# Optimizing Automatic Abstraction Refinement for Generalized Symbolic Trajectory Evaluation

Yan Chen
Dept. of Computer Science
Portland State University
Portland, OR, 97207
chenyan@cs.pdx.edu

Fei Xie
Dept. of Computer Science
Portland State University
Portland, OR, 97207
xie@cs.pdx.edu

Jin Yang
Strategic CAD Labs, DTS
Intel Corporation
Hillsboro, OR 97124
jin.yang@intel.com

## ABSTRACT

In this paper, we present a suite of optimizations targeting automatic abstraction refinement for Generalized Symbolic Trajectory Evaluation (GSTE). We optimize both model refinement and spec refinement supported by AutoGSTE: a counterexample-guided refinement loop for GSTE. Experiments on a family of benchmark circuits have shown that our optimizations lead to major efficiency improvements in verification involving abstraction refinement.

## Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*Verification, Optimization*

## General Terms

Verification, Performance, Algorithms

## Keywords

Model Checking, Generalized Symbolic Trajectory Evaluation, Automatic Abstraction Refinement

## 1. INTRODUCTION

Generalized Symbolic Trajectory Evaluation (GSTE) [10] is a highly scalable hardware model checking technique based on a form of quaternary symbolic simulation. It extends Symbolic Trajectory Evaluation (STE) [4, 5, 7] to verification of properties over infinite time intervals, while maintaining the efficiency, capacity, and ease-to-use of STE. 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. Quaternary abstraction allows circuit nodes to have unknown values and propagates these unknown values in verification. Propagation of unknown values can both reduce the sizes of the state space representations and cause false negatives in verification. The false negative problem is further worsened by the fixed-point computation of GSTE. Wide application of GSTE has been hindered by manual efforts which are needed in identifying the right level of abstraction that enables successful verification at reasonable time and memory usages.

In [3], we proposed AutoGSTE, a comprehensive approach to automatic abstraction refinement for GSTE. It 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 automatically verify circuits that could not be automatically verified by GSTE before.

Application of AutoGSTE has also revealed that the verification complexities grow rapidly as the number of refinement iterations increases and the model and the spec are made more precise. This is affected by both the counterexample analysis algorithm and the refinement algorithms. It is, therefore, highly desired to have optimized algorithms that can facilitate quicker termination of the refinement loop to a less detailed abstraction that can either verify or falsify the spec.

In this paper, we present a suite of optimizations targeting automatic abstraction refinement supported by AutoGSTE. In the counterexample analysis, as we conduct backward chasing of unknown values, instead of chasing all the unknowns, we conduct case-by-case analysis at each circuit node so that we only chase unknowns that may contribute to the false negatives. In the model refinement, instead of making circuit nodes identified by the counterexample analysis have precise values throughout the whole verification, we extend the precise node concept so that precise nodes can have lifespans. In the spec refinement, instead of requiring exact matches of state sets to determine terminations of loop-unrollings and case-splittings, we conduct state set containment checks that effectively leverage the quaternary state representation to eliminate unnecessary loop-unrollings and case-splittings.

We have implemented all the above optimizations in the Intel *Forte* environment [8] and upon GSTE and AutoGSTE. We have conducted experiments on a family of benchmark FIFO circuits from Intel. The experiments have shown that our optimizations lead to major efficiency improvements in verification involving abstraction refinement.

**Related Work.** There has been much research on abstraction refinement for model checking of both hardware and software. Space limitation precludes a detailed discussion. Closely relevant to our research is the work on abstraction refinement for STE. In [9], symbolic constants are automatically introduced to constrain under-constrained input nodes of circuits based on counterexamples from

STE. In [6], a SAT-based algorithm was developed to assist manual refinement of STE assertions. In [1], an automatic symbolic indexing discovery technique was introduced for STE.

