Automatic Abstraction Refinement for Generalized Symbolic Trajectory Evaluation

Yan Chen*, Yujing He*, Fei Xie* and Jin Yang†
*Department of Computer Science, Portland State University, Portland, OR 97207. {chenyan, hey, xie}@cs.pdx.edu
†Validation Research Lab, Intel Corporation, Hillsboro, OR 97124. jin.yang@intel.com

Abstract—**In this paper, we present AutoGSTE, a comprehensive approach to automatic abstraction refinement for generalized symbolic trajectory evaluation (GSTE). This approach addresses imprecision of GSTE's quaternary abstraction caused by under-constrained input circuit nodes, quaternary state set unions, and existentially quantified-out symbolic variables. It follows the counterexample-guided abstraction refinement framework and features an algorithm that analyzes counterexamples (symbolic error traces) generated by GSTE to identify causes of imprecision and two complementary algorithms that automate model refinement and specification refinement according to the causes identified. AutoGSTE completely eliminates false negatives due to imprecision of quaternary abstraction. Application of AutoGSTE to benchmark circuits from small to large size has demonstrated that it can quickly converge to an abstraction upon which GSTE can either verify or falsify an assertion graph efficiently.**

I. Introduction

Symbolic trajectory evaluation (STE) [1]–[3] is a powerful model checking technique based on quaternary symbolic simulation. Within its application domain, STE is often much easier to use and less sensitive to state space explosions, compared to classic symbolic model checking (SMC) techniques [4]–[6]. Despite its efficiency, capacity, and ease-to-use, STE is limited in property expressiveness: properties over infinitely long time intervals cannot be specified and verified using STE.

Generalized symbolic trajectory evaluation (GSTE) [7], [8] represents a significant extension to STE. GSTE supports verification of properties over infinitely long time intervals, namely ω-regular properties, thus making it as expressive as classic SMC for linear time logics. Meanwhile, GSTE maintains the efficiency, capacity, and ease-to-use of STE, in particular, GSTE inherits STE's automatic abstraction techniques.

The key to the high capacity of STE and GSTE is the abstraction based on a quaternary state representation (a.k.a., quaternary abstraction) which, however, is also their weakness. The imprecision of quaternary abstraction may lead to false negatives: STE and GSTE may report the result of X (unknown) instead of 0 or 1. This is worsen by the fixed-point computation of GSTE which introduces additional possibilities for unknowns to creep into verification. As a result, there are three possible causes of abstraction imprecision in GSTE:

1. Under-constrained input circuit nodes.
2. Quaternary state set unions.
3. Existentially quantified-out symbolic variables.

Currently in GSTE, imprecision due to Cause 1 is addressed by manually introducing symbolic constants or variables to constrain input nodes. Imprecision due to Causes 2 and 3 is eliminated manually by (1) model refinement: introducing boolean variables (a.k.a., precise nodes) to represent the state space more accurately or (2) specification refinement: applying semantics-preserving transformations to the assertion graph [9]. These manual refinements are often quite involved and require in-depth knowledge of the circuit being verified. Furthermore, the level of abstraction determines the verification complexity of GSTE: the more detailed the abstraction, the higher the verification complexity. Therefore, automatic abstraction refinement algorithms that can quickly converge to an appropriate level of abstraction are highly desired.

In this paper, we present AutoGSTE, a comprehensive approach to automatic abstraction refinement for GSTE, which addresses abstraction imprecision due to under-constrained input nodes, quaternary state set unions, and quantified-out symbolic variables. It follows the counterexample-guided abstraction refinement framework and features an algorithm that analyzes the counterexample (symbolic error trace) generated by GSTE to identify causes of imprecision and two complementary algorithms that automate model refinement and specification refinement according to the causes identified.

- The analysis algorithm identifies these causes by backtracking through the counterexample and conducting fan-in analysis over the circuit.
- If imprecision is due to under-constrained input nodes, symbolic constants are introduced on non-loop edges of the assertion graph while symbolic variables are introduced on loop edges of the assertion graph.
- Using model refinement, the circuit nodes which obtain the unknown values due to imprecision caused by quaternary state set unions or quantified-out symbolic variables are identified and marked as precise nodes in the circuit.
- Using specification refinement, according to the causes identified, the appropriate kind of semantic-preserving transformation is applied to the assertion graph: for imprecision due to quaternary state set unions, loop-unrolling is applied and for imprecision due to quantified-out symbolic variables, case-splitting is applied.

We have implemented AutoGSTE in the Intel *Forte* environment [10] and upon GSTE, and applied it to benchmark circuits from small to large size. The experiments demonstrate that AutoGSTE can quickly converge to an abstraction upon which GSTE can verify or falsify an assertion graph efficiently.

Related Work. There has been much research on abstraction refinement for model checking of both hardware and software. Space limitation precludes a detailed discussion. Counterexample-guided abstraction refinement is a well-known methodology in model checking to combat the state space explosion problem [4], [11], [12]. Particularly relevant to our research is the work on abstraction refinement for STE. In [13], symbolic constants are automatically introduced to constrain under-constrained input nodes of circuits based on counterexamples from STE. In [14], a SAT-based algorithm was developed to assist manual refinement of STE assertions. To our best knowledge, our approach is the first automatic abstraction refinement framework for GSTE that completely eliminates false negatives caused by abstraction imprecision.

