Test Automation Meets Static Analysis

Jan Peleska*
University of Bremen
jp@tzi.de

Helge Löding†
University of Bremen
hloeding@tzi.de

Tatiana Kotas*
Verified Systems International GmbH
kotas@verified.de

Abstract: In this article we advocate an integrated approach for the automation of module or software integration testing and static analysis. It is illustrated how fundamental methods of static analysis, in particular abstract interpretation by interval analysis, contribute to the solution of problems typically encountered in the field of automated test case/test data generation. Conversely, test data generation algorithms are useful to improve results obtained in static analyses: Potential errors identified in the unit under test (UUT) during an analysis can be confirmed by constructing concrete test data leading to the erroneous UUT state. False alarms resulting from over-approximating abstractions applied during the analyses can be uncovered using test automation algorithms disproving the reachability of associated code portions and program states.

1 Introduction

In the general literature discussing software quality assurance, there is a common understanding that the software verification process should be supported by both tests and static analyses [Lig02], the latter ranging from informal inspections to rigorous application of formal methods [CCF+06]. This understanding is also reflected by the applicable standards for – potentially safety-related – software development in avionics [SC-92] and railway control. Moreover, it is advisable that the persons performing tests simultaneously perform the associated analyses, since the more intimate knowledge of the unit under test (UUT), which is usually gained from the inspections, helps to specify more relevant test data and more comprehensive test oracles. As a consequence, integrated tool support for software testing and static analysis is desirable from the perspective of verification experts responsible for performing these tasks within a software development project. From the tool builders' perspective, it turns out that test automation and static analysis share a considerable amount of common methodology as well as concrete techniques and algorithms. Therefore, the objective of this article is twofold: First, we illustrate how fundamental techniques from static analysis, in particular abstract interpretation, are important pre-

*Partially supported by the BIG Bremer Investitions-Gesellschaft under research grant 2INNO1015B.
†Partially supported by Siemens Transportation Systems in Braunschweig.

requisites for the solution of the test case/test data generation problem. Second, it is shown how an integrated test case/test data generation component can be applied to improve static analysis tools by confirming potential UUT errors with concrete test data leading to erroneous program states, or uncovering false alarms by disproving the reachability of these undesired states.

2 Combined Module Testing and Static Analysis

Functional and Structural Module Testing. Recall that for *structural module testing*, test cases are derived from the objective that certain aspects of the code control structure (statements, branches, atoms in branching conditions etc.) should be covered during test executions. In *functional testing* the input data to the UUT is derived from functional specifications, such as pre- and postconditions and intermediate assertions (e. g. invariants) describing the expected module behaviour. It is obvious that a functional specification is also required for structural testing, in order to check the module behaviour when executed with the test cases derived from the code structure. In any case, automated test data generation boils down to the solution of reachability problems, typically represented by edges, states, paths or regions of control flow graphs (CFGs) representing the UUT: In structural testing, we aim to cover nodes (for statement coverage) or branches (for decision coverage) of the CFG. For functional testing consider some UUT $f()$ to be tested against precondition $P(v)$ and a postcondition $Q(v, v@pre)$, where $v@pre$ denotes the pre-state of the variable symbols whose valuation may be changed by execution of $f()$. For illustration purposes, assume that $Q(v, v@pre)$ can be written as a conjunction of implications

$$Q(v, v@pre) \equiv \bigwedge_{i=1}^{k} (C_i(v, v@pre) \Rightarrow Q_i(v, v@pre))$$

so that a functional test should at least find test data such that every situation $C_i(v, v@pre)$ is covered, while always observing the precondition $P(v)$. Now consider the augmented UUT function

```
f_aug(...) {
 if ( P(v) ) {
    f(...);
    if ( C_1(v,v@pre) ) { assert(Q_1(v)); /* B1 */ }
    ...
    if ( C_k(v,v@pre) ) { assert(Q_k(v)); /* Bk */ }
}
```

Then functional test data generation for $f()$ requires coverage of `f_aug()`-branches `B1, B2, ...,Bk`.

