verification may still be valuable and sometimes even necessary. We propose to apply formal verification to small designs that are reused many times, such as library modules. Because they are small, formal verification is practically possible. On the other hand, they are intended to be used in many environment, some of which will be designed long after the module itself is designed and verified. Therefore, it is hard to imagine all simulation scenarios that need to be verified. It is better to characterize the modules with a set of properties that it satisfies. This will not only increase the confidence in the correctness, but these properties can be used as a precise specification of a design's behavior as well. The lack of such a specification is a major source of design errors, because informal specifications of library modules are often ambiguous and misunderstood.

Unfortunately, it is undecidable whether a system satisfies an LOC formula, even if some strong restrictions are placed on the system specification and the formula [17]. On the positive side, for a significant subset of LOC, it is possible to decide whether a finite-state system satisfies an LOC formula. The decision procedure is based on constructing a formula of Presburger arithmetic that is satisfied if and only if the formula is violated by some behavior of the system. The LOC subset that can be verified in this way includes all properties described in Section II, except latency property (7).

Manipulating Presburger formulas is very expensive in practice, so we propose an alternative formal verification approach based on existing finite-state model checking tools. Our approach represents a complete verification procedure for a subset of LOC that defines ω -regular properties. We have shown in [17] that rate (1), throughput (4), and burstiness constraints (5) belong to this subset, but other properties in Section II do not. The proposed approach may still be applied to these properties, but the procedure is incomplete in this case, because it can terminate with an inconclusive result.

The simulation approach described in Section IV suggests our formal verification approach. A trace checker can be interpreted as an automaton accepting executions. We could thus use existing model-checking tools to verify that each execution of the system is accepted by the trace checker. In the example shown in this paper, the translation was manual. However, there

is no technical difficulty in automatically generating such descriptions in a language understood by a model checking tool through modifying our trace checker generator.

The only significant difference between a simulation-trace checker and an automaton description suitable for model checking is that former can rely on dynamic memory allocation to store trace data that may be needed, while the latter must have all memory statically allocated. Unfortunately, as we have shown in Section III, for some LOC formula it is not possible to determine memory requirement *a priori*. Our approach is to fix the memory size anyway, and designate special states where checking the formula would require allocating additional memory, but none is available. Such a state may or may not be reached during the reachability analysis. If it is, the result of the formula verification is inconclusive. More precisely, the formal verification can have one of three outcomes:

- a counterexample is found showing that the system does not satisfy the property;
- the property is satisfied, all reachable state are searched without finding a counterexample, or reaching a state where memory is exhausted;
- inconclusive—analysis finds no counterexamples, but states where memory is exhausted are reachable.

For example, the latency constraint

$$t(\text{Display}[i]) - t(\text{Stimuli}[i]) < 25 \quad (16)$$

cannot be modeled by any finite automata because there can be arbitrarily many occurrences of Stimuli before x th occurrence of Display (intuitively, we assume that $\text{Display}[x]$ always occurs after $\text{Stimuli}[x]$). However, if we limit the number of stored time stamps of Stimuli to, say, 50, then we can simultaneously check the following two properties:

P1: There are never more than 50 occurrences of Stimuli between i th occurrences of Stimuli and Display.

P2: If **P1** holds, then (16) holds.

Obviously, if **P1** and **P2** both hold, then so does (16), and if **P2** is false, so is (16). However, if **P2** holds, but **P1** does not, the result is inconclusive.

