Simulation-Based Bug Trace Minimization With BMC-Based Refinement

Kai-Hui Chang, *Student Member, IEEE,* Valeria Bertacco, *Member, IEEE,*
and Igor L. Markov, *Senior Member, IEEE*

Abstract—**Finding the cause of a bug can be one of the most time-consuming activities in design verification. This is particularly true in the case of bugs discovered in the context of a random-simulation-based methodology, where bug traces, or counterexamples, may be several hundred thousand cycles long. In this paper, BUg TRAce MINimization (Butramin), which is a bug trace minimizer, is proposed. Butramin considers a bug trace produced by a random simulator or semiformal verification software and produces an equivalent trace of shorter length. Butramin applies a range of minimization techniques, deploying both simulation-based and formal methods, with the objective of producing highly reduced traces that still expose the original bug. Butramin was evaluated on a range of designs, including the publicly available picoJava microprocessor, and bug traces up to one million cycles long. Experiments show that in most cases, Butramin is able to reduce traces to a very small fraction of their initial sizes, in terms of cycle length and signals involved. The minimized traces can greatly facilitate bug analysis and reduce regression runtime.**

Index Terms—**Bug trace minimization (Butramin), counterexample minimization, error diagnosis, verification.**

I. INTRODUCTION

MODERN integrated circuit design has reached unparalleled levels of size and overall complexity. In this context, design verification has become a pivotal aspect of electronic design automation. In fact, various estimates indicate that functional errors are still responsible for 40% of failures at the first tape-out and that verification accounts for two-thirds of the design cycle and effort [2], [17]. Resolving design bugs in the early development stages is, at the same time, a sophisticated and time-consuming activity, as well as a crucial task for the project development and for the success of a design team. With mask costs approaching a million dollars per set, being able to find and *fix* bugs before the first tape-out offers a significant economic advantage.

Among the techniques and methodologies available for functional verification, simulation-based verification is prevalent in the industry because of its linear and predictable complexity and its flexibility in being applied, in some form, to any design. A common methodology in this context is *random simulation*. Random simulation involves connecting a logic simulator with stimuli coming from a constraint-based random generator, that is, an engine that can automatically produce random legal input

for the design at a very high rate based on a set of rules (or constraints) derived from the specification document. In order to detect bugs, assertion statements, or checkers, are embedded in the design and continuously monitor the simulated activity for anomalies. When a bug is detected, the simulation trace leading to it is stored and can be replayed at later times to analyze the conditions that led to the failure. Because of the randomized nature of this methodology, and because it is usually applied in late design stages (i.e., when simple bugs have already been flushed out), it is very common for the bug traces generated to be very complex, often as much as hundreds of thousands of cycles long.

Another family of techniques attracting increasing attention from the industry is that of semiformal verification. These tools combine a mix of formal and simulation-based techniques with the goal of producing high-coverage verification results on complex designs. These results may entail generating tests that cover a specific state configuration, proving or disproving a property (or a checker), etc. Pure formal verification techniques, such as symbolic simulation, Bounded Model Checking (BMC), and reachability analysis [3], [13], would be ideal to generate compact high-coverage tests, such as a minimum-length counterexample that disproves a property. Unfortunately, they do not scale well and can only be applied to very small designs.

In the more general context of semiformal techniques [1], [10], [12], heuristics and randomized exploration allow designers to obtain high-coverage results on designs of medium and large complexity, but they must sacrifice the generation of minimum-length counterexamples. While these tools are a promising direction in terms of high-quality verification, little concern has been given to the reduction of the complexity of the bug traces generated. The result is that, once a bug is found, a copious amount of effort is dedicated to tracking it back to its cause: either an incorrect design implementation or an erroneous property definition.

Current trends attempt to generate high-quality results with less effort on the part of the verification engineer, such as the previously mentioned random simulation and semiformal verification techniques. These two techniques are more attractive when compared to a traditional direct-test simulation approach, which can be extremely demanding, requiring the manual development of entire sets of specific test stimuli. However, these techniques tend to generate extremely long and complex bug traces, exasperating the debugging phase of verification.

Contributions: We address the problem of debugging complex bug traces by proposing a technique for trace minimization

called BUg TRAce MINimization (Butramin). The objective of Butramin is to consider a bug trace and the checker (or property) that it triggers and seek a much shorter and simpler trace to falsify the same property. Previous work in this area has been mostly centered on using formal techniques to simplify a property's counterexample [5], [18]. Simulation-based techniques to address this problem have been proposed in [4], which is a preliminary version of this paper. In a separate context, the problem of trace minimization has also been addressed in software verification [8], [11].

Butramin simplifies a trace by iteratively eliminating redundant portions of the trace. For instance, it checks if there are redundant sequential steps or sequential loops that can be removed. It also checks if combinational input events in a bug trace are redundant. For each candidate, a simplified trace is resimulated to check if it still exposes the original bug. When this mechanism is exhausted, Butramin further simplifies a trace by using X-value simulation to evaluate which input signals are essential in exposing a bug. Finally, a SATisfiability (SAT)-based fixed-window bounded model checker seeks additional "shortcuts" in the reduced and simplified trace. Our approach to trace minimization is novel in the following aspects.

1) It iteratively simplifies the trace by targeting the length (total number of clock cycles) as well as the number of input events of the trace.
2) It combines simulation and formal techniques, which exploits the performance of logic simulation as far as possible, and only applies formal techniques to a greatly reduced trace, requiring a much simpler analysis.
3) It is capable of classifying input variable assignments as essential or nonessential by marking a nonessential assignment with an X value in three-value simulation.
4) Experimental results show that Butramin can greatly simplify counterexamples generated by semiformal and constrained random verification tools down to a small fraction of their original sizes, and it produces consistent results across a range of design sizes and characteristics. The compact traces lead to a much easier interpretation of the activity causing the bug.

In developing Butramin, we gave top consideration to the quality of the results, since the engineering time saved by the latter well outweighs the execution time of the software. We envision a deployment scenario where Butramin is run overnight to prepare simplified traces to be analyzed and found that all of our execution times are well within this limit. Within this context, we additionally evaluated the potential of Butramin in minimizing high-coverage regression traces, that is, traces that visit multiple coverage goals. We found that even in this scenario, Butramin was capable of exposing a lot of minimization potential.

The remainder of this paper is organized as follows: Section II describes relevant previous work on bug trace minimization for random simulation and BMC. Section III analyzes the source of redundancy in bug traces and possible ways to identify and remove them. Section IV presents our new bug trace minimization technique that relies on logic simulation and describes the BMC-based search for counterexample shortcuts.
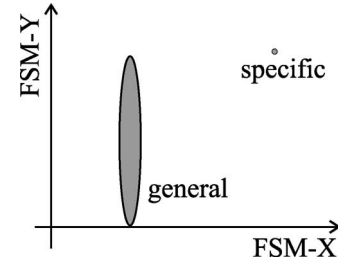

Fig. 1. Illustration of two types of bugs based on whether one or many states expose a given bug. The x axis represents FSM-X, and the y axis represents FSM-Y. A specific bug configuration contains only one state, whereas a general bug configuration contains many states.

Sections V and VI discuss algorithmic aspects of Butramin and experimental results. Finally, Section VII summarizes the contributions and concludes the paper.

II. BACKGROUND AND PREVIOUS WORK

Research on minimizing property counterexamples or, more generally, bug traces, has been pursued both in the context of hardware and software verification. In hardware verification, existing solutions typically minimize traces generated by BMC. Before discussing these techniques, we give some preliminary background and provide a brief overview of the BMC methodology.

A. Anatomy of a Bug Trace

A *bug state* is an undesirable state that exposes a bug in the design. Depending on the nature of the bug, it can be exposed by a unique state (a specific bug configuration) or any one of several states (a general bug configuration), as shown in Fig. 1. In the figure, suppose that the x axis represents one state machine called FSM-X (Finite State Machine X) and the y axis represents another machine called FSM-Y. If a bug occurs only when a specific state in FSM-X and a specific state in FSM-Y appear simultaneously, then the bug configuration will be a very specific single point. On the other hand, if the bug is only related to a specific state in FSM-X but is independent of FSM-Y, then the bug configuration will be all states on the vertical line intersecting the one state in FSM-X. In this case, the bug configuration is very broad.

