CHIMP: a Tool for Assertion-Based Dynamic Verification of SystemC Models

Sonali Dutta
Rice University
Email:Sonali.Dutta@rice.edu

Deian Tabakov
Schlumberger
Email:deian@dtabakov.com

Moshe Y. Vardi
Rice University
Email:vardi@cs.rice.edu

***Abstract*—CHIMP is a tool for assertion-based dynamic verification of SystemC models. The various features of CHIMP include automatic generation of monitors from temporal assertions, automatic instrumentation of the model-under-verification (MUV), and three-way communication among the MUV, the generated monitors, and the SystemC simulation kernel during the monitored execution of the instrumented MUV. Empirical results show that CHIMP puts minimal runtime overhead on the monitored execution of the MUV.**

A newly added path in CHIMP results in a significant (over 75%) reduction of average monitor generation and compilation time. The average size of the monitors is reduced by over 60%, without increasing runtime overhead.

I. Introduction

SystemC (IEEE Standard 1666-2005) has emerged as a de facto standard for modeling of hardware/software systems [4], supporting different levels of abstraction, iterative model refinement, and execution of the model during each design stage. SystemC is implemented as a C++ library with macros and base classes for modeling processes, modules, channels, signals, ports, and the like, while an event-driven simulation kernel allows efficient simulation of concurrent models. Thus, on one hand, SystemC leverages the natural object-oriented encapsulation, data hiding, and well-defined inheritance mechanism of C++, and, on the other hand, it allows modeling and efficient simulation of hardware/software designs by its simulation kernel and predefined hardware-specific macros and classes. The SystemC code is available as open source, including a single-core reference simulation kernel, referred to as the *OSCI kernel*; see http://www.accellera.org/downloads/standards/systemc.

Work supported in part by NSF grants CNS 1049862 and CCF-1139011, by NSF Expeditions in Computing project "ExCAPE: Expeditions in Computer Augmented Program Engineering", by BSF grant 9800096, and by gift from Intel.

The growing popularity of SystemC has motivated research aimed at assertion-based dynamic verification (ABDV) of SystemC models [9]. ABDV involves two steps: generating run-time monitors from input assertions [8], and executing the model-under-verification (MUV) while running the monitors along with the model. The monitors observe the execution of the MUV and report if the observed behavior is consistent with the specified behavior. For discussion of related work in ABDV of SystemC see [7].

CHIMP, available as an open-source tool[1], implements the monitoring framework for temporal SystemC properties described in [9]–see discussion below. CHIMP consists of (1) *off-the-self components*, (2) *modified components*, and (3) an *original component*. CHIMP has two *off-the-self components*: (1) `spot-1.1.1` [3], a C++ library used for LTL-to-Büchi conversion , and (2) `AspectC++-1.1` [6], a C++ aspect compiler that is used for instrumentation of the MUV. CHIMP has two *modified components*: (1) a patched version of the `OSCI kernel-2.2` to facilitate communication between the kernel and the monitors [9], and (2) an extension of `Automaton-1.11` [5], a Java tool used for determinization and minimization of finite automata, with the ability to read automata descriptions from file. Finally, the *original component* is `MONASGEN`, a C++ tool for automatic generation of monitors from assertions [8] and for automatic generation of an aspect advice file for instrumentation [11]. Fig. 1 shows the five components of CHIMP as described above. The component Automaton-1.11 is dotted because a newly added improved path in CHIMP has been able to remove the dependency of CHIMP on Automaton-1.11 (explained in Section V).

CHIMP takes the MUV and a set of temporal assertions about the behavior of that MUV as inputs,

[1] www.sourceforge.net/projects/chimp-rice

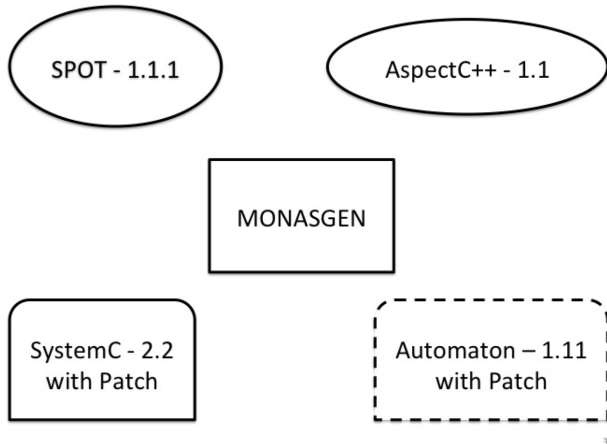

Fig. 1. CHIMP components

and outputs "FAILED" or "NOT FAILED", for each assertion. CHIMP performs white-box validation of user code and black-box validation of library code. (If a user wishes to do white-box validation of library code, it can be accomplished by treating the library code as part of the user code.) The two main features of CHIMP are: (1) CHIMP generates C++ monitors, one for each assertion to be verified, and (2) CHIMP automatically instruments the user model [11] to expose the user model's states and syntax to the monitors. CHIMP puts nominal overhead on the runtime of the MUV, supports a rich set of assertions, and can handle a wide array of abstractions, from statement level to system level.

