Efficient Generation of Monitor Circuits for GSTE Assertion Graphs

Alan J. Hu*
Dept. of Computer Science
University of British Columbia
ajh@cs.ubc.ca

Jeremy Casas
Strategic CAD Lab
Intel Corporation
jeremy.casas@intel.com

Jin Yang
Strategic CAD Lab
Intel Corporation
jin.yang@intel.com

ABSTRACT

Generalized symbolic trajectory evaluation (GSTE) is a powerful, new method for formal verification that combines the industrially-proven scalability and capacity of classical symbolic trajectory evaluation with the expressive power of temporal-logic model checking. GSTE was originally developed at Intel and has been used successfully on Intel's next-generation microprocessors. However, the supporting algorithms and tools for GSTE are still relatively immature.

GSTE specifications are given as *assertion graphs*, an extension of $\forall$-automata. This paper presents a linear-time, linear-size translation from GSTE assertion graphs into monitor circuits, which can be used with dynamic verification both as a quick "sanity check" of the specification before effort is invested in abstraction and formal verification, and also as means to reuse GSTE specifications with other validations methods. We present experimental results using real GSTE assertion graphs for real industrial circuits, showing that the circuit construction procedure is efficient in practice and that the monitor circuits impose minimal simulation overhead.

1. INTRODUCTION

Generalized symbolic trajectory evaluation (GSTE) is a powerful, new method for formal design verification [24]. GSTE is based on classical symbolic trajectory evaluation [20], which has proven itself able to handle large, industrial designs and has been in active use at Compaq (now HP), IBM, Intel, and Motorola (e.g., [16, 14, 1, 6]). Classical symbolic trajectory evaluation, although efficient, is very limited in the types of properties that it can specify and verify. GSTE extends classical symbolic trajectory evaluation to handle a full range of temporal properties (all ω-regular properties), giving it comparable expressive power to more established model-checking approaches [8, 17, 22, 12], while still maintaining the efficiency and capacity of classical symbolic trajectory evaluation. GSTE was originally developed at Intel and has been used successfully on Intel's next-generation microprocessors, where users reported superior efficiency and capacity for some challenging formal verification tasks [4].

However, a formal verification algorithm alone, no matter how automatic or efficient, does not constitute an entire verification flow. In practice, numerous supporting algorithms and tools are needed to connect any given formal verification method to the overall verification effort. Especially needed for formal verification are the abilities to quickly debug and revise specifications before substantial effort is invested in the formal verification process, to generate counter-examples and diagnose the causes of bugs, to relate and compose smaller verification results to solve a larger verification problem, and to bridge between different verification methods (e.g., formal vs. simulation-based) and between different levels of abstraction (e.g., system-level vs. RTL). GSTE, being a very recent development, currently has less of this supporting infrastructure than older formal verification methods do. For example, the efficiency of GSTE model checking relies, in part, on the particular specification style used, but no work has been published connecting GSTE specifications to other verification methods.

An especially useful piece of methodological glue is the *monitor circuit.* A monitor is simply a small circuit that watches, without interfering, the system being verified and flags whether or not the system is obeying some user-specified correctness property. Implementing the monitor as a circuit (rather than in, for example, a formal specification language) allows the same monitor to be used at all levels of the design cycle and with both formal and informal verification tools.[1] Extensive research has demonstrated the value of monitor circuits as the cornerstone of a practical verification methodology [3], as an enabler of hierarchical, compositional verification [11, 21, 10], and as a testbench generator for simulation [26]. Monitor circuits could even be synthesized into an emulation system to aid error observability and debugging.

This paper presents a linear-time, linear-size translation from the specifications used by GSTE into monitor circuits, thereby enabling new ways to integrate GSTE into the verification flow. Our generated monitor circuits handle fully general GSTE specifications if the simulator can perform a small amount of symbolic simulation; we also describe how GSTE specifications with some restrictions could be translated into monitor circuits suitable for fully scalar simulation or emulation. The immediate application for our translation is to allow quick, (symbolic-)simulation-based "sanity checks" of GSTE specifications before trying to apply model checking — in practice, considerable effort is often spent combatting state-space explosion before a formal verification engine yields it's first counter-examples, and we'd like to avoid this effort until we've eliminated simple specification errors. If we can simulate our formal specifications, we can quickly catch many erroneous specifications before investing in trying (and failing) to formally prove them correct. We also envision the generated monitors connecting GSTE-based and monitor-based verification methodologies. In addition, our monitor construction is a building block for initial work on compositional verification with GSTE [9].