The rest of this paper is organized as follows. In Section II, we introduces the basics of STE and GSTE and the quaternary abstraction that they employ. In Section III, we discuss in detail the potential causes for imprecision of the quaternary abstraction. In Section IV, we present AutoGSTE, our approach to automatic abstraction refinement. In Section V, we evaluate our algorithms on small to large size benchmark circuits. In Section VI, we conclude this paper and touch on future work.

II. Background

In this section, we introduce a simple circuit model upon which STE and GSTE were developed, the basics of STE and GSTE, and their quaternary abstraction. For more details on STE and GSTE, we refer the readers to [1], [2], [7], [8].

A. A Simple Circuit Model

A *circuit* M consists of a set of boolean nodes N. A *state* is an assignment to all the nodes in N. The node set N can be partitioned into two disjoint sets: *state nodes* N_S and *input nodes* N_I. There is a *next-state function* $\chi_n(N)$ for each state node $n \in N_S$. The set of next state functions defines how the circuit 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 *trace* of the circuit is a state sequence $\sigma = [s_0, s_1, s_2, \ldots]$ such that for every index i, $s_{i+1}(n) = \chi_n(s_i(N))$ for every state node $n \in N_S$ and $s_{i+1}(n)$ is unconstrained for every input node in $n \in N_I$.

B. Symbolic Trajectory Evaluation

An STE assertion can be viewed as a labeled linear graph representing a finite time line. Each edge in the graph represents a time unit and is labeled with two sets of circuit states (or equivalently state predicates), one of which is called the *antecedent label* and the other the *consequent label*.

The symbolic simulation algorithm in STE starts from the first edge $(v0, v1)$ in the graph and computes the set of states $\mathit{sim}(v0, v1)$ *simulated* by the edge. In this case, it is the set of states satisfying the antecedent label on the edge. The algorithm then performs a single simulation step by computing the *post-image*, $\mathit{post}(\mathit{sim}(v0, v1))$, of this simulation state set and intersects the result with the set of states satisfying the antecedent label on the second edge. The result is the set of states $\mathit{sim}(v1, v2)$ simulated by the second edge. The post-image of a state set $S(N)$, denoted by $\mathit{post}((S(N))$, is given by $\exists N^-.S(N^-) \wedge R(N, N^-)$, i.e., the set of states reachable from a state in $S(N)$ in a single state transition. The algorithm repeats this step until it reaches the last edge in the graph. Once the simulation is complete, the consequent label on each edge is checked against the set of states simulated by this edge to see if it is satisfied, i.e., the set of states simulated by the edge is a subset of states allowed by the consequent predicate.

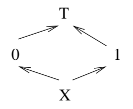
C. Generalized Symbolic Trajectory Evaluation

A *GSTE assertion graph* is defined as a quintuple $G = (V, v_0, E, \mathit{ant}, \mathit{cons})$, where V is a set of *vertices*, v_0 is the *initial vertex*, E is a set of *directed edges*, and *ant* and *cons* are functions that map each edge to an antecedent label and a consequent label, respectively. Every finite path in the GSTE assertion graph from the initial vertex is an STE assertion graph. Therefore, a circuit M *satisfies* a GSTE assertion graph G if it satisfies every STE assertion graph derived from the GSTE assertion graph by following a finite path from the initial vertex in the graph. More formally, the circuit satisfies the assertion graph, denoted by $M \models G$, if for every boolean assignment to all the symbolic constants, every finite path in the graph, and every finite state trace in the circuit of the same length, the trace satisfies the antecedent sequence on the path implies it also satisfies the consequent sequence on the path.

To model check a GSTE assertion graph against a circuit, a symbolic simulation is performed to compute a set of states simulated by each edge in the graph. A state is in this set if there is a finite path from the initial vertex leading to the edge and a finite trace leading to the state such that the trace satisfies the sequence of the antecedent labels along the path. Obviously, for an edge coming out of the initial vertex, any state satisfying the antecedent label on the edge is simulated by the edge. Further, for an edge e and a successor edge e' of e, if s is a state simulated by e, then any next state s' of s that satisfies the antecedent label on e' is simulated by e'. This leads to a GSTE model checking algorithm with an iterative symbolic simulation phase. Since an assertion graph may contain loops and an edge may be reached from the initial vertex through different paths, the algorithm requires a form of least fixed-point computation in the simulation phase [7].

D. Quaternary Abstraction

When the circuit becomes large, the likelihood for a symbolic model checking algorithm to encounter the *state explosion* problem increases drastically. To overcome the problem, some sort of abstraction must be applied to the circuit. In STE, it is the quaternary modeling of the circuit where each node has the four quaternary values $\{0, 1, X, \top\}$ instead of the two boolean values. There is a partial information order among the four quaternary values as shown in Figure 1 (a). For instance, X contains less information than either 0 or 1. Both 0 and 1 contain less information than $\top$, which represents the over-constrained value. Besides the quaternary generalization of the boolean operations, two new operations are defined: the greatest lower bound $\sqcap$ and the least upper bound $\sqcup$ of any two quaternary values. Figure 1 (b) lists the truth tables for the basic quaternary operations.