Given a sequential circuit and an initial state, a *bug trace* is a sequence of test vectors that exposes a bug, i.e., causes the circuit to assume one of the bug states. The *length* of the trace is the number of cycles from the initial state to the bug state, and an *input event* is a change of an input signal at a specific clock cycle of the trace. One input event is considered to affect only a single input bit. An *input variable assignment* is a value assignment to an input signal at a specific cycle. The term "input variable assignment" is used in the literature when traces are modeled as sequences of symbolic variable assignments at the design's inputs. The number of input variable assignments in a trace is the product of the number of cycles and the number of inputs. A *checker signal* is a signal used to detect a violation of a property, that is, if the signal changes to a specific value, then the property monitored by the checker is violated, and a bug is found. The objective of *bug trace minimization* is to
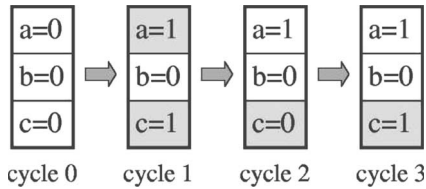
Fig. 2.   Bug trace example. The boxes represent input variable assignments to the circuit at each cycle; shaded boxes represent input events. This trace has three cycles, four input events, and 12 input variable assignments.
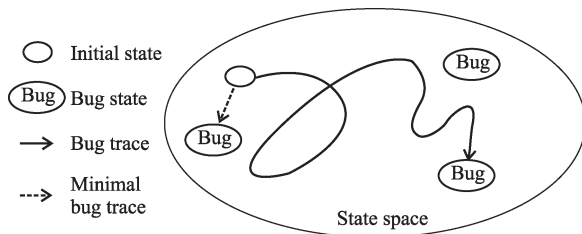

Fig. 3.   Another view of a bug trace. Three bug states are shown. Formal methods often find the minimal length bug trace, whereas semiformal and constrained random techniques often generate longer traces.

```
1   SAT-BMC(circuit, property, maxK) {
2       p̄=CNF(not(property));
3       for k=1 to maxK do {
4           C = CNF ( unroll(circuit, k) );
5           I = C ∧ p̄; //SAT instance
6           if (I is satisfiable)
7               return (SAT solution);
8       }
9   }
```

Fig. 4.   BMC pseudocode.

reduce the number of input events and cycles in a trace while still detecting the checker violation.

Example 1: Consider a circuit with three inputs a, b, and c, which are initially set to zero. Suppose that a bug trace is available where a and c are assigned to 1 at cycle 1. At cycle 2, c is changed to 0, and it is changed back to 1 at cycle 3, after which a checker detects a violation. In this situation, we count four input events, 12 input variable assignments, and three cycles for our bug trace. The example trace is illustrated in Fig. 2.

Another view of a bug trace is a path in the state space from the initial state to the bug state, as shown in Fig. 3. By construction, formal methods can often find the minimal length bug trace as shown in the dotted line. Therefore, we focus our minimization on semiformal and constrained random traces only. However, if Butramin is applied to a trace obtained with a formal technique, it may still be possible to reduce the number of input events and variable assignments.

B. BMC Overview

BMC [3] is a formal method that can prove or disprove properties of bounded length in a design, frequently using SAT solving techniques to achieve this goal. A high-level flow of the algorithm is given in Fig. 4. The central idea of BMC is to "unroll" a given sequential circuit k times to generate a combinational circuit that has behavior equivalent to k clock cycles of the original circuit. In the process of unrolling, the

circuit's memory elements are eliminated, and the signals that feed them at cycle i are connected directly to the memory elements' output signals at cycle $i - 1$. In Conjunctive Normal Form (CNF)-based SAT, the resulting combinational circuit is converted to a CNF formula C. The property to be proved is also complemented and converted to CNF form $\bar{p}$. These two formulas are conjoint, and the resulting SAT instance I is fed into a SAT solver. If a satisfiable assignment is found for I, then the assignment describes a counterexample that falsifies the (bounded) property; otherwise, the property holds true.

C. Known Techniques in Hardware Verification

Traditionally, a counterexample generated by BMC reports the input variable assignments for each clock cycle and for each input line of the design. However, it is possible, and common, that only a portion of these assignments are required to falsify the property. Several techniques that attempt to minimize the trace complexity have been recently proposed, for instance, Ravi and Somenzi [18]. To this end, they propose two techniques, namely: 1) Brute-Force Lifting (BFL), which attempts to eliminate one variable assignment at a time, and 2) an improved variant that eliminates variables in such a way so as to highlight the primary events that led to the property falsification. The basic idea of BFL is to consider the "free" variables of the bug trace, that is, all input variable assignments in every cycle. For each free variable v, BFL constructs a SAT instance $SAT(v)$ to determine if v can prevent the counterexample. If that is not the case, then v is irrelevant to the counterexample and can be eliminated. Because this technique minimizes BMC-derived traces, its focus is only on reducing the number of assignments to the circuit's input signals. Moreover, each single assignment elimination requires solving a distinct SAT problem, which may be computationally difficult. More recent work [19] further improves the performance of BFL by attempting the elimination of sets of variables simultaneously. Our technique for removing individual variable assignments is similar to BFL as it seeks to remove an assignment by evaluating a trace obtained with the opposite assignment. However, we apply this technique to longer traces obtained with semiformal methods, and we perform testing via resimulation.

Another technique applied to model checking solutions is by Gastin *et al.* [8]. Here, the counterexample is converted to a *Büchi automaton*, and a depth-first search algorithm is used to find a minimal bug trace. Minimization of counterexamples is also addressed in [14], where the distinction between control and data signals is exploited in attempting to eliminate data signals first from the counterexample.

All of these techniques focus on reducing the number of input variable assignments to disprove the property. Because the counterexample is obtained through a formal model checker, the number of cycles in the bug trace is minimal by construction. Butramin's approach considers a more general context where bug traces can be generated by simulation or semiformal verification software, attacking much more complex designs than BMC-based techniques. Therefore: 1) traces are, in general, orders of magnitude longer than the ones generated by BMC, and 2) there is much potential for reducing the

trace in terms of number of clock cycles, as our experimental results indicate. On the downside, the use of simulation-based techniques does not guarantee that the results obtained are of minimal length. As the experimental results in Section VI indicate, our heuristics provide in practice optimal results for most benchmarks.

Aside from minimization of bug traces generated using formal methods, techniques that generate traces by random simulation have also been explored in the context of hardware verification. One such technique is by Chen and Chen [5] and proceeds in two phases. The first phase identifies all the distinct states of the counterexample trace. The second phase represents the trace as a state graph, applies one step of forward state traversal [6] to each of the individual states, and adds transition edges to the graph based on it. Dijkstra's shortest path algorithm is applied to the final graph obtained. This approach, while very effective in minimizing the trace length (the number of clock cycles in the trace): 1) does not consider elimination of input variable assignments and 2) makes heavy use of formal state-traversal techniques, which are notoriously expensive computationally and can usually be applied only to small-size designs, as indicated also by the experimental results in [5].

D. Techniques in Software Verification

The problem of trace minimization has been a focus of research also in the software verification domain. Software bug traces are characterized by involving a very large number of variables and very long sequences of instructions. The delta debugging algorithm [11] is fairly popular in the software world. It simplifies a complex software trace by extracting the portion of the trace that is relevant to exposing the bug. Their approach is based exclusively on resimulation-based exploration, and it attacks the problem by partitioning the trace (which, in this case, is a sequence of instructions) and checking if any of the components can still expose the bug. The algorithm was able to greatly reduce bug traces in Mozilla, which is a popular web browser. A recent contribution that draws upon counterexamples found by model checking is by Groce and Kroening [9]. Their solution focuses on minimizing a trace with respect to the primitive constructs available in the language used to describe the hardware or software system and on trying to highlight the causes of the error in the counterexample to produce a simplified trace that is more understandable by a software designer.

III. ANALYSIS OF BUG TRACES

In this section, we analyze the characteristics of bug traces generated using random simulation, point out the origins of redundancy in these traces, and propose how redundancy can be removed. In general, redundancy exists because some portions of the bug trace may be unrelated to the bug, there may be loops or shortcuts in the bug trace, or there may be an alternative and shorter path to the bug. Two examples are given below to illustrate the idea, while the following subsections provide a detailed analysis.

Example 2: In Intel's first-generation Pentium processor, there was a bug in the floating point unit, which affected the
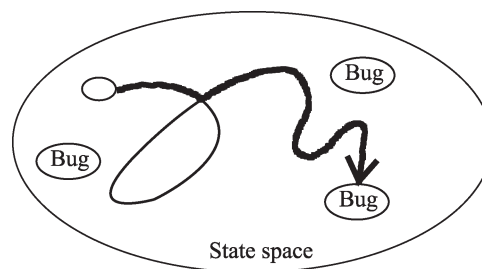

Fig. 5. Bug trace may contain sequential loops, which can be eliminated to obtain an equivalent but more compact trace.
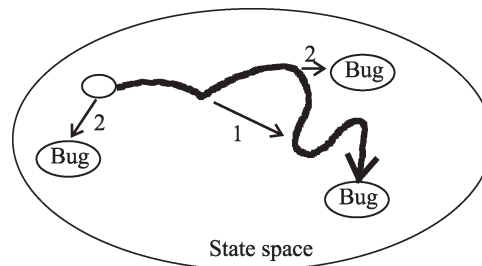

Fig. 6. Arrow 1 shows a shortcut between two states on the bug trace. Arrows marked "2" show paths to easier-to-reach bug states in the same bug configuration, which violates the same property.

fdiv instruction. This bug occurred when fdiv was used with a specific set of operands. If there had been a checker testing for the correctness of the fdiv operation during the simulation-based verification of the processor, it is very probable that a bug trace exposing this problem could have been many cycles long. However, only a small portion of the random program would have been useful to expose the fdiv bug, while the majority of the other instructions could have been eliminated. The redundancy of the bug trace comes from the cycles spent testing other portions of the design, which are unrelated to the flawed unit and can, thus, be removed.