To specify **P1** and **P2**, assume that the trace checker keeps the 51 most recent time stamps for Stimuli and Display in arrays Display_t and Stimuli_t such that x th time stamp is stored at position $(x \bmod 51)$ of the array. Also, assume that variables Display_i and Stimuli_i (which take values from 0 to 50) keep the index of the most recent time stamps in the arrays. Finally, assume that binary variables $\text{Display}_\text{occur}$ and $\text{Stimuli}_\text{occur}$ are true when Display and Stimuli occur, respectively, and that integer variable diff counts the difference between the numbers of occurrences of the Stimuli and Display events, i.e., it is initialized to 0, incremented on each $\text{Stimuli}_\text{occur}$, and decremented on each $\text{Display}_\text{occur}$. Then, **P1** can be specified with the following state predicate:

$$\text{diff} \leq 51. \quad (17)$$

Constraint (16) can be expressed as follows:

$$\begin{aligned} \text{Display}_\text{occur} \implies & \text{Display}_t[\text{Display}_i] \\ & - \text{Stimuli}_t[\text{Display}_i] < 25 \end{aligned} \quad (18)$$

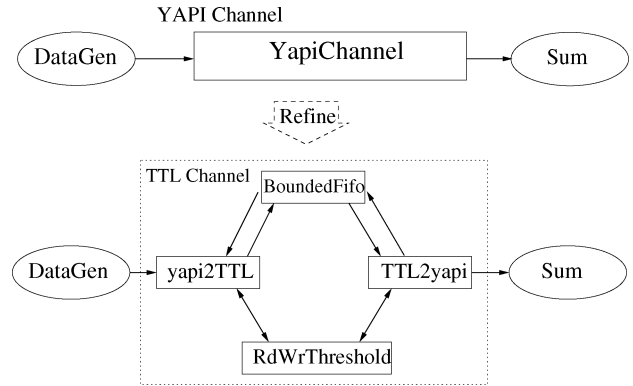


Fig. 8. YAPI and TTL channels.

and finally **P2** can be expressed as follows:

$$\text{Assumption (17)} \implies \text{Formula (18)}. \quad (19)$$

In the following section, we use our formal verification technique to verify the data-consistency constraint of the TTL channel, which is a library module used in our previous PIP example, and show how the formal verification approach works on checking important library modules.

A. Formal Verification for TTL Channel

YAPI is a model of computation for designing signal-processing systems [33]. It is basically a Kahn process network [34], extended with the ability to nondeterministically select an input port to consume and an output port to produce. A YAPI channel models an unbounded first-in-first-out (FIFO) buffer. Asynchronously, a writer process writes data into one end of the channel and a reader process reads data from the other end of the channel. A design methodology based on YAPI was proposed in [15]. It includes refinement of the YAPI channel into a lower-level abstraction called TTL [16]. The refinement is shown in Fig. 8. In our previous PIP example, the TTL channel is instantiated many times and used to interconnect the various components of the design.

At the TTL level, the channel is modeled with a bounded FIFO buffer. The mutual exclusion and boundary checking of the bounded FIFO buffer is guaranteed by a central protocol. As Fig. 8 shows, the TTL channel has a bounded FIFO (*BoundedFifo*) whose size is set at design time, and a control medium (*RdWrThreshold*) which implements a protocol to guarantee correctly writing to and reading from the FIFO buffer. We use a writer process (*DataGen*) to write a series of data into the channel and a reader process (*Sum*) to read the data from it. To verify the correctness of the refinement, we focus on the verification of the TTL channel, which is normally a library module that needs to be frequently reused.

When the writer *DataGen* writes a data into the TTL channel, it produces an event of *prepared*; when the reader *Sum* reads a data from the channel, it produces an event of *processed*. We use the annotation data to represent the value of data written into or read from the channel. An important property that can be expressed with LOC is data consistency of the TTL channel, i.e., the input data of the TTL channel should be read from the

TABLE IV
SUMMARY OF FORMAL VERIFICATION FOR (21) AND (23)

LTL Formula	(21)	(23)
Depth reached	51257	57221
States stored ($\times 10^8$)	2.2431	2.3156
State transitions ($\times 10^8$)	2.85523	3.09726
Total memory (MB)	735.098	819.517
CPU time	1h37m55s	3h03m18s
Hash factor	4.78686	4.63699

channel in exactly the same order without a loss. The data-consistency constraint is defined as

$$\text{data}(\text{prepared}[i]) = \text{data}(\text{processed}[i]). \quad (20)$$

The TTL channel shown in Fig. 8 is initially specified in Metropolis meta-model (MMM) [3]. From the MMM specification of the TTL channel design, we use the Metropolis-backend tool to generate a corresponding Promela (SPIN's modeling language) description [7], which can be verified by the model checker SPIN for a particular LTL formula. The TTL channel design has 634 lines of MMM source code and 2049 lines of Promela code after translation.