(a) quaternary values and information ordering

&	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

$\sqcup$	X	0	1	T
X	X	0	1	T
0	0	0	T	T
1	1	T	1	T
T	T	T	T	T

$\sqcap$	X	0	1	T
X	X	X	X	X
0	X	0	X	0
1	X	X	1	1
T	X	0	1	T

(b) quaternary operations

Fig. 1. Quaternary Operations

Given such a quaternary abstraction, 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 state set. Otherwise, it has value X. The empty set is represented by assigning $\top$ to one or more variables depending on where the conflict occurs in the circuit. For instance, consider a circuit with three nodes p, q and r. The quaternary assignment for the singleton state set $\{[p=1, q=0, r=1]\}$ is $[p=1, q=0, r=1]$, and the assignment for the state set $\{[p=1, q=0, r=1], [p=1, q=1, r=1]\}$ is $[p=1, q=X, r=1]$.

With this abstraction, the state space becomes much smaller. In general, for a circuit with n boolean nodes, there are 2^n different states and thus 2^{2^n} different sets of states. On the other hand, there are only 4^n different quaternary assignments. Furthermore, all the operations become much more efficient in the quaternary abstraction. For instance, the intersection $\cap$ of two state sets becomes a bit-wise $\sqcup$ of the two corresponding quaternary assignments, and the union $\cup$ becomes the bit-wise $\sqcap$. The post-image function becomes the bit-wise quaternary generalization of the next state functions for the state nodes together with X for the input nodes. Instead of mapping a state set to another state set, it maps the quaternary assignment representing the first state set to the one representing the second state set.

A problem with the scalar quaternary model, though, is that it is often too coarse. This could easily lead to false negative verification results, since many types of constraints among nodes are lost in the abstraction. To overcome this problem, STE uses a technique called *symbolic indexing* to encode a set of quaternary assignments. For instance, for the circuit with three nodes p, q and r, consider the following set of boolean assignments

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

It captures two basic boolean constraints: (1) p and q are mutual exclusive, and (2) q is true implies r is true. These two constraints are completely lost in the single quaternary assignment corresponding to the set $[p = X, q = X, r = X]$. However, using a symbolic constant C, the two constraints can be precisely encoded in the following symbolic indexing expression

$$(!C \rightarrow [p=0, q=1, r=1]) \wedge (C \rightarrow [p=1, q=0, r=X]),$$

which means that when C is 0 then the first quaternary is chosen, and when C is 1 then the second is chosen. For simplicity, we omit X's in the quaternary assignments below.

When verifying an STE assertion against a circuit, any boolean constraint in the STE assertion can be expressed as a symbolic quaternary assignment using symbolic constants in order to drive the symbolic quaternary simulation in STE. Furthermore, the abstract symbolic quaternary simulation of the circuit can be made into the precise symbolic boolean simulation by assigning enough symbolic constants to the circuit nodes in the antecedent of the assertion. However, the same is no longer true in GSTE for assertion graphs with loops. To address this problem, GSTE allows introducing symbolic variables into symbolic quaternary assignments. Unlike a symbolic constant, a *symbolic variable* is a boolean variable that can change its value, and is existentially quantified out after every single step simulation. One way to look at the difference between symbolic variables and symbolic constants is that symbolic variables symbolically index a set of edges with scalar values while symbolic constants symbolically index a set of assertion graphs with scalar values.

III. Imprecision of Quaternary Abstraction

A. Three Causes of Quaternary Abstraction Imprecision

There are three causes of imprecision in GSTE's quaternary abstraction: one inherited from STE and the other two new in GSTE. In this section, we elaborate on these causes and present a simple illustrative example.

1) Under-constrained input circuit nodes: In (G)STE, the input nodes of a circuit are constrained by the antecedent on each edge of an assertion graph. If an input node is unconstrained, it will be assigned the value of X. Such X's may cause consequent violations in the symbolic simulation.

2) Quaternary state set union: In the fixed-point computation of GSTE, a set of circuit states simulated by each edge in the assertion graph is computed. The set of states is represented approximately by a quaternary assignment and the union of two sets of states is approximated by the bit-wise $\sqcap$ of the two corresponding quaternary assignments. Such unions may introduce additional X's into the verification.

3) Existentially quantified-out symbolic variables: In GSTE, a symbolic variable is a boolean variable that can change its value every time the corresponding edge is visited, and is existentially quantified out after every single step simulation. So after leaving the edge with symbolic variables, circuit nodes associated with these variables may become X.

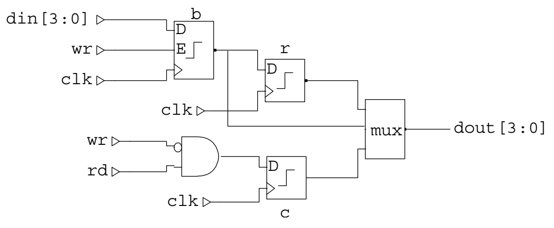


Fig. 2. Buffered Register

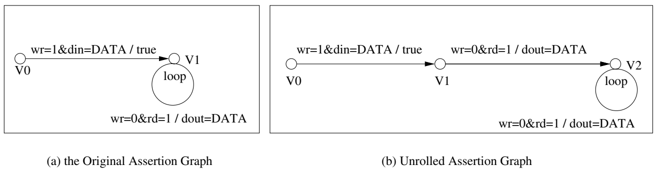


Fig. 3. Assertion Graph for Buffered Register

TABLE I
QUATERNARY SIMULATION

	(V0, V1)	(V1, V1)
0	[din=DATA, wr=1]	TOP
1	[din=DATA, wr=1]	[b=DATA, c=0, rd=1, wr=0]
2	[din=DATA, wr=1]	[b=DATA, rd=1, wr=0]
3	[din=DATA, wr=1]	[b=DATA, rd=1, wr=0]