A recently added path in CHIMP from LTL to monitor can bypass the old performance bottleneck, Automaton-1.1, and improves the monitor generation and compilation time by a significant amount. It also reduces the size of the generated monitors notably. This entirely removes the dependency of CHIMP on Automaton-1.1.

The theoretical and algorithmic foundations of CHIMP were described in [8]–[11]. In this paper we describe the architecture, usage, and recent evolution of CHIMP. The rest of the paper is organized as follows. Section II describes the syntax and semantics of assertions. Section III presents an overall picture about the usage, implementation and performance of CHIMP. Section IV describes the C++ monitor generated by CHIMP. Section V describes the new improved path in CHIMP and its improved performance. Finally, Section VI presents a summary and talks about future work.

II. ASSERTIONS: SYNTAX AND SEMANTICS

CHIMP accepts input assertions, defined using the temporal specification framework for SystemC described in [10], where a temporal assertion consists of a linear temporal formula accompanied by a Boolean expression serving as a clock. This supports assertions written at different levels of abstraction with different temporal resolutions. The framework of [10] proposes a set of SystemC-specific Boolean variables that refer to SystemC's software features and its simulation semantics; see examples below. Input assertions in CHIMP are of the form "$\langle LTL\ formula\rangle @\langle clock\ expression\rangle$", where the $LTL\ formula$ expresses a temporal property and the $clock\ expression$ denotes when CHIMP should sample the execution trace of the MUV.

Several of the additional Boolean variables proposed in [10] refer to the simulation phases. According to SystemC's formal semantics, there are 18 predefined kernel phases. CHIMP has Boolean variables that enable referring to these phases. For example MON_DELTA_CYCLE_END denotes the end of each delta cycle and MON_THREAD_SUSPEND denotes the suspension moment of each thread. By using these variables as clock expressions, we can sample the execution trace at different temporal resolutions. (By default, CHIMP samples at each kernel phase.) Other Boolean variables refer to SystemC events, which are key elements of SystemC's event-driven simulation semantics. CHIMP is also able to sample the execution at various key locations, e.g., function calls and function returns. This gives the user the flexibility to write assertions at different levels of abstraction, from the level of individual C++ statements to the level of SystemC kernel phases.

The Boolean primitives supported by CHIMP are summarized below; see, [10] and [11] for further details:

Function primitives: Let `f()` be a C++ function in the user or library code. The Boolean primitives **f:call** and **f:return** refer to locations in the source code that contain the function call, and to locations immediately after the function call, respectively. The primitives **f:entry** and **f:exit** refer to the locations immediately before the first executable statement and immediately after the last executable statement in `f()`, respectively. If `f()` is a library function then the entry and exit primitives are not supported (black-box verification model for libraries).

Value primitives: If a function `f()` has k arguments, CHIMP defines variables $f : 1, \ldots, f : k$, where the value and type of $f : i$ are equal to the value and type

of the i_{th} parameter of function `f()` before executing the first statement in the definition of `f()`. CHIMP also defines the variable $f : 0$, whose value and type are equal to the value and type of the object that `f()` returns. For example, if the function `int divide(int dividend, int divisor)` is defined in the MUV, then the formula *G (division:entry -> "division:2 != 0")* asserts that the divisor is nonzero whenever division function starts execution.

Phase primitives: The user can refer to the 18 predefined kernel states [10] in the assertions to specify when the state of the MUV should be sampled. For example, the assertion *G ("p == 0") @ MON_DELTA_CYCLE_END* requires the value of variable p to be zero at the end of every delta cycle.

Event primitives: For each SystemC event E, CHIMP provides a Boolean primitive `E.notified` that is true only when the OSCI kernel actually notifies E. For example, the assertion *G (s.value_changed_event().notified) @MON_UPDATE_PHASE_END* says that the signal `s` changes value at the end of every update phase.

