# Simulation-based Bug Trace Minimization with BMC-based Refinement

Kai-hui Chang
University of Michigan
EECS Department
Ann Arbor, MI 48109-2122
changkh@umich.edu

Valeria Bertacco
University of Michigan
EECS Department
Ann Arbor, MI 48109-2122
valeria@umich.edu

Igor L. Markov
University of Michigan
EECS Department
Ann Arbor, MI 48109-2122
imarkov@umich.edu

## 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 contain several hundred thousand cycles. In this work we propose Butramin, a bug trace minimizer. Butramin considers a bug trace produced by a random simulator or a semi-formal 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. We evaluated Butramin on a range of designs, including the publicly available picoJava microprocessor. Our experiments show that in most cases Butramin is able to reduce traces to a small fraction of their initial size, in terms of cycle length and signals involved.

## 1. 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 first tape-out, and that verification accounts for two thirds of the design cycle and effort [2, 14]. 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 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 (when simple bugs have already been flushed out), it is very common that the bug traces generated are very complex and can often be hundreds of thousands of cycles long.

Another family of techniques gaining increasing attention from industry is that of semi-formal 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) or reachability analysis [12, 3], would be ideal to generate compact high-coverage tests, such as, for instance, 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 semi-formal techniques, such as [1, 11, 9], heuristics and randomized exploration allow designers to obtain high-coverage results on designs of medium and large complexity, but, to this end, 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 specification or an erroneous property definition.

Future trends are to generate high quality results while demanding limited effort from the verification engineer, such as the previously mentioned random simulation and semi-formal verification techniques. Both these techniques are very appealing when compared to the most traditional direct-test simulation approach, which is extremely demanding, requiring the direct development of entire sets of specific test stimuli. At the same time, these trends, and the growing complexity of digital designs, are deemed to exasperate the debugging phase of verification by producing increasingly long and complex bug traces.

**Contributions**. This work addresses the problem of debugging complex bug traces by proposing a technique for trace minimization called Butramin ("BUg TRAce MINimization"). 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 arena has been centered on using formal techniques to simplify the counterexample [15, 4]. In a separate context, the problem of trace minimization has also been addressed in software verification [10, 7].

Butramin simplifies a trace by incrementally 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 some input events in the bug trace are redundant. For every candidate simplification the trace is re-simulated to check if it is still a valid counterexample. Once no additional simplifications can be performed, Butramin attempts to further reduce the trace by using a formal verification engine, a SAT-based bounded

0-7803-9254-X/05/$20.00 ©2005 IEEE.

model checker, and to search for "short-cuts" along the trace path. A directed graph is built according to the shortcuts and a shortest-path algorithm is used to find the best path from the initial state to the bug state. Our approach to trace minimization is novel in the following aspects:

- It simplifies the trace incrementally targeting the total number of clock cycles as well as the input-events of the trace.
- It combines simulation and formal techniques, which exploits the performance of logic simulation as far as possible and only applies formal techniques to an already reduced trace, requiring a much simpler analysis.
- Experimental results show that Butramin can greatly simplify counterexamples generated by semi-formal verification tools down to a small fraction of their original size, 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, in this context, we found that almost all of our execution time are well within this limit.

The remainder of this paper is organized as follows: Section 2 describes relevant previous work on bug trace minimization for random simulation and bounded model-checking. Section 3 presents our new bug trace minimization technique that relies on logic simulation, and describes the BMC-based search for counterexample shortcuts. Sections 4 and 5 discuss algorithmic aspects of Butramin and experimental results. Finally, Section 6 summarizes the contributions and concludes the paper.

# 2. BACKGROUND AND PREVIOUS WORK

Research focusing on minimizing property counterexamples or, more in general, bug traces has been pursued both in the context of hardware and software verification. In the hardware verification domain, the majority of the solutions proposed is concerned with minimizing traces generated by deploying a bounded model checker (BMC). Before discussing these techniques, we give some preliminary background and provide a brief overview of the BMC algorithm.