## 2. BACKGROUND

### 2.1 GSTE

GSTE checks whether a circuit satisfies the spec given by an *assertion graph*. An assertion graph is defined as a quintuple $G = (V, v_0, E, ant, cons)$, 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 assertion graph from the initial vertex is an STE assertion. Therefore, a circuit $M$ *satisfies* an assertion graph $G$ if it satisfies every STE assertion derived from $G$ by following a finite path from the initial vertex. 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 an assertion graph against a circuit, GSTE performs a symbolic simulation 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. Furthermore, 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 [10].

### 2.2 AutoGSTE

Using quaternary abstraction, each node has quaternary values $\{0, 1, X, \top\}$ instead of boolean values and 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 the unknown value $X$. The empty set is represented by assigning $\top$, which denotes the overconstrained value, to one or more nodes. 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 $[p=1, q=X, r=1]$.

The imprecision of quaternary abstraction may lead to false negatives: STE and GSTE may report the result of X 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, and (3) existentially quantified-out symbolic variables.

AutoGSTE [3] is a comprehensive approach to automatic abstraction refinement for GSTE, which addresses abstraction imprecision due to the three causes identified above. 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 spec refinement according to the causes identified. The analysis algorithm identifies these causes by backtracking through the counterexample and conducting fan-in analysis on 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 spec refinement, according to the causes identified, the right 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.

## 3. OPTIMIZING AUTOGSTE

In this section, we present our optimizations of the counterexample analysis, model refinement, and spec refinement algorithms. For each optimization, we first discuss the inefficiency in the original algorithm and we then discuss our improvement.

### 3.1 Optimizing Counterexample Analysis

#### *3.1.1 Inefficiency in Counterexample Analysis*

A counterexample generated by GSTE is a sequence of state transitions leading to a consequent violation. Each transition is represented by a triple $(edge, src, dest)$ where $edge$ is the assertion graph edge simulated, and $src$ and $dest$ are the circuit states before and after the simulation of $edge$. The counterexample analysis algorithm of AutoGSTE is shown in Figure 1. It inputs the counterexample $CE$, the post-image function $post$ and outputs a set of circuit nodes $Candidate$ which get unknown values directly due

```
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 un-
            known 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 $Candidate$
```

**Figure 1: Counterexample Analysis Algorithm**

to inputs, quaternary state set unions, and quantified-out symbolic variables. In this algorithm, $fanin(n)$ is the function that identifies all one-level fan-in nodes of $n$.

Given the counterexample $CE$, the algorithm starts with the time step $l$ at which a consequent violation is reported. It first decides which circuit nodes have unknown values and cause the violation. If a circuit node $n$ has an unknown value in $CE[l].dest$, while having a boolean value in $cons(CE[l].edge)$, then $n$ 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, the algorithm identifies $n$ and mark the cause as "INPUT" (Steps 4-7). If $n$ depends on a symbolic variable in $ant(CE[l-1].edge)$, it identifies $n$ and marks the cause as "WEAK" (Steps 8-9). If $n$ has precise value in $post(CE[l-1].dest)$, it identifies $n$ and marks 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, the algorithm backtracks to the previous time step $l-1$. It also conducts 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 $CE[l].dest$, this analysis is repeated for $n'$ until one of the three causes above is found in the counterexample, by putting $n'$ into $Q$ (Steps 13-14).

In the backtracking phase (Steps 13-14) of the above counterexample analysis algorithm, the backtracking is conducted for each fan-in node $n'$ of $n$ if $n'$ has an unknown value in $CE[l-1].dest$. This strategy is conservative and guaranteed to identify all causes for the false negative. However, it can be very inefficient in many cases when the value of $n$ is not affected by $n'$. In such cases, it not only conducts unnecessary backtracking, but also identifies false causes which will lead to unnecessary refinement actions that lead to an overly refined model or spec. Figure 2 illustrates such a case: a simple mux circuit: When the control signal $C$ is 0, $A$ is selected as the output; otherwise, $B$ is selected. For the counterex-

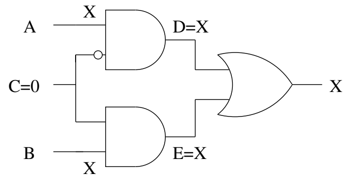

**Figure 2: A Mux Circuit Example**

ample shown in Figure 2, the algorithm will backtrack on both $A$ and $B$ even when the value of the control signal $C$ is known. In fact, when $C = 0$ (or $C = 1$, respectively) is known, we only need to backtrack on $A$ (or $B$) and eliminate the imprecision in $A$ (or $B$). When $A$ and $B$ are complex, the original algorithm can result in significant increases in verification time and memory usages.

