GSTE Through A Case Study

Jin Yang
Strategic CAD Labs, Intel Corp.
jin.yang@intel.com

Amit Goel
ECE, Carnegie-Mellon University
agoel@ece.cmu.edu

Abstract

Generalized Symbolic Trajectory Evaluation (GSTE) [17, 18, 19] is a very significant extension of STE that has the power to verify all ω-regular properties but at the same time preserves the benefits of the original STE [16]. It also extends the symbolic quaternary model used by STE to support seamless model refinement for efficiency and accuracy trade-off in GSTE model checking. In this paper, we present a case study on FIFO verification to illustrate the strength of GSTE and demonstrate its methodology in specifying and verifying large scale designs.

1 Introduction

STE is a model checking technique based on a form of quaternary symbolic simulation [16, 9]. It has shown great promise in verifying medium to large scale industrial hardware designs, with a high degree of automation and at both the gate level and the transistor level. STE has been in active use at Intel, Compaq, IBM, and Motorola. In Motorola, it has been used to verify several memory units, some with millions of transistors [14, 15, 13]. In Intel, it has been used, for instance, to verify a floating point arithmetic unit against the IEEE standard 754 [12], and to verify a complex IA-32 instruction length decoder unit with 12,000 gates and 1100 latches [2].

Despite its efficiency, STE is very limited in the kinds of properties it can handle. Any property spreading over an indefinitely long time interval cannot be expressed in STE, let alone be verified. Further, any property requiring backward reasoning is beyond the reach of STE as the nature of simulation only allows constraints to be propagated forward.

In recent years, several efforts have been made to extend the expressiveness of STE. The first such work was proposed by Seger and Bryant in [16], where they introduced non-nested loops into STE assertions. In [3], Beatty proposed a more general extension of STE assertions by introducing a form of labeled transition graphs where each vertex is associated with an antecedent and a consequent. In [10], Jain developed a generalized STE algorithm to model check this form of generalized STE assertions, which was mathematically clarified by Chou in [6]. This work, however, has two major limitations. First, it does not support backward reasoning, making it very difficult, if not impossible, to use assumptions in the future or at the output of a circuit. Second, it is built on the same symbolic quaternary simulation mechanism as in STE, making it impossible to support arbitrary boolean constraints and model refinement.

The work in [17, 18, 19] significantly advanced previous state-of-the-art. The contributions are twofold. First, STE was extended with backward simulation and further generalized to verify all ω-regular properties. Second, the symbolic quaternary model was extended to work with sets of quaternary assignments. Such an enhancement provides a great deal of flexibility in choosing the right abstraction level for a circuit in GSTE to achieve model-checking efficiency while avoiding excessive over-approximation.

We do not intend to cover all the aspects of GSTE in this paper. Instead, we illustrate the essence and usage of GSTE through a FIFO verification case study. In Section 2, we provide a brief introduction to GSTE and the symbolic quaternary model. In Section 3, we give motivations for selecting the FIFO verification as our case study and describe what we want to accomplish. In Section 4, we discuss previous similar work in [4, 11], and point out the advantages of GSTE in supporting implementation-independent specifications and making verification more efficient and more productive. In Section 5, we develop a high level FIFO specification using an *assertion graph*, the specification language for GSTE, and show how to refine it to meet various design requirements. In Section 6, we demonstrate GSTE model checking through several examples. In particular, we discuss an extension to the symbolic quaternary model to handle symbolic sets of quaternary assignments, and show it enables model refinement in GSTE model checking. In Section 7, we present the experimental results on verifying several FIFO implementations with different sizes. The results show only a near linear increase in memory consumption with respect to the depth of a FIFO. We also show one data point from real life FIFO verification. All verifications were done on the original unpruned gate-level RTL models. In Section 8, we briefly discuss the relation of GSTE to classic symbolic model checking. We conclude the paper in Section 9.

2 GSTE Overview

We first introduce a simple circuit model. A *circuit* consists of a set of boolean nodes N, which is partitioned into two sets: *state nodes* N_S and *input nodes* N_I. A *state* is an assignment to all the nodes in N. There is a *next state function* $\chi_n(N)$ for each node $n \in N_S$. The set of next state functions defines how the circuit

0-7803-7607-2/02/$17.00

transitions between states. The transition can also be defined by the equivalent *transition relation* $R(N,N') = \bigwedge_{n \in N_S}(n' = \chi_n(N))$, where N' is a copy of N to hold the values for N after the transition and $n' \in N'$ is the copy of n. A *state trace* is a state sequence such that every pair of consecutive states in the sequence satisfies the transition relation.

In GSTE, a specification is expressed as an assertion graph. An *assertion graph* is a quintuple $G = (V, v_0, E, ant, cons)$, where V is a set of *vertices* with v_0 being the *initial vertex*, E is a set of *directed edges*, *ant* and *cons* are functions that map each edge to a state predicate called *antecedent* and a state predicate called *consequent*, respectively. A circuit *satisfies* the graph, if for **every** path in the graph from the initial vertex and every state trace in the circuit of the same length, if the trace satisfies the antecedent sequence on the path, then it also satisfies the consequent sequence on the path.

