Logic of Constraints: A Quantitative Performance and Functional Constraint Formalism

Xi Chen*, Harry Hsieh*, Felice Balarin†, Yosinori Watanabe†

*University of California at Riverside, {xichen, harry}@cs.ucr.edu

†Cadence Berkeley Laboratories, {felice, watanabe}@cadence.com

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 language that is particularly suited to express quantitative performance constraints as well as functional constraints. We explore the complexity of LOC formal verification, and show that some versions of the problem are undecidable, and some are decidable, but with very complex algorithms. For practical purposes, we therefore propose a partial formal verification methodology and an automatic simulation trace checking/monitoring methodology, both can be used to verify system designs. We analyze the expressiveness of LOC and show that it is important and different from Linear Temporal Logic (LTL), on which traditional hardware assertion languages (e.g. IBM's Sugar and Synopsys's OpenVera) are based. Through several industrial case studies, we demonstrate the usefulness of the LOC formalism and the corresponding verification approaches at the higher, transaction level of abstraction.

I. Introduction

The increasing complexity of embedded systems today demands more sophisticated design and verification methodologies. Systems are becoming more integrated as more and more functionalities and features are required for the product to succeed in the market. Embedded system

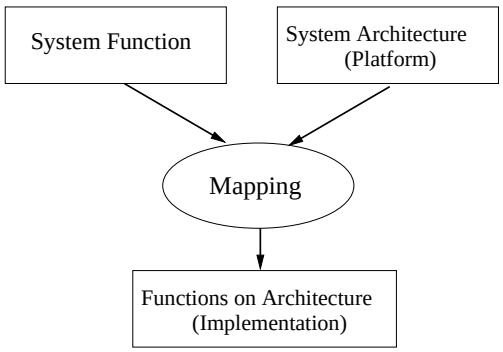

Fig. 1. System Design Methodology.

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 logics) all utilized on a single chip. Designing at the Register Transfer Level (RTL) [11] 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 refine the abstract function, refine the abstract architecture, and map the function onto the architecture through automatic tools or manual means with tool support [17], [4]. 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 Figure 1).

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 [6]. Designers can prove the property of a specification by writing down the property they want to check in some logic (e.g. Linear Temporal Logic (LTL) [21]) and use a formal verification tool (e.g. the model checker SPIN [12]) 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 HDL (Hardware Description Language) 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 IBM's Sugar2.0 [7] and Synopsys' OpenVera [1].

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 formal constraint language: Logic of Constraints (LOC). LOC is particularly suited for specification and simulation analysis of real time performance constraints at the transaction level, as will be shown later in this paper. A constraint language is not meaningful unless there exists a clear and efficient path to verification. We propose an efficient simulation-based approach for analyzing 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 [3] 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 three separate case studies: a high level description of a *Picture-In-Picture (PIP)* design, an RTL level design of a *Finite Impulse Response (FIR)* filter, and a *Task Transition Level (TTL)* [9] channel design, which is a refinement of a Y-Chart Application Programming Interface (YAPI) [18] communication entity.

A simulation-based approach can only disprove the LOC formula (if a violation is found), but it can never prove it conclusively, as that would require analyzing the design space exhaustively. However, for small designs or library modules that will be instantiated many times across different designs, it is necessary to formally prove the desired properties. We give an exact verification algorithm for a broad class of LOC formulas. However, because of the high complexity of this algorithm, we provide an alternative. That is, we propose a formal verification

methodology where LOC formulas are translated into verification models in Promela (SPIN's input language [12]) 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 formal verification of the TTL library module.

While similar in spirit to the hardware embedded assertion languages, our LOC simulation and verification approaches are indeed useful in at least three fundamental aspects. First, Logic of Constraints is designed for expressing 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, to express performance constraints effectively, LOC can be used to express properties that cannot be handled by LTL. 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, as will be shown in the remainder of this paper.

The rest of the paper is organized as follows. In the next section, we introduce the quantitative constraint formalism, Logic of Constraints (LOC) and its typical usage. We also compare our work with the related research. In Section III, we analyze the expressiveness of LOC, and show that LOC can be used to express important constraints that can not be expressed with LTL for the specification of system designs. In Section IV, we present some results on the complexity of LOC verification, most of the proofs for which we give in Appendix I. In Section V, we first present the methodology for building a trace checker or monitor from any given LOC formula, and then discuss the partial formal verification approach for LOC formulas. We demonstrate the usefulness and efficiency of these approaches with three verification case studies in Section VI. In Section VII, we conclude the paper and provide our future research directions.

II. Logic of Constraints

In this section, we introduce our quantitative constraint formalism, Logic of Constraints (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. We will 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. After the formal presentation, we discuss the typical usage of LOC and the typical constraints that it can express.

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*[1] 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 $n \leq 0$.[2]

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 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 constraints that involve 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 with respect to T is a partial function f from $E \times \mathbb{Z}$ to T, such

[1] In 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.

[2] Clearly, 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.

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. 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.

B. 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 $T_1 \times \cdots \times T_n \mapsto T_{n+1}$, where n is a natural number, and $T_1, \ldots, T_{n+1}$ 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 $T_1 \times \cdots \times T_n \mapsto \{true, false\}$. 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 $(=, <, >, \ldots)$.

LOC formulas may contain only one variable, namely i. The value domain of i is the set of integers. 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. The advantages of a single variable are simpler syntax (fewer names), and more efficient simulation monitoring.

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,[3]
- if $o \in \mathcal{O}$ is a T-valued n-ary operator, and $\tau_1, \ldots, \tau_n$ are appropriately valued terms, then $o(\tau_1, \ldots, \tau_n)$ 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 $\tau_1, \ldots, \tau_n$ are appropriately valued terms, then $r(\tau_1, \ldots, \tau_n)$ is an LOC formula,
- if ϕ and ψ are LOC formulas, so are $\overline{\phi}$, $\phi \wedge \psi$, 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:

$$\text{val}(a[i]) = 5 \ \wedge \ \text{val}(a[i+1]) = 5$$

$$f(a[i+4]) + f(b[g(a[i])]) < 20$$

$$\overline{\text{val}(a[i]) = 0} \ \vee \ f(b[i]) = 0 \ .$$

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

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

- the variable i evaluates to any integer,
- if τ evaluates to some integer n, then $\text{val}(e[\tau])$ evaluates to $\beta(e, n)$, and $f(e[\tau])$ evaluates to $f(e, n)$,
- all other operators and relations are evaluated in the standard way if all their operands are defined, and they are undefined otherwise,

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

- Boolean functions are evaluated as in a usual three-value logic [19],
- an annotated behavior *satisfies* an LOC formula if it does not evaluate to $false$ for any value of i.

More formally, 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 $\mathcal{V}^n_{(\beta,A)}[\![\alpha]\!]$, 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 $\mathcal{V}^n_{(\beta,A)}[\![\alpha]\!]$, is in $T \cup \{undef\}$, and if α is a formula, then $\mathcal{V}^n_{(\beta,A)}[\![\alpha]\!]$ is in $\{true, false, undef\}$. Note that this implies that for some k-ary T-valued operator *o*, the formula $o(\tau_1, \ldots, \tau_k)$ 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:

- $\mathcal{V}^n_{(\beta,A)}[\![i]\!] = n$,
- $\mathcal{V}^n_{(\beta,A)}[\![c]\!] = c$ for each element c of each value domain T,
- $\mathcal{V}^n_{(\beta,A)}[\![\text{val}(e[\tau])]\!] = \begin{cases} undef \text{ if } \mathcal{V}^n_{(\beta,A)}[\![\tau]\!] = undef \text{ or } \beta(e, \mathcal{V}^n_{(\beta,A)}[\![\tau]\!]) \text{ is not defined} \\ \beta(e, \mathcal{V}^n_{(\beta,A)}[\![\tau]\!]) \text{ otherwise} \end{cases}$ for each event name e and each integer-valued term τ,
- $\mathcal{V}^n_{(\beta,A)}[\![f(e[\tau])]\!] = \begin{cases} undef \text{ if } f \notin A \text{ or } \mathcal{V}^n_{(\beta,A)}[\![\text{val}(e[\tau])]\!] = undef \\ f(e, \mathcal{V}^n_{(\beta,A)}[\![\tau]\!]) \text{ otherwise,} \end{cases}$ for each annotation f, each event name e, and each integer-valued term τ,
- $\mathcal{V}^n_{(\beta,A)}[\![o(\tau_1, \ldots, \tau_k)]\!] = \begin{cases} undef \text{ if } \mathcal{V}^n_{(\beta,A)}[\![\tau_j]\!] = undef \text{ for some } j = 1, \ldots, k \\ o(\mathcal{V}^n_{(\beta,A)}[\![\tau_1]\!], \ldots, \mathcal{V}^n_{(\beta,A)}[\![\tau_k]\!]) \text{ otherwise,} \end{cases}$ for each k-ary operator *o*,
- $\mathcal{V}^n_{(\beta,A)}[\![r(\tau_1, \ldots, \tau_k)]\!] = \begin{cases} undef \text{ if } \mathcal{V}^n_{(\beta,A)}[\![\tau_j]\!] = undef \text{ for some } j = 1, \ldots, k \\ r(\mathcal{V}^n_{(\beta,A)}[\![\tau_1]\!], \ldots, \mathcal{V}^n_{(\beta,A)}[\![\tau_k]\!]) \text{ otherwise,} \end{cases}$ for each k-ary relation r,

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

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

If we imagine representing a formula by its syntax tree, then its evaluation can be interpreted as propagating values from the leaves up. The value $undef$ can be created only at nodes of the form val$(e[\tau])$ or $f(e[\tau])$, if $f \notin A$, or τ evaluates to a negative integer or to a number larger than the number of instances of e in β. Once created, value $undef$ propagates up, and it can be stopped only at a node of the form $\phi \wedge \psi$ (if the other branch has value $false$) or $\phi \vee \psi$ (if the other branch has value $true$). Thus, it is not hard to show that the value of a formula that does not contain any terms of the form val$(e[\tau])$ or $f(e[\tau])$ must be either $true$ or $false$.