III. Usage, Implementation and Performance

Running CHIMP consists of three steps: (1) In the first step, the user writes a configuration file containing all assertions to be verified, as well as other necessary information. The user can also provide inputs through command-line switches. For each LTL assertion in the configuration file, MONASGEN first generates a nondeterministic Büchi automaton on words (NBW) using SPOT, then converts that NBW to a minimal deterministic finite automaton on words (DFW), using the Automaton-1.11 tool for determinization and minimization. Then, MONASGEN generates the C++ monitors from the DFW, one monitor for each assertion (Fig. 2). (2) MONASGEN produces an aspect-advice file that is then used by AspectC++ to generate the instrumented MUV (Fig. 3). (3) Finally, the monitors and the instrumented MUV are compiled together and linked to the patched OSCI kernel, and the code is then run to execute the simulation with inputs provided by user. The inputs can be generated using any standard stimuli-generation technique. For every assertion, CHIMP produces output indicating if the assertion held or failed for that particular input (Fig. 4).

For experimental evaluation we used a SystemC model with about 3,000 LOC, implementing a system for reserving and purchasing airplane tickets. The users

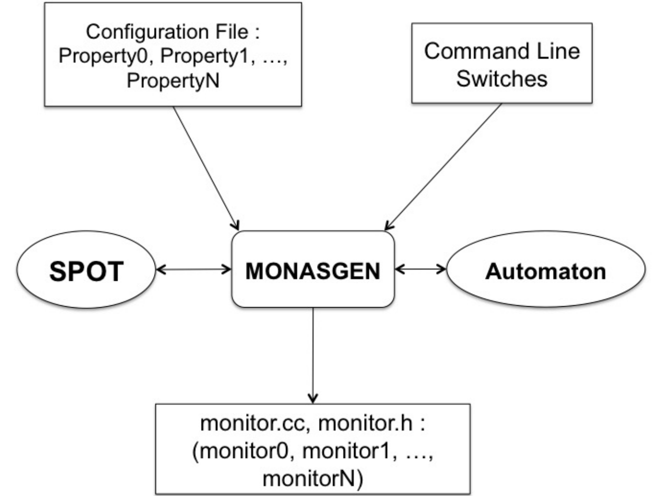

Fig. 2. Monitor generation flow

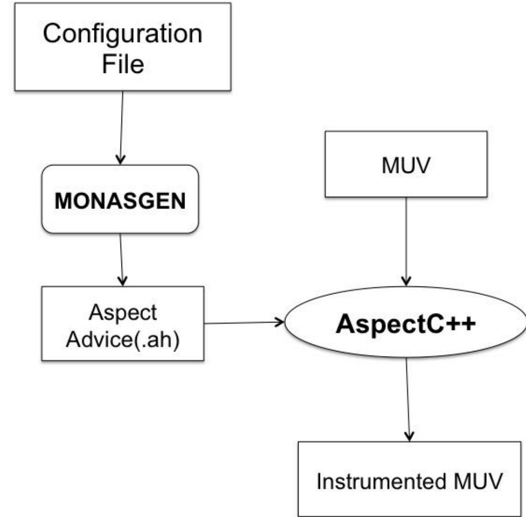

Fig. 3. MUV instrumentation flow

of the system submit requests and the system uses a randomly generated flight database to find a direct flight or a sequence of up to three connecting flights. Those are returned to the user for approval, payment and booking. This model is intended to run forever. It is inspired by actual request/grant subsystems currently used in hardware design.

We used a patched version of the 2.2.0 OSCI kernel and compiled it using the default settings in the Makefile. The empirical results below were measured on a Pentium 4, 3.20GHz CPU, 1 GB RAM machine running GNU