B. A Simple Illustrative Example of Imprecision

Figure 2 shows a simple buffered-register circuit: the input to the register is first written into a buffer and then transferred into the register in the next clock cycle after the write. Figure 3(a) shows an assertion graph capturing a property to be verified on the circuit: after a write, if there are only read requests, the output is always the data accompanying the write. We run GSTE to verify the circuit against the assertion graph. The step-by-step result of the quaternary simulation is listed in Table I. GSTE reports a violation of the consequent $dout = DATA$ on the loop edge $(V1, V1)$ due to an X assigned to $dout$. However, it is not difficult to observe that the property holds on the circuit. Therefore, GSTE has reported a false negative. The quaternary assignment of $(V1, V1)$ in Iteration 2, $[b = DATA, rd = 1, wr = 0]$, is obtained from the union of the assignment of $(V1, V1)$ in Iteration 1, $[b = DATA, c = 0, rd = 1, wr = 0]$, and the assignment representing the set of new reachable states in Iteration 2, $[b = DATA, r = DATA, c = 1, rd = 1, wr = 0]$. This union introduces additional X's, which lead to the X value of $dout$.

IV. Abstraction Refinement

In this section, we start by reviewing the state-of-art manual abstraction refinement for GSTE and then present AutoGSTE, our approach to automatic abstraction refinement for GSTE. First, we give its overarching framework. Then, we introduce our algorithm for identifying causes of abstraction imprecision from counterexamples generated by GSTE. After that, we discuss how we automate abstraction refinement according to the causes identified. Finally, we discuss the correctness of AutoGSTE.

TABLE II
QUATERNARY SIMULATION

	(V0, V1)	(V1, V2)	(V2, V2)
0	[din=DATA, wr=1]	TOP	TOP
1	[din=DATA, wr=1]	[b=DATA, c=0, rd=1, wr=0]	TOP
2	[din=DATA, wr=1]	[b=DATA, c=0, rd=1, wr=0]	[b=DATA, r=DATA, c=1, rd=1, wr=0]
3	[din=DATA, wr=1]	[b=DATA, c=0, rd=1, wr=0]	[b=DATA, r=DATA, c=1, rd=1, wr=0]

A. Manual Abstraction Refinement

Currently in GSTE, the imprecision due to quaternary state set unions and quantified-out symbolic variables is addressed with two manual strategies. The first strategy is to manually mark a set of circuit nodes as *precise nodes* [9]. For these nodes, by using the parametric representation, their values and the relationships among them are always represented exactly by using boolean expressions as their values. In the above example, we can mark the node c as a precise node. This will prevent the union of $[b = DATA, c = 0, rd = 1, wr = 0]$ and $[b = DATA, r = DATA, c = 1, rd = 1, wr = 0]$. We rerun GSTE to verify the circuit with the precise node against the original assertion graph in Figure 3(a). GSTE reports that the assertion graph holds on the circuit.

The second strategy is to manually apply semantics-preserving transformations to the assertion graph. Typical transformations include case-splitting of an edge and unrolling of a loop. In the case of the above example, we can apply loop-unrolling to the assertion graph in Figure 3(a) to obtain a semantically equivalent assertion graph shown in Figure 3(b). We run the circuit against the refined assertion graph and the step-by-step simulation is listed in Table II. In essence, the loop-unrolling prevents the union of $[b = DATA, c = 0, rd = 1, wr = 0]$ and $[b = DATA, r = DATA, c = 1, rd = 1, wr = 0]$, which are now associated with two separate edges $(V1, V2)$ and $(V2, V2)$, respectively. Intuitively, this refines the assertion graph to mimic the real computation flow of the circuit.

Both strategies above require manual efforts. To apply these strategies, one must have in-depth understanding of the circuit being verified and be able to identify where the imprecision is introduced by GSTE. Therefore, it is highly desired that the causes for imprecision can be automatically identified and these manual strategies can be automated. (For more discussion of the manual strategies, we refer readers to [9]).

B. AutoGSTE: Automatic Abstraction Refinement

AutoGSTE employs a counterexample-guided abstraction refinement loop formed by GSTE, the counterexample analysis algorithm, and the abstraction refinement algorithms, as shown in Figure 4. GSTE applies quaternary abstraction to verify a circuit M against an assertion graph G. If GSTE reports an assertion violation: a consequent in the assertion graph is violated, it generates a counterexample (a symbolic error trace), which is a sequence of simulation steps that lead

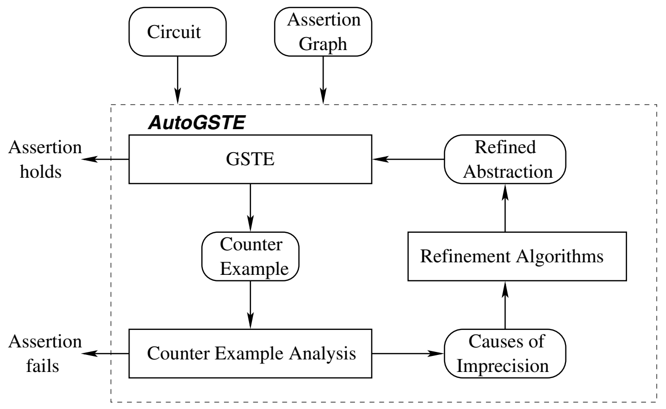


Fig. 4. Automatic Abstraction Refinement Loop for GSTE

to the consequent violation. The counterexample analysis algorithm exams the counterexample to identify the causes of the consequent violation. If it is caused by conflicting values to certain circuit nodes, the assertion fails. Otherwise, the refinement algorithms conduct abstraction refinement according to the causes identified. The model refinement algorithm automatically identifies precise nodes that need to be marked in the circuit. The specification refinement algorithm applies semantics-preserving transformations to certain edges of the assertion graph on-the-fly in the GSTE symbolic simulation.

