Testing versus Static Analysis of Maximum Stack Size

Mahdi Eslamimehr
mahdi@cs.ucla.edu
UCLA, University of California, Los Angeles

Jens Palsberg
palsberg@ucla.edu
UCLA, University of California, Los Angeles

***Abstract*—For event-driven software on resource-constrained devices, estimates of the maximum stack size can be of paramount importance. For example, a poor estimate led to software failure and closure of a German railway station in 1995. Static analysis may produce a safe estimate but how good is it? In this paper we use testing to evaluate the state-of-the-art static analysis of maximum stack size for event-driven assembly code. First we note that the state-of-the-art testing approach achieves a maximum stack size that is only 67 percent of that achieved by static analysis. Then we present better testing approaches and use them to demonstrate that the static analysis is near optimal for our benchmarks. Our first testing approach achieves a maximum stack size that on average is within 99 percent of that achieved by static analysis, while our second approach achieves 94 percent and is two orders of magnitude faster. Our results show that the state-of-the-art static analysis produces excellent estimates of maximum stack size.**

I. Introduction

Event-driven programming has found pervasive acceptance, from high-performance servers to embedded systems, as an efficient method for interacting with a complex world. However, loose coupling of event handlers obscures control flow and makes dependencies hard to detect, leading to subtle bugs. Event-driven software on resource-constrained devices has the additional challenge that if swamped with events, the software may run out of memory. Thus, estimates of the maximum stack size can be of paramount importance [14].

For example, a poor estimate led to software failure and closure of a German railway station in 1995. Specifically, the designers had estimated that 3,500 bytes of stack space would be sufficient but actually 4,000 bytes were needed. As a result, the railroad station's computer experienced stack overflow and failed [8].

Intuitively, the maximum stack size during a run is the high water mark or the peak value of the stack pointer. We focus on a much-studied question about stack space for event-driven software:

Q: what is the maximum stack size across all inputs?

A programmer can use the answer to ensure that sufficient stack memory is available for a particular application. Additionally, the programmer can use the smallest or cheapest memory unit that has sufficient capacity and thereby help control size and cost. This is welcome for many event-driven applications that run in embedded systems for which physical size and hardware cost are major concerns.

Like most other interesting questions about programs, the above question is undecidable. Ideally, we would answer the above question by running the program on all inputs, possibly indefinitely in case of nontermination. Each run has a maximum stack size and we can then take the maximum across all runs to get the answer to the question. The result is the *true* maximum stack size.

The above question can be answered approximately by testing (running the program) and by static analysis (analyzing the program text). A testing approach underestimates the true answer by finding the maximum stack size for some runs on some inputs. A static analysis overestimates the ideal answer by working with conservative abstractions of program constructs and values. In slogan form, we have the following relationships for maximum stack size:

tested $\leq$ *true* $\leq$ *static*

How close are *tested* and *static*? In some situations no nontrivial sound static analysis exists, and we have only the trivial sound static analysis that says that the stack is unbounded. A typical such scenario is an embedded system for which some of the event-driven software is written in assembly code. The assembly code usually contains instructions that add or subtract from the stack pointer, to enable the stack to shrink or grow. Can current nontrivial sound static analyses handle such instructions? The answer is *Yes* if the instructions add or subtract *constants*; while the answer is *No* if the instructions add or subtract the contents of a *register*. If no nontrivial sound static analysis exists, then a programmer must use the best testing approach, and perhaps take a chance with an unsound static analysis. Such techniques are inherently unsafe and a standard engineering solution is to over-provision: if the testing approach estimates the maximum stack size to be n, then go with memory of size $2n$, for example, though even $2n$ may be insufficient.

If a sound static analysis exists, then we can use it to safely allocate the estimated amount of memory and be sure that no stack overflow will occur. Ideally we can find an *optimal* static analysis that always produces the true maximum stack size. However, static analysis must terminate, including for nonterminating programs, so usually static analysis is forced to be conservative and nonoptimal. For maximum stack size of event-driven software, the state-of-the-art static analysis was presented in [12], [21] (see also [2], [9], [20]) and has been implemented in multiple tools. In this paper we address the following question.

Q: how good is the state-of-the-art static analysis of maximum stack size?

We use testing to answer the above question. We have done an experiment with the state-of-the-art testing approach [19] (see also [9]) on benchmarks that are event-driven assembly code programs. In those benchmarks, all arithmetic on the stack pointer either adds or subtracts constants, according to our manual inspection, so the static analysis is sound, we believe. We found a big gap between the estimates: the testing approach achieves a maximum stack size that on average is only 67 percent of that achieved by static analysis. Our benchmark suite consists of software for sensor nodes and proved to be a major challenge for the testing approach. For a different benchmark suite, Regehr [19] found that testing and static analysis are much closer.

Our experiment raises a classical question that arises for a variety of problems that can be addressed with both testing and static analysis. *Is the gap mostly due to weak testing or overly conservative static analysis?* The answer is that better testing is possible and that the static analysis is near optimal for our benchmarks. We make those points by presenting two new testing approaches that almost match the static analysis. The first approach is called DTall (Directed Testing all) and achieves a maximum stack size that on average is within

99 percent of that achieved by static analysis. The second approach is called VICE (Virgil Integrated Concolic Engine) and achieves a maximum stack size that on average is within 94 percent of that achieved by static analysis. VICE is two orders of magnitude faster than DTall. Our results show that the state-of-the-art static analysis produces excellent estimates of maximum stack size. Additionally, our results show that VICE is useful for practical testing: VICE is faster and gives better branch coverage than the previous state-of-the-art testing approach.

