SHIVA – A Fast Hybrid Constraint Solver for Circuits

Madhu K. Iyer
madiyer@ece.ucsb.edu

Ganapathy Parthasarathy
gpartha@ece.ucsb.edu

Kwang-Ting Cheng
timcheng@ece.ucsb.edu

Abstract

We present a novel hybrid constraint solving framework called SHIVA. Our approach makes efficient use of various levels of circuit abstraction automatically. We target CAD problems, such as formal verification and functional test generation. Our approach seamlessly combines sequential Boolean ATPG, and linear integer arithmetic solving techniques into an efficient solver. We use the strengths of each solver in different algebraic domains of the circuit. We describe the design and implementation of SATORI – a fast sequential justification engine based on state-of-the-art SAT and ATPG techniques. We use conflict-based learning in each time-frame and illegal state learning across time-frames. This enables both combinatorial and sequential back-jumping. We show results for SATORI versus a commercial ATPG engine and VIS [4] on ISCAS ’89 and ITC’99 benchmark circuits for an application to assertion checking. SHIVA is still under development and work is proceeding on completing full sequential search using SHIVA. We analyze SHIVA using some example circuits and discuss the results of these experiments.

1. Introduction

Sequential justification is the problem of finding an ordered sequence of input assignments to a sequential circuit, such that the desired objective is satisfied, or proving that no such sequence exists. Digital circuits can be represented as Boolean formulae and hence as a general SAT problem. This is well known to be NP-complete. The exponential decision spaces in sequential SAT make exhaustive search of the circuit state space computationally impractical for large sequential circuits.

One of the most important weaknesses of many search techniques that depend on a single computational model is their lack of robustness. Problem instances of seemingly comparable complexity require vastly different computational resources. The main problems for the major constraint solving techniques today are: a) State explosion for large state machines in state enumeration based methods, b) Time explosion for branch-and-bound methods like Automatic Test-Generation(ATPG) [2]. c) Problem size explosion for function representation based methods like Decision Diagrams [3]. However, each of these techniques work well on “well-behaved” examples. ATPG has been proven to work well on pure Boolean logic as compared to BDDs particularly for large, complex circuits. On the other hand, BDD-, or Hybrid DD-based symbolic simulation methods are very effective for large memories, register files etc. BDD-based methods have been used on arithmetic circuits with varying degrees of success, but they all suffer from the space complexity problem. Arithmetic approaches [1], are a natural way of handling structured data-paths, but they cannot be used on random control logic.

Our approach attempts to find when these techniques can be applied in an overall search-based strategy. We show that integrating orthogonal solvers can leverage their strengths across a wide range of problem instances.

Paper Outline The rest of this paper is organized as follows: In Section 2, we describe a prototype framework called SHIVA – a novel branch-and-bound hybrid constraint-solving algorithm, which tightly integrates a sequential Boolean search engine, constraint propagation engine, and a Presburger arithmetic solver – the Omega Library [6]. We present the basic architecture and algorithms used in SATORI in Section 3. In Section 4, we address the issues in integrating these key solvers, such that the strengths of each are used effectively. In Section 5, we describe our experiments to show proof-of-concept of our approach. We discuss the results of the experiments and, finally draw some conclusions from these experiments in Section 6.

2. SHIVA Hybrid Solver

SHIVA is a constraint solving framework for formal verification and multi-level ATPG of circuits. It is composed of a set of closely integrated solver engines, that work on a high-level circuit description composed of Boolean and arithmetic operators. The framework currently integrates an arithmetic solver and a sequential ATPG solver. These are explained in detail in Sections 2.1 and 2.2.

The hybrid solver dynamically partitions the search space into the domains of particular sub-solvers. The search space for a domain is based on the number of primary inputs to that domain. This includes the actual primary inputs and the inputs from the cut on the boundary between domains. For example, the control module has inputs from primary inputs, data-path and memory. A sub-search now consists of searching the space for a module with a specific solver. The results of a sub-search is used to bound the search space for the overall circuit as illustrated in Figure 1.

