Automatic Trace Analysis for Logic of Constraints

Xi Chen, Harry Hsieh
University of California at Riverside
Riverside, CA 92521
{xichen, harry}@cs.ucr.edu

Felice Balarin, Yosinori Watanabe
Cadence Berkeley Laboratories
Berkeley, CA 94704
{felice, watanabe}@cadence.com

ABSTRACT

Verification of system designs continues to be a major challenge today. Simulation remains the primary tool for making sure that implementations perform as they should. We present algorithms to automatically generate trace checkers from formulas written in the formal quantitative constraint language, Logic Of Constraints (LOC), to analyze the simulation traces for functional and performance constraint violations. For many interesting formulas, the checkers exhibit linear time complexity and constant memory usage. We illustrate the usefulness and efficiency of this approach with large designs and traces.

Categories and Subject Descriptors

B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; D.2.4 [**Software Engineering**]: Software/Program Verification—*Assertion checkers*

General Terms

Verification

Keywords

Performance constraint, Trace analysis, Logic of Constraints, Simulation checker

1. INTRODUCTION

The increasing complexity of embedded systems today demands more sophisticated design and test methodologies. Systems are becoming more integrated as more and more functionality and features are required for the product to succeed in the market. Embedded system architectures likewise have become more heterogeneous as it is becoming more economically feasible to have various computational resources (e.g. microprocessor, digital signal processor, reconfigurable logics) all utilized on a single chip. Designing at the Register Transfer Level (RTL) or sequential C-code level is no longer efficient. The specification of the functionality and the architecture should be done at a high level of abstraction, and the design procedures refine the abstract functionality, refine the abstract architecture, and map the functionality onto the architecture through automatic tools or manual means with tools support [10, 4]. High level design procedures allow the designer to tailor their architecture to the functionality at hand or to modify their functionality to suit the available architectures(see Figure 1).

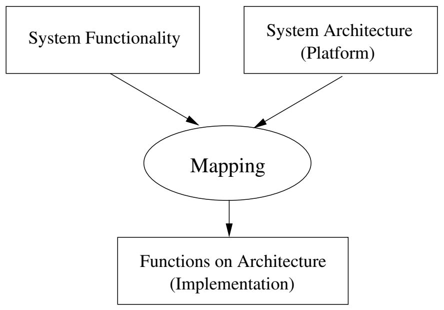

System Functionality
System Architecture (Platform)
Mapping
Functions on Architecture (Implementation)

Figure 1: System Design Methodology.

To make the practice of designing from high level system specification a reality, verification methods must accompany every step in the design flow. Specification at the system level makes formal verification possible [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) [8]) and use a formal verification tool (e.g. the model checker Spin [9]) to run the verification. At the lower level, however, the complexity can quickly overwhelm the automatic tools and simulation becomes the workhorse for verification.

While the coverage achieved by simulation is limited by the number of simulation vectors, simulation-based property analysis is still the standard vehicle for design analysis in practical designs. One major problem of simulation-based property analysis is that it is not always straightforward to evaluate the simulation traces and deduce the absence or presence of an error. In this paper, we propose an efficient automatic approach to analyze simulation traces and check whether they satisfy quantitative functional and performance properties specified by denotational logic formulas. The property to be verified is written in Logic of Constraints (LOC) [5], a logic particularly suitable for specifying constraints at the abstract system level, where coordination of executions, not the low level interaction, is of paramount concern. We generate C++ trace checkers automatically from the quantitative LOC formulas. The checkers analyze the traces and report any violations of the LOC formulas. Like any other simulation-based approach, the checker can only disprove the LOC formula (if a violation is found), but it can never prove it conclusively, as that would require analyzing infinitely many traces. The automatic checker generation is parameterized, so it can be customized for fast analysis for specific verification environment (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 [2] simulator for runtime monitoring. No theoretical difficulty exists to generate checkers in Hardware Description Languages (HDLs) for integration with hardware simulators, or in Java for concurrent execution with the simulator. We illustrate the concept and demonstrate the usefulness of our approach through two case studies: a higher level description of a Picture-In-Picture design, and a Register Transfer Level description of a Finite Impulse Response filter.

