ABSTRACT

Infeasible execution paths reduce the precision of structural testing coverage and limit the industrial applicability of structural testing criteria. In this paper, we propose a technique that combines static and dynamic analysis approaches to identify infeasible program elements that can be eliminated from the computation of structural coverage to obtain accurate coverage data. The main novelty of the approach stems from its ability to identify a relevant number of infeasible elements, that is, elements that belong statically to the code, but cannot be executed under any input condition. The technique can also generate new test cases that execute uncovered elements, thus increasing the structural coverage of the program. The experimental results obtained on a prototype implementation for computing accurate branch coverage and reported in this paper indicate that the technique can effectively improve structural coverage measurements and can thus increase the industrial applicability of complex structural coverage criteria.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging

General Terms
Algorithms

Keywords
Structural testing, concolic execution, automatic test generation

1. INTRODUCTION

Structural testing coverage has been widely studied as a means for assessing the adequacy of test suites with respect to the code. Structural coverage measures the adequacy of test suites as the amount of code elements of a given type executed by the test cases with respect to the total amount of those elements in the program. For example, statement and branch coverage measure the portion of executed statements and branches, respectively [18]. Quality managers use structural coverage criteria to evaluate test suites, determine when to terminate testing, and identify portions of the code that require additional testing [13].

Despite the definition of many structural coverage criteria, only few find common industrial application. Mature practical processes refer mostly to statement coverage, the simplest structural coverage criterion, and refer to more sophisticated coverage criteria only when required by domain regulations. For example safety-critical avionic applications use the modified condition decision coverage, as required by the standard DO-178B [16].

The limited industrial success of most structural coverage criteria depends on the difficulty of both identifying inputs that execute uncovered elements and computing accurate coverage values. The first problem amounts to finding the inputs that exercise a specific statement, branch or other element. The second problem stems from the difficulty of identifying infeasible elements, that is, elements that cannot be executed under any input condition, and therefore should not be counted. Both problems are undecidable in general, and hard to solve in practice.

Finding test cases to increase structural testing coverage is being recently tackled by approaches that generate test cases using symbolic and concolic (that is, interwoven concrete and symbolic) execution [17, 9, 14]. These approaches drive the exploration of the executable paths of a program, typically in depth-first order, and generate test cases accordingly. Since most programs have infinitely many paths, a depth-first search is in general ill-suited for the goal of covering a finite domain: It leads to a fine-grained exploration of only small portions of the program state space, easily diverges, and often finds many test cases that do not increase the coverage of the structure of the program. Other search strategies select paths that lead to uncovered elements in the control-flow graph. These strategies rely on heuristics to direct the search towards the most promising paths [10, 2]. Heuristics can increase coverage, but do not prevent the search to be stuck in exploring an infinite set of infeasible paths.

Current tools for computing structural coverage sidestep the infeasibility problem, and compute the structural coverage as the ratio between the elements executed during testing and the elements that belong statically to the code. This produces inaccurate results due to infeasible elements. The inaccuracy produced by a relatively small portion of infeasible elements can thus increase the industrial applicability of structural coverage measures and can thus increase the structural coverage measurement. The main novelty of the approach stems from its ability to identify a relevant number of infeasible elements, that is, elements that belong statically to the code, but cannot be executed under any input condition. The technique can also generate new test cases that execute uncovered elements, thus increasing the structural coverage of the program. The experimental results obtained on a prototype implementation for computing accurate branch coverage and reported in this paper indicate that the technique can effectively improve structural coverage measurements and can thus increase the industrial applicability of complex structural coverage criteria.
sible elements can be tolerated by defining relaxed coverage thresholds. For example, if 100% statement coverage cannot be achieved due to infeasible elements, a 90% statement coverage can provide enough confidence in the test suites, and thus partially satisfy quality managers. However, the portion of infeasible elements grows with the complexity of the coverage criterion, and becomes a big obstacle to practical applicability of sophisticated criteria.