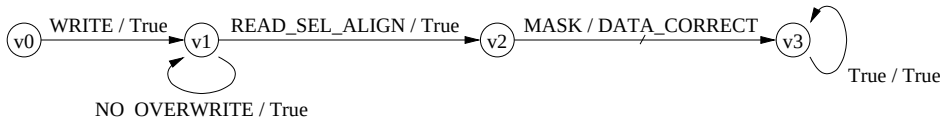
2. GSTE AND ASSERTION GRAPHS

GSTE is explained in detail in several sources (e.g., [24, 25, 23], etc.). Here, we give only a brief overview of the specification style used by GSTE in order to make this paper self-contained.

GSTE is a model-checking method, where the possible behav-

*Work done while visiting Intel SCL.

[1]Obviously, formalisms designed for specifications have other advantages over circuits, so it's very valuable to be able to go from a formal specification to a monitor circuit, hence this paper.



$$
\begin{aligned}
\text{WRITE} &:= (\text{we} = 1) \wedge (\text{addr} = A) \wedge (\text{datawr} = D) \\
\text{NO\_OVERWRITE} &:= (\text{we} = 0) \vee (\text{addr} \neq A) \\
\text{READ\_SEL\_ALIGN} &:= (\text{ck} = 0) \wedge (\text{we} = 0) \wedge (\text{addr} = A) \wedge (\text{sel} = S) \wedge (\text{align} = R) \\
\text{MASK} &:= (\text{ck} = 1) \wedge (\text{maskbegin} = B) \wedge (\text{maskend} = E) \\
\text{DATA\_CORRECT} &:= (\text{dataout} = \text{mask}(\text{align}(\text{select}(D,S),R),B,E))
\end{aligned}
$$

Figure 1: GSTE Assertion Graph Example. The property specified is that a value written to a memory will be read correctly an arbitrary number of cycles later, subject to alignment and masking operations, provided it was not overwritten. Edges are labeled by an antecedent followed by a consequent. Every path through the assertion graph is a temporal assertion: if every antecedent along that path is satisfied in the system being verified at the corresponding clock cycle, then every consequent must be satisfied as well.

iors of the system being verified are considered to be the (usually infinite) set of all possible execution traces, and verification consists of checking that all of these traces obey the specification. The specification in GSTE is called an *assertion graph*, and is basically a special kind of automaton. One can think of the assertion graph as defining the set of execution traces that it accepts (i.e., the execution traces that obey the specification), so the verification problem is to check that the set of execution traces the system can produce is contained in the set of execution traces that the assertion graph accepts (i.e., GSTE model checking follows the language containment paradigm, as advocated by, for example, Cospan [12]).

Figure 1 shows an example assertion graph, adapted from [24]. It was used in the verification of an industrial memory design, which reads and writes data with a variety of selection and alignment options. The property being verified is that, if data value D is written to address A, followed by an arbitrary number of clock cycles that don't overwrite the same address, followed by a read of the address, then the value returned is the value that was written, appropriately aligned and masked. The edge labels are of the form "*antecedent* / *consequent*", where the antecedents and consequents are simply combinational formulas over the state of the system at a given clock cycle. For example, the antecedent WRITE specifies that the value of the write-enable input `we` is high, that the address input `addr` is equal to some value A, and that the data input `datawr` is equal to some value D. The capital letters denoting values, like A, D, etc., are *symbolic constants* that can be equal to any value, making the verification result hold for all possible values of the symbolic constants. A path is a sequence of edges that start from the initial vertex $v0$. A terminal path is a path that ends with a terminal edge (indicated in the figure by a tic-mark on the edge, e.g., the edge from $v2$ to $v3$). A path accepts an execution trace if at least one antecedent on that path fails (evaluates to false on the state of the system at that clock cycle) or if all antecedents and all consequents on the path succeed (evaluate to true on the corresponding clock cycle). Intuitively, a path is an if-then assertion: the antecedents say if the assertion is relevant; the consequents say what must hold in the case that the assertion is relevant. If an antecedent fails, the assertion is vacuously true; if all the antecedents are satisfied, then the consequents must be satisfied as well. The assertion graph as a whole accepts an execution trace if **every** terminal path in the assertion graph accepts that trace.[2] Intuitively, the assertion graph takes a potentially infinite set of assertions about the system and rolls them up into a graph; therefore, every trace must satisfy every assertion (vacuously or otherwise).