Example 3: Suppose that the design under test is a First-In–First-Out (FIFO) unit and a bug occurs every time the FIFO is full. Furthermore, assume that there is a pseudorandom bug trace containing both read and write operations until the trace reaches the "FIFO full" state. Obviously, cycles that read data from the FIFO can be removed because they create state transitions that bring the trace away from the bug configuration instead of closer to it.

A. Making Traces Shorter

In general, a trace can be made shorter if any of the following situations arise: 1) it contains loops; 2) there are alternative paths (shortcuts) between two design states; and 3) there is another state that exposes the same bug and can be reached earlier.

The first situation is depicted schematically in Fig. 5. In random simulation, a state may be visited more than once, and such repetitive states will form loops in the bug trace. Identifying such loops and removing them can reduce the length of the bug trace.

In the second case, there may be a shortcut between two states as indicated by arrow 1 in Fig. 6, which means that an alternative path may exist from a state to another state

using fewer cycles. Such situations may arise in random traces frequently because constrained random simulation often selects transitions arbitrarily, and it is possible that longer paths are generated in place of shorter ones.

The third condition occurs when multiple design states that expose the same bug exist, and some of them can be reached in fewer steps compared to the original one, as shown by arrows marked "2" in Fig. 6. If a path to those states can be found, then it is possible to replace the original one.

A heuristic approach that can be easily devised to search for alternative shorter traces is based on generating perturbations on a given trace. A bug trace can be perturbed locally or globally to find shortcuts or a path to an alternative bug state. In a *local perturbation*, cycles or input events are added or removed from an original trace. As mentioned previously, random simulation selects state transitions in a pseudorandom fashion. By local perturbation, alternative transitions can be explored, and shorter paths to a trace state or to another state exposing the bug may be found. In a *global perturbation*, a completely new trace is generated and used to substitute the original one if it is shorter. One reason why perturbation has the potential to work effectively on random traces is that a pseudorandom search tends to do a lot of local exploration compared to a formal trace that progresses directly to a bug. Because of this, opportunities of shortcuts within a trace abound.

B. Making Traces Simpler

After all redundant cycles are removed, many input events may still be left. For example, if a circuit has 100 inputs and a bug trace is 100 cycles long, there are 10 000 input variable assignments in the trace. However, not all assignments are relevant to expose the bug. Moreover, redundant events increase the complexity of interpreting the trace in the debugging phase. Therefore, it is important to identify and remove such redundancy.

We envision two ways of simplifying the input assignments in a trace, namely: 1) by removing input events and 2) by eliminating assignments that are not essential to reach our goal. In the latter approach, input assignments can be marked as essential or not based on their impact in exposing the bug. By removing nonessential input variable assignments, the analysis of the bug trace during debugging can be made much simpler. For example, a trace with two input events will be much easier to analyze than a trace with 10 000 input events.

IV. PROPOSED TECHNIQUES

Based on our analysis, we propose several techniques to minimize a bug trace. An overview of these techniques is given in the following, and they are discussed in detail in the subsections that follow.

1) *Single-cycle elimination* shortens a bug trace by resimulating a variant of the trace that includes less simulation cycles.
2) *Alternative path to bug* is exploited by detecting when changes made on a trace produce an alternative, shorter path to the bug.

3) *State skip* identifies all the unique state configurations in a trace. If the same state occurs more than once, it indicates the presence of a loop between two states, and the trace can be reduced.
4) *BMC-based refinement* attempts to further reduce the trace length by searching locally for shorter paths between two trace states.

In addition, we propose the following techniques to simplify traces.

1) *Input event elimination* attempts to eliminate input events by resimulating trace variants that involve fewer input events.
2) *Essential variable identification* uses three-value simulation to distinguish essential input variable assignments from nonessential ones, and marks the nonessentials with "X."
3) Indirectly, all cycle removal techniques may also remove redundant input events.

A bug trace can be perturbed by either adding or removing cycles or input events. However, trying all possibilities is unfeasible. Since the purpose of minimization is to reduce the number of cycles and input events, we only use removal in the hope to find shorter and simpler traces. Our techniques are applied in the following order: Butramin first tries to shorten a trace by removing certain clock cycles and simulating such trace variants, after which it tries to reduce the number of input events. While analyzing each perturbed trace, the two techniques of alternative path to bug and state skip monitor for loops and shorter paths. Once these techniques run out of steam, Butramin applies a series of BMC refinements. The BMC search is localized so that we never generate complex SAT instances for solving, which could become the bottleneck of Butramin. If our SAT solver times out on some BMC instances, we simply ignore such instances and potential trace reductions since we do not necessarily aim for the shortest traces possible.

A. Single-Cycle Elimination

Single-cycle elimination is an aggressive but efficient way to reduce the length and the number of input events in a bug trace. It tentatively removes a whole cycle from the bug trace and checks if the bug is still exposed by the new trace through resimulation, in which case the new shorter trace replaces the old one. This procedure is applied iteratively on each cycle in the trace, starting from cycle 1 and progressing to the end of the trace. The reason we start from the first simulation cycle is that this perturbation has the best chance to move far away from the original trace, because it perturbs the early stages of a trace. The later a removal, the less the opportunity to visit states far away from the original trace.

Example 4: Consider the trace of Example 1. During the first step, single-cycle elimination attempts to remove cycle 1. If the new trace still exposes the bug, we obtain a shorter bug trace that is only two cycles long and has two input events, as shown in Fig. 7. Note that it is possible that some input events become redundant because of cycle elimination, as it is the case in this example for the event on signal c at cycle 2. This is because the previous transition on c was at cycle 1, which has now
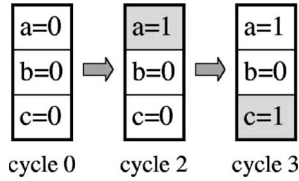
Fig. 7. Single-cycle elimination attempts to remove individual trace cycles, generating reduced traces, which still expose the bug. This example shows a reduced trace where cycle 1 has been removed.
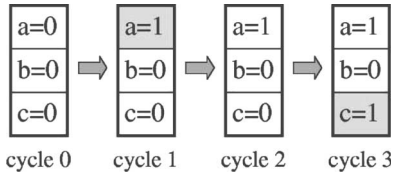

Fig. 8. Input event elimination removes pairs of events. In the example, the input events on signal c at cycle 1 and 2 are removed.

Fig. 9. Alternative path to bug. The variant trace at the bottom hits the bug at step t_2. The new trace replaces the old one, and simulation is stopped.

Fig. 10. State skip. If state $s_{j_2} = s_{i_4}$, then cycles t_3 and t_4 can be removed, obtaining a new trace that includes the sequence "$\ldots, s_{j_1}, s_{j_2}, s_{i_5}, \ldots$."

been removed. After events that have become redundant are eliminated, single-cycle elimination can be applied to cycles 2 and 3 iteratively.

To reduce Butramin's runtime, we extend single-cycle elimination to work with several cycles at once. When three consecutive cycles are eliminated one by one, Butramin will try to eliminate pairs of consecutive cycles. If that succeeds, the next attempt will consider twice as many cycles. If it fails, the number of cycles considered at once will be halved. This *adaptive cycle elimination* technique can dynamically extend its "window size" to quickly eliminate large sequences of cycles when this is likely but will roll back to single-cycle removal otherwise.

Note that when dependency exists between blocks of cycles, removing a single cycle at a time may invalidate the bug trace. For example, removing any cycle within a PCI-X transaction will almost always corrupt the transaction, rendering the bug trace useless. This problem can be addressed by removing whole transactions instead of cycles. With some extra input from the user to help identify transaction boundaries, Butramin can be easily adapted to handle transaction-based traces.

B. Input Event Elimination

Input event elimination is the basic technique to remove input events from a trace. It tentatively generates a variant trace where one input event is substituted with the complementary value assignment. If the variant trace still exposes the bug, then the input event can be removed. In addition, the event immediately following on the same signal becomes redundant and can be removed as well.

Example 5: Consider once again the trace of Example 1. The result after elimination of input event c at cycle 1 is shown in Fig. 8. Note that the input event on signal c at cycle 2 becomes redundant and is also eliminated.

