Generating Monitor Circuits for Simulation-Friendly GSTE Assertion Graphs*

Kelvin Ng
Department of Computer Science
University of British Columbia
kng@cs.ubc.ca

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

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

Abstract

Formal and dynamic (simulation, emulation, etc.) verification techniques are both needed to deal with the overall challenge of verification. Ideally, the same specification/testbench would work with both formal and dynamic techniques, with the same semantics in both. Unfortunately, this is typically not the case. In particular, Generalized Symbolic Trajectory Evaluation *(GSTE) is a powerful formal verification technique developed by Intel and successfully used on next-generation microprocessor designs, but the specification formalism for GSTE relies on "symbolic constants", which intrinsically exploit the underlying formal verification engine and cannot be reasonably handled via non-symbolic means. In this paper, we propose a modified version of GSTE specifications, and we present efficient, automatic constructions to convert from the new simulation-friendly GSTE specifications into conventional GSTE specifications (to access the formal verification tool flow) as well as into completely non-symbolic monitor circuits suitable for conventional dynamic verification. We demonstrate empirically that our simulation-friendly specification style is expressive enough for almost all real GSTE specifications, that our monitor construction is linear-size, and that our monitor construction imposes minimal overhead over a previously published monitor construction that was not fully non-symbolic.*

1. Introduction

Formal verification and dynamic verification (i.e., simulation, emulation, etc.) are both needed to deal with the overall challenge of verification. Formal techniques provide unparalleled coverage, whereas dynamic techniques have superior capacity, ramp-up more quickly, and support more detailed modeling. Ideally, the same specification/testbench would work with both formal and dynamic techniques, with the same semantics in both, allowing a methodology that seamlessly chooses whatever technique is most appropriate at any given point in the verification process. Unfortunately, this is typically not the case: formal specifications often have a declarative aspect that can be difficult to convert to the operational style needed for dynamic verification, and vice-versa.

A particularly convenient bridge between formal and dynamic specifications is the *monitor circuit* or *assertion monitor*. A monitor is simply a small circuit that watches, without interfering, the system being verified and flags whether or not the system is obeying a formal correctness property. Monitor circuits have the declarative style of typical formal specifications, yet are operational and can be used with conventional simulation. Extensive research has demonstrated the value of monitor circuits as the cornerstone of a practical verification methodology [1], as an enabler of hierarchical, compositional verification [6, 12, 5], and as a testbench generator for simulation [17]. Monitor circuits could even be synthesized into an emulation system to aid error observability and debugging.

In this paper, we focus on *Generalized Symbolic Trajectory Evaluation* (GSTE) [15][1]. GSTE was developed by Intel and is emerging as an important formal verification technique that has been successful on leading-edge designs in industry, where users reported superior efficiency and capacity (e.g., [2]), as well as having demonstrated efficiency advantages in academic research [11].

GSTE uses a particular specification formalism, called an *assertion graph*, and the efficiency of GSTE depends, in part, on the specifics of assertion graphs. Assertion graphs, in turn, rely on a concept called "symbolic constants" (described in Section 2), which intrinsically exploit the underlying formal verification engine. Furthermore, when specifications become retriggerable and multi-threaded, the semantics of symbolic constants become even more complex. Previous work building monitor circuits for GSTE assertion

* Supported in part by grants from Intel Corporation and the Natural Science and Engineering Research Council of Canada.

1 GSTE is important in its own right, but we also hope that these ideas can prove helpful for other specification formalisms. In particular, GSTE symbolic constants are used similarly to auxiliary variables in temporal logics, so our ideas may be more broadly applicable.

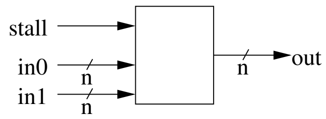

Figure 1. Generic Example Circuit

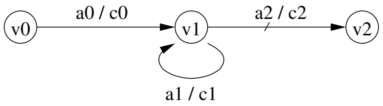

Figure 2. Generic Example Assertion Graph

graphs could not handle symbolic constants, so the resulting monitor "circuit" needed a symbolic simulator to have correct (i.e., agreeing with the formal verification) simulation semantics [4].

In this paper, we address this problem. We first propose a modified version of GSTE assertion graphs with clearer (but somewhat restricted) semantics for symbolic constants. We then present efficient, automatic constructions to convert from the new simulation-friendly GSTE specifications into traditional GSTE specifications (in order to access the existing formal verification tool flow), as well as into completely non-symbolic monitor circuits suitable for conventional dynamic verification. We demonstrate empirically that our simulation-friendly specification style is expressive enough for almost all real GSTE specifications, that our monitor construction is linear-size, and that our monitor construction imposes minimal overhead over the previously published partially-symbolic monitor construction.

