Abstract—Automated test generation helps programmers to test their software with minimal intervention. Automated test generation tools produce a set of program inputs that maximize the possible execution paths, presented as a test coverage metric. Proposed approaches fall within three main approaches. Search-based methods work on any program by randomly searching for inputs that maximize coverage. Heuristic-based methods can be used to have better performance than pure random-search. Constraint-based methods use symbolic execution to restrict the random inputs to those guaranteed to explore different paths. Despite making the execution slower and supporting very few programs, these methods are more efficient because the search space is vastly reduced. The third approach combines the previous two to support any program and takes advantage of the space search reduction when able, at the cost of slower execution. We propose a fourth approach that also refines search-based with constraints. However, instead of requiring a slower symbolic execution when measuring coverage, constraints are statically extracted from the source code before the search procedure occurs. Our approach supports all programs (as in Search-Based) and reduces the search-space (as in Constraint-based methods). The innovation is that static analysis occurs only once and, despite being less exact than symbolic execution, it can significantly reduce the execution cost in every coverage measurement. This paper introduces this approach, describes how it can be implemented and discusses its advantages and drawbacks.

Index Terms—Evolutionary Computation, Test Generation, Program Synthesis

I. INTRODUCTION

Testing plays a vital role in the software development process [1]. However, testing often involves the tedious and error-prone task of manually writing test cases. Automated test generation reduces this burden by automatically generating program inputs that exercise different execution paths. Proposed methods fall within three main approaches: Search-based, Constraint-based and hybrid approaches. Figure 1 shows the tradeoffs between families of test generation techniques in terms of search efficiency and supported program features (e.g., robustness against non-deterministic behavior).

Search-based test generation techniques evolve test suites [2] maximizing “test adequacy” using test coverage metrics (e.g., statement, branch, MC/DC) [3]. By treating the System Under Test (SUT) as a black box, search-based techniques are generic and can scale to any program. Although effective, their performance depends on whether the heuristic provides sufficient guidance [4].

Constraint-based test generation techniques, such as Seeker [5], and the one used by Achour and Benattou [6], use Static and Dynamic Symbolic Execution (DSE) to maximize coverage by synthesizing inputs that exercise different execution paths.

Although these approaches do not require heuristics, they are limited in their applicability and the number of supported program features. Furthermore, the underlying constraint solvers have considerable computational costs of symbolic execution. Moreover, a recent study [7] identified different categories of problems where the application DSE-based approaches face challenges, from which the following are considered in this work: Environment: dealing with unknown and non-deterministic behavior in the program; and, Constraint solving: efficiency issues when dealing, for instance, with non-linear arithmetic and complex data structures.

Hybrid approaches have successfully combined search- and constraint-based [8, 9, 10, 11, 12, 13] to obtain higher coverage than search-based techniques in fewer iterations and the same or higher coverage than DSE. In programs unsupported by symbolic execution (e.g. dynamically complying with the values in an object [13]), hybrid approaches have the same performance as pure search-based ones.

In this paper, we propose an approach that augments search-based techniques with optimistic static analysis-guided input generation.

Our approach requires a white box view of the SUT (i.e., source code access) but does not restrict the program. If the constraint solver does not support a given condition, that condition contributes to the heuristic measure, and, unlike in DSE, it does not prevent the extraction of conditions from the remainder of the program. The advantage over pure search-based methods is the guarantee of the diversity in the initial population w.r.t. the supported conditions. Furthermore, mutation operators do not reduce this diversity. This diversity acts as a proxy for coverage as the inputs exercise the different branching conditions’ paths.
Fig. 1. A comparison of test generation techniques in terms of their efficiency (x-axis) and supported programs (y-axis).

II. MOTIVATIONAL EXAMPLE

Listing 1 presents a function that transfers an amount of money \( x \) from one account \( a1 \) to another \( a2 \) provided that the transfer is not deemed to be fraudulent (randomly checked one-tenth of the times), and sufficient funds exist to complete the transfer. The \( \text{check\_fraud} \) and \( \text{get\_tax} \) function validate and obtain information from a server, respectively.

```python
def transfer(x: int, a1: Account, a2: Account):
    if x % 10 == 0:
        if check_fraud(x, a1, a2):
            raise FraudException()
    if x >= 100:
        tax = get_tax(x)
        if x - tax < a1.amount:
            raise NoBalanceException()
    else:
        tax = 0.0
    a1.amount -= (x + tax)
    a2.amount += (x)
    return a2
```

Listing 1. Function \( \text{transfer} \) transfers an amount of money \( x \) between two accounts in Python.