D. Typical Usage of LOC

In the following examples, we assume that the set of event names is $E = \{Display, Stimuli\}$, and that a real-valued annotation t is defined. Intuitively, we assume that $t(event[x])$ corresponds to the time of the x-th occurrence of an event $event$. [4] The following common constraints are now easy to express:

- **rate,** e.g. "$Display$'s are produced every 10 time units":

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

- **latency,** e.g. "$Display$ is generated no more than 25 time units after $Stimuli$":

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

[4] In this paper, we always use i as the index variable in an LOC formula and x to represent an arbitrary value of i.

- **jitter,** e.g. “every $Display$ is no more than 4 time units away from the corresponding tick of the real-time clock with period 10”:

$$\mid t(Display[i]) - (i+1) * 10 \mid \leq 4 \ , \tag{3}$$

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

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

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

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

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”:

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

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) 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:

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

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

E. Related Work

Real-Time Logic (RTL) [15] 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. The safety properties expressed with RTL can be analyzed efficiently [15], [20]. Based on RTL, several run-time monitoring techniques for real-time systems have been proposed [**?**], [**?**]. Different from RTL, LOC is designed for transaction level quantitative properties, which include not only timing but also power, data, memory, and so on, due to the introduction of annotations.

There are several decidable logics that can be translated into automata and formally verified. It is well known that LTL can be translated into equivelant Buchi automata. MONA system [**?**] provides a decision procedure for reasoning monadic second-order logic. But decidability is usually achieved with sacrifice of expressiveness and convenience. It is proven that only a subset of LOC is decidable (see Appendix II), but we will show it is still very useful and efficient for simulation-based assertion checking.

III. Expressiveness of LOC

In this section, we discuss the expressiveness property of LOC especially in its relationship with the well known Linear Temporal Logic (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. Linear Temporal Logic

Like LOC, Linear Temporal Logic (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 $\overline{\ }$ (not), $\vee$ (or), $\wedge$ (and), $\Longrightarrow$ (imply), and the linear temporal operators $\Box$ (always), $\Diamond$ (eventually), **X** (next) and **U** (strong until):

- $\Box$(A) is true if A is true for any state.
- $\Diamond$(A) is true if A eventually becomes true in a future state.
- **X**(A) requires that A will be true in the following state.
- A **U** B requires that B will eventually hold in a future state, and A must hold from the current state to that future state.

With these temporal operators, LTL is very powerful in expressing the functional constraints, especially the ones that contain complex temporal patterns, for example:

- **response:** $\Box$(A $\Longrightarrow$ $\Diamond$ B), i.e. once A holds, B will eventually become true sometime later.
- **precedence:** $\overline{\text{A}}$ **U** B, i.e. B will eventually hold, and B must become true before A becomes true.
- **infinite often:** $\Box$ $\Diamond$ A, i.e. A will become true infinitely often.

It has been proven that LTL formulas can be translated to equivalent Büchi automata [22]. Based on this theory, formal techniques like model checking are developed and utilized for verification of both digital designs (e.g. FormalCheck [2]) and software protocols(e.g. SPIN [10]). LTL is also widely used in the formal property specification for simulation-based assertion verification [7], [1], which is important to assure the integration and correctness of reusable IP (Intellectual Property) blocks.

B. LOC v.s. 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:

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

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:

$$\Box(Display\_occur \Longrightarrow (Display\_data > 100)) \ . \tag{9}$$

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 latency constraint:

$$t(Display[i]) - t(Stimuli[i]) \leq 25 \tag{10}$$

requires comparing each instance of $Stimuli$ with the instance of $Display$ with the same instance index. After the x-th $Stimuli$ occurs, it is unknown when the x-th $Display$ will occur, i.e. the number of $Stimuli$ instances that may occur before the x-th instance of $Display$ is arbitrarily large. Therefore, this constraint cannot be modeled by a finite-state system, and it is impossible to express it using a formalism built on finite automata such as LTL.

It is interesting to note that there are simple LOC formulas that cannot be expressed by LTL even though they can be easily represented by a finite-state automaton. For example, the property "the value of event A on every even occurrence is 1", can be expressed by LOC formula $\text{val}(A[2i]) = 1$, as well as with a simple two-state automaton, but it is well known that it cannot be expressed by LTL [23].

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 never 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. [5]

Claim 3: LOC can express only safety properties.

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

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 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 $\Box \diamondsuit A$, i.e. "A occurs infinitely often". From claims (1)-(3), we can conclude the following:

Conclusion: 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.

IV. Complexity of Verification of LOC Formulas

In this section, we address the following fundamental question: How hard is it to check if a system satisfies an LOC formula? This question has many versions, depending on how the system is represented, and which subset of LOC formulas is being considered. We present answers for several versions. Some versions of the problem are undecidable, and some are decidable, but with very complex algorithms. We use these "negative" results to justify the development of efficient algorithms which may not always give the full answer. These algorithms, based either on simulation, or partial formal verification, are described in Section V.

In the most general case, systems are represented by arbitrary programs, and annotations can be of any type. This case is clearly expressive enough to encode the halting problem [14], so checking LOC formulas is undecidable in this case.

The first restriction we consider is to limit system specification to a *infinitely-valued* finite-state system, where the number of states of the system is finite, but value domains of annotations can be infinite. Unfortunately, this case is also undecidable. To show this we can encode two

counter machines using a finite-state system, two integer annotations to represent counters, and an LOC formula to ensure that counters are incremented or decremented as necessary.

The next restriction we consider are so-called *finitely-valued* finite-state systems, where annotations and event values are required to be finitely valued. With regards to annotation specification, we will consider three cases:

1) annotations completely undefined,
2) annotation must satisfy certain axioms, expressed by an LOC formula,
3) annotations defined by a finite state system.

The third case is typical of later design stages. At that point annotations can be considered as part of event values, so we will not study it separately.

The first case is typical at the beginning of the design process, where constraints on annotations are stated, but nothing is yet known about their actual values. At that point, annotations are uninterpreted functions, but they still have to satisfy properties of equalities. For example, the formula:

$$\overline{f(e[3i]) = f(e[i+2])}$$

is not satisfied by any behavior in which e occurs at least 3 times.

We consider the second case because, even if the values of annotations are not known, some properties, captured by axioms, may be. Consider, for example, time annotations. All possible timing annotations share certain properties, e.g. time can never decrease. Just from these basic properties of time, we could deduce some system properties, which are then valid for any timing. Therefore, it is useful to be able to express properties that all annotations of certain type must have. Specifying axioms could be done in many ways. For example, an extended version of LOC is used for this purpose in the Metropolis system [4]. However, the following results state that LOC checking is undecidable even if annotation axioms are restricted to the basic LOC.

Theorem 1: It is undecidable whether a finitely-valued finite-state system with LOC axioms satisfies an LOC formula with a single event indexed by expression i.

As usual, the proof proceeds by reducing a known undecidable problem to LOC checking. The details are given in Appendix I.

At first glance, it may appear that checking an LOC formula ϕ for a finite state system with annotation axioms α may be reduced to checking that the system satisfies implication of ϕ by

α without any axioms. Unfortunately, this approach does not work, and to see why we will for a moment make quantification over i appear explicitly in the syntax. Thus, the axioms can be written as $\forall i : \alpha$ and the formula can be written as $\forall i : \phi$. Solving the problem requires checking $(\forall i : \alpha) \Longrightarrow (\forall i : \phi)$, but LOC can only express $\forall i : (\alpha \Longrightarrow \phi)$, which is not the same. In fact, this seemingly minor restriction makes the problem decidable, as stated by Theorem 2.