2. GSTE Assertion Graphs

GSTE is a model-checking [3, 10] method based on the language-containment paradigm [13, 7]: the specification is given as an automaton, and verification proves that all possible behaviors of the system are accepted by the automaton. For GSTE, the specification automaton is a variant of $\forall$-automata [8] called an assertion graph. GSTE is explained in detail in several sources (e.g., [15, 16, 14], etc.). Here, we give only a brief overview of assertion graphs to make this paper self-contained.

Figure 1 shows a generic example (sequential) circuit with two data inputs, a stall input, and a data output, and Figure 2 shows a generic example assertion graph. An assertion graph has a set of vertices (with an initial vertex v_0), and a set of edges. Each edge is labeled with an *antecedent* a_i and a consequent c_i, which are boolean formulas on the signal names in the circuit (and symbolic constants, explained below). Furthermore, some edges can be labeled as *terminal edges*. Every path starting from the initial vertex and ending on a terminal edge represents a distinct temporal assertion, with each edge corresponding to a clock cycle.[2] For example, the path that goes from v_0 to v_1, loops back to v_1, and then proceeds to v_2 corresponds to the temporal assertion "**If** a_0 holds on the first cycle, **and** a_1 holds on the second cycle, **and** a_2 holds on the third clock cycle, **then** c_0 must hold on the first cycle, **and** c_1 must hold on the second cycle, **and** c_2 must hold on the third cycle." In general, a run of a circuit satisfies a path if either at least one antecedent fails (Intuitively, the assertion is satisfied vacuously, because one of the preconditions isn't met.) or else all antecedents are satisfied, and so are all the consequents. Assertion graphs being $\forall$-automata, a circuit run satisfies an assertion graph if it satisfies **every** path from initial vertex to terminal edge. Intuitively, the assertion graph rolls up an infinite set of temporal assertions into a finite graph.

For example, suppose the circuit in Figure 1 is a stallable adder with one cycle minimum latency. We could verify that $1+1=2$ (with correct stalling behavior) by having a_0 be $(\texttt{in0}=1)\wedge(\texttt{in1}=1)\wedge\neg\texttt{stall}$, having a_1 be the formula $\texttt{stall}$, having a_2 be $\neg\texttt{stall}$, and having c_2 be $(\texttt{out}=2)$. (The consequents c_0 and c_1 are just the formula **true**.) The assertion graph would represent an infinite family of temporal assertions (for each possible length of stall), asserting that if the inputs happen to be 1 and 1, then the output must be 2. (If the inputs happen to be other values, then the assertion graph is vacuously satisfied.)

Symbolic Constants Obviously, we'd like to verify the adder for all possible input values, not just $1+1$. To handle this, GSTE introduces *symbolic constants*, which can take arbitrary values. GSTE model checking verifies the assertion graph for all possible values of the symbolic constants. Continuing the example, we could change a_0 to be $(\texttt{in0}=A)\wedge(\texttt{in1}=B)\wedge\neg\texttt{stall}$, and c_2 to be $(\texttt{out}=A+B)$. Intuitively, we are using A and B to "remember" the values seen on the inputs and check that the output is correct. Formally, the use of symbolic constants is equivalent to having 2^{2n} copies of the original assertion graph, with the different copies "guessing" all possible numeric values for A and B. We call each of these copies, with specific values for all symbolic constants, an *assigned instance* of the assertion graph. For any given circuit execution, only one assigned instance will guess correctly; the others will guess wrong and accept vacuously. Because of this guessing effect, we could equally well have specified the adder with a_0 being, for example, $(\texttt{in0}=A-B)\wedge(\texttt{in1}=B)\wedge\neg\texttt{stall}$, and c_2 being $(\texttt{out}=A)$. Note that symbolic constants intrinsically rely on an underlying formal verification engine — the different assigned instances are encoded symbolically via BDDs, and antecedents and consequents can entail arbi-

2 We consider here only GSTE over finite-length executions, because that is what is used most in practice and corresponds to properties that can be checked using dynamic validation. There is also a theory of GSTE over infinite-length behaviors.

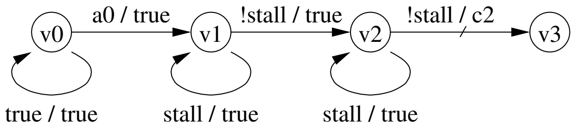

Figure 3. Example Pipelined Assertion Graph

trary constraint solving over the symbolic constants.

Retriggering and Knots The formal semantics of symbolic constants lead to unexpected results when a portion of an assertion graph is reused for a new transaction. For example, suppose our example in Figure 1 is actually part of a much larger assertion graph, and there is a path from v_2 back to v_0 when the system is ready to perform another addition. Intuitively, we'd like A and B to remember the new input values for the new addition problem. However, symbolic constants are fixed for all time. During the first addition, one assigned instance guesses the correct values for A and B, and all the other instances vacuously accept. When the assertion graph returns to v_0 for the next addition, only the one assigned instance is still active, and that instance is fixated on the previous values of A and B. The assertion graph is unable to retrigger and check a new transaction.