The goal of our research is to define a new generation of structural testing techniques to compute accurate structural coverage measurements also for sophisticated testing criteria by both automatically generating test suites with high structural coverage, and correctly accounting for a relevant number of infeasible elements. To this end, we combine dynamic analysis, concolic execution and abstraction refinement to overcome the limitations of traditional approaches. Dynamic analysis monitors test execution to identify the covered elements and the feasible execution paths in the program state space. Concolic execution computes the path conditions that indicate how to execute unexplored paths to exercise uncovered elements, and solves these path conditions to identify new test cases. Abstraction refinement prunes the infeasible elements that are identified by unsatisfiable path conditions. An element is identified as infeasible when all the execution paths that lead to that element are infeasible. Our technique is rooted in an existing procedure for deciding the reachability of program statements [11, 1], which we adapt and extend to the problem of structural coverage of feasible code.

This paper contributes to scientific knowledge in several ways:

- It combines automatic test case generation based on concolic execution with dynamic analysis and abstraction refinement to compute accurate structural coverage measurements. The approach works for arbitrary structural adequacy criteria, and can produce test suites that cover most feasible program elements, while identifying many infeasible elements.

- It extends abstraction refinement by introducing an algorithm based on abstraction refinement and coarsening. Coarsening boosts the scalability of abstraction refinement and allows its application to the demanding problem of generating test suites with high structural coverage.

- It introduces STAR, a prototype implementation of the proposed technique that we used for experimental validation. STAR automatically generates test suites for C programs pursuing full branch coverage.

- It presents a set of experimental results collected by applying STAR to sample C programs. These data provide initial empirical evidence of the advantages of our technique with respect to both random testing and directed testing based on either concolic or symbolic execution.

The paper is organized as follows. Section 2 exemplifies the impact of infeasible elements on structural testing coverage by discussing the branch coverage of a sample program that we use in the experimental validation. Section 3 provides background on the existing work that inspired our technique, and motivates its evolution to the abstraction refinement and coarsening technique. Section 4 presents our approach, discusses the principles on which it is based, and defines the analysis algorithm in detail. Section 5 briefly illustrates the prototype implementation of our technique. Section 6 presents the preliminary empirical results obtained by applying the prototype on some sample programs. Section 7 surveys the related work. Section 8 summarizes the results of this paper, and outlines our current research agenda.

2. THE INFEASIBILITY PROBLEM

Infeasible program elements reduce the precision of structural coverage criteria that are defined as the ratio between the executed and the total amount of elements in a program [13]. For simple criteria, like statement coverage, the amount of infeasible elements is not particularly high, and practitioners can take advantage of structural testing coverage by either referring to an approximate satisfaction threshold, or by manually justifying the infeasible elements. However, even for slightly more demanding criteria, the impact of infeasible elements on structural coverage can be high, not even allowing the satisfaction of partial thresholds, while manual justification can be unacceptably complex. In this section, we discuss the infeasibility problem by showing through a simple but representative example that infeasible elements can have a relevant impact even on simple coverage criteria.

Figure 1 shows the C function calc_week that we excerpted from the code of the MySQL database management system. Function calc_week takes a date (the first parameter l_time formatted after line 7), and returns the corresponding week of the year (an integer value between 0 and 53). The second parameter week_behavior sets the week counting options. This parameter is interpreted as bit sequence: The three less significant bits indicate the day that starts the week (either Sunday or Monday) the baseline to count weeks (either 0 or 1), and reference standard for the date representation (ISO standard 8601:1988 or not), respectively. The constant masks at lines 2–4 are used to extract the values of the three less significant bits from the parameter week_behavior (lines 31–34).

The parameter week_behavior increases the reusability of function calc_week across applications that address different user contexts. In the context of a specific application, calc_week is usually specialized by passing a fixed constant value of week_behavior to all calls. In our example, we consider a program P that uses the function calc_week in a context where the weeks start on Sunday, are computed in the range between 0 and 53, and does not use the ISO 8601:1988 standard.