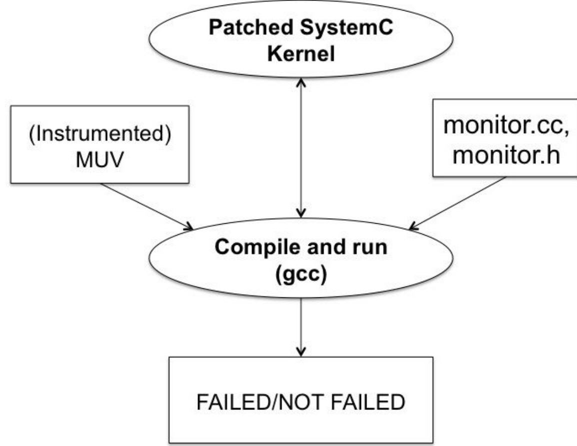

Fig. 4. Running instrumented MUV with monitors using patched OSCI kernel

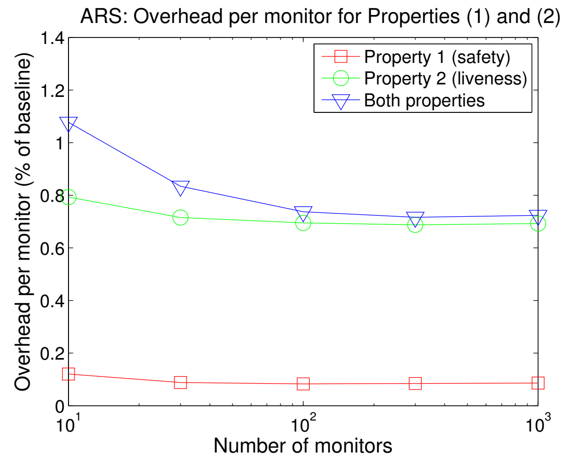

Fig. 5. Run time overhead for monitoring Properties (1) and (2)

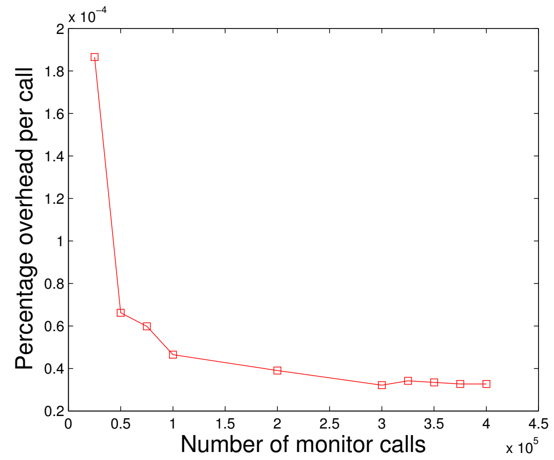

Fig. 6. Instrumentation overhead per monitor call as a percentage of the baseline run time. Y axis is ((instrumentation overhead $-$ baseline)/(baseline $\times$ number of monitors)) $\times$ 100%

Linux. To assess runtime overhead imposed by the monitors, we measured the effect of running with different assertions and also increasing number of assertions [9]. In each case we first ran the model without monitors and user-code instrumentation to establish the baseline, and then ran several simulations with instrumentation and an increasing number of copies of the same monitor. The results report runtime overhead per monitor as a percentage of the baseline. (For these experiments monitors were constructed manually.)

The first property we checked is a safety property asserting that whenever the event *new_requests_nonfull* is notified, the corresponding queue *new_planning_requests* must have space for at least one request.

```
 G "new_planning_requests.size() < capacity"
@ new_requests_nonfull.notified
```
(1)

The second property says that the system must propagate each request through each channel (or through each module) within 5 cycles of the slow clock. This property is a conjunction of 16 bounded liveness assertions similar to the one shown here.

```
...
// Propagate through module within 5 clock ticks
ALWAYS (io_module_receive_transaction($1) ->
  ( within [5 slow_clock.pos()]
    io_module_send_to_master($2) & ($1 == $2)
  ) AND ...
```
(2)

Fig. 5 presents the runtime overhead of monitoring the above two properties as a percentage of the baseline. Checking Property 2 is relatively expensive because it requires a lot of communication from the model to the monitor. It is shown in [1] that the finite state monitors have less runtime overhead than transaction-based monitors generated by tools like Synopsys VCS.

We also evaluated the cost of instrumentation separately by simulating one million SystemC clock cycles with focus on the overhead of instrumentation [11]. The average wall-clock execution time of the system over 10 runs without instrumentation was 33 seconds. We call this “baseline execution”. Fig. 6 shows the cost of the instrumentation per monitor call, as a percentage of the

baseline execution. The data suggest that there is a fixed cost of the instrumentation, which, when amortized over more and more calls, leads to lower average cost. The average cost per call stabilizes after 300,000 calls, and is less than $0.5 \times 10^{-4}\%$.

Another testbench we used is an Adder model that implements the squaring function by repeated increment by 1. It uses all three kinds of event notifications. It is scalable as it spawns a SystemC thread for each addition in the squaring function. We monitored several properties of the Adder model using CHIMP.

