Chapter 1

SIMULATION TRACE VERIFICATION FOR QUANTITATIVE CONSTRAINTS

Xi Chen[1], Harry Hsieh[1], Felice Balarin[2], Yosinori Watanabe[2]
[1]*University of California, Riverside, CA, USA*

[2]*Cadence Berkeley Laboratories, Berkeley, CA, USA*

Abstract: System design methodology is poised to become the next big enabler for highly sophisticated electronic products. Design verification continues to be a major challenge and simulation will remain an important tool for making sure that implementations perform as they should. In this paper we present algorithms to automatically generate C++ checkers from any formula written in the formal quantitative constraint language, Logic Of Constraints (LOC). The executable can then be used to analyze the simulation traces for constraint violation and output debugging information. Different checkers can be generated for fast analysis under different memory limitations. LOC is particularly suitable for specification of system level quantitative constraints where relative coordination of instances of events, not lower level interaction, is of paramount concern. We illustrate the usefulness and efficiency of our automatic trace verification methodology with case studies on large simulation traces from various system level designs.

Key words: Quantitative constraints, Trace analysis, Logic of Constraints

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 marketplace. Embedded system architecture likewise has 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 board module or a single chip. Designing at the Register Transfer Level (RTL) or sequential C-code level, as is done by embedded hardware and software developers today, is no longer efficient. The next major productivity gain will come in the form of system level design. The specification of the functionality and the architecture should be done at a high level of abstraction, and the design procedures will be in the form of refining the abstract functionality and the abstract architecture, and of mapping the functionality onto the architecture through automatic tools or manual means with tools support [1, 2]. 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). Significant advantages in flexibility of the design, as compared to today's fixed architecture and *a priori* partitioning approach, can result in significant advantages in the performance and cost of the product.

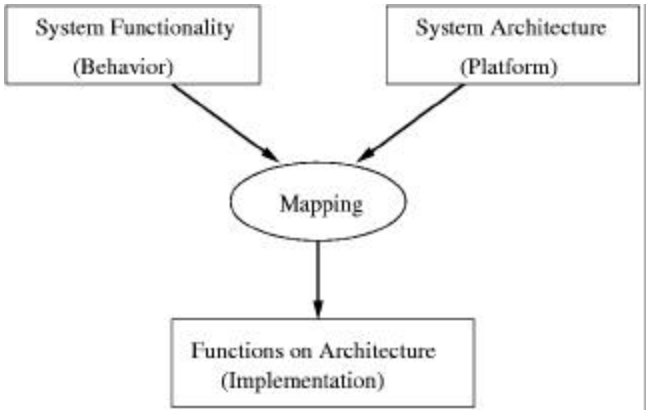

Figure 1. System Design Methodology

In order to make the practice of designing from high-level system specification a reality, verification methods must accompany every step in the design flow from high level abstract specification to low level implementation. Specification at the system level makes formal verification possible [3]. 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) [4], Computational Tree Logic (CTL) [5]) and use a formal verification tool (e.g. Spin Model checker [6, 7], FormalCheck [8], SMV [9]) to run the verification. At the lower level, however, the complexity can quickly overwhelm the automatic tools and the simulation quickly becomes the workhorse for verification.

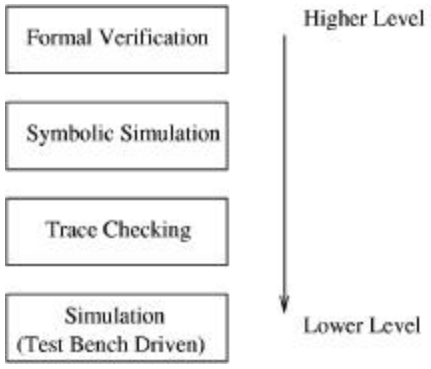

Figure 2. System Verification Approaches

The advantage of simulation is in its simplicity. While the coverage achieved by simulation is limited by the number of simulation vectors, simulation is still the standard vehicle for design analysis in practical designs. One 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 properties specified by denotational logic formulas. The property to be verified is written in Logic of Constraints (LOC) [10], 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 then automatically generate a C++ trace checker from the quantitative LOC formula. The checker analyzes the traces and reports any violations of the LOC formula. 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. We illustrate the concept and demonstrate the usefulness of our approach through case studies on two system level designs. We regard our approach as similar in spirit to symbolic simulation [11], where only particular system trajectory is formally verified (see Figure 2). The automatic trace analyzer can be used in concert with model checker and symbolic simulator. It can perform logic verification on a single trace where the other approaches failed due to excessive memory and space requirement.