SHIVA conducts a branch-and-bound search in a transformed state space of the circuit. The branching decision variables derive from the interaction between control and data-path sections of the circuit. The transformed search space can be much smaller than original state space, since we use word-level abstraction.

The algorithm performs dynamic justification in the Boolean control logic and on the interface between control and data-path. We accomplish this by using an event-based mechanism for determining when the data-path should be evaluated, during the branch-and-bound process. Boolean implications in the controller are used to guide the search in the circuit space. The algorithm uses a combination of constraint propagation in the data-path and constraint solving using the Omega library [6] to find these solution states. In the next section, we describe SATORI – the sequential Boolean search engine.
3. SATORI

In this section, we describe SATORI, which is a fast sequential Boolean ATPG engine for the branch-and-bound solver. We use a reverse time processing model for objective and state justification. We store previously visited good circuit state and unjustifiable or illegal states, in order to limit the search. The ATPG engine incorporates conflict analysis based combinational learning and decision re-ordering. The chief components of SATORI are described below:

3.1. Implication Engine

The implication engine in SATORI operates on a SAT-style data-base of clauses for speed. The efficiency of implications is significantly higher in a CNF formulation than in a gate-level circuit. The main cost of the implication engine comes from evaluating when a gate is ready for implication. A clause-based implication engine using watched-literals [3] can do this very efficiently. However, a gate-level implementation of watched-literals is not as efficient since the update of the watched-literals is complicated by the multiple implication rules for a gate level circuit. Currently we use the CNF representation for efficient implications.

3.2. The Decision Engine

The decision engine makes decisions on primary inputs and state variables for a user-defined number of decisions using back-tracing. Back-tracing is a structural circuit traversal method that selects a gate from a restricted subset of gates called the justification frontier or J-frontier. The signals in the justification frontier satisfy the property that the original justification objective is satisfied iff every signal in the justification frontier is satisfied.

The choice of decision variable is guided by sequential controllability measures [2]. If a satisfying assignment cannot be found by the cut-off number of decisions, the decision strategy changes to a VSIDS [3]-style approach, which weights the decision variables based on the number of clauses in which a literal appears on an exponentially decaying scale over time. At this stage, all gates are considered as potential decision points. SATORI incorporates conflict-based learning [3] and back-jumping for efficiency in both the combinational and sequential search space. The interested reader is referred to [7] for more details of conflict-based learning and back-jumping.

Once a satisfying assignment for an objective in a time-frame is found, the assignments at the independent variables i.e., PIs and PPIs are used to form a conflict clause, which represents the state that is required to satisfy the objective in the given time-frame. This state clause is used to prune the sequential search space.

3.2.1 Assignment Reduction Algorithm

We now describe REDUCEASSIGMENTS, which is a heuristic based on list propagation from the set of assignments on state variables that satisfies the given state objective for a time-frame. The algorithm for REDUCEASSIGMENTS takes as inputs a topologically sorted circuit Ckt, an assignment vector A, and a set of objectives ob j; and produces a new assignment a, given by:

\[ a \subseteq A \text{ s.t. } f(a) \wedge \text{ob } j = 1, \text{iff } f(A) \wedge \text{ob } j = 1 \]  

(1)

Given a satisfying assignment A, there can exist several subsets a1, . . . , ak ∈ A, with different sets of state variables quantified out, and each of which satisfy the objective in the given time-frame. The algorithm uses a cost function based on the number of PPIs required to satisfy the objective, in order to minimize the size of a. REDUCEASSIGMENTS is conceptually equivalent to existentially quantifying out the state variables in the circuit one by one. In practice, the algorithm parallelizes this by propagating sets of necessary assignments. If there is no initial state specified, all the state variables in an assignment A can be quantified out only if a does not have any necessary state-variant assignments.