We computed the branch coverage of function calc_week tested in the context of program P with our tool STAR, and we manually investigated the feasibility of the uncovered branches. The function calc_week has 50 control flow branches, many of which are infeasible in the context of P that specialize the use of the library function. For example, the condition weekday >= 4 at line 41 is never executed, because the value of variable first_weekday is true for all the test cases valid in P. Similarly, the code within the last outermost if statement (line 54) is never executed, because the value of variable week_year is always false in P. In total,
Figure 1: The \texttt{calc\_week} function of MySQL

3. COVERAGE REFINEMENT

This paper proposes abstraction refinement and coarsening (ARC), as an approach for improving structural testing coverage by accounting for infeasible code elements. Such approach extends over an algorithm introduced by Beckman et al. for computing the reachability of program statements [1]. This section briefly overviews the characteristics of the referred algorithm, as required to understand our proposal, and pinpoints the key challenges that we faced while adapting it to the structural test generation problem. We then present the details of ARC in Section 4.

3.1 Background: Static-dynamic reachability

In previous work, Beckman et al. introduced DASH, an algorithm to compute the reachability of (faulty) statements of programs. DASH tries to either prove that the faulty statement is not reachable, or produce a test case that executes the statement. The two activities proceed incrementally, and interplay with each other.

DASH looks for a test case that executes the faulty statement by exploring program paths that are increasingly closer to the statement, adapting the approach of concolic execution [9, 14, 15]. It tries to prove that the faulty statement is not reachable by progressively refining a finite abstract model that conservatively overapproximates all possible transitions between program states, until the model contains no abstract trace that includes the faulty statement. DASH stores a history of the abstract states covered by the test cases, and uses such information to coordinate test case construction with model refinement, focusing one of the two activities at each iteration, as follows:

1. Execute the set of test cases, and identify the abstract states covered by the concrete states reached by the tests. If a test case executes the faulty statement, then terminate and return it.
2. Search the model for traces that reach the faulty state (error traces). If the model contains no error traces, change the path condition to reach the faulty state, and check for the satisfiability of the computed state transition that belongs to an error trace, and connects an abstract state \( s_1 \) covered by at least a test case \( t \), to an abstract state \( s_2 \) not covered by any test case.
3. Identify a frontier transition in the model, i.e., a transition that belongs to an error trace, and connects an abstract state \( s_1 \) covered by at least a test case \( t \), to an abstract state \( s_2 \) not covered by any test case.
4. Execute the program symbolically along the test case \( t \) up to state \( s_1 \), change the path condition to reach state \( s_2 \), and check for the satisfiability of the computed path condition using an automatic solver.
5. If the solver finds a solution, add it to the set of test cases, and proceed to step 1.

only 37 out of 50 control flow branches of function \texttt{calc\_week} are indeed feasible in the context of program \( P \). As a result, a classic coverage tool used to compute the branch coverage of a test suite that covers all feasible branches of \texttt{calc\_week} within program \( P \) would return 74% coverage, giving an erroneous indication about the completeness of the test suite with respect to the chosen criterion.

This example illustrates how testing a software module in a context that does not elicit all its possible behaviors yields many infeasible elements. This happens in general when reusable libraries are integrated in systems that use only subsets of their functionalities. Additionally, we notice that, the application of more demanding structural testing criteria, such as data flow coverage criteria [7], further emphasizes the problem. As a matter of fact, a more demanding criterion requires to exercise the program more thoroughly, and thus results in increased numbers of statically identified elements, and increased probability that a statically identified element is dynamically infeasible.
6. If the solver does not find a solution, conservatively refine the model by eliminating the infeasible transition between states s1 and s2 (see details below), and proceed to step 2.

Figure 2 illustrates how DASH refines the model conservatively. Given a frontier transition from a state s1 to a state s2, DASH splits s1 into two new states annotated with complementary refinement predicates, namely the weakest precondition of s2 through the frontier transition (wp) and its negation (!wp). All test cases that reach s1 reach also [wp]s1, while state s2 may be reachable from [wp]s1, but not from ![wp]s1. Neither [wp]s1 nor s2 is reached by any test case, by construction.