Our simulation checker approach is similar in spirit to the hardware embedded assertion languages which has become very popular in recent years. Sugar [7] and OpenVera [1] are two such languages that are the *de facto* standards in the industry. Embedded assertions start out as simple comparison logics that designers insert into their HDL descriptions to help uncover bugs 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 Linear Temporal Logic (LTL) to operate on those blocks for expressing more complex assertions. Our approach can complement the embedded assertion languages and are indeed different in at least four fundamental aspects. First, Logic of Constraints is designed for expressing all performance and functional constraints, not just functional ones. Second, to express performance constraints effectively, LOC can be used to express properties that belong to the infinite automata domain (through the use of an integer index variable), while LTL is equivalent to finite automata. Third, the checkers are stand-alone objects that do not have to be embedded into any simulation language, hence are more platform independent. Fourth, system level functional and performance constraints written in LOC can be automatically and efficiently synthesized into static checkers or runtime monitors, as will be shown in the remainder of this paper.

In the next section, we discuss the definition and usage of LOC. In section 3, we present the methodology for building a trace checker or monitor for any given LOC formula. We demonstrate the usefulness and efficiency with two verification case studies in section 4. Finally, in section 5, we conclude the paper and provide some future directions.

2. LOGIC OF CONSTRAINTS

Logic Of Constraints [5] is a formalism designed to reason about simulation traces. 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. The basic components of an LOC formula are:

- **event names:** An input, output, or intermediate signal in the system. E.g. "*in*", "*out*", "*Stimuli*", and "*Display*".
- **instances of events:** An instance of an event denotes one of its occurrence in the simulation trace. Each instance is tagged with a positive integer index, in the order they appeared in the simulation trace. E.g., "$Stimuli[0]$" denotes the first instance of the event "*Stimuli*".
- **index and index variable:** There can be only one index variable i, a positive integer. Index expressions of events may be any arithmetic operations on i and the annotations of the instances of events. E.g. "$Display[i-5]$", "$Stimuli[value(Display[i])]$".
- **annotation:** Each instance of the event may be associated with one or more annotations. Annotations can be used to denote the time, power, area, or any value related to the event instance. E.g., "$t(Display[i-5])$" denotes the "t" annotation (probably time) of the "$i-5$"th instance of the "*Display*" event.

LOC can be used to specify some very common and useful real-time 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":
$$t(Display[i+100]) - t(Display[i]) \leq 1001 \qquad (4)$$
- **burstiness:** E.g. "no more than 1000 *Display* events will arrive in any period of 9999 time units":
$$t(Display[i+1000]) - t(Display[i]) > 9999 \qquad (5)$$

It should be emphasized that time is only one of the possible annotations and in the case of concurrent events, time annotation values should be the same. The indexes of instances of the same event name denote the strict order as they appear in the simulation 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).

As pointed out in [5], 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

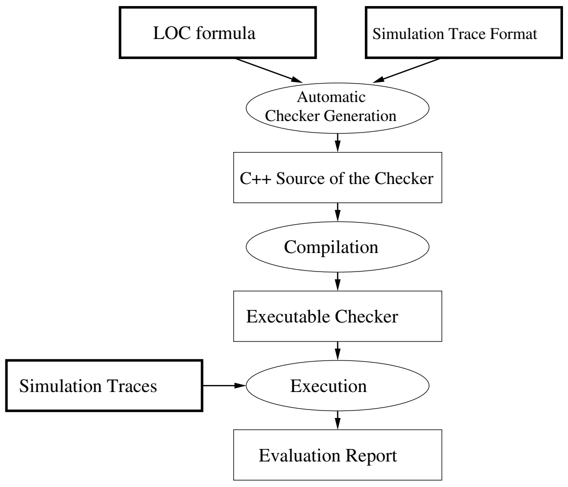


Figure 2: Trace Analysis Methodology.

is available, the latency constraint can be more accurately written as:

$$t(Display[i]) - t(Stimuli[cause(Display[i])]) \leq 25 \qquad (6)$$

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

For a formula to be "formally proven" for a design, it needs to hold for all possible traces and all values of i, as it appears in the index expression of the formula. For a formula to be "checked" (or "checked for conformance") for a simulation trace, it needs to hold for that trace only and all values of i, which result in index values corresponding to instances of the events that actually appear in the trace. By adding additional index variables and quantifiers, LOC can be extended to be at least as expressive as S1S [3] and LTL. There is no inherent problem in generating checkers for them. However, the efficiency of the checker will suffer greatly as memory recycling becomes very difficult (as will be explained in the next section).