## 2.1 Preliminaries

A *bug trace* is a sequence of test vectors that leads a logic simulator to hit a bug. 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* is a value assignment to an input signal at a specific cycle. A *checker signal* is a signal used to detect a violation of a checker, that is, if the signal changes to a specific value, then the checker is violated, and a bug is found. The objective of *bug trace minimization* is to reduce the number of input events and cycles in a trace.

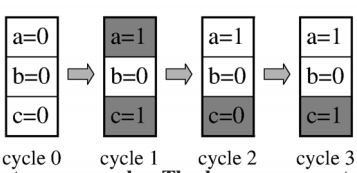


**Figure 1: Bug trace example. The boxes represent input variables to the circuit at each cycle, shaded boxes represent input events. This trace has three cycles, four input events and twelve input variables.**

EXAMPLE 1. *Consider a circuit with three inputs a, b and c, 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. *In this situation we count four input events, twelve input variables, and three cycles for our bug trace. The example trace is illustrated in Figure 1.*

## 2.2 Bounded Model Checking Overview

Bounded model checking (BMC) [3] is a formal method which can prove or disprove properties of bounded length in a design, frequently using SATisfiability solving techniques to achieve this goal. A high level flow of the algorithm is given in Figure 2. 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 ($\overline{p}$). 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.

```
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  }
```

**Figure 2: Bounded Model Checking pseudo-code.**

## 2.3 BMC-based Techniques

Traditionally, a counterexample generated by this technique reports the input value assignments (also called input variables) 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 et al. [15] are concerned with removing input variables from a counterexample. They propose two techniques to this end: brute force lifting (BFL), which attempts to eliminate one variable assignment at a time, and an improved variant that eliminates variables in such a way to highlight the primary events that led to the property falsification. The basic idea of BFL considers the "free" variables of the bug trace, that is, all input variables 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 minimization technique is concerned with the minimization of 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, commonly a fairly resource demanding task. More recent work in [16] 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 in its attempting to remove an assignment by testing a trace obtained with the opposite assignment. However, we apply this

technique to a longer trace obtained with semi-formal methods and we test the alternative assignment via re-simulation. Another technique applied to model checking solutions is by Gastin et al. [7]. 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 [13], 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 variables 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 semi-formal 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 we show in the experimental result section.

## 2.4 Techniques Based on Random Simulation

Techniques that consider a random simulation approach for trace generation have also been explored in the context of hardware verification. One such technique is by Chen et al. [4] 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, it applies one step of forward state traversal [5] 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 variables and (2) makes heavy use of formal state traversal techniques, which are notoriously very demanding in terms of computing power and can usually be applied only to small-size designs.

## 2.5 The Software Verification Domain

The problem of trace minimization has been a focus of critical 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. A fairly popular technique in the software world has been introduced by [10], who proposed the "delta debugging" algorithm that simplifies a complex software trace by extracting the portion of a trace that is relevant to expose 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, a popular web browser. A recent contribution that makes reference to counterexamples found by model checking is by Groce *et al.* [8]. Their solution focuses on minimizing the trace with respect to the primitives available in the language used to describe the system and on trying to highlight the causes of the error in the counterexample, so to produce a simplified trace that is more understandable by a software designer.

# 3. PROPOSED TECHNIQUES

Butramin searches for several possible reductions in bug traces. It first tries to minimize a trace by simulating reduced variants of the trace both in terms of shorter length (clock cycles) and number of assignments to input signals. Once these techniques run out of steam, Butramin applies a series of SAT-BMC refinements. The SAT-based search is limited so that we never unroll the circuit more than a fixed number of cycles which allows us to limit the complexity of the BMC search. We present the following techniques, which are then discussed in detail in the sections below:

1. Cycle removal shortens a bug trace by re-simulating a variant of the trace with one less input vector from the trace.
2. Input event elimination attempts to eliminate events, by re-simulating trace variants which involve fewer input events.
3. Alternative trace finish is exploited when the changes made on the test vectors result in an alternative path to the bug, which is shorter than the original bug trace.
4. State skip identifies all the unique state configurations, in a bug trace. If the same state occurs more than once, it indicates the presence of a loop, and the trace can be reduced.
5. When no improvements can be obtained by the previous techniques, a limited-depth BMC technique is deployed, which attempts to reduce further the trace length by finding shorter paths between two states.