To someone familiar with formal verification theory, two characteristics of assertion graphs stand out: the antecedent/consequent labeling of edges, and the graph accepting based on acceptance **for all** paths. The antecedent/consequent style comes from classical symbolic trajectory evaluation [20] and is a natural way to specify temporal properties. For example, timing diagrams are typically interpreted this way (e.g., if some sequence of events happens, then some other events must happen) [2]. In addition, the GSTE model-checking algorithm exploits the explicit antecedent/consequent labeling to limit the search space during fixpoint computation, providing some efficiency gains and aiding user control of the model checker. The "for all paths" acceptance criteria makes assertion graphs a variety of $\forall$-automata [13], which are less familiar than the usual existential acceptance of non-deterministic automata (where a trace is accepted if there exists a corresponding path through the automata), but the $\forall$ semantics provides both usability and efficiency benefits. The usability arises because an assertion graph defines a set of **assertions**, and one typically wants all assertions to be true; in contrast, usually with automata as specifications, the automata directly defines a set of possible **behaviors**, so verification consists of determining if the system's behavior exists in the set provided by the specification. The theoretical basis for the efficiency is that a $\forall$-automata is essentially pre-complemented, so model checking can bypass the expensive step of complementing a non-deterministic automaton. GSTE shares this theoretical efficiency advantage with other approaches that have used $\forall$-automata as specifications [13, 12, 2].

As an aside, we note that assertion graphs are a low-level specification style. On one hand, this low-level orientation gives the user precise, fine-grained control over the GSTE model-checking algorithm, making it easier to avoid blow-up when verifying large designs. On the other hand, in an overall verification flow, translating from a higher-level property specification language into assertion

[2] GSTE theory actually includes four different kinds of acceptance: *strong*, in which all finite paths must be satisfied; *terminal*, described here, in which all paths that end at terminal edges must be satisfied; *normal*, in which all infinite-length paths must be satisfied; and *fair*, in which all fair (generalized Büchi fairness) paths must be satisfied. Since we are building monitor circuits that should work with simulation and synthesis, we are concentrating on terminal acceptance, which includes strong acceptance as a special case.

graphs might enhance ease-of-use. We are not advocating assertion graphs as the ultimate property specification formalism; rather, we accept that assertion graphs are an entry point to an industrially-proven, practically-efficient formal verification approach — GSTE model checking — and seek to provide methodological support for dealing with assertion graphs.

3. MONITOR CIRCUIT CONSTRUCTION

We now present how to construct a monitor circuit for an assertion graph. The construction runs in linear time and produces a circuit that is linear size relative to the size of the assertion graph. Our approach is inspired by very efficient methods for generating circuits directly from regular expressions [19, 18, 15].

The monitor circuit should watch the system being verified and check that the execution trace so far is legal. To check whether the execution trace is legal, the monitor must verify that all (terminal) paths in the assertion graph accept the execution trace. The intuition of our construction is as follows:

> Imagine that the monitor circuit has an internal copy of the assertion graph. The monitor can use this copy to track all paths in the assertion graph by placing tokens on the edges. Each token indicates that there is a path that ends at that edge, on that clock cycle. The tokens move forward one edge at each clock cycle, and fork into multiple tokens when a vertex has multiple outgoing edges (corresponding to multiple continuations of the current path). Each token corresponds to a path, which must accept the execution trace so far.
>
> For example, consider the assertion graph in Figure 1 after, say, 2 clock cycles. At that point, there would be 2 tokens: one on the edge from $v1$ to $v2$, corresponding to the path from $v0$ to $v1$ to $v2$; and another token on the self-loop on $v1$, corresponding to the path from $v0$ to $v1$ and looping to $v1$ again. After another clock cycle, the first token would move to the edge between $v2$ and $v3$, and the second token would fork into two tokens: one on edge $v1$-$v2$, corresponding to path $v0$-$v1$-$v1$-$v2$; the other on self-loop $v1$-$v1$, corresponding to path $v0$-$v1$-$v1$-$v1$.

The actual implementation follows this intuition closely. The structure of the generated circuit itself forms the "copy of the assertion graph". Latches at each edge are set or cleared to indicate the presence or absence of tokens. Vertices are basically fan-out stems, distributing incoming tokens to all outgoing edges. The challenge is to keep the number of tokens finite (and small) to make creating a circuit possible.