If the value on the gate is a controlling value, then the PI/PPI that uniquely sets the gate value is chosen irrespective of whether it is a PI or a PPI. On the other hand, if there exists a choice between two controlling input variables, then we pick the first assigned PI with a controlling value in the lists that has been propagated to the inputs of the gate. If the gate is at a non-controlling value, then the lists at the inputs of the gate are merged at the output of the gate. Gates that are an unknown or X value are ignored. The final list at the justification objective gives the desired minimal assignment set.

3.3. State Clause learning

SATORI uses conflict clauses to record the state requirements for a given time-frame. Given an assignment A, that satisfies the objectives for time-frame ti, a set of necessary assignments, a = REDUCEASSIGMENTS(A), is determined and a clause corresponding to a is stored in a state cache. If the problem is satisfiable in a given time-frame, SATORI does a fast lookup to check if the state requirement covers a previously visited state. The results of the lookup are used to determine if:

- the decision procedure should terminate, or
- backtrack to an earlier time-frame, or
- backtrack in the current time-frame and continue the search.

Note that we keep only one copy of the circuit in memory at a time. If we continue search in time, the formula fT1 for the next time-frame is generated from the current fT, and the learned state clauses \( T, I \) as follows:

\[ f_{T+1} \equiv f_t \wedge \{ t_i \in T \} \wedge \{ i \in I \} \]  

(2)

In practice, this amounts to adding a few clauses to the current net-list. Note that identified Illegal states, which cannot be justified further back in time, are avoided implicitly from the formulation fT1.

3.3.1 Backtrack Clauses

If we backtrack to an earlier time-frame, we create a back-track clause bi corresponding to the literals in the last state clause learned in that time-frame and the value assignments on the PIs. Hence fT ⇒ fT bi. This constraint that the last satisfying assignment cannot be repeated. A typical DLL back-tracking procedure will backtrack to the last decision made in the decision tree in a given time-frame and continue search. However, with the addition of back-tracking clauses, all decisions that lead to the same state-cube that are already explored in the same time-frame are avoided or detected before a new state-cube is found, with a minimum of conflicts in the decision procedure.

3.3.2 Loop Detection

SATORI performs DFS-search in the sequential state space. Hence, it is necessary to check for revisited states in order to avoid getting stuck in sequential loops.

We check for loops by explicitly checking a new state against the set of visited states. This is done by comparing the literals in the new state versus the literals in the states in the cache. This check is actually quite fast, since we hash the states based on the literal values in a hash-table. A unique hash is generated for each clause and is used to compare or retrieve visited states. We also check for Boolean covers in the visited states. If a state assignment is covered by an illegal state, then we can backtrack to the earliest time-frame where we detected the illegal state.

Additional optimizations include static illegal state identification, where state-cubes which are reachable only from illegal states are classified as illegal, and identification of legal state-cubes where known detection sequences exist. Note that this is different from extracting a complete state-transition graph (STG), which can be quite expensive.

SATORI terminates if the sequential search returns UNSAT or if the state requirements in a time-frame are covered by the initial state.

4. Arithmetic Solver

The most natural domain in digital systems for representing data-path arithmetic operations is the integer domain. Hence, we use an constraint propagation and integer arithmetic solver to solve for constraints on data-path blocks.

We use a solver implemented with the Omega library [8], which is based on Presburger arithmetic. The Omega library is a set of C++ functions for manipulating integer tuple relations and sets. Integer tuple relations are mappings from an input integer set to an output integer set. The omega library can represent and solve sets and relations that can be
described by Presburger formulae. Presburger formulae are those that can be constructed by combining affine constraints on integer variables using logical operations such as $\land$, $\lor$, $\neg$, and quantifiers $\forall$ and $\exists$.