To study the effect of monitor encoding on runtime overhead, Tabakov and Vardi [8] describes 33 different monitor encodings, and shows that front_det_switch encoding with state minimization and no alphabet minimization is the best in terms of runtime overhead. The downside is that this flow suffers from a rather slow monitor generation and compilation time. This led us to develop a new flow for the tool, as described below.

IV. CHIMP MONITORS

In CHIMP, the $LTL\ formula$ in every assertion $\langle LTL\ formula\rangle$ @ $\langle clock\ expression\rangle$ is converted into a C++ monitor class. Each C++ monitor has a **step()** function that implements the transition function of the DFW generated from the $LTL\ formula$ (as described in Fig. 9). This step() function is called at every sampling point defined by $clock\ expression$. The monitor.h and monitor.cc files, generated by MONASGEN, contains one monitor class for every assertion and a class called `local_observer` that is responsible for invoking the callback function, which invokes *step()* function of the appropriate monitor class at the right sampling point during the monitored simulation. Different encodings have been explored for writing the monitor's *step()* function. For the new flow of CHIMP (described below), Fig. 13 shows that front_det_ifelse encoding is the best among all of them in terms of runtime overhead.

In front_det_ifelse encoding, the C++ monitor produced is deterministic in terms of transitions. This means from one state, either one or no transition is possible. If no transition is possible from a state, the monitor rejects and outputs "FAILED". Else, the monitor keeps executing until the end of simulation and outputs "NOT FAILED". In front_det_ifelse encoding, each state of the monitor is encoded as an integer, from 0 upto the total number of states. The *step()* function of the monitor uses an outer if-elseif statement block to determine the next state. The possible transitions from each state are encoded using an inner if-elseif block, the condition statement being the guard on a transition.

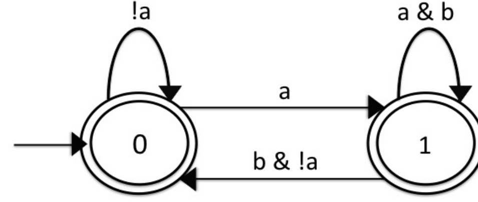

Fig. 7. The DFW generated by MONASGEN from the LTL formula $G(a \rightarrow Xb)$. All the states are accepting. The DFW rejects bad prefixes by no available transition. State 0 is the initial state.

As a running example, we show how the monitor *step()* function is encoded for an assertion $G(a \rightarrow Xb)$ @ clk. clk here is some boolean expression specifying when the *step()* function needs to be called during the simulation. This assertion asserts that, always, if a is true, then in the next clock cycle b has to be true. The DFW generated by CHIMP for this assertion is shown in Fig. 7.

Listing 1 shows the *step()* function of the C++ monitor generated by CHIMP for the assertion $G(a \rightarrow Xb)$ @ clk. The variables `current_state` and `next_state` are member variables of the monitor class and store the current automaton state and next automaton state respectively. If next state becomes -1 after the execution of the *step()* function, it means that no transition can be made from the current state. In this case, the monitor outputs "FAILED". The initial state 0 of the monitor is assigned to the variable `current_state` inside the constructor of the monitor class as shown in Listing 2.

Listing 1. The *step()* function of the monitor for $G(a \rightarrow Xb)$

```
/**
* Simulate a step of the monitor.
*/
void
monitor0::step() {
  //If the property has not failed yet
  if (status == NOT_FAILED) {
    //number of steps executed so far
    num_steps++;
    //assign next state to current state
    current_state = next_state;
    //make next state invalid
    next_state = -1;
```

```
    if (current_state == 0) {
      if(!(a)  )
        { next_state = 0; }
      else if((a)  )
        { next_state = 1; }
    } // if (current_state == 0)

    else if (current_state == 1) {
      if((b) && !(a)  )
        { next_state = 0; }
      else if((a) && (b)  )
        { next_state = 1; }
    } // if (current_state == 1)

  //FAILED if no transition possible
   bool not_stuck = (next_state != -1);
   if (! not_stuck) {
     property_failed();
   }
  } // if (status == NOT_FAILED)
} // step()
```

Listing 2. The constructor of the monitor for $G(a \rightarrow Xb)$

```
/**
 * The constructor
 */
monitor0::monitor0(...):
sc_core::mon_prototype() {
  next_state = 0; // initial state id
  current_state = -1;
  status = NOT_FAILED;
  num_steps = 0;
  . . .
} // Constructor
```

The sampling points can be kernel phases, e.g., MON_DELTA _CYCLE_END, or event notification, e.g., E.notified (E is an event). In such cases, the OSCI kernel needs to communicate with the `local_observer` at the right time (when a delta cycle ends or when event E is notified) to call the *step()* function of the monitor. This communication is done by the patch, put on OSCI kernel. This patch contains a class called `mon_observer`, which communicates with the `local_observer` class on behalf of the OSCI kernel.