The key insight is that paths are almost memoryless. All paths that reach an edge at some point in time share the same future. The only difference between paths is three different kinds of pasts, which we dub "blessed", "happy", and "condemned":

Blessed Blessed paths have had an antecedent fail already. These paths will always accept, from here to eternity, and need not be tracked by the circuit.

Happy Happy paths have had all antecedents and all consequents succeed so far. These paths are currently accepting, but their continuations may or may not accept extensions of the current trace.

Condemned Condemned paths have had all antecedents succeed, but at least one consequent has already failed. These paths do not accept (and therefore, the existence of any condemned paths means the current trace is rejected), but the continuations from these paths may eventually become blessed.

Because of the limited history information required, the circuit can merge all tokens of the same type that arrive at a given edge at the same time. Therefore, the number of latches required to track all the tokens is only two per edge: one to track if any happy paths are at this edge, and the other to track if any condemned paths are at this edge. The circuit structure perfectly matches the structure of the assertion graph, with sub-circuits corresponding to each edge and each vertex of the assertion graph. The circuit works by passing tokens from edge to edge, updating the kinds of tokens depending on whether the current antecedents and consequents succeed or fail.

More formally, given an assertion graph G, we build a monitor circuit such that the set of traces that the assertion graph accepts (under terminal satisfiability) is the same as the set of input sequences that cause the monitor circuit's `accept` output to be high. We assume that the labels on the edges for the antecedents $\text{ant}(e)$ and consequents $\text{cons}(e)$ are given as combinational logic over the signals (e.g., the inputs and state variables) of the system being verified, as well as over the symbolic constant latches (defined below).

- The monitor circuit has inputs that are driven by signals in the system being verified. In particular, each signal name that is mentioned in the assertion graph G is an input to the monitor circuit. There is one additional input `init`. The monitor has one output `accept`.

- For all the symbolic constants in G, the monitor circuit has latches that are initialized to non-deterministic values, and then hold the values indefinitely.

- The monitor circuit is composed of sub-circuits, one for each vertex and each edge in G. These sub-circuits are wired together exactly as the graph G is connected, with two signals, `happy` and `condemned`, between sub-circuits. For example, if directed edge e ends at vertex v, then the $\texttt{happy}_{\texttt{out}}$ and $\texttt{condemned}_{\texttt{out}}$ outputs from the circuit for e connect to a $\texttt{happy}_{\texttt{in}}$ and $\texttt{condemned}_{\texttt{in}}$ input of the circuit for v.

- The sub-circuit for a vertex v is combinational. It basically ORs all incoming $\texttt{happy}_{\texttt{in}}$ signals and fans the result out to all of its $\texttt{happy}_{\texttt{out}}$ outputs. Likewise, it ORs all of it's incoming `condemned` signals and fans the result out on all of its `condemned` outputs.

$$\texttt{happy}_{\texttt{out}} = \bigvee_{\text{incoming edges}} \texttt{happy}_{\texttt{in}}{}^{i}$$

$$\texttt{condemned}_{\texttt{out}} = \bigvee_{\text{incoming edges}} \texttt{condemned}_{\texttt{in}}{}^{i}$$

  The circuit for the initial vertex $v0$ has an additional input which is ORed in with the happy signals. This input is used to start the first token in the circuit at initialization, described below.

- The sub-circuit for an edge e is more complicated (Figure 2). Build combinational circuits $\text{ant}(e)$ and $\text{cons}(e)$ that check the current value on the monitor's inputs against this

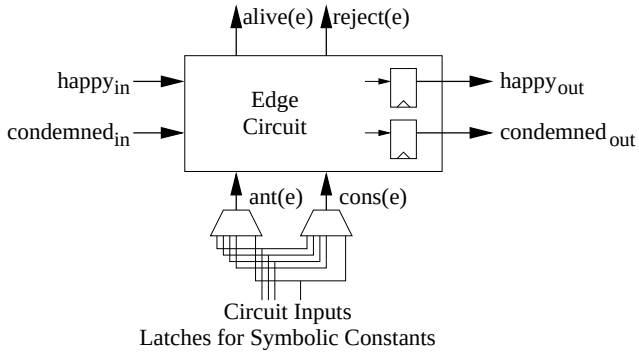


Figure 2: Sub-Circuit for Each Edge. Each edge corresponds to a sub-circuit that receives incoming tokens for happy and condemned paths, updates the tokens depending on whether the current antecedent and consequent pass, and passes the tokens onward on the next clock cycle.