We now turn our attention to the case without axioms, i.e. annotations are either completely unconstrained, or folded into event values.

Theorem 2: It is decidable whether a finitely-valued finite-state system without annotation axioms satisfies an LOC formula in which all index expressions are of the form $ai + b$, where a and b are integer constants, and variable i appears only in such expressions and linear inequalities.

The proof consists of a decision algorithm. To describe the algorithm, we need some notation. An *event expression* is an LOC term of the form $\mathrm{val}(e[\tau])$, or of the form $f(e[\tau])$, where τ is an integer-valued term, e is an event name, and f is an annotation. Note that conditions in Theorem 2 restrict τ to be a linear expression, i.e. it must be of the form $ai + b$, where a and b are constants. The *value domain* of an event expression is the set of values it can take, i.e. it is the value domain of e if the expression is of the form $\mathrm{val}(e[\tau])$, and it is the value domain of f if the expression is of the form $f(e[\tau])$.

Given an LOC formula ϕ, we use $\mathcal{E}_\phi$ to denote the set of event expressions appearing in it. An *interpretation* of a set of event expressions is a function which assigns to each expression in the set a value from its value domain. Since Theorem 2 requires the system to be finitely-valued, there can be only finitely many distinct interpretations of $\mathcal{E}_\phi$. Given an LOC formula ϕ, and an interpretation I of $\mathcal{E}_\phi$, we use ϕ_I to denote the formula obtained from ϕ by replacing each event expression ϵ in ϕ by the value $I(\epsilon)$. We call ϕ_I an interpretation of ϕ. Note that because ϕ_I contains no event expressions, $\mathcal{V}^n_{(\beta,A)}[\![\phi_I]\!]$ actually depends only on n and must be either $true$ or $false$.

The conditions of Theorem 2 also insure that ϕ_I is a formula in *Presburger arithmetic*. Such formulas consist of linear inequalities of integer variables combined with usual Boolean connectives and quantification of variables [8]. Presburger formulas can be evaluated to $true$ or $false$ by choosing values for all free integer variables. LOC formula interpretations can have only i as a free variable, and we will use $\phi_I(n)$ to denote the value of ϕ_I when i is set to n.

Assume, for example, a system with two binary events, x_1 and x_2, and let ϕ be the formula:

$$(\text{val}(x_1[3i]) = \text{val}(x_2[i])) \Longrightarrow (i \geq 5) \quad . \tag{11}$$

It has two binary event expressions, val$(x_1[3i])$ and val$(x_2[i])$, hence it has four interpretations. To denote interpretations, we use 00, 01, 10, and 11, where the first number represent the value of val$(x_1[3i])$, an the second number represents the value of val$(x_2[i])$. It is easy to check that $\phi_{01} = \phi_{10} = true$ and $\phi_{00} = \phi_{11} = (i \geq 5)$.

It is not hard to check that LOC formula interpretations have the following property:

$$\Big(\forall \epsilon \in \mathcal{E}_\phi : \mathcal{V}^n_{(\beta,A)}[\![\epsilon]\!] = I(\epsilon)\Big) \Longrightarrow \Big(\mathcal{V}^n_{(\beta,A)}[\![\phi]\!] = \mathcal{V}^n_{(\beta,A)}[\![\phi_I]\!] = \phi_I(n)\Big) \quad . \tag{12}$$

In words, if behavior (β, A) and integer n agree with interpretation I on the values of all event expressions, then they agree also on the value of the whole formula. In addition, formula ϕ_I is both a Presburger formula (because it has no events nor indexing) and an LOC formula (because it has no quantifiers and its only free variable is i), so it may be evaluated in both ways, but the two values are always the same.

To check whether a system satisfies an LOC formula, we will combine formula interpretations with Presburger formulas characterizing the system, and we will reduce the original problem to checking satisfiability of the combined formula. That will complete the proof, as there are known algorithms to check satisfiability of a Presburger formula. In the following Lemma, we establish that it is indeed possible to construct a Presburger formula characterizing a finitely-valued finite-state system. The construction is described in Appendix I.

Lemma 1: For a given finitely-valued finite-state system with no annotation axioms, and a given LOC formula ϕ, it is possible to construct, for each interpretation I of $\mathcal{E}_\phi$, a Presburger formula SYS_I in which i is the only free variable, such that for all integers n, $SYS_I(n)$ is true if and only if there exists an annotated behavior (β, A) of the system such that $\mathcal{V}^n_{(\beta,A)}[\![\epsilon]\!] = I(\epsilon)$ for all $\epsilon \in \mathcal{E}_\phi$.

Consider, for example, the system shown in Figure 2. It has eight states, two binary valued events, x_1 and x_2, and no annotations. A transition label of the form $x_k : v$ indicates that x_k is generated with value v on that transition. The system in Figure 2 satisfies formula (11), because $x_1[3i]$ is always 1, and $x_2[i]$ is 0 for all $i < 5$. With respect to interpretations of (11), one can

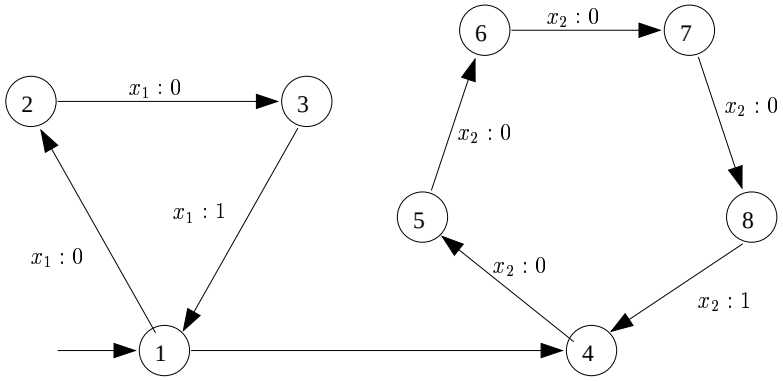

Fig. 2. A system generating 1 for every third value of x_1 and every fifth value of x_2.

easily verify that $SYS_{00} = SYS_{01} = false$, because $x_1[3i]$ is never 0, and SYS_{11} and SYS_{10} are[6] $(\exists j > 0 : i = 5j)$ and $(i > 0) \wedge \overline{(\exists j : i = 5j)}$ respectively, because every fifth values of $x_2[i]$ is 1.

Theorem 3: For a given finitely-valued finite-state system with no annotation axioms, and a given LOC formula ϕ, let formulas SYS_I satisfy the property from Lemma 1, for each interpretation I of $\mathcal{E}_\phi$. The system satisfies ϕ if and only if the the following Presburger formula is *not* satisfiable:

$$\bigvee_I SYS_I \wedge \overline{\phi}_I \ , \tag{13}$$

where the finite disjunction ranges over all interpretations of $\mathcal{E}_\phi$.

To show one direction, assume that the system does not satisfy the property, i.e. assume that there exists an annotated behavior (β, A), and an integer n such that, $\mathcal{V}^n_{(\beta,A)}[\![\phi]\!] = false$, or equivalently $\mathcal{V}^n_{(\beta,A)}[\![\overline{\phi}]\!] = true$. Let I be the interpretation induced by (β, A) and n, i.e. set $I(\epsilon)$ to $\mathcal{V}^n_{(\beta,A)}[\![\epsilon]\!]$ for all $\epsilon \in \mathcal{E}_\phi$. By Lemma 1, $SYS_I(n)$ is true, and by (12) so is $I(\overline{\phi})(n)$, so the formula is satisfiable.

For the other direction, assume that the formula is satisfiable, and let I and n be such that both $SYS_I(n)$ and $I(\overline{\phi})(n)$ are true. By Lemma 1, there exists an annotated behavior (β, A)

[6]We use $\exists j > 0 : \phi$ to abbreviate $\exists j : (j > 0) \wedge \phi$.

such that $\mathcal{V}^n_{(\beta,A)}[\![\epsilon]\!] = I(\epsilon)$ for all $\epsilon \in \mathcal{E}_\phi$, and by (12) $\mathcal{V}^n_{(\beta,A)}[\![\overline{\phi}]\!] = I(\overline{\phi})(n) = true$, implying that $\mathcal{V}^n_{(\beta,A)}[\![\phi]\!] = false$, i.e. the system does not satisfy the property.

For example, the negation formula (11) has the following interpretations: $\overline{\phi}_{01} = \overline{\phi}_{10} = false$ and $\phi_{00} = \phi_{11} = (i < 5)$, so for the system is Figure 2, formula (13) becomes

$$(\exists j > 0 : i = 5j) \wedge (i < 5) \ ,$$

which is clearly not satisfiable.