Core components for test case generation tools. It is interesting to note that the core components of test automation systems (Fig. 1) were already suggested in 1976 by Ramamoorthy *et. al.* [RHC76]: (1) The *path selector* identifies program paths p which are suitable candidates for execution according to the underlying (structural or functional) test

strategy. (2) Based on the branching conditions along p and the sequential statements between branches, the *constraint generator* constructs the (first-order) predicates to be solved for the (sequence of) input variables in order to stimulate the execution of path p. (3) The *constraint solver* constructs a solution, that is, an appropriate value assignment to the input variables, for the given constraints or indicates why p is *infeasible*, that is, why no value assignment resulting in execution of p can be found.

Intermediate SUT model representation. In order to support various formalisms and test paradigms, such as code-based white-box and model-based black-box testing, the system under test (SUT) and/or its specification are first transformed[1] into an *intermediate model representation (IMR)* which is independent on the concrete SUT code or specification syntax (Fig. 1): The IMR consists of a class library allowing to encode hierarchic hybrid transition systems. For testing C++ modules (methods or C-functions), a compilation front-end derived from `gcc` [Löd] parses the SUT code and generates (1) a control flow graph (CFG) representation in 3-address code of the SUT and (2) a detailed information base supporting queries about types, variables/objects and their sizes. Based on these information, the IMR model of the SUT is instantiated. (3) Sub-functions/methods invoked by the SUT are represented by their own CFGs, and function/sub-function relation defines a hierarchy between them. The type and variable information base contains scope information, so that identically named local variables of different methods, as well as synonymous global, static and local variables can be distinguished.

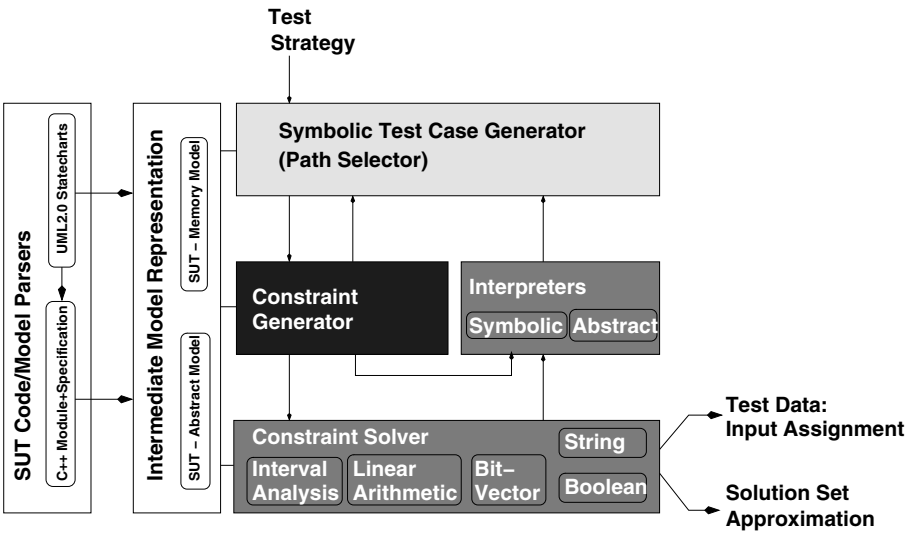

Figure 1: Generic architecture for test automation systems.

For functional testing, the UUT CFG is augmented by graph components representing pre- and postconditions and intermediate assertions as indicated above.

[1] Currently, parser front-ends for C/C++ and UML 2.0 Statecharts are available.

Abstract Interpretation by Interval Analysis. Given a path through a CFG as suggested by the symbolic test case generator and the associated collection of constraints it remains to decide about the solution strategies to be applied by the solver. To this end, we observe that (1) one path can generally be covered by more than one input vector and (2) it is often desirable (e. g. for the purpose of *boundary value testing*) to be able to approximate the full solution set $\mathbb{S}$ for a given conjunction of constraints. As a consequence a solver should not only be capable to calculate one isolated solution $v \in \mathbb{S}$. These consideration suggest an *abstract interpretation* approach by means of *interval analysis*: For each variable symbol x we consider interval valuation $I(x) = [\underline{x}, \overline{x}]$. This valuation is performed in the infinite-hight lattice of intervals which has been extensively studied within the context of abstract interpretation as well as numerical analysis [JKDW01, BFPT06]. The partial order $\sqsubseteq$ is given by the subset relation between intervals; the smallest and largest elements are $\bot = \{\}$ (empty interval) and $\top = [-\infty, \infty]$, respectively. Given intervals $I = [\underline{i}, \overline{i}], J = [\underline{j}, \overline{j}]$, their least upper bound is given by the *interval hull* $I \sqcup J = [min\{\underline{i}, \underline{j}\}, max\{\overline{i}, \overline{j}\}]$ and their greatest lower bound is given by $I \sqcap J = I \cap J$. Operations ω on program variables are lifted to their abstracted monotonic equivalents by

