VERIFYING LOC BASED FUNCTIONAL AND PERFORMANCE CONSTRAINTS

Xi Chen, Harry Hsieh
University of California, Riverside, CA

Felice Balarin, Yosinori Watanabe
Cadence Berkeley Laboratories, Berkeley, CA

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. Assertion languages (e.g. IBM's Sugar2.0, Synopsys's OpenVera) have gained wide acceptance for specifying functional properties for automatic validation. They are, however, based on Linear Temporal Logic (LTL), and hence have certain limitations. Logic of Constraints (LOC) was introduced for specifying quantitative performance constraints, and is particularly suitable for automatic transaction level analysis. We analyze LTL and LOC, and show that they have different domains of expressiveness. Using both LTL and LOC can make the verification process more effective in the context of simulation assertion checking as well as formal verification. Through industrial case studies, we demonstrate the usefulness of this verification methodology.

1 Introduction

As embedded systems today are becoming more integrated and complex, system verification continues to be a major challenge. More than ever, design and verification methodologies at higher levels of abstraction are required to minimize the design cost of an electronic product. 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) [10]) and use a formal verification tool (e.g. the model checker Spin [11]) to run the verification. Formal verification checks the entire state space of a design to verify some specified property without any uncertainty. At the lower level, however, the complexity can quickly overwhelm the automatic tools, and the simulation becomes the primary means for verifying the behavior of a design.

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 system designs during simulation. Today's embedded assertion languages capture those simple logics as language/platform specific library blocks (e.g. handshake assertion) and use a set of extended temporal logic to operate on those blocks for expressing more complex assertions. IBM's Sugar2.0 [8] is chosen by Accellera as the standard formal property specification language (PSL) for assertion-based verification. A similar assertion specification language from Synopsys, OpenVera [1], can be used to specify both simple temporal sequences of events and complex temporal formulas.

To effectively specify high-level quantitative performance and functional constraints, a formal constraint language, Logic Of Constraints (LOC), has been proposed in [4]. LOC can be used to express a wide range of interesting constraints including real time performance constraints and functional data consistency constraints. In this paper, we focus on the expressiveness and analyzability of LOC. Through several examples, we claim that LOC and LTL are expressively incomparable. Although it has been proven that an LTL formula can be translated to an equivalent Büchi automaton for verification [15], LOC appears to be more difficult to be formally analyzed. We review the simulation-based trace analysis methodology for LOC formulas [7], and then present our formal verification approach for LOC with help of the model checker Spin [11]. In the case studies, a TTL (Task Transition Level) communication channel design [5, 9] specified in Metropolis Meta-Model [3] is used to demonstrate the usefulness and effectiveness of our LOC-based quantitative constraint verification.

The rest of the paper is organized as follows. In the next section, we review Logic Of Constraints (LOC), Linear Temporal Logic (LTL), and their typical usage. In Section 3, we analyze the expressiveness of LOC. In Section 4, the approach of simulation trace analysis for LOC formulas is reviewed. Then, we discuss our formal verification approach for LOC formulas. In Section 5, we present the case studies on validating a TTL channel design with LOC constraints. Both techniques of simulation trace analysis and model checking are utilized and tested. Section 6 concludes the paper and gives our future work.

2 Background

In this section, we review the salient aspects of our quantitative constraint formalism, Logic Of Constraints (LOC) and the popular temporal logic for functional property speci-

fication, Linear Temporal Logic (LTL). We discuss their typical usage and the typical constraints they can express.

2.1 Logic Of Constraints

Logic Of Constraints [4] is a formalism designed to reason about traces from the execution of a system. It consists of all the terms and operators allowed in sentential logic, with additions that make it possible to specify system level quantitative functional and performance constraints without compromising the ease of analysis. LOC can be used to specify very common and useful real-time performance constraints:

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

$$t(Display[i+1]) - t(Display[i]) = 10 \ , \qquad (1)$$

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

$$t(Display[i]) - t(Stimuli[i]) \leq 25 \ , \qquad (2)$$

- 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":

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

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

$$t(Display[i+100]) - t(Display[i]) \leq 1001 \ . \qquad (4)$$

The basic components of an LOC formula are: event names (e.g. $Display$ and $Stimuli$), instances of events (e.g. $Display[4]$ for the fourth instance of the event $Display$ in the current execution given as a trace of event instances), indices of event instances (e.g. 0, 1, ..., etc), the index variable i and annotations (e.g. t). The rate constraint (1) requires that the difference between the values of annotation t for any two consecutive $Display$ instances should equal to 10.

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). In addition, LOC can also be used to specify quantitative functional constraints like data consistency.