## 3.1 Cycle Removal

Cycle removal is an efficient but aggressive way to reduce the length and the number of input events on a bug trace. It tentatively removes a whole cycle from the bug trace and checks if the bug is still exposed by re-simulating the new trace. If this is the case, the cycle is removed from the trace. This procedure is applied iteratively on all cycles in the trace.

EXAMPLE 2. *Consider the trace of Example 1. During the first step, cycle removal attempts to remove cycle* 1. *If the new trace still exposes the bug, we obtain a shorter bug trace with two cycles and two input events, as shown in Figure 3. Note that it is possible that some input events become redundant because of cycle removal, as it is case for the event on signal c at cycle 2. This is because the previous transition on c was at cycle 1, which has now been removed. After redundant events are eliminated, cycle removal can be applied to cycle* 2 *and* 3*, iteratively.*

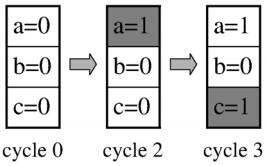


**Figure 3: Cycle removal attempts to eliminate individual trace cycles, generating reduced traces which still expose the bug. This example shows a reduced trace where cycle 1 has been removed.**

## 3.2 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, the input event can be removed. In addition, the event immediately following on the same signal becomes redundant and can be removed as well.

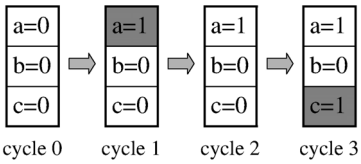


**Figure 4: Input event elimination removes pair of events. In the example, the input events on signal *c* at cycle** 1 **and** 2 **are removed.**

EXAMPLE 3. *Consider once again the trace of Example 1. The result after elimination of input event c at cycle* 1 *is shown in Figure 4. Note that the input event on signal c at cycle* 2 *becomes redundant and it is also eliminated.*

## 3.3 Alternative Trace Finish

An alternative trace finish occurs when a variant trace reaches a state that is different from the final state of the trace, but it also exposes the bug. The alternative state is obviously reached in fewer simulation steps than the original one. This leads to an alternative and shorter trace to the same bug. As shown in Figure 5, if $s_{j_2}$, reached at time $t_2$ by the variant trace, exposes the bug, the new variant trace is substituted for the original one.

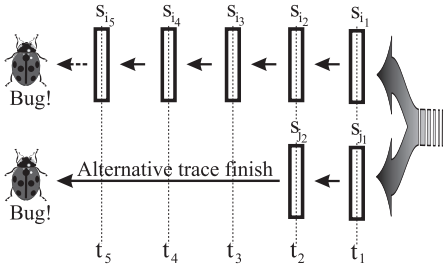


**Figure 5: Alternative trace finish: The variant trace hits the bug at step $t_2$. The new trace replaces the old one, and simulation is stopped.**

## 3.4 State Skip

State skip is exploited when two identical states exist in a bug trace. This happens when there is a sequential loop in the bug 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 Figure 6: 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 $t_5$ on from the original trace, to the prefix extracted from the variant trace up to and including step $t_2$.

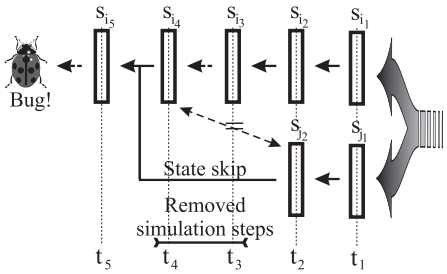


**Figure 6: State skip: If state $s_{j_2} = s_{i_4}$, cycles $t_3$ and $t_4$ can be removed, obtaining a new trace which includes the sequence "... $s_{j_1}$, $s_{j_2}$, $s_{i_5}$, ...".**