edge's antecedent and consequent. By combining $\texttt{happy}_{\texttt{in}}$, $\texttt{condemned}_{\texttt{in}}$, ant($e$), and cons($e$), we can compute the correct values of $\texttt{happy}_{\texttt{out}}$ and $\texttt{condemned}_{\texttt{out}}$, which will be delayed by one cycle in latches, as well as output signals alive(e) and reject(e), which will be used to determine whether the monitor circuit overall accepts or rejects. The signal alive(e) indicates if there exist any unblessed paths at this edge at this point in time. The signal reject(e) indicates if there exist any condemned paths at this edge at this point in time. Intuitively, alive(e) is just $\texttt{happy} \vee \texttt{condemned}$, and reject($e$) is just condemned. However, to allow the monitor to respond immediately to the current cycle's inputs (Mealy machine), some additional combinational logic is needed:

$$\begin{aligned}
\texttt{happy}_{\texttt{now}} &= \text{ant}(e) \wedge \text{cons}(e) \wedge \texttt{happy}_{\texttt{in}} \\
\texttt{condemned}_{\texttt{now}} &= (\text{ant}(e) \wedge \texttt{condemned}_{\texttt{in}}) \vee \\
&\quad (\text{ant}(e) \wedge \neg\text{cons}(e) \wedge \texttt{happy}_{\texttt{in}}) \\
\text{alive}(e) &= \texttt{happy}_{\texttt{now}} \vee \texttt{condemned}_{\texttt{now}} \\
\text{reject}(e) &= \texttt{condemned}_{\texttt{now}} \\
\texttt{happy}_{\texttt{out}} &= \text{DFF}(\texttt{happy}_{\texttt{now}}) \\
\texttt{condemned}_{\texttt{out}} &= \text{DFF}(\texttt{condemned}_{\texttt{now}})
\end{aligned}$$

- The global accept output of the monitor is based on all terminal edges. Basically, if an edge is alive, it must not be rejecting:

$$\texttt{accept} = \bigwedge_{\text{all terminal edges } e} (\text{alive}(e) \Rightarrow \neg\text{reject}(e))$$

- Initialization is accomplished by asserting the init signal. When init is asserted, the outputs of all the edge latches ($\texttt{happy}_{\texttt{out}}$ and $\texttt{condemned}_{\texttt{out}}$) are forced to 0. (This happens combinationally because the construction creates a Mealy machine.) In addition, a 1 is asserted on the extra initialization input for the initial vertex $v0$. For certain kinds of assertion graphs (basically, those that are expected to hold in any state of the circuit, without forcing an initialization sequence as an antecedent), we can obtain additional simulation coverage for free by continuously inserting happy tokens at the initial vertex. Doing so checks all suffixes of a given trace in a single simulation run. To enable this functionality, our construction gives the user the option of having the initialization input to $v0$ always tied to 1 instead of being dependent on the init signal.

This completes the construction.

The overall construction is obviously linear-size and linear-time in the size of the assertion graph, because each portion of the assertion graph requires a constant amount of work to translate and produces a constant amount of circuitry. Some straightforward optimizations are possible. For example, the acceptance signal is really based on not having any condemned paths at a terminal edge, so the alive(e) and reject(e) signals can be simplified. Also, there is no need to track happy paths if there is also a condemned token on the same edge.

From a conventional simulation or emulation perspective, our method for handling symbolic constants is problematic. The issue is that symbolic constants are intrinsically symbolic, encoding slightly different assertions for **every** possible value. In our construction, the latches for the symbolic constants "guess" the values of the symbolic constants; if the guess is wrong, the assertion graph accepts vacuously and incorrectly. If the symbolic constants are simulated symbolically, however, the simulation is simultaneously guessing all possible values of the constant, giving the correct results. We have chosen this translation because it is simple and compact, and because our simulator is actually a symbolic simulator, so it can correctly simulate all possible values of the symbolic constants at the same time. Note that this is very lightweight symbolic simulation, because it is simply matching the symbolic constants to what occurs in the trace being monitored; all other aspects of the assertion graph are being simulated normally. It is far less expensive than full symbolic simulation or model checking.

