Timing Analysis of Event-Driven Programs with Directed Testing

Mahdi Eslamimehr and Hesam Samimi

Communications Design Group, SAP Labs, Los Angeles, USA
{eslamimehr,hesam@ucla}@ucla.edu

Abstract
Accurately estimating the worst-case execution time (WCET) of real-time event-driven software is crucial. For example, NASA’s study of unintended acceleration in Toyota vehicles highlights poor support in timing analysis for event-driven code, which could put human life in danger. WCET occurs during the longest possible execution path in a program. Static analysis produces safe but overestimated measurements. Dynamic analysis, on the other hand, measures actual execution times of code under a test suite. Its performance depends on the branch coverage, which itself is sensitive to scheduling of events. Thus dynamic analysis often underestimates the WCET. We present a new dynamic approach called event-driven directed testing. Our approach combines aspects of prior random-testing techniques devised for event-driven code with the directed testing method applied to sequential code. The aim is to come up with complex event sequences and choices of parameters for individual events that might result in execution times closer to the true WCET. Our experiments show that, compared to random testing, genetic algorithms, and traditional directed testing, we achieve significantly better branch coverage and longer WCET.

1998 ACM Subject Classification B.2.2 Performance Analysis and Design Aids, D.2.5 Testing and Debugging

Keywords and phrases worst-case execution time, timing analysis, event-driven, directed testing

Digital Object Identifier 10.4230/OASIcs.WCET.2015.21

1 Introduction

Real-time event-driven systems have become ubiquitous, from high performance servers to smart devices. The correctness of such systems becomes of utmost importance when human safety is concerned. Testing and analyzing real-time event-driven programs is notoriously hard, mainly due to the non-linear control flow in the execution of event handlers. Dependencies are complicated to track in event-driven code, leading to subtle bugs that can go unnoticed with traditional event-driven testing. For example, since 2002 more than 89 people have been killed and 60 injured, due to the unintended acceleration of Toyota cars, which has made the corporation recall more than 1 million cars due to safety issues. The 2011 NASA study of unintended acceleration in Toyota vehicles [12] highlights poor support in timing analysis for event driven code as a contributor to the safety holes.

Thus a precise calculation of the worst-case execution time (WCET) of real-time event-driven software is crucial. WCET is a much studied problem for event-driven software. For time-critical embedded systems, designers must avoid excessive over-provisioning of task deadlines, because it wastes processor’s availability that could otherwise be used for other functions. Designers shall also avoid under-provisioning as it can undermine the validity of the computation or lead to partial or complete loss of functionality.

The exact WCET in a program happens during the longest possible execution path.
among all processes.\footnote{ignoring per-statement execution time which depends on the underlying machine architecture.} In an event-driven code events can be fired at arbitrary times and a scheduler may interrupt the execution of one event handler to yield to another new or unfinished event handler. The choice of scheduling for the execution of handler codes affects the executed paths, since there can be shared state and exclusivity requirements among events. Thus WCET can only be precisely computed by checking the execution times of all possible execution paths over every possible schedule for the execution of every possible combination of triggered event handlers.

Static analysis can find an upper-bound on the true WCET\cite{18} without executing the program, yet it is necessarily conservative and can be difficult or currently impossible to perform on arbitrary code. Dynamic approaches\cite{18}, on the other hand, are easier to perform since they measure a program’s WCET by executing it on a test suite and tracking the longest execution time. Thus the accuracy of dynamic WCET calculation depends on the percentage of all possible execution paths covered. A straightforward way to estimate a program’s WCET is to sum up the WCET calculations of individual handlers when executed in isolation. Yet this estimate will necessarily be conservative and an upper-bound for the true WCET. This is because the WCET of one event may not occur on conditions that would cause the WCET of another event due to intra-dependencies among event handlers. The tested WCET is a lower-bound on the true WCET that occurs during some run of the program. In slogan form:

\[
\text{tested WCET} \leq \text{true WCET} \leq \text{static WCET}
\]

Therefore the success of a dynamic approach is closely tied to the branch coverage of the test suite used. Building a test suite with high coverage is a known challenge, and is even harder for event-driven software. Not only a tester must choose appropriate inputs to guide the execution to unexplored paths, she must devise a suite of event sequences (schedule of events fired) that dictate the execution context switches between concurrent processes. For creating an event sequence, a tester must decide on the number of events, the types of events, the argument values for each event, as well as a concrete timing schedule for the interrupts.