To address this problem, GSTE enhancements introduced the concept of *knots* to assertion graphs. Intuitively, a knot is a point in the assertion graph where the value of a symbolic constant is forgotten. The name knot arises because conceptually, the knot is a point where the different assigned instances are "tied together", allowing a path to move from one assigned instance to another, with different values for symbolic constants. If we introduce a knot at v_0, the assertion graph becomes retriggerable.

The intuitive effect of knots can be subtle. Consider the assertion graph in Figure 3, based on our previous example, with a_0 being $(\texttt{in0} = A) \wedge (\texttt{in1} = B) \wedge \neg\texttt{stall}$, and c_2 being $(\texttt{out} = A + B)$. This graph specifies a 2-stage pipelined adder: the self-loop on v_0 indicates a 1-cycle issue rate, and the extra edge from v_1 to v_2 means the result is available with a 2-cycle latency. We need a knot at v_0 so that the symbolic constants can change for different addition transactions. Intuitively, it may not be obvious that the assertion graph can keep the different copies of the symbolic constants distinct, but recall that each assigned instance is effectively a separate copy of the assertion graph, so they do not interfere. Furthermore, it may appear that we could make infinite-state assertion graphs by, for example, changing a_0 to be $(\texttt{in0} = A) \wedge (\texttt{in1} = B)$, so that we can load new values every clock cycle, while delaying the output of results by asserting `stall`. However, the resulting assertion graph is actually finite-state, because the different instances will all get stuck on the `stall` self-loops, where the temporal ordering of the different symbolic constant values will be lost. The graph will record only the set of symbolic constant values that are possible on each edge, which is extremely large, but still finite-state, rather than the sequence of values, which is infinite-state.

3. Simulation-Friendly Assertion Graphs

Three main difficulties appear to prevent creating fully non-symbolic monitor circuits for assertion graphs: first, symbolic constants initially take on all possible values, and then the set of possible values is pruned essentially by constraint solving (where the antecedents are the constraints); second, the semantics of knots are defined specifically according to the formal model — how the exponential number of assigned instances interact; and third, as we saw in Figure 3, assertion graphs can record an intractable amount of history, such as the exact set of data values that have been seen. These difficulties mean that a fully general monitor construction for assertion graphs would need to generate circuits that are exponential in the number of bits of symbolic constant and perform constraint solving. Clearly, such a monitor circuit would not be practical. An additional drawback of fully general assertion graphs is that the semantics can be unintuitive, requiring a good understanding of the underlying formal model. A unified specification style for formal and dynamic verification should be simpler to understand.

Fortunately, the assertion graphs we have observed in practice are not fully general. In particular, the typical usage of symbolic constants is intuitively an assignment statement to the "constant" to record some information, followed by subsequent use of that information. The key to simulation-friendly assertion graphs is to capture this intuitive usage.

A *simulation-friendly assertion graph* is basically the same as a normal assertion graph. However, we introduce explicit assignment statements and eliminate the knot construct. Assignment statements can be placed on edges, and they assign to a symbolic constant some value computed as an arithmetic/logical expression over the signal names in the circuit. The assignment takes effect before the antecedent and consequent on that edge are evaluated. We impose the additional restriction that on every path from the initial vertex, each symbolic constant must be assigned before it is used.

Simulation-friendliness eliminates the need for constraint solving, provides retriggerability automatically without knots, and is intuitively clear. For example, for the pipelined adder in Figure 3, on the edge from v_0 to v_1, we would have assignment statements (assign $A = \texttt{in0}$) and (assign $B = \texttt{in1}$), and would make a_0 be $\neg\texttt{stall}$. Retriggering is obvious from the assignment statements. Each token carries its own copy of the values of A and B, so the pipelined nature of the assertion graph is clear.

The translation of a simulation-friendly assertion graph into an equivalent traditional assertion graph is straightforward. Each assignment statement is converted into a clause in the antecedent, asserting equality between the symbolic constant and the value being "assigned" to it. To get the retriggering effect, we also introduce a partial knot on an edge. (A partial knot erases the value of only the variable being assigned. Formally, it corresponds to connecting only the assigned instances that agree on all symbolic constants except for the one being assigned.)

The remaining questions are: (1) Are simulation-friendly assertion graphs general enough for practical industrial usage? (2) Can we build a fully non-symbolic monitor circuit efficiently? And (3) how can we limit the size of the monitor circuit to avoid recording intractable amounts of history information? We address the first question empirically in Section 5. The next section addresses the other two.