C. Identification of Causes for Consequent Violation

1) Counterexample Generation from GSTE: When GSTE reports an assertion failure, it also presents a computing history which is a symbolic searching tree of triples $(e, src, dest)$, where e is an edge that GSTE simulated, and src and $dest$ are the quaternary states before and after the simulation of the edge e. Given the triple which causes the consequent violation, we backtrack through the computing history to find the occurrence of the triple (e', src', src), and then continue the backtracking until we reach an edge originated from the initial vertex in the assertion graph. The sequence of triples traversed during the backtracking forms a counterexample CE that leads to the consequent violation. Formally, we define the counterexample as a sequence of triples $CE = [t_1, t_2, \ldots, t_l]$ such that for every index $1 \leq i \leq l$, $t_i = (edge_i, src_i, dest_i)$, where $edge_i$ is the edge traversed in step i and src_i and $dest_i$ are the states before and after this simulation of $edge_i$.

2) Identification of Causes: Once the counterexample is generated, the counterexample analysis algorithm is applied to determine the causes of the consequent violation. If the violation is due to conflicting values to certain circuit nodes in a circuit state and a corresponding consequent, we know the circuit does not satisfy the assertion graph. If the violation is caused by the unknown value of a circuit node, starting from the circuit node, our algorithm backtracks through the error trace and conducts fan-in analysis over the circuit to identify the circuit nodes where the unknown values were introduced.

The analysis algorithm is shown in Figure 5. It inputs the counterexample CE, the post-image function $post$ and

```
Algorithm: $AnalyzeCounterExample(CE[1:l], post)$
1:  $Violators \leftarrow \{n | n \in N, n$ violates $cons(CE[l].edge)$
    due to unknown value$\}$
2:  $Candidate \leftarrow \emptyset, Q \leftarrow \emptyset$
3:  **forall** $n \in Violators$ **do** $Q.enqueue((n, l))$
4:  **while** $Q \neq \emptyset$ **do**
5:      $(n, step) \leftarrow Q.dequeue()$
6:      **if** $n \in N_I$ **then**
7:          add ($n$, $step$, INPUT) to $Candidate$
8:      **else if** $n$ depends on symbolic variables from
        $ant(CE[step-1].edge)$ **then**
9:          add ($n$, $step$, WEAK) to $Candidate$
10:     **else if** $n$ has precise value in $post(CE[step-1].dest)$ **then**
11:         add ($n$, $step$, UNION) to $Candidate$
12:     **else**
13:         $New \leftarrow \{(n', step-1) | n' \in fanin(n), n'$ is unknown in $CE[step-1].dest$ and may contribute
            to the unknown value of $n\}$
14:         $Q.enqueue(New)$
15:     **end if**
16: **end while**
17: **return** $Candidate$
```

Fig. 5. Counterexample Analysis Algorithm

outputs a set of circuit nodes $Candidate$ which get unknown values directly due to inputs, quaternary state set unions, and quantified-out symbolic variables. The cause for the unknown value is attached to each node. In this algorithm, $fanin(n)$ is the function that identifies all circuit nodes that affect the value of a circuit node n.

Given the counterexample CE, we start with the time step l at which a consequent violation is reported. We first decide which circuit nodes have unknown values and cause the violation. This can be done by comparing the quaternary assignment representing the set of states simulated by the edge, $CE[l].dest$, and the quaternary assignment representing the consequent of the edge, $cons(CE[l].edge)$. If a circuit node n has an unknown value in $CE[l].dest$, while having a boolean value in $cons(CE[l].edge)$, we know that n violates $cons(CE[l].edge)$. All violating circuit nodes are put into a queue Q (Steps 1-3).

For each node n in Q, if n is an input node, we identify n and mark the cause as "INPUT" (Steps 4-7). If n depends on a symbolic variable in $ant(CE[l-1].edge)$, we identify n and mark the cause as "WEAK" (Steps 8-9). If n has precise value in $post(CE[l-1].dest)$, we identify n and mark the cause as "UNION" (Steps 10-11). All the identified circuit nodes are added into the node-cause set $Candidate$.

If none of the above conditions holds, we backtrack to the previous time step $l-1$. We also conduct a one-level fan-in analysis from n in the circuit to identify all nodes that affect n. Among these nodes, for each node n' with an unknown value in the quaternary assignment of $CE[l-1].dest$ and that may contribute to the unknown value of n in $CE[l].dest$, we repeat this analysis for n' until one of the three causes above is found in the counterexample, by putting n' into Q (Steps 13-14). We improve the accuracy of this analysis through examining the different types of gates case-by-case.

It is possible that at the same time, a circuit node can get unknown value due to both a state set union and a quantified-out symbolic variable. In our refinement loop, one cause is identified at a time and eventually both causes are identified.

Using the parametric representation, a circuit node can have a partially unknown value, i.e., under certain condition, node n has an unknown value. In the backtracking, we keep track of the condition under which node n has an unknown value. In the fan-in analysis of n, the condition is propagated to n' the same way as backward reasoning in bidirectional GSTE [8].