3. TRACE ANALYSIS METHODOLOGY

The methodology for verification with an automatically generated LOC checker is illustrated in Figure 2. From the specification of LOC formulas and the trace format, 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 3. Each LOC formula is preceded by a label and followed by the format for extracting event names and their annotations out of the simulation traces. The format described in the figure is written to work with the trace shown in figure 7. It specifically looks for a line that starts with a string which ends in a ":", followed by an integer, a string pattern " at time ", then followed by a float. The string is taken as an event name, which may be discarded if not mentioned in the formula. The integer is taken as the "value" annotation of the event instance, which

```
[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"
```

Figure 3: Definition of LOC formulas and Trace Formats for Rate and Latency Constraints.

is not used by the given formula and is discarded. Finally, the float is taken as the "t" annotation of the event instance which is needed for the evaluation of the formula. Any line that does not match this format will be ignored. Multiple formulas may be checked at the same time with possibly different extraction formats.

The automatic checker generator parses the definition file to generate the 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 in section 3.1.

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 4 shows the case where latency between the 23rd event instance of $Display$ and 23rd event instance of $Stimuli$ violate the given formula. The checker is designed to keep checking and reporting any violation until stopped by the user or if the trace terminates.

```
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
                        ⋮
```

Figure 4: Example of Error Report.

3.1 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 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 data structure (double-ended 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 2), the algorithm uses a fixed amount of memory no matter how long the traces are (see table 1) [1]. 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 any more once all the formula instance with i less than a certain number are all evaluated. For example, let's say we have an LOC formula:

$$t(Display[i+10]) - t(Stimuli[i+5]) < 300 \qquad (7)$$

and the current value of i is 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.

Combination of multiple LOC formulas with logical operations is also an LOC formula. We employ a partial evaluation mechanism that is based on 3-value logic [12], i.e. *true*, *false* and *unknown*. 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 *unknown*. For example, given the following LOC formula:

$$t(Display[i+10]) > 100 \ || \ t(Stimuli[i-5]) < 300 \qquad (8)$$

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. However, if the current value of i is 4, -1 is then an invalid index for annotation t of event *Stimuli*. The expression $t(Stimuli[-1]) < 300$ is evaluated to *unknown* and the whole formula can be evaluated according to the evaluation of $t(Display[14]) > 100$. Thus the LOC formula instances can be evaluated as soon as possible, which further minimizes the memory usage.

3.2 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 [2]. 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 4.2. 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.

[1]The verification of the constraint 2 also has constant memory usage due to the regular structure of the given trace

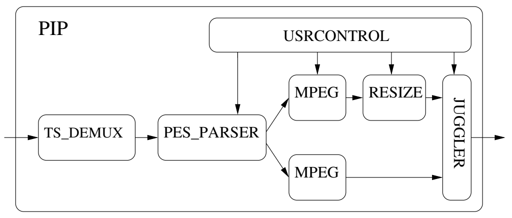


Figure 5: Picture-In-Picture Design.

3.3 Dealing with Memory Limitation

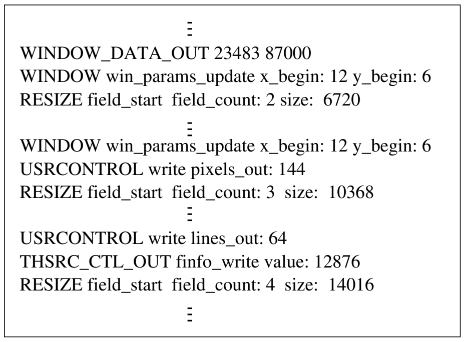
Despite the memory efficient for most LOC formulas, some LOC formula may require high memory usage which the verification environment can not support. To deal with the case of preset memory limitation, another extension has been added to the checker generator. Generally, the checker tries to read the 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 (current 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 case study in section 4.2) 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.