4. Monitor Circuit Construction

Overall, the monitor circuit construction for simulation-friendly assertion graphs is based on the assertion-graph-to-monitor construction proposed in [4], with added complexity to properly handle symbolic constants. To limit the amount of history information the monitor needs to record, we assume the user provides a constant k, the maximum number of assertion graph instances that the monitor can handle at a time. Section 4.6 contains a brief discussion on determining the value of k. The monitor circuit observes the system under verification and determines whether the execution trace is legal according to the assertion graph. It has two output signals, `accept` and `overflow`. The `accept` signal is asserted when the assertion graph accepts the trace. The `overflow` signal is asserted when the monitor determines that it has run out of storage space to monitor system execution, rendering the `accept` output incorrect. The monitor inputs include system signals that appear on antecedents and consequents, and `reset`, which initializes the monitor for a new trace when asserted.

Intuitively, the monitor circuit has an internal copy of the assertion graph and uses tokens to track relevant paths. It starts by placing a token on edges that start from the initial vertex (and clearing tokens on the rest). A token arriving on an edge means that at least one path ends on that edge, on that clock cycle. Tokens also carry history information about their represented paths. An edge receiving a token checks its antecedent/consequent, and forwards a token to its outgoing edge(s) on the next cycle if necessary. Multiple tokens arriving on the same edge at the same time with the same histories (described below) are merged, because those paths share the same future. To determine trace acceptance, the monitor checks the terminal edges for any token that suggests violation of the assertion.

Following the terminology in [4], there are three types of paths: *blessed*, *happy* and *condemned*. A *blessed* path has at least one failed antecedent and therefore accepts the trace vacuously. Furthermore, any extension of a blessed path is also blessed. A *happy* path has all its antecedents and consequents satisfied and accepts the current trace, but may reject its extensions. A *condemned* path has all its antecedents satisfied but at least one consequent failed. A condemned path rejects the current trace, but its extensions may later be blessed.

When any token arrives on an edge and the antecedent fails, the represented path(s) and all its(their) extensions are blessed. The token disappears on the next cycle. When a happy token arrives, an edge forwards a happy token to its outgoing edges if both the antecedent and the consequent hold. If the consequent fails, the edge forwards a condemned token. An edge receiving a condemned token forwards a condemned token if its antecedent holds. The monitor asserts its `accept` signal if and only if no condemned token is generated on any terminal edge.

To evaluate antecedents/consequents that contains symbolic constants, the history of a path should also include the assigned values to symbolic constants. Therefore, tokens remember assigned values, updating them when they visit an edge with an assign statement, which we call an *assigning edge*. Tokens with different assigned values cannot be merged. (If the execution requires more that k different values, `overflow` is asserted.) A key optimization is to clear the assigned value on a token as soon as all future edges the token may visit do not need symbolic constants. It is straightforward to decide which edges should receive tokens with assigned values. We will refer to them as *instance edges*, as opposed to *simple edges*, which do not require assigned values on tokens. *Instance vertices* are starting vertices of instance edges; *simple vertices* are not. Figure 4 presents an algorithm for finding instance edges.

The hardware implementation closely follows the above intuition. The monitor circuit is structured like the assertion graph, with modules for each vertex and edge, connected as in the graph. Tokens are passed around the circuit via a pair of signals, `happy` and `condemned`, indicating the forwarding/arrival of the corresponding type of token. Simple edges need not track multiple assigned instances, so they communicate via a single `happy`/`condemned` pair; instance edges, however, might need to distinguish between up to k instances, so they have k `happy`/`condemned` pairs, that operate in parallel. Each pair is labeled with its *instance id* from 1 to k. The *instance manager* module is responsible for allocating instance ids. A token on an assigning edge activates a request to the instance manager. The instance manager also maintains k banks of latches storing up to k assigned values for each symbolic constant. For each instance id i, the ith bank of latches is connected only to the parts of the cir-

function findInstanceEdges(G)
instanceEdges $\leftarrow \emptyset$
for all symbolic constants C of G **do**
    visited $\leftarrow \emptyset$
    frontier $\leftarrow$ edges that have C in labels
    while *frontier* $\neq \emptyset$ **do**
        $e \leftarrow$ a member of *frontier*
        frontier $\leftarrow$ *frontier* $- \{e\}$
        visited $\leftarrow$ *visited* $\cup \{e\}$
        if e does not assign C **then**
            incoming $\leftarrow$ incoming edges of e that do not assign C
            frontier $\leftarrow$ *frontier* $\cup$ (*incoming* $-$ *visited*)
        end if
    end while
    instanceEdges $\leftarrow$ *instanceEdges* $\cup$ *visited*
end for
return *instanceEdges*

Figure 4. *An algorithm for finding instance edges in assertion graph G*

cuit with instance id i, so all instances can read/write their assigned values simultaneously.