$$I[\omega]J = [min\{x\ \omega\ y \mid x \in I \wedge y \in J\}, max\{x\ \omega\ y \mid x \in I \wedge y \in J\}]$$

which can be evaluated very efficiently if the operations ω themselves are monotonic: For example, $[\underline{x}, \overline{x}][+][\underline{y}, \overline{y}] = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$ and $[exp]([\underline{x}, \overline{x}]) = [exp(\underline{x}), exp(\overline{x})]$. Integral-valued variables are abstracted into the sub-lattice of intervals with integer bounds, and Boolean valuations are represented as single-point integer intervals $[1, 1](=$ `true`$)$ and $[0, 0](=$ `false`$)$. Interval analogues of arithmetic constraints can be handled in three-valued logic, for example, $[\underline{x}, \overline{x}][<][\underline{y}, \overline{y}]$ evaluates to `true` if $\overline{x} < \underline{y}$, to `false` if $\underline{x} \geq \overline{y}$ and otherwise to `undecided`$(= [0, 1])$.

Numerical interval analysis provides a wide variety of solvers for interval constraints. In contrast to Boolean SAT solvers, for example, interval analysis solvers can handle Boolean, integer and floating point arithmetics, including transcendent mathematical function, within the same framework. We will now illustrate the application of this theory for both static analysis and test case generation purposes in the examples below.

Example 1: Identification of unreachable code. Consider structural test case generation for C function

```
1  double globx;
2  double globy;
3
4  double f(double x, double y, int i) {
5      double z;
6      int j, k, error0 = 0;
7      if ( i < 0 ) k = 0; else k = i;
8      if ( (x < 5.0) and (y < exp(x)) ) x = x - y - globy;
9      else x = y - x - globx;
10     for ( k += 1; k < x and error0 == 0; k *= 2) {
11         if ( 0 >= k ) error0 = 1;
12     }
13     if ( error0 == 0 ) z = log((double)k-x); else z = 0;
14     return z;
15 }
```

with precondition $P_1 \equiv x, y, globx, globy, i \in [-10, 10]$. Before even starting to generate test data suitable to cover certain branches or statements it is advisable to perform a quick check whether some code portions are obviously unreachable. This can be performed using abstract interpretation by interval analysis, with the following execution rules: (1) The effect of constant assignments $x = c;$ on interval level is the single-point interval $I(x)[x = c;] = [c, c]$. (2) The effect of a variable assignment $x = y;$ is assignment of intervals, $I(x)[x = y;] = I(y)$. (3) The effect of expression evaluation $x\ \omega\ y$ is the interval $I(x)[\omega]I(y)$. (4) The effect of a branching condition `if (C) B_1 else B_2` on a variable x is $I(x)[B_1]$ if $I[C] = [1, 1]$ (that is, the interval interpretation of condition C evaluates to `true`), $I(x)[B_2]$ if $I[C] = [0, 0]$ (i. e. interval evaluation of C to `false` and $I(x)[B_1] \sqcup I(x)[B_2]$ otherwise (i. e. interval evaluation of C to `undecided`).

Applying these rules to the conditional statement in line 7 results in effect $I(k)[k = 0;] \sqcup I(k)[k = i;] = [-10, 10]$, because $I(i) = [-10, 10]$, so $I(i)[<][0, 0] = [0, 1]$. Using constraint propagation techniques for this abstract interpretation, we can improve this result by noting that assignment $k = i;$ only happens for the subinterval $[0, 10]$ of $I(i)$, so the resulting interval for k can be contracted by $[0, 10]$. The effect of the interval interpretation of lines 8 — 9 is calculated in a similar way, and neither if- nor else branch in lines 8,9 can be proven to be unreachable. The interval valuation at the entry of the for loop in line 11 is $I(k) = [0, 10]$, $I(error0) = [0, 0]$, $I(x) = [-30, 30]$. The interval interpretation of the loop in lines 10 — 12 is repeated until a fixpoint is reached[2] with valuations $I(k) = [1, 352]$, $I(error0) = [0, 0]$, $I(x) = [-30, 30]$ (again, the result for $I(k)$ can be further contracted to $I(k) = [1, 32]$). This proves that the if-branch of line 11 can never be reached, and, since $I(error0) = [0, 0]$ implies that $I(error0)[==][0, 0]$ evaluates to `true`, this also shows that the else-branch of line 13 is unreachable.