Theorem 3 provides a constructive way of reducing the original problem to satisfiability of a Presburger formula. Theorem 2 then follows as a simple corollary. The described algorithm proves decidability, but it has a very high complexity. The number of interpretations may be exponential in the size of formula, and the best known algorithm for checking satisfiability of Presburger formulas is doubly exponential in the worst case. There may be cases in practice that are much better than the worst case, but it is still unlikely that the proposed algorithm will have a wide-spread use. It is therefore reasonable to search for alternative, more efficient verification algorithms, applicable to some reasonable subset of LOC. In Section V, we will propose a couple of approaches along these lines. But first we show that several approaches that one may consider are in fact not feasible.

Each LOC formula defines a language consisting of annotated behaviors that it satisfies. If we could construct an automaton with the same language, we could reduce LOC verification to the language containment problem, which has known algorithms linear in the number of states of the system and the property automaton. Indeed, this approach is possible for a very limited subset of LOC (as shown in Section V), but languages of many simple LOC formulas cannot be represented by a finite-state automaton. Here are a few example:

- two events, all index expression just i, e.g.

$$\text{val}(x[i]) = \text{val}(y[i]) \ ,$$

- a single event, all index expressions linear, e.g.

$$\text{val}(x[i]) = \text{val}(x[2i]) \ ,$$

- a single event, and a single event expression, e.g.

$$\text{val}(x[i^2]) = 1 \ .$$

In the examples above we assume all events to be finitely valued. Still, it is not hard to show, using the pumping lemma for regular sets [14], that none of the formulas above define a regular language. Note that first two examples satisfy the conditions of Theorem 2 and could be checked with the proposed algorithm.

Another approach might be to use a class of automata that is more expressive than finite-state ones. For example one may consider pushdown automata that can define context-free languages. Unfortunately, this is not possible in general, either. For example, if event x takes values from $\{0, 1, 2, 3\}$, the formula:

$$\begin{aligned}
&(\text{val}(x[i]) = 0 \implies (\text{val}(x[i+1]) = 0 \vee \text{val}(x[i+1]) = 1)) \wedge \\
&(\text{val}(x[i]) = 1 \implies (\text{val}(x[i+1]) = 1 \vee \text{val}(x[i+1]) = 2)) \wedge \\
&(\text{val}(x[i]) = 2 \implies (\text{val}(x[i+1]) = 2 \vee \text{val}(x[i+1]) = 3)) \wedge \\
&(\text{val}(x[i]) = 3 \implies (\text{val}(x[i+1]) = 3)) \wedge \\
&((\text{val}(x[i-1]) = 0 \wedge \text{val}(x[i]) = 1) \implies \\
&\quad (\text{val}(x[2i-1]) = 1 \wedge \text{val}(x[2i]) = 2 \wedge \\
&\quad \text{val}(x[3i-1]) = 2 \wedge \text{val}(x[3i]) = 3))
\end{aligned}$$

defines the language:

$$\{s : s \text{ is a prefix of } 0^n 1^n 2^n 3^* \text{ for some } n \geq 0\} \ ,$$

for which it is easy to show that it is not context-free (e.g. see Example 6.1 in [14]).

One approach to generating an automaton for an LOC formula is to buffer event values. Once all the values needed to evaluate the formula for a particular value of i are in the buffer, the formula can be evaluated for that value of i. Once all values of i that need a particular event value are evaluated, the event value can be removed from the buffer. The results above indicate that the buffer sizes cannot be bounded in general. However, one may hope that for a specific finite-state system, a suitable bound can be found. Ideally, a bound may be found for any finite-state system.

For example, any implementation of a FIFO queue needs to satisfy the data consistency property (6) , i.e. the i-th value retrieved from the FIFO must match the i-the value put into it. Clearly, we cannot represent this property with a finite-state automaton, as we cannot bound in general the difference between the number of $input$ events and the number of $output$ events. However, for any particular FIFO implementation, this bound can be easily established, it is just

the size of the FIFO. Thus, the size of the buffer in the checking automaton need not be bigger than the size of the FIFO. One may hope that this reasoning generalizes to any similar property and any finitely-valued finite-state system.

To the best of our knowledge, it is not known whether a bound on buffers can be found for any finite-state systems. However, we will use an example to show that even if such a bound can be found, it will sometimes be too big for an efficient verification algorithm. In general, the example is a finitely-valued finite-state system that may generate n different binary events $x_1, \ldots, x_n$, and has $p_1 + \cdots + p_n$ states, where $p_1, \ldots, p_n$ are first n primes. The system has n loops, and the k-th loop has p_k states. The system first circles through the first p_1 states, generating x_1 with value 0 $p_1 - 1$ times followed by generating x_1 with value 1 once. At the end of the loop there is a choice of repeating it or moving to the next loop. The system in Figure 2 is actually a part of such a system for $p_2 = 3$ and $p_3 = 5$. The language generated the system with n loops consists of all prefixes of strings defined by regular expression:

$$(x_1 : 0^{p_1-1}\ x_1 : 1)^+\ (x_2 : 0^{p_2-1}\ x_2 : 1)^+\ \ldots (x_n : 0^{p_n-1}\ x_n : 1)^+\ .$$

Now, consider the LOC formula $\overline{\text{val}(x_1[i]) = \text{val}(x_2[i]) = \cdots = \text{val}(x_n[i]) = 1}$. (For readability and conciseness, we abbreviate formulas of the type $\tau_1 = \tau_2\ \wedge\ \tau_2 = \tau_3$ to $\tau_1 = \tau_2 = \tau_3$.) It is not satisfied, but the smallest value of i that violates it is $p_1 * p_2 * \cdots * p_n$. Since the system generates all x_1's before generating any other events, all $p_1 * p_2 * \cdots * p_n$ values of x_1 (and $x_2, \ldots, x_{n-1}$ for that matter) would have to be buffered. Therefore, the size of the buffer have to be at least exponential in the number of states of the checked automaton, implying that the number of states of the checking automaton has to be at least doubly exponential. More practical approaches are needed.

V. Verification Approaches for LOC Formulas

In this section, we first propose a simulation-based trace analysis approach, and show that LOC constraints can be easily analyzed in an assertion-based simulation verification environment. Then, we discuss how to utilize the existing formal verification technique, i.e. model checking, to verify an LOC formula.

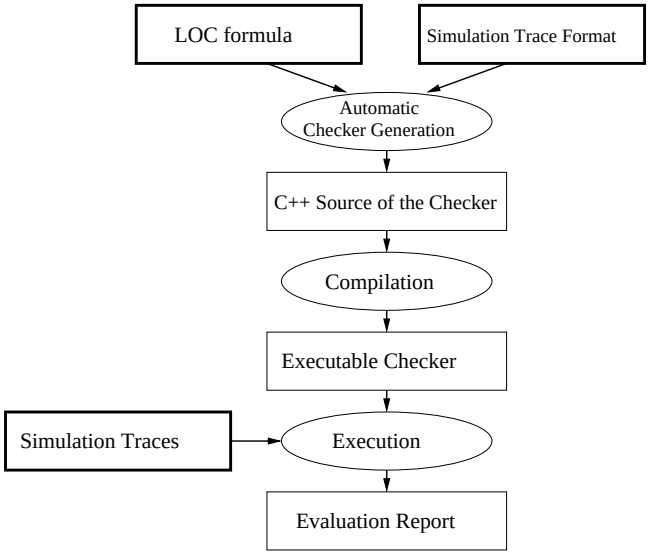

Fig. 3. Trace Analysis Methodology.

A. Trace Analysis Methodology

The methodology for simulation-based verification with an automatically generated LOC checker is illustrated in Figure 3. From the specification of LOC formulas and trace formats, an automatic checker generator is used to generate a C++ source of the 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 Figure 4. 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 Figure 8. 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 a 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

```
[LOC: rate]
    formula: t(Display[i + 1] − t(Display[i]) == 10
    annotation: event value t
    trace: "%s : %d at time %f"
[LOC: latency]
    formula: t(Display[i])−t(Stimuli[i]) <= 25
    annotation: event value t
    trace: "%s : %d at time %f"
```

Fig. 4. Definition of LOC formulas and Trace Formats.

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

Fig. 5. Example of Error Report.

