Structural coverage of feasible code

Article *in* Proceedings - International Conference on Software Engineering · January 2010

DOI: 10.1145/1808266.1808275

CITATIONS
18

READS
132

4 authors:

Mauro Baluda
Fraunhofer Institute for Secure Information Technology SIT
8 PUBLICATIONS **56** CITATIONS
SEE PROFILE

Pietro Braione
Università degli Studi di Milano-Bicocca
18 PUBLICATIONS **165** CITATIONS
SEE PROFILE

Giovanni Denaro
Università degli Studi di Milano-Bicocca
61 PUBLICATIONS **920** CITATIONS
SEE PROFILE

Mauro Pezzè
University of Lugano
189 PUBLICATIONS **4,001** CITATIONS
SEE PROFILE

Structural coverage of feasible code

Mauro Baluda[†] Pietro Braione[§] Giovanni Denaro[§] Mauro Pezzè[†§]

[†]University of Lugano
via Buffi 13, 6900
Lugano, Switzerland
mauro.baluda@unisi.ch

[§]University of Milano-Bicocca
Viale Sarca 336, 20126
Milano, Italy
{braione|denaro|pezze}@disco.unimib.it

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 pahts.

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 infea-

AST '10, May 2-8 2010, Cape Town, South Africa

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 satisfiability 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 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[1], 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,

[1]Section 6 gives more details on the static branch count.

```
1  /* Flags for calc_week() function.  */
2  #define WEEK_MONDAY_FIRST    1
3  #define WEEK_YEAR            2
4  #define WEEK_FIRST_WEEKDAY   4
5
6  typedef struct
7    TIME{uint year; uint month; uint day;} TIME;
8
9  /* Calc days since year 0 (from 1615) */
10 long calc_daynr(uint year, uint month, uint day);
11
12 /* Calc weekday from daynr: 0 for mon, 1 for tue... */
13 int calc_weekday(long daynr,
14                  bool sunday_first_day_of_week);
15
16 /* Calc days in a year. works with 0 <= year <= 99 */
17 uint calc_days_in_year(uint year);
18
19 /* Meaning of the bits in week_behaviour:
20    WEEK_MONDAY_FIRST (0): set ==> Mon, else Sun
21    WEEK_YEAR (1): set ==> Week in range 1-53, else 0-53
22    WEEK_FIRST_WEEKDAY (2): not set ==> ISO 8601:1988
23 */
24 uint calc_week(TIME *l_time,
25                uint week_behaviour, uint *year){
26   uint days;
27   ulong daynr =
28     calc_daynr(l_time->year, l_time->month, l_time->day);
29   ulong first_daynr = calc_daynr(l_time->year, 1, 1);
30   bool monday_first =
31     week_behaviour & WEEK_MONDAY_FIRST;
32   bool week_year = week_behaviour & WEEK_YEAR;
33   bool first_weekday =
34     week_behaviour & WEEK_FIRST_WEEKDAY;
35
36   uint weekday=calc_weekday(first_daynr, !monday_first);
37   *year=l_time->year;
38
39   if (l_time->month == 1 && l_time->day <= 7-weekday){
40     if (!week_year && (first_weekday && weekday != 0 ||
41                        !first_weekday && weekday >= 4))
42         return 0;
43     week_year= 1;
44     (*year)--;
45     first_daynr-= (days=calc_days_in_year(*year));
46     weekday= (weekday + 53*7- days) % 7;
47   }
48
49   if ((first_weekday && weekday != 0) ||
50       (!first_weekday && weekday >= 4))
51     days= daynr - (first_daynr+ (7-weekday));
52   else days= daynr - (first_daynr - weekday);
53
54   if (week_year && days >= 52*7){
55     weekday= (weekday + calc_days_in_year(*year)) % 7;
56     if (!first_weekday && weekday < 4 ||
57         first_weekday && weekday == 0){
58       (*year)++;
59       return 1;
60     }
61   }
62   return days/7+1;
63 }
```

Figure 1: The `calc_week` function of MySQL

only 37 out of 50 control flow branches of function `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 `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.

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, progressing 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 statement (error traces). If the model contains no error trace, then terminate, and confirm that the faulty statement is not reachable.

3. Identify a *frontier transition* in the model, i.e., a transition that belongs to an error trace, and connects an abstract state $s1$ covered by at least a test case t, to an abstract state $s2$ not covered by any test case.

4. Execute the program symbolically along the test case t up to state $s1$, change the path condition to reach state $s2$, 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.

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 ($!wp$)[2]. 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.

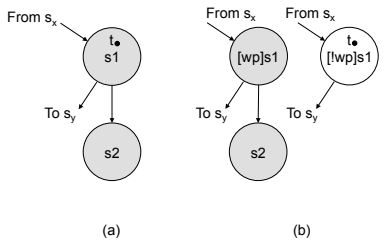


Figure 2: Refinement of an infeasible transition

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 $\overline{DASH}$.