In the next section, we review the definition of LOC and compare it with other forms of logic and constraint specification. In section 3, we discuss the algorithm for building a trace checker 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 and provide some future directions.

2. LOGIC OF CONSTRAINTS (LOC)

Logic Of Constraints [10] 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 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. Examples of event names are *"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, strictly ordered and starting from *"0"*. For example, *"Stimuli[0]"* denotes the first instance of the event *"Stimuli"* and *"Stimuli[1]"* denotes the second instance of the event;
- **index and index variable:** There can be only one index variable *i*, a positive integer. An index of an event can be any arithmetic operations on *i* and the annotations. Examples of the use of index variables are *"Stimuli[i]"*, *"Display[i-5]"*;
- **annotation:** Each instance of the event may be associated with one or more annotations. Annotations can be used to denote the time, power, or area related to the event occurrence. For example, *"t(Display[i-5])"* denotes the *"t"* annotation (probably time) of the *"i -5"*th instance of the *"Display"* event. It is also possible to use annotations to denote relationships between different instances of different event. An example of such a relationship is causality. *"t(in[cause(out[i]))]"* denotes the *"t"* annotation of an instance of *"in"* which in turn is given by the *"cause"* annotation of the *"i"*th instance of *"out"*.

LOC can be used to specify some very common real-time constraints:

- **rate**, e.g. "a new *Display* will be produced every 10 time units":

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

- **latency**, e.g. "*Display* is generated no more than 45 time units after *Stimuli*":

  $$t(Display[i]) - t(Stimuli[i]) \leq 45 \quad (2)$$

- **jitter**, e.g. "every *Display* is no more than 15 time units away from the corresponding tick of the real-time clock with period 15":

  $$| t(\text{Display}[i]) - i * 10 | \leq 15 \quad (3)$$

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

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

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

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

As pointed out in [10], the latency constraints 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 constraints can be more accurately written as:

$$t(\text{Display}[i]) - t(\text{Stimuli}[\text{cause}(\text{Display}[i])]) \leq 45 \quad (6)$$

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.

By adding additional index variables and quantifiers, LOC can be extended to be at least as expressive as S1S [12] and Linear Temporal Logic. There is no inherent problem in generating simulation monitor for them. However, the efficiency of the checker will suffer greatly as memory recycling becomes impossible (as will be discussed in the next section). In similar fashion, LOC differs from existing constraint languages (e.g. Rosetta [13], Design Constraints Description Language [14], and Object Constraint Language [15]) in that it allows only limited freedom in specification to make the analysis tractable. The constructs of LOC are precisely chosen so system-level constraints can be specified and efficiently analyzed.

3. THE LOC CHECKER

We analyze simulation traces for LOC constraint violation. The methodology for verification with automatically generated LOC checker is illustrated in Figure 3. From the LOC formula and the trace format specification, an automatic tool is used to generate a C++ LOC checker. The

checker is compiled into an executable that will take in simulation traces and report any constraint violation. To help the designer to find the point of error easily, the error report will include the value of index *i* which violates the constraint and the value of each annotation in the formula (see Figure 4). The checker is designed to keep checking and reporting any violation until stopped by the user or if the trace terminates.

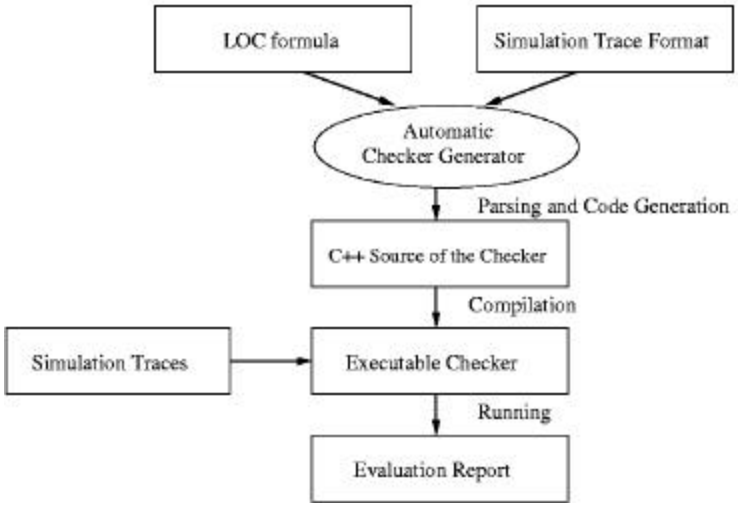

Figure 3. Trace Analysis Methodology

```
cs220@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

The algorithm progresses based on 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 fix positive integer number. The basic algorithm used in the checker is given as follows:

Algorithm of LOC Checker:

```
i = 0;
memory_used = 0;

Main {
  while(trace not end){
    if(memory_used < MEM_LIMIT){
      read one line of trace;
      store useful annotations;
      check_formula();
    }
    else{
      while(annotations for current
            formula instance is not
            available && trace not end)
        scan the trace for annotation;
      check_formula();
    }
  }
}

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. The memory usage, however, may become prohibitively high if we try to keep the entire trace in the memory for analysis. As the trace file is scanned in, the proposed checker attempts to store only the useful annotations and in addition, evaluate as many formula instances as possible and remove from memory parts of the trace that are no longer needed (memory recycling). The algorithm tries to read and store the trace only once. However, after the memory usage reaches the preset limit, the algorithm will not store the annotation information any more. Instead, it 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 the reading and storing of annotation from the same location. The analysis time will certainly be impacted in this case (see Table 3). However, it will also allow the checker to be as efficient as possible, given the memory limitation of the analysis environment.

For many LOC formulas (e.g. constraints 1 - 5), the algorithm uses a fixed amount of memory no matter how long the traces are (see table 2). Memory efficiency of the algorithm comes from being able to free stored annotations as their associated formula instances are evaluated (memory recycling). This ability is directly related to the choice made in designing LOC. From the LOC formula, we often know 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:

$$t(input[i+10]) - t(output[i+5]) < 300 \quad (7)$$

and the current value of *i* is 100. Because the value of *i* increases monotonically, we know that event *input*'s annotation *t* with index less than 111 and event *output*'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.

4. CASE STUDIES

In this section, we apply the methodology discussed in the previous section to two very different design examples. The first is a Synchronous Data Flow (SDF) [16] design called Expression originally specified in Ptolemy and is part of the standard Ptolemy II [17] distribution. The Expression design is respecified and simulated with SystemC simulator [18]. The second is a Finite Impulse Response (FIR) filter written in SystemC and is actually part of the standard SystemC distribution. We use the generated trace checker to verify a wide variety of constraints.

4.1 Expression

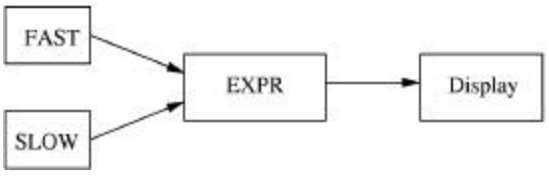

Figure 5. Expression Design Example

Figure 5 shows a SDF design. The data generators SLOW and FAST generate data at different rates, and the EXPR process takes one input from

each, performs some operations (in this case, multiplication) and outputs the result to DISPLAY. SDF designs have the property that different scheduling will result in the same behavior. A snapshot of the simulation trace is shown in Figure 6.

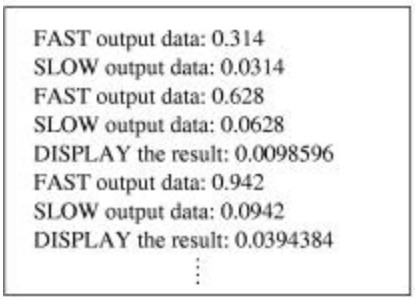

Figure 6. Expression Simulation Trace

The following LOC formula must be satisfied for any correct simulation of the given SDF design:

$$SLOW[i] * FAST[i] = DISPLAY[i] \quad (8)$$

We use the automatically generated checker to show that the traces from SystemC simulation adhere to the property. This is certainly not easy to infer from manually inspecting the trace files, which may contain millions of lines. As expected, the analysis time is linear to the size of the trace file and the maximum memory usage is constant regardless of the trace file size (see table 1). The platform for experiment is a dual 1.5GHz Athlon system with 1GB of memory.

Table 1. Results of Constraint (8) on EXPR

Lines of Traces	10^4	10^5	10^6	10^7
Time Used (s)	< 1	1	12	130
Memory Usage	8KB	8KB	8KB	8KB

4.2 FIR Filter

Figure 7 shows a 16-tap FIR filter that 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.