The rest of this paper. In Section II we discuss event sequences, in Section III we introduce seven testing approaches, in Section IV we give an example of how VICE works, in Section V we present VICE in detail, and in Section VI we show our experimental results.

II. Event Sequences

The classical notion of a program first consumes an input, then computes, and finally produces an output. In contrast, an event-based program receives its input via events *during* the program execution. The task of the event-based program is to process those events.

For example, our benchmarks run on sensor nodes (Berkeley Motes) and receive events that are generated by devices that are connected to the CPU. Among those devices are a timer, an analog-to-digital converter (ADC), a universal synchronous asynchronous receiver/transmitter (USART) [5], and a serial peripheral interface bus (SPI) [4]. The sensor node can use the timer to wake itself up periodically, use the ADC to convert sensor data to digital form, use the USART for serial communication with terminals, and use the SPI to communicate on a synchronous serial data link with external devices in master or slave mode.

Event-based programs such as sensor-network software are usually designed to run indefinitely (or until the battery dies). Thus, events can keep coming. Notice though that a finite test run consumes only a finite number of events.

Each event consists of a name and a value. The name specifies the source of the event and also the event handler that will process the event. The value is input to the program.

From the program's viewpoint, consecutive events have a *wait time* between them. This wait time can be completely arbitrary and depend on uncoordinated devices beyond the programs control. However, for a particular run we can record both the events and the wait times. Or, for the purpose of planning a test run, we can first *generate* an event sequence and then use that to test the program.

In this paper, we represent an event sequence as a sequence of triples:

(event name, event value, wait time)

The idea is to wait the number of milliseconds specified by *wait time* and then fire an event called *event name* and paired with *event value*.

For example, here is our representation of an event sequence with four events:

$$\begin{aligned}[\quad & (\texttt{main}, 673, 100), (\texttt{m\_intr}, -86347, 200), \\ & (\texttt{main}, -991, 400), (\texttt{m\_intr}, 34, 800) \quad]\end{aligned}$$

The first event (`main`, 673) will occur after 100 milliseconds, the second event (`m_intr`, -86347) will occur after 300 milliseconds, the third event (`main`, -991) will occur after 700 milliseconds, and the fourth event (`m_intr`, 34) will occur after 1,500 milliseconds.

We will evaluate the state-of-the-art static analysis of maximum stack size by finding an event sequence that achieves a large maximum stack size. For creating a suite of candidate event sequences, a designer must decide on the number of event sequences, the number of events in each event sequence, the event names, the event values, and the wait times.

III. Seven Testing Approaches

We now present seven testing approaches that all automatically test event-driven software without a human in the loop. Testing approaches 1–2 are from previous work [9], [19], while 3–7 are new.

How to determine the number of events in each event sequence. Our benchmarks work with 2–5 event handlers. For simplicity we want every event sequence for every benchmark to have the same number of events. We determined the number of events via the following preliminary experiment that anyone can repeat for any benchmark suite. First we noted that the number of events for our benchmark suite should be at least 5 such that we can hope to exercise every handler during a single run. Second we observed that more events may exercise longer program paths. The question is: when does an increase of the number of events begin to produce diminishing returns? We use testing approach 1 (see below for details) to run experiments with different numbers of events in each event sequence. We doubled the number of events, doubled it again, and so on, until we saw no major improvement in maximum stack size. We found that 40 events in each event sequence appear to be a good number for our benchmarks so all our experiments use event sequences with 40 events.

Now we must generate event sequences that each contains 40 event names, 40 event values, and 40 wait times.

How to determine samples of wait times. Four of the testing approaches use *samples* of the wait times. We chose to fix *three* different samples and use them across all those four testing approaches. The number *three* is somewhat arbitrary; we wanted a number greater than one to give diversity in the experiments yet small enough that our experiments could finish in a reasonable time. We determined the three particular samples via the following preliminary experiment that anyone can repeat for any benchmark suite. For each benchmark we ran each event handler in isolation to determine the worst-case time to execute any handler alone (in any of the benchmarks). That worst-case time is the longest time any single handler may be able to block other handlers from running. Once we had that number, we divided the time interval from 0 to that number into three equally sized intervals. Finally, from each of those three intervals we sampled a wait time using a uniform distribution.

We compare seven testing approaches:

approach	#	event names	event values	wait times
	1	Sample	Sample	Sample
	2	GA	GA	Sample
	3	All	Sample	All
	4	Sample	DT	All
VICE:	5	SA-Tree	DT	Sample
	6	All	DT	Sample
DTall:	7	All	DT	All

We will use the numbering (1–7) of the approaches throughout the paper. Those seven approaches span a wide variety of techniques that one might try. Ultimately, testing approach 7 is the best we are able to do given a reasonable amount of time. Testing approaches 4–6 can be understood as restrictions of testing approach 7.

Testing approach 1 is a form of random testing that tries 3,000 event sequences based on randomly chosen samples of event names and event values, and the three particular samples of wait times that we found as discussed above. The number 3,000 is somewhat arbitrary; we wanted a number that was large enough to produce good results yet small enough that our experiments could finish in a reasonable time. The experiments justified the use of three wait times because, somewhat surprisingly, we encountered some cases where a *longer* wait time leads to a *larger* stack size. This phenomenon stems from situations such as the following. Suppose we have reached a state S of the computation where a run of the handler for event B would reach a maximally large stack. Suppose also that in state S, events A and B have fired and the handlers for A and B are enabled. The hardware *arbits deterministically* which handler will run; and let us assume that the hardware chooses A. So, B will run later; possibly in a state with a smaller stack than state S so the run of B will fail to reach a maximally large stack. Can we get the hardware to choose B instead of A? One potential answer is: increase the wait time such that A is disabled in state S. Hence, a longer wait time has the potential to produce a larger stack size. Testing approach 1 is our base line; the other six approaches do better.