From the discussion above, we know that the data-consistency constraint (20) of the TTL channel cannot be expressed by LTL directly. Therefore, we have to assume that, "after the x th write by *DataGen*, at most 31 writes can be done before the x th read by *Sum*."⁴ Then, we use arrays *prepared_data*[32] and *processed_data*[32] to store the recent 32 pieces of data written by *DataGen* and read by *Sum*, respectively. We also use variables *prepared_i* and *processed_i* (which take values of 0 to 31) to keep the index of the most recent data in the arrays, and variable *diff* to keep the difference between the numbers of write and read events. The assumption is written in LTL as

$$\mathcal{G}(\text{diff} \leq 32) \quad (21)$$

and it is verified to hold by SPIN. The data-consistency constraint is written in LTL as

$$\mathcal{G}(\text{processed_occur} \implies \text{prepared_data}[\text{processed_i}] = \text{processed_data}[\text{processed_i}]). \quad (22)$$

Because *processed*[x] always follows *prepared*[x], the data consistency only needs to be checked when an instance of *processed* is occurring. The formula:

$$\text{Assumption (21)} \implies \text{Constraint (22)} \quad (23)$$

is also verified to hold by SPIN.

With the bitstate technique [35], SPIN verifies (21) and (23), using about 1.5 and 3 h of CPU time, respectively, on our 1.5-GHz Athlon machine with 1 GB of memory. And all the other relevant verification parameters are listed in Table IV. From this case study (compared to the case studies in Section V), we can clearly see the tradeoff between the simulation trace checking and the formal verification. The simulation-trace checking is usually much more efficient in terms of memory and CPU time usage, but its verification results totally depend on the design of test cases for simulation. On the other hand,

⁴This assumption is derived from the actual buffer size of the TTL channel.

the formal verification is more expensive but the results are largely independent of test cases. Therefore, it should be used for small but important design modules like the TTL channel.

VII. CONCLUSION

In this paper, we discuss the verification aspects of the quantitative constraint formalism, LOC. We compare LOC with LTL, find that LOC has a different domain of expressiveness than LTL, and conclude that LOC can express important properties that cannot be expressed by LTL. We propose two feasible verification approaches, simulation-trace analysis and model checking. We also present a set of case studies on these approaches to demonstrate their usefulness and effectiveness.

We are currently working on a few future enhancements and novel applications. One such application we are considering is to integrate the LOC monitor with a simulator that is capable of nondeterministic simulation, nondeterminism being crucial for design at high levels of abstraction. We will use the checker to check for constraint violations, and once a violation is found, the simulation could roll back and look for another nondeterminism resolution that does not violate the constraint. In addition, to help the designer easily produce traces for constraint checking, we plan to develop embedded-code blocks for trace generation in the form of libraries, similar to embedded constraint languages. We also plan to retarget the backend checker generation for different development environments (e.g., SystemVerilog, Verilog, VHDL) to allow tight integration of monitors for those environments as well.

APPENDIX

FORMAL LOC SYNTAX AND SEMANTICS

A. 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. In general, we want to consider both finite and infinite sequences, as well as hybrids where some inputs or outputs appear infinitely many times, and some appear only finitely many times. Formally, let E be a set of *event names*⁵ and for each $e \in E$ let $V(e)$ be its *value domain*. Then, a *behavior* β is a partial function from $E \times \mathbb{Z}$ to $\bigcup_{e \in E} V(e)$ such that:

- 1) $\beta(e, n) \in V(e)$ for each $e \in E$, and each positive integer n for which $\beta(e, n)$ is defined;
- 2) if $\beta(e, n)$ is not defined for some $e \in E$ and positive integer n , then $\beta(e, m)$ is not defined for any $m > n$;
- 3) $\beta(e, n)$ is not defined for any $e \in E$ and any $n \leq 0$.⁶

If n is the largest integer for which $\beta(e, n)$ is defined, then we say that there are n *instances* of e in β . We also say for all positive integers $k \leq n$ that $\beta(e, k)$ is the value of the k th instance of e in β .