The expressiveness of assertion graphs can be further enriched with terminal conditions and fairness conditions that limit the kinds of paths to be examined. In the former case, only paths that end at a terminal vertex are checked. In the latter case, only fair infinite paths are checked, and it can be proven that such an extension has the expressive power of the ω-regular languages. Further, the algorithm described below has been generalized to model check the two extensions. For an in-depth treatment of GSTE, we refer the reader to [17, 18, 19].

To model check an assertion graph against a circuit, we first compute a *simulation relation* that maps each edge e in the graph to a set of circuit states $sim(e)$. A state s is in $sim(e)$, if some trace leading to s satisfies the antecedent sequence on some path from the initial vertex to the edge e. Once the computation is done, we check the consequent on each edge e to see if it is satisfied by every state in $sim(e)$. Figure 1 lists the entire GSTE algorithm, where *post* is the *post-image* function of a state set $S(N)$ given by $post((S(N)) = \exists N.S(N) \wedge R(N,N')$, i.e., the set of states reachable from a state in $S(N)$ in a single transition. It can be proven that the algorithm returns true if and only if the circuit satisfies the assertion graph.

Although the algorithm in Figure 1 can be implemented using a BDD-based symbolic technique, building transition relations and working on precise sets of circuit states are very sensitive to the state explosion problem. GSTE significantly alleviates the problem by using the symbolic circuit simulation technique in STE [5] to compute the sets of next states, and doing so on the quaternary abstraction of the circuit, where each node in the circuit has the four values $\{0,1,X,\top\}$. X denotes an *unknown* and $\top$ an *over-constraint*. Besides the quaternary generalization of the boolean operations, two new operations are defined: the least upper bound $\sqcup$ and the greatest lower bound $\sqcap$ of any two quaternary values. It should be pointed out that we are using a slightly different partial order than was used in the original STE theory [16]. We place X at the top and $\top$ at the bottom. The reason for this is that we can then interpret X as the set $\{0,1\}$ and $\top$ as $\emptyset$. As a result, the least upper bound $\sqcup$ corresponds to the set union and the greatest lower bound $\sqcap$ corresponds to the set intersection. Figure 2 lists the truth tables for the basic quaternary operations.

```
Algorithm: GSTE(G,post)
(* initialize simulation relation computation *)
1. for each edge e in the assertion graph G
2.     if e is from the initial vertex
3.         sim(e) := ant(e);
4.         put e into the event queue;
5.     else
6.         sim(e) := ∅;
(* perform simulation relation computation *)
7. while the event queue is not empty
8.     get an edge e from the queue;
9.     for each successor edge e' of e
10.        sim(e') := sim(e') ∪ post(sim(e)) ∩ ant(e');
11.        if there is a change in sim(e')
12.            put e' into the event queue;
(* check consequents *)
13. for each edge e in the graph
14.    if sim(e) ⊈ cons(e)
15.        return(false);
16. return(true);
end.
```

Figure 1. GSTE Algorithm

&	X	0	1	T
X	X	0	X	T
0	0	0	0	T
1	X	0	1	T
T	T	T	T	T

|	X	0	1	T
X	X	X	1	T
0	X	0	1	T
1	1	1	1	T
T	T	T	T	T

!	
X	X
0	1
1	0
T	T

⊓	X	0	1	T
X	X	0	1	T
0	0	0	T	T
1	1	T	1	T
T	T	T	T	T

⊔	X	0	1	T
X	X	X	X	X
0	X	0	X	0
1	X	X	1	1
T	X	0	1	T

Figure 2. Quaternary Operations

Any state set in the circuit can be represented either precisely or approximately by a quaternary assignment to the nodes in the circuit. A node has a boolean value in the quaternary assignment if it has the same boolean value in every state of the set. Otherwise, it has value X. The empty set is represented by assigning $\top$ to one or more nodes depending on where the conflict occurs in the circuit. With this abstraction, the state space becomes much smaller, and all operations become much more efficient. For instance, the intersection $\cap$ (union $\cup$) of two state sets becomes a bit-wise $\sqcap$ ($\sqcup$) of the two corresponding quaternary assignments. The post-image function becomes the bit-wise quaternary generalization of the next state functions together with X for the input nodes.

It was shown in [6] that the quaternary model of a circuit is essentially an *abstract interpretation* of the boolean model via a *Galois connection* [8].

The quaternary model can be made symbolic by allowing a quaternary assignment to take boolean functions over symbolic constants [5, 9] as its values. All the quaternary operations can be easily extended to handle such symbolic quaternary assignments. To be model checked in this quaternary model, an assertion graph must express its antecedents and consequents using symbolic quaternary assignments. However, such a symbolic assignment can only capture a very limited set of boolean con-

straints among signals along an arbitrary path in the graph, since a symbolic constant would take the same value whenever it is mentioned on the path. In Section 6, we shall propose an extension to the symbolic quaternary model that will completely eliminate this limitation.

3 FIFO Case Study

A FIFO is a very common data structure in microprocessor design. Despite being a simple and well-understood subject in textbooks, the design and implementation of a FIFO in a microprocessor can be very complex and diversified. At minimum, a FIFO can be implemented as either a circular stationary structure or a marching buffer with a fixed back-end. Sometimes the implementation is embedded in a memory. In many situations, the data entries in a FIFO are split into many parts, either logically or physically, and are mingled with other parts of the microprocessor. Each part may be implemented differently. As an example, a wide datapath element may be split so that only its identifier goes through the FIFO, while the rest is stored in a content addressable memory (CAM) and is picked up and assembled when the element comes out of the FIFO.