Bit-vector arithmetic, can easily be represented as Presburger formulae. This includes $\text{modulo-2 arithmetic}$, which is required in some special cases for Bit-vector arithmetic. The worst case complexity for algorithms verifying formulae on Presburger arithmetic is exponential. However, this is seen only when using variables with unbounded range and with modulo-2 arithmetic.

4.1. Solver Scheduling Algorithm

We use a branch-and-bound algorithm in the hybrid solver. The search frontier or Justification Frontier for the hybrid solver has 2 components – Boolean gates and arithmetic primitives. Arithmetic primitives comprise operators over linear bit-vector arithmetic i.e., $\{+, -, =, \neq, \leq, \geq\}$, and are stored as a conjunction of presburger relations.

Control and data-path in designs interact at specific nodes called interface points. These interface points exist in these designs due to Bus splitting, Bit Concatenation and Word-level primitives with Boolean outputs such as comparators and muxes. They represent the combined search frontier. Boolean assignments on the interface points are propagated as constraints through the word-level relations. If the constraint propagation does not result in a satisfying solution, then the Omega Library is used to check satisfiability for the constraints imposed by the branch-and-bound. This result is interpreted by the search algorithm, to decide whether to stop or continue searching. In the next section, we describe how Presburger arithmetic is used to propagate constraints efficiently in the data-path.

4.2. Hybrid Search Pruning

In our current implementation, we support only constraint propagation over the data-path i.e., we do not make any decisions inside the data-path, since the arithmetic solver can find all solutions to the constraints in the data-path simultaneously, in polynomial time for linear data-paths.

A BFS traversal is performed from all relations that are affected by a Boolean assignment on an interface point. These relations are then incrementally evaluated by equating data-path input and output variables. A Conflict condition is determined when the constraints newly added to the relation are unsatisfiable. If this is the case, we try to find the smallest set of sufficient Boolean assignments that caused the conflict. We then create a clause called an interface clause to these assignments, which prevents this conflict space from being revisited. These clauses implicitly capture the state of the branch-and-bound and are sufficient to ensure completeness of the search.

5. Experiments

In this section, we present some experimental results on SATORI and SHIVA. The experiments were run on an Intel Pentium-4 machine with 1 Gigabyte of RD-DRAM running Linux and a Sun UltraSpace-3d machine running Solaris 8.0. The results were scaled using an independent set of experiments so that the run-times can be directly compared.

5.1. SATORI

One interesting application of SATORI is in assertion checking on sequential circuits. Typically, we have a trade-off between BDD-based approaches and search-based approaches like Bounded Model Checking. Since SATORI is a complete search engine, we contrast it with VIS, and a commercial ATPG tool on assertion checking on some large examples in the ITC’99 and ISCAS’89 benchmark suites. We ran 2 assertions on each example, which were:

1. $AG(p_1)$, where $p_1 = (s_1 = 0) \lor (s_2 = 0) \ldots \lor (s_n = 0), s_i \in S$, where $S$ is the set of state variables in the circuit.
2. $AG(p_2)$, where $p_2 = (s_1 = 1) \land (s_2 = 1) \ldots \land (s_n = 1), s_i \in S$.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
Ckt & SATORI & Rslt$^*$ & Comm. ATPG & Rslt & VIS & Rslt \\
\hline
b14 & 0.07 & T/T & 3.3 & T/T & >1hr & A/A \\
b17 & 3.5 & T/T & 9.8 & T/T & >1hr & A/A \\
b20 & 0.15 & T/T & 3.1 & T/T & >1hr & A/A \\
b21 & 0.16 & T/T & 3.2 & T/T & >1hr & A/A \\
b22 & 0.23 & T/T & 4.9 & T/T & >1hr & A/A \\
s13207 & 1.2 & T/F & 2.8 & T/F & >1hr & A/A \\
s15850 & 0.15 & T/F & 2.6 & T/F & >1hr & A/A \\
s35932 & 119.5 & T/T & >1hr & A/A & – & – \\
s38417 & 0.77 & T/F & 8.6 & T/F & >1hr & A/A \\
s38584 & 0.41 & T/F & 7.0 & T/F & >1hr & A/A \\
s3578 & 0.02 & T/T & 1.1 & T/T & >1hr & A/A \\
s820 & 0.09 & F/F & 5.7 & F/F & 0.1 & F/F \\
s832 & 0.09 & F/F & 5.6 & F/F & 0.1 & F/F \\
s838 & 0 & F/T & 0.5 & F/T & 0.2 & F/T \\
s9234 & 0.04 & F/T & 1.4 & T/F & >1hr & A/A \\
\hline
\end{tabular}
\caption{Assertion Checking Experiments}
\end{table}