Example 2: Test case generation by interval constraint solution. Only after having ruled out (some) unreachable branches, the path generator starts to select paths for the constraint generator, leading to an activation of the solver. Suppose we need a test case for covering the if-branches of lines 7 and 8, such that the for-loop in line 10 is skipped. This path leads to generation of constraints

$$x, y, globx, globy, i \in [-10, 10] \wedge (i < 0) \wedge (x < 5.0) \wedge (y < exp(x)) \wedge (1 \geq x - y - globy)$$

the last conjunct resulting from the symbolic execution performed by the constraint generator along the suggested path: Since the if-branches have been selected in lines 7—8, k has value $0 + 1$ after loop initialisation and x evaluates according to the assignment $x = x - y - globy$ in line 8. The solver separates the first conjunct from the others since they do not share any variables and assigns maximal interval solution $I(i) = [-10, -1]$. Inequation $x < 5.0$ and precondition induce $I(x) \subseteq [-10, 5[$, so it remains to solve

$$(y < exp(x)) \wedge (1 \geq x - y - globy)$$

while observing $I(x) \subseteq [-10, 5[\wedge I(y), I(globy) \subseteq [-10, 10]$. Interval constraint solvers for this purpose proceed as follows: (1) Check that each conjunct at least evaluates to $[0, 1]$

[2] Observe that for more general loops existence of a fixpoint is not guaranteed since the interval lattice is of infinite height, so that the application of a *widening operation* may be required.

(undecided) with the current interval assignment. If, for example, $I(y)[<][exp](I(x))$ evaluates to $[0,0]$ the constraint system is unsolvable with *interval vector* $(I(x), I(y))$. (2) If one conjunct evaluates to $[1,1]$ with vector $V = (I(x), I(y), I(globy))$ it is solved, and monotonicity guarantees that all interval vectors which are subsets of V remain solutions. (3) If at least one conjunct is still in solution state undecided, perform *bi-partitioning* of an interval $I(v), v \in \{x, y, globy\}$ (typically, the one with largest diameter) into $I_0(v) = [Inf(I(v)), Sup(I(v))/2]$ and $I_1(v) = [Sup(I(v))/2, Sup(I(v))]$. (4) Continue with step (1), using new vectors $V_0 = (\ldots, I_0(v), \ldots)$ and $V_1 = (\ldots, I_1(v), \ldots)$ instead of V. (5) For test data generation an *under approximation* of the constraint solution set suffices: As soon as an interval vector $(I_{i_0}(x), I_{i_1}(y), I_{i_2}(globy))$ has been found such that all constraints evaluate to $[1,1]$ over these intervals, the generation process stops, and concrete test data is selected at random or at interval boundaries for each variable v from its associated interval. (6) For avoiding too many bi-partitioning steps (bi-partitioning obviously requires exponential time and storage) *forward-backward constraint propagation* can be applied as *contractor* for narrowing interval solution candidates with linear time effort; this is described in [BFPT06].

Example 3: Using the test case generator to support static analysis. As illustrated in the examples above, the test case generator uses under approximation of constraints in order to create test data. This capability is very useful for distinguishing false alarms resulting from over approximation from real UUT errors uncovered by the static analyser. To illustrate this, consider another variant of the for-loop in lines 10—12 in the example above, which also enforces loop termination:

```
1       int kOld = k;
2       for ( k += 1; k < x and error0 == 0; k *= 2) {
3           if ( kOld > k ) error0 = 1; else kOld = k;
4       }
```