4.1. Vertices

A vertex module is just a bit of combinational logic that forwards tokens from incoming to outgoing edges. For simple vertices, the output `happy` and `condemned` signals are the disjunction (OR) of all the incoming `happy` and `condemned` signals, respectively. Some incoming edges to a simple vertex might be instance edges, but their k token signals can be merged (by disjunction) since the assigned values become irrelevant at this point. Instance vertices, on the other hand, always have instance edges as inputs, so there are k pairs of token signals on each incoming edge, and each output signal is the disjunction of the corresponding input signals with the same instance id.

4.2. Edges

Each edge module contains logic and latches to evaluate the antecedent and consequent, to determine whether (and what) tokens to forward, and to delay tokens one clock cycle. (Details of the basic edge module logic are available in [4].) This is true for both simple and instance edges, but instance edges have everything duplicated k times. Each of the k copies connects to a separate set of assigned values (if needed) and outputs its own pair of token signals.

An assigning edge module invokes the instance manager by passing it the current results ($\texttt{happy}_{\texttt{now}}$ and $\texttt{condemned}_{\texttt{now}}$). The instance manager (Section 4.3) returns the correct values for the k pairs of outgoing token signals. A special case is when the edge assigns a symbolic constant that its own antecedent/consequent uses. Instead of looking up latches for the assigned value, the antecedent/consequent logic should replace references to the symbolic constant with the signal assigned to it. This enables the immediate use of assigned value to evaluate the antecedent/consequent on an assigning edge. Moreover, the monitor avoids unnecessarily storing the assigned value if the antecedent fails on the assigning edge.

4.3. Instance Manager

The instance manager module allocates instance ids to assigning edges, updates assigned values, and determines if overflow has occurred. The main challenge is to arbitrate among all the requests for new instance ids. To simplify arbitration, we impose a fixed, arbitrary priority order on all assigning edges. At each cycle, the instance manager looks at the set of all requests and the set of all available instance ids, and matches them up in priority order. Overflow occurs when there is any unacknowledged request.

We give the formal definition of the instance manager logic below. Subscripts denote instance ids. We will use the notation $\forall e' < e$ to denote the set of all edges with higher priority than an edge e. Intuitively, the signal $\texttt{inUse}_i$ indicates whether instance id i is being used by any token, and for any assigning edge e, the signal $\texttt{ack}(e)_j$ or $\texttt{ack}(e)_{i,j}$ indicates that the request (from instance id i) has been granted an instance id of j.

$$\texttt{inUse}_i = \bigvee_{\text{all instance vertices } v} (\texttt{happy}(v)_i \vee \texttt{condemned}(v)_i)$$

Each assigning edge e has signals that interact with the instance manager. To save space, we show the signal names implicitly assuming that they are local to e, except in cases where there are formulas that refer to signals from multiple edges. For each assigning simple edge (*ASE*) e,

$$\begin{aligned}
\texttt{active} &= \texttt{happy_now} \vee \texttt{condemned_now} \\
\texttt{ack}(e)_j &= \texttt{active} \wedge \neg\texttt{inUse}_j \\
&\quad \wedge \bigwedge_{g<j} \neg\texttt{ack}(e)_g \wedge \bigwedge_{\forall e'<e} \neg\texttt{ack}(e')_j \\
\texttt{happy_next}_j &= \texttt{happy_now} \wedge \texttt{ack}(e)_j \\
\texttt{happy_out}_j &= \mathrm{DFF}(\texttt{happy_next}_j) \\
\texttt{condemned_next}_j &= \texttt{condemned_now} \wedge \texttt{ack}(e)_j \\
\texttt{condemned_out}_j &= \mathrm{DFF}(\texttt{condemned_next}_j) \\
\texttt{overflow}(e) &= \texttt{active} \wedge \bigwedge_{g \in 1..k} \neg\texttt{ack}(e)_g
\end{aligned}$$

For each assigning instance edge (*AIE*) e,

$$\texttt{active}_i = \texttt{happy\_now}_i \vee \texttt{condemned\_now}_i$$

$$
\begin{aligned}
\texttt{ack}(e)_{i,j} &= \texttt{active}_i \wedge \neg\texttt{inUse}_j \\
&\wedge \bigwedge_{g<j} \neg\texttt{ack}(e)_{i,g} \\
&\wedge \bigwedge_{\forall e'<e} \bigwedge_{h\in 1..k} \neg\texttt{ack}(e')_{h,j} \\
&\wedge \bigwedge_{\text{all ASE } a} \neg\texttt{ack}(a)_j \\
\texttt{ack}(e)_j &= \bigvee_{i\in 1..k} \texttt{ack}(e)_{i,j} \\
\texttt{happy\_next}_j &= \bigvee_{i\in 1..k} (\texttt{happy\_now}_i \wedge \texttt{ack}(e)_{i,j}) \\
\texttt{happy\_out}_j &= \mathrm{DFF}(\texttt{happy\_next}_j) \\
\texttt{condemned\_next}_j &= \bigvee_{i\in 1..k} (\texttt{condemned\_now}_i \wedge \texttt{ack}(e)_{i,j}) \\
\texttt{condemned\_out}_j &= \mathrm{DFF}(\texttt{condemned\_next}_j) \\
\texttt{overflow}(e) &= \bigvee_{i\in 1..k} (\texttt{active}_i \wedge \bigwedge_{j\in 1..k} \neg\texttt{ack}(e)_{i,j})
\end{aligned}
$$