If symbolic simulation is not possible (e.g., if the monitor is being compiled into an emulator), then an alternative construction could be to use a three-valued (0, 1, or X) encoding for the symbolic constant latches. The latches would start uninitialized (all Xs), and the first time the symbolic constant is used, the latch would get the specific value written to it. For example, in the assertion graph in Figure 1, when the WRITE happens, the *A* and *D* symbolic constants would record the values of the addr and datawr inputs, and these values would be checked by the later edges. The cost of this alternative is higher hardware complexity and some syntactic restrictions on the assertion graphs — antecedents involving symbolic constants would have to be convertible into assignments to those constants. Symbolic constants are typically used in this manner (intuitively, to remember values for comparison later), so the syntactic restrictions may not matter in practice. A better solution may be to make assignments to symbolic constants explicit; this would also allow creating monitors that are "retriggerable", e.g., when no longer needed to monitor one transaction, the same symbolic constants could be reused to monitor another transaction.

4. EXPERIMENTAL RESULTS

We have implemented the above translation into Intel's Forte verification system[3] and report empirical results measuring the sizes of generated circuits as well as the impact of the monitor circuits on simulation speed. Our system is implemented in an interpreted,

[3]Forte is available for download at http://www.intel.com/software/products/opensource/tools1/verification/ but our new algorithms are not yet part of the standard distribution.

FIFO	Assertion Graph		Monitor Circuit		
Depth	Vertices	Edges	Gates	Latches	Time
2	7	15	1127	68	0.0
4	11	29	2219	124	0.1
8	19	57	4403	236	0.2
16	35	113	8771	460	0.6
32	67	225	17507	908	1.4
64	131	449	34979	1804	3.8
128	259	897	69923	3596	11.9
256	515	1793	139811	7180	42.4
512	1027	3585	279587	14348	164.7
1024	2051	7169	559139	28684	606.1

Table 1: Results for FIFO Example. This example shows time and size behavior of our construction as we scale the number of vertices and edges in an assertion graph. The "Depth" column indicates the depth of the FIFO. (It is 8 bits wide.) "Vertices" and "Edges" indicate the size of the assertion graph. "Gates" and "Latches" indicate the size of the monitor circuit. "Time" is how many seconds it took to generate the monitor circuit. Size is clearly linear; run time is almost linear.

functional programming language, which provides an excellent prototyping environment at some cost in raw performance. Accordingly, run times should be considered as a relative indication of performance, rather than as absolute speed measures. All experiments were run using an Intel Pentium 4 processor running at 2.8 Ghz. Memory size was not a factor in our experiments, despite the small amount of symbolic simulation used to handle symbolic constants.

We have run three types of experiments: two using realistic, but somewhat idealized, assertion graphs that are easily scalable, and a third experiment using an actual industrial circuit. The first two experiments allow us to measure the run time and the size of the generated monitor circuit as a function of different kinds of scaling of the assertion graph. The third experiment lets us measure simulation slow-down due to our generated monitors.

4.1 FIFO

The first example is a scalable family of assertion graphs used to verify a FIFO buffer. The property specified is that the empty and full signals are being set properly, and that the enqueued data is not corrupted and comes out at the right time. In this example, the size of the assertion graph, measured in terms of vertices and edges, is easily scaled for different buffer depths. Indeed, the assertion graph itself is generated via a script. Table 1 shows the results as we scale the assertion graph size for different buffer depths. The size of the generated monitor circuits is clearly growing linearly in the size of the assertion graph. The generation time appears to be growing slightly faster than linearly, probably due to implementation overheads. In any case, both the monitor sizes and the times required to generate them are quite reasonable. Some simple logic optimization would likely further reduce the sizes of the monitors.

4.2 Memory

The next example is an assertion graph for verifying memories. The property being specified is that a read from a given address will return the most recent data value written to that address, similar to the assertion graph shown in Figure 1. In this example, the structure

Address	Assertion Graph		Monitor Circuit		
Size	Vertices	Edges	Gates	Latches	Time
2	3	3	1766	136	0.2
4	3	3	1815	138	0.2
6	3	3	1866	140	0.2
8	3	3	1910	142	0.2
10	3	3	1954	144	0.2
12	3	3	1998	146	0.2

Table 2: Results for Memory Example. This example shows time and size behavior of our construction as we scale the complexity of the edge labels. The "Size" column indicates the width of addresses to the memory. (The data size is 128 bits.) Here, the number of vertices and edges are constant, but the edge labels grow linearly in the address size, so the monitor circuit does, too.

of the assertion graph remains constant as we scale up the number of addresses, but the complexity of the antecedent and consequent labels grows linearly with the size of the addresses, because of operations that compare addresses. Table 2 shows results for this example. The number of latches is clearly growing linearly. The number of gates is also growing roughly linearly, although not perfectly smoothly due to idiosyncrasies of our translation of Boolean expressions. The run time is swamped by constant overhead.