V. Evolution of CHIMP

Fig. 8 shows the old path of generating C++ monitor from an LTL property in CHIMP. First, MONASGEN

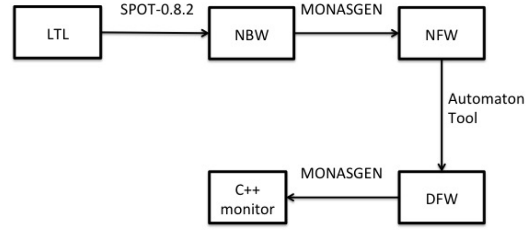

Fig. 8. Old LTL to monitor path in CHIMP

Fig. 9. New LTL to monitor path in CHIMP

uses SPOT to convert the LTL formula to a nondeterministic Büchi Automaton (NBW). Then MONASGEN prunes the NBW to remove all states that do not lead to any accepting state, and generate the corresponding nondeterministic finite automaton (NFW), see [2]. MONASGEN then uses the Automaton Tool to determinize and minimize the NFW to generate minimal deterministic finite automaton (DFW). Finally MONASGEN converts the DFW to C++ monitor.

The main bottleneck in this flow was minimization and determination of NFW using the Automaton tool as it consumes 90% of total monitor generation and compilation time. Also, the Automaton tool may generate multiple edges between two states, resulting in quite large monitors. The newest version of CHIMP introduces a new path as shown in Fig. 9, which bypasses Automaton tool completely and uses only SPOT. So the component Automaton-1.11 in Fig. 1 is not needed by CHIMP anymore. This new path leverages the new functionality of SPOT-1.1.1 to convert an LTL formula to a minimal DFW that explicitly rejects all bad prefixes.

After replacing the Automaton Tool by SPOT to convert the NBW to minimal DFW, the improvement in compilation time and monitor size is evident. This new flow results in 75.93% improvement in monitor generation and compilation time. To evaluate the performance of the revised CHIMP, we use the same set of 162 pattern formulas and 1200 random formulas as mentioned in [8]. The scatter plot on Fig. 10 shows the comparison of monitor generation and compilation time of the new flow vs the old flow. Most of the points are above the line with slope 1, which indicates that for most of the monitors, the

generation and compilation time by old CHIMP is more than by new CHIMP. Fig. 10 - Fig. 14 are all scatter plots[2] and interpreted in the same way.

The new flow also merges multiple edges between two states into one edge guarded by the disjunction of the guards on all edges between the states. In this way the average reduction in monitor size in bytes is 61.27%. Fig. 11 shows the comparison of the size in bytes of the monitors generated by the new flow vs the old flow.

Since the main focus of CHIMP has always been minimizing runtime overhead, we need to ensure that this new flow does not incur more runtime overhead compared to the old flow as a cost of reduced monitor generation and compilation time. So we ran the same set of 162 pattern formulas and 1200 random formulas as mentioned above, to compare the runtime overhead incurred by the new flow vs the old flow. Fig. 12 shows that the runtime overhead of CHIMP using the new flow has been reduced compared to the old flow. The reduction is 7.97% on average.

As CHIMP has evolved to follow a different and more efficient path and the monitors have been reduced in size, it was not clear a priori which monitor encoding is the best. We conducted the same experiment as in [8] with the same set of formulas, 162 pattern formulas and 1200 random formulas, to identify the best encodings in terms of both runtime overhead and monitor generation and compilation time. We identified two new best encodings. Fig. 13 shows that the new best encoding in terms of runtime overhead is now front_det_ifelse. Fig. 14 shows that the new best encoding in terms of monitor generation and compilation time is back_ass_alpha. CHIMP now provides two configurations for monitor generation, best_runtime, which has minimum runtime overhead and best_compiletime, which has minimum monitor generation and compilation time. Since the bigger concern is usually runtime overhead, best_runtime is the default configuration, given that its monitor generation and compilation time is very close to that of best_compiletime.