Previous work on testing event-driven software uses event sequences that are generated randomly\cite{2} or by genetic algorithms\cite{1}. In the domain of sequential software, the idea of directed testing with concolic execution (hybrid concrete and symbolic) has been receiving much attention in both research and practice. The idea is to execute the code concretely to explore one possible execution path, yet simultaneously employ symbolic execution to collect path constraints from each condition on the control-flow. Those constraints will be modified and solved iteratively, in order to find input values for subsequent runs, aiming to force the execution to unseen paths and increase the coverage. However, we showed previously\cite{4, 5} that using the classical directed testing on concurrent and event-driven programming paradigms is not as successful, because the scheduling of concurrent processes is a factor.

The challenge. Improve the accuracy of WCET measurement of dynamic test-based approaches by devising a method that achieves higher branch coverages.

Our approach. We present a new technique called event-based directed testing for testing of event-driven software. Our technique combines good features of random testing of event-driven software with the idea of directed testing for sequential code. Similar to directed
testing, after each round of concolic execution we generate a new scenario for further testing. Unlike directed testing where the scenario is uniquely described by input values, here we need to generate an event sequence. An event sequence not only provides input values for individual events, it provides a scheduling for the set of events that execute concurrently.

We have implemented our technique in an existing tool called VICE [4]. This approach, implemented by VICE, was previously used for maximum stack size analysis, and we have augmented it to automatically test event-driven software without a human in the loop. Our experiments show that compared to random testing, genetic algorithms, and traditional directed testing, we achieve significantly better branch coverage, and subsequently longer WCET. For 8 out of 11 benchmarks, we achieve more than 70% branch coverage. Compare to random testing, genetic algorithm, and traditional directed testing we improve WCET by 203%, 176%, and 97%, respectively.

The rest of this paper. In the next section we illustrate our approach via an example. in Sec. 3 we formalize our approach, and in Sec. 4 we show our experimental results.

2 Example

We now explain our approach through an example program shown in Listing 1—a simplified version of our antenna benchmark. There are two event handlers: main and alt. The program has three if-statements, hence six branches. The aim is to estimate the WCET for the entire program, represented by the main event, under a bounded number of possible interrupts.

An event sequence of a bounded length (here we pick four) is used to impose a particular scheduling of interrupts that occur during the execution of main. Each event in the sequence is a 4-tuple \( \langle id, name, args, timeout \rangle \). Each named event must be fired with the given arguments and maximum allotted time specified by the timeout value. If the identifier is seen before, this means rather to resume the execution of an earlier interrupt. Whether the event handler is finished or interrupted by the scheduler due to a timeout, the scheduler proceeds to fire the next event in the sequence. Once the sequence is finished, the scheduler is left alone to let all unfinished interrupts run their course one by one. Our objective is to produce new sequences that will lead us to new paths in search of longer execution times.

Testing proceeds in rounds. In each round we choose a new event sequence to execute. For each tuple, we pick randomly either a new or old identifier, and, in the former case, the name of the event. We use concolic execution to determine the arguments. It is possible to generate invalid sequences, since the interrupt mask register follows interrupt rules (e.g., a handler cannot interrupt itself), in which case we move onto the next round. During the execution of each round, we monitor the branch coverage so far, as well the execution time of the main interrupt, which will be used to determine the WCET thus far.