2.2 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 $\neg$ (not), $\vee$ (or), $\wedge$ (and), $\rightarrow$ (imply), and the linear temporal operators $\Box$ (always), $\Diamond$ (eventually), X (next) and U (strong until). For example, $\Box$(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) is true if A is true in the following state, and A U B is true if B eventually becomes true in a future state and A is true from the current state to that future state.

It has been proven that LTL formulas can be translated to equivalent Büchi automata [15]. 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 [11]). LTL is also widely used in the formal property specification for simulation-based assertion verification [1, 8], which is important to assure the integration and correctness of reusable IP (Intellectual Property) blocks.

3 LOC v.s. LTL

In this section, we discuss the expressiveness of LOC and compare it with LTL. For LTL, we only restate here its well known properties. Note that LTL is defined on the state transition level where any change on the system state can be accounted for, while LOC works on a higher abstraction level, in which only the events observable from the system and their annotations are considered. However, this difference 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. 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:

$$data(Display[i]) > 100 \ . \qquad (5)$$

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:

$$\Box(Display\_occur \rightarrow (Display\_data > 100)) \ . \qquad (6)$$

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

Many quantitative constraints that can be easily expressed with LOC are not suitable for LTL. Specifically, when more than one events need to be compared in the same constraint, LTL is not expressive enough to be used. For example, the latency constraint (2) 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 $Display$ could be arbitrarily large. [1] 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.

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. [2]

Claim 3: LOC can express only safety properties.

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

4 Verification Approaches for LOC Formulas

In this section, we discuss the analyzability of LOC. We first review the simulation-based trace analysis approach presented in [7], and show that LOC constraints can be easily verified 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.

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

[2] To disprove a liveness property, we need to show that the system can enter an infi nite cycle in which there are unfulfi lled requests.

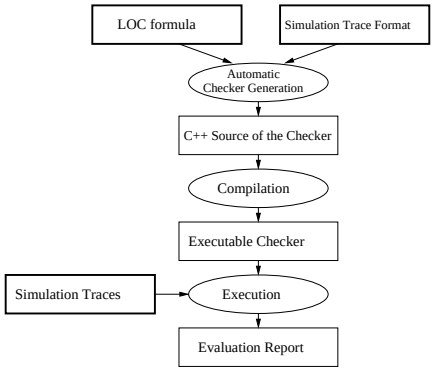

Figure 1. Trace Analysis Methodology.

4.1 Simulation Verification of LOC Formulas

The methodology for simulation verification with an automatically generated LOC checker is illustrated in Figure 1 [7]. From the specification of LOC formulas and a trace format, the 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.

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 0, 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 0, the LOC checker scans the trace sequentially. If any relevant data is read in, the checker stores it into a queue data structure and checks the formula; otherwise, the checker keeps scanning the trace.

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 one tries 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 [7]. The trace checking technique can be easily extended to runtime constraint monitoring, which is usually run concurrently with the simulation.

4.2 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 and other small designs for frequent use. 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. 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 not always the case, as exemplified by the trace checker for the latency constraint (2).

On the other hand, many LOC formulas do have corresponding finite-state trace checkers. For example, the rate constraint:

$$t(Display[i+1]) - t(Display[i]) = 10 \qquad (7)$$

compares the annotation t of any two consecutive occurrences of the event $Display$. To verify this formula, the trace checker (see Section 4.1) 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 $\text{val}(A[i^2]) = 1$), 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 $\text{val}(A[i]) = \text{val}(A[2i])$).

Although Rule 1 may appear quite restrictive, still many interesting properties satisfy it, including throughput (4) and rate (7) 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 (7), 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 (7) into the state predicate:

$$Display\_occur \implies Display\_t - Display\_t\_last = 10 \ . \qquad (8)$$

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 the three outcomes, unsatisfied if a counter-example is found showing that the system does not satisfy the property, satisfied if all reachable state are searched without finding a counter-example or reaching a state where memory is exhausted, or inconclusive if reachability analysis finds no counter-examples, but states where memory is exhausted are reachable.

For example, the latency constraint (2) cannot be modeled by any finite automata because there can be arbitrarily many occurrences of $Stimuli$ before the 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 will simultaneously check the following two properties:

P1: There are never more than 50 occurrences of $Stimuli$ between the x-th occurrences of $Stimuli$ and $Display$.

P2: If **P1** holds, then (2) holds.