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. Figure 5 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 of details: the algorithm of the LOC checking, the runtime monitoring and how to deal with memory limitation.

a) The 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, ... etc. A formula instance is a formula with i evaluated to some fixed positive integer value, e.g. $Display[30] - Display[29] = 10$ 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 formula instance i) {
```

```
    evaluate formula instance i;
    i++;
    memory recycling;
  } } .
```

The time complexity of the algorithm is linear to 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).[7] 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:

$$t(Display[i+10]) - t(Stimuli[i+5]) < 300 \tag{14}$$

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 index less than 111 and event $Stimuli$'s annotation t with 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 allows 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

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

could then be evaluated according to the standard three-value logic [19] evaluation. For example, given the following LOC formula:

$$t(Display[i+10]) > 100 \ \vee \ t(Stimuli[i-5]) < 300 \ , \tag{15}$$

let the current value of i be 10. If we know, from the trace, that the value of $t(Display[20])$ is 200, the formula can already be evaluated to be $true$ even if the value of $t(Stimuli[5])$ 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 $Stimuli$. The expression $t(Stimuli[-1]) < 300$ is evaluated to $undef$ and the whole formula can be evaluated to $true$ if the evaluation of $t(Display[14]) > 100$ 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's 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 [3]. 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 VI-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 can not 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 VI-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.

B. Partial Formal Verification of LOC Formulas

Although our trace analysis enables efficient verification of LOC formulas in a simulation environment, formal verification may still be necessary to formally prove properties of library modules (e.g. the TTL channel in Section VI-C) and other small designs. The simulation approach described above 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. We believe that there is no inherent difficulty to automate this partial formal verification process for LOC formulas, though it has not been done yet.

Model checking tools usually reduce this *language containment* problem to reachability analysis of the state space that includes states of both the system and the trace checker. Unfortunately, model checkers can typically deal only with finite state spaces. A checker for an LOC formula can be interpreted as a finite state automaton only if the size of the queue it uses can be fixed *a priori*. This is not always the case, as exemplified by the trace checker for the constraint (2). On the other hand, many LOC formulas do have corresponding finite-state trace checkers. For example, the rate constraint:

$$t(Display[i+1]) - t(Display[i]) = 10 \tag{16}$$

compares the annotation t of any two consecutive occurrences of the event $Display$. To check this formula, the trace analysis algorithm (see Section V-A) only needs to store the annotation t of two consecutive occurrences of $Display$ at any given time, i.e. only a constant amount of memory is needed.

From the above discussion, we give the following conservative rule to decide if the checker for a particular LOC formula can be expressed by a finite-state automaton.

Rule 1: If an LOC formula satisfy the following conditions, then it has a corresponding finite-state trace checker:

(a) the index variable i appears only in index expressions (ruling out, for example, the jitter constraint (3)),

(b) all index expressions index the same event, (ruling out, for example, the latency constraint (2)),

(c) all index expressions are linear expressions in i (ruling out, for example, the formula val$(A[i^2]) = 1$), and the difference between any two of them is a constant, i.e. they all have the same i coefficient, but possibly different constant coefficients (ruling out, for example, the formula val$(A[i])$ = val$(A[2i])$).

Although Rule 1 may appear quite restrictive, still many interesting properties satisfy it, including rate (1) and throughput (4) formulas.

Let n be the maximum difference between two index expressions in a given formula satisfying Rule 1, and let m_i be the largest of all index expressions evaluated for a particular value of i. Evaluating the formula for any value of i requires knowing annotations of at most $n + 1$ consecutive occurrences of the indexed event. Thus, if the trace checker maintains a list of $n+1$ most recent annotations of the indexed event, the value of the formula for some value of i can be computed as a state predicate after the m_i-th occurrence of the indexed event.

For example, for the rate constraint (16), n is 1, and m_i are $2, 3, \ldots$ for $i = 1, 2, \ldots$. Assuming that variables $Display\_t$ and $Display\_t\_last$ are used to store the values of the annotation t for the current and last instances of $Display$ respectively, and that Boolean variable $Display\_occur$ is $true$ whenever $Display$ occurs, except for the first time (first occurrence must be skipped since m_i is never 1), we can convert the rate constraint (16) into the state predicate:

$$Display\_occur \Longrightarrow Display\_t - Display\_t\_last = 10 \ . \tag{17}$$

Note that state predicates can be easily converted into LTL formulas by prefixing them with the *always* operator $\Box$.

To formally verify formulas not satisfying Rule 1, we limit checkers to finite memory, 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 counter-example is found showing that the system does not satisfy the property,
- the property is satisfied, all reachable state are searched without finding a counter-example or reaching a state where memory is exhausted,
- inconclusive, reachability analysis finds no counter-examples, but states where memory is exhausted are reachable.

For example, the latency constraint:

$$t(Display[i]) - t(Stimuli[i]) < 25 \tag{18}$$

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 $Display[x]$ always occurs after $Stimuli[x]$). 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 (18) holds.

Obviously, if **P1** and **P2** both hold then so does (18), and if **P2** is $false$, so is (18). However, if **P2** holds, but **P1** does not, the result is inconclusive.

To specify **P1** and **P2**, assume that the trace checker keeps 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 arrays. Also assume that variable $Display\_i$ 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 $Display\_occur$ and $Stimuli\_occur$ are $true$ when $Display$ and $Stimuli$ occur, respectively. Then, **P1** can be specified with the following state predicate:

$$Stimuli\_occur \implies (Stimuli\_i \neq Display\_i) \ . \tag{19}$$

Since we assume that $Display$ always follows $Stimuli$, the condition where $Display\_i$ equals $Stimuli\_i$ just after $Stimuli$ occurs, indicates the buffer overflow. Constraint (18) can be

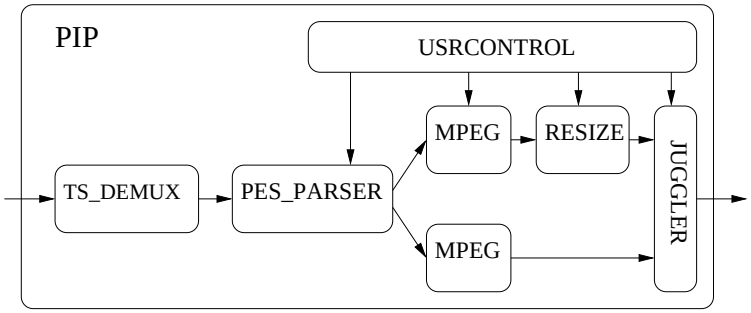

Fig. 6. Picture-In-Picture Design.

expressed as follows:

$$Display\_occur \implies Display\_t[Display\_i] - Stimuli\_t[Display\_i] < 25 \ , \tag{20}$$

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

$$Assumption\ (19) \implies Formula\ (20) \ . \tag{21}$$

VI. Case Studies

We apply our LOC-based verification methodologies to three design examples. The first is a system level design for set-top video processing, *Picture-In-Picture (PIP)*, which is originally specified with YAPI [18]. PIP is partially respecified and simulated with Metropolis environment [4]. The second is an RTL model of a *Finite Impulse Response (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. The third is a *Task Transition Level (TTL)* [9] refinement of a YAPI channel design. We use both trace analysis and formal verification techniques to verify the data consistency constraints of the TTL channel, and show how the formal verification approach works on checking important library modules.

A. Picture-In-Picture

Figure 6 shows the PIP design. TS_DEMUX demultiplexes the single input transport stream (TS) into multiple packetized elementary streams (PES). PES_PARSER parses the packetized elementary streams 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 that combines the images to produce the picture-in-picture videos. The entire description consists of approximately 19,000 lines of Metropolis and YAPI code. With the sample input stream we used, it produced 120,000 lines of output representing header information for the processed frames.

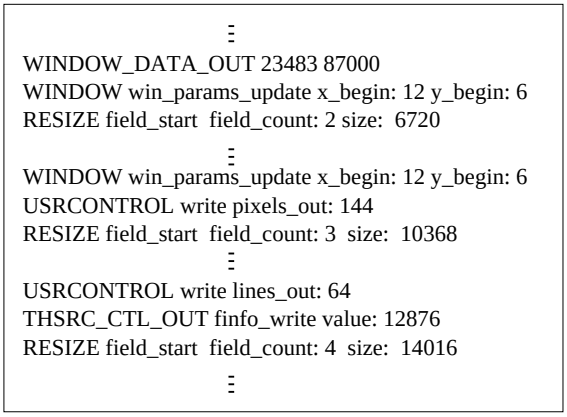

Fig. 7. PIP Simulation Trace.

At the system level, we can use LOC to specify the functional properties. 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 2 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:

$$size(field\_start[2i+2]) - size(field\_start[2i+1]) = size(field\_start[2i+1]) - size(field\_start[2i]) \quad , \tag{22}$$

where $field\_start$ is an event at which RESIZE starts to output a new image field. The annotation $size$ is the total number of pixels processed by RESIZE. Figure 7 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 minute 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:

$$field\_count(in[i]) = field\_count(out[i]) \ . \tag{23}$$

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 $field\_count$ 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 minute 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 specifies 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 ($change\_size$):

$$field\_count(change\_size[i]) - field\_count(read\_size[i]) \leq 5 \ , \tag{24}$$

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 $field\_count$ 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 minute of CPU time.

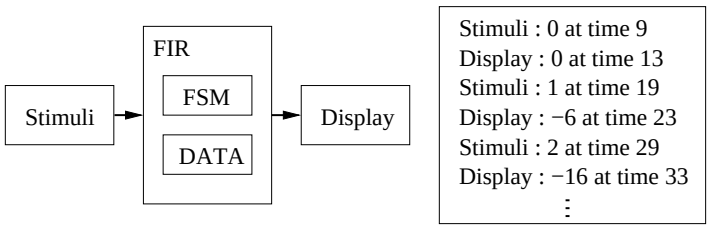

Fig. 8. FIR Design and Simulation Trace.

B. FIR filter

Figure 8 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 same trace files are used for all the analyses 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.

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

The simulation times for these traces are listed in Table II. Given the large file size, runtime monitoring (see Section V-A.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. For the trace size of 100 million lines, the static checking approach requires 1404 seconds of simulation time and 1229 seconds of checking time for a total of 2633 seconds. Runtime monitoring technique requires only 1420 seconds for both simulation and monitoring.

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 Runtime Monitoring (s)	1	14	148	1404
Static Trace Checking Only (s)	1	12	120	1229
Simulation w/ Runtime Monitoring (s)	2	14	145	1420

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 V-A.c. The result is shown in Table III where the memory usage is limited to 50KB. 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 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.

TABLE III

COSTS OF CHECKING CONSTRAINT (7) ON FIR

Lines of Trace ($\times 10^4$)		2	3	4	5
Unlimited Memory	Time(s)	<1	<1	<1	1
	Mem(KB)	40	60	80	100
Mem Limit (50KB)	Time(s)	<1	61	656	1869
	Mem(KB)	40	50	50	50

C. Simulation Trace Analysis for TTL Channel

Y-chart Application Programming Interface (YAPI) is a model of computation for designing signal processing systems [18]. It is basically a Kahn process network [16], extended with the ability to non-deterministically 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 [5]. It includes refinement of the YAPI channel into a lower-level abstraction called *Task Transition Level (TTL)* [9]. The refinement is shown in Figure 9.

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 Figure 9 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. We first check a property that is suitable for both LOC and LTL, “the data read by *Sum* is always greater than or equal to 0”, and we call it “non-negative” property. Another 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

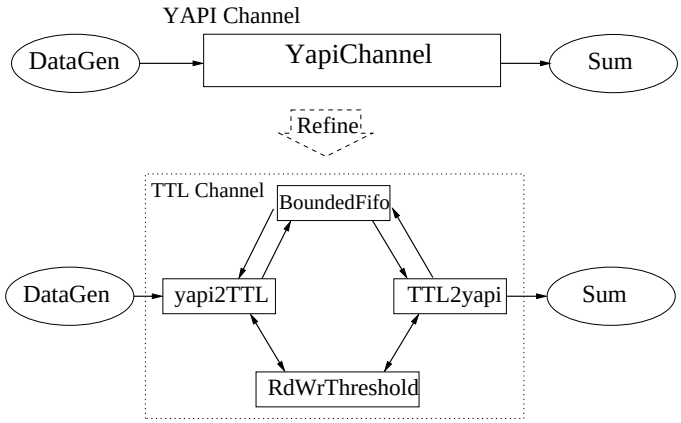

Fig. 9. YAPI Channel and TTL Channel.

read from the channel in exactly the same order without a loss. In the rest of this section, we apply both assertion checking and formal verification techniques on the TTL channel for these two properties.

The TTL channel shown in Figure 9 is initially specified in Metropolis Meta-Model (MMM) [4]. We simulate it in the Metropolis environment, and produce simulation traces with different lengths. 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. The non-negative constraint is defined in LOC as:

$$data(processed[i]) \geq 0 \ , \tag{25}$$

and the data consistency constraint is defined as:

$$data(prepared[i]) = data(processed[i]) \ . \tag{26}$$

The automatic checker generator is used to parse the definition file (see Figure 10) for the trace format and LOC formulas, and generate a C++ source for the trace checker. After compilation, we use the executable checker to verify that both of the LOC formulas (25) and (26) hold on traces of 10^5 to 10^8 lines. The time and memory usage of the trace analysis are shown in Table IV.