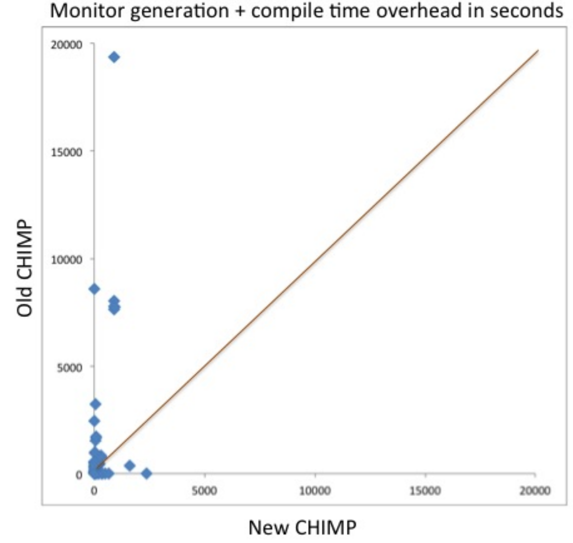

Fig. 10. Comparison of monitor generation and compilation time of the old CHIMP vs new CHIMP

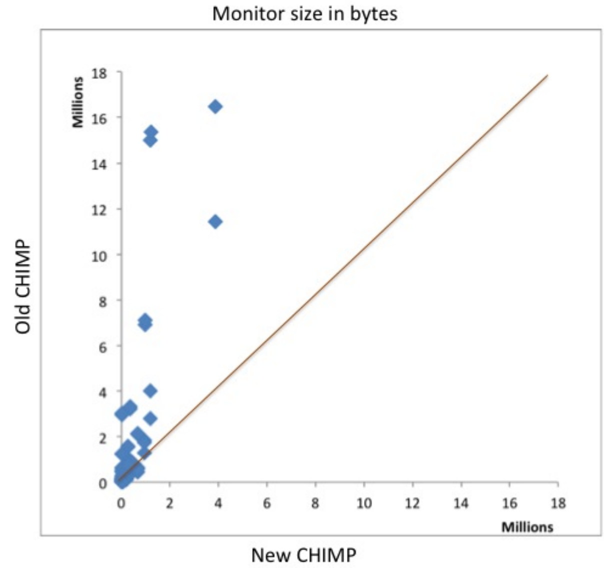

Fig. 11. Comparison of monitor size (bytes) generated by the old CHIMP vs new CHIMP

VI. Conclusion

We present CHIMP, an Assertion-Based Dynamic Verification tool for SystemC models, and show that it puts minimal overhead on the runtime of the MUV. We show how the new path in CHIMP results in significant reduction of monitor generation and compilation time and monitor size, as well as runtime overhead. In the future we plan to look at the possibility of verifying parametric properties, for example $G(send(id) \rightarrow F(receive(id)))$ where one can pass a parameter (here id), to the variables in the LTL formula of the assertion. The above property means that always if the message with ID id is sent, it should be received eventually. Also at present the user needs to know the system architecture to declare an assertion. We would like to make CHIMP work with assertions that are declared in the elaboration phase.