The refinement sets the frontier one step backwards: The reachability of s2 is reduced to the reachability of [wp]s1. If the frontier reaches the entry state of the model, DASH can safely conclude that such a frontier is infeasible, and can remove the corresponding transition from the model.

![Figure 2: Refinement of an infeasible transition](image)

The DASH implementation referred by Beckman et al. in [1] has not publicly released yet. We implemented the algorithm on top of the open-source CREST concolic execution engine [2]. In this paper we refer to our implementation of the DASH algorithm as DASH.

### 3.2 From reachability to structural coverage

To satisfy a structural coverage criterion, we must extend test suites with test cases that execute elements not covered yet. To measure coverage precisely, we must identify and ignore infeasible elements. Both problems can be restated in terms of DASH-style reachability problems, one for each target code element\(^2\). However, straightforward applications of DASH do not scale. In this subsection, we report the results of our study about the scalability of straightforward applications of DASH, before presenting our approach in the next section.

We can use DASH to solve the coverage problems in two ways, which we refer to as external DASH (eDASH) and incremental DASH (iDASH), respectively. The naive eDASH approach consists of calling a different instance of a DASH implementation for each target element. The less naive iDASH approach operates on a single abstract model, and shares the set of identified test inputs, thus avoiding repeat-

\(^2\)A slightly more sophisticated condition than the raw weakest precondition is required in presence of aliases [1]. Here we assume no aliases for the sake of simplicity.

\(^3\)This implicitly assumes that the target code elements can be expressed as code locations, possibly after a suitable instrumentation of the code. This is generally true for common structural coverage criteria.

The readers should notice that both approaches can still yield partial coverage since DASH may not be able to decide on the reachability of some element. Our experiments indicate that both approaches do not scale. For eDASH, the predominant penalizing factor is the large amount of abstract traces that are re-analyzed at any new invocation of DASH. This entails many redundant recomputations of the same concolic executions, calls to the solver and refinements of the model. For programs with many paths and code elements, the burden of this redundancy determines dramatic loss in performance, and several invocations of DASH do not to terminate within reasonable time. iDASH experiences less disastrous performance albeit at the cost of eager memory request that causes the procedure to run out-of-memory even for simple programs, as the calc seeker procedure exemplified in Section 2. In fact as shown in Figure 2, every refinement adds a new state and two refinement predicates to the model, thus progressively leading to a heavy memory occupation.

Moreover, both procedures generate increasingly complex predicates, with large amounts of conjunctions and disjunctions that quickly become hard if not impossible to solve automatically. The approaches generate predicates with many conjunctions and disjunctions when a model state is refined against multiple abstract traces that intersect in it, as illustrated in Figure 3. In Figure 3(a) the states s0, s1 and s2 are already covered by a concrete execution, while state s3 is not yet covered: s1 → s3 and s2 → s3 are frontier transitions since they can lead from covered states to an uncovered target. The figure shows how DASH incrementally refines the model when s3 is infeasible. At each iteration, DASH splits a state and adds two refinement predicates that are propagated backwards in disjunctive form, as illustrated by the annotations of the white states in the figure. In this refinement process, state s1 is refined twice, against the traces ⟨s1, s2, s3⟩ and ⟨s1, s3⟩, since these two traces share s1. The last refinement generates the predicate c3 ∧ (c1 ∨ c2) that identifies an abstract state yet to be covered, and contains both conjunctive and disjunctive operators.

The amount of atomic clauses grows quickly with the size of the program. Our preliminary experiments led already to predicates with more than 300,000 atomic clauses and a consequent explosion of solving time.

### 4. REFINEMENT AND COARSENING

In this section, we explain abstraction refinement and coarsening (ARC), a new approach that automatically generates test suites with high coverage, and optimizes the coverage information by detecting infeasible code elements. ARC over-
Figure 3: A sample sequence of refinements comes the scalability issues observed in the previous section.