Testing approach 2 is a genetic algorithm (GA) [9], [19] that uses 20 generations of each 50 event sequences, for each of the three chosen samples of wait times. We chose 20 generations and 50 event sequences because the total number of runs would be $20 \times 50 \times 3 = 3,000$, which matches the number of runs with testing approach 1. The first generation has a randomly chosen sample of event names and event values. Each later generation hopes to improve on the previous one by swapping and mutating the event names and event values. Specifically we map a generation to a new generation in the following way. We first do 50 swaps of subsequences of length 25 among the event sequences. We then mutate one event in each event sequence; each mutation replaces the event name with a randomly chosen event name, and it replaces the event value with a randomly chosen event value. The fitness function is the maximum stack size observed during a run.

Testing approach 3 is similar to testing approach 1 in that it samples the event values, but also goes much further in that it tries all combinations of i) all sequences (of length 40) of event names, and ii) all integer wait times in a wide interval. The interval of wait times is handler specific and defined as follows. The lower bound of the interval is 8 milliseconds; we found that going lower often caused testing to run out memory. The upper bound of the interval is the worst-case time to execute the handler for the previous event in isolation. Note that if the upper bound is high, trying all integer wait times in the interval may lead to a lengthy testing effort. In such a case, we recommend the use of a large number of samples drawn from a uniform distribution across the interval.

Our preliminary experiment, mentioned in Section 1, tried testing approaches 1 and 2. When we found that the results from those approaches are suboptimal, we tried the much slower testing approach 3 which gave just a small improvement. We concluded that we need a better approach to generate event values.

Testing approaches 4–7 all use directed testing (DT) to generate event values. Directed testing [17], [18] is based on concolic execution [10], [11], [22], [23], which is a technique related to model checking [6], theorem proving [13], symbolic execution [15], and run-time monitoring and testing [7]. The idea of directed testing is to execute the code with concrete and symbolic values simultaneously, and to use the result to generate new inputs for another execution. The term concolic combines the words "concrete" and "symbolic". In each round, the symbolic part of an execution collects constraints from each condition on the control-flow. Those constraints represent the executed control-flow path and they have *the concrete input to the run* as one of the possible solutions. We can now easily construct constraints for a different potential control-flow path by taking a prefix of the collected constraints and *negating* the last constraint from the prefix. Concolic execution will submit those new constraints to a constraint solver, and if they are solvable, the concolic execution will use the solution as concrete input to a new round of execution. In the first round, the input is chosen randomly. Experience shows that concolic execution achieves better branch coverage with fewer test cases than testing with random inputs.

Testing approach 4 samples the event names, does DT to determine event values, and tries all integer wait times in a wide interval. In essence, testing approach 4 is standard DT applied to many combinations of event names and wait times. This gives a significant improvement over testing approach 3, yet falls well short of the results from static analysis. We conclude that we must do better to generate challenging event names.

Testing approach 5 is the one we call VICE (Virgil Integrated Concolic Engine). Compared to testing approach 4, VICE handles event names more accurately and wait times less accurately. Specifically, VICE uses a novel technique called SA-Tree to generate event names, uses DT to determine event values and tries three samples of wait times. VICE is the fastest of the seven approaches and gives a good trade-off between testing time and quality of the results. Additionally, VICE is useful for practical testing: we will show that VICE gives better branch coverage than testing approaches 1–4.

Testing approach 6 does more than testing approach 5 by trying *all* sequences (of length 40) of event names, in addition to use DT to determine event values and to try three samples of wait times. However, the exhaustive coverage of the sequences of event names cannot improve on VICE because SA-Tree generates all event sequences that matter. We have included testing approach 6 in our experiments to demonstrate the large impact SA-Tree has on static analysis time.

Testing approach 7 is the one we call DTall and is both the slowest and the closest to optimal. DTall uses DT to determine event values and it tries all combinations of i) all sequences (of length 40) of event names, and ii) all integer wait times in a wide interval. DTall comes close to the results from static analysis and demonstrates that the best known static analysis is near optimal for our benchmarks.

IV. VICE Example

Overview. We now explain the initial portion of a run of VICE on the example program in Figure 1, which is a simplified version of one of our benchmarks. The program has four if-statements and two event handlers: `main` and `m_intr`. Our description of the example run is high level and ignores some details. VICE proceeds in phases that each consists of multiple rounds. We will explain just one phase with five rounds. During a phase, the event names stay unchanged in each round; the example uses the sequence of event names: (`main`, `m_intr`, `main`, `m_intr`). So, all event sequences in the example will have length four.