C. Alternative Path to Bug

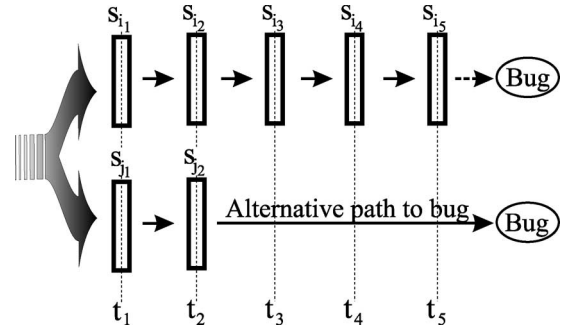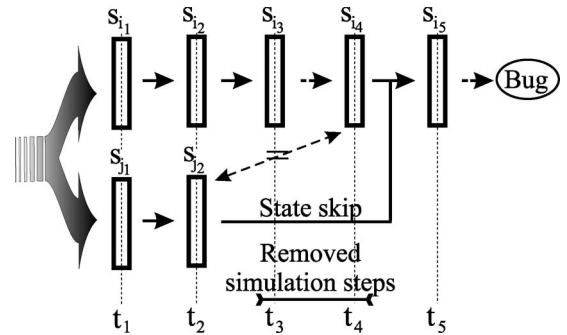An alternative path to bug occurs when a variant trace reaches a state that is different from the final state of the trace, but it also exposes the same bug. The alternative state must obviously be reached in fewer simulation steps than in the original trace. As shown in Fig. 9, if state s_{j_2}, which is reached at time t_2 by the variant trace (shown at the bottom), exposes the bug, the new variant trace replaces the original one.

D. State Skip

The state skip rule is useful when two identical states exist in a bug trace. This happens when there is a sequential loop in the trace or when, during the simulation of a tentative variant trace, an alternative (and shorter) path to a state in the original trace is found. Consider the example shown in Fig. 10: If states s_{j_2} and s_{i_4} are identical, then a new more compact trace can be generated by appending the portion from step t_5 and on of the original trace to the prefix extracted from the variant trace up to and including step t_2. This technique identifies all reoccurring states in a trace and remove cycles between them, guaranteeing that all the states in the final minimized trace are unique. States are hashed for fast lookup so that state skip does not become a bottleneck in execution.

E. Essential Variable Identification

We found that after applying our minimization techniques, bug traces are usually much shorter. However, many input variable assignments may still be part of the trace, and their relevance in exposing the bug may vary—some may be essential, while others are not. Butramin includes an "X-mode" feature for filtering out irrelevant input variable assignments, where input variable assignments are classified as essential or not based on a three-value ($0/1/X$) simulation analysis.

To implement this technique, two bits are used to encode each signal value, and each input assignment in each cycle is assigned in turn the value X. If the X input propagates to the checker's output and an X is sampled on the checker's output signal, then the input is marked essential, and the original input assignment is kept. Otherwise, the input assignment is deemed irrelevant for the purpose of exposing the bug. The set of input assignments that are marked irrelevant contribute to simplify the debugging activity, since a verification engineer does not need to take them into consideration when studying the cause of the system's incorrect behavior. We present experimental results that indicate that this analysis is capable of providing substantial simplifications to the signals involved in an already-reduced bug trace.

Note, finally, that our simplification technique, which relies on three-value simulation, is overly conservative, flagging irrelevant input assignments as essential. Consider, for instance, the simulation of a multiplexer where we propagated an X value to the select input and a 1 value to both data inputs. A three-valued logic simulator would generate X at the output of the simulator; however, for our purposes, the correct value should have been 1, since we consider X to mean "don't care." If more accuracy is desired for this analysis, a hybrid logic/symbolic simulator can be used instead [15], [20].

Alternatively, essential variable identification could be performed using a BMC-based technique with a pseudo-Boolean SAT solver, for instance, [22] and [23]. Such solvers satisfy a given SAT formula with the smallest possible number of assigned variables (maximal number of don't cares). Aside from these solvers, even mainstream Boolean SAT solvers can be specialized to do this, as suggested in [18]. Since assignments in the SAT solution correspond to input variable assignments in the bug trace, those input variable assignments are obviously essential. Essential variable identification naturally follows by marking all other input variable assignments as irrelevant. A similar idea has been deployed also by Lu *et al.* [16] to find a minimal three-valued solution that minimizes the number of assignments to state variables.

F. BMC-Based Refinement

This technique can be used after simulation-based minimization to further reduce the length of a bug trace. Because of state skip, after applying simulation-based minimization, no two states in a trace will be the same. However, the distance between any pair of states may not be minimal. We propose here an approach based on model checking to find the shortest path between two states. The algorithm, which is also outlined in Fig. 11, considers two states, say s_i and s_j, which are k cycles apart in the trace and attempts to find the shortest path connecting them. This path can then be found by unrolling the circuit from 1 to $k-1$ times, asserting s_i and s_j as the initial and final states, and attempting to satisfy the corresponding Boolean formula. If we refer to the CNF formula of the unrolled circuit as CNF_c, then $CNF_c \wedge CNF_{s_i} \wedge CNF_{s_j}$ is the Boolean formula to be satisfied. If a SAT solver can find a solution, then we have a shortcut connecting s_i to s_j. Note that the SAT instances generated by our algorithm are simplified by the fact

```
1   Select two states s_i and s_j, k cycles apart;
2   for l = 1 to k-1 do {
3       C = circuit unrolled l times;
4       Transform C into a Boolean formula CNF_c;
5       I=CNF_c ∧ CNF_s_i ∧ CNF_s_j;
6       if (I is satisfiable)
7           return (shortcut s_i → s_j, l steps);
8   }
```

Fig. 11.   BMC-based shortcut detection algorithm.
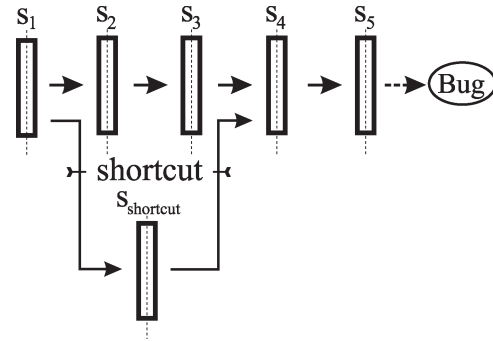

Fig. 12.   BMC-based refinement finds a shortcut between states s_1 and s_4, reducing the overall trace length by one cycle.
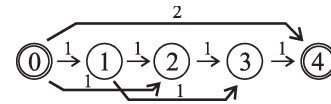

Fig. 13.   Shortest path algorithm is used to find the shortest sequence from the initial state to the bug state. The edges are labeled by the number of cycles needed to go from the source vertex to the sink. The shortest path from state 0 to state 4 in the figure uses two cycles.

that CNF_{s_i} and CNF_{s_j} are equivalent to a partial satisfying assignment for the instance. An example is given in Fig. 12.

The algorithm described in Fig. 11 is applied iteratively on each pair of states that are k steps apart in the bug trace and by using varying values for k from 2 to m, where m is selected experimentally so that the SAT instance can be solved efficiently. We then build an explicit directed graph using the shortcuts found by the BMC-based refinement and construct the final shorter path from the initial state to the bug state. Fig. 13 shows an example of such graph. Each vertex in the graph represents a state in the starting trace, edges between vertices represent the existence of a path between the corresponding states, and the edge's weight is the number of cycles needed to go from the source state to the sink. Initially, there is an edge between each two consecutive vertices, and the weight labels are 1. Edges are added between vertices when shortcuts are found between the corresponding states, and they are labeled with the number of cycles used in the shortcut. A single-source shortest path algorithm for directed acyclic graphs is then used to find the shortest path from the initial to the bug state. While some of the shortcuts discovered by BMC may be incompatible because of the partial constraints in CNF_{s_i} and CNF_{s_j}, the shortest path algorithm that we describe selects an optimal set of compatible shortcuts within the selected window size m.

Although simulation-based techniques are effective, they are heuristic in nature and may miss local optimization
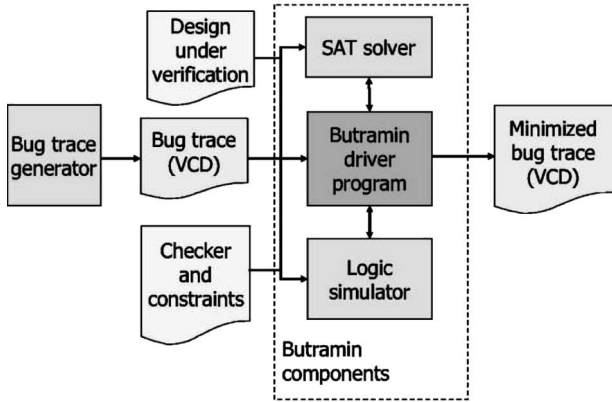
Fig. 14.   Butramin system architecture.

Fig. 15.   Early exit. If the current state s_{j_2} matches a state s_{i_2} from the original trace, then we can guarantee that the bug will eventually be hit.

opportunities. BMC-based refinement has the potential to improve on local optimizations by performing short-range optimal cycle elimination.

## V. IMPLEMENTATION INSIGHTS

We built a prototype implementation of the techniques described in Section IV to evaluate Butramin's performance and trace minimization capability on a range of digital designs. Our implementation strives to simplify a trace as much as possible while providing good performance at the same time. This section discusses some of the insights that we gained while constructing a Butramin's prototype.