Additionally, a FIFO may need to satisfy different design requirements. For instance, data elements may have variable lengths. Some computation could happen to a data element while it is in a FIFO, either changing some of its attributes or adding new attributes. Data may arrive and depart a FIFO at different rates and under different conditions. A FIFO may contain multiple slots to support parallelism. Timing is yet another complication factor.

Given these facts, FIFO verification makes a very good case study to demonstrate the strength of GSTE and develop its practical methodology. In particular, we would like to achieve three goals in this case study: (1) to develop a high level GSTE specification for the standard FIFO which can then be refined to meet different design requirements, (2) to make sure that a refined specification is as implementation-independent as possible, i.e., describing what an implementation should do rather than how it does it, and (3) to make sure that GSTE is capable of efficiently model-checking such a specification.

We consider several FIFO implementations for this case study. We start with the two basic FIFO implementations. In the *marching implementation*, the element at the head of the FIFO is read out during every dequeue operation and all other elements in the FIFO shift one entry toward the head (Figure 3 (1)). The *stationary implementation* is a circular structure with head and tail pointers. On a dequeue operation, an element is read from the entry pointed to by the head pointer which is then incremented (Figure 3 (2)). In both implementations, an element is enqueued to the entry pointed to by the tail pointer which is then updated.

Using these two implementations, we show how a high level FIFO specification can be developed independent of any implementation and yet can still be efficiently model checked against by making non-data-path state elements precise. We discuss a key technique called *extended symbolic quaternary model* that

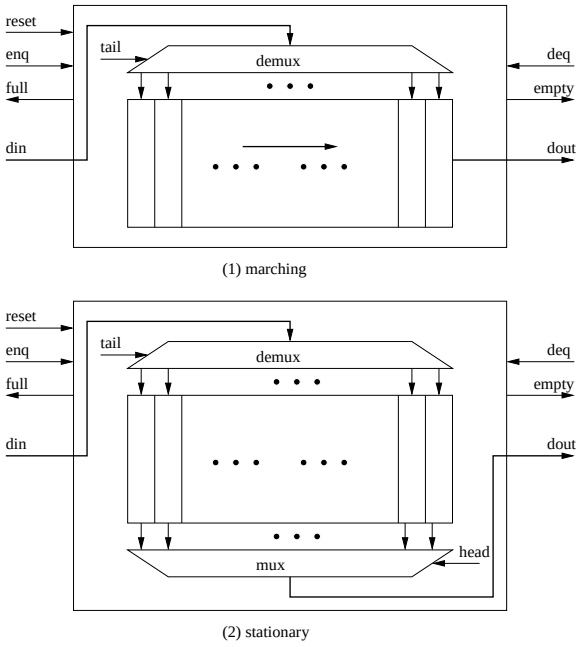

Figure 3. Two Implementations of FIFO

overcomes the coarseness of the original symbolic quaternary model and enables seamless model refinement.

We further consider a more complex variation of the FIFO where every entry has two slots (Figure 4). The double-slot FIFO is derived from a real industrial design. There are two rows of data in the FIFO and each enqueue operation involves two data inputs, *din0* and *din1*. Each input can be directed to either row, depending on the row selection signals *row0* and *row1*. It is possible that both are written to the same row, thereby creating *bubbles* in the other row. If both inputs are written to the same row, *din0* should be written ahead of *din1*. When we try to dequeue from a non-empty FIFO, we may get either one or two data outputs depending on the presence or absence of a bubble. Signal *valid0* is low if a bubble is read from the top row, and *valid1* is low if a bubble is read from the bottom row.

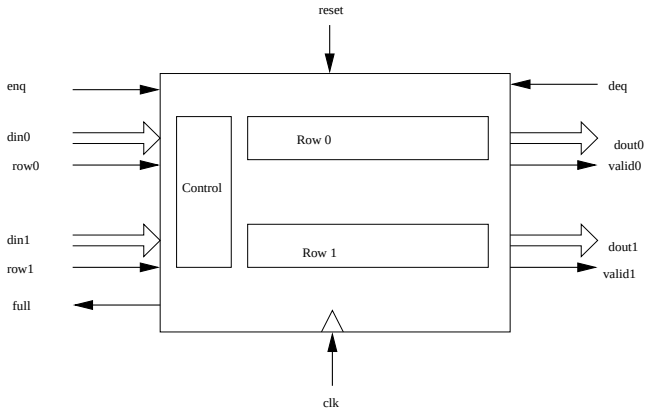

Figure 4. Double-Slot FIFO

Using this more complex design, we demonstrate how the

high level specification can be refined to reflect the precise requirement of a double-slot FIFO. To model check the refined specification against any implementation, the same simple model refinement heuristics applies, i.e., making non-datapath state elements precise. We conduct several experiments on verifying each of the specifications against both stationary and marching implementations, and analyze the time and BDD complexities in model checking with respect to the depth of the FIFO.