D. Refinement Algorithms

1) Constraining Input Nodes: If a circuit node identified by the analysis algorithm is an input node, a symbolic constant or variable is introduced in the antecedent of the corresponding edge of the assertion graph to rule out the unknown value of this node. In STE, this type of imprecision can be addressed by constraining the inputs using symbolic constants. In [13], an approach to automatically constraining the input nodes with symbolic constants has been proposed for STE. The same heuristics can be applied in our approach. However, in GSTE, constraining symbolic constants on loop edge may cause the input nodes have the same input signals every time the edge is simulated. Therefore, if the edge is on a loop, a symbolic variable should be introduced.

2) Marking Precise Nodes: After we identify the circuit nodes (i.e., the nodes in $Candidate$) that get unknown values due to quaternary state set unions or quantified-out symbolic variables, we can refine the circuit model M by marking these nodes as precise nodes, and then rerun GSTE with the refined model. The correctness of this refinement is guaranteed by Lemma 4.1. Let $precise(M, p)$ be the circuit model with the circuit node p marked as a precise node in M.

Lemma 4.1: (Yang and Seger [9])
Given G, $\forall p \in N_S(M), M \models G$ *iff* $precise(M, p) \models G$.

There are two strategies in marking the identified nodes: we can mark them either all at once or one at a time. It is easy to see that there is a trade off between these two strategies. The first strategy may converge faster while it may lead to more precise nodes than necessary. The second approach may find a smaller set of precise nodes while it may require more iterations to converge. Heuristics can be applied to improve these two strategies, for instance, an effective heuristic in practice is to pick control nodes as precise nodes over data nodes when using the second strategy.

3) On-the-fly Transformations of Assertion Graphs: Making too many nodes precise may cause state space explosions. And identifying a minimal set of circuit nodes as precise nodes is challenging. An alternative approach to address imprecision caused by unions or symbolic variables is to conduct on-the-fly semantics-preserving transformations to the assertion graph G. The motivation is that in some cases, the circuit nodes only need to keep precise values on some assertion graph edges and making them precise in the whole simulation is too costly.

```
Algorithm: $ExtendedGSTE(G, post, Edges)$
1: for all $e$ from $v_0$ do $Q.enqueue((e, ant(e)))$
2: for all $e \in Edges$ do $Hash(e) \leftarrow \emptyset$
3: while $Q \neq \emptyset$ do
4:   $(e', sim(e')) \leftarrow Q.dequeue()$
5:   for all successor edge $e$ of $e'$ do
6:     $NewState \leftarrow post(sim(e')) \cap ant(e)$
7:     if $e \in Edges$ then
8:       if $e$ is marked UNION then
9:         if $NewState \notin Hash(e)$ then
10:          $Hash(e) \leftarrow Hash(e) \cup \{NewState\}$
11:          $Q.enqueue((e, NewState))$
12:        end if
13:      else if $e$ is marked WEAK then
14:        $Weak \leftarrow \{$states derived from $NewState$ by assigning all combination of boolean values for symbolic variables in $ant(e)\}$
15:        for all $s \in Weak$ do
16:          if $s \notin Hash(e)$ then
17:            $Hash(e) \leftarrow Hash(e) \cup \{s\}$
18:            $Q.enqueue((e, s))$
19:          end if
20:        end for
21:      end if
22:    else
23:      $sim(e) \leftarrow sim(e) \cup NewState$
24:      if there is a change in $sim(e)$ then $Q.enqueue(e, sim(e))$
25:    end if
26:  end for
27: end while
28: for all $e \in V$ do consequent check
```

Fig. 6. GSTE Extended with Semantic-Preserving Transformation

Our algorithm for on-the-fly transformation of assertion graphs, shown in Figure 6, extends the basic GSTE algorithm to support dynamic loop-unrolling and case-splitting of assertion graph edges in the symbolic simulation. Here $Edges = \bigcup_{(n,i) \in Candidate} \{CE[i].edge\}$, namely the set of assertion edges that need to be transformed. The algorithm includes two parts: the on-the-fly transformation (Steps 7-21) and the normal GSTE fixed-point computation (Steps 22-25). In the transformation, to keep precise states, we built a hash table for each edge e in $Edges$ (Steps 1-2). When a new post-image $NewState$ of edge e is generated, we first check if edge e is marked as UNION. If yes, we exam the hash table. If $NewState$ is not in the hash table, $NewState$ is added to the hash table and the simulation continues with $NewState$; otherwise, a fixed point is reached (Steps 8-12). In essence, this realizes loop-unrolling: Loops in G are expanded to mimic the real computation flow of the circuit. The correctness of this loop-unrolling transformation is guaranteed by Lemma 4.2. Let l be a set of edges forming a loop in G and $unroll(G, l)$ be the assertion graph with l unrolled in G.

Lemma 4.2:

Given M, for any loop l in G,
$M \models G$ *iff* $M \models unroll(G, l)$.

The key intuition of the proof for Lemma 4.2 is as follows. For any finite path in G, there is one and only one finite path in $unroll(G, l)$ with the same length and labels, and vice versa. (Space limitation precludes presentation of detailed proofs for Lemma 4.2 and the lemmas and theorems below.)