The instance manager controls the storage of assigned values via the write-enable and input signals to the latches. Intuitively, write is enabled if any edge assigning to that instance of a constant has requested and been acknowledged. The data input is basically a multiplexor that selects the new assigned value or existing value based on which edges were acknowledged; the only complication is when an instance i assigns some of its constants and gets allocated new instance id j, the new instance must copy over the values of the other constants from instance i. Formally, for a given bit position of a given symbolic constant, let $l_1,\ldots,l_k$ denote the k latches for storing this bit of the constant. We partition the set of assigning edges into three sets: E_a, those that assign to this constant; E_b, those instance edges that do not assign to this constant; and, E_c, those simple edges that do not assign to this constant. For $e \in E_a$, let $s(e)$ denote the value that e wants to assign to the symbolic constant. The signals `we`, `Din`, and `Dout` denote the write-enable, input, and the output signals of a latch.

$$
\begin{aligned}
\texttt{we}(l_j) &= \bigvee_{e\in E_a\cup E_b} \texttt{ack}(e)_j \\
\texttt{Din}(l_j) &= \bigvee_{e\in E_a} (\texttt{ack}(e)_j \wedge s(e)) \\
&\vee \bigvee_{e\in E_b} \bigvee_{i\in 1..k} (\texttt{ack}(e)_{i,j} \wedge \texttt{Dout}(l_i))
\end{aligned}
$$

4.4. Monitor Output

The `accept` signal is asserted when any terminal simple edge (TSE) or terminal instance edge (TIE) has generated a condemned token. The `overflow` signal is asserted when any assigning edge asserts its overflow.

$$
\begin{aligned}
\texttt{accept} &= \bigwedge_{\text{all TSE } e} \neg\texttt{condemned\_now}(e) \\
&\wedge \bigwedge_{\text{all TIE } e} \bigwedge_{i\in 1..k} \neg\texttt{condemned\_now}(e)_i \\
\texttt{overflow} &= \bigvee_{\text{all assigning edge } e} \texttt{overflow}(e)
\end{aligned}
$$

4.5. Special Case: $k = 1$

It is possible to significantly reduce the size of the monitor circuit when $k = 1$. As there is only one place to store each constant, the instance manager becomes redundant except for the overflow logic. If the user is certain that overflow is impossible (Section 4.6), our implementation allows the user the option to build the monitor circuit without the instance manager and related signals when $k = 1$.

4.6. Bounding k

A natural question is what value of k should the user supply. We have observed that it is often easy to determine an upper bound on the k that a given assertion graph requires.

A common special case is when, on all outgoing edges from each vertex, the antecedents are mutually exclusive. In this case, the number of instances required is $k = 1$. For example, the unpipelined adder example in Section 2 obeys this constraint and only needs $k = 1$, whereas the pipelined adder example does not obey this constraint and requires $k > 1$. As noted in the preceding subsection, this is a very desirable special case because the monitor circuit can be built with no overhead for instance management.

More generally, note that edges with assignment statements request a new instance id every time they are active; we call these the *requesting edges*. If there is only one requesting edge in the assertion graph, the number of instances required is the same as the maximum number of times that that edge receives a new token before a previously assigned token is released. For example, returning to the pipelined adder in Section 2, Figure 3, the edge from v_0 to v_1 is the requesting edge. If we set antecedent a_0 correctly as $\neg$`stall`, then the requesting edge can only receive three tokens (that aren't immediately blessed) during the lifetime of any token, so $k = 3$. On the other hand, if we set the antecedent a_0 to be **true**, then an unbounded number of new tokens can pass through the requesting edge while another token is stuck in a loop, so our analysis conservatively determines that k is unbounded. If there are multiple requesting edges, the same analysis can be performed for them individually, and the edges can be partitioned into groups where the lifetimes of the assigned tokens by group members may overlap. The sum of required number of instances of all group members is taken. The overall number of instances required for the assertion graph is bounded by the largest sum amongst all the requesting edge groups.

5. Experimental Results

Experiments have been conducted to test our assumptions about simulation-friendly assertion graphs and monitors. We implemented our monitor construction method in FL, an interpreted, functional language used by Intel's FORTE verification system.

5.1. Simulation Friendliness in Real Life

We have inspected 18 GSTE assertion graphs used in real, industrial verification to study the practicality of our simulation-friendly assertion graphs. The 18 specifications cover various units in a microprocessor design, ranging from memory to datapath to control intensive circuits and from the frontend to the backend of the microarchitecture flow. Each of the specifications describes a non-trivial functionality of a circuit. A majority of them cover the entire circuit from inputs to outputs. The sizes of the circuits range from ∼500 latches and ∼12k gates all the way to ∼45k latches and ∼240k gates. All the specifications have been verified using GSTE without any prior model abstraction/pruning.