Round one. In the first round, the event sequence is random so we might begin with this event sequence:

$$[\ (\texttt{main}, 673), (\texttt{m\_intr}, -86347), (\texttt{main}, -991), (\texttt{m\_intr}, 34)\ ]$$

(We avoid discussion of the wait times in this section, for simplicity.) The concolic execution will fire the first event and now let us say that before `main` calls `transmitValue` in line 09, the execution fires the second event and interrupts `main`. We now have two event handlers on the stack. Next `m_intr` calls `transmitValue` in line 24 and we collect the constraint

$$y = a$$

```
00 program TestProgram {
01   entrypoint main = TestMe.main;
02   entrypoint timer_comp = testMe.m_intr;
03 }
04
05 component TestMe {
06   field sending:bool = false;
07   method main(x:int):void {
08     computeValue();
09     transmitValue(x);
00   }
11   method computeValue():void { ... }
12   method transmitValue(a:int):void {
13     local buffer:int, b:int;
14     b = rand(100);
15     local bufferSize:int = (a+b) * 256;
16     if (atomic_swap(sending,true)) return;
17     if (a > 2000) {
18        buffer = checks(a,b);
19        sending = false;
20        return;
21     }
22   }
23   method m_intr(y:int):void {
24     transmitValue(y);
25   }
26   method checks(s:int, t:int):int {
27     if (s==5000) {
28        t=square(s);
29        if (s<-5) return square(-s);
30        else return 0;
31     }
32     return 1;
33   }
34   method square(root:int):int { ... }
35   method rand(seed:int):int { ... }
36   method atomic_swap(cur:bool,status:bool)
37                       :bool { ... }
38 }
```

Fig. 1. Example program.

that relates the actual parameter (line 24) to the formal parameter (line 12). We use y to denote a symbolic variable related to the program variable `y`, and similarly for a and `a`. In the body of `transmitValue` in line 16, let us assume that the condition `atomic_swap(sending,true)` returns false. We collect constraints from the conditions of the if-statements provided that they are arithmetic or logical equations. So we collect no constraints from the if-statement in line 16, while we do collect the constraint

$$a > 2000$$

from the if-statement in line 17 because $(a > 2000)$ failed: `a` has the value -86347 so the execution does not take the branch that requires `a > 2000`. Now the second event handler terminates and we return to the first event handler. That event handler eventually calls `transmitValue` in line 09 and we collect the constraint

$$x = a$$

In the body of `transmitValue` we collect the same constraints as before and again the execution does not take the branch that requires `a > 2000` because `a` has the value 673. Now the first event handler terminates. Later the execution fires the third and fourth events, and we can see that no new branches will be executed while handling those events.

During the first round of concolic execution, the maximum stack size occurred when we had two event handlers on the stack and `m_intr` called `transmitValue` which, in turn, called `atomic_swap`. The execution took the same branch each time in lines 16 and 17, while it never reached line 27 or 29.

After completion of the first round, we solve the three collected constraints above, pick a solution at random, and use it to help generate another event sequence. We use the four event-handler names from before and pair each of them up with values from the picked solution to the constraints. For example, we may get the event sequence:

$$[\,(\texttt{main}, 2833), (\texttt{m\_intr}, 4756), (\texttt{main}, 77733), (\texttt{m\_intr}, 6500)\,]$$

Round two. In the second round of concolic execution, let us assume that the firing of events proceeds like in the first round. The execution will four times reach line 17 and find each time that the condition `a > 2000` is satisfied. So, the execution will exercise a new branch and eventually call `checks` in line 18 and from the call collect the constraint

$$a = s \wedge b = t$$

In the body of `checks` we will in each of the four cases find that `s` is different from 5000 so also in this round the execution does not reach line 29. Along the way, we collect the constraint

$$s = 5000$$

in line 27.

During the second round of concolic execution, the maximum stack size occurred when the stack contained two event handlers and stack frames for `transmitValue` and `checks`. That maximum stack size is similar to the maximum stack size encountered in the first round.

After completion of the second round, we find that the above constraints have a unique solution ($y = x = a = s = 5000$) that we use to help generate another event sequence. And again, we use the four event-handler names from before and pair each of them up with values from the solution to the constraints. For example, we may get the event sequence:

$$[\,(\texttt{main}, 5000), (\texttt{m\_intr}, 5000), (\texttt{main}, 5000), (\texttt{m\_intr}, 5000)\,]$$

Round three. In the third round of concolic execution, let us assume that the firing of events proceeds like in the second round. The execution will four times reach line 27 and find each time that the condition `s==5000` is satisfied. So, the execution will exercise a new branch and eventually call `square` from which we collect the constraint $s = root$ Then the execution will reach line 29 and find that the condition `s<-5` is unsatisfied. By the way, notice that the chance of reaching line 28 with event sequences generated randomly or by genetic algorithms is vanishingly small. We will collect the constraint $s < -5$ During the third round of concolic execution, the maximum stack size occurred when the stack contained two event handlers and stack frames for the methods `transmitValue`, `checks`, and `square`, which is the highest so far.

Rounds four and five. After completion of the third round, we find that the collected constraints are unsolvable (because we have both $s == 5000$ and $s < -5$). We then repeatedly remove the last added constraint until we find that the remaining constraints are solvable, and then we proceed as before. We note that the third round has already achieved as much as one can do for the example program. VICE continues with a fourth and a fifth round until it notices that in two consecutive rounds, no improvements were achieved for the maximum stack size. At that point, the phase of the concolic execution terminates.

V. VICE Description

VICE uses six data types and six tools, see Figure 2.

Types. Each program that we test is a VirgilProgram, that is, a program in the Virgil programming language [24], [26], which is an object-oriented language for resource-constrained devices. Virgil is a full-fledged language with classes, objects, loops, recursion, etc.

We compile Virgil programs to machineCode, that is, AVR assembly code. The key input to each execution is an eventSequence, which is a list of triples, where each triple consists of an event name (an identifier), an event value (an int), and a wait time (an int that measures milliseconds). Each of the constraint is a Virgil arithmetic or logical expression. For our benchmarks, we found no need to use other forms of constraints; arithmetic or logical constraints are sufficient for our testing approaches to almost match the static analysis. We leave to future work to investigate whether other benchmarks require use of other forms of constraints to almost match the static analysis.