ARC integrates data from concrete executions of the program with the results of statically analyzing an overapproximate, finite model of the program state space. It incrementally guides the construction of new test cases that increase code coverage, and discovers infeasible code elements that can be therefore excluded from the coverage count. ARC extends an initial test suite with new test cases that increase the code coverage, and at the same time computes a set of infeasible code elements to refine the coverage measurement. The approach is independent from the coverage criterion. Our prototype implementation, described in detail in Section 5, refers to branch coverage.

ARC shares the model and the test cases across multiple targets similarly to iDASH. Our original contribution is a process that introduces coarsening steps into iDASH. In other words, the process partially re-aggregates the states generated by the refinement process as the analysis of the program progresses. The rationale elaborates on the observation that every refinement in DASH aims to refine the coverage measurements that are independent from the coverage measurement. When ARC meets the goal, coarsening drops the refinements generated for the decision process.

Figure 4 shows the ARC pseudocode. ARC inputs a program P, a set of target elements T and a nonempty set I of program inputs. The target elements T are the elements to be covered, and the set of program inputs I is the initial test suite. It returns both a test suite that extends I and a set U \subseteq T of unreachable targets.

ARC works on a model M of the program P. The model is a labelled rooted graph where nodes represent abstract states, and are annotated with predicates over the program variables, while edges are annotated with the corresponding statements. An abstract state corresponds to a program location, and represents a set of concrete states that reach the location. Predicates on states identify subsets of concrete states. We say that a concrete state covers a node when it corresponds to the location represented by the node and satisfies the associated predicate.

ARC derives the initial model M₀ from the program P according to the coverage criterion to be satisfied, as the most conservative model that can be statically derived from P. When referring to control flow coverage criteria, as in the experiments reported in this paper, ARC initializes the model to the control flow graph of the program P. It annotates edges with the corresponding program statements and sets all predicates to true.

ARC iteratively executes the current test suite, and computes the set C of the nodes covered by at least one test case (line 16). Then, it coarsens the model by invoking the coarsen procedure described below (line 17) in the case of newly covered nodes, and updates the set of target elements (line 18). As defined in DASH, it tries to cover a not-yet-covered transition with a new test case. Otherwise, it refines the model, and searches all the unreachable nodes in it. Then, it coarsens the model (line 29) and removes the unreachable nodes from it.

Figure 4: ARC pseudocode
In more details, ARC adapts the DASH step that generates new test cases and extends the model as follows. It looks for a frontier transition within an abstract trace from the root to an uncovered target (lines 19–20). If there are no traces from the root to an uncovered target, all target elements have been either covered or excluded, and ARC terminates (lines 21–22). Otherwise, ARC calls the function \texttt{extend_frontier} (line 23) to cover the newly identified frontier transition. The function returns either a new test input \( i \) that covers the transition or a refinement predicate \( \text{RP} \) to refine the frontier. ARC either refines the model with \( \text{RP} \) (call to function \texttt{refine} at line 25) or adds the new test input \( i \) to \( \text{I} \) (line 36) before iterating (line 37). Function \texttt{extend_frontier} exploits conjunctive evaluation to build a test input and weakest precondition calculation to compute \( \text{RP} \). For a precise description of functions \texttt{extend_frontier} and \texttt{refine}, the readers can refer to [1].

To assist the coarsening step described below, ARC tracks the associations between the nodes and the refinements done to investigate their reachability. When the invocation of function \texttt{refine} splits a pre-frontier node \( n_{\text{pre}} \) according to a predicate \( \text{RP} \), ARC updates the map \texttt{split_for} to add the triple \((n_{\text{pre}}, \text{stmt}, \text{RP})\) to the set of triples associated to the post-frontier node \( n_{\text{post}} \).