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[3]. 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 naïve eDASH approach consists of calling a different instance of a DASH implementation for each target element. The less naïve iDASH approach operates on a single abstract model, and shares the set of identified test inputs, thus avoiding repeating the same refinements, and re-identifying the same test inputs across different targets.

Both approaches mark the code elements executed by any test case reported by $\overline{DASH}$ as *covered*, and the code elements that $\overline{DASH}$ proves to be unreachable as *infeasible*. Once executed for all *target* code elements, both approaches return the sets of covered and infeasible elements, and compute the obtained coverage cov as

$$cov = \frac{|covered|}{|target| - |infeasible|} \quad (1)$$

that yields a coverage indicator improved by the identified infeasible code elements. The readers should notice that both approaches can still yield partial coverage since $\overline{DASH}$ 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 $\overline{DASH}$. 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 $\overline{DASH}$ 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_week` 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; $s1 \rightarrow s3$ and $s2 \rightarrow s3$ 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 $\langle s1, s2, s3\rangle$ and $\langle s1, s3\rangle$, since these two traces share $s1$. The last refinement generates the predicate $c3 \wedge (!c1 \vee !c2)$ that identifies an abstract state yet to be covered, and contains both conjunctive and disjunctive operators.

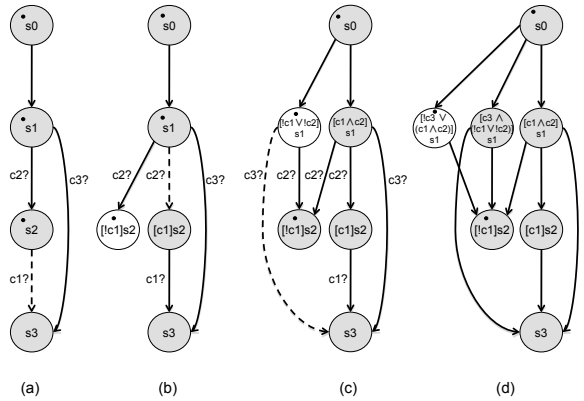
The amount of atomic clauses grows quickly with the size of the program. Our preliminary experiments led already to predicates with more than $300,000$ atomic clauses and a consequent explosion of solving time.

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-

[2]A slightly more sophisticated condition than the raw weakest precondition is required in presence of aliases [1]. Here we assumes 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.



Legend: All the transitions are associated to conditional statements with conditions `c2` ($s1 \rightarrow s2$), `c1` ($s2 \rightarrow s3$), and `c3` ($s1 \rightarrow s3$). Dotted lines indicate frontier transitions along which refinements are performed.

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 decide the reachability of an abstract state. 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 $\mathtt{U} \subseteq \mathtt{T}$ 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 $\mathtt{M}_0$ from the program `P` according to the coverage criterion to be satisfied, as the most

```
1  ARC(P, T, I):
2  //P is the program under test
3  //T is a set of target nodes to be covered
4  //I is a set of inputs for P (the initial test suite)
5
6  // nodes(M) is the set of nodes of the model M
7  // edges(M) is the set of edges of M
8  // paths(M) is the set of paths in M
9  // root(M) is the initial node of M
10
11   M := M_0 //M is the model (initially extracted from CFG)
12   U := {} //U is the set of the unreachable targets
13   C := {} //C us the set of the covered targets
14   split_for[nodes(M)] := {}
15   loop
16     C := all n ∈ nodes(M) s.t. n covered by run(P, I)
17     coarsen(M, C, split_for)
18     T := T - C - U
19     choose p ∈ paths(M), e = n_pre -stmt-> n_post ∈ edges(p) s.t.
20       p[0] = root(M) ∧ p[end] ∈ T ∧ n_pre ∈ C ∧ n_post ∉ C
21     if no such p, e exists
22       return I, U //test suite, unreachable nodes
23     ⟨i,RP⟩ := extend_frontier(P, I, e)
24     if i = ϵ //the chosen frontier cannot be extended
25       n'_pre := refine(M, e, RP)
26       if n'_pre = ϵ
27         N := all n ∈ nodes(M) s.t.
28              n unreachable from root(M)
29         coarsen(M, N, split_for)
30         remove from M all nodes in N
31         U := T - nodes(M)
32       else
33         split_for[n_post] := split_for[n_post] ∪
34                              {⟨n_pre,stmt,RP⟩}
35     else
36       I := I ∪ {i}
37   forever
38
39 coarsen(M, N, split_for):
40 // companions(n) is the companions set of a node n
41 // predicate(n) is the predicate which annotates n
42
43   for all n_post ∈ N
44     for all ⟨n_pre,stmt,RP⟩ ∈ split_for[n_post]
45       for all n'_pre ∈ companions(n_pre)
46         remove RP (or ¬ RP) from predicate(n'_pre)
47         edges(M) := edges(M) ∪ {n'_pre -stmt-> n_post}
48       for all n'_pre ∈ companions(n_pre)
49         if exists n''_pre ≠ n'_pre ∈ companions(n_pre) s.t.
50         predicate(n''_pre) ⟹ predicate(n'_pre)
51           remove from M the node n'_pre
52     split_for[n_post] := {}
```

Figure 4: ARC pseudocode

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.