A. System Architecture

The architecture of Butramin consists of three primary components, namely: 1) a driver program; 2) commercial logic simulation software; and 3) a SAT solver. The driver program is responsible for: 1) reading the bug trace; 2) interfacing to the simulation tool and SAT solver for the evaluation of the compressed variant traces; and 3) finding simplifications introduced in the previous sections. The logic simulation software is responsible for simulating test vectors from the driver program, notifying the system if the trace reaches the bug under study, and communicating back to the driver each visited state during the simulation. BMC-based minimization was implemented using MiniSAT [7], which analyzes the SAT instances generated by converting the unrolled circuits to CNF form using a CNF generator. The system architecture is shown in Fig. 14.

B. Algorithmic Analysis and Performance Optimizations

In the worst case scenario, the complexity of our simulation-based techniques is quadratic in the length of the trace under evaluation and linear in the size of the primary input signals of the design. In fact, consider an m-cycle-long bug trace driving an n-input design. The worst case complexity for our cycle elimination technique is $O(m^2)$, where the one of the input event elimination technique is $O(n \times m^2)$. All the other simulation-based techniques have simpler complexity or are independent from the size of the trace or design. In order to
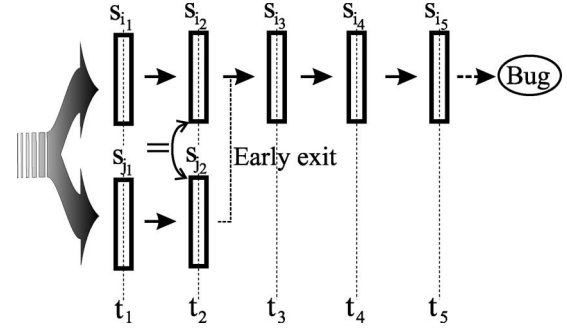
improve on the wall-clock profile of Butramin, we developed an extra optimization, as described in the following. Experimental results show that the worst case situation did not occur due to our optimization, adaptive cycle elimination, and the nature of practical benchmarks.

The optimization focuses on identifying all multiple occurrences of a state so that we can identify when the simulation of a variant trace falls into the original trace and avoid simulating the last portion of the variant. To achieve this, we hash all states visited by a trace and tag them with the clock cycle in which they occur. During the simulation of variant traces, we noted that in some special conditions, we can improve the performance of Butramin by reducing the simulation required. After the time when the original and the variant traces differ, if a variant state matches a state in the original trace tagged by the same clock cycle, then we can terminate the variant simulation and still guarantee that the variant trace will hit the bug. In other words, simulation can be terminated early because the result of applying the same test vectors after the matched state will not change. We call this an *early exit*. As illustrated in Fig. 15, early exit points allow the simulation to terminate immediately. Often simulation can also be terminated early by state skip optimization because the destination state is already in the trace database. Experimental results show that this optimization is crucial to the efficiency of simulation-based minimization techniques.

C. Use Model

To run Butramin, the user must supply four inputs, namely: 1) the design under test; 2) a bug trace; 3) the property that was falsified by the trace; and 4) an optional set of constraints on the design's input signals. Traces are represented as value change dump (VCD) files—a common compact format that includes all top-level input events. Similarly, the minimized bug traces are output as VCD files.

Removing input events from the bug trace during trace minimization may generate illegal input sequences, which, in turn, could erroneously falsify a property or make the trace useless. For example, removing the reset event from a bug trace may lead the design into an erroneous state, generating a spurious trace that does not reflect a possible legal activity of the design under verification, even if the simulation of such trace does expose the original design flaw. Consequently, when testing

subcomponents of a design with constrained inputs, it becomes necessary to validate the input sequences generated during trace minimization. There are several ways to achieve this goal. One technique is to mark required inputs so that Butramin does not attempt to remove the corresponding events from the trace. This approach is a viable solution to handle, for instance, reset and the clock signals. For complex sets of constraints, it is possible to convert them into an equivalent circuit block connected to the original design, such as the techniques described in the work by Yuan *et al.* [21]. This extra circuit block takes random input assignments and converts them into a set of legal assignments that satisfy all the required environment constraints. We deployed the former approach for simple situations, and we adapted the latter to the context of our solution for benchmarks with more complex environments. Specifically, since Butramin starts already with a valid input trace that it attempts to simplify, we wrote our constraints as a set of monitors that observe each input sequence to the design. If the monitors flag an illegal transition during simulation, the entire "candidate trace" is deemed invalid and removed from consideration. For BMC-based refinement, these environmental constraints are synthesized and included as additional constraints to the problem instance. Note, however, that this limits BMC-based techniques to be applied to designs whose environmental constraints are synthesizable. On the other hand, this requirement is lifted for the simulation-based minimization techniques. From our experimental results, we observe that most minimization is contributed by simulation-based techniques, which renders this requirement optional for most practical benchmarks.

We also developed an alternative model to apply Butramin to reduce regression runtime. In this context, the approach is slightly different since the goal now is to obtain shorter traces that achieve the same functional coverage as their longer counterpart. To support this, coverage points are encoded by properties. Each of them is "violated" only when the corresponding point is covered by the trace. Butramin can then be configured to generate traces that violate all of the properties, instead of just one, so that the same coverage is maintained.

VI. EXPERIMENTAL RESULTS

We evaluated Butramin by minimizing traces generated by a range of commercial verification tools: a constrained random simulator, semiformal verification software, and, again, a semiformal tool where we specified to use extra effort in generating compact traces. We considered nine benchmark designs from OpenCores (FPU), ISCAS89 (S15850, S38584), ITC99 (B15), IWLS2005 (VGALCD), picoJava (picoJava, ICU), and two internally developed benchmarks (MULT, DES), whose characteristics are reported in Table I. We developed assertions to be falsified when not already available with the design, and we inserted bugs in the design that falsify the assertions. Table II describes assertions and bugs inserted. The checker for VGALCD is a correct duplicate of the original design (which we modified to contain one design error); hence, the circuit size we worked with is twice as the one reported in Table I. Finally, experiments were conducted on a Sun Blade 1500 (1 GHz UltraSPARC IIIi) workstation running Solaris 9.

TABLE I
BENCHMARK CHARACTERISTICS. THE BENCHMARK SETUP FOR VGALCD INVOLVES DUPLICATING THIS DESIGN AND MODIFYING ONE CONNECTION IN ONE OF THE COPIES. BUTRAMIN THEN MUST MINIMIZE THE TRACE EXPOSING THE DIFFERENCE. IT FOLLOWS THAT THE SIZE OF THE BENCHMARK WE WORK WITH IS ACTUALLY TWICE THE ONE REPORTED FOR THIS DESIGN

Benchmark	Inputs	Latches	Gates	Description
S38584	41	1426	20681	Unknown
S15850	77	534	10306	Unknown
MULT	257	1280	130164	Wallace tree multiplier
DES	97	13248	49183	DES algorithm
B15	38	449	8886	Portion of 80386
FPU	72	761	7247	Floating Point Unit
ICU	30	62	506	PicoJava Instr. cache unit
picoJava	53	14637	24773	PicoJava full design
VGALCD	56	17505	106547	VGA/LCD controller

A. Simulation-Based Experiments

Our first set of experiments attempts to minimize traces generated by running a semiformal commercial verification tool with the checkers specified and subsequently applying only the simulation-based minimization techniques of Butramin, as described in Sections IV-A–D. We were not able to complete the generation of traces with the semiformal verification tool for VGALCD; therefore, we only report results related to constrained random traces for this benchmark. Table III shows the absolute values of cycles and input events left in each trace and the overall runtime of Butramin using only simulation-based techniques. Figs. 16 and 17 show the percentages of cycles and input events removed from the original bug trace using different techniques. Note that for all benchmarks, we are able to remove the majority of cycles and input events.

With reference to Figs. 16 and 17, we observe that the contribution of different minimization techniques varies among benchmarks. For example, almost all the cycles and input events are removed by cycle elimination in FPU and picoJava. On the other hand, state skip removes more than half of the cycles and input events in B15 and ICU. This difference can be attributed to the nature of the benchmark. If there are fewer state variables in the design, state skip is more likely to occur. In general, state skip has more opportunities to provide trace reductions in designs that are control heavy, such as ICU, compared to designs that are data path heavy, such as FPU and picoJava. Although input event elimination does not remove cycles, it has great impact in eliminating input events for some benchmarks, such as S38584. Overall, we found that all these techniques are important to compact different types of bug traces.