## 3.5 BMC-based Reduction

This technique can be used after simulation-based minimization to further reduce the length of the bug trace. Due to 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, also outlined in Figure 7, 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 call $CNF_c$ the CNF formula of the unrolled circuit, 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 that $CNF_{s_i}$ and $CNF_{s_j}$ are equivalent to a partial satisfying assignment for the instance. An example is given in Figure 8.

```
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_si ∧ CNF_sj
6     if (I is satisfiable)
7           return (shortcut s_i → s_j, l steps);
8  }
```

**Figure 7: BMC-based shortcut detection algorithm.**

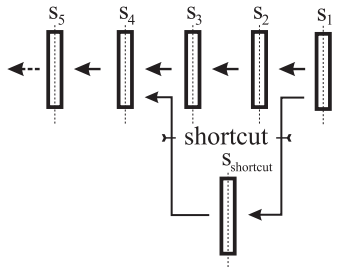


**Figure 8: Our BMC-based method finds a shortcut between state $S_1$ and $S_4$, reducing the overall trace length.**

The algorithm described in Figure 7 is applied iteratively on each pair of states that are $k$ steps apart in the bug trace, and 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 with the BMC-based method and construct the final shortest path from the initial state to the bug state. Figure 9 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 any 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. The single-source shortest path algorithm for directed acyclic graphs is then used to find the shortest path from the initial to the bug state.

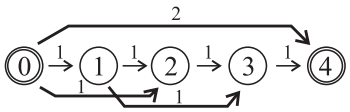

**Figure 9: A shortest-path algorithm is used to find the shortest sequence from the initial state to the bug state. The edges are labeled by the number of cycles needed to go from the source vertex to the sink. The shortest path from state 0 to state 4 in the figure uses 2 cycles.**

The path constructed by this shortest-path algorithm is optimal within the selected window size $m$ and can compensate local optimizations overlooked by simulation-based techniques, which may exist due to their heuristic nature.

# 4. IMPLEMENTATION INSIGHTS

We built a prototype implementation of the techniques described in the previous section 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 at the same time providing good performance. This section discusses some of the insights we gained while constructing a Butramin's prototype.

## 4.1 System Architecture

The architecture of Butramin consists of three primary components: a driver program, a commercial simulation software, and 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) generating the minimizations discussed in the previous sections. The logic simulation software is responsible for simulating test vectors from the driver program and 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 [6] which analyzes the SAT instances generated by converting the unrolled circuits to CNF form using a CNF generator. The system architecture is shown in Figure 10.

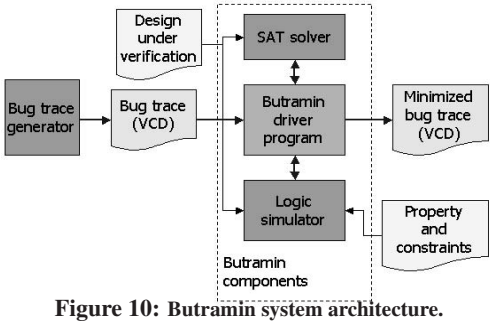


**Figure 10: Butramin system architecture.**

## 4.2 Performance Optimizations

To identify multiple occurrences of a state, all states visited by a trace are hashed and tagged 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: If at any point after the time where the original and the variant traces differ, a variant state matches a state in the original trace, and they are both tagged by the same clock cycle, then we can terminate the variant simulation knowing that the variant trace will hit the bug. We call this an *early exit*. As illustrated in Figure 11, early exit points allow the simulation to terminate immediately. Often state skip optimization leads to early exits as the destination state is already in the trace database.

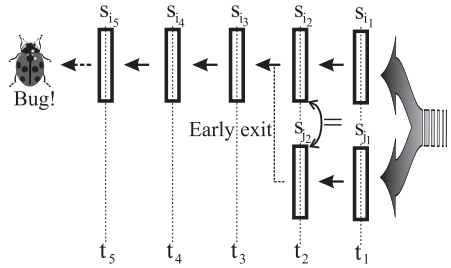


**Figure 11: Early exit. If the current state $s_{j_2}$ matches a state $s_{i_2}$ from the original trace, we can guarantee that the bug will eventually be hit.**

