SAT Sweeping with Local Observability Don't-Cares

Qi Zhu Nathan Kitchen
University of California at Berkeley, CA, USA

1 Introduction

SAT sweeping is a method for simplifying an AND-INVERTER graph by systematically merging graph vertices from the inputs towards the outputs using a combination of structural hashing, simulation, and SAT queries [1]. Due to its robustness and efficiency, SAT sweeping provides a solid algorithm for Boolean reasoning in functional verification and logic synthesis. The original SAT-sweeping algorithm merges two vertices only if they are functionally equivalent. However, functional differences between vertices are not always observable at the outputs of a circuit. An extended algorithm, which we call *ODC SAT sweeping*, achieves greater graph simplification with moderately increased computational effort by exploiting observability along paths of bounded length [2].

1.1 Local Observability Don't-Cares

For a given AIG, we define the *observability* of a vertex $v \in V$, denoted by a predicate $obs(v)$, as follows:

$$obs(v) = \begin{cases} 1 & \textbf{if } v \in O \\ \bigvee_{v' \in FO(v)} \ (obs(v') \wedge f(other(v',v))) & \textbf{otherwise} \end{cases} \quad (1)$$

In other words, the value at the output of a vertex is not observable iff for each of its fanouts either that fanout vertex is itself not observable or the other input of it evaluates to 0 — thus blocking the logical path.

The concept of *k-bounded observability* or *local observability* is based on limiting the length of the paths being considered for observability, e.g., to reduce the effort for its computation. The k-bounded observability, $obs(v,k)$ is defined as:

$$obs(v,k) = \begin{cases} 1 & \textbf{if } v \in O \ \vee \ k = 0 \\ \bigvee_{v' \in FO(v)} \ (obs(v',k-1) \wedge & \\ \qquad f(other(v',v))) & \textbf{otherwise} \end{cases} \quad (2)$$

For example, for $k = 0$ a vertex is always observable, whereas $obs(v,1)$ considers only the possible blockings at the immediate fanouts of v.

Clearly, for every input assignment that results in $obs(v) = 0$ we can change the functionality of v without affecting the output functions of the AIG. The idea of using local observability in SAT sweeping is to exploit the fact that vertex u can be merged onto v if the functions of u and v are equal for all k-bounded observable input assignments, i.e., u and v have to be equal only if $obs(u,k) = 1$. More formally:

Theorem 1 *For a given k, vertex u can be merged onto vertex v if,*

$$(f(u) \Leftrightarrow f(v)) \vee \neg obs(u,k) = 1$$

Sketch of Proof: It is easy to show that the original circuit is equivalent to the circuit after merging u onto v by arguing that for every input assignment either $f(u) = f(v)$ or $obs(u,k) = 0$, that is, either their functions are identical or no path from u to any vertex within k levels is sensitizable. Therefore, no differences can propagate to any circuit output.

2 Algorithm

2.1 Overall Flow

In SAT sweeping, nodes are sorted into equivalence classes according to their simulation vectors, and the classes are refined as functional equivalences between nodes are verified. This technique is not applicable to ODC SAT sweeping because the mergeability condition is not an equivalence relation. Instead, it is necessary to find independently for each node the nodes onto which it may be merged. A dictionary of nodes, indexed by subsequences of simulation vectors, is used to find small sets of candidate replacement nodes quickly. In order to mask unobserved simulation bits, observability vectors are computed from the simulation vectors. The overall flow of ODC SAT sweeping is shown in Algorithm 1.

Algorithm 1 SAT SWEEPING WITH LOCAL ODCS

```
1:  {Given: AIG C = (V,E); bound k}
2:  initialize simulation vectors F(v) with random input values
3:  compute observability vectors OBS(v,k)
4:  insert each node v in dictionary D
5:  for all u ∈ V do
6:    update OBS(u,k)
7:    for all v ∈ SEARCH(D,u,OBS(u,k)) do
8:      if (F(u) ⇔ F(v)) ∨ OBS(u,k) = 1 {bitwise} then
9:        res := SAT-CHECK ((f(u) ⊕ f(v)) ∧ obs(u,k))
10:       if res = SAT then
11:         add SAT counterexample to vectors F(v)
12:       else if res = UNSAT then
13:         MERGE (u,v)
14:         update vectors F(v), D
15:         go to next u
```

If a vertex u is merged onto a vertex v in its transitive fanout, a cycle will be created in the AIG. Both the original SAT-sweeping algorithm and our extension avoid this case conservatively by comparing the levels of the vertices: u can be merged onto v only if $\text{LEVEL}(u) \geq \text{LEVEL}(v)$. This constraint can be exploited for efficiency by iterating over the vertices in level order. Only the vertices at the current level and below are searched for merging candidates. We included this refinement in our implementation, but it is omitted from Algorithm 1 for simplicity.

2.2 Use of Simulation Vectors

Let $F(v)$ denote the simulation vector computed for v by simulating the circuit with random input assignments. Then the observ-

ability vector $OBS(v)$ is computed as follows:

$$OBS(v,k) = \begin{cases} [111\ldots11] & \text{if } v \in O \vee k = 0 \\ \bigvee_{v' \in FO(v)} (OBS(v', k-1) \wedge & \\ \qquad F(other(v', v))) & \text{otherwise} \end{cases} \quad (3)$$

To implement the search dictionary efficiently, we use a binary trie. Each leaf node of the trie corresponds to a set of AIG vertices. Each internal node is associated with a branching bit index. For a trie T with branching bit index i, AIG vertices with $F_i(v) = 0$ are stored in the sub-trie T_0 on the 0-branch and vertices with $F_i(v) = 1$ are stored in the subtrie T_1 on the 1-branch. Note that vertices with different simulation vectors may be stored in the same leaf since the bits that differ may not be indexed by the branching bits.