Another problem for the counterexample analysis is the lack of backward reasoning capability with symbolic indexing. With symbolic indexing, the circuit nodes may contain partially unknown values, rather than X. GSTE uses dual-rail representation [2] to encode the quaternary state representation and symbolic indexing. Each symbolic quaternary value is encoded as a pair of binary values $(H, L)$, where $H$ and $L$ represent the conditions under which the value should be true or false, respectively. Therefore the condition under which the value is unknown can be represented as $H \wedge L$. The counterexample analysis algorithm did not consider the condition under which the value is unknown and, therefore, may identify unnecessary circuit nodes. For example, suppose a fan-in of a circuit node is $(c, True)$ while the fan-out of the node is $(True, \neg c)$ in the counterexample trace. That is when $c$ is True, the fan-in is X, the fan-out is 1; when $c$ is False, the fan-in is 0, the fan-out is X. If the partially unknown of fan-out causes the imprecision, we do not need to continue to chase for the partially unknown of its fan-in, because under the condition that the fan-out is unknown, the fan-in has precise value rather than X. Therefore, we can conclude that the imprecision is caused by the Union at this circuit node and not by its fan-in.

### 3.1.2 Backward Reasoning

We optimize the counterexample analysis algorithm by conducting more accurate fan-in analysis at each circuit node as the algorithm is backtracking through the circuit. The optimized counterexample analysis algorithm is shown in Figure 3. Steps 3-5 add an

```
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: for all $n \in Violators$ do $Q.enqueue((n, l, True))$
 4: while $Q \neq \emptyset$ do
 5:   $(n, step, XCond) \leftarrow Q.dequeue()$
 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
11:     get value $(H, L)$ of node $n$ from $CE[step].dest$
12:     $XCond' \leftarrow H \wedge L \wedge XCond$
13:     $State_{fanin} \leftarrow CE[step-1].dest$
14:     for all $n' \in fanin(n)$ do
15:       Get value $(H', L')$ of node $n'$ from $State_{fanin}$
16:       $(H', L') \leftarrow (XCond' \wedge H', XCond' \wedge L')$
17:       Update the value of $n'$ with $(H', L')$ in $State_{fanin}$
18:     end for
19:     if $n$ has precise value in $post(State_{fanin})$ then
20:       add $(n$, $step$, UNION) to $Candidate$
21:     else
22:       get the Boolean function $f$ for $n$ from $post$
23:       for all $n' \in fanin(n) \wedge n'$ has precise value $v$ in
          $State_{fanin}$ do substitute $n'$ with $v$ in $f$
24:       canonicalize $f$
25:       $New \leftarrow \{(n', step-1, XCond') | n' \in$
          $depend(f)\}$
26:       $Q.enqueue(New)$
27:     end if
28:   end if
29: end while
30: return $Candidate$
```