Column 2 and 3 show the run-times and result of SATORI, 4 and 5 for a commercial ATPG tool, and 6 and 7 for VIS. Note that most of these assertions required significant search in the sequential space. VIS aborted on most of the big benchmarks after about 3600 secs. The peak memory usage on these cases was close to 800 Mb. The memory consumed by SATORI and the commercial engine did not exceed 300MB.

In the cases where VIS managed to finish building the BDD-based images, it outperformed the commercial engine by 2x to 50x. However, SATORI finished all the test-cases in reasonable time, including cases where both the commercial engine and VIS failed. SATORI outperformed commercial ATPG by 10x to 50x and finished on 1 case where ATPG aborted after 3600 secs.

5.2. SHIVA – Current Status

SHIVA is still a work in progress. SHIVA currently handles time-frame expanded combinational circuits, and we are implementing a full sequential approach. SHIVA has several similarities to SVC [11], since our decision procedure combines Boolean and bit-vector integer arithmetic. We differ from SVC primarily on how we propagate information between the two domains and how efficiently we partition a given logic formula for propositional satisfiability. In this section, we analyze some of the problems with using time-frame expansion as compared to full sequential search. We also illustrate some problems with search bounding in the context of SHIVA.

A justification objective of 1 was set at a primary output in each of the ITC benchmarks shown in Table 2 and 3. Each circuit is time-frame expanded for 2.5, 10, 15 time-frames. The column LACs shows the total number of Linear Arithmetic Constraints solved, column Calls shows the number of arithmetic satisfiability checks made, and column Contfl. shows the number of conflicts encountered in the search. As the number of time-frames is increased, the number of LACs to be solved increases which increases ILP run time. Also, the number of solver calls made increase when the value propagation between various solution domains is not complete.

Another parameter of importance not captured in the above is the time taken for dynamic maintenance of the constraint set. We show CPU times between 2 versions of SHIVA in Table 3. modSHIVA re-uses the previously created constraint set while SHIVA does not. As we can see, reusing the constraints prevents the solver from re-exploring the same states again and hence translates to a significant performance differential. This effect is emphasized due to the relatively high cost of each arithmetic solver call. The size of the LACs can be reduced by using a fully sequential algorithm that limits the logic to a single combinational time-frame and allows learning of powerful symbolic state relations as shown in Figure 5.

Reducing the number of arithmetic solver calls is addressed by generating tight conflict clauses as shown in Figure 2. The values in circles show the current Boolean assignment at the interface points.
Table 2: Shiva performance data for sample circuits

<table>
<thead>
<tr>
<th>Circuit</th>
<th>LACs</th>
<th>Calls</th>
<th>Confli.</th>
<th>LACs</th>
<th>Calls</th>
<th>Confli.</th>
<th>LACs</th>
<th>Calls</th>
<th>Confli.</th>
<th>LACs</th>
<th>Calls</th>
<th>Confli.</th>
</tr>
</thead>
<tbody>
<tr>
<td>b01</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>12788</td>
<td>1960</td>
<td>217</td>
<td>300820</td>
<td>41694</td>
<td>1839</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>b02</td>
<td>12</td>
<td>2</td>
<td>1</td>
<td>35196</td>
<td>5258</td>
<td>240</td>
<td>285382</td>
<td>41323</td>
<td>792</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>b04</td>
<td>27</td>
<td>2</td>
<td>1</td>
<td>518</td>
<td>63</td>
<td>11</td>
<td>1492</td>
<td>192</td>
<td>18</td>
<td>4154</td>
<td>537</td>
<td>42</td>
</tr>
<tr>
<td>b11</td>
<td>7423</td>
<td>1126</td>
<td>146</td>
<td>43914</td>
<td>5896</td>
<td>244</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Table 3: Performance improvement with Constraint Re-Use in Shiva