4. CASE STUDIES

We apply our methodology to two very different design examples. The first is a system level design for set-top video processing, Picture-In-Picture (PIP), which is originally specified with YAPI [11]. 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.

4.1 Picture-In-Picture

Figure 5 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 (RESIZE) or directly feed to the 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.



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

Figure 6: PIP Simulation Trace.

At the system level, we can use LOC to check the functional properties. In the RESIZE component 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 output and not in the middle of a frame. Therefore, the field sizes of paired even and odd fields should be the same. This property can be express as an LOC formula:

$$size(field\_start[2i+2]) - size(field\_start[2i+1]) = size(field\_start[2i+1]) - size(field\_start[2i]) \qquad (9)$$

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 6 shows snapshots of the PIP trace. The generation of the checker for this LOC and the actual trace checking on the simulation trace take less than 1 minute of CPU time.

Another functionality 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 the 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

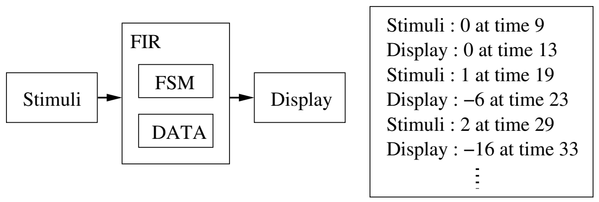
$$field\_count(in[i]) = field\_count(out[i]) \qquad (10)$$

The events *in* and *out* are generated by the RESIZE input and output parts respectively whenever they finish processing a whole piece of video. The annotation *field_count* is the number of fields the RESIZE input and output parts processed. The generation of the checker for this LOC 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*):

$$field\_count(change\_size[i]) - field\_count(read\_size[i]) \leq 5 \qquad (11)$$

read_size is generated whenever RESIZE reads a new size specification from USRCONTROL. *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 and the actual trace checking also take less than 1 minute of CPU time.



```
Stimuli : 0 at time 9
Display : 0 at time 13
Stimuli : 1 at time 19
Display : −6 at time 23
Stimuli : 2 at time 29
Display : −16 at time 33
            ⋮
```

Figure 7: FIR Design and Simulation Trace.

4.2 FIR filter

Figure 7 shows a 16 tap FIR filter which reads in samples when the input is valid and writes out the result when output is ready. The filter design is divided into a control FSM and a data path. The test bench feeds sampled data of arbitrary length and the output is displayed with the simulator.

We utilize our automatic trace checker generator and verify the properties specified in constraints (1) - (5) (of section 2). The same trace files are used for all the analysis and each constraint is checked one at a time. The time and maximum memory usage are shown in Table 1. We can see that the time required for analysis grow 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 1: Result 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 2. Given the large file size, runtime monitoring (see section 3.2) may reduce the total verification time (simulation and checking) since no trace file need to be actually generated. For the latency constraint (constraint 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 approach requires only 1420 seconds for both simulation and monitoring.

Table 2: Result of Runtime Monitoring on FIR for Constraint (2)

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

We also verify constraint (6) to illustrate verification with memory limitation since this constraint is particularly expensive in terms of memory usage. Table 3 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 3.3. The result is shown in Table 3 where the memory usage is experimentally limited to 50KB. We see that the analysis takes more time when the memory limitation 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 3: Result of Constraint (6) 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

5. CONCLUSIONS

We have presented a methodology for design verification through automatic trace analysis. We have demonstrated how we can take any formula written in the formal quantitative constraint language, Logic Of Constraint, and automatically generate a trace checker that can efficiently analyze the simulation traces for constraint violations. The checker can also operate as a simulation monitor. Even under memory limitation, the checker continues to operate and provides debug information. We have applied the methodology to two case studies and demonstrated that automatic LOC trace analysis can be very useful.

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. Lastly, we plan to retarget the backend checker generation for different development environments (e.g. Superlog, VHDL) to allow tight integration of monitors for those environments as well.

6. REFERENCES

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

[2] http://www.systemc.org, Mar. 2003.

[3] A. Aziz, F. Balarin, R. Brayton, and A. Sangiovanni-Vincentelli. Sequential synthesis using S1S. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 19(10):1149–62, Oct. 2000.

[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] 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.

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

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

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

[10] 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.

[11] 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.

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