In this example, DSE suffers from a couple of important limitations. Firstly, pure DSE approaches are incapable of handling the \( \text{check\_fraud} \) and \( \text{get\_tax} \) functions. Both functions represent invocations over code that is not accessible, i.e., available in dynamically linked libraries requiring server calls (Lines 3 and 7), and so, it cannot be analyzed. Furthermore, both calls can have a non-deterministic behaviour: Hybrid approaches are more robust in this sense. Although DSE cannot reason about the specified conditional expressions, SBST techniques try to overcome this limitation by blindly searching for values that hold for the entire conditional expression. However, conditional expressions not supported by DSE do not provide extra help in guiding the search-based techniques, reducing the search efficiency.

Secondly, test inputs generated by DSE suffer from a lack of diversity. SMT solvers can efficiently find an input that satisfies a given condition, but is ill-suited for producing a diverse set of such inputs. The resulting lack of diversity within test inputs is detrimental to the exploration of the search space within hybrid approaches [14]. These two limitations motivate developing an approach capable of reusing the SMT-unsupported conditional expressions as heuristics and generating diverse values. Both these requirements are useful in improving the search. More exact heuristic metrics and a diverse population are determinant in the evolutionary algorithms used by Search-based approaches.

III. APPROACH

Our approach uses optimistic static analysis to restrict the search-based procedure.

Figure 2 illustrates the three high-level steps of our approach: Firstly, the conditional expressions within the SUT are extracted (Section III-1); Secondly, the conditional expressions are categorized in terms of their static verifiability (Section III-2); Finally, the conditional expressions are used to synthesize and evolve populations of test inputs via a multi-objective genetic algorithm (Section III-3).

1) Propagate and extract the conditionals: The first step of the approach works quite similar to DSE, but statically: Conditional expressions are extracted from the function code. To ensure more meaningful and correct properties, the definitions of the used variables in the conditional expressions are propagated throughout the program, as happens with the \( \text{tax} \) declaration. To maximize the coverage of the \( \text{transfer} \) function, the system should be capable of generating input variables for the \( x, a1, \) and \( a2 \) variables that comply with the following conditional expressions, and where the infeasible conditional expressions are filtered.
1) \( x \% 10 == 0 \)
2) \( \text{not}(x \% 10 == 0) \)
3) \( x \% 10 == 0 \) and 
   \( \text{check}_\text{fraud}(x, a1, a2) \)
4) \( x \% 10 == 0 \) and 
   \( \text{not}(\text{check}_\text{fraud}(x, a1, a2)) \)
5) \( \text{not}(x \% 10 == 0) \) and \( x >= 100 \)
6) \( x \% 10 == 0 \) and 
   \( \text{not}(\text{check}_\text{fraud}(x, a1, a2)) \) \( x >= 100 \)
7) \( \text{not}(x \% 10 == 0) \) and \( x >= 100 \) and 
   \( x - \text{get}_\text{tax}(x) < a1.\text{amount} \)
8) \( \text{not}(x \% 10 == 0) \) and \( x >= 100 \) and 
   \( \text{not}(x - \text{get}_\text{tax}(x) < a1.\text{amount}) \)
9) \( x \% 10 == 0 \) and 
   \( \text{not}(\text{check}_\text{fraud}(x, a1, a2)) \) \( x >= 100 \) and 
   \( x - \text{get}_\text{tax}(x) < a1.\text{amount} \)
10) \( x \% 10 == 0 \) and \( x >= 100 \) and 
    \( \text{not}(\text{check}_\text{fraud}(x, a1, a2)) \) \( x - \text{get}_\text{tax}(x) < a1.\text{amount} \)
11) \( \text{not}(x \% 10 == 0) \) and \( \text{not}(x >= 100) \)

To ensure reachability for a specific condition, \( c_1 \), all 
the previous conditional expressions present in the possible 
exection paths are prepended to it.

2) **Categorize conditional expressions:** Conditional expres-
sions are split into two sets in terms of whether or not they 
are statically verifiable using an SMT solver. The use of language 
features or unknown or non-deterministic functions prevent a 
constraint from being statically verifiable.

3) **Assign conditions to individuals:** Figure 2 illustrates the 
use of these conditions throughout the evolution.

Individuals in the initial population are assigned one of 
the constraint conditions. This, thus is expected that the initial 
population will be diverse regarding the conditions that they 
fulfill.