[2]http://en.wikipedia.org/wiki/Scatter_plot

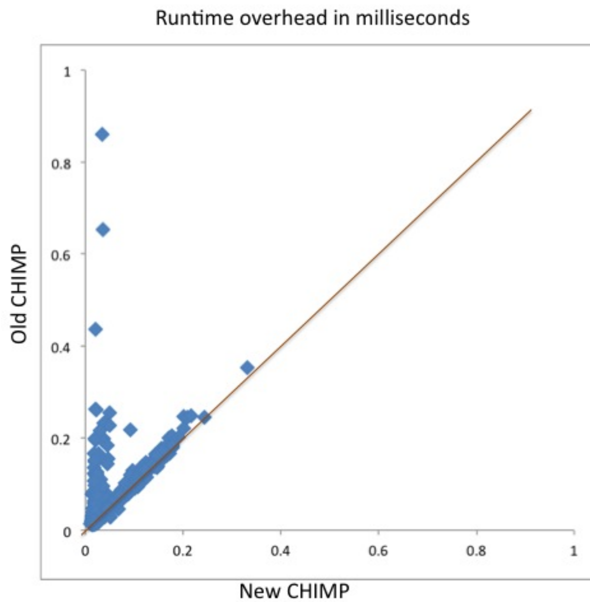

Fig. 12. Comparison of runtime overhead incurred by old CHIMP vs new CHIMP

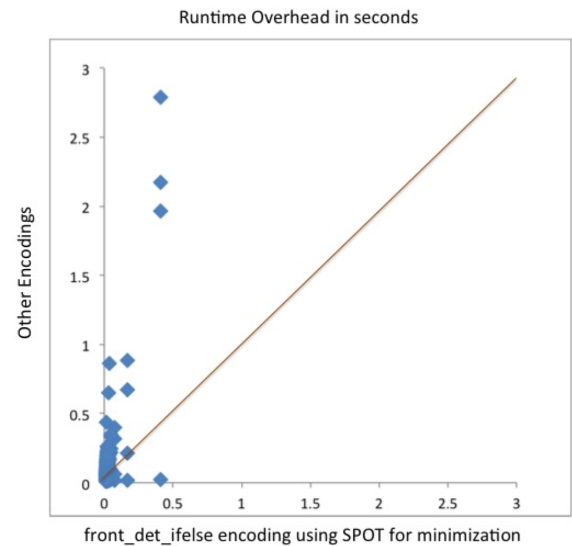

Fig. 13. Comparison of runtime overhead of front_det_ifelse encoding vs all other encodings

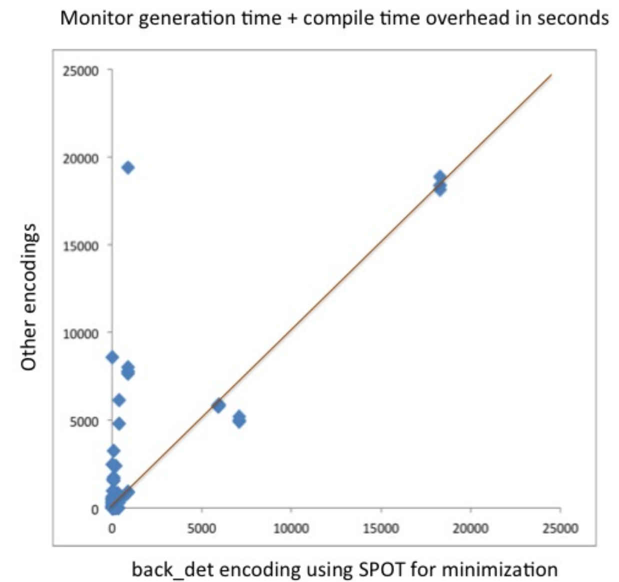

Fig. 14. Comparison of monitor generation time and compile time overhead of back_ass_alpha encoding vs all other encodings

REFERENCES

[1] R. Armoni, D. Korchemny, A. Tiemeyer, M. Vardi, and Y. Zbar. Deterministic dynamic monitors for linear-time assertions. In *Proc. Workshop on Formal Approaches to Testing and Runtime Verification*, volume 4262 of *Lecture Notes in Computer Science*. Springer, 2006.

[2] M. d'Amorim and G. Roşu. Efficient monitoring of ω-languages. In *Proc. 17th International Conference on Computer Aided Verification*, pages 364–378, 2005.

[3] A. Duret-Lutz and D. Poitrenaud. SPOT: An extensible model checking library using transition-based generalized Büchi automata. *Modeling, Analysis, and Simulation of Computer Systems*, 0:76–83, 2004.

[4] C. Helmstetter, F. Maraninchi, L. Maillet-Contoz, and M. Moy. Automatic generation of schedulings for improving the test coverage of Systems-on-a-Chip. In *FMCAD '06: Proceedings of the Formal Methods in Computer Aided Design*, pages 171–178, Washington, DC, USA, 2006. IEEE Computer Society.

[5] A. Møller. dk.brics.automaton. http://www.brics.dk/automaton/, 2004.

[6] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: an aspect-oriented extension to the C++ programming language. In *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific*, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[7] D. Tabakov. *Dynamic Assertion-Based Verification for SystemC*. PhD thesis, Rice University, Houston, 2010.

[8] D. Tabakov, K. Rozier, and M. Y. Vardi. Optimized temporal monitors for SystemC. *Formal Methods in System Design*, 41(3):236–268, 2012.

[9] D. Tabakov and M. Vardi. Monitoring temporal SystemC properties. In *Proc. 8th Int'l Conf. on Formal Methods and Models for Codesign*, pages 123–132. IEEE, July 2010.

[10] D. Tabakov, M. Vardi, G. Kamhi, and E. Singerman. A temporal language for SystemC. In *FMCAD '08: Proc. Int. Conf. on Formal Methods in Computer-Aided Design*, pages 1–9. IEEE Press, 2008.

[11] D. Tabakov and M. Y. Vardi. Automatic aspectization of SystemC. In *Proceedings of the 2012 workshop on Modularity in Systems Software*, MISS '12, pages 9–14, New York, NY, USA, 2012. ACM.