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 `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 `RP` to refine the frontier. ARC either refines the model with `RP` (call to function `refine` at line 25) or adds the new test input `i` to `I` (line 36) before iterating (line 37). Function `extend_frontier` exploits concolic execution to build a test input and weakest precondition calculation to compute `RP`. For a precise description of functions `extend_frontier` and `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 `refine` splits a pre-frontier node $\mathtt{n}_{pre}$ according to a predicate `RP`, ARC updates the map `split_for` to add the triple $\langle \mathtt{n}_{pre}, \mathtt{stmt}, \mathtt{RP} \rangle$ to the set of triples associated to the post-frontier node $\mathtt{n}_{post}$.

Here we introduce the core ARC contribution, the coarsening step, by discussing function `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 `coarsen` works with the map `split_for`, the model `M` and the set `N` of nodes to be coarsened, and reverts the refinements originated from the nodes in `N` as follows. For each node $\mathtt{n}_{post}$ in `N`, it gets the originating refinements $\langle \mathtt{n}_{pre}, \mathtt{stmt}, \mathtt{RP} \rangle$ from $\mathtt{split\_for}(\mathtt{n}_{post})$, and for each pair identifies the *companion set* of $\mathtt{n}_{pre}$, $\mathtt{companions}(\mathtt{n}_{pre})$ (line 45), i.e., the set of nodes that correspond to the same program location of `n` and that have been annotated with `RP` or its negation. All nodes in a companion set derive from a common ancestor in the initial model. Then, for companion set $\mathtt{companions}(\mathtt{n}_{pre})$, it simplifies the refinement predicates of all the nodes in $\mathtt{companions}(\mathtt{n}_{pre})$ by removing `RP` or its negation from the predicate associated to each node (line 46) and puts back the edge to $\mathtt{n}_{post}$ 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 $\mathtt{predicate}(\mathtt{n})$ 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, STAR (Software Testing by Abstraction Refinement), built on top of CREST [4], an automatic test generation tool for C, based on concolic execution. CREST in turns relies on CIL[5] for the instrumentation and static analysis of C code, and on the YICES[6] SMT solver.

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. 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 concolic execution of CREST to investigate the feasibility of frontier transitions. STAR selects frontier transitions with a heuristics that tries to minimize the size of the refinement predicates.

STAR traces the coverage information against the model by running the program within the GDB[7] debugger. This allows 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 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: *linsearch* and *binsearch* implement the linear and binary search of an integer datum in an array, respectively; *tcas* is the implementation of a component of an aircraft traffic control and collision avoidance system, as available from the Software-artifact Infrastructure Repository [6]; *week0..7* are programs that call function `calc_week` (from MySQL) that we described in Section 2 in different customized ways; *week_* is a program that call function `calc_week` with no specific customization. The column *size* reports the program sizes in lines of code counted by the GNU utility `wc`. Column *br* reports the number of static branches of each program counted by STAR [8].

We used 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] STAR counts the static branches after the CIL precompilation 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 `calc_week`, the constant propagation determines slightly different counts of static branches across the different specializations of the program.

subject	size	br	STAR tc	cbr	ibr	cov_1	cov_2
linsearch	23	6	3	6	0	100	100
binsearch	39	12	6	12	0	100	100
tcas	180	106	22	99	7	93	100
week0	154	48	15	46	0	96	96
week1	154	48	17	46	0	96	96
week2	154	46	9	44	0	96	96
week3	154	46	13	44	0	96	96
week4	154	50	14	37	12	74	97
week5	154	50	15	47	1	94	96
week6	154	52	15	45	7	87	100
week7	154	52	16	45	7	87	100
week_	154	72	32	72	0	100	100
TOTAL	1628	588	177	543	34	-	-

size: size in LOC
br: number of branches computed 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
cov_1: cbr / br [as percentage]
cov_2: cbr / (br - ibr) [as percentage]

Table 1: Results of STAR

STAR generated for each program (column *tc*), the numbers of covered branches (column *cbr*), the number of branches that STAR identified as infeasible (column *ibr*), and the coverage computed with respect to the set of branches identified statically both before (column cov_1) and after pruning the ones identified as infeasible (column cov_2).