Figure 7. FIR Design Example

We utilize our automatic trace checker generator and verify the properties specified in constraints (1) - (5). The same trace files are used for all the analysis. The time and memory requirements are shown in table 2. 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 verification of common real-time constraints is indeed very efficient.

Table 2. Result of Constraints (1 – 5) on FIR

Lines of Traces		10^4	10^5	10^6	10^7
Constraint (1)	Time(s)	< 1	1	8	89
	Memory	28B	28B	28B	28B
Constraint (2)	Time(s)	< 1	1	12	120
	Memory	28B	28B	28B	28B
Constraint (3)	Time(s)	< 1	1	7	80
	Memory	24B	24B	24B	24B
Constraint (4)	Time(s)	< 1	1	7	77
	Memory	0.4KB	0.4KB	0.4KB	0.4KB
Constraint (5)	Time(s)	< 1	1	7	79
	Memory	4KB	4KB	4KB	4KB

We also verify constraint (6) using the simulation analyzer approach. 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 with the algorithm in the previous section 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 memory. If the memory is a limiting factor in the simulation environment, the analysis speed must be sacrificed to allow the verification to continue. This is shown in Table 3 where the memory usage is 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, however, the memory space can be recycled and the memory requirements are small.

Table 3. Result of Constraint (6) on FIR

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

5. CONCLUSION

In this paper we have presented a methodology for system-level verification through automatic trace analysis. We have demonstrated how we take any formula written in the formal quantitative constraint language, Logic Of Constraints, and automatically generate a trace checker that can efficiently analyze the simulation traces for constraint violations. The analyzer is fast even under memory limitation. We have applied the methodology to many case studies and demonstrate that automatic LOC trace analysis can be very useful.

We are currently considering a few future enhancements and novel applications. One such application we are considering is to integrate the LOC analyzer with a simulator that is capable of non-deterministic simulation, non-determinism being crucial for design at high level 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 will not violate the constraint.

ACKNOWLEDGEMENTS

We gratefully acknowledge the preliminary work by Artur Kedzierski who did experiments on LOC formula parsing and checker generation. We also would like to thank Lingling Jin who wrote and debug the Metropolis Meta-Model source code for the EXPR example.

REFERENCES

1. K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. " System level design: orthogonalization of concerns and platform-based design". *IEEE Transactions on Computer-Aided Design*, 19(12), December 2000.
2. 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*, November 2001.
3. X. Chen, F. Chen, H. Hsieh, F. Balarin, and Y. Watanabe. " Formal verification of embedded system designs at multiple levels of abstraction". *Proceedings of International Workshop on High Level Design Validation and Test - HLDVT02*, September 2002.
4. 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.
5. T. Hafer and W. Thomas. " Computational tree logic and path quantifiers in the monadic theory of the binary tree". *Proceedings of International Colloquium on Automata, Languages, and Programming*, July 1987.
6. Gerard J. Holzmann. "The model checker Spin". *IEEE Transactions on Software Engineering*, 23(5):279-258, May 1997.
7. Spin manual, " http://netlib.bell-labs.com/netlib/spin/whatispin.html".
8. FormalCheck, " http://www.cadence.com/products/formalcheck.html".
9. K. McMillan. " Symbolic Model Checking". *Kluwer Academic Publishers*, 1993.
10. F. Balarin, Y. Watanabe, J. Burch, L. Lavagno, R. Passerone, and A. Sangiovanni-Vincentelli. " Constraints specification at higher levels of abstraction". *Proceedings of International Workshop on High Level Design Validation and Test - HLDVT01*, November 2001.
11. C. Blank, H. Eveking, J. Levihn, and G. Ritter. " Symbolic simulation techniques: state-of-the-art and applications". *Proceedings of International Workshop on High-Level Design Validation and Test – HLDVT01*, November 2001.
12. A. Aziz, F. Balarin, R.K. Brayton and A. Sangiovanni-Vincentelli. " Sequential synthesis using S1S". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(10):1149-62, October 2000.
13. P. Alexander, C. Kong, and D. Barton. " Rosetta usage guide". *http://www.sldl.org*, 2001.
14. Quick reference guide for the Design Constraints Description Language, *http://www.eda.org/dcwg*, 2000.
15. Object Constraint Language specification, *http://www.omg.org*, 1997.
16. E. Lee and D. Messerschmitt. " Synchronous data flow". *Proceedings of IEEE*, 55-64, September 1987.
17. Ptolemy home page, " http://www.ptolemy.eecs.berkeley.edu".
18. SystemC home page, " http://www.systemc.org".