A *system* is specified by a set of event names, their value domains and a *set of behaviors*. In a typical system, event

⁵In 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.

⁶Clearly, we could have defined β as a partial function on positive integers, but this definition happens to be more convenient when we define the semantics.

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 observed on the wires, or sequences of messages to mailboxes.

Behaviors, by themselves, are not sufficient to evaluate performance constraints that may involve quantities like timing or power of the system. For this, we need additional information regarding performance measures. We represent this information as annotations to behaviors. Formally, given an arbitrary set T , an *annotation* of behavior β of type T is a partial function f from $E \times \mathbb{Z}$ to T , such that $f(e, n)$ is defined if and only if $\beta(e, n)$ is. We refer to f as a T -valued annotation of β . 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 of β .

In this paper, we show a few uses of annotations, but make no proposal for their specification. We assume that they are part of the functional specification, and thus specified with the same language as the 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. In other words, annotated behaviors are models of LOC formulas.

B. LOC Syntax

LOC formulas are defined relative to a multisorted 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 $T_1 \times \dots \times T_n \mapsto \{\text{true}, \text{false}\}$. We require that \mathcal{A} contains at least the set of integers, 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 ($=, <, >, \dots$).

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

- i is an integer-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 integer-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;
- 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.

We say that τ in a term of the form $\text{val}(e[\tau])$ or $f(e[\tau])$ is an *index expression*.

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 integer-valued events, and f and g are integer-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.

C. LOC Semantics

We first define the *value* of formulas and terms with respect to an annotated behavior and a value of the variable i . We use a special symbol undef to denote that the value of a term or a formula is not defined, and assume that undef is distinct from any element of any sort in \mathcal{A} . 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 . If α is a T -valued term, then $\mathcal{V}_{(\beta, A)}^n \llbracket \alpha \rrbracket$ is in $T \cup \{\text{undef}\}$, and if α is a formula, then $\mathcal{V}_{(\beta, A)}^n \llbracket \alpha \rrbracket$ is in $\{\text{true}, \text{false}, \text{undef}\}$. Note that this implies that for some k -ary T -valued operator o , the formula $o(\tau_1, \dots, \tau_k)$ can take value undef , while o itself cannot, because it is T -valued. There is no contradiction here, only a slight abuse of notation, as we use the same symbol o to represent both the operator and its name appearing in LOC formulas. This ambiguity in the meaning of o , can always be easily resolved from the context in which o appears. Also note that we do not make a requirement that all annotations appearing in the formula must be defined in A . For such undefined annotations, we use value undef . The value of an LOC formula is defined recursively as follows:

- $\mathcal{V}_{(\beta, A)}^n \llbracket i \rrbracket = n$;

$$\mathcal{V}_{(\beta, A)}^n \llbracket \text{val}(e[\tau]) \rrbracket = \begin{cases} \text{undef}, & \text{if } \mathcal{V}_{(\beta, A)}^n \llbracket \tau \rrbracket = \text{undef}, \text{ or } \beta(e, \mathcal{V}_{(\beta, A)}^n \llbracket \tau \rrbracket) \text{ is not defined} \\ \beta(e, \mathcal{V}_{(\beta, A)}^n \llbracket \tau \rrbracket), & \text{otherwise} \end{cases}$$

$$\mathcal{V}_{(\beta, A)}^n \llbracket f(e[\tau]) \rrbracket = \begin{cases} \text{undef}, & \text{if } f \notin A, \text{ or } \mathcal{V}_{(\beta, A)}^n \llbracket \text{val}(e[\tau]) \rrbracket = \text{undef} \\ f(e, \mathcal{V}_{(\beta, A)}^n \llbracket \tau \rrbracket), & \text{otherwise} \end{cases}$$