## 4.3 Use Model

To run Butramin, the user must supply four inputs: (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 VCD (Value Change Dump) files, which is a 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. Consequently, when testing subcomponents of a design with constrained inputs, it becomes necessary to validate input sequences 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. For instance, reset and the clock signals could be handle with this approach. For complex sets of constraints, it is possible to convert them into an equivalent circuit block connected to the original design. The block's inputs would be random, independent signals, and the outputs would be legal design's inputs. We deployed a validation technique whereas a set of assumptions monitors the input events generated in the candidate traces that Butramin produces; if a trace invalidates an assumption, than the trace is declared invalid and dropped. For BMC-based reduction, these assumptions are synthesized and included as additional constraints to the problem instance.

Finally, we found that, after applying our minimization techniques, bug traces are usually much shorter. However, many input variables may still be part of the trace, and their relevance in exposing the bug may be uneven - some may be essential, while others are not. Butramin includes an "x_mode" feature for this purpose, where input variables are classified as essential or not, based on a 3-value (0/1/X) simulation analysis. Each input variable in turn is assigned the value X; if the assignment implies a value X on the checker's output, than the variable is tagged essential and the original assignments are restored in the trace. Otherwise the variable is kept at X, so that a designer would not spend effort analyzing that signal's transitions.

# 5. EXPERIMENTAL RESULTS

We tested Butramin by attempting to minimize traces generated by a range of commercial verification tools: a constraint random simulation, a semi-formal verification software, and a semi-formal tool where we specified to put extra effort to generate compact traces. We considered eight benchmarks designs from OpenCores, ISCAS89, and ITC99, and whose characteristics are reported in Table 1. We developed assertions to be falsified, when not available with the design, and we inserted proper bugs to falsify the assertions. Table 2 reports assertions and bugs inserted. Finally, experiments were run on a Sun Blade 1500 (1 GHz UltraSPARC IIIi) workstation running Solaris 9.

| 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 |

**Table 1: Benchmarks characteristics.**

| Circuit | Bug injected | Assertion used |
|---|---|---|
| S38584 | None | Output signals forced to value |
| S15850 | None | Output signals forced to value |
| MULT | AND gate changed with XOR | Compute correct output |
| DES | Complemented output | Timing between receive_valid, output_ready and transmit_valid |
| B15 | None | Output signals forced to value |
| FPU | divide_on_zero conditionally complement | Assert divide_on_zero when divisor=0 |
| ICU | None | Buffer-full condition |
| picoJava | None | Assert SMU's spill and fill |

**Table 2: Bugs injected and assertions for trace generation.**

Our first set of experiments attempts to minimize traces generated by running a semi-formal commercial verification tool with the checkers specified, and subsequently applying only the simulation-based minimization techniques of Butramin. Figures 12 and 13 show the number of cycles and input events removed from the original bug trace. The height of the bars represent the number of cycles or input events in the baseline bug trace, and the patterns show the contribution of the techniques discussed in removing cycles and input events. Note that for all benchmarks we are able to remove the majority of cycles and input events. Table 3 shows the absolute values of cycles and input events removed from each trace and the overall runtime of Butramin using only simulation techniques.

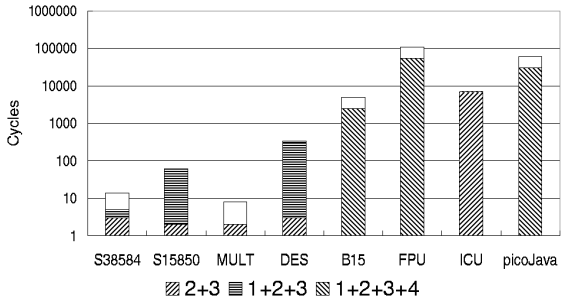


**Figure 12: Number of cycles removed using simulation-based techniques. 1 = cycle removal, 2 = input event elimination, 3 = alternative trace finish, and 4 = state skip + early exit.**