Finally, we consider a complex real-life FIFO design with 7506 latches and 17367 gates, from which the simplified double-slot FIFO was derived. The interface behavior of the FIFO is much more complex. For instance, up to four sets of data can come into the FIFO simultaneously, but at most two sets can go into any of the two rows. There may be an additional bubble generated in the FIFO due to a tricky hand-shake protocol between the FIFO and the downstream unit, and so forth. We briefly discuss how our work was extended by the verification expert on the design and show the verification result obtained from the expert.

4 Previous STE Work on FIFO Verification

In [4], Beatty, Bryant and Seger presented a transistor-level FIFO verification case study using STE. Due to the limitation of STE, the high level specification for the FIFO was given in terms of single state transitions. For the marching implementation, these transitions include, for instance, that if the FIFO has k entries, then after a distinct data is enqueued without a dequeue operation, the FIFO will have $k+1$ entries and the $k+1$-th entry will contain the distinct data, and if the data is in the k-th entry ($k > 0$), then it will move to the $k-1$ entry after a dequeue operation. In order to verify a transistor-level implementation against this specification, states in the specification must be mapped to concrete states in the implementation. This is a very painstaking and tedious task that is strongly implementation-dependent and requires a very good understanding of how the implementation works internally.

Although Jain's work on generalizing STE overcomes the expressiveness limitation of STE, it still cannot escape the impreciseness of the symbolic quaternary model. Consequently, a high-level specification has to be augmented with an often fairly complex implementation mapping to form a complete specification for model checking. This mapping is essentially a set of interacting state machines that describe how key state elements in the implementation interact with each other. Therefore, this approach would suffer the same drawbacks as the original STE approach. Furthermore, the manually constructed mapping can be complex and error-prone.

As we shall show in the next few sections, GSTE overcomes these problems by specifying and verifying the behavior of a FIFO at its IO boundaries, and thus drastically simplifies the complexity of the verification task. Without the need for implementation mapping, the same high level specification can be verified against different implementations. Finally, the black-box approach is insensitive to the internals of an implementation, which makes regression much easier.

5 FIFO Specification

The correct behavior of a FIFO can be summarized as follows: (1) the full and empty flags of the FIFO must be set correctly, and (2) elements enqueued into the FIFO must be dequeued in the right order uncorrupted. Developing an assertion graph to express the correct behavior of a FIFO is a relatively straightforward task. First, we keep track of the numbers of entries in the FIFO and how they get updated.

1. After reset, the FIFO has 0 entry.
2. The number of entries increments by 1 after an enqueue only operation, if the number is less than the depth. It decrements by 1 after an dequeue only operation, if it is not 0. Otherwise, it remains unchanged.

Furthermore, the empty flag is set if and only if the number of entries is 0, and the full flag is set if and only if it is the depth of the FIFO.

This part of specification for a 3-deep FIFO is captured by the assertion graph in Figure 5. The initial vertex of the graph is *init*, corresponding to an arbitrary state before reset. The other vertices are the counting states keeping track of the numbers of entries in the FIFO. The edges in the graph correspond to the transitions between states. Each edge is labeled with an antecedent (i.e. what is assumed when the transition happens) and a consequent (i.e. what needs to be proven), separated by a "/".

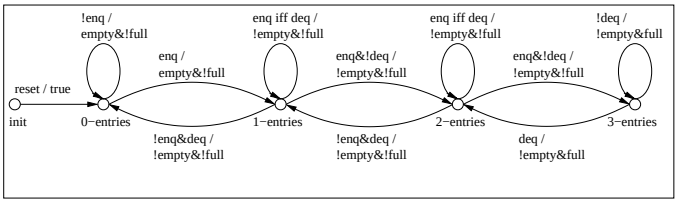

Figure 5. Counting Entries for A 3-Deep FIFO

Obviously, specifying an assertion graph directly for a very deep FIFO would be very tedious and error-prone. Instead, in the Intel FORTE environment ([1]), the assertion graph is generated by calling, with depth $k = 3$, the following parameterized code written in the functional language *fl*. This code follows pretty much the text description of the FIFO specification.

```
let FIFO_counting_spec k =
  let reset_state =
    (state init, //* vertex/state *//
      [ //* edges/transitions from the vertex *//
       ( transition "reset" to (state (0-entries))
         antececent (reset)
         consequent (true)
       )
      ]
    )
   in
  let counting_state i =
    (state (i-entries), //* vertex/state *//
      if (i = 0) then //* 0-entries *//
```

```
    [ //* edges/transitions from the vertex *//
     ( transition "incr" to (state (1-entries))
       antecedent (enq)
       consequent (empty&!full)
     ),
     ( transition "stay" to (state (0-entries))
       antecedent (!enq)
       consequent (empty&!full)
     )
    ]
  else if (i = k) then //* full entries *//
    [ //* edges/transitions from the vertex *//
     ( transition "decr" to (state ((k-1)-entries))
       antecedent (deq)
       consequent (!empty&full)
     ),
     ( transition "stay" to (state (k-entries))
       antecedent (!deq)
       consequent (!empty&full)
     )
    ]
  else // non-empty and non-full entries *//
    [ //* edges/transitions from the vertex *//
     ( transition "incr" to (state ((i+1)-entries))
       antecedent (enq&!deq)
       consequent (!empty&!full)
     ),
     ( transition "decr" to (state ((i-1)-entries))
       antecedent (!enq&deq)
       consequent (!empty&!full)
     ),
     ( transition "stay" to (state (i-entries))
       antecedent (enq=deq)
       consequent (!empty&!full)
     )
    ]
 )
 in
let init_vertex =
 (state init)
 in
let edges =
 reset_state @ (for i from 0 to k (counting_state i))
 in
AG init_vertex edges ;
```