- $\mathcal{V}_{(\beta,A)}^n[[c]] = c$ for each element c of each value domain T ;
- for each annotation f , each event name e , and each integer-valued term, τ , let $\mathcal{V}_{(\beta,A)}^n[[\text{val}(e[\tau])]]$ and $\mathcal{V}_{(\beta,A)}^n[[f(e[\tau])]]$ be defined by the equations at the bottom of the previous page;
- for each k -ary operator o , using v_j to denote $\mathcal{V}_{(\beta,A)}^n[[\tau_j]]$ for each $j = 1, \dots, k$

$$\mathcal{V}_{(\beta,A)}^n[[o(\tau_1, \dots, \tau_k)]] = \begin{cases} \text{undef}, & \text{if } v_j = \text{undef for some } j \\ o(v_1, \dots, v_k), & \text{otherwise} \end{cases}$$

- for each k -ary relation r , using v_j to denote $\mathcal{V}_{(\beta,A)}^n[[\tau_j]]$ for each $j = 1, \dots, k$

$$\mathcal{V}_{(\beta,A)}^n[[r(\tau_1, \dots, \tau_k)]] = \begin{cases} \text{undef}, & \text{if } v_j = \text{undef for some } j \\ r(v_1, \dots, v_k), & \text{otherwise} \end{cases}$$

- $\mathcal{V}_{(\beta,A)}^n[[\phi]] = \begin{cases} \text{true}, & \text{if } \mathcal{V}_{(\beta,A)}^n[[\phi]] = \text{false} \\ \text{false}, & \text{if } \mathcal{V}_{(\beta,A)}^n[[\phi]] = \text{true} \\ \text{undef}, & \text{otherwise} \end{cases}$
-

$$\mathcal{V}_{(\beta,A)}^n[[\phi \wedge \psi]] = \begin{cases} \text{true}, & \text{if } \mathcal{V}_{(\beta,A)}^n[[\phi]] = \text{true}, \text{ and } \mathcal{V}_{(\beta,A)}^n[[\psi]] = \text{true} \\ \text{false}, & \text{if } \mathcal{V}_{(\beta,A)}^n[[\phi]] = \text{false}, \text{ or } \mathcal{V}_{(\beta,A)}^n[[\psi]] = \text{false} \\ \text{undef}, & \text{otherwise} \end{cases}$$

•

$$\mathcal{V}_{(\beta,A)}^n[[\phi \vee \psi]] = \begin{cases} \text{true}, & \text{if } \mathcal{V}_{(\beta,A)}^n[[\phi]] = \text{true}, \text{ or } \mathcal{V}_{(\beta,A)}^n[[\psi]] = \text{true} \\ \text{false}, & \text{if } \mathcal{V}_{(\beta,A)}^n[[\phi]] = \text{false}, \text{ and } \mathcal{V}_{(\beta,A)}^n[[\psi]] = \text{false} \\ \text{undef}, & \text{otherwise.} \end{cases}$$

We say that an annotated behavior (β, A) satisfies a formula ϕ , if $\mathcal{V}_{(\beta,A)}^n[[\phi]] = \text{false}$ does not hold for any integer n .

ACKNOWLEDGMENT

The authors acknowledge team members of the Metropolis project led by Prof. Sangiovanni-Vincentelli from the University of California, Berkeley. The work presented here has been done within the framework of the Metropolis project, and it has benefited greatly from many discussions with team members.