STAR generated a total of 177 test cases that cover 543 out of 588 branches, and identified 34 infeasible branches, failing only for 11 branches. All runs completed within minutes on a common laptop. STAR produces test suites of manageable size that cover most feasible branches (100% in many cases and 96% in the worst cases). The table shows also that STAR improves the measurement of branch coverage (column cov_2) wrt to coverage measurements computed without accounting for infeasible branches (column cov_1). The improvement is evident for *tcas* (from 93% to 100%), *week6* and *week7* (from 87% to 100%), *week5* (from 94% to 96%), and *week4* (from 76% to 97%) where the improvement is maximum (+21%)

We compared the effectiveness of ARC against plain random testing, directed random testing and automatic generation of test suites for branch coverage, as implemented by two test case generation tools, CREST and KLEE. CREST generates test cases either randomly or based on concolic execution, and in this latter case can be configured for either depth-first search or control-flow graph guided path exploration [2]. Hereafter we refer to these three modes of the tool as CRESTrand, CRESTdfs and CRESTcfg, respectively. KLEE tries to maximize branch coverage by means of a more traditional approach based on static, depth-first symbolic execution [3].

Table 2 reports the branch coverage obtained, respectively, with CRESTrand, CRESTdfs, CRESTcfg and KLEE on the same subject programs of Table 1. As in the case of STAR, we allocated a maximum of 60 minutes for each tool to complete the analysis of each program. We executed KLEE with the option that searches and eliminates statically identifiable dead code at the beginning of the analysis. In standard mode results were horribly poor.

subject	Reported coverage (%) CRESTrand	CRESTdfs	CRESTcfg	KLEE
linsearch	33	83	$100^{=}$	50
binsearch	17	83	$100^{=}$	36
tcas	4	93	93	97
week0	87	83	85	97^{+}
week1	83	83	83	97^{+}
week2	54	85	85	97^{+}
week3	91	85	89	93
week4	46	50	52	94
week5	80	78	68	94
week6	85	77	81	97
week7	83	77	77	97
week_	12	69	69	91

$^{=}$ $^{+}$: equals to or greater than cov_2 from Table 1

Table 2: Result of CREST and KLEE

Table 2 marks the few cases where a tool performs as well or better than STAR. CREST reaches a much lower coverage than STAR, except for two cases, where both tools cover all branches (*linsearch* and *binsearch*). KLEE reaches a coverage comparable to STAR: it outperforms STAR by 1% in three cases (*week0..2*) and produces much lower coverage only for *linsearch* and *binsearch*. For the sake of precision, we remark that the counts of the static branches differs for KLEE and STAR. KLEE counts the number of branches based on the raw number of decisions in the program, while STAR unrolls the decisions with multiple conditionals. Thus the data of KLEE and STAR are not completely comparable, and STAR approximate finer condition coverage metrics, like MC/DC (Modified Condition Decision Coverage [13]), better than KLEE. We also observe that KLEE does never produce 100% coverage, which suggests that the combined static-dynamic analysis of STAR works better than the static dead code analysis of KLEE, to identify infeasible code.

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

[1] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA 2008)*, pages 3–14, 2008.

[2] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, pages 443–446, 2008.

[3] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, 2008.

[4] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS '06)*, pages 322–335, New York, NY, USA, 2006. ACM.

[5] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using model-checking. In *Proceedings of the 1996 SPIN Workshop (SPIN 1996). Also WVU Technical Report NASA-IVV-96-022.*, 1996.

[6] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[7] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, 1988.

[8] G. Fraser, F. Wotawa, and P. E. Ammann. Testing with model checkers: a survey. *Software Testing, Verification and Reliability*, 19(3):215–261, Sept. 2009.

[9] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI 2005)*, pages 213–223, 2005.

[10] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing, 2007.

[11] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: A new algorithm for property checking. In *Proceedings of the 14th ACM SIGSOFT symposium on Foundations of Software Engineering (FSE-14)*, pages 117–127, 2006.

[12] H. S. Hong and I. Lee. Automatic test generation from specifications for control-flow and data-flow coverage criteria. In *Proceedings of the Monterey Workshop, Monterey, Calif.: Naval Postgraduate School*, pages 230–246, 2001.

[13] M. Pezzè and M. Young. *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, April 2007.

[14] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-13)*, pages 263–272, 2005.

[15] N. Tillmann and J. de Halleux. Pex — white box test generation for .net. In *Proceedings of the 2nd International Conference on Tests and Proofs (TAP 2008)*, pages 134–153, 2008.

[16] United States. *RTCA, Inc., Document RTCA/DO-178B*. U.S. Department of Transportation, Federal Aviation Administration, Washington, D.C., 1993.

[17] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 97–107, 2004.

[18] E. J. Weyuker. The evaluation of program-based software test data adequacy criteria. *Communications of the ACM*, 31(6):668–675, 1988.