With reference to Figure 12, note that input event elimination can impact the number of cycles removed only when the variant trace can lead to an alternative trace finish. On the other hand, the addition of our cycle removal technique produces most of the reduction. In the graph, we report, for the larger testbenches, only the contribution of cycle removal together with early exit. This is because early exit contributes greatly to the performance of Butramin, and the larger benchmarks timeout (at 40,000 seconds) without this optimization. With reference to Figure 13, the major contributor to the reduction is now input event elimination, while cycle removal has only minimal impact. Once again, some of the larger tests can be attacked within our performance boundaries only after our early exit optimization is plugged in. Overall the simulation-based reduction techniques of Butramin can remove 81% of all cycles and 95% of input events leading to bugs. Designs for which minimal bug traces are inherently difficult to generate using formal methods, such as multipliers, can also be efficiently handled by the techniques proposed in this work. In performing our experiments we noted that the occurrence of early exit situations or state skips are the most advantageous in positively affecting performance. For instance, state skip occurred 4 times in our ICU experiment, and removed 6978 cycles, reducing greatly the overall runtime.

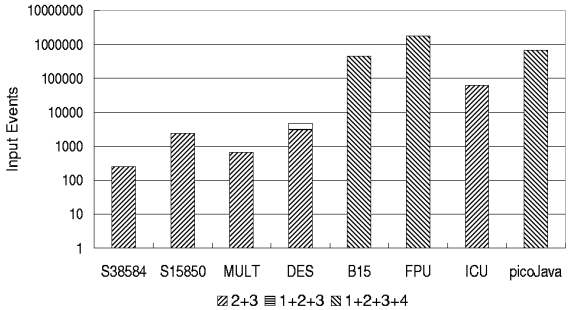


**Figure 13: Number of input events eliminated with simulation-based techniques. 1 = cycle removal, 2 = input event elimination, 3 = alternative trace finish, and 4 = state skip + early exit.**

| Circuit | Cycles | | Input events | | Runtime |
|---|---|---|---|---|---|
| | Original | Removed | Original | Removed | (seconds) |
| S38584 | 14 | 5 | 255 | 253 | 358 |
| S15850 | 62 | 60 | 2412 | 2403 | 67 |
| MULT | 8 | 2 | 660 | 658 | 277 |
| DES | 331 | 309 | 4802 | 3117 | 227 |
| B15 | 2501 | 2489 | 451291 | 451263 | 222 |
| FPU | 53712 | 53706 | 1756431 | 1756414 | 12032 |
| ICU | 7000 | 6991 | 62781 | 62737 | 7 |
| picoJava | 30018 | 30007 | 674521 | 675503 | 34727 |
| Average | 100% | 81.34% | 100% | 95.42% | 5890s |

**Table 3: Cycles and input events removed by simulation techniques of Butramin on traces generated by semi-formal verification.**

Our second set of experiments apply Butramin to a new set of traces, also generated by a semi-formal tool, but this time we specified to the software to dedicate extra effort in generating short traces, by emphasizing the time spent in the formal analysis of the bug. Similarly to Table 3 discussed earlier, Table 4 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 more the benefit from the application of Butramin. Even if the overall impact is reduced, we still observe a 52% reduction in number of cycles and 94% in input events, on average.

| Circuit | Cycles | | Input events | | Runtime |
|---|---|---|---|---|---|
| | Original | Removed | Original | Removed | (seconds) |
| S38584 | 14 | 5 | 255 | 253 | 159 |
| S15850 | 20 | 18 | 671 | 662 | 57 |
| MULT | 7 | 1 | 660 | 658 | 276 |
| DES | 429 | 101 | 4934 | 4927 | 833 |
| B15 | 30 | 18 | 462 | 434 | 87 |
| FPU | 24 | 18 | 800 | 783 | 43 |
| ICU | 25 | 16 | 183 | 134 | 3 |
| picoJava | 28 | 17 | 717 | 699 | 226 |
| Average | 100% | 52.91% | 100% | 94.99% | 210.5s |

**Table 4: Cycles and input events removed by the simulation-based techniques of Butramin on traces generated by the compact-mode semi-formal verification tool.**