REFERENCES

- [1] J. P. Hayes, *Computer Architecture and Organization*. New York: McGraw-Hill, 1988.
- [2] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System level design: Orthogonalization of concerns and platform-based design," *IEEE Trans. Computer-Aided Design*, vol. 19, pp. 1523–1543, Dec. 2000.
- [3] F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, M. Sgroi, and Y. Watanabe, "Modeling and designing heterogeneous systems," Cadence Berkeley Laboratories, Berkeley, CA, Tech. Rep. 2001/01, 2001.
- [4] X. Chen, F. Chen, H. Hsieh, F. Balarin, and Y. Watanabe, "Formal verification of embedded system designs at multiple levels of abstraction," in *Proc. Int. Workshop High-Level Design Validation Test*, Oct. 2002, pp. 125–130.
- [5] A. Pnueli, "The temporal logic of programs," in *Proc. 18th IEEE Symp. Foundation Comput. Sci.*, 1977, pp. 46–57.
- [6] K. McMillan, *Symbolic Model Checking*. Norwell, MA: Kluwer, 1993.
- [7] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Software Eng.*, vol. 23, pp. 279–295, May 1997.
- [8] PSL Homepage (2003). [Online]. Available: <http://www.eda.org/vfv>
- [9] System Verilog Assertions Homepage (2003). [Online]. Available: <http://www.eda.org/sv-ac>
- [10] "OpenVera Assertions White Paper," Synopsys, Inc., Mountain View, CA, 2002.
- [11] F. Balarin, Y. Watanabe, J. Burch, L. Lavagno, R. Passerone, and A. Sangiovanni-Vincentelli, "Constraints specification at higher levels of abstraction," in *Proc. Int. Workshop High-Level Design Validation Test*, Nov. 2001, pp. 129–133.
- [12] X. Chen, H. Hsieh, F. Balarin, and Y. Watanabe, "Automatic trace analysis for logic of constraints," in *Proc. 40th Design Automation Conf.*, June 2003, pp. 460–465.
- [13] —, "Verifying LOC based functional and performance constraints," in *Proc. Int. Workshop High-Level Design Validation Test*, Nov. 2003, pp. 83–88.
- [14] SystemC Homepage (2003). [Online]. Available: <http://www.systemc.org>
- [15] J. Brunel, E. A. de Kock, W. M. Kruijtzter, H. J. H. N. Kenter, and W. J. M. Smits, "Communication refinement in video systems on chip," in *Proc. 7th Int. Workshop Hardware/Software Codesign*, 1999, pp. 142–146.
- [16] O. Gangwal, A. Nieuwland, and P. Lippens, "A scalable and flexible data synchronization scheme for embedded HW-SW shared-memory systems," in *Proc. Int. Symp. Syst. Synthesis*, Oct. 2001, pp. 1–6.
- [17] X. Chen, H. Hsieh, F. Balarin, and Y. Watanabe, "Logic of constraints: A quantitative performance and functional constraint formalism," Univ. California, Riverside, Tech. Rep. UCR-CS-04-87, 2004.
- [18] P. Alexander, C. Kong, and D. Barton. (2001) Rosetta Usage Guide. [Online]. Available: <http://www.sldl.org>
- [19] Object Constraint Language Specification (1997). [Online]. Available: <http://www.omg.org>
- [20] Quick Reference Guide for the Design Constraints Description Language (2000). [Online]. Available: <http://www.eda.org/dcwg>
- [21] J. R. Buchi, "On a decision method in restricted second order arithmetic," in *Proc. Int. Congress Logic, Method. Phil. Sci.*, 1960, pp. 1–11.
- [22] E. Cerny, B. Berkane, P. Girodias, and K. Khordoc, *Hierarchical Annotated Action Diagrams: An Interface-Oriented Specification and Verification Method*. Norwell, MA: Kluwer, 1998.
- [23] B. Alpern and F. Schneider, "Verifying temporal properties without temporal logic," *ACM Trans. Program. Lang.*, vol. 11, no. 1, pp. 147–167, 1989.
- [24] Z. Har'El and R. P. Kurshan, "Software for analysis of coordination," in *Proc. Int. Conf. Syst. Sci.*, 1988, pp. 382–385.
- [25] J. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm, "Mona: Monadic second-order logic in practice," in *Proc. Tools Algorithms Construction Anal. Syst., 1st Int. Workshop (TACAS)*, vol. 1019, LNCS, 1995.
- [26] F. Jahanian and A. K. Mok, "Safety analysis of timing properties in real-time systems," *IEEE Trans. Software Eng.*, vol. 12, pp. 890–904, Sept., 1986.
- [27] A. Mok and G. Liu, "Early detection of timing constraint violation at runtime," in *Proc. IEEE Real-Time Syst. Symp.*, Dec. 1997, pp. 176–186.
- [28] —, "Efficient run-time monitoring of timing constraints," in *Proc. Real-Time Technol. Applicat. Symp.*, June 1997, pp. 252–262.
- [29] M. Y. Vardi, "An automata-theoretic approach to linear temporal logic," *Logics Concurrency. Structure Versus Automata*, vol. 1043, pp. 238–266, 1996.
- [30] C. Eisner and D. Fisman, "Sugar 2.0 Proposal, presented to the Accellera Formal Verification Tech. Comm.," 2002.
- [31] P. Wolper, "Temporal logic can be more expressive," *Info. Contr.*, vol. 56, pp. 72–99, 1983.
- [32] E. J. McCluskey, *Logic Design Principles*. Englewood Cliffs, NJ: Prentice-Hall, 1986.
- [33] E. d. Kock, G. Essink, W. Smits, P. v. d. Wolf, J. Brunel, W. Kruijtzter, P. Lieverse, and K. Vissers, "Yapi: Application modeling for signal processing systems," in *Proc. 37th Design Automation Conf.*, June 2000, pp. 402–405.
- [34] G. Kahn, "The semantics of a simple language for parallel programming," in *Proc. IFIP Congress*, 1974, pp. 471–475.
- [35] G. J. Holzmann, "An analysis of bitstate hashing," *Formal Methods Syst. Design*, vol. 13, no. 3, pp. 289–307, Nov. 1998.