If edge e is marked as WEAK, we generate all possible combinations of boolean assignments to the symbolic variables in $ant(e)$, and apply these combinations to $NewState$ to get a set of states $Weak$. Each state in $Weak$ which is not reached before is put into the queue (Steps 13-21). This is equivalent to case splitting of certain edges. The correctness of this transformation is guaranteed by Lemma 4.3. Let $split(G, e)$ be the assertion graph with edge e case-split in G.

Lemma 4.3:

Given M, $\forall e \in E(M), M \models G$ *iff* $M \models split(G, e)$.

The key intuition of the proof for Lemma 4.3 is as follows. As e is case-split into $e_1, \cdots, e_k$, where $ant(e_1) \cup \cdots \cup ant(e_k) = ant(e)$ and $cons(e_1) = \cdots = cons(e_k) = cons(e)$. In $split(G, e)$, $e_1, \cdots, e_k$ cover all the possible cases which are covered by e in G, without introducing any new cases.

E. Correctness of AutoGSTE

The abstraction-refinement loop terminates when either GSTE reports verification success or it reports a consequent violation due to conflicting values to certain circuit nodes. The termination of this loop is guaranteed by Theorem 4.1.

Theorem 4.1:

AutoGSTE(M,G) terminates.

The basic idea for proving this theorem is that the circuit to verify is finite-state. Our refinement algorithms add input node constraints, mark precise nodes, and apply semantics preserving transformations in a monotonic fashion. The correctness of whole *AutoGSTE* immediately follows from Lemma 4.1, Lemma 4.2, Lemma 4.3, and Theorem 4.1.

Theorem 4.2:

$M \models G$ *iff AutoGSTE(M,G)* returns *true*.

V. EXPERIMENTAL RESULTS

We have implemented AutoGSTE in the Intel *Forte* environment [10] and upon GSTE. We have conducted experiments on a family of benchmark FIFO circuits from Intel, and analyzed how our approach scale with the depth of FIFO. Figure 7 shows a simple stationary 3-entry 8-bit FIFO circuit. The assertion graph for this circuit is shown in Figure 8. The assertion graph checks whether the empty and full signals of the FIFO circuit are set correctly. The assertion graph is independent of the circuit implementation and data width, and exposes imprecision of quaternary abstraction due to both state set

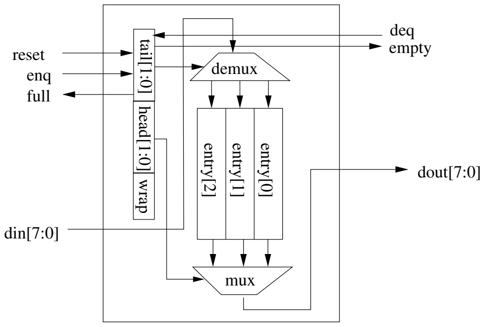


Fig. 7. Stationary FIFO Implementation

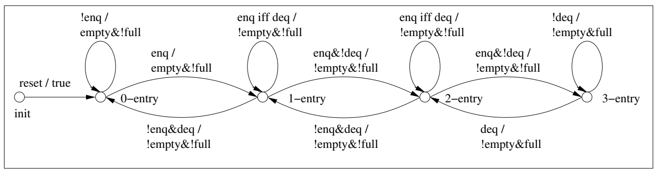


Fig. 8. FIFO Assertion Graph

TABLE III
MODEL REFINEMENT RESULTS FOR 8-BIT FIFOS

Circuit		Mark precise nodes all at once					Mark precise nodes one a time			
FIFO Depth	# of Nodes	# of Iter.	# of P. Nodes	Time (Sec.)	BDD Nodes	Mem (MB)	# of P. Nodes.	Time (Sec.)	BDD Nodes	Mem (MB)
3	181	1	5	0.12	10232	17	3	0.26	8996	17
8	296	1	7	0.4	32923	19	4	0.81	26708	18
16	476	1	9	1.1	72189	22	5	2.37	58250	22
24	787	1	11	2.38	131236	24	6	6.83	104246	24

unions and quantified-out symbolic variables. All experiments were conducted on a workstation with 3GHz Intel® Xeon® processor with 2GB memory, and all verifications were done on the original circuits with no prior abstraction.

Table III lists the verification results for model refinement. In [9], manual analysis showed the imprecision for the stationary FIFO implementation is caused by different values in the head and tail pointers as well as the wrap bit. Our counter example analysis algorithm identified the same set of potential circuit nodes. If we mark these circuit nodes all at once, only one refinement iteration is needed, and it makes all the non-data-path elements in the circuit precise, which has the same effect as manual effort. If we mark precise nodes one at a time, more iterations are needed. However, interestingly, it finds a smaller set of precise nodes than manual analysis. That is we only need to make the head pointer and the first bit of tail pointer precise. This leads to a smaller number of BDDs generated, which takes less memory than the first strategy. Therefore, it is reasonable to conclude the first strategy requires fewer iterations but more BDDs, while the second strategy is more likely to converge into a smaller set of precise nodes but take more iterations. In practice, there is no definite evidence that which strategy would terminate more quickly since in the first strategy, each iteration takes more time and in the second strategy, more iterations are needed.