This assertion graph, however, does not specify that an enqueued data element will come out uncorrupted at the right time. To model this, we enqueue a vector of distinct symbolic constants at an arbitrary time, and then keep track of the position of the data in the FIFO. The position decrements by 1 upon each dequeue operation. When the position is the first in the FIFO, the next dequeue operation should produce the data. In addition, transitions need to be added from each counting state to appropriate position states to enqueue the distinct data. Figure 6 shows the assertion graph augmented from Figure 5 to express the complete specification of a 3-deep FIFO. The vertices at the bottom correspond to the position states, and the downward edges correspond to the distinct enqueue operations.

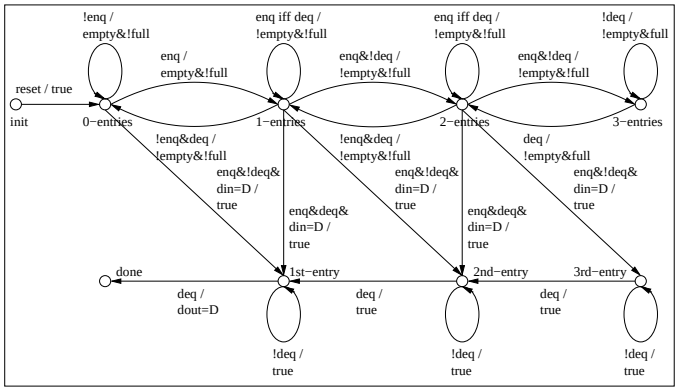

Figure 6. The Complete Specification for A 3-Deep FIFO

This assertion graph can be easily refined to specify the correct behavior of each of the two rows in a double-slot FIFO. The number of entries in the row could now be incremented by either 1 or 2 by an enqueue operation, depending on whether the row selection signals *row1* and *row2* are the same or different. Figure 7 shows the refined assertion graph for row 0 of the double-slot FIFO. For clarity, the consequents except for the output are omitted. The black dots indicate where the antecedents are. Note that if two elements are enqueued to the same row when the FIFO only has one available entry left, then the second element is dropped from the FIFO and needs to be resent later. The graph for row 1 can be obtained symmetrically.

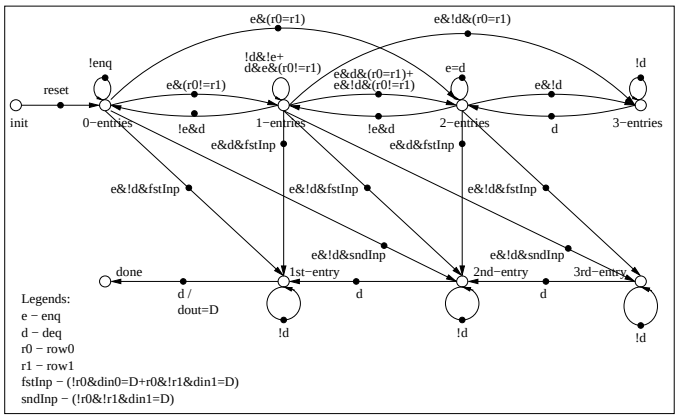

Figure 7. Specification for A 3-Deep Double-Slot FIFO

For the real-life FIFO design, the assertion graph for the double-slot FIFO was further refined to reflect the more complex IO interactions without mentioning any internal behavior. For instance, to reflect the additional bubble in the FIFO, the edge $(\mathit{1st\text{-}entry}, \mathit{done})$ was split into two consecutive ones, and would say that if the dequeue operation on the first edge does not produce the distinct data, then the dequeue operation on the second edge must do.

6 FIFO Verification

Now let us apply the GSTE algorithm in Figure 1 to verify the standard FIFO specification against the two different implementations. Figure 8 shows the quaternary simulation result on the 3-deep marching FIFO implementation from the algorithm in Figure 1. On every edge, we only list state elements with boolean values. The others have value X. The entries in the FIFO are numbered 0 to 2 from right to left. The tail pointer maintains a scalar value after reset until the self loop at vertex *2nd-entry*. Since the tail pointer may have 10 or 11 on the self-loop depending on if there is an enqueue operation, the lower bit becomes X. The situation becomes worse on the self loop at vertex *1st-entry*, as both bits become X. Consequently, the content in the first entry become X as D could potentially be erased by an enqueue operation, and GSTE would eventually fail.