**Figure 3: Counterexample Analysis Algorithm with More Accurate Fan-in Analysis**

extra field XCond to the queue $Q$, which indicates the constraint to $n$ propagated from its fan-outs. This condition will propagate into its fan-ins during the backtrack. Steps 11-18 propagate the XCond into the fan-ins of node $n$, constrain the state $CE[step-1].dest$, and store the constrained state as $State_{fanin}$. Here we do not update the states in the original $CE[step-1].dest$, because one node can be the fan-in of several other nodes, and these nodes can have different XCond's. So we need to store different XCond's in the queue. If the post-image of $State_{fanin}$ can derive a precise value for node $n$, we identify $n$ and mark the cause as UNION (Steps 19-20). By propagating the XCond, we eliminate the unnecessary refinement for partial unknowns as indicated in the previous section. Otherwise, we extract the Boolean function $f$ for $n$, and for each fan-in $n'$ of $n$ that has precise value in $State_{fanin}$, we substitute $n'$ with its precise value in function $f$, simplify the function by BDD canonicalization, and put all the remaining circuit nodes in $f$ into the queue (Steps 21-27). Since the simplified Boolean function $f$ can eliminate some potentially unused circuit nodes, we reduce the number of unnecessary backtracking. Note that the effectiveness of the simplification depends on the order that we conduct substitutions and the order of BDD variables, this process does not always derive the most simplified function. But in most cases, especially when the fan-in values have no dependencies between each other, most of the non-contributing fan-ins can be eliminated.

## 3.2 Optimizing Model Refinement

### *3.2.1 Inefficiency in Model Refinement*

In AutoGSTE's model refinement, after the circuit nodes (i.e., the nodes in $Candidate$) that get unknown values due to quaternary state set unions or quantified-out symbolic variables are identified, the circuit model $M$ is refined by marking these nodes as precise nodes, and then rerun GSTE with the refined model. By marking these nodes as precise, GSTE forces these nodes always have precise value by utilizing symbolic indexing. To ensure that the abstraction refinement loop terminates, marking of precise nodes are done in a monotonic way, i.e., once a node is marked as precise, it will be precise throughout the whole verification run.

This approach can be quite inefficient when certain circuit nodes only need to be precise during selected time frames in the verification. Making such nodes always precise in general will lead to larger number of BDDs and, therefore, more memory and time usages. This is especially true when verifying a staged design. Within each stage of the design, we need to know the precise control flow, however, after entering the next stage, we often care about the output data of the previous stage rather than its actually control flow, provided that the two stages are independent. Marking control signals of a stage as precise nodes only during the computation of the stage rather than the whole design is especially effective.

### *3.2.2 Precise Nodes with Lifespans*

To address the above problem, we extend the precise node concept of GSTE so that precise nodes now have lifespans. In practice, we realize this concept by marking different precise nodes on different assertion edges. After the counterexample analysis phase, if $n$ causes imprecision when going through assertion edge $e$, we mark $n$ as precise only on edge $e$. This will be beneficial in the staged design, because at different stages we only need to precise circuit nodes for a particular stage rather than the whole design.

To ensure that the refinement loop still terminates, we require that marking of precise nodes be monotonic with respect to each edge of the assertion graph. That is once a node is marked precise on an edge, it will remain precise on this edge throughout the abstraction refinement loop.

## 3.3 Optimizing Spec Refinement

### *3.3.1 Inefficiency in Spec Refinement*

AutoGSTE's spec refinement algorithm is shown in Figure 4, which extends the basic GSTE algorithm. $Edges$ is the set of as-

```
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 as-
            signing all combination of boolean values for sym-
            bolic 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
```

**Figure 4: GSTE Extended with Spec Refinement**

sertion graph edges that are identified in the counterexample analysis and 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 examine 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. 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 $Weak$ not reached before is put into the queue (Steps 13-21). This is equivalent to case splitting of certain edges.

The inefficiency of this algorithm lies in the fixed-point computation based on the hash tables on each edges. When checking a state in a hash table, it requires an exact match of the state. This can be very inefficient when the quaternary state representation and symbolic indexing is used. For example, if an assertion edge generates three different states: $s_0 = [A = 1, B = 0], s_1 = [A = v_1, B = 0], s_2 = [A = v_1 \vee v_2, B = v_2]$. The algorithm will treat them as different states and put them into the queue to continue the computation from each state. However, it is easy to see that $s_2$ is more general then $s_1$, while $s_1$ is more general then $s_0$. In this

```
Algorithm: $Containment(s_1, s_2)$
1: $Ok \leftarrow T$
2: for all $(n, Val_2) \in s_2$ do
3:     get value $Val_1$ of node $n$ from $s_1$
4:     $Ok \leftarrow Ok \wedge (Val_2 \rightarrow Val_1)$
5:     $Quant$ ← variables that only appear in the current
       and previous $(Val_2 \rightarrow Val_1)$ pairs
6:     $Ok \leftarrow \exists Quant.Ok$
7: end for
8: return $Ok$
```

**Figure 5: State Set Containment Check**

case, we only need to put $s_2$ in the queue. Therefore, the algorithm above may cause the unnecessary state space blow-ups.