Round one. In the first round, the arguments too are chosen randomly so we might begin with this event sequence (we omit the timeouts and show names and ids together):

\[
\langle \text{main}, (723452) \rangle, \langle \text{alt1}, (-10038) \rangle, \langle \text{main}, \_ \rangle, \langle \text{alt1}, \_ \rangle
\]

The concolic execution will fire the first event. main’s execution continues with calling dispatch_data in line 4 and we collect the constraint \( data_1 = msg \), which maps the actual parameter (line 4) to the formal (line 6). We use \( data_1 \) to represent the symbolic value of \( data_1 \), and likewise for \( msg \) and \( msg \). main’s execution reaches the second if-statement of line 9. Assuming only arithmetic equations from conditionals are collected as constraints,
we proceed to collect $-2048 < msg \land msg < 1024$ at line 9. Since $msg$'s concrete value is 723452, the conditional evaluates to false. Let us assume this handler gets interrupted before resetting the sending flag. alt1 is launched now, which proceeds to calling dispatch_data, and the constraint $data_2 = msg$ is collected. The alt1 event terminates early at line 8, due the sending flag being set. main now resumes and runs to completion. The fourth event in the sequence is skipped since alt1 is already terminated. During the first round, the path (of statements visited in all handlers) and the elapsed time for the execution are recorded. The branch coverage was 50% (two false branches and one true branch out of six). The execution never reached any farther than line 12.

After the completion of the first round, a new sequence of event ids and names is generated randomly. Each event in the sequence will be paired with argument values obtained by solving for all three constraints that we collected above. For example, for the next round we may produce the event sequence:

$$\langle \text{main},(-338)\rangle, \langle \text{alt1},(1001)\rangle, \langle \text{alt2},(6)\rangle, \langle \text{main},_\rangle$$

where the solution set for the first element was $data_1 = data_2 = msg = -338$, and so on.

**Round two.** This time main's execution takes the true branch in line 9 and goes on to invoke the check_and_send routine. Consequently we collect new constraints $msg = s \land tmp = t$. Let's assume at this point main's handler times out. The second and third handlers—alt1 and alt2—will run and terminate early one at a time, again due to boolean flag. The last event in the sequence is main, which resumes in the body of check_and_send. It encounters a new constraint at line 14: $s = 512$, which evaluates to false on the current argument value. The handler runs more code at line 16 and finishes. At this round, a longer execution is found due to a better branch coverage, when the handler’s execution reached into deeper parts of the program. The total branch coverage over the first two rounds is 83% ($5 / 6$).

**Round three.** Suppose in the third round we use the same sequence as before, yet timeouts are set differently, so that both main and alt1 handlers proceed to invoke the routine and terminate at line 12. We note that while branch coverage wasn’t improved, a longer WCET so
far was measured. Upon the completion of the execution of this round all collected constraints are sent to the constraint solver to generate arguments for each event in a new sequence, getting the solution data$_1 = data_2 = msg = s = 512$. Thus, the new event sequence for the next round might be:

$\langle [\text{main}, (512)], \langle \text{alt1}, (512) \rangle, \langle \text{main}, _\rangle, \langle \text{alt1}, _\rangle \rangle$

**Round four.** In the fourth round, let us assume main’s timeout is small and it quickly gets interrupted by alt1, after executing the first statement. alt1’s handlers proceeds to calling check\_and\_send and this time will take the true branch of the if-statement. Again, before resetting the sending flag, the execution may yield back to main which will terminate early at line 8. During this round, we achieved 100% coverage, yet we observe that the longest execution hence the largest WCET so far occurred during the previous round.

**More rounds.** New rounds will be carried out until we measure no improvement in both WCET and the branch coverage. At this point, the algorithm terminates.

**Discussion.** The example illustrates several strengths of our approach. First, the combined monitoring of WCET and branch coverage as the terminating condition for the algorithm leads to a more accurate estimate of WCET, as opposed to relying on one of them only. Second, the combination of a randomly chosen sequence of interrupts, with arguments for each obtained by constraint solving leads to the exploration of a diverse set of control-flow paths. For example, the chance of reaching line 15 with input sequences generated randomly or by genetic algorithms is very small. Third, we get good branch coverage with a fairly short number of event sequences, each with a small length, as a direct benefit of directed testing method. In our example, a length of two would have sufficed.