4.3 Industrial Cache Circuit

The last example is a real, industrial circuit. The circuit is a 32K cache with non-trivial read/write logic. (The circuit has 403972 gates and 35157 latches in total.) The property being specified/checked is similar to that for the memory example: a read will return the value of the most recent write to that location. The assertion graph has 3 vertices and 3 edges, the generated monitor circuit has 9363 gates and 46 latches, and generation took 2.7 seconds. The point of this example is to measure the slowdown of our monitor circuit on simulation.

We generated a 25000 cycle random trace. At each cycle, the write probability was 50% and the read probability was 80%. All memory cells were the target of at least one write in the trace, and all reads were to addresses that had previously been written.

Using the simulator built into our verification environment, we could simulate this trace on the cache circuit alone in 1109.6 seconds. Adding the monitor to the simulation resulted in a run time of 1283.1 seconds, or roughly a 16% overhead. Since our simulator is a symbolic simulator, the simulation run with the monitor circuit covered all possible values of the symbolic constants in a single simulation run.

In a production environment, running on a multiprocessor, it should be possible to eliminate the overhead of the monitor circuit entirely. The information flow is one-way from the system being verified to the monitor, so the monitor could be simulated asynchronously on a second processor. As long as the monitor simulation can keep up with the system simulation, there would be no slowdown. Even without further optimizations, however, the relative simulation overhead of our generated monitor is minimal.

5. CONCLUSION AND FUTURE WORK

We have presented a procedure to construct monitor circuits for GSTE assertion graphs. The construction is highly efficient in theory, and experimental results confirm that monitor circuits for real industrial GSTE assertion graphs can be constructed in negli-

gible time and impose minimal simulation overhead. The ability to build monitor circuits from formal assertions has numerous applications for tying formal verification into the overall verification flow.

The most obvious direction for future work is to improve the handling of symbolic constants under simulation, as described earlier. Our current construction, however, does work very well with symbolic simulation. Similarly, efficiency improvements are possible in the sub-circuits generated to check antecedents and consequents.

In this paper, our assumption has been that the motivation for using assertion graphs is to access the efficiency of GSTE model checking. The monitor construction, however, translates an assertion graph into a circuit, which allows using assertion graphs as specifications with other formal verification engines (e.g., symbolic model checking [7], bounded model checking [5], etc.). An intriguing exercise would be to explore for what types of problems each verification engine works best.

Our main direction for future work is to investigate compositional reasoning. Because of the capacity limitations of formal verification tools, being able to compose smaller verification results into larger conclusions is a critical part of a scalable formal verification flow. For example, we may wish to assume one property while trying to verify another (e.g., assume-guarantee reasoning, environment constraints). Because the GSTE model-checking algorithm verifies a relationship between a circuit and a specification, the ability to convert specifications into monitor circuits creates the possibility of using GSTE to verify relationships involving multiple specifications (e.g., assume one assertion graph while verifying another). We have preliminary results along these lines, showing that the construction presented here is a valuable building block toward efficient compositional verification with GSTE [9].

6. REFERENCES

[1] Mark Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Combining theorem proving and trajectory evaluation in an industrial environment. In *35th Design Automation Conference*, pages 538–541. ACM/IEEE, 1998.

[2] Nina Amla, E. Allen Emerson, and Kedar S. Namjoshi. Efficient decompositional model-checking for regular timing diagrams. In *Correct Hardware Design and Verification: 10th IFIP WG 10.5 Advanced Research Working Conference (CHARME'99)*, pages 67–81. Springer, 1999. Lecture Notes in Computer Science Number 1703.

[3] Lionel Bening and Harry Foster. *Principles of Verifiable RTL Design: A Functional Coding Style Supporting Verification Processes in Verilog*. Kluwer Academic Publishers, 2nd edition, 2001.