During abstract interval interpretation, $I(kOld)$ is initialised with $[0, 10]$ in line 1, since this is the current interval valuation of k as shown above. As a consequence, $I(kOld)[>]I(k)$ evaluates to $[0,1]$, so that the fixpoint of the loop will evaluate $I(error0) = [0,1]$. As a consequence, the abstract interval interpretation indicates a false alarm with respect to reachability of the if-branch in line 3. Conventional static analysis tools might stop at this point. With the path selector, constraint generator and solver at hand, however, it is possible to prove that this if-branch is really not reachable for the precondition P_1 defined for $f()$ above. To this end, the path selector can suggest n-fold unwindings of the for-loop ($n = 0, 1, 2, \ldots$), and for each unwinding the constraint solver proves that no solution of the if condition in line 3 exists if P_1 holds on entry. Moreover, path selector, constraint generator and solver cooperate to prove that given P_1, the maximal number of unwindings is bounded by 6. As a consequence, the if-branch of line 6 remains unreachable. Observe that this method requires far more computing power than the simple interval interpretation that was successful for the first variant of the for-loop. As a consequence, the quick checks by interval interpretation are always the first choice before the investigation of feasible paths and constraint solutions. Finally, path selector, constraint generator and solver can construct explicit test data proving the reachability of the if-branch of line 6 if the bound for x in P_1 is dropped: It is then shown (suppose 32-bit word length for type int) that with initial assignments $x \in [2^{31}, \infty]$, $i = 2^{30}$ the if-branch will be reached and the error

flag set. To achieve this, the tool uses typed intervals and associated operations taking overflows of concrete C/C++ datatypes into account.

3 Conclusion and Ongoing Work

We have described the building blocks and basic concepts for an integrated test automation and static analysis system suitable for model based testing (not described in this article) of C/C++ software, structural/specification-based module testing and abstract interpretation by interval analysis. The main development and research tasks currently performed for improving the tool and its underlying methods focus on (a) optimisation of the memory model, so that constraints for complex expressions over pointers, unions and cast operations can be handled, (b) the extension of static analysis functions by integration of additional abstractions and (c) the addition of more specialised solver components to speed up linear arithmetic problems, bit vectors and their operations and string handling with associated pattern matching constraints. For the latter task we follow the framework of *satisfiability modulo theory* which currently is of considerable interest in the SAT solving and theorem proving communities [RT06]. The techniques described have been integrated in the RT-Tester tool [Ver07] and are applied in an industrial context since 2007. The tool capabilities applied to "real-world" embedded software (C++ code from railway control systems and C code from avionics control systems) will be shown during the tool demonstration session.

References

[BFPT06] Bahareh Badban, Martin Fränzle, Jan Peleska, and Tino Teige. Test Automation for Hybrid Systems. In *Proceedings of the Third International Workshop on SOFTWARE QUALITY ASSURANCE (SOQUA 2006)*, Portland Oregon, USA, November 2006.

[CCF+06] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of Abstractions in the ASTRÉE Static Analyzer. In M. Okada and I. Satoh, editors, *Eleventh Annual Asian Computing Science Conference (ASIAN'06)*, pages 1–24, Tokyo, Japan, LNCS, December 6–8 2006. Springer, Berlin. (to appear).

[JKDW01] Luc Jaulin, Michel Kieffer, Olivier Didrit, and Éric Walter. *Applied Interval Analysis*. Springer-Verlag, London, 2001.

[Lig02] Peter Liggesmeyer. *Software-Qualität*. Spektrum Akademischer Verlag, Heidelberg, Berlin, 2002.

[Löd] Helge Löding. Behandlung komplexer Datentypen in der automatischen Testdatengenerierung. Master's thesis, University of Bremen. To appear in May 2007.

[RHC76] C. V. Ramamoorthy, S.-B. F. Ho, and W. T. Chen. On the Automated Generation of Program Test Data. *IEEE Transaction on Software Engineering*, SE-2(4):293–300, 1976.

[RT06] S. Ranise and C. Tinelli. Satisfiability Modulo Theories. *TRENDS and CONTROVERSIES–IEEE Magazine on Intelligent Systems*, 21(6):71–81, 2006.

[SC-92] SC-167. *Software Considerations in Airborne Systems and Equipment Certification*. RTCA, 1992.

[Ver07] Verified Systems International GmbH, Bremen. *RT-Tester 6.2 – User Manual*, 2007.