### 3 Approach

Each tested program is a VirgilProgram (see http://compilers.cs.ucla.edu/virgil); we compile each to machineCode, that is, AVR assembly code. A key input to each execution is an eventSequence—a list of tuples—where each tuple consists of an event-handler name (an identifier), a unique ID for each call to a handler (an int), a list of event argument values (as ints), and a timeout—the maximum time given for the execution of the event—measured in milliseconds. Each of our constraints is a Virgil arithmetic or logical expression, and a solution maps each relevant program variable (identifier) to an int, or is otherwise None. Our approach uses a data structure of type state that is a tuple of five values. If $s$ is of type state, the first component ($s.$wcet) is the WCET found so far. The second ($s.$coverage) is the highest branch coverage found so far. The third ($s.$eventSeq) is the event sequence that led to WCET, and the fourth ($s.$constraints) the collected constraints during such pass. Finally the fifth component ($s.$noChange) is the number of consecutive rounds without improvements to either the WCET or branch coverage.

**Tools.** Our approach uses 7 tools, whose types are shown in Fig. 1: Compiler is an open-source Virgil compiler [16] that generates AVR machine-code. The tool avrora is an open-source simulator for AVR machine code [17]. The tool random takes no inputs and produces a random event sequence. The tool timeoutCombos takes an event sequence with undefined timeouts and duplicates the sequence for all possible combinations of timeout values for each event in the sequence. The tool concolic is a concolic execution engine for
Virgil that takes a Virgil program and an event sequence. Concolic will run \texttt{avrora} to fire the events from the event sequence with the given set of timeout choices. The result of a run of concolic is a measurement of WCET, of the branch coverage achieved, plus a collection of constraints. Solver is a constraint solver used for the directed-testing approach. The tool generator takes a mapping of variable names to values and generates an event sequence.

**Event-Based Directed Testing.** Fig. 2 lists our algorithm. We compile each Virgil program to AVR assembly code. The algorithm starts from a randomly generated event sequence, and generates all combinations of timeout choices (sweeping a range) to produce a list of event sequences. For each sequence (which now has a particular choice for timeouts), it executes concolic on the Virgil program to get new values for the branch coverage, new constraints, as well as WCET on the assembly code by running \texttt{avrora}. The solver and generator will convert the constraints into a new event sequence to be used in the next round. After each run, we invoke an update function (omitted for brevity) which updates the state variable \texttt{s} to reflect the latest information of the WCET, branch coverage, and path constraints that led to WCET, found so far. We also update \texttt{s.noChange} to reflect how many recent unique rounds with no change to either WCET or the branch overage have occurred. This is used as the terminating condition for the algorithm.

**4 Experimental Results**

We compare EBDT to random testing, genetic algorithms, and traditional directed testing.
4.1 Methodology

To have fair comparisons, we implemented random testing, a genetic algorithm, and traditional directed testing for Virgil. Our implementation of event-based directed testing—VICE—is written in Java. We used an existing Virgil interpreter as the basis for our concolic execution engine, which tracks symbolic expressions alongside concrete values. For constraint solving, we use the open-source solver Choco [13]. We implemented the genetic algorithm on top of the Java genetic algorithm package (JGAP). In order to perform traditional directed testing with VICE, we ran our algorithm on each event in isolation and summed up the individual WCET estimates. For timeout values, we exhaustively sweep a range from the smallest allowed interval between context switches (measured 8 ms, empirically) up to the full execution time of the event handler when run without any interruptions.