### *3.3.2 Containment Check for Spec Refinement*

Our approach to address this problem is to add a state containment check procedure to replace the equality test in the above algorithm. Figure 5 shows the state containment check algorithm. It takes two quaternary state $s_1, s_2$ and check whether $s_1$ is contained by $s_2$. For each circuit node $n$, we find the corresponding value $Val_1, Val_2$ in circuit state $s_1, s_2$ respectively. We then build the Boolean expression $Ok = \bigwedge_n (Val(n, s_2) \rightarrow Val(n, s_1))$, where $Val(n, s)$ represents the value of node $n$ in state $s$. To prevent the expression growing too large, we enforce a fixed order for circuit nodes in the state representation, and existentially quantify out the variables that only appear in the current and previous $(Val(n, s_2) \rightarrow Val(n, s_1))$ pairs. When all variables in $Ok$ are quantified out, if $Ok$ is true, $s_1$ is contained in $s_2$.

We modify the AutoGSTE spec refinement algorithm as follows. Every time a new state $s'$ is generated, we check if $s'$ is contained in a previously generated state $s$ for the same edge or vice versa. If $s'$ is contained in $s$, we discard $s'$. If $s$ is contained in $s'$, we replace $s$ with $s'$ in the hash table, remove $s$ from the queue, and put $s'$ in the queue.

## 4. EXPERIMENTAL EVALUATIONS

We have implemented the above optimizations in the Intel *Forte* environment [8] and upon GSTE and AutoGSTE. And we have conducted experiments on a family of benchmark FIFO circuits from Intel. 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.

Our first example is the classical stationary FIFO circuit. Figure 6

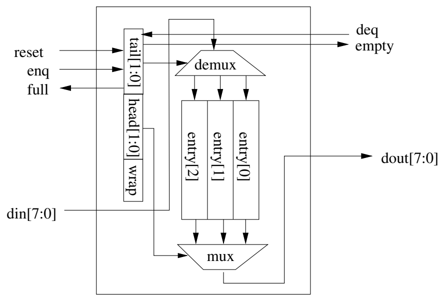

**Figure 6: Stationary FIFO Implementation**

shows a simple stationary 3-entry 8-bit FIFO circuit. In this experiment, we use an assertion graph that checks whether the empty

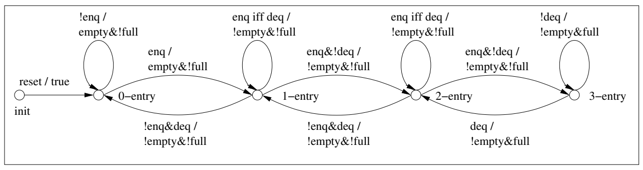

**Figure 7: FIFO Assertion Graph**

| Assertion | | Pure Spec Ref | | | Hybrid w/o cont. | | | Hybrid w/cont. | | |
|---|---|---|---|---|---|---|---|---|---|---|
| FIFO Depth | # of Edges | # of Edges | Time (Sec.) | Mem (MB) | # of Edges | Time (Sec.) | Mem (MB) | # of Edges | Time (Sec.) | Mem (MB) |
| 3 | 11 | 31 | 0.23 | 11 | 51 | 0.32 | 11.8 | 25 | 0.3 | 11.7 |
| 8 | 26 | 201 | 2.69 | 15 | 296 | 2.98 | 14.3 | 145 | 2.56 | 15.8 |
| 16 | 50 | 785 | 17.31 | 22.5 | 1104 | 16.97 | 21.6 | 545 | 13.9 | 17 |
| 24 | 74 | 1785 | 54.23 | 33.8 | 2424 | 50 | 33.1 | 1201 | 44.3 | 20 |