Here we introduce the core ARC contribution, the coarsening step, by discussing function \texttt{coarsen} (lines 39–52). In a nutshell, ARC coarsens nodes when, after either covering a node or proving the node to be unreachable, it realizes that the refinements that originated that node are not required anymore. Function \texttt{coarsen} works with the map \texttt{split_for}, the model \( \text{N} \) and the set \( \text{N} \) of nodes to be coarsened, and reverses the refinements originated from the nodes in \( \text{N} \) as follows. For each node \( n_{\text{post}} \) in \( \text{N} \), it gets the originating refinements \((n_{\text{pre}}, \text{stmt}, \text{RP})\) from \texttt{split_for}(\text{post}), and for each pair identifies the companion set of \( n_{\text{pre}} \), \texttt{companions}(\text{pre}) (line 45), i.e., the set of nodes that correspond to the same program location of \( n \) and that have been annotated with \( \text{RP} \) or its negation. All nodes in a companion set derive from a common ancestor in the initial model. Then, for companion set \texttt{companions}(\text{pre}), it simplifies the refinement predicates of all the nodes in \texttt{companions}(\text{pre}) by removing \( \text{RP} \) or its negation from the predicate associated to each node (line 46) and puts back the edge to \( n_{\text{post}} \) removed during the refinement (line 47). Finally, ARC conservatively removes from the model all redundant nodes in each companion set (lines 48–51). An abstract state \( n \) is redundant if its associated predicate logically implies the disjunction of the predicates of its companions, signifying that all the concrete program states in \( n \) are also in its companions. Refinement and coarsening ensure that either two abstract states are disjoint, or one contains the other. Thanks to this fact, ARC detects redundancy of \( n \) by checking whether \texttt{predicate}(\text{pre}) is a logical consequence of at least one of its companions’ predicates (lines 49–50). This check can be done without the overhead of a decision procedure invocation, by syntactically comparing the clauses that compose the refinement predicates of the states.

Coarsening eliminates useless predicates and nodes. In this way, we can alleviate the scalability problems that derive from memory consumption and from the size and complexity of the predicates, and thus computation complexity. Our hypothesis is that the additional computational effort introduced by coarsening computation, and by the recomputation of some refinements that may be lost by coarsening, is counterbalanced by the reduced solver time because of shorter predicates. The empirical results reported in Section 6 indicate major improvements in computation time and scalability with respect to both eDASH and iDASH, that failed to scale when analyzing the programs of the experiments.

5. PROTOTYPE

We have implemented a prototype tool for branch coverage, \textsc{Star} (Software Testing by Abstraction Refinement), built on top of \textsc{Crest} \(^4\), an automatic test generation tool for C, based on conjunctive execution. \textsc{Crest} in turns relies on \textsc{Cil} \(^5\) for the instrumentation and static analysis of C code, and on the \textsc{Yices} \(^6\) \textsc{SMT} solver.

\textsc{Star} refines an abstract model that represents the branches of the program and the flow relations between them. The initial model is extracted from the static control flow graph. \textsc{Star} implements the iterative refinement and coarsening algorithm presented in Section 4 to determine the feasibility of the branches in the model, and exploits the conjunctive execution of \textsc{Crest} to investigate the feasibility of frontier transitions. \textsc{Star} selects frontier transitions with a heuristics that tries to minimize the size of the refinement predicates.

\textsc{Star} traces the coverage information against the model by running the program within the \textsc{Gdb} \(^7\) debugger. This allows \textsc{Star} to dynamically intercept the execution of each statement, determine the last executed branch and evaluate the corresponding predicates.

6. PRELIMINARY EVALUATION

We used the \textsc{Star} prototype to validate the technique proposed in this paper in terms of the ability of generating test suites to cover branches not yet covered and to identify infeasible branches.

Table 1 lists the 12 subject programs that we experimented with: \texttt{linsearch} and \texttt{binsearch} implement the linear and binary search of an integer datum in an array, respectively; \texttt{fcas} is the implementation of a component of an aircraft traffic control and collision avoidance system, as available from the Software-artifact Infrastructure Repository [6]; \texttt{week0..7} are programs that call function \texttt{calc_week} (from MySQL) that we described in Section 2 in different customized ways; \texttt{week} is a program that call function \texttt{calc_week} with no specific customization. The column \texttt{size} reports the program sizes in lines of code counted by the GNU utility \texttt{wc}. Column \texttt{br} reports the number of static branches of each program counted by \textsc{Star} \(^8\).