Our second set of experiments applies Butramin to a new set of traces, which are also generated by a semiformal tool, but this time, we configured the software to dedicate extra effort in generating short traces by allowing more time to be spent on the formal analysis of the checker. Analogous to Table III discussed earlier, Table IV reports the results obtained by applying the simulation-based minimization techniques of Butramin to these traces. We still find that Butramin has a high impact in compacting these traces, even if, generally speaking, they present less redundancy, since they are closer to be minimal. Note, in particular, that the longer the traces, the greater the benefit from the application of Butramin. Even if the overall impact

TABLE II
BUGS INJECTED AND ASSERTIONS FOR TRACE GENERATION. FOR ICU AND picoJava, NO BUGS WERE
INJECTED BUT THE CONSTRAINTS FOR RANDOM SIMULATION WERE RELAXED

Circuit	Bug injected	Assertion used
S38584	None	Output signals forced to a specific value
S15850	None	Output signals forced to a specific value
MULT	AND gate changed with XOR	Compute the correct output value
DES	Complemented output	Timing between receive_valid, output_ready and transmit_valid
B15	None	Coverage of a partial design state
FPU	divide_on_zero conditionally complement	Assert divide_on_zero when divisor=0
ICU	Constraints relaxed	Buffer-full condition
picoJava	Constraints relaxed	Assert SMU's spill and fill
VGALCD	Circuit duplicated with one wire changed in one copy	Outputs mismatch condition

TABLE III
CYCLES AND INPUT EVENTS REMOVED BY SIMULATION-BASED TECHNIQUES OF BUTRAMIN ON TRACES GENERATED BY SEMIFORMAL VERIFICATION

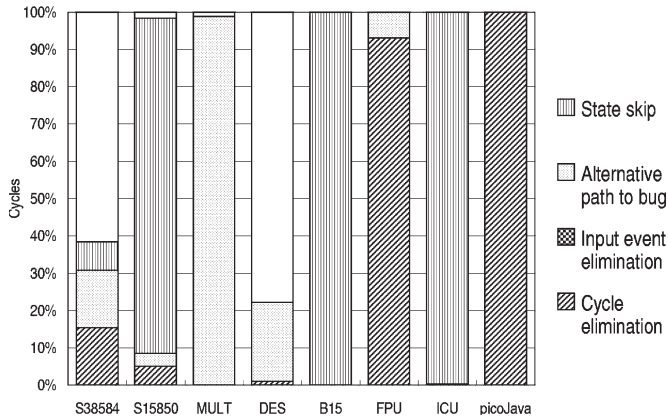| Circuit	Cycles			Input events			Runtime
	Original	Remaining	Removed	Original	Remaining	Removed	(seconds)
S38584	13	8	38.46%	255	2	99.22%	19
S15850	59	1	98.31%	2300	3	99.87%	5
MULT	345	4	98.84%	43843	2	99.99%	35
DES	198	154	22.22%	3293	3	99.91%	254
B15	25015	11	99.96%	450026	15	99.99%	57
FPU	53711	5	99.99%	1756431	17	99.99%	27
ICU	6994	3	99.96%	62740	3	99.99%	5
picoJava	30016	10	99.97%	675485	11	99.99%	3359

Fig. 16. Percentage of cycles removed using different simulation-based techniques. For benchmarks like B15 and ICU, state skip is the most effective technique because they contain small numbers of state variables, and state repetition is more likely to occur. For large benchmarks with long traces like FPU and picoJava, cycle elimination is the most effective technique.
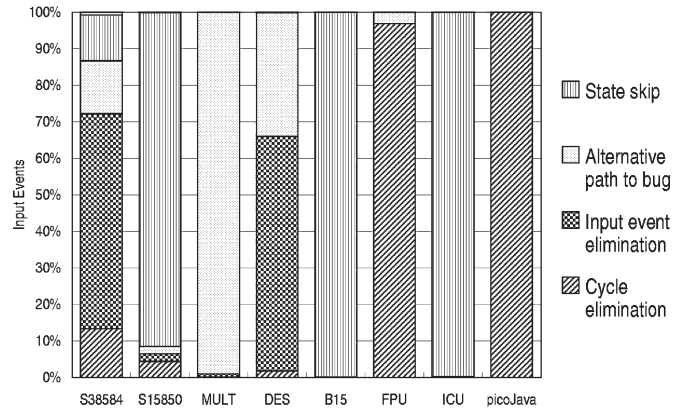

Fig. 17. Number of input events eliminated with simulation-based techniques. The distributions are similar to cycle elimination because removing cycles also removes input events. However, input event elimination works the most effectively for some benchmarks like S38584 and DES, showing that some redundant input events can only be removed by this technique.

is reduced, we still observe a 61% reduction in the number of cycles and 91% in input events on the average.

The third set of experiments evaluated traces generated by constrained random simulation. Results are summarized in Table V. As expected, Butramin produced the most impact on this set of traces, since they tend to include a lot of redundant behavior. The average reduction is 99% in terms of cycles and input events.

B. Performance Analysis

Table VI compares Butramin's runtime with and without different optimization techniques. The traces are generated using semiformal methods in this comparison. The execution runs that exceeded 40 000 s were timed out (T/O in the table). The runtime comparison shows that early exit and state skip have great impacts on the execution time. Early exit can stop resimulation early, and state skip may reduce the length of a trace by many cycles at a time. Although these two techniques require extra memory, the reduction in runtime shows that they are worthwhile. In ICU, state skip occurred four times, removing 6977 cycles, which resulted in a very short runtime. The comparison also shows that adaptive cycle elimination is capable of reducing minimization time significantly. This technique is especially beneficial for long bug traces, such as FPU and picoJava.

TABLE IV
CYCLES AND INPUT EVENTS REMOVED BY SIMULATION-BASED TECHNIQUES OF BUTRAMIN ON
TRACES GENERATED BY A COMPACT-MODE SEMIFORMAL VERIFICATION TOOL

Circuit	Cycles			Input events			Runtime
	Original	Remaining	Removed	Original	Remaining	Removed	(seconds)
S38584	13	8	38.46%	255	2	99.22%	21
S15850	17	1	94.12%	559	56	89.98%	4
MULT	6	4	33.33%	660	2	99.70%	34
DES	296	17	94.26%	3425	3	99.91%	17
B15	27	11	59.26%	546	5	99.08%	6
FPU	23	5	78.26%	800	17	97.88%	1
ICU	19	14	26.32%	142	80	43.66%	1
picoJava	26	10	61.54%	681	11	98.38%	39

TABLE V
CYCLES AND INPUT EVENTS REMOVED BY SIMULATION-BASED METHODS OF BUTRAMIN ON
TRACES GENERATED BY CONSTRAINED RANDOM SIMULATION

Circuit	Cycles			Input events			Runtime
	Original	Remaining	Removed	Original	Remaining	Removed	(seconds)
S38584	1003	8	99.20%	19047	2	99.99%	16
S15850	2001	1	99.95%	77344	3	99.99%	2
MULT	1003	4	99.60%	128199	2	99.99%	34
DES	25196	154	99.39%	666098	3	99.99%	255
B15	148510	10	99.99%	2675459	9	99.99%	395
FPU	1046188	5	99.99%	36125365	17	99.99%	723
ICU	31992	3	99.99%	287729	3	99.99%	5
picoJava	99026	10	99.99%	2227599	16	99.99%	5125
VGALCD	36595	4	99.99%	1554616	19	99.99%	28027

TABLE VI
IMPACT OF THE VARIOUS SIMULATION-BASED TECHNIQUES ON BUTRAMIN'S RUNTIME. BENCHMARKS THAT EXCEEDED TIME LIMIT
(40 000 s) ARE NOT INCLUDED IN THE AVERAGE. EACH OF THE RUNTIME COLUMNS REPORTS THE RUNTIME USING ONLY A SUBSET
OF OUR TECHNIQUES: THE FIRST COLUMN DEPICTS CYCLE ELIMINATION AND INPUT-EVENT ELIMINATION, THE SECOND
INCLUDES IN ADDITION EARLY EXIT AND STATE SKIP, AND THE THIRD ADDS ADAPTIVE CYCLE ELIMINATION

Benchmark	Runtime(seconds)		
	[1]: cycle elimination+ input event elimination	[2]: [1]+state skip+ early exit	[3]: [2]+adaptive cycle elimination
S38584	21	19	19
S15850	11	5	5
MULT	48	43	35
DES	274	256	254
B15	T/O	58	57
FPU	T/O	235	27
ICU	8129	5	5
picoJava	T/O	T/O	3359
Average	1697	66	64

A comparison of Butramin's impact and runtime on the three sets of traces is summarized in Fig. 18. The result shows that Butramin can effectively reduce all three types of bug traces in a reasonable amount of time. Note, in addition, that in some cases, the minimization of a trace generated by random simulation takes similar or less time than applying Butramin to a trace generated by a compact-mode semiformal tool, even if the initial trace is much longer. That is the case for S38584 or S15850. We explain this effect by the nature of the bug traces: Traces generated by random simulation tend to visit states that are easily reachable; therefore, states are likely to be repetitive, and state skip occurs more frequently, leading to a shorter minimization time. On the other hand, states visited in a compact-mode generated trace mode are more frequently produced by formal engines and can be highly specific, making state skip a rare event. The cases of FPU and picoJava are relevant in this context: Here, state skips do not occur, and the

minimization time is highly related to the original trace length. They also demonstrate the benefits of Butramin in verification methodologies.