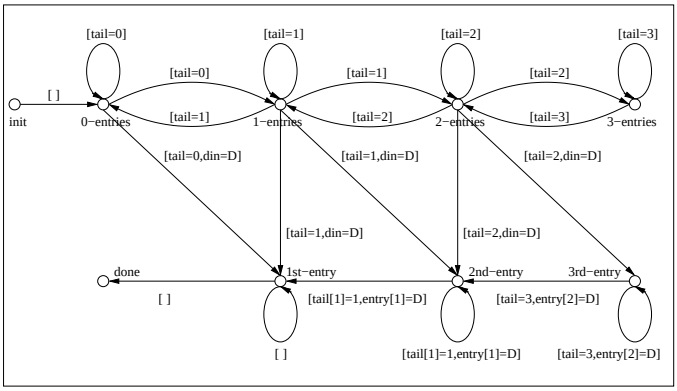

Figure 8. Quaternary Simulation for the Marching FIFO

The problem of losing precision is caused by the coarseness of the STE symbolic quaternary model, as many needed constraints among nodes cannot be captured precisely and are thus lost. In the marching implementation, for instance, the constraint "the tail pointer cannot be 0" is lost. This problem is much more severe for the stationary implementation. Even for an edge in the top half of the graph, the head and tail pointers will quickly become X during simulation as they go through all possible values as long as a constant difference is maintained between the two pointers.

To overcome this problem, GSTE is extended to work on symbolic sets of quaternary assignments. In this approach, a set of *precise* nodes can be specified for model checking. The values and the relations among these nodes are kept precise in any quaternary assignment. For instance, consider the following set of three states for three nodes p, q, r:

$$\{[p=0,q=1,r=1],[p=1,q=0,r=0],[p=1,q=0,r=1]\}.$$

In the quaternary model, this set is abstracted as a single assignment $[p=X,q=X,r=X]$. Useful constraints such as $p \neq q$ and $q \rightarrow r$ are lost. However, if p and q are precise nodes, then the set is abstracted as a set of two quaternary assignments:

$$\{[p=0,q=1,r=1],[p=1,q=0,r=X]\},$$

and the useful constraints are preserved. Using a parametric variable v_p, This abstract set can be represented as the symbolic quaternary assignment

$$[p=v_p,q=\neg v_p,r=(\neg v_p?1{:}X)],$$

which generates the set by assigning all possible values to v_p.

In [20] we presented a unique parameterization algorithm that builds a canonical symbolic quaternary assignment from any symbolic quaternary assignment for a given set of quaternary assignments. This is needed for the termination of the fix-point computation in the GSTE algorithm. In the following, we shall use the marching implementation example to show how this extension improves the simulation result. Let us specify the two bits of the tail pointer as *precise*, and assume the simulation relation for the self-loop at *1st-entry* has been partially computed:

$$sim_0 = [tail[1]=0,tail[0]=1,entry[1]=D].$$

To update the simulation relation through the self-loop (Step 10 in Figure 1), we first compute the next set (i.e., post-image) of states simulated by the edge:

$$\begin{aligned} & nextsim \\ = \;& post(sim_0) \cap ant(1st\text{-}entry, 1st\text{-}entry) \\ = \;& post([tail[1]=0,tail[0]=1,entry[1]=D]) \sqcap [enq=z,deq=0] \\ = \;& [tail[1]=z,tail[0]=\neg z,entry[1]=D]. \end{aligned}$$

Note a free variable z is assigned to *enq* in the antecedent in order to increase precision. In the subsequent $\sqcup$ operation, a special variable *$c* is temporally introduced to represent the union result in order to preserve the precision:

$$\begin{aligned} & newsim \\ = \;& sim_0 \cup nextsim \\ = \;& c?[tail[1]=z,tail[0]=\neg z,entry[1]=D] \\ & \quad :[tail[1]=0,tail[0]=1,entry[1]=D] \\ = \;& [tail[1]=c\&z,tail[0]=!c+!z,entry[1]=D]. \end{aligned}$$

At the end, the new simulation result is canonicalized using the unique parameterization algorithm:

$$[tail[1]=v_{tail[1]},tail[0]=\neg v_{tail[1]},entry[1]=D].$$

Figure 9 shows the improved simulation result by assigning each edge with a symbolic representation of a set of quaternary assignments, where shorter variable names v_1 and v_0 are used in place of $v_{tail[1]}$ and $v_{tail[0]}$ for clarity.

For the stationary implementation, every bit in both the head and tail pointers must be precise in order for GSTE to pass the verification. In addition, the wrap bit that indicates the FIFO is full must also be made precise. In general, a good heuristics to start with is to make every non-data-path element precise. The verification of the double-slot FIFO uses exactly the same heuristics for model refinement. With the extended symbolic quaternary model, the specification can focus on what an implementation should do rather than how it does it.

7 Experimental Results

The GSTE model checker is implemented using the functional language *fl* on top of STE in the Intel FORTE environment. The Figure 10 lists the experimental results on verifying

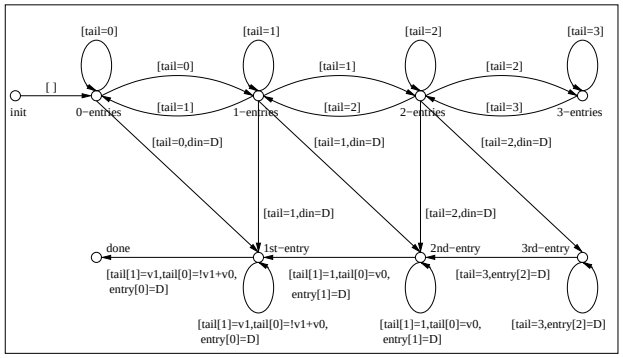

Figure 9. Improved Simulation for the Marching FIFO

the standard high level FIFO specification against both stationary and marching implementations of width 10 with various depths. Figure 11 lists the experiment results on verifying the double-slot FIFO specification against the two implementations. All the experiments were done on a computer with 1.5 GHz Intel® Pentium® 4 with 1 GB memory.