We used \textsc{Star} to maximize the branch coverage of each subject program starting from a randomly generated input test case. Table 1 reports the numbers of test cases that

\(^4\)http://code.google.com/p/crest/
\(^5\)http://sourceforge.net/projects/cil
\(^6\)http://yices.csl.sri.com
\(^7\)http://www.gnu.org/software/gdb
\(^8\)\textsc{Star} counts the static branches after the \textsc{Cil} pre-compilation pass that unrolls decisions with multiple conditions as an equivalent cascade of single condition decisions, and performs simple code optimizations based on constant propagation. For \texttt{calc_week}, the constant propagation determines slightly different counts of static branches across the different specializations of the program.
Table 1: Results of Star

<table>
<thead>
<tr>
<th>subject</th>
<th>size</th>
<th>br</th>
<th>tc</th>
<th>cbr</th>
<th>ibr</th>
<th>cov1</th>
<th>cov2</th>
</tr>
</thead>
<tbody>
<tr>
<td>linsearch</td>
<td>23</td>
<td>6</td>
<td>3</td>
<td>6</td>
<td>0</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>binsearch</td>
<td>39</td>
<td>12</td>
<td>6</td>
<td>12</td>
<td>0</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>tcas</td>
<td>180</td>
<td>106</td>
<td>22</td>
<td>99</td>
<td>7</td>
<td>93</td>
<td>100</td>
</tr>
<tr>
<td>week0</td>
<td>154</td>
<td>48</td>
<td>15</td>
<td>46</td>
<td>0</td>
<td>96</td>
<td>96</td>
</tr>
<tr>
<td>week1</td>
<td>154</td>
<td>48</td>
<td>17</td>
<td>46</td>
<td>0</td>
<td>96</td>
<td>96</td>
</tr>
<tr>
<td>week2</td>
<td>154</td>
<td>46</td>
<td>16</td>
<td>44</td>
<td>0</td>
<td>96</td>
<td>96</td>
</tr>
<tr>
<td>week3</td>
<td>154</td>
<td>46</td>
<td>13</td>
<td>44</td>
<td>0</td>
<td>96</td>
<td>96</td>
</tr>
<tr>
<td>week4</td>
<td>154</td>
<td>50</td>
<td>14</td>
<td>37</td>
<td>12</td>
<td>74</td>
<td>97</td>
</tr>
<tr>
<td>week5</td>
<td>154</td>
<td>50</td>
<td>15</td>
<td>47</td>
<td>1</td>
<td>94</td>
<td>96</td>
</tr>
<tr>
<td>week6</td>
<td>154</td>
<td>52</td>
<td>15</td>
<td>45</td>
<td>7</td>
<td>87</td>
<td>100</td>
</tr>
<tr>
<td>week7</td>
<td>154</td>
<td>52</td>
<td>16</td>
<td>45</td>
<td>7</td>
<td>87</td>
<td>100</td>
</tr>
<tr>
<td>TOTAL</td>
<td>1628</td>
<td>588</td>
<td>177</td>
<td>543</td>
<td>34</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

size: size in LOC
br: number of branches covered statically (after unrolling decisions with multiple conditions in equivalent cascade of single condition decisions)
tc: number of generated test cases
cbr: number of covered branches
ibr: number of identified infeasible branches
cov1: cbr / br [as percentage]
cov2: cbr / (br - ibr) [as percentage]