4.2 Benchmarks

The following table shows some statistics about our eleven benchmarks that test device drivers for Berkeley Motes, including the Virgil and translated C lines of code counts. The C code is compiled into AVR assembly. The table also shows the number of event handlers.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LOC (Virgil)</th>
<th>LOC (C)</th>
<th>#handlers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryTree</td>
<td>114</td>
<td>167</td>
<td>1</td>
<td>a simple (unbalanced) binary tree</td>
</tr>
<tr>
<td>LinkedList</td>
<td>124</td>
<td>181</td>
<td>1</td>
<td>a simple doubly-linked list</td>
</tr>
<tr>
<td>BubbleSort</td>
<td>55</td>
<td>519</td>
<td>3</td>
<td>the common but slow bubblesort algorithm</td>
</tr>
<tr>
<td>Decoder</td>
<td>772</td>
<td>1015</td>
<td>3</td>
<td>decode instructions and other binary data</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>920</td>
<td>1338</td>
<td>11</td>
<td>a visualizer for sensor readings</td>
</tr>
<tr>
<td>Fannkuch</td>
<td>422</td>
<td>605</td>
<td>3</td>
<td>measures impact of compiler optimizations on runtime performance</td>
</tr>
<tr>
<td>MsgKernel</td>
<td>773</td>
<td>1519</td>
<td>2</td>
<td>adaptation of the core message-passing mechanism from SOS operating system</td>
</tr>
<tr>
<td>TestRadio</td>
<td>1695</td>
<td>2833</td>
<td>6</td>
<td>tests the functionality of the Radio (wireless signal) driver.</td>
</tr>
<tr>
<td>TestUSART</td>
<td>1,226</td>
<td>1,737</td>
<td>5</td>
<td>tests the Universal Synchronous Asynchronous Receiver Transmitter driver.</td>
</tr>
<tr>
<td>TestSPI</td>
<td>859</td>
<td>1,109</td>
<td>3</td>
<td>tests the Serial Peripheral Interface driver.</td>
</tr>
<tr>
<td>TestADC</td>
<td>605</td>
<td>1,055</td>
<td>4</td>
<td>tests the Analog to Digital Converter driver.</td>
</tr>
</tbody>
</table>

4.3 Measurements

We performed our experiments on a 2.3 GHz Intel Core i7 iMac, with Sun Java2 SDK 1.5. All time measurements are in milliseconds. We used event sequences with 100 events for all experiments, except for traditional directed testing where each event sequence consists of a single event. In runs of the genetic algorithm, each generation has 500 event sequences. The genetic algorithm stops after two generations result in no improvement to the branch coverage or the WCET. We use the same number of event sequences for random testing and the genetic algorithm, for a fair comparison. Tab. 1 compares the results of EBDT and existing solutions, while Tab. 2 lists the timing of the runs. We also report the WCET found by Avrora’s static deadline analyzer.

VICE source with all benchmarks can be found at https://github.com/Mah-D/VICE.
Timing Analysis of Event-Driven Programs with Directed Testing

Table 1: Experimental results: EBDT, plus static WCET.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Random #ES WCET BC</th>
<th>GA #ES WCET BC</th>
<th>DSE #ES WCET BC</th>
<th>VICE #ES WCET BC</th>
<th>Static WCET</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryTree</td>
<td>2000 14 26%</td>
<td>2000 15 41%</td>
<td>55 24 45%</td>
<td>1 25 45%</td>
<td>118</td>
</tr>
<tr>
<td>LinkedList</td>
<td>2000 20 29%</td>
<td>2000 20 48%</td>
<td>70 30 55%</td>
<td>1 30 55%</td>
<td>91</td>
</tr>
<tr>
<td>BubbleSort</td>
<td>1500 8 22%</td>
<td>1500 8 30%</td>
<td>13 13 43%</td>
<td>104 69 80%</td>
<td>153</td>
</tr>
<tr>
<td>Decoder</td>
<td>7000 21 20%</td>
<td>7000 24 34%</td>
<td>194 29 50%</td>
<td>499 62 73%</td>
<td>88</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>17500 39 7%</td>
<td>175000 47 19%</td>
<td>502 61 39%</td>
<td>1019 90 65%</td>
<td>555</td>
</tr>
<tr>
<td>Fannkuch</td>
<td>9000 15 18%</td>
<td>9000 19 29%</td>
<td>322 27 51%</td>
<td>717 81 76%</td>
<td>304</td>
</tr>
<tr>
<td>MsgKernel</td>
<td>7000 38 26%</td>
<td>7000 48 52%</td>
<td>172 50 63%</td>
<td>315 82 92%</td>
<td>218</td>
</tr>
<tr>
<td>TestRadio</td>
<td>11500 44 13%</td>
<td>115000 51 28%</td>
<td>249 63 40%</td>
<td>293 74 63%</td>
<td>323</td>
</tr>
<tr>
<td>TestUSART</td>
<td>8500 53 28%</td>
<td>8500 66 52%</td>
<td>279 73 72%</td>
<td>411 94 100%</td>
<td>176</td>
</tr>
<tr>
<td>TestSPI</td>
<td>9500 25 35%</td>
<td>9500 28 31%</td>
<td>44 37 43%</td>
<td>107 39 51%</td>
<td>401</td>
</tr>
<tr>
<td>TestADC</td>
<td>7000 14 33%</td>
<td>7000 22 65%</td>
<td>18 30 58%</td>
<td>336 41 97%</td>
<td>102</td>
</tr>
</tbody>
</table>