Out of the 18 GSTE specifications, 15 are immediately convertible into our simulation-friendly assertion graph format. The test for convertibility was automatic using a short program. The remaining 3 specifications include the use of symbolic constants for symbolic indexing [9], which is not directly simulation-friendly. However, all 3 specifications were still convertible with some manual effort. Specifically, typical usage of symbolic indexing is for case-splitting and for exploiting symmetry. For case-splitting, the number of bits of symbolic index is small, so a symbolically-indexed antecedent can be made simulation friendly by duplicating the edge and enumerating the cases. For symmetry, if there is an array of n presumed-symmetric storage locations (e.g., bits in a word, words in a memory, etc.), the symbolically-indexed assertion graph uses $\log_2 n$ bits of symbolic constant to index one of the n locations. To convert to simulation-friendliness, either we can generate n instances, one for each possible value of the symbolic index, exactly as we handled case splitting, or we can give up on symmetry and verify all n locations in a single instance. For example, if we are verifying a 64-bit wide memory, we might have an antecedent like $(\mathtt{din}[63:0] = D[63:0])$ without symbolic indexing, or $(\mathtt{din}[K[5:0]] = D)$ with symbolic indexing, where D and K are symbolic constants. In the former case, the antecedent is obviously simulation-friendly. In the latter case, we can make 64 copies of this edge for each possible value of K, where the ith copy of the edge will have antecedent $(K[5:0] = i) \wedge (D = \mathtt{din}[i])$, which is simulation-friendly. In general, it may be possible to automate these conversions for common usage idioms. Overall, these results confirm our belief that simulation-friendly assertion graphs are expressive enough for useful real-life applications.

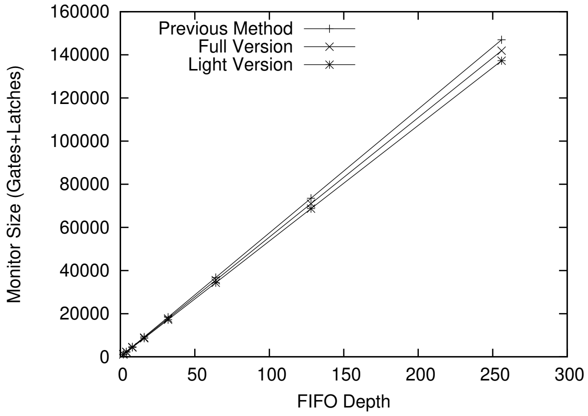

Figure 5. Monitor Size vs. Previous Construction for FIFO Example (from [4]). *The FIFO example specifies correct operation of a FIFO. The assertion graph size scales linearly with FIFO depth.*

5.2. Comparison with Previous Construction

The proposed monitor construction removes the need for symbolic simulation, which arose for assertion graphs with symbolic constants in the previous monitor construction [4]. That construction relied on the user to guess a single value (or explore all possible values using symbolic simulation) for each constant during initialization, which it stored and used to monitor the whole trace. This is analogous to the $k = 1$ case in our translation and provides a point of comparison. We have run experiments with three different monitor constructions: *previous*, *light*, and *full* using the same assertion graph examples from the previous paper [4]. The light version differs from the full and the normal version in that it does not have an overflow signal (See Section 4.5). As expected, the newly constructed monitors behave similarly to the previous one; they scale linearly with both the assertion graph size and the antecedent/consequent size. Figures 5 and 6 show a plot of the results. The new construction sometimes produced smaller circuits, even with the overhead of the instance manager module. The reason is that there is a significant reduction in size after removing logic for antecedents that become assignment statements.

5.3. Effect of Changing the Parameter k

It is obvious from our translation that the monitor size should scale linearly with the value of k. For the sub-circuits that depend on k, which include any sub-circuit that needs

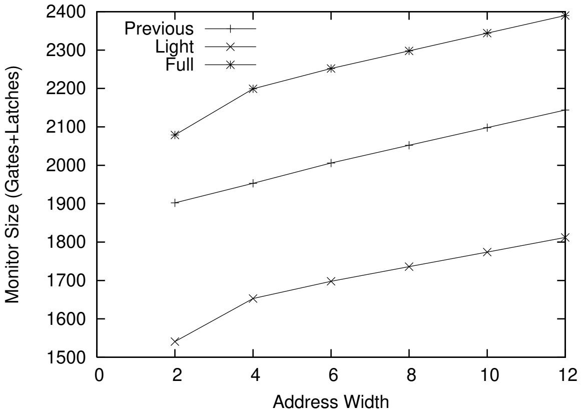

Figure 6. Monitor Size vs. Previous Construction for Memory Example (from [4]). *In this example, the specification is for the correct operation of a memory. The complexity of the antecedents and consequents scales linearly with address width.*