A prefixTree is a prefix-tree of sequences of event names.

Tools. The tool concolic is a concolic execution engine that executes a Virgil program while firing events from an event sequence, with the specified wait time between consecutive events. The result of a run of concolic is a constraint and the branch coverage that was recorded. We implemented concolic on top of an existing Virgil interpreter. The concolic execution engine works with *concolic values*, that is, a pair of a concrete value and a constraint.

The tool compiler is an open-source Virgil compiler [24] that generates AVR assembly code.

The tool avrora is an open-source simulator for AVR assembly code [25] that executes an AVR assembly code program while firing events from an event sequence, with the specified wait time between consecutive events. The result of a run of avrora is the maximum stack size that was recorded. A run of avrora is deterministic, hence reproducible. Specifically, avrora measures time in terms of machine cycles and we use the wait times to determine the exact machine cycle at which to fire an event. Additionally, avrora implements all aspects of the hardware, including the "breaking of a tie" that happens when two events have fired and both handlers are enabled. So, any two runs of avrora on a benchmark and an event sequence always proceed in exactly the same way.

The tool SA-Tree-Gen applies a static analysis to a Virgil program [9]. The static analysis determines conservatively, for each program point, which event handlers are enabled. The result of a run of SA-Tree-Gen is a prefixTree called the SA-Tree that represents the static information as a collection of sequences of event names. According to the static analysis, each sequence of event names can be the basis for an event sequence for which each event will be handled. The SA-Tree avoids names of events that have no chance of being handled because the corresponding event handler is disabled. We can compare the generated SA-Tree with a *full* prefix-tree that represents all possible sequences of event names (up to a given length). For each of our benchmarks, the SA-Tree is a much pruned version of the full tree. Testing approach 6 explores the full prefix-tree.

The tool random takes a nameSequence and a wait time as input and produces an event sequence based on the input nameSequence, with event values generated according to an exponential distribution, and with each wait time equal to the input wait time. The tool generator takes a nameSequence, a wait time, and a constraint, and generates an event sequence. The generator uses the open-source constraint solver Choco [1], [16] to solve the constraint. Notice that we generate event sequences based on source-level information and use them to test code at the assembly level.

```
int VICE(VirgilProgram $p$, int $waitTime$) {
    prefixTree $tree$ = SA-Tree-Gen($p$)
    machineCode $code$ = compiler($p$)
    int $maxStack = 0$
    **for each** nameSequence $ns \in tree$ **do** {
        int $noChange = 0$
        float $branchCoverage = 0$
        eventSequence $seq$ = random($ns$, $waitTime$)
        **while** $(noChange < 2)$ {
            int $ms$ = avrora($code$, $seq$)
            (constraint × float) $(c, bc)$ = concolic($p$, $seq$)
            $seq$ = generator($ns$, $waitTime$, $c$)
            **if** $((ms > maxStack) \vee (bc > branchCoverage))$
            **then** { $maxStack = ms$; $branchCoverage = bc$;
                    $noChange = 0$ }
            **else** { $noChange = noChange + 1$ }
        }
    }
    **return** $maxStack$
}
```

Approach. Above is pseudo-code for VICE; Figure 3 illustrates how VICE works. The input to VICE is a Virgil program and a wait time. VICE proceeds in phases that each consists of multiple rounds. Each phase focuses on one nameSequence in the SA-Tree for the Virgil program. In each phase, VICE iterates until two consecutive rounds found no improvement to the maximum stack size or the branch coverage. In each round VICE updates the variable $noChange$ to count how many recent rounds had no change. The condition $noChange < 2$ tells when to terminate a phase. The variable $maxStack$ contains the maximum stack size found so far, the variable $branchCoverage$ contains the branch coverage found so far, and the variable seq holds the current event sequence, which is based on the chosen nameSequence and the input wait time, and which initially has event values chosen randomly.

We compile each Virgil benchmark program to AVR assembly code. In each round, the algorithm executes both avrora on the assembly code and concolic on the Virgil program to get a new maximum stack size, a new constraint, and a new measure of the branch coverage.

The generator uses a constraint solver to find new event values for an event sequence that otherwise has the same event names and wait times as all other event sequences in the current phase.

A worse alternative. VICE measures maximum stack size at the assembly level in every round of concolic execution. We have experimented with an alternative approach that measures maximum stack size at the *source* level, and only after a completed run measures the maximum stack size at the assembly level for the most challenging event sequence. The alternative approach is faster because it uses the assembly-level simulator just once. However, the results are considerably worse because the source-level stack-size estimates are imprecise.

VI. Experimental Results

We compare a static analysis and the seven testing approaches listed in Section III. We wrote all the implementations in Java and ran them on Sun Java2 SDK 1.5 on on a 2.8 GHz iMac. Most of the runs used less than 60 MB. We implemented the genetic

Types:	VirgilProgram	=	see http://compilers.cs.ucla.edu/virgil
	machineCode	=	AVR assembly code
	eventSequence	=	(identifier × int × int)list
	constraint	=	a Virgil arithmetic or logical expression
	nameSequence	=	(identifier)list
	prefixTree	=	a prefix-tree of elements of nameSequence
Tools:	concolic	:	(VirgilProgram × eventSequence) → (constraint × float)
	compiler	:	VirgilProgram → machineCode
	avrora	:	machineCode × eventSequence → int
	SA-Tree-Gen	:	VirgilProgram → prefixTree
	random	:	nameSequence × int → eventSequence
	generator	:	nameSequence × int × constraint → eventSequence

Fig. 2. VICE.