Circuit depth	Marching				Stationary			
	#lats	#gates	time sec.	mem MB	#lats	#gates	time sec.	mem MB
3	42	314	0.2	17.3	45	231	1.2	17.4
7	83	799	1.0	17.5	87	424	7.7	19.8
15	164	1936	3.6	18.8	169	799	60.5	28.0
31	325	4734	12.9	21.8	331	1834	508.4	35.7
63	646	11931	52.2	27.7	653	4825	4227.3	63.7

Figure 10. GSTE Results for Single-Slot FIFOs

Circuit depth	Marching				Stationary			
	#lats	#gates	time sec.	mem MB	#lats	#gates	time sec.	mem MB
3	68	889	0.5	17.5	92	1051	1.2	17.7
7	157	2231	2.5	19.6	182	2249	10.2	23.9
15	334	5040	9.6	23.0	360	5227	95.4	31.6
31	687	11098	39.7	31.4	714	12658	853.2	48.8
63	1392	23396	168.5	45.5	1420	33635	7849.8	110.8

Figure 11. GSTE Results for Double-Slot FIFOs

From Figure 12, we can see that for all four implementations, the memory usage, which includes a 12MB overhead for loading the GSTE system, grows almost linearly with respect to the depth of a FIFO. The time complexity (Figure 13) also grows quite nicely, though not linearly. The complexity for verifying a stationary implementation is much higher and grows faster than that for verifying a marching implementation, since in the stationary implementation, the state space is determined by both the head pointer and the tail pointer and thus is much larger. It is worth mentioning that the complexity does not depend very much on the width of a FIFO. Furthermore, the majority of the time in model checking was spent in GSTE written in *fl* interacting with the core STE engine. BDD complexity was never an issue! Variable re-ordering was never invoked.

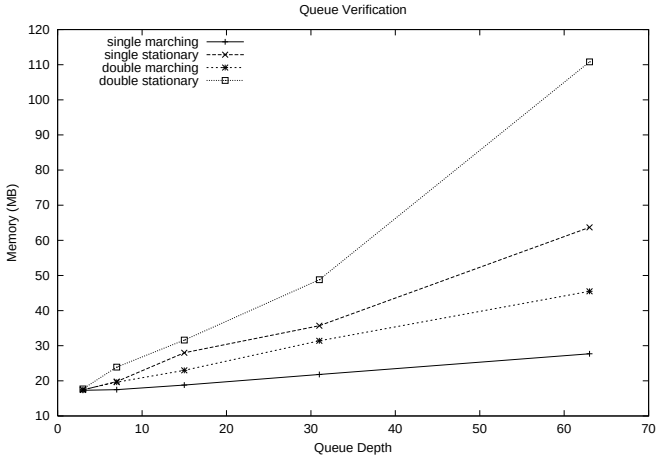

Figure 12. Verification Memory Usage

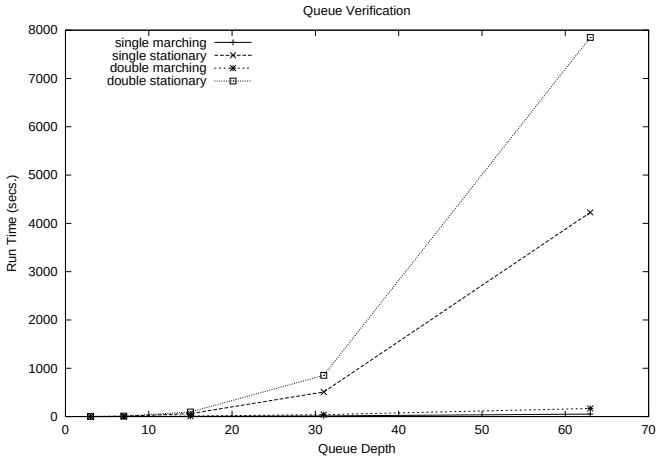

Figure 13. Verification Runtime

Finally, we report the verification result of the complex real-life FIFO design with 7506 latches and 17367 gates, obtained from the verification expert on the design. The assertion graph uses 55 symbolic constants and 40 symbolic variables. 41 state elements are made precise for the model refinement. The model checking took 5220 seconds and 260 MB memory.

We would like to emphasize that all the verifications were done without any prior abstraction or pruning on the original gate-level RTL models.

8 Relation to Classic Symbolic Model Checking

Classic symbolic model checking (SMC) has been studied extensively for over a decade and has been applied to many application areas (see [7] for a good survey). GSTE extends its expressive power by using the same fundamental formal techniques, such as state transition diagrams for specification, pre-image computation and fix-point computation, well developed for SMC. However, GSTE often promises orders of magnitudes higher capacity than SMC for linear temporal logic in hardware

verification, because it benefits from preserving efficient and high capacity techniques used in STE.