Xi Chen received the B.E. degree in computer science and engineering from Zhejiang University, Hangzhou, China, in 2000 and the M.S. degree in computer science in 2002 from the University of California, Riverside, where he is currently pursuing the Ph.D. degree in computer science.

His research interests include verification of embedded-system designs, system-level design methodologies, and distributed computing.



Felice Balarin (S'90–M'95) received the Ph.D. degree in electrical engineering and computer science from the University of California, Berkeley, in 1994.

He has been a Research Scientist at Cadence Berkeley Labs, Berkeley, CA. His research is focused on the development and application of formal methods to design, verification, specification, and analysis of embedded systems implemented by both hardware and software. He is the author of numerous papers and coauthor of two books on these topics.



Harry C. Hsieh received the B.S. degree from the University of Wisconsin, Madison, in 1988, the M.S. degree from Stanford University, Stanford, CA, in 1991, and the Ph.D. degree from the University of California, Berkeley, in 2000.

From 1991 to 1993, he was a member of the Technical Staff at Mainline Systems Lab, Hewlett Packard. He was also a Researcher at Cadence Berkeley Laboratories in 2000. He joined the faculty of the Computer Science and Engineering Department, University of California, Riverside, in

2001. He has been actively involved as a Researcher and Teacher in the area of design technology, especially for methodology, synthesis, and verification of very large scale integration and embedded systems. He has coauthored two books and numerous journal, workshop, and conference papers in that area.

Prof. Hsieh has served on many review boards and program committees. He recently served on the organizing committee for the International Conference on Hardware/Software Codesign and System Synthesis in 2003 and 2004, as well as the Design Automation and Test in Europe in 2004.



Yosinori Watanabe (S'88–M'93) received the B.Eng. degree from Waseda University, Tokyo, Japan, in 1988 and the Ph.D. degree from the University of California, Berkeley, in 1994.

In 1994, he joined the Digital Equipment Corporation. He was a member of the Design Team for the ALPHA microprocessor while also being engaged in logic synthesis for high-performance microprocessors. Since 1997, he has been with Cadence Laboratories, where he has been involved in a research project for developing a design environment and methodologies for embedded systems.

Dr. Watanabe received the IEEE Circuits and Systems (CAS) Outstanding Young Author Award and the IEEE CAS Best Paper Award on Transactions of Computer-Aided Design in 1995 and 1998, respectively.