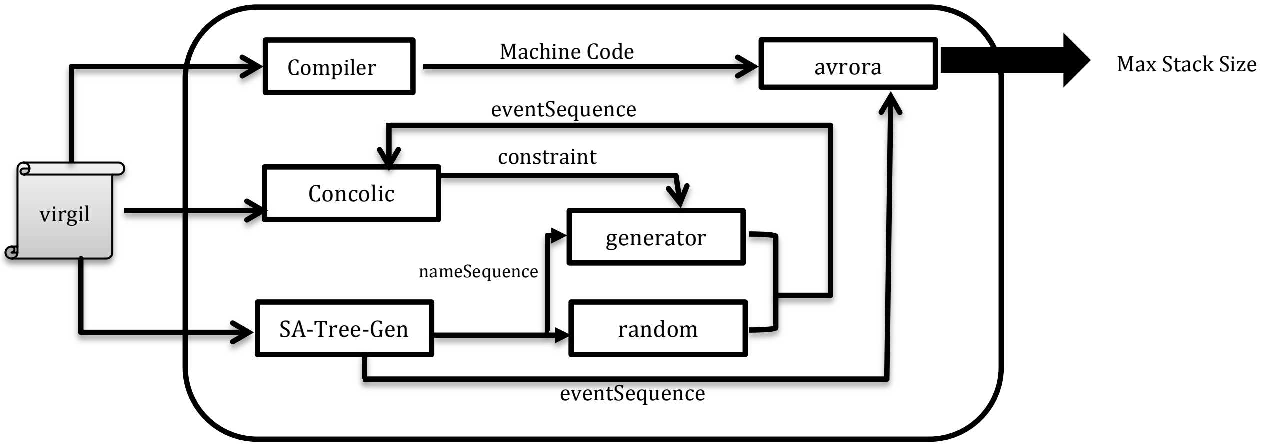

Fig. 3. Illustration of how VICE works.

algorithm on top of the Java Genetic Algorithm Library (JGAL) from http://jgal.sourceforge.net. For testing approaches 1, 2, 5, 6, our samples of the wait times are 153 ms, 327 ms, and 594 ms. For each testing approach we find the maximum stack size of a program in the same way: we first compile the program and then use Avrora to run the assembly code and return the maximum stack size.

A. Benchmarks

The following table shows some statistics about our seven benchmarks, including the number of lines of Virgil code and also the number of lines of code after translation to C, which is a step on the way in the translation to AVR assembly code. The table also shows the number of event handlers.

Benchmark	LOC (Virgil)	LOC (C)	no. of handlers
TestCon1	329	461	4
TestCon2	347	528	3
StackTest1	293	513	2
StackTest2	251	483	2
TestUSART	1,226	1,737	5
TestSPI	859	1,109	3
TestADC	605	1,055	4

We use four microbenchmarks and three benchmarks that test device drivers for Berkeley Motes. We designed the microbenchmarks testCon1 and testCon2 to test VICE's power to explore different execution paths. These programs have many complex numerical expressions and nested conditional statements and loops. TestCon1 has four event handlers, all without parameters, more than 300 LOC and its nesting depth of control structures is 11. TestCon2 has 3 event handlers each of which has 8 formal parameters, almost 350 LOC, and 37 complex numerical expressions.

The microbenchmark StackTest1 is a more complete version of the example program in Figure 1 and includes nested function calls, unreachable code, and atomic structures. StackTest2 consists of nested functions of depth 23.

The TestUSART benchmark tests the USART driver; the TestSPI benchmark tests the SPI driver; and the TestADC benchmark tests the ADC driver.

Previous work [12] has shown that even for programs with a bounded stack, the maximum stack size can grow exponentially in the number of event handlers. The number of handlers in our benchmarks, namely 2–5, is typical of event-driven AVR applications that we have found.

In summary, our benchmarks are nontrivial and turn out to be a major challenge for the previous-best testing approaches.

B. Measurements

Figure 4 shows a plot of the mean percentages in Figures 5 and 6; note that the x-axis uses a log-scale.

Figure 5 shows the maximum stack sizes found by the seven testing approaches (numbered 1–7) and by a static analysis of maximum stack size (labeled SA) that comes with the Avrora distribution. Note that the static analysis guarantees an upper bound on the stack size for every benchmark. This implies that even if each device that generates events should malfunction and generate an event every millisecond,

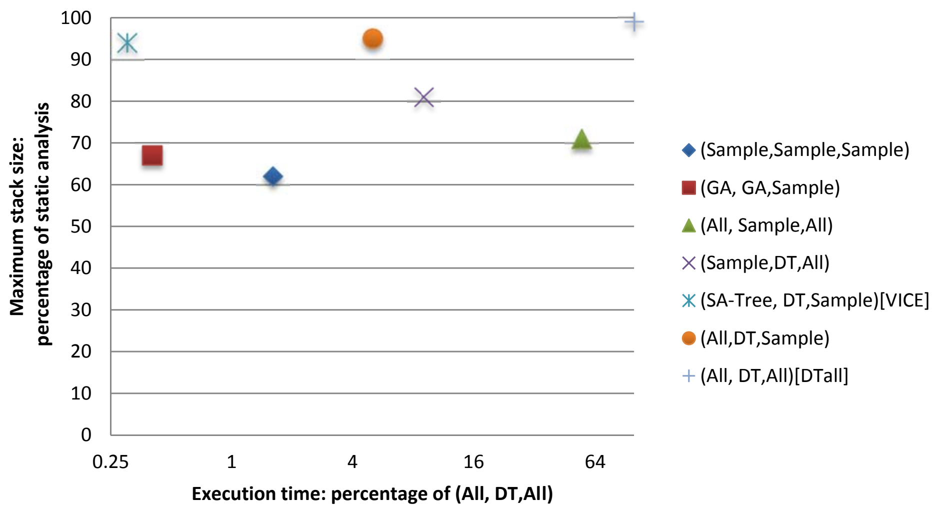

Fig. 4. Comparison of seven testing approaches.