C. Essential Variable Identification

We also applied the technique from Section IV-E to identify essential variables from the minimized traces that we generated. Table VII shows that after this technique is applied, many input variable assignments are marked nonessential, further simplifying the trace. Note that the comparison is now between input variable assignments and not between input events. Since all nonessential input variable assignments are simulated with X, the simulation will propagate X values to many internal signals as well. As a result, it will be easier to understand the impact of essential variable assignments on violated properties.
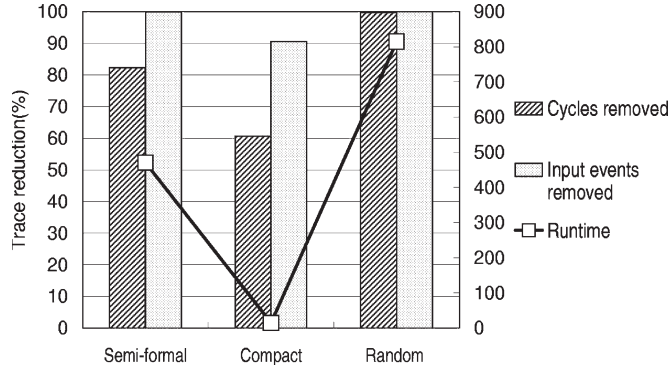
Fig. 18. Comparison of Butramin's impact when applied to traces generated in three different modes. The graph shows the fraction of cycles and input events eliminated as well as the average runtime.

TABLE VII

ESSENTIAL VARIABLE ASSIGNMENTS IDENTIFIED IN X-MODE. THE TABLE COMPARES THE NUMBER OF INPUT VARIABLE ASSIGNMENTS IN THE MINIMIZED TRACES WITH THE NUMBER OF ASSIGNMENTS CLASSIFIED ESSENTIAL. ALL THE REMAINING ASSIGNMENTS ARE NONESSENTIAL AND CAN BE SUBSTITUTED BY X VALUES IN SIMULATION. THE INITIAL TRACES WERE GENERATED BY SEMIFORMAL VERIFICATION SOFTWARE

Circuit	Input variables	Essential variables
S38584	320	2
S15850	76	2
MULT	1024	1019
DES	14748	2
B15	407	45
FPU	355	94
ICU	87	21
picoJava	520	374

D. Generation of High-Coverage Traces

In order to evaluate the effectiveness of Butramin applied to reducing regression runtime, we selected three benchmarks, namely DES, FPU, and VGALCD, as our multiproperty benchmarks. The original properties in the previous experiments were preserved, and the same traces generated by constrained random simulation were used. In addition, we included a few extra properties so that our original traces would expose them before reaching their last simulation step, which still exposes the original property that we used, as described in Table II. Those extra properties specify a certain partial state to be visited or a certain output signal to be asserted. Butramin is then configured to produce minimized traces that violate all properties. The results are summarized in Table VIII. Compared with Table V, it can be observed that in order to cover extra properties, the length of the minimized traces are now longer. However, Butramin continues to be effective for these multiproperty traces. We also found that the order of property violations is preserved before and after minimization, suggesting that Butramin minimizes segments of bug traces individually. From an algorithmic complexity point of view, minimizing a multiproperty trace is similar to minimizing many single-property traces with different initial states.

While the original traces of FPU and VGALCD require 20–30 min to be simulated, post-Butramin traces are short enough to be simulated in just a few seconds. The benefits of adding the minimized trace to a regression suite, instead of the original one, are obvious.

E. BMC-Based Experiments

We applied our BMC-based technique to traces already minimized by simulation-based methods to evaluate the potential for further minimization. For VGALCD, we report only data related to the minimization of random trace since semiformal traces are not available. The results are summarized in Table IX, where *Orig* is the original number of cycles in the trace, and *Removed* is the number of cycles removed by this method. We used a maximum window of ten cycles (i.e., $m = 10$). The main observation that can be made is that simulation-based techniques are very effective in minimizing bug traces. In fact, only in two cases, i.e., ICU and B15, was our BMC-based technique able to extract additional minimization opportunities. Potentially, we could repeat the application of simulation-based techniques and BMC-based methods until convergence when no additional minimization can be extracted.

In order to compare the performance of the BMC-based technique with our simulation-based methods, we applied the former directly to minimize the original bug traces generated by semiformal verification and by constrained random simulation. For this experiment, the time-out limit was set to 40 000 s. Results are summarized in Table X, where benchmarks that timed out are marked by "T/O." The findings reported in the table confirm that our BMC-based method should only be applied, if at all, after the simulation-based techniques have already greatly reduced the trace complexity.

F. Evaluation of Experimental Results

We attempted to gain more insights into the results obtained by evaluating two additional aspects of the minimized traces. We first checked how close the minimized traces are to optimal-length traces, such as those generated by formal verification. To do so, we run full-fledged SAT-based BMC on our minimized traces. The results show that our techniques found minimal-length bug traces for all benchmarks except DES (both traces generated by random simulation and semiformal verification). For those two traces, the SAT solver ran out of memory after we unrolled the design by 118 cycles and could not finish the experiment, while no shorter traces were found between 1 and 118 cycles long.

We also tried to evaluate if the potential for simulation-based trace reduction was mostly due to a large number of bug states, that is, a high number of design configurations that expose a given bug (an example of this situation is provided in Fig. 1). To evaluate this aspect, we considered the original nonminimized traces in our experimental results, we sampled the final state of the design after simulating the traces, and we fixed the goal of Butramin to generate a minimized trace that reaches that exact same final state. The results of this experiment are summarized in Table XI. The table shows that, for most benchmarks, the difference in the number of input events and cycles removed is small, showing that the size of the bug configuration has a minimal impact on the ability of Butramin to reduce and

TABLE VIII
CYCLES AND INPUT EVENTS REMOVED BY SIMULATION-BASED METHODS OF BUTRAMIN ON TRACES THAT VIOLATE MULTIPLE PROPERTIES

Circuit	Number of properties	Cycles			Input events			Runtime (seconds)
		Original	Remaining	Removed	Original	Remaining	Removed	
DES	2	25196	184	99.27%	666098	17	99.99%	549
FPU	3	1046188	9	99.99%	36125365	264	99.99%	580
VGALCD	3	36595	5	99.98%	1554616	22	99.99%	25660

TABLE IX
CYCLES REMOVED BY BMC-BASED METHOD: ICU AND B15 CAN BE MINIMIZED FURTHER AFTER BUTRAMIN'S SIMULATION TECHNIQUES

Circuit	Semi-formal			Compact-trace			Constrained random		
	Orig	Removed	Time	Orig	Removed	Time	Orig	Removed	Time
S38584	8	0	55s	8	0	55s	8	0	55s
S15850	1	0	2s	1	0	2s	1	0	2s
MULT	4	0	20s	4	0	20s	4	0	20s
DES	154	0	23h3m	17	0	357s	154	0	23h3m
B15	**11**	**1**	**121s**	**11**	**1**	**121s**	10	0	97s
FPU	5	0	5s	5	0	5s	5	0	5s
ICU	**3**	**1**	**1s**	**14**	**2**	**1s**	**3**	**1**	**1s**
picoJava	10	0	70s	10	0	70s	10	0	104s
VGALCD	N/A	N/A	N/A	N/A	N/A	N/A	4	0	985s

TABLE X
ANALYSIS OF A PURE BMC-BASED MINIMIZATION TECHNIQUE. THIS TABLE SHOWS POTENTIAL FOR MINIMIZING TRACES USING OUR BMC-BASED TECHNIQUE ALONE. COLUMN "ORIGINAL" SHOWS LENGTH (IN CYCLES OF THE ORIGINAL TRACE), AND COLUMN "REMAINED" SHOWS THE LENGTH OF THE MINIMIZED TRACE OBTAINED AFTER APPLYING THE BMC-BASED METHOD. TRACES IN THE TOP HALF WERE GENERATED BY SEMIFORMAL VERIFICATION, THE ONES IN THE BOTTOM HALF WERE GENERATED BY CONSTRAINED RANDOM SIMULATION. EXPERIMENTS ARE TIMED OUT AT 40 000 s. THE RESULTS OF THIS TABLE SHOULD BE COMPARED WITH TABLES III AND V

Circuit	Original	Remained	Runtime(s)
S38584	13	9	403
S15850	59	59	338
MULT	345	T/O	T/O
DES	198	T/O	T/O
B15	25015	T/O	T/O
FPU	53711	T/O	T/O
ICU	6994	700	856
picoJava	30016	T/O	T/O
FPU	1046188	T/O	T/O
picoJava	99026	T/O	T/O
VGALCD	36595	T/O	T/O

TABLE XI
ANALYSIS OF THE IMPACT OF A BUG RADIUS ON BUTRAMIN EFFECTIVENESS. THE TABLE COMPARES THE NUMBER OF CYCLES AND INPUT EVENTS IN THE ORIGINAL TRACES TO THE SAME VALUES FROM MINIMIZED TRACES THAT HIT THE SAME BUG AND TO MINIMIZED TRACES THAT REACH THE EXACT SAME BUG CONFIGURATION. TRACES IN THE TOP HALF WERE GENERATED BY SEMIFORMAL SOFTWARE AND TRACES IN THE BOTTOM HALF WERE GENERATED BY CONSTRAINED RANDOM SIMULATION

Circuit	Cycles			Input events		
	Original trace	Same bug	Same state	Original trace	Same bug	Same state
S38584	13	8	9	255	2	41
S15850	59	1	1	2300	3	3
MULT	345	4	4	43843	2	380
DES	198	154	193	3293	3	1022
B15	25015	11	11	450026	15	40
FPU	53711	5	5	1756431	17	112
ICU	6994	3	5	62740	3	6
picoJava	30016	10	75	675485	11	1575
FPU	1046188	5	6	36125365	17	120
picoJava	99026	10	22	2227599	16	42
VGALCD	36595	4	199	1554616	19	2068

simplify a given bug trace, and our proposed solution remains effective even when the bug configuration is very specific.