Table IV lists the verification results for specification refinement. In the stationary implementation, every assertion

TABLE IV
SPECIFICATION REFINEMENT RESULTS FOR 8-BIT FIFOS

Circuit	GSTE on Original assertion graph					Semantic-Preserving Transformation				
FIFO Depth	# of Edges	Time (Sec.)	BDD Nodes	Mem (MB)	Result	# of Edges	Time (Sec.)	BDD Nodes	Mem (MB)	Result
3	11	0.01	5	17	Unknown	31	0.23	6	17	Pass
8	26	0.02	5	17	Unknown	201	2.69	6	19	Pass
16	50	0.04	5	17	Unknown	785	17.31	6	26	Pass
24	74	0.07	5	17	Unknown	1753	54.23	6	39	Pass

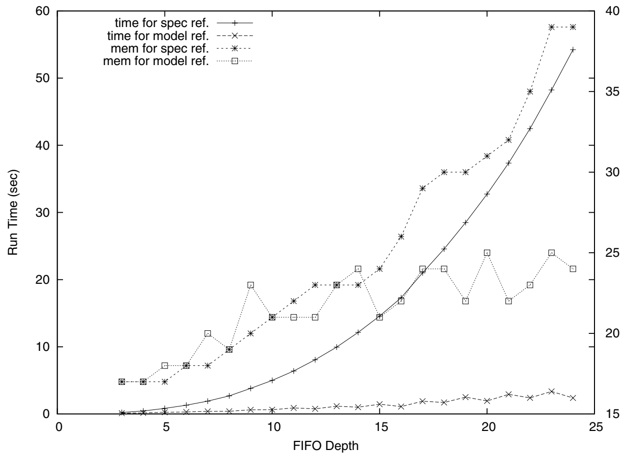


Fig. 9. Time and Memory for Verification of FIFOs

graph edge corresponds to several different combinations of the head and tail pointer values, so the refined assertion graphs grow quickly as the depth of FIFO increases. It is very difficult to conduct this refinement manually [9]. Our counter-example analysis algorithm identified all the edges to be refined, and our refinement algorithm applies the appropriate transformations. The transformations eliminate all the UNION and WEAK cases without increasing the number of BDDs needed. Different from the model refinement, here the memory usage is largely dependent on the number of actual assertion graph edges generated rather than the number of BDDs.

We plot the time and space complexity data of both model refinement and specification refinement in Figure 9. The time and space complexities for model refinement grow almost linearly while those of the specification refinement grow exponentially, with respect to the depth of FIFO. The reason for the later case is that the symbolic variable on every loop edge will split every state into two states carrying different values for the symbolic variable, the verification complexity is determined by the stationary implementation itself. In this experiment, the model refinement approach is more favorable than the specification refinement approach. However, in some cases, the circuit nodes only need to keep precise values on certain edges, or the number of different states simulated by these edges is small, specification refinement would be more efficient than model refinement, and is less likely to suffer from state space explosions. Furthermore, the specification refinement reveals the real computation flow of the circuit, which allows the initial specification to be very high level, and provides a good guidance for debugging in practice.

As we can observe, the verification time for our automatic refinement approach is fairly small, and the amount of memory used is of reasonable size. None of the two kinds of refinement is easy to conduct manually as the sizes of the circuit and assertion graph increase. Our experiments demonstrate the correctness and effectiveness of our approach.

VI. CONCLUSIONS

In this paper, we have presented AutoGSTE, a comprehensive approach to automatic abstraction refinement for GSTE. It completely addresses the imprecision of GSTE's quaternary abstraction caused by under-constrained input circuit nodes, quaternary state set join, and quantified-out symbolic variables. Its application to small to large size circuits has demonstrated that it is able to quickly converge to an abstraction upon which GSTE can either verify or falsify an assertion graph efficiently.

Regarding model refinement, further research is needed to explore effective methods for determining a minimal set of circuit nodes as precise nodes in order to minimize the state space that has to be explored. Regarding specification refinement, further research is needed to develop heuristics that can reduce unnecessary loop-unrollings and case-splittings. It is also interesting to see how these two automatic refinement approaches can be integrated to further speed up the convergence of our refinement loop into the right level of abstraction.

ACKNOWLEDGMENT

The authors thank Bryant York for his support. This research is partially supported by Intel Equipment Grant #33768.

REFERENCES

[1] 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*, vol. 14, no. 4, pp. 413–422, April 1995.

[2] C.-J. Seger and R. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories," *Formal Methods in System Design*, vol. 6, no. 2, pp. 147–190, March 1995.

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

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

[5] O. Coudert, J. Madre, and C. Berthet, "Verifying temporal properties of sequential machines without building their state diagrams," in *Proc. of CAV'90*, 1990, pp. 23–32.

[6] K. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic, 1993.

[7] J. Yang and C.-J. Seger, "Generalized symbolic trajectory evaluation," *Intel SCL Technical Report*, 2000.

[8] J. Yang and C.-J. H. Seger, "Introduction to generalized symbolic trajectory evaluation," *Transaction on VLSI Systems*, vol. 11, no. 3, June 2003.

[9] J. Yang and C. J. H. Seger, "Generalized symbolic trajectory evaluation - abstraction in action," in *Proc. of FMCAD*, November 2002.

[10] C.-J. Seger, R. Jones, J. O'Leary, T. Melham, M. Aagaard, C. Barrett, and D. Syme, "An industrially effective environment for formal hardware verification," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 9, 2005.

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

[12] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification*, 2000, pp. 154–169.

[13] R. Tzoref and O. Grumberg, "Automatic refinement and vacuity detection for symbolic trajectory evaluation," in *STE Symposium, CAV*, 2006, pp. 190–204.

[14] J.-W. Roorda and K. Claessen, "Sat-based assistance to abstraction refinement for symbolic trajectory evaluation," in *Proc. of CAV*, 2006.