Benchmark	1	2	3	4	5	6	7	SA
TestCon1	318	441	417	455	505	506	511	516
TestCon2	366	612	798	703	846	866	882	894
StackTest1	421	353	318	619	703	749	958	979
StackTest2	353	324	390	420	564	564	564	566
TestUSART	459	481	472	525	664	664	664	665
TestSPI	490	350	481	490	518	522	529	533
TestADC	247	306	283	302	306	306	308	310
% of SA	62	67	71	81	94	95	99	100

Fig. 5. Maximum stack sizes in bytes. The last line gives a geometric mean.

Benchmark	1	2	3	4	5	6	7	SA
TestCon1	7.2	8.83	281	38	1.53	16	439	0.10
TestCon2	10.1	3.11	173	29	2.48	45	381	0.12
StackTest1	12.9	2.55	179	23	0.56	26	307	0.05
StackTest2	2.9	2.29	165	72	4.13	56	266	0.05
TestUSART	7.4	3.05	204	43	1.18	16	452	0.32
TestSPI	4.0	3.11	197	26	0.79	9	393	0.15
TestADC	3.0	1.37	289	33	0.45	6	444	0.13
% of (7)	1.6	0.4	55	9	0.3	5	100	0.03

Fig. 6. Execution time in minutes. The last line gives a geometric mean.

we can be sure that the stack is bounded by the value given by the static analysis.

Figure 6 shows the timings of the testing runs and the timings of running the static analysis. All time measurements are in minutes and are averages of 10 runs after some warm-up runs to fill the caches.

Figure 7 shows the branch coverage that each testing approach achieved.

In Figure 5 the last line gives a geometric mean for each testing approach. The mean is taken over the fractions of the maximum stack size found by the testing approach and the maximum stack size found by static analysis. For example, for testing approach 1, we take the geometric mean of these fractions:

$$\frac{318}{516}\ \frac{366}{894}\ \frac{421}{979}\ \frac{353}{566}\ \frac{459}{665}\ \frac{490}{533}\ \frac{247}{310}$$

Benchmark	1	2	3	4	5	6	7
TestCon1	23	56	61	72	92	93	94
TestCon2	21	60	78	78	89	89	90
StackTest1	26	40	73	80	64	71	73
StackTest2	20	43	69	81	99	99	99
TestUSART	23	58	66	85	96	96	96
TestSPI	32	56	71	75	67	73	75
TestADC	22	62	69	78	98	98	98
% of (7)	24	53	69	78	85	88	89

Fig. 7. Branch coverage in percent. The last line gives a geometric mean.

Similarly, in Figure 6 the last line gives a geometric mean for each testing approach; the denominator is the execution time of testing approach 7 (which is DTall). In Figure 7 the last line gives a geometric mean for each testing approach.

C. Assessment

Testing approaches 1–3. Testing approach 1 uses a total of 3,000 random event sequences and the result is a stack-size-fraction mean of 62%. Testing approach 2 uses a genetic algorithm to improve the choice of event names, and that improves the stack-size-fraction mean to 67%. Testing approach 3 goes further by trying all combinations of event names and all integer wait times within a wide interval; the stack-size-fraction mean goes up to 71%. Note that testing approach 2 is almost two orders of magnitude faster than testing approach 3. Note also that in some cases testing approach 3 gives *worse* results than testing approach 2 because of poorer samples of the event values. Notice finally that the genetic algorithm in most cases is faster than random testing. The reason is that the procedure for generating random event sequences is quite slow, while one of the main ways the genetic algorithm produces new event sequences is to swap subsequences from existing event sequences.

Testing approaches 4–7. Testing approach 4 samples the event names and tries all integer wait times within a wide interval; the result is a stack-size-fraction mean of 81%. Thus, testing approach 4 dominates testing approaches 1–3 so we conclude that the use of directed testing to determine event values is essential to get good results. Testing approach 5 is the VICE approach, which, in sharp contrast to testing approach 4, samples the wait times but uses our SA-Tree technique to generate event names. VICE is 30x faster than testing approach 4 and yet it produces a better stack-size-fraction mean, namely 94%. Note also that VICE is within 10x of the running time of the static analysis. Testing approach 6 tries all combinations of event names and samples the wait times. The result is marginally better than VICE, namely 95%, but more than an order of magnitude slower. Finally, testing approach 7 is the DTall approach which tries all combinations of event names and all integer wait times within a wide interval. DTall achieves a result of 99%, though at the expense of the longest execution time of all the approaches. We conclude that testing *can* almost match the static analysis, which shows that the static analysis is about as good as it can be. We also conclude that VICE gives an excellent trade-off between precision and execution times; it is faster than all the other testing approaches and it is outperformed only by two much slower approaches.

Number of event sequences. VICE achieves its results with significantly fewer event sequences than random testing and the genetic algorithm. For four benchmarks, the difference is 2x, while for three benchmarks, the difference is 10x.