<table>
<thead>
<tr>
<th>Circuit</th>
<th>SHIVA</th>
<th>modSHIVA</th>
<th>SHIVA</th>
<th>modSHIVA</th>
<th>SHIVA</th>
<th>modSHIVA</th>
</tr>
</thead>
<tbody>
<tr>
<td>b01</td>
<td>0</td>
<td>0</td>
<td>1.7</td>
<td>0.28</td>
<td>395.78</td>
<td>11.12</td>
</tr>
<tr>
<td>b02</td>
<td>0</td>
<td>0</td>
<td>24.35</td>
<td>1.08</td>
<td>&gt;1200</td>
<td>28.9</td>
</tr>
<tr>
<td>b04</td>
<td>0</td>
<td>0</td>
<td>0.06</td>
<td>0.02</td>
<td>1.14</td>
<td>0.08</td>
</tr>
<tr>
<td>b11</td>
<td>1.19</td>
<td>0.2</td>
<td>42.48</td>
<td>1.8</td>
<td>&gt;1200</td>
<td>&gt;1200</td>
</tr>
</tbody>
</table>

In this example, the assignments of \{1,0,1\} on lines \{g,h,i\} are unsatisfiable. A loose conflict clause would be \((g + h + i + s0 + s1)\). The tightest conflict clause is \((g' + i')\), which avoids the entire conflict space due to these assignments. This translates to a lower number of solver calls and corresponding performance improvement.

Figure 2. Tight Interface Clauses

Enhanced Learning in SHIVA Figure 3 shows an example of efficient word-level state pruning. The objective is to justify line \(e\) to 1. The assignment of lines \(\{d,b\}\) to \(\{1,0\}\) is unsatisfiable. SHIVA is much more efficient at finding this than a Boolean engine. In time-frame-0 the following constraints can be satisfied: \((E = 20) \land (\overline{REG1} + \overline{REG2} = 20)\) and \((\overline{REG3} < 255)\). \{(C + D = 20) \land (B + 8 = C) \land (A + 5 = D) \land (A < 255)\} can be satisfied in timeframe-1. In timeframe-2 the constraint \(REG1 \leq 7\) is unsatisfiable. SATORTI learns 4096 Boolean state constraints and 7688 Boolean combinational constraints on a bit-level version of this example. Shiva learns 2 state constraints \((REG1 + REG2 = 20) \land (\overline{REG3} \leq 255)\) and \((\overline{REG1} \leq 7)\), and 1 interface constraint \(c\). We will present results on the full sequential SHIVA by the time of the presentation.

6. Conclusion

We described a new framework for verification and test based on a hybrid constraint solving approach. We showed the effectiveness of using both Boolean justification and arithmetic solving techniques on functional properties, for both algorithmic circuits and data-path intensive circuits. Though SHIVA is currently stable and shows reasonably good performance, there are still issues in various stages of resolution, which need to be incorporated before they can be finalized. The major issues are a) combining implications in Boolean and arithmetic domains and b) making arithmetic solver fully sequential.

We have described how we integrate path-based justification with conflict-based learning and SAT techniques in SATORTI for efficient sequential search. We have presented results on SATORTI versus VIS, which show that SATORTI can be more efficient than BDD-based image computation when exploring huge state spaces.

References