**Table 1: Refinement Results for 8-bit FIFOs**

and full signals of the FIFO circuits are correctly set for all types of FIFO circuits. The assertion graph is shown in Figure 7. The assertion graph is independent of the circuit implementation and data width, and exposes imprecision of quaternary abstraction due to both UNION and WEAK. We have conducted verifications for different depths of FIFO. Table 1 compares the verification results for the pure spec refinement algorithm and a hybrid refinement approach where spec refinement is used to address UNION and model refinement is used to address WEAK. Such a hybrid approach is of interest in that it allows the original assertion graph to be high-level and the refinement algorithm automatically adapts the graph to the real computation flow of the circuit without being too detailed (i.e., case-splitting all symbolic variables). Using the hybrid refinement without containment check, we got even more assertion edges than the pure spec refinement approach, because many edges are contained in other edges and the algorithm cannot detect such redundancy. With containment check, we reduced about half of the assertion edges and got lower running time and memory usages than the pure spec refinement.

Our second example is a speculative design whose high-level model is shown in Figure 8. Multiple components are connected

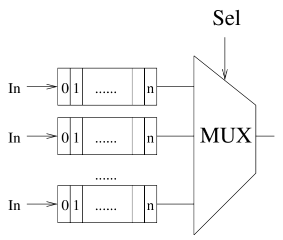

**Figure 8: Speculative Design of FIFO**

to a control unit and are executing in parallel. The final results is determined by a control signal which selects the appropriate fan-in. Here, we use 16-depth 8-bit FIFOs as components and a mux as the control unit. The assertion graphs used in this experiment are constructed based on the original assertion graph above and reflect the control flow of the speculative design. We plot the time and memory usages of model refinement with and without backward reasoning optimization in Figure 9. As discussed in [3], to check each FIFO component, we need to mark at least 5 circuit nodes precise. For our largest experiment with 24 fan-ins to the mux, we need to mark 120 circuit nodes precise to make all the fan-ins of the mux precise. However, by backward reasoning, only one of the

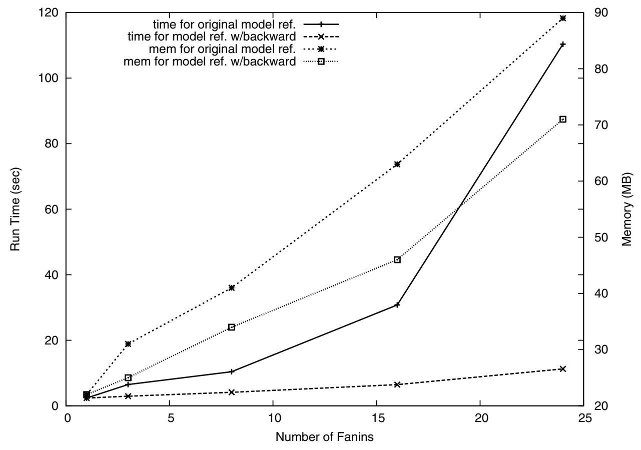

**Figure 9: Time and Memory for Speculative Design**

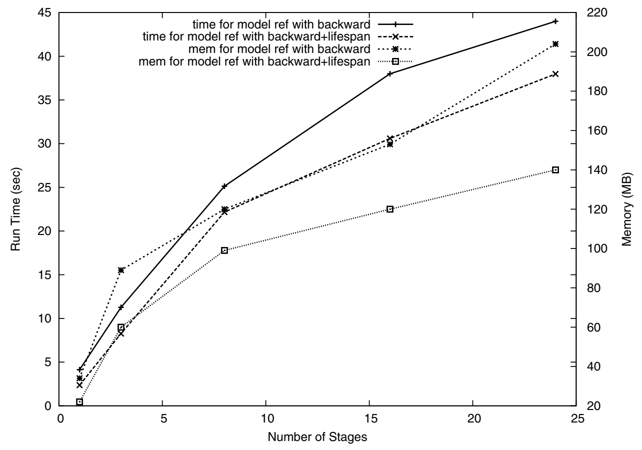

**Figure 11: Time and Memory for Staged Design**

fan-ins needs to be marked as precise according to the *sel* signal, which significantly reduces time and memory usages. This example suggests that the backward reasoning approach is particularly effective when control signals are indicated in the assertion graph.