Branch coverage. Figure 7 shows that VICE and DTall produce excellent branch coverage numbers. Notice that the previous best testing-approach (approach 2) achieved a much lower branch coverage (53 percent) than VICE (85 percent) and DTall (89 percent). The wide spread of coverage numbers support that the benchmarks are nontrivial: we can find event sequences that lead most branches to go either way and yet only the best testing approaches achieve that.

VII. Conclusion

Our results show that the state-of-the-art static analysis produces excellent estimates of maximum stack size. Our testing approach DTall can almost match the results of the static analysis. Additionally, our approach VICE comes close and is two orders of magnitude faster than DTall, though also 10 times slower than the static analysis. The keys to produce challenging event sequences are to use directed testing to get event values and to use our SA-Tree technique to get event names. The SA-Tree technique is an example of how static analysis can help testing be more efficient.

Our technique is useful for other languages than Virgil. The availability of a source-level interpreter greatly facilitates the collection of constraints.

Our results show that VICE is useful for practical testing: VICE is faster and gives better branch coverage than the previous state-of-the-art testing approach. VICE quickly generates a small number of challenging event sequences that drive the execution into "dark corners" of the software. Such event sequences may reveal faults or help confirm that the software works correctly even for corner cases.

Acknowledgments. We thank Xiaoli Gong for help to instrument Avrora. We also thank Hong Hong for help to integrate the constraint solver into our system. We thank Matt Brown, Todd Millstein, and Hesam Samimi for helpful comments on a draft of the paper. National Science Foundation grant number 0820245 supported us in part.

References

[1] CHOCO. http://www.emn.fr/z-info/choco-solver/choco-documentation.html, September 2010.

[2] R. Alur and P. Madhusudan. Visibility pushdown languages. In *Proceedings of the thirty-sixth Annual ACM Symposium on Theory of Computing*, 2004.

[3] Atmel. Adc deriver manufacturer datasheet. http://www.atmel.com/dyn/resources/prod_documents/doc8078.pdf, September 2010.

[4] Atmel. Spi deriver manufacturer datasheet. http://www.atmel.com/dyn/resources/prod_documents/doc2582.pdf, September 2010.

[5] Atmel. Usart serial deriver manufacturer datasheet. http://www.atmel.com/dyn/resources/prod_documents/doc32006.pdf, September 2010.

[6] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R.Majumdar. Generating test from counterexamples. In *Proceedings of ICSE*, pages 326–335, 2004.

[7] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proc. Of International Symposium on Software Testing and Analysis*, pages 123–133, 2002.

[8] Klaus Brunnstein. About the "Altona railway software glitch". *The Risks Digest*, 16(93), 1995.

[9] Dennis Brylow, Niels Damgaard, and Jens Palsberg. Static checking of interrupt-driven software. In *Proceedings of ICSE'01, 23rd International Conference on Software Engineering*, pages 47–56, Toronto, May 2001.

[10] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 443–446, 2008.

[11] Cristian Cadar, Paul Twohey, Vijay Ganesh, and Dawson Engler. Exe: A system for automatically generating inputs of death using symbolic execution. In *Proceedings of 13th ACM Conference on Computer and Communications Security*, 2006.

[12] Krishnendu Chatterjee, Di Ma, Rupak Majumdar, Tian Zhao, Thomas A. Henzinger, and Jens Palsberg. Stack size analysis of interrupt driven software. *Information and Computation*, 194(2):144–174, 2004. Special issue dedicated to Paris Kanellakis. Preliminary version in Proceedings of SAS'03, International Static Analysis Symposium, Springer-Verlag (*LNCS* 2694), pages 109–126, San Diego, June 2003.

[13] RE Fikes and NJ Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208, Winter 1971.

[14] J. Gilson. *A New Approach to Engineering Tolerances*. The Machinery Publishing C. Ltd, 1951.

[15] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. of TACAS*, pages 553–568, 2003.

[16] Francois Laburthe. Choco: implementing a CP kernel. In *Proceedings of CP00 Post Conference Workshop on Techniques for Implementing Constraint programming Systems (TRICS)*, September 2000.

[17] Rupak Majumdar and Ru-Gang Xu. Directed test generation using symbolic grammars. In *Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering*, 2007.

[18] Koushik Sen Patrice Godefroid, Nils Klarlund. Dart: directed automated random testing. In *Proceedings of PLDI'05, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[19] John Regehr. Random testing of interrupt-driven software. In *ACM International Conference On Embedded Software*, pages 290–298, 2005.

[20] John Regehr and Alastair Reid. HOIST: a system for automatically deriving static analyzers for embedded systems. *ACM SIGARCH Computer Architecture News*, 2004.

[21] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. In *Proceedings of EMSOFT'03, Third International Conference on Embedded Software*, pages 306–322, 2003.

[22] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Proc. 18th International Conference on Computer Aided Verification*, pages 419–423, 2006.

[23] Koushik Sen. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering*, pages 571–572, 2007.

[24] Ben L. Titzer. Virgil: Objects on the head of a pin. In *Proceedings of OOPSLA'06, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 2006.

[25] Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proceedings of IPSN'05, Fourth International Conference on Information Processing in Sensor Networks*, pages 477–482, Los Angeles, April 2005.

[26] Ben L. Titzer and Jens Palsberg. Vertical object layout and compression for fixed heaps. In *Proceedings of CASES'07, International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 170–178, Salzburg, Austria, September 2007. A revised version of the paper appeared in Semantics and Algebraic Specification, Essays Dedicated to Peter D. Mosses on the Occasion of His 60th Birthday, Springer, LNCS 5700, 2009.