VII. CONCLUSION

This paper presented Butramin, which is a bug trace minimizer that combines simulation-based techniques with formal methods. Butramin applies simple but powerful simulation-based bug trace reductions, such as *cycle elimination*, *input event elimination*, *alternative path to bug*, *state skip*, and *essential variable identification*. An additional BMC-based refinement method is used after these techniques to exploit the potential for further minimizations. Compared to purely formal methods, Butramin has the following advantages: 1) it can reduce both the length of a bug trace and the number of its input events; 2) it leverages fast logic-simulation engines for bug

trace minimization and it can scale to industrial size designs; and 3) it leverages the existing simulation-based infrastructure, which is currently prevalent in the industry. This significantly lowers the barriers for industrial adoption of automatic design verification techniques.

Our experimental results show that Butramin can reduce a bug trace to just a small fraction of its original length and complexity (which is estimated as the number of input events in the trace) by using only simulation-based techniques. In fact, for most of the benchmarks considered, we found that Butramin found an alternative trace of minimum length. In addition, we showed that these results are largely independent of the verification methodology used to generate the trace, whether based on simulation or semiformal verification techniques. The impact of Butramin appears to be uncorrelated with the size of the bug configuration targeted by the trace, that is, the number of distinct design states that expose the bug.

REFERENCES

[1] M. Aagaard, R. Jones, and C.-J. Seger, "Combining theorem proving and trajectory evaluation in an industrial environment," in *Proc. DAC*, 1998, pp. 538–541.

[2] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models*, 2nd ed. Norwell, MA: Kluwer, 2003.

[3] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proc. TACAS*, 1999, vol. 1579, pp. 193–207.

[4] K.-H. Chang, V. Bertacco, and I. L. Markov, "Simulation-based bug trace minimization with BMC-based refinement," in *Proc. ICCAD*, 2005, pp. 1045–1051.

[5] Y. A. Chen and F. S. Chen, "Algorithms for compacting error traces," in *Proc. ASPDAC*, 2003, pp. 99–103.

[6] O. Coudert, C. Berthet, and J. C. Madre, "Verification of synchronous sequential machines based on symbolic execution," in *Proc. Autom. Verification Methods for Finite State Syst.*, 1990, vol. 407, pp. 365–373.

[7] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Proc. Theory and Appl. Satisfiability Testing*, 2003, pp. 502–518.

[8] P. Gastin, P. Moro, and M. Zeitoun, "Minimization of counterexamples in SPIN," in *Proc. SPIN*, 2004, vol. 2989, pp. 92–108.

[9] A. Groce and D. Kroening, "Making the most of BMC counterexamples," in *Proc. Workshop BMC*, 2004, pp. 71–84.

[10] S. Hazelhurst, O. Weissberg, G. Kamhi, and L. Fix, "A hybrid verification approach: Getting deep into the design," in *Proc. DAC*, 2002, pp. 111–116.

[11] R. Hildebrandt and A. Zeller, "Simplifying failure-inducing input," in *Proc. Int. Symp. Software Testing and Anal.*, 2000, pp. 134–145.

[12] P. H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long, "Smart simulation using collaborative formal and simulation engines," in *Proc. ICCAD*, 2000, pp. 120–126.

[13] A. Hu, "Formal hardware verification with BDDs: An introduction," in *Proc. PACRIM*, 1997, pp. 677–682.

[14] H. Jin, K. Ravi, and F. Somenzi, "Fate and free will in error traces," in *Proc. TACAS*, 2002, vol. 2280, pp. 445–459.

[15] A. Kolbl, J. Kukula, and R. Damiano, "Symbolic RTL simulation," in *Proc. DAC*, 2001, pp. 47–52.

[16] F. Lu, M. K. Iyer, G. Parthasarathy, L.-C. Wang, K. T. Cheng, and K. C. Chen, "An efficient sequential SAT solver with improved search strategies," in *Proc. DATE*, 2005, pp. 1102–1107.

[17] P. Rashinkar, P. Paterson, and L. Singh, *System-on-a-Chip Verification: Methodology and Techniques*. Norwell, MA: Kluwer, 2002.

[18] K. Ravi and F. Somenzi, "Minimal satisfying assignments for bounded model checking," in *Proc. TACAS*, 2004, vol. 2988, pp. 31–45.

[19] S. Shen, Y. Qin, and S. Li, "A fast counterexample minimization approach with refutation analysis and incremental SAT," in *Proc. ASP-DAC*, 2005, pp. 451–454.

[20] C. Wilson and D. L. Dill, "Reliable verification using symbolic simulation with scalar values," in *Proc. DAC*, 2000, pp. 124–129.

[21] J. Yuan, K. Albin, A. Aziz, and C. Pixley, "Constraint synthesis for environment modeling in functional verification," in *Proc. DAC*, 2003, pp. 296–299.

[22] N. Eén and N. Sörensson, *The MiniSat Page*. [Online]. Available: http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/

[23] H. M. Sheini and K. A. Sakallah, *Pueblo SAT Solver*. [Online]. Available: http://www.eecs.umich.edu/~hsheini/pueblo/

Valeria Bertacco (S'95–M'03) received the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 2003, and the Laurea degree (*summa cum laude*) in computer engineering from the University of Padova, Padova, Italy.

She is an Assistant Professor with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor. Her research interests are in the areas of formal and semiformal design verification with emphasis on full design validation and digital system reliability. She joined the faculty at the University of Michigan after being with the Advanced Technology Group, Synopsys, Mountain View, CA, for four years as a Staff Research Engineer. Prior to Synopsys, she was with Systems Science Inc., Palo Alto, CA, a startup which developed Vera, a testbench development language for verification, which was later acquired by Synopsys.

Dr. Bertacco is an Associate Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS. She served in the Program Committee of the International Conference on CAD and as Chair of the Verification Committee of the Design Automation and Test in Europe Conference. She has been leading the effort for the development of the verification section of the International Technology Roadmap for Semiconductors report since 2004.

Kai-Hui Chang (S'04) received the B.S. and M.S. degrees in electrical engineering from the National Taiwan University, Taipei, Taiwan, R.O.C., in 1999 and 2001, respectively. He is currently working toward the Ph.D. degree at the University of Michigan, Ann Arbor.

From 2001 to 2004, he worked as a Software Engineer with Avery Design Systems, Andover, MA. His research interests include design verification and postplacement optimizations.

Igor L. Markov (S'97–M'01–SM'05) received the M.A. degree in mathematics and the Ph.D. degree in computer science from the University of California, Los Angeles.

He is with the Advance Computer Architecture Laboratory, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, as an Associate Professor. His interests and publications are in combinatorial optimization, with applications to the design and verification of integrated circuits, as well as in quantum logic circuits. His research contributions include new algorithmic techniques for Boolean satisfiability, hypergraph partitioning, block packing, large-scale circuit layout, synthesis of reversible and quantum circuits, and quantum-mechanical simulation with compressed matrices. Some of these algorithms lead to order-of-magnitude improvements in practice, and many of them are implemented in software, including open-source projects and major commercial tools. He has coauthored more than 100 publications. He has graduated three Ph.D. and four M.S. students and is currently working with six graduate students. His current and former students interned at or are employed by Advanced Micro Devices, Amazon.com, Cadence, Calypto, the U.S. Department of Defense, the U.S. Department of Energy, Google, IBM, Lockheed Martin, Microsoft, Qualcomm, Synplicity, and Xilinx.

Dr. Markov has served on program committees of most major conferences in electronic design automation. He was awarded the DAC Fellowship and received the IBM University Partnership Award in 2001. He received the 2004 IEEE CAS Donald O. Pederson Paper of the Year Award, the 2004 ACM SIGDA Outstanding New Faculty Award, and the Best Paper Award at DATE 2005. He also received the Outstanding Ph.D. Student Award in 2000.