The third set of experiments evaluated traces generated by constrained random simulation. Results are summarized in Table 5. Testbenches which timed-out (that is, the random simulator could not generate a trace within 40,000 seconds) are not reported in the table. As expected, Butramin produced the most impact on these set of traces, since they tend to include a lot of redundant behavior. The average reduction is of 99% on both cycles and input events.

| Circuit | Cycles | | Input events | | Runtime |
|---|---|---|---|---|---|
| | Original | Removed | Original | Removed | (seconds) |
| S38584 | 1004 | 995 | 19047 | 19045 | 158 |
| S15850 | 2004 | 2002 | 77456 | 77447 | 67 |
| MULT | 1004 | 998 | 128199 | 128197 | 277 |
| DES | 25329 | 25305 | 667607 | 667597 | 273 |
| FPU | 1046188 | 1046183 | 36125365 | 36125348 | 266121 |
| ICU | 31998 | 31989 | 287770 | 287726 | 11 |
| Average | 100% | 99.71% | 100% | 99.99% | 44484.5s |

**Table 5: Cycles and input events removed by simulation the methods of Butramin on traces generated by constrained random simulation.**

A comparison of Butramin's impact and runtime on the three sets of traces is summarized in Figure 14. It shows that Butramin can effectively reduce all three types of bug traces in 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

semi-formal tool, even if the initial trace is much longer. This is, for instance, the case for S38584, S15850, MULT, DES or ICU. The authors suspect that the reason of this lies in the nature of the traces: Random simulation generated traces 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 visit in a compact-mode generated trace mode are commonly produced by formal engines and can be highly specific, making state-skip a rare event. The case of FPU is relevant in this context: here state-skips do not occur, and the minimization time is proportional to the original trace length. FPU is also an example of the benefits of using Butramin in a verification methodology context. The original trace is a million cycle long and requires 20 minutes to be simulated. However, the post-Butramin trace only requires a few seconds of simulation. The benefits of adding the minimized trace to a regression suite, instead of the original one, are obvious.

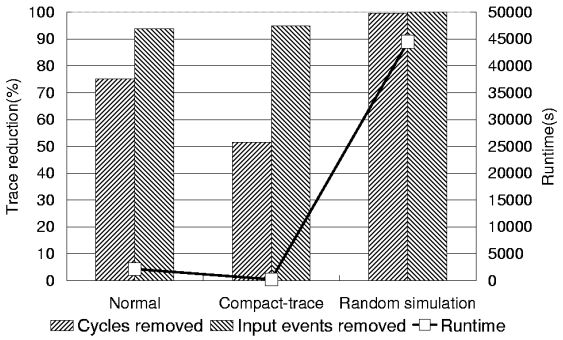


**Figure 14: Comparison of Butramin's impact when applied to traces generated in three different modes. The graph shows the fraction of cycles and input events eliminated and the overall runtime.**

| Circuit | Normal | | | Compact-trace | | | Random | | |
|---|---|---|---|---|---|---|---|---|---|
| | Orig | Rem | Time | Orig | Rem | Time | Orig | Rem | Time |
| S38584 | 9 | 0 | 34 | 9 | 0 | 34 | 9 | 0 | 34 |
| S15850 | 2 | 0 | 0 | 2 | 0 | 0 | 2 | 0 | 0 |
| MULT | 6 | 0 | 11 | 6 | 0 | 11 | 6 | 0 | 11 |
| DES | 22 | 4 | 109 | 328 | 238 | 5548 | 24 | 8 | 1016 |
| B15 | 12 | 0 | 76 | 12 | 0 | 76 | N/A | N/A | N/A |
| FPU | 6 | 0 | 2 | 6 | 0 | 2 | 5 | 0 | 2 |
| ICU | 9 | 0 | 5 | 9 | 0 | 5 | 9 | 0 | 4 |
| picoJava | 11 | 0 | 56 | 11 | 0 | 58 | N/A | N/A | N/A |

**Table 6: Cycles removed by the BMC-based method: DES can be minimized further after Butramin's simulation techniques.**

We applied our BMC-based technique to the traces already minimized by simulation-based methods to evaluate the potential for further minimization. Results are summarized in Table 6, where *Orig* is the original number of cycles in the trace, and *Rem* is the number of cycles removed. We used a maximum windows of 10 cycles ($m = 10$). In two cases random simulation timeout in attempting to generated a bug trace. The main observation that can be made is that simulation-based techniques are very effective in minimizing the bug traces. In one case, DES, the BMC-based technique was able to extract additional minimization opportunities, in particular when the starting trace was longer than average. Potentially, we can repeat the application of simulation-based techniques and BMC-based methods alternating them until convergence, that is, no additional minimization can be extracted.