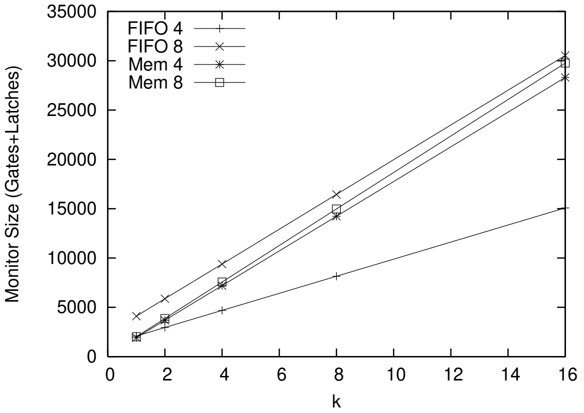

Figure 7. Monitor Scaling with k**.** *The graph shows the monitor size for different example instances as we scale k. FIFO n denotes an n stage FIFO, and Mem n denotes a memory with n-bit addresses.*

an instance id, the size always scales linearly with k. Experimental results confirmed this relationship between k and the generated monitor size. Run time also scales linearly with k. Figure 7 shows the scaling of monitor size with k for some of the various assertion graph examples run.

6. Conclusions

We have introduced a novel, simulation-friendly style of GSTE assertion graph and have presented a method to construct a fully non-symbolic monitor circuit for such assertion graphs with symbolic constants. Combined with our straightforward translation from simulation-friendly assertion graphs into conventional assertion graphs, our work allows using the same specification with both formal GSTE model checking as well as dynamic verification. Our empirical results show that simulation-friendly assertion graphs are expressive enough for real, industrial usage, and that the monitor circuit generation is efficient, scaling linearly with assertion graph size and the number of instances needed. This work is an important step towards seamlessly integrating formal and dynamic verification.

References

[1] L. Bening and H. Foster. *Principles of Verifiable RTL Design: A Functional Coding Style Supporting Verification Processes in Verilog*. Kluwer, 2nd ed., 2001.

[2] B. Bentley. High level validation of next generation microprocessors. In *Int'l Workshop on High-Level Design, Validation, and Test*. IEEE, 2002.

[3] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Workshop on Logics of Programs*, pages 52–71, May 1981. Published 1982 as LNCS Vol 131.

[4] A. J. Hu, J. Casas, and J. Yang. Efficient generation of monitor circuits for GSTE assertion graphs. In *Int'l Conf. on Computer-Aided Design*, pages 154–159. IEEE/ACM, 2003.

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

[6] M. Kaufmann, A. Martin, and C. Pixley. Design constraints in symbolic model checking. In *Computer-Aided Verification: 10th Int'l Conf.*, pages 477–487. Springer, 1998. LNCS Vol 1427.

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

[8] Z. Manna and A. Pnueli. Specification and verification of concurrent programs by ∀-automata. In *Symp. on Principles of Programming Languages*, pages 1–12. ACM, 1987.

[9] T. F. Melham and R. B. Jones. Abstraction by symbolic indexing transformations. In *Formal Methods in Computer-Aided Design: 4th Int'l Conf.*, pages 1–18. Springer, 2002. LNCS Vol 2517.

[10] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *5th Int'l Symp. on Programming*, pages 337–351. Springer, 1981. LNCS Vol 137.

[11] R. Sebastiani, E. Singerman, S. Tonetta, and M. Y. Vardi. GSTE is partitioned model checking. In *Computer-Aided Verification: 16th Int'l Conf.*. Springer-Verlag, 2004. To appear.

[12] K. Shimizu, D. L. Dill, and A. J. Hu. Monitor-based formal specification of PCI. In *Formal Methods in Computer-Aided Design*, pages 335–353. Springer, 2000. LNCS Vol 1954.

[13] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Symp. on Logic in Computer Science*, pages 332–344. IEEE, 1986.

[14] J. Yang and A. Goel. GSTE through a case study. In *Int'l Conf. on Computer-Aided Design*, pages 534–541. IEEE/ACM, 2002.

[15] J. Yang and C.-J. H. Seger. Introduction to generalized symbolic trajectory evaluation. In *Int'l Conf. on Computer Design*, pages 360–365. IEEE, 2001.

[16] J. Yang and C.-J. H. Seger. Generalized symbolic trajectory evaluation — abstraction in action. In *Formal Methods in Computer-Aided Design: 4th Int'l Conf.*, pages 70–87. Springer, 2002. LNCS Vol 2517.

[17] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz. Modeling design constraints and biasing in simulation using BDDs. In *Int'l Conf. on Computer-Aided Design*, pages 584–589. IEEE/ACM, 1999.