Table 2: Testing time: EBDT testing.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Random</th>
<th>Genetic</th>
<th>DSE</th>
<th>VICE</th>
<th>Benchmark</th>
<th>Random</th>
<th>Genetic</th>
<th>DSE</th>
<th>VICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryTree</td>
<td>8220</td>
<td>4187</td>
<td>40</td>
<td>2079</td>
<td>MsgKernel</td>
<td>7341</td>
<td>5710</td>
<td>48</td>
<td>1463</td>
</tr>
<tr>
<td>LinkedList</td>
<td>12641</td>
<td>8347</td>
<td>55</td>
<td>3753</td>
<td>TestRadio</td>
<td>6291</td>
<td>22080</td>
<td>40</td>
<td>928</td>
</tr>
<tr>
<td>BubbleSort</td>
<td>463</td>
<td>219</td>
<td>13</td>
<td>176</td>
<td>TestUSART</td>
<td>7812</td>
<td>4189</td>
<td>33</td>
<td>314</td>
</tr>
<tr>
<td>Decoder</td>
<td>10686</td>
<td>9118</td>
<td>64</td>
<td>2325</td>
<td>TestSPI</td>
<td>14432</td>
<td>11729</td>
<td>19</td>
<td>3271</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>41785</td>
<td>15959</td>
<td>66</td>
<td>7295</td>
<td>TestADC</td>
<td>6803</td>
<td>2460</td>
<td>11</td>
<td>893</td>
</tr>
<tr>
<td>Fannkuch</td>
<td>16804</td>
<td>12012</td>
<td>37</td>
<td>2104</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.4 Assessment

**WCET.** Results are shown in Tab. 1. VICE found the longest WCET across random testing, genetic algorithm, and the traditional single event directed testing (DSE). Compare to random testing, genetic algorithm, and traditional directed testing VICE improves WCET by 203%, 176%, and 97%, respectively. Big differences are seen in BubbleSort and Fannkuch because of several nested conditional structures of these benchmarks. VICE’s estimates were closest to the static (over-estimate) computation of WCETs in all benchmarks. None of the other testing approaches come close to match VICE’s results consistently.

**Branch coverage (BC).** Tab. 1 also illustrates branch coverage results. For all benchmarks VICE gives the best result. For one benchmarks VICE gives 100% branch coverage, and in seven others VICE covered more than 70% of branches. BinaryTree shows the lowest coverage. This benchmark has numerous non-numeric conditionals which VICE cannot currently handle. None of the other approaches come close to match VICE’s results consistently.

**Number of event sequences (#ES).** VICE achieves its results with significantly fewer event sequences than random testing and the genetic algorithm. For two benchmarks, the difference is about 4X, while for nine benchmarks, the difference is more than 10X. In contrast, VICE uses about 3X more event sequences on average than traditional directed testing with a single event per event sequence. These results show that VICE achieves good results with a fairly low number of event sequences.
Testing time. In almost all cases, VICE is significantly faster than random testing and the genetic algorithm, and slower than DSE.

The constraint solver. We found that CHOCO does a good job with number types while has poor support for other types and operations, e.g., array and bit operations, user types. Therefore, path constraints that aren’t boolean or arithmetic were not collected by our concolic tool, hurting the branch coverage and subsequently WCET estimates.

