ATGen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution

Christophe Meudec
Computing, Physics & Mathematics Department
Institute of Technology, Carlow
Kilkenny Road
Carlow, Ireland
+353 (0)503 70455
meudecc@itcarlow.ie

ABSTRACT
The verification and validation of software through dynamic testing is an area of software engineering where progress towards automation has been slow. In particular the automatic design and generation of test data remains, by in large, a manual activity. This is despite the high promises that the symbolic execution technique engendered when it was first proposed as a method for automatic test data generation.

In this work, we propose, and implement, a new approach based on constraint logic programming for the automatic generation of test data using symbolic execution.

After reviewing the symbolic execution technique, we present our approach for the resolution of the technical difficulties that have so far prevented symbolic execution from reaching its full potential. We then describe ATGen, our automatic test data generator, which is based on symbolic execution and uses constraint logic programming.

Keywords
Software Testing, Automatic Test Data Generation, Symbolic Execution, Constraint Logic Programming

1 INTRODUCTION

Developing software that is correct and behaves as expected is difficult. It is, at best, time consuming and expensive. To address these problems we propose a general approach using constraint logic programming and a tool for automating the design and generation of test data for software verification and validation.

Software verification involves checking that the software respects its specification. Software verification techniques include software inspections, formal proving of program correctness, static analysis of programs and testing.

Software validation involves checking that the software as implemented meets the expectations of the customer. It includes software reviews and acceptance testing where the software is exercised using tests provided by the customer. The customer may also want the software to be tested for particular circumstances for which tests have yet to be devised.

While we focus on testing for software verification and validation, we recognize that other techniques may be complementary in this area (in particular software inspections).

The testing phase can be supported by automatic tools. Three main categories of automation can be distinguished [34, 2]:

- Automation of administrative tasks, e.g. recording of test specifications and outcomes (useful for regression testing), test reports generation;
- Automation of mechanical tasks, e.g. the running and monitoring (for testing coverage analysis purposes) of the software under test within a given environment, capture/replay facilities allowing the automation of test suites execution;
- Automation of test generation tasks, i.e. the selection and the actual generation of test inputs;

While the first two areas are being well served by commercial tools—to a point that the expression '*automatic testing*' is often used as a synonym for automation of the tests execution only—the actual generation of test inputs is mostly still performed manually (with the exception of random testing).

It is in fact still the case that the automatic selection and generation of test inputs remains a challenge for tool developers [34].

This manual generation of test inputs implies that rigorous testing is laborious, time consuming, and costly. It also implies that rigorous testing is not actually widely applied.

The symbolic execution* technique, as proposed by King [27] more than 20 years ago, has the potential to help with the automation of the selection and generation of test inputs for a variety of problems. However, this potential has so far never been fully realized due to many technical problems [12].

* symbolic execution is also called symbolic evaluation

For completeness we acknowledge that new test data generation techniques with as wide a range of applications as symbolic execution have been investigated, e.g. [18, 39]. Other techniques, with a smaller focus, have also been proposed e.g. [30, 14].

It is our contention that our work places the automatic generation of test inputs for a variety of applications, as provided through symbolic execution, firmly in our grasp as demonstrated by our tool, ATGen.

After presenting the symbolic execution technique and its many potential applications, we review the traditional technical problems attached to it. We then give an overview of previous work in this area.

Our general approach for the resolution of the technical difficulties associated with symbolic execution is presented next. This general approach has been applied to a non-trivial test generation problem resulting in ATGen, our tool, which is presented and discussed before concluding.

2 SYMBOLIC EXECUTION

The symbolic execution of computer programs is an automatic static analysis technique that allows the derivation of symbolic expressions encapsulating the entire semantics of programs. It was first introduced by King [27] to help with the automatic generation of test data for dynamic software verification. As we shall see, verification is not the only important area where symbolic execution can be used.

The Symbolic Execution Technique

Symbolic execution extracts information from the source code of programs by abstracting inputs and sub-program parameters as symbols rather than by using actual values as during actual program execution. For example, consider the following Ada procedure that implements the exchange of two integer variables:

```
procedure Swap(X, Y : in out integer)
is
  T : integer;
begin
  T := X;
  X := Y;
  Y := T;
end Swap;
```

After actual execution of, say, `Swap(5, 10)`, `X` will be equal to `10` and `Y` will be equal to `5`, i.e. the values of X and `Y` have been swapped. Actual execution provides a snapshot of the semantics of the source code.

Using symbolic execution captures exactly and entirely the semantics of the source code. This is performed by associating the assigned variables with a symbolic expression made up of input variables only. Here, we denote symbolic expressions by delimiting them using single quotation marks. In our example therefore, `T` is first assigned to `'X'`, `X` is then assigned to `'Y'` and finally, `Y` is assigned to the symbolic expression `'X'`.

Most programs are not simple sequential composition of assignments. In particular, the presence of a conditional statement, such as an `if`…`then`…`else`…, splits the execution of programs into different paths. In general therefore, symbolic execution records for each potential execution path, a traversal condition. This path traversal condition is the logical conjunction of the Boolean conditions encountered by the path. This condition must be satisfiable for the path to be feasible. Infeasible paths (i.e. paths which cannot be traversed because no input data exists which satisfies its path traversal condition) are not uncommon and cannot be ignored.

Consider the example below where `Max` is a global integer variable.

```
procedure Order(X, Y : in out integer)
is
begin
  if X > Y then
    Max := X;
  else
    Swap(X, Y);
    Max := X;
  end if;
end Order;
```

Symbolically executing the procedure `Order` we obtain two paths:

1. Path Traversal Condition: `'X > Y'`
   Path Actions: `Max = 'X'`
2. Path Traversal Condition: `'Not(X > Y)'`
   Path Actions: `Max = 'Y'`
   `X = 'Y'`
   `Y = 'X'`

A more advanced example is provided later. We do not review here, for lack of space, techniques for the actual implementation of symbolic execution. Rather the reader is referred to a comprehensive survey of implementation techniques [6]. As we shall see, the difficulties do not so much lie with the implementation of the symbolic execution technique per se but more with the exploitation of its potential.

Exploitation of Symbolic Execution

The verification and validation of software are the main areas of applications for symbolic execution.

For completeness, we also mention that symbolic execution can help with the following:

- software debugging, re-engineering and comprehension [6] (e.g. by providing condensed information about program paths);

- software optimization, simplification and specialization [8, 28, 6] (e.g. by helping to identify loop invariants which can be moved out of iterative constructs, or by identifying unnecessary automatically inserted exception handling code [33]);
- applications to formal specifications can also be found [32, 29, 1];

Software Verification
We can distinguish four areas of interest:

- Automatic Test Data Generation for Coverage Testing

This is the first powerful usage of symbolic execution historically identified [27]. It can be extended to include data flow testing [5, 16].

Testing coverage criteria such as statement or decision coverage [5] have as their objective the execution of all statements or all decision outcomes, respectively, of the program under test. A symbolic executor can generate the path traversal condition of paths selected to achieve complete coverage. The path traversal conditions can then be sampled to obtain a set of test inputs which, by construction, achieves 100% (excluding unreachable code of course) coverage for the chosen testing criterion.

This application of symbolic execution requires the implementation of a path selection strategy, the ability to detect infeasible paths and the ability to sample satisfiable path traversal conditions to generate test inputs.