Our last example is a staged design with speculative execution, further extending the scale and complexity of FIFO circuits. Figure 10 shows the high-level design of this circuit. Instead of connecting one FIFO to each fan-in of the mux, we connect a staged design of FIFOs. Here we fixed the fan-in of the mux to be 16, and

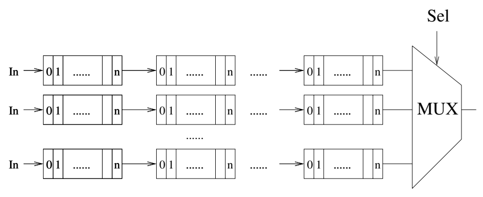

**Figure 10: Staged Design with Speculation**

see how our optimization affects verification performance for different number of stages. Again the assertion graphs are constructed based on the original assertion graph and reflect the staged design. The scale for this design and assertion graph is so complicated that the original AutoGSTE without optimization cannot finish within reasonable amount of time. We plot the time and memory usage data for model refinement with and without the precise-nodes-with-lifespans optimization in Figure 11. With the backward reasoning optimization, both refinement algorithms identified the correct fan-in of the FIFO. In this staged design, there is no control dependency between the stages and only the data is transferred after a dequeue signal, so we only need to mark the control signals of a FIFO precise during the period when the FIFO is working. In this way, the maximal number of precise nodes for different assertion edges is 5. In contrast, without the precise-nodes-with-lifespans optimization, we end up with marking 120 precise nodes for the 24-depth staged design. We see from Figure 11 that the precise-nodes-with-lifespan optimization reduce both the time and memory usages for the verification, especially the memory usage. Since we mark one precise node on one edge per iteration, this requires multiple iterations in order to reach the desired level of precision. As a result, the running time is a little more than we expected, but it is still better than the refinement with only backward reasoning.

## 5. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a suite of optimizations for improving efficiency of automatic abstraction refinement for GSTE. These optimizations are applied throughout the AutoGSTE refinement loop: counterexample analysis, model refinement, and spec refinement. Experimental evaluations have shown that each optimization is efficient for certain type of circuits. For the next step, we will investigate analysis algorithms that can match a given circuit to the right types of optimizations. We will also examine more efficient symbolic indexing for fine-grained model refinement.

## 6. ACKNOWLEDGMENT

This research is partially supported by grants from Intel Corporation. We thank the anonymous reviewers for their helpful feedback.

## 7. REFERENCES

[1] S. Adams, M. Bjork, T. Melham, and C. Seger. Automatic abstraction in symbolic trajectory evaluation. In *Proc. of FMCAD*, 2007.

[2] R. Bryant and C.-J. Seger. Digital circuit verification using partially-ordered state models. In *Twenty-Fourth International Symposium on Multiple-Valued Logic*, 1994.

[3] Y. Chen, Y. He, F. Xie, and J. Yang. Automatic abstraction refinement for generalized symbolic trajectory evaluation. In *Proc. of FMCAD*, 2007.

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

[5] 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), April 1995.

[6] J.-W. Roorda and K. Claessen. SAT-based assistance to abstraction refinement for symbolic trajectory evaluation. In *Proc. of CAV*, 2006.

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

[8] 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*, 24(9), 2005.

[9] R. Tzoref and O. Grumberg. Automatic refinement and vacuity detection for symbolic trajectory evaluation. In *CAV*, 2006.

[10] J. Yang and C.-J. H. Seger. Introduction to generalized symbolic trajectory evaluation. *IEEE Transaction on VLSI Systems*, 11(3), June 2003.