Table 2: Result of Crest and Klee

<table>
<thead>
<tr>
<th>subject</th>
<th>CrestStrand</th>
<th>CrestStdfs</th>
<th>CrestStcfg</th>
<th>Klee</th>
</tr>
</thead>
<tbody>
<tr>
<td>linsearch</td>
<td>33</td>
<td>83</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>binsearch</td>
<td>17</td>
<td>83</td>
<td>100</td>
<td>36</td>
</tr>
<tr>
<td>tcas</td>
<td>4</td>
<td>93</td>
<td>93</td>
<td>97</td>
</tr>
<tr>
<td>week0</td>
<td>87</td>
<td>83</td>
<td>85</td>
<td>97</td>
</tr>
<tr>
<td>week1</td>
<td>83</td>
<td>73</td>
<td>78</td>
<td>97</td>
</tr>
<tr>
<td>week2</td>
<td>54</td>
<td>85</td>
<td>85</td>
<td>97</td>
</tr>
<tr>
<td>week3</td>
<td>91</td>
<td>85</td>
<td>89</td>
<td>93</td>
</tr>
<tr>
<td>week4</td>
<td>46</td>
<td>50</td>
<td>52</td>
<td>94</td>
</tr>
<tr>
<td>week5</td>
<td>80</td>
<td>78</td>
<td>68</td>
<td>94</td>
</tr>
<tr>
<td>week6</td>
<td>85</td>
<td>77</td>
<td>81</td>
<td>97</td>
</tr>
<tr>
<td>week7</td>
<td>83</td>
<td>77</td>
<td>77</td>
<td>97</td>
</tr>
<tr>
<td>week</td>
<td>12</td>
<td>69</td>
<td>69</td>
<td>91</td>
</tr>
</tbody>
</table>

7. RELATED WORK

The research in the field of automated structural testing attracted considerable industrial as well as academic interest in the last decade. Most proposals rely on symbolic techniques to evaluate a program along a set of paths, and generate structural test cases by solving the resulting path constraints. The most successful tools and research prototypes exploit either symbolic execution (for instance EXE [4] and Klee [3]), or concolic (concrete-symbolic) execution (DART [9], Cute [14], Pex [15], Crest [2] and SAGE [10]). Most tools explore the executable program paths in some depth-first order. As a consequence, when executed for finite time against programs with infinitely many paths, they generate massive test suites but cover only small regions of the program state space.

Our approach exploits concolic execution to generate test cases, and maintains an abstract model of the frontier between covered and uncovered regions of the program state space, as relevant for the coverage criterion that is being addressed. Such abstract model steers the generation of
test cases towards yet uncovered elements, and is refined over non-executable transitions up to revealing infeasible elements. Identifying infeasible elements prevents the tool to infinitely try to cover infeasible code, and improves the coverage measurements. We are aware of some other approaches that monitor the coverage against the program control-flow graph to overcome the limitations of a full path exploration [2] and [10], but to the best of our knowledge our approach is first to integrate test case generation and proof of infeasibility.

A more recent research line recasts the problem as a model checking one by abstracting the program under test to a model, expressing the target coverage criterion in temporal logic formulas, and then returning the counter examples produced by the model checker as test cases [5, 12, 8]. As other software model checkers, these techniques experience problems to automatically build tractable but sufficiently detailed abstractions of the system under test.

8. CONCLUSIONS

This paper combines dynamic (concolic execution) and static (abstraction refinement) techniques to generate test suites with high structural coverage and precise coverage measurements. Addressing multiple code targets challenges automatic test case generators with demanding scalability requirements. To this end this paper introduces a new abstraction refinement and coarsening procedure that builds and improves over abstraction refinement. The preliminary experimental results are encouraging. Refinement and coarsening can analyze programs that are not handled by refinement alone, achieving in most cases higher coverage with smaller test suites than popular state-of-the-research test case generation tools.

Our research agenda is busy: We are working towards improving the abstraction refinement and coarsening procedure, gathering additional experimental evidence, investigating other coverage criteria as dataflow ones, better coping with solver incompleteness [9]. We are currently cleaning the preliminary prototype implementation to be able to experiment with industry-size software systems, assess the scalability of the approach, and distribute the tool as open source to gather results from independent users.

9. REFERENCES