Obviously, if **P1** and **P2** both hold then so does (2), and if **P2** is false, so is (2). 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 the 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 indices 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:

$$Stimuli\_occur \implies (Stimuli\_i \neq Display\_i) \ . \qquad (9)$$

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 (2) can be expressed as follows:

$$Display\_occur \implies Display\_t[Display\_i] - Stimuli\_t[Display\_i] < 25 \qquad ,(10)$$

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

$$Assumption\ (9) \implies Formula\ (10) \ . \qquad (11)$$

It is natural to search for a general algorithm to check any LOC formula. Unfortunately, checking LOC is undecidable in general. 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 decidability of LOC restricted to finitely-valued annotations remains an open problem, however there are indications that it is quite hard. For example, in a system with a single event x taking values from $\{0, 1, 2, 3\}$, the formula:

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

defines the language:

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

It is not hard to show (e.g. see Example 6.1 in [12]) that this language is not context-free, and thus cannot be recognized even with a pushdown automaton, let alone a finite-state one.

5 Case Studies

Y-chart Application Programmer's Interface (YAPI) is a model of computation for designing signal processing systems [14]. It is basically a Kahn process network [13], 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 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 2.

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 2 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. 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 read from the channel in exactly the same order without a loss.

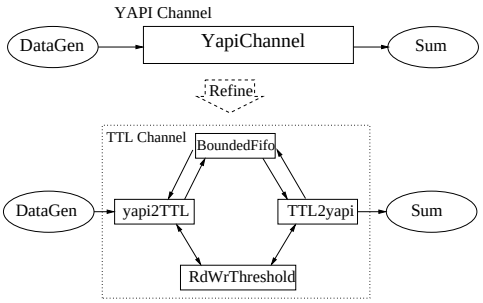

Figure 2. YAPI Channel and TTL Channel.

5.1 Simulation Trace Analysis for TTL Channel

The TTL channel shown in Figure 2 is initially specified in Metropolis Meta-Model (MMM) [3]. 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:

$$data(processed[i]) \geq 0 \ , \tag{12}$$

and the data consistency constraint is defined as:

$$data(prepared[i]) = data(processed[i]) \ . \tag{13}$$

The automatic checker generator is used to parse the definition file 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 (12) and (13) hold on traces of 10^5 to 10^8 lines. The time and memory usage of the trace analysis are shown in Table 1.

Table 1. Results of Checking Formulas(12),(13)

Lines of Trace		10^5	10^6	10^7	10^8
Formula (12)	Time(s)	<1	5	44	432
	Mem(Bytes)	4	4	4	4
Formula (13)	Time(s)	<1	8	84	767
	Mem(Bytes)	172	172	176	172

5.2 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 language) description [11], 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 (12) can be expressed in LTL as:

$$\Box(processed\_occur \rightarrow processed\_data > 0) \ , \qquad (14)$$

where $processed\_data$ stores the most recent data read by Sum. With the bitstate technique [11], Spin verifies the LTL formula (14) within 2 hours on our 1.5GHz Athlon machine with 1GByte of memory. The same setup is used for all case studies in this paper. All the relevant verification parameters are listed in Table 2.

From the discussion in Section 4.2, we know that the data consistency constraint (13) 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". [3] 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 indices of the most recent data in the arrays. The assumption is written in LTL as:

$$\Box(prepared\_occur \rightarrow prepared\_i \neq processed\_i) \ , \qquad (15)$$

and it is verified to hold by Spin (see Table 2). The data consistency constraint is written in LTL as:

$$\Box(processed\_occur \rightarrow prepared\_data[processed\_i] = processed\_data[processed\_i]) \qquad (16)$$

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

$$Assumption\ (15) \rightarrow Constraint\ (16) \qquad (17)$$

is verified to hold by Spin, and all the relevant verification parameters are also listed in Table 2.

Table 2. Verification Results for Formulas (14), (15) and (17)

LTL Formula	(14)	(15)	(17)
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

6 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 directly expressed by LTL. Although it appears that LOC formulas are more difficult to be analyzed, we discuss two feasible verification approaches, simulation trace analysis and model checking. We also present case studies on these approaches to demonstrate their usefulness and effectiveness.

Our future work includes extending the LOC formalism with the universal quantifier $\forall$ and existential quantifier $\exists$, and constraint-guided non-deterministic simulation.

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

References

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

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

[3] 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.

[4] F. Balarin, Y. Watanabe, J. Burch, L. Lavagno, R. Passerone, and A. Sangiovanni-Vincentelli. Constraints specification at higher levels of abstraction. *International Workshop on High Level Design Validation and Test - HLDVT01*, Sept. 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. *International Workshop on High Level Design Validation and Test - HLDVT02*, Sept. 2002.

[7] X. Chen, H. Hsieh, F. Balarin, and Y. Watanabe. Automatic trace analysis for logic of constraints. 40^{th} *Design Automation Conference*, June 2003.

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

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

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

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

[12] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, languages and Computation*. Addison Wesley, 1979.

[13] G. Kahn. The semantics of a simple language for parallel programming. *Proceedings of IFIP Congress 74*, pages 471–475, 1974.

[14] 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. *Proceedings of the 37th Design Automation Conference*, 2000.

[15] 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.