The statically-verifiable component of the condition will 
be used to synthesize input values. We rely on the non-
deterministic synthesis presented in Refined Typed Genetic 
Programming [15] to ensure value diversity, typically not 
provided by SMT solvers.

To maintain this value diversity in future generations, muta-
tions to an individual preserve the same constraints. The same 
synthesis procedure is used from the same constraints, but 
different values can be generated, thus mutating the individual. 
Similarly, recombination should comply with the statically 
verifiable conditions of each parent.

While preserving the conditions assigned to an individual 
during the evolution may restrict the possible combinations 
of operators, this is how we can reduce the search space, 
 improving search efficiency.

Fitness evaluation also takes into account the constraint con-
ditions, but only those that are not statically guaranteed. These 
boolean conditions are converted to a continuous function, 
using one of several available methods [16, 17]. Continuous 
functions have finer granularity, thus being more useful in 
heuristic methods. This fitness function can be combined with 
the main test coverage metric, augmenting the heuristic with a 
test-case diversity. This approach is more robust in preventing 
a single high-coverage test from eliminating other smaller but 
complementary tests from the population.

At the end of the evolutionary process, different individuals 
can be combined to create a test suite that will guarantee 
diversity in terms of the source code’s extracted conditions.

Most hybrid approaches between search-based and 
constraint-based methods rely on DSE.

In general, three main directions have been explored: (1) 
integration of DSE within the search, e.g., by using DSE as a 
single mutation operator [10] or to aid the genetic operations 
in the evolutionary algorithm [18]; (2) integration of a search 
algorithm within DSE [8, 19] to, e.g., solve floating-point 
constraints [8]; and (3) adaptive integration of DSE and 
search, whereas the hybrid approach switches between DSE 
and search to explore other properties of the software under 
test or other areas of the search space [13, 20].

Our approach does not rely on DSE, but it shares similarities 
with the first group: it uses the extracted constraints to restrict 
the initial population and genetic operators like recombination 
and synthesis.

But there are significant differences between using opti-
mistic static analysis and DSE. Optimistic Static Analysis 
requires access to the source code, while DSE can be in-
strumented to existing binaries. Because not all source code 
is available, some constraints in runtime-linked code may 
not be extracted. DSE does not suffer from this issue as 
instrumentation can occur at runtime. However, DSE intro-
duces overhead with this instrumentation, which is constant at 
every generation for each individual. Our approach introduces 
overhead in evaluation because it does not modify the 
program. Instead, only a one-time overhead occurs before the 
evolution. We believe this presents a significant computational 
cost reduction. The time saved by not instrumenting the code 
can be used to explore more inputs in the search procedure. 
Finally, our approach is optimistic because if SMT solvers do 
not support one constraint, it is still used to improve the fitness 
function. DSE cannot support several features of programming 
languages, thus excluding those programs from using these 
techniques. Our approach supports the same programs are pure 
search-based approaches.

IV. CONCLUSION

Automatic testing plays an essential role in the development 
of software. Different techniques were introduced to help 
programmers automate test generation. Hybrid generation-
based testing combines constraint-based techniques, widely 
known for their search efficiency, with search-based techniques 
known for their robustness and extensive support of program 
features. Nevertheless, hybrid approaches are limited by the 
same aspects of the underlying techniques. Hybrid approaches 
rely on DSE, which does not produce diverse values, essential 
for search-based algorithms. Additionally, SBST techniques 
require the help of DSE to improve its search efficiency, but
the limited support for programming language features makes hybrid approaches limited in the real-world.

This work proposes to overcome both limitations, using optimistic static analysis instead of DSE.

Similar to other approaches, this technique tries to generate tests that maximize program coverage efficiently. We introduced the categorization of conditional expressions to improve the search-based heuristics’ expressiveness and use a non-deterministic synthesizer to maximize generated expressions’ diversity. Finally, we presented an example where this approach is more efficient than existing approaches.

V. ACKNOWLEDGEMENTS

This work was supported by the Fundação para a Ciência e a Tecnologia (FCT) under LASIGE Research Unit, ref. (UIDB/00408/2020) and (UIDP/00408/2020), the GADgET project (DSAIPA/DS/0022/2018), the CMI-Portugal project CAMELOT (POCI-01-0247-FEDER-045915), and the U.S. Air Force Research Laboratory (#OSR-4066). The authors are grateful for their support. Any opinions, findings, or recommendations expressed are those of the authors and do not necessarily reflect those of the US Government.

REFERENCES