From the specification point of view, an assertion graph clearly distinguishes between antecedents and consequents, and ties them tightly together with the underlying structure representing some kind of computation flow. Further, it universally quantifies over paths in the graph. It is not only a more natural way for describing hardware properties, but also a key enabler for GSTE-style efficient model checking. Automata-based specifications, on the other hand, do not make this distinction. The majority of them existentially quantifies over paths. Even though $\forall$-automata universally quantify over paths, they have a completely different acceptance criterion in terms of recurrent and/or stable automata states.

From the verification point of view, an assertion graph drives symbolic circuit simulation based on its underlying structure and antecedents, and thus confines its search space to what is useful for proving the consequents. SMC, on the other hand, works on the product machine of the specification automaton (or its complement) and the circuit model. It builds some form of the transition relation for the product machines, and then performs a global reachability analysis. The latter approach is generally more computationally expensive and easier to cause state explosion.

Perhaps what is more significant in achieving high capacity in GSTE is that the circuit abstraction is an integrated part of model checking and the level of the abstraction depends heavily on the assertion graph. In fact, the complexity of GSTE model checking is primarily determined by the number of variables in the assertion graph. For SMC, however, abstraction techniques are used mostly as an external pre-processing step. Such an abstraction step is often done manually and relies on a good understanding of nicely structured circuit model. However, this is usually very difficult, if not impossible, to do on flattened, customized gate-level RTL models.

9 Conclusion

In this paper, we described the techniques in GSTE to verify a much richer set of properties while preserving the benefits of STE, and to support seamless model refinement. We illustrated the essence of GSTE and demonstrated its strength through a FIFO case study. To the best of our knowledge, this is the first scalable black-box solution for formally specifying and model checking FIFO structures.

Acknowledgment
We would like to thank Brian Moore, Carl Seger, Rajnish Ghughal, Andreas Tiemeyer and Alan Hu for reading the paper and providing many useful suggestions.

References

[1] M. Aagaard, R. Jones, T. Melham, J. O'Leary, and C.-J. Seger. A methodology for large-scale hardware verification. In *FMCAD'2000*, November 2000.

[2] M. Aagaard, R. Jones, and C.-J. Seger. Combining theorem proving and trajectory evaluation in an industrial environment. In *Proc. of the 35th DAC*, pages 538–541, June 1998.

[3] D. Beatty and R. Bryant. Formally verifying a microprocessor using symbolic simulation methodology. In *Proc. of the 31st DAC*, June 1994.

[4] D. Beatty, R. Bryant, and C.-J. Seger. Synchronous circuit verification by symbolic simulation: An illustration. In *Proc. of the 6th MIT Conference on Advanced Research in VLSI*, pages 98–112, 1990.

[5] R. Bryant and C.-J. Seger. Formal verification of digital circuits using symbolic ternary system models. In *DIMAC Workshop on Computer-Aided Verification*, June 1990.

[6] C.-T. Chou. The mathematical foundation of symbolic trajectory evaluation. In *CAV'1999*, July 1999.

[7] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[9] S. Hazelhurst and C.-J. Seger. A simple theorem prover based on symbolic trajectory evaluation and OBDDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(4):413–422, April 1995.

[10] A. Jain. *Formal Hardware Verification by Symbolic Trajectory Evaluation*. PhD thesis, ECE, Carnegie-Mellon University, August 1997.

[11] K. Nelson, A. Jain, and R. Bryant. Formal verification of a superscalar execution unit. In *Proc. of the 34th DAC*, June 1997.

[12] J. O'Leary, X. Zhao, R. Gerth, and C.-J. Seger. Formally verifying ieee compliance of floating-point hardware. *Intel Technology Journal*, Q1:147–190, 1999.

[13] M. Pandey and R. Bryant. Exploiting symmetry when verifying transistor-level circuits by symbolic trajectory evaluation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(7):918–935, July 1999.

[14] M. Pandey, R. Raimi, D. Beatty, and R. Bryant. Formal verification of PowerPC(TM) arrays using symbolic trajectory evaluation. In *Proc. of the 33rd DAC*, June 1996.

[15] M. Pandey, R. Raimi, D. Beatty, R. Bryant, and M. Abadir. Formal verification of content addressable memories using symbolic trajectory evaluation. In *Proc. of the 34th DAC*, June 1997.

[16] C.-J. Seger and R. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–190, March 1995.

[17] J. Yang. Generalized symbolic trajectory evaluation. *STE Symposium, A Satellite Workshop of CAV'2000*, 2000.

[18] J. Yang and C.-J. Seger. Generalized symbolic trajectory evaluation. *Intel SCL Technical Report (submitted for journal publication)*, 2000.

[19] J. Yang and C.-J. Seger. Introduction to generalized symbolic trajectory evaluation. In *Proc. of ICCD-2001*, pages 360–365, September 2001.

[20] J. Yang and C.-J. Seger. Generalized symbolic trajectory evaluation - abstraction in action. In *LNCS: Proc. of FMCAD2002*, November 2002.