```
[LOC: non_negative]
    formula: data(processed[i]) >= 0
    annotation: event data
    trace: "Data is %s %d"
[LOC: consistency]
    formula: data(prepared[i ]) == data(processed[i])
    annotation: event data
    trace: "Data is %s %d"
```

Fig. 10. Definition of the Trace Format and LOC formulas

TABLE IV

COSTS OF TRACE ANALYSIS FOR THE CONSTRAINTS (25) AND (26)

Lines of Trace		10^5	10^6	10^7	10^8
Formula (25)	Time(s)	<1	5	44	432
	Mem(Bytes)	4	4	4	4
Formula (26)	Time(s)	<1	8	84	767
	Mem(Bytes)	172	172	176	172

D. Formal Verification for TTL Channel

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 [12], 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. In the Promela code, We use Boolean variables $prepared\_occur$ and $processed\_occur$ to indicate the conditions that instances of $prepared$ and $processed$ occur, respectively. The code blocks, which manipulate the auxiliary data structures, are embedded into the Promela code appropriately. Thus, the non-negative constraint (25) can be expressed in LTL as:

$$\Box(processed\_occur \Longrightarrow processed\_data > 0) \ , \tag{27}$$

where $processed\_data$ stores the most recent data read by *Sum*. With the bitstate technique [13], SPIN verifies the LTL formula (27) within 2 hours on our 1.5GHz Athlon machine with 1GByte

of memory. The same setup is used for all case studies in this section. All the relevant verification parameters are listed in Table V.

From the discussion in Section V-B, we know that the data consistency constraint (26) 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*". [8] 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 $prepared\_i$ and $processed\_i$ (which take values of 0 to 31) to keep the index of the most recent data in the arrays. The assumption is written in LTL as:

$$\Box(prepared\_occur \Longrightarrow prepared\_i \neq processed\_i) \ , \tag{28}$$

and it is verified to hold by SPIN (see Table V). The data consistency constraint is written in LTL as:

$$\Box(processed\_occur \Longrightarrow prepared\_data[processed\_i] = processed\_data[processed\_i]) \ . \tag{29}$$

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

$$Assumption\ (28) \Longrightarrow Constraint\ (29) \tag{30}$$

is verified to hold by SPIN, and all the relevant verification parameters are also listed in Table V.

VII. Conclusions

In this paper, we discuss the verification aspects of the quantitative constraint formalism, Logic of Constraints. 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, on which the traditional property specification languages are based. We analyze complexity of LOC verification, 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.

[8] This assumption is derived from the actual buffer size of the TTL channel.

TABLE V

SUMMARY OF FORMAL VERIFICATION FOR THE LTL FORMULAS (27), (28) AND (30)

LTL Formula	(27)	(28)	(30)
Depth reached	48669	51257	57221
States stored ($\times 10^8$)	2.21872	2.2431	2.3156
State transitions ($\times 10^8$)	2.86427	2.85523	3.09726
Total memory (MB)	747.936	735.098	819.517
CPU time	1h37m24s	1h37m55s	3h03m18s
Hash factor	4.83946	4.78686	4.63699

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 non-deterministic simulation, non-determinism 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 non-determinism 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. Lastly, we plan to extend the LOC formalism with the universal quantifier $\forall$ and existential quantifier $\exists$, so that we can express more interesting constraints.

REFERENCES

[1] Openvera assertions white paper. *Synopsys, Inc*, 2002.

[2] http://www.cadence.com/products/formalcheck.html, 2003.

[3] http://www.systemc.org, 2003.

[4] F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, M. Sgroi, and Y. Watanabe. Modeling and designing heterogeneous systems. *Technical Report 2001/01 Cadence Berkeley Laboratories*, Nov. 2001.

[5] J. Brunel, E. A. de Kock, W. M. Kruijtzer, H. J. H. N. Kenter, and W. J. M. Smits. Communication refinement in video systems on chip. In *Proceedings of the* 7^{th} *International Workshop on Hardware/Software Codesign*, pages 142–146, 1999.

[6] X. Chen, F. Chen, H. Hsieh, F. Balarin, and Y. Watanabe. Formal verification of embedded system designs at multiple levels of abstraction. In *Proceedings of International Workshop on High Level Design Validation and Test*, pages 125–130, Oct. 2002.

[7] C. Eisner and D. Fisman. Sugar 2.0 proposal presented to the accellera formal verification technical committee. Mar. 2002.

[8] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, Inc., 1972.

[9] O. Gangwal, A. Nieuwland, and P. Lippens. A scalable and flexible data synchronization scheme for embedded hw-sw shared-memory systems. In *Proceedings of International Symposium on System Synthesis*, Oct. 2001.

[10] P. Godefroid and G. J. Holzmann. On the verification of temporal properties. In *Proceedings of IFIP/WG6.1 Symposium on Protocols Specification, Testing, and Verification*, June 1993.

[11] J. P. Hayes. *Computer Architecture and Organization.* McGraw-Hill, 1988.

[12] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–258, May 1997.

[13] G. J. Holzmann. An analysis of bitstate hashing. *Formal Methods in Systems Design*, 13(3):289–307, Nov. 1998.

[14] J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages, and computation.* Addison Wesley, 1979.

[15] F. Jahanian and A. K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, pages 890–904, 1986.

[16] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of IFIP Congress*, pages 471–475. North Holland Publishing Company, 1974.

[17] 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. on Computer-Aided Design*, 19(12):1523–1543, Dec. 2000.

[18] E. d. Kock, G. Essink, W. Smits, P. v. d. Wolf, J. Brunel, W. Kruijtzer, P. Lieverse, and K. Vissers. Yapi: application modeling for signal processing systems. In *Proceedings of the* 37^{th} *Design Automation Conference*, June 2000.

[19] E. J. McCluskey. *Logic Design Principles*. Prentice Hall, 1986.

[20] O. Millet. Multicycles and rtl logic satisfiability. In *Formal Techniques in Real-Time and Fault Tolerant Systems*, pages 73–86.

[21] A. Pnueli. The temporal logic of programs. In *Proceedings of the* 18^{th} *IEEE Symposium on Foundation of Computer Science*, pages 46–57, 1977.

[22] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. *Logics for Concurrency. Structure versus Automata, LNCS Vol 1043, Springer-Verlag*, pages 238–266, 1996.

[23] P. Wolper. Temporal logic can be more expressive. *Information and Control*, (56):72–99, 1983.

APPENDIX I

Before we present the proofs of Theorem 1 and Lemma 1 (in Section IV), we need to define systems that we are dealing with. Formally, a finitely-valued finite-state system is a sextuple $(S, S_0, T, E, G, \alpha)$ where:

- S is the set of *states* that must be finite,
- $S_0 \subseteq S$ is the set of *initial* states,
- $T \subseteq S \times S$ is the *transition relation*,
- E is the set of event names such that for each $e \in E$ the value domain $V(e)$ is finite,
- $G : \{(e, v) : e \in E, v \in V(e)\} \mapsto 2^T$ is the *generation function*,
- *annotation axiom* α is an LOC formula that may refer to values of event in E, but also to some annotations. Value domains of all the annotations appearing in α must be finite.

We use $G(e)$ as an abbreviation of $\bigcup_{v \in V(e)} G(e, v)$. Intuitively, $G(e, v)$ is the set of transitions on which e is generated with value v.

An annotated behavior (β, A) is in the set of behaviors of the system $(S, S_0, T, E, G, \alpha)$ if it satisfies α, and there exists a (possibly finite) sequence of states $s_0, s_1, \ldots$ such that:

- $s_0 \in S_0$,
- $(s_{i-1}, s_i) \in T$ for all $i > 0$ for which s_i exists,
- for all $e \in E$, all transitions (s_{i-1}, s_i), and all positive integers n: if e is generated on (s_{i-1}, s_i) for the n-th time, then it must be possible to generate the value $\beta(e, n)$ on that transition, i.e. if it holds that:

$$(s_{i-1}, s_i) \in G(e) \ ,$$
$$n = \left|\{j : 1 \leq j \leq k, (s_{j-1}, s_j) \in G(e)\}\right| \ ,$$

  then the following must also hold:

$$(s_{i-1}, s_i) \in G(e, \beta(e, n)) \ .$$

A. Proof of Theorem 1

We will reduce the *Post Correspondence Problem (PCP)* [14] to checking whether a finitely-valued finite-state system with LOC annotation axioms satisfies an LOC formula. Recall that a PCP instance is given by two ordered lists of strings, $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$. The question is whether there is a sequence of integers $i_1, \ldots, i_k$ (all form 1 to n) such that strings $a_{i_1} a_{i_2} \ldots a_{i_k}$ and $b_{i_1} b_{i_2} \ldots b_{i_k}$ are the same.

We now describe the system used in the reduction. The states of the system are 4-tuples (i_a, j_a, i_b, j_b) where i_a and i_b range from 1 to n, j_a ranges between 1 and the length of the string a_{i_a}, and j_b ranges between 1 and the length of the string b_{i_b}. Initial states are those where $j_a = j_b = 0$. In addition, there is a special state denoted by $DONE$. The system has two events a and b, both valued from 0 to n. Informally, the system moves into (i_a, j_a, i_b, j_b) after it sees the j_a-th letter of a_{i_a}, which must also be the j_b-th letter of b_{i_b}. Formally, the transitions in the system are the following:

- From $(i_a, j_a - 1, i_b, j_b - 1)$ to (i_a, j_a, i_b, j_b) if j_a-th letter in string a_{i_a} is the same as j_b-th letter in string b_{i_b}. If $j_a = j_b = 0$, then event a with value i_a and event b with value i_b are generated. Otherwise, no events are generated on this type of transitions.
- From $(i_a, j_a - 1, i_b, j_b)$ to $(i_a, j_a, i'_b, 1)$ if b_{i_b} has j_b letters, and j_a-th letter in string a_{i_a} is the same as the first letter in string $b_{i'_b}$. A b event with value i'_b is generated on this type of transitions.
- From $(i_a, j_a, i_b, j_b - 1)$ to $(i'_a, 1, i_b, j_b)$ if a_{i_a} has j_a letters, and j_b-th letter in string b_{i_b} is the same as the first letter in string $a_{i'_a}$. An a event with value i'_a is generated on this type of transitions.
- From (i_a, j_a, i_b, j_b) to $(i'_a, 1, i'_b, 1)$ if a_{i_a} has j_a letters, b_{i_b} has j_b letters, and the first letters in strings $a_{i'_a}$ and $b_{i'_b}$ are the same. An a event with value i'_a, and a b event with value i'_b are generated on this type of transitions.
- From (i_a, j_a, i_b, j_b) to $DONE$ if a_{i_a} has j_a letters, and b_{i_b} has j_b letters. Events a and b are generated on this type of transitions, both with value 0.

The system has a single binary annotation called $good$, and the annotation axiom is:

$$good(a[i]) \iff (\text{val}(a[i]) = \text{val}(b[i]) \wedge ((i = 1) \vee good(a[i-1]))) \quad .$$

PCP has a solution if and only if the system above does not satisfy the LOC formula:

$$\overline{\text{val}(a[i]) = 0} \ .$$

Indeed the formula above is violated if and only if there is a path in the system from some initial state to $DONE$, such that along this path a and b are generated the same number of times (say $k+1$), and the first k values of of a and b are not only equal but also larger than 0. If $i_1, \ldots, i_k$ denotes those values, then it is not hard to check that strings $a_{i_1} a_{i_2} \ldots a_{i_k}$ and $b_{i_1} b_{i_2} \ldots b_{i_k}$ are the same.

We have just shown that PCP can be reduced to checking whether a finitely-valued finite-state system with LOC annotation axioms satisfies an LOC formula. Since the former is known to be undecidable, it follows that the latter is also undecidable.

B. Proof of Lemma 1

In this section we define the Presburger formula SYS_I whose existence was claimed by Lemma 1. We do so in several steps. First, we characterize the transition relation with formulas $TRAN_{sq}$ for each pair of states (s, q). These formulas have free variables t_{pr}, one for each transition $(p, r) \in T$. We construct $TRAN_{sq}$ such that an assignment $t_{pr} = n_{pr} \in \mathbb{Z}$ satisfies $TRAN_{sq}$ if an only if there is a path in T from s to q that crosses transition (p, r) exactly n_{pr} times. We set:

$$TRAN_{sq} = FLOW_{sq} \wedge CONN_s$$

Formula $FLOW_{sq}$ requires that the number of times a path enters the state must be equal to the number of times it leaves the state. The exceptions to this rule are states s, which must be exited one extra time, and q, which must be entered one extra time. Formally:

$$\begin{aligned} FLOW_{sq} = & \bigwedge_{(p,r)\in T} (t_{pr} \geq 0) \\ & \wedge \bigwedge_{r \in S} \Big(\sum_{(p,r)\in T} t_{pr} + Ind_{r=s} = \sum_{(r,w)\in T} t_{rw} + Ind_{r=q} \Big) \ , \end{aligned}$$

where Ind_P is 1 if proposition P holds, and it is 0 otherwise.

For example, for the system in Figure 2:

$$\begin{aligned} FLOW_{13} =& (t_{12} \geq 0) \wedge \cdots \wedge (t_{84} \geq 0) \\ & \wedge (t_{12} = t_{23} = t_{31} + 1 = t_{12} + t_{14}) \\ & \wedge (t_{14} + t_{84} = t_{45} = t_{56} = t_{67} = t_{78} = t_{84}) \ . \end{aligned}$$