[4] Bob Bentley. High level validation of next generation microprocessors. In *International Workshop on High-Level Design, Validation, and Test*. IEEE, 2002.

[5] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207. Springer, 1999. Lecture Notes in Computer Science Vol. 1579.

[6] Per Bjesse, Tim Leonard, and Abdel Mokkedem. Finding bugs in an Alpha microprocessor using satisfiability solvers. In *Computer-Aided Verification: 13th International Conference*, pages 454–464. Springer, 2001. Lecture Notes in Computer Science Number 2102.

[7] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Conference on Logic in Computer Science*, pages 428–439, 1990. An extended version of this paper appeared in *Information and Computation*, Vol. 98, No. 2, June 1992.

[8] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Workshop on Logics of Programs*, pages 52–71, May 1981. Published 1982 as Lecture Notes in Computer Science Number 131.

[9] Alan J. Hu, Jeremy Casas, and Jin Yang. Reasoning about GSTE assertion graphs. In *Correct Hardware Design and Verification Methods: 12th IFIP WG 10.5 Advanced Research Working Conference (CHARME'03)*. Springer, 2003. Lecture Notes in Computer Science. To appear.

[10] M. S. Jahanpour and E. Cerny. Compositional verification of an ATM switch module using interface recognizer/suppliers (IRS). In *International High-Level Design, Validation, and Test Workshop*, pages 71–76. IEEE, 2000.

[11] Matt Kaufmann, Andrew Martin, and Carl Pixley. Design constraints in symbolic model checking. In *Computer-Aided Verification: 10th International Conference*, pages 477–487. Springer, 1998. Lecture Notes in Computer Science Number 1427.

[12] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[13] Zohar Manna and Amir Pnueli. Specification and verification of concurrent programs by ∀-automata. In *Symposium on Principles of Programming Languages*, pages 1–12. ACM, 1987.

[14] Kyle L. Nelson, Alok Jain, and Randal E. Bryant. Formal verification of a superscalar execution unit. In *34th Design Automation Conference*, pages 161–166. ACM/IEEE, 1997.

[15] Márcio T. Oliveira and Alan J. Hu. High-level specification and automatic generation of IP interface monitors. In *39th Design Automation Conference*, pages 129–134. ACM/IEEE, 2002.

[16] Manish Pandey, Richard Raimi, Derek L. Beatty, and Randal E. Bryant. Formal verification of PowerPC arrays using symbolic trajecory evaluation. In *33rd Design Automation Conference*, pages 649–654. ACM/IEEE, 1996.

[17] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in Cesar. In *5th International Symposium on Programming*, pages 337–351. Springer, 1981. Lecture Notes in Computer Science Number 137.

[18] Pascal Raymond. Recognizing regular expressions by means of dataflow networks. In *23rd International Colloquium on Automata, Languages, and Programming*, pages 336–347. Springer, 1996. Lecture Notes in Computer Science Number 1099.

[19] Andrew Seawright and Forrest Brewer. High-level symbolic construction techniques for high performance sequential synthesis. In *30th Design Automation Conference*, pages 424–428. ACM/IEEE, 1993.

[20] Carl-Johan H. Seger and Randal E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–190, 1995.

[21] Kanna Shimizu, David L. Dill, and Alan J. Hu. Monitor-based formal specification of PCI. In *Formal Methods in Computer-Aided Design*, pages 335–353. Springer, 2000. Lecture Notes in Computer Science Number 1954.

[22] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science*, pages 332–344. IEEE Computer Society, 1986.

[23] Jin Yang and Amit Goel. GSTE through a case study. In *International Conference on Computer-Aided Design*, pages 534–541. IEEE/ACM, 2002.

[24] Jin Yang and Carl-Johan H. Seger. Introduction to generalized symbolic trajectory evaluation. In *International Conference on Computer Design*, pages 360–365. IEEE, 2001.

[25] Jin Yang and Carl-Johan H. Seger. Generalized symbolic trajectory evaluation — abstraction in action. In *Formal Methods in Computer-Aided Design: Fourth International Conference*, pages 70–87. Springer, 2002. Lecture Notes in Computer Science Number 2517.

[26] Jun Yuan, Kurt Shultz, Carl Pixley, Hillel Miller, and Adnan Aziz. Modeling design constraints and biasing in simulation using BDDs. In *International Conference on Computer-Aided Design*, pages 584–589. IEEE/ACM, 1999.