To find candidate vertices for merging, we use a recursive search routine on the trie. At each level, a branch is selected according to the bit $F_i(v)$. However, the branching bit may not be observed, because it is masked by the observability vectors $OBS(v,k)$. In this case, it cannot be used to discriminate between branches to follow, so both branches are followed.

The number of leaves visited in a search is exponential in the number of don't-care bits among the branching bits. In order to minimize the number of leaves visited, we select the branching bit indices with a straightforward heuristic: For each index i, we compute the number of observed bits in the simulation vectors with index i, i.e., $\sum_v OBS_i(v,k)$, and use the bits with the largest sums. This heuristic does not take into account any correlation between bits. In preliminary experiments, we found that the extra computation required to consider correlation outweighed the benefit.

3 Implementation

We implemented ODC SAT sweeping for and/inverter graphs (AIGs) represented using the Ai package in OAGear.

3.1 Application Programming Interface

The API for our Ssw package consists of two main classes:

- OBSAIGRAPH is an observable AIG. Derived from the oagAi::Graph class, it adds the capability to mark nodes as observable.
- ODCSATSWEEPINGENGINE performs ODC SAT sweeping on an ObsAiGraph. Its public interface is shown in Figure 2.

The public interfaces of the OBSAIGRAPH and ODCSATSWEEPINGENGINE are shown in Figures 1 and 2, respectively. For simplicity, we omit the oagAi namespace qualifier here.

class OBSAIGRAPH – Public interface			
	OBSAIGRAPH	( )	
bool	ISOBSERVABLE	(Ref	*x*)
void	SETOBSERVABLE	(Ref	*x*)

Figure 1: Public interface of the ObsAiGraph class

An OBSAIGRAPH can be constructed easily from Verilog or OpenAccess designs by means of the QueryOcc and ReasoningEngine classes in the OAGear Func package. Once a graph is constructed, it is passed to an ODCSATSWEEPINGENGINE for sweeping. In some applications, such as bounded model checking, only a subset of the nodes in the graph should be swept. For this reason, the set of nodes to operate on can be explicitly specified. Likewise, the set of possible replacement nodes can be specified by the SETREPPOOL method.

class ODCSATSWEEPINGENGINE – Public interface			
ODCSATSWEEPINGENGINE		(ObsAiGraph	*graph*)
ODCSATSWEEPINGENGINE		(ObsAiGraph	*graph*,
		list<Ref>	*nodes*)
void	SETREPPOOL	(list<Ref>	*reps*)
void	RUN	(uint	*nObsLevels*)
bool	HASREP	(Ref	*x*)
Ref	GETREP	(Ref	*x*)
Params	*params*		

Figure 2: Public interface of the OdcSatSweepingEngine class

The RUN method performs ODC SAT sweeping for a given number of levels of local observability. Separating this method from the constructors enables the user to sweep the same nodes multiple times, e.g., with increasing values of *nObsLevels* in an iterative deepening scheme.

After ODC SAT sweeping is performed, the nodes that are replaced can be identified with the HASREP method. The GETREP method provides the replacement for a given node. These methods provide sufficient information for aggressive restructuring of the graph when desired. Within ODCSATSWEEPINGENGINE itself, merges are conservative, even when structural hashing is enabled: The fanout edges of merged nodes are redirected, and merges are propagated forward through the graph, but merged nodes are not detached from their inputs nor deallocated. As a result, external references into the graph are preserved. A user can simplify the graph more aggressively by detaching replaced nodes and running garbage collection.

The performance of ODC SAT sweeping can be tuned by adjusting several parameters in the ODCSATSWEEPINGENGINE, including the initial number of simulation bits, the maximum number of backtracks per SAT query, and the maximum number of nodes stored in any leaf of the trie.

3.2 Subcomponents

Our Ssw package uses MINISAT [3] for satisfiability checks. Although we created a separate package in OAGear for it, we use its interface directly rather than through a wrapper interface. Replacing it with another SAT solver would require modifications in multiple places in our code.

4 Results

We have tested our implementation of ODC SAT sweeping with several modules from the OpenCores repository (available at `http://www.opencores.org`), ranging in size from 200 to 50K AIG nodes. In our experiments, we found that ODC SAT sweeping can merge significantly more nodes than the original SAT-sweeping algorithm, and the runtime of the algorithm scaled well with circuit size. For more information, see [2].

We would like to point that the number of equivalent vertices typically has a strong effect on the power of equivalence checking techniques beyond SAT sweeping [1]. Equivalent vertices provide cutpoints that reduce the size of the subproblems, and thus the complexity. In addition, in bounded model checking the equivalences have a great effect on simplification in later time frames.

References

[1] A. Kuehlmann, "Dynamic transition relation simplification for bounded property checking," in *ICCAD 2004*, pp. 50–57.

[2] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli, "SAT sweeping with local observability don't-cares," in *DAC 2006*.

[3] N. Eén and N. Sörensson, "An extensible sat-solver [ver 1.2]."