Finally, we evaluated the "x_mode" feature, described in Section 4.3, and extracted the essential variables from the minimized traces (obtained from our first set of experiments). Table 7 shows that after this *trace explanation* technique is applied, many input variables are removed. Note that the comparison is now between input variables, not input events.

| Circuit | Input Variables | With x_mode |
|---|---|---|
| S38584 | 360 | 2 |
| S15850 | 152 | 4 |
| MULT | 1536 | 1531 |
| DES | 2112 | 1209 |
| B15 | 444 | 25 |
| FPU | 426 | 131 |
| ICU | 261 | 61 |
| picoJava | 572 | 44 |

**Table 7: Input variables removed with trace explanation (x_mode).**

# 6. CONCLUSIONS

This work presented Butramin, 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 removal*, *redundant input event elimination*, *alternative trace finish* and *state skip*. An additional BMC-based minimization 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 is more scalable than formal methods, (3) it can find bug trace reductions in large designs that cannot be handled by purely formal methods, (4) it leverages the existing simulation-based infrastructure, which is currently prevalent in the industry. The latter significantly decreases the barriers to the adoption of Butramin in practical verification of industrial designs.

# 7. REFERENCES


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

[2] Janick Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, 2nd edition, 2003.

[3] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs", *TACAS - LNCS1579*, 193–207, 1999.

[4] Y. A. Chen and F. S. Chen, "Algorithms for Compacting Error Traces", *ASPDAC, Proc. of Asia and South-Pacific Design Automation Conference*, 99–103, January 2003.

[5] O. Coudert, C. Berthet and J. C. Madre, "Verification of synchronous sequential machines based on symbolic execution", In *Automatic Verification Methods for Finite State Systems - LNCS 407*, 1989.

[6] N. Eén and N. Sörensson, "An Extensible SAT-solver", *Theory and Applications of Satisfiability Testing, SAT*, pages 502–518, 2003.

[7] P. Gastin, P. Moro, and M. Zeitoun, "Minimization of Counterexamples in SPIN", *SPIN - LNCS2989*, pages 92–108, 2004.

[8] A. Groce and D. Kroening, "Making the Most of BMC Counterexamples", *Workshop on BMC*, 71–84, 2004.

[9] S. Hazelhurst, O. Weissberg, G. Kamhi, and L. Fix. A hybrid verification approach: Getting deep into the design. In *DAC, Proc. Design Automation Conference*, pages 111–116, 2002.

[10] R. Hildebrandt and A. Zeller, "Simplifying failure-inducing input", *Int. Symposium on Software Testing and Analysis*, 135-145, 2000.

[11] 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 *ICCAD*, 120–126, 2000.

[12] A. Hu. Formal hardware verification with BDDs: An introduction. In *Pacific Rim Conf. on C.C.S.P. (PACRIM)*, 677–682, 1997.

[13] H. Jin, K. Ravi, and F. Somenzi, "Fate and free will in error traces", In *TACAS'02 - LNCS 2280*, pages 445–459, 2002.

[14] Prakash Rashinkar, Peter Paterson, and Leena Singh. *System-on-a-chip Verification: Methodology and Techniques*. Kluwer Academic Publishers, 2002.

[15] K. Ravi and F. Somenzi, "Minimal Satisfying Assignments for Bounded Model Checking", *TACAS'04 - LNCS 2988*, 31–45, 2004.

[16] S. Shen, Y. Qin, and S. Li, "A Fast Counterexample Minimization Approach with Refutation Analysis and Incremental SAT", *Proc. ASP-DAC*, 451-454, 2005.