Single event versus VICE. Traditional directed testing can be employed for finding WCET, by computing WCETs for each event in isolation and adding them as the total WCET of the program. Yet result are inaccurate and overly conservative. The conditions that result in WCET of individual event handlers may be impossible to have at the same time, since handlers can interrupt each other and interact indirectly via shared state. Our choice of event sequences of length 100 was chosen based on experience with the benchmarks. The more event handlers we have, the longer event sequences are needed for good testing. Finding suitable lengths of event sequences for each application is a subject of future work.

5 Related Work

In Sec. 4, we mentioned four event sequence generation techniques for WCET analysis: random technique [2], genetic algorithm [1], traditional directed testing [15, 7], and static analysis [3]. Here we highlight some of the notable techniques and tools in the area of WCET.

Static techniques examine the source code without running it and return an upper-bound for the true WCET. Different static techniques have been used to find WCET. aiT WCET [6], Bound-T [11], and SWEET [8] use value analysis, in which register and memory values are approximated, without running the program, at every program point. The results are necessarily conservative. For example, they perform poorly when loop structures are present. Control-flow analysis can be used (e.g., [9]) to determine possible execution paths, which aids WCET calculation. In control-flow analysis a superset of all execution paths is created with a control-flow graph. An input range and tasks will be passed as an input to the CFG, and the worst timing is computed. Performing control-flow analysis on source code is simpler than on machine code. However, compilation, code optimization, and linkage may change program control flow and make the analysis cumbersome. Processor-behavior analysis, where exact behaviors of a processor, as in memory accesses, caching, and pipelining, are mimicked, is employed by [10] for WCET calculation. In practice these tools’ calculations related to the processor, memory hierarchies, buses, and peripherals are all approximates and thus their timing results are conservative.

Dynamic techniques, in which the program is executed, have been discussed in the previous section. These use a variety of ways to generate inputs, e.g., random and probabilistic techniques [2], genetic algorithms [1], or other heuristics [14]. The closest work to ours is pathcrawler [19], which is very similar to the single event traditional directed testing.

6 Conclusion

Testing of event-driven software is difficult because of the need for event sequences rather than single inputs. We have shown that a combination of random testing and directed
testing can be used to automatically produce effective event sequences that are challenging
to produce manually. We presented our approach—event-driven directed testing—as a major
improvement over traditional directed testing, which also generally produces better results
than random testing and genetic algorithms.

There are many opportunities for future work. More general classes of path constraints
need to be supported. Constraint solving may also be useful for both generating schedules (we
currently use randomly generated sequences), as well as effectively finding suitable interrupt
timeout values. These improvements will make coverage and WCET estimates more accurate,
while reducing analysis times.

References

1 Austrian Computer Society (OCG). Heuristic Worst-Case Execution Time Analysis, 10th
European Workshop on Dependable Computing, 1999.
2 Guillem Bernat, Antoine Colin, and Stefan M. Petters. WCET analysis of probabilistic
hard real-time systems. In Proceedings of the 23rd IEEE Real-Time Systems Symposium,
SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-
4 Mahdi Eslamimehr and Jens Palsberg. Testing versus static analysis of maximum stack
Society.
5 Mahdi Eslamimehr and Jens Palsberg. Sherlock: Scalable deadlock detection for concur-
rent programs. In Proceedings of the 22Nd ACM SIGSOFT International Symposium on
ACM.
6 Christian Ferdinand. aiT: Worst-case execution time prediction by static program analysis.
8 Jan Gustafsson and Andreas Ermedahl. Automatic derivation of loop bounds and infeasible
9 Jan Gustafsson, Andreas Ermedahl, and Björn Lisper. Towards a flow analysis for embed-
ded system C programs. In Object-Oriented Real-Time Dependable Systems, 2005. WORDS
10 Reinhold Heckmann. The influence of processor architecture on the design and the results
11 Niklas Holsti, Thomas Langbacka, and Sami Saarinen. Using a worst-case execution time
tool for real-time verification of the DEBIE software. EUROPEAN SPACE AGENCY-
12 M. Kirsch. Technical support to the national highway traffic safety administration (NHTSA)
on the reported Toyota motor corporation (TMC) unintended acceleration (UA) investiga-