Unfortunately, $FLOW_{sq}$ is not sufficient to fully characterize paths from s to q. For example, the assignment $t_{12} = t_{23} = 1$, $t_{31} = t_{14} = 0$, $t_{45} = t_{56} = t_{67} = t_{78} = t_{84} = 2$ satisfies $FLOW_{13}$ but it does not describe a path from 1 to 3. Rather, it describes a path and a loop not connected to the path. To eliminate such loops, in addition to $FLOW_{sq}$ we must state that if $t_{pr} > 0$, then there must exist a simple path from s to p, i.e. there must exist a sequence $s_1, \ldots, s_{k-1}, s_k$ of no more than $|S|$ states, such that $s_1 = s$, $s_k = p$, and $t_{s_{i-1}s_i} > 0$ for all $i = 2, \ldots, k$. This is stated by formula $CONN_s$ which uses $|S|$ variables v_k to represent this path. Here, we assume that S is a subset of integers. This assumption can be made without loss of generality, as integer encodings can be easily defined for any finite set. If the path is of length $l < |S|$, we require that $v_k = p$ for all $k > l$. So, if the value of v_k is not p, we are still in the active portion of the path and we must require that $t_{xy} > 0$, where x and y are values of v_k and v_{k+1} respectively. Formally, we define:

$$\begin{aligned} CONN_s = \bigwedge_{(p,r)\in T} (t_{pr} > 0) \Longrightarrow & \exists v_1 \ldots \exists v_{|S|} : \Big((v_1 = s) \wedge (v_{|S|} = p) \\ & \wedge \bigwedge_{k=1}^{|S|-1} (v_k = p) \Longrightarrow (v_{k+1} = p) \\ & \wedge \bigwedge_{k=1}^{|S|-1} \overline{(v_k = p)} \Longrightarrow \Big(\bigvee_{(x,y)\in T} (v_k = x) \wedge (v_{k+1} = y) \wedge (t_{xy} > 0) \Big) \Big) \ . \end{aligned}$$

It may appear $TRAN_{sq}$ needs a term similar to $CONN_s$ stating that if $t_{pr} > 0$, there must exists a simple path from r to q, but in fact, this statement is already implied by the conjunction of $FLOW_{sq}$ and $CONN_s$.

For example, for the system in Figure 2:

$$CONN_1 = ((t_{45} > 0) \Longrightarrow (t_{14} > 0)) \wedge \ldots \ ,$$

implying that:

$$\begin{aligned} TRAN_{13} =& (t_{31} \geq 0) \\ & \wedge (t_{12} = t_{23} = t_{31} + 1) \\ & \wedge (t_{14} = t_{45} = t_{56} = t_{67} = t_{78} = t_{84} = 0) \ . \end{aligned}$$

In the next step, we use $TRAN_{sq}$ to characterize generation relation with formulas GEN_{sq} for each pair of states (s, q). These formulas have a free variables g_e for each event $e \in E$. We construct GEN_{sq} such that an assignment $g_e = n_e \in \mathbb{Z}$ satisfies $TRAN_{sq}$ if an only if there exists a path in T from s to q along which event e is generated exactly n_e times. It is not hard to see that :

$$GEN_{sq} = \underbrace{\exists \ldots \exists t_{pr} \ldots}_{\text{over all } t_{pr} \text{ s.t. } (p,r) \in T} : TRAN_{sq} \wedge \bigwedge_{e \in E} \big(g_e = \sum_{(x,y) \in G(e)} t_{xy}\big)$$

For example, for the system in Figure 2:

$$\begin{aligned} GEN_{13} =& \exists t_{12} \ldots \exists t_{84} : TRAN_{13} \\ & \wedge (g_{x_1} = t_{12} + t_{23} + t_{31}) \\ & \wedge (g_{x_2} = t_{45} + t_{56} + t_{67} + t_{78} + t_{84}) \ , \end{aligned}$$

which can be simplified to $(g_{x_2} = 0) \wedge (\exists j \geq 0 : g_{x_1} = 3j + 2)$.

So far, we have characterized a system independently of the LOC formula. Next, we will define SYS_I for a specific interpretation I of the set of event expressions $\mathcal{E}_\phi$. But first, we need to introduce some additional notation. In the rest of the section, we will use e_ϵ, a_ϵ, and b_ϵ to denote the event name and constants appearing in event expression ϵ, i.e. we will assume that every ϵ is of the form $\text{val}(e_\epsilon[a_\epsilon i + b_\epsilon])$ or $f(e_\epsilon[a_\epsilon i + b_\epsilon])$, where f is an annotation. We say that two event expressions ϵ and ϵ' are *similar*, and write $\epsilon \sim \epsilon'$, if they refer to the same event, i.e. $e_\epsilon = e_{\epsilon'}$ and they both refer to the value of e_ϵ, or they both refer to the same annotation of e_ϵ.

We say that an ordered tuple $(q_0, s_1, q_1, \ldots, s_N, q_N) \in S^{2N+1}$ is an *instance* of interpretation I of $\mathcal{E}_\phi$ if the following is satisfied:

(1) q_0 is an initial state, i.e $q_0 \in S_0$.

(2) (s_n, q_n) is a transition, i.e $\forall n = 1, \ldots, N : (s_n, q_n) \in T$

(3) There exists a partition $\mathcal{E}_1, \ldots, \mathcal{E}_N$ of $\mathcal{E}_\phi$ such that for all $n = 1, \ldots, N$ and all $\epsilon \in \mathcal{E}_n$ the following holds:

a) if ϵ is of the form val$(e_\epsilon[a_\epsilon i + b_\epsilon])$, then the event e_ϵ can be generated on transition (s_n, q_n) with the value required by I, i.e. the following holds:

$$(s_n, q_n) \in G(e_\epsilon, I(\epsilon)) \ ,$$

b) I assigns the same value to all similar event expressions in the same partition, i.e.:

$$\forall \epsilon' \in \mathcal{E}_n : (\epsilon' \sim \epsilon) \Longrightarrow \big(I(\epsilon') = I(\epsilon)\big) \ .$$

We call any such a partition an *instantiating partition* of instance $(q_0, s_1, \ldots, q_N)$.

Intuitively, by traversing a path visiting $(s_1, q_1) \ldots (s_N, q_N)$ we could generate all event values required by I. However, SYS_I must also ensure that these values are generated at correct values of index expression. To do so, SYS_I uses a variable y_{ej}, for each $e \in E$ and each $j = 1, \ldots, N$, to count how many times event e is generated on a path segment form q_{j-1} to s_j. Formally:

$$SYS_I = \bigvee_{(q_0, s_1, \ldots, q_N)} \bigvee_{(\mathcal{E}_1, \ldots, \mathcal{E}_N)} \underbrace{\exists \ldots \exists y_{ej} \ldots}_{\text{over all } y_{ej} \text{ s.t. } e \in E, 1 \le j \le N} : \bigwedge_{n=1}^{N} \Big(GEN_{q_{n-1} s_n}(\ldots, y_{en}, \ldots)$$

$$\wedge \bigwedge_{\epsilon \in \mathcal{E}_n} \big(\sum_{k=1}^{n} (y_{e_\epsilon k} + Ind_{(s_k, q_k) \in G(e_\epsilon)}) = a_\epsilon i + b_\epsilon \big) \Big) \ ,$$

where the first disjunction ranges over all instances of I, the second disjunction ranges over all instantiating partitions of the current instance, and $GEN_{q_{n-1} s_n}(\ldots, y_{en}, \ldots)$ denotes the formula obtained form $GEN_{q_{n-1} s_n}$ by substituting variables g_e with y_{en} for all $e \in E$. The equation requires for all $\epsilon \in \mathcal{E}_n$ that the total number of times that e_ϵ is generated on the path from the initial state to the transition (s_n, q_n) is exactly as required by the index expression $a_\epsilon i + b_\epsilon$.

For example, the interpretation I which assigns 1 both to val$(x_1[3i])$ and val$(x_2[i])$ in formula (11) has a single instance $(1, 3, 1, 8, 4)$ with the unique instantiating partition $\mathcal{E}_1 = \{\text{val}(x_1[3i])\}$, $\mathcal{E}_2 = \{\text{val}(x_2[i])\}$. Therefore:

$$SYS_I = \exists y_{x_1 1} \exists y_{x_1 2} \exists y_{x_2 1} \exists y_{x_2 2} : \big(GEN_{13}(y_{x_1 1}, y_{x_2 1}) \wedge GEN_{18}(y_{x_1 2}, y_{x_2 2})$$

$$\wedge (y_{x_1 1} + 1 = 3i)$$

$$\wedge (y_{x_2 1} + y_{x_2 2} + 1 = i)\big) \ .$$

One can check that:

$$GEN_{13}(y_{x_1 1}, y_{x_2 1}) = (y_{x_2 1} = 0) \wedge (\exists j \ge 0 : y_{x_1 1} = 3j + 2)$$

$$GEN_{18}(y_{x_1 2}, y_{x_2 2}) = (\exists j \ge 0 : y_{x_1 2} = 3j) \wedge (\exists j \ge 0 : y_{x_2 2} = 5j + 4) \ ,$$

so SYS_I can be simplified to $(\exists j > 0 : 5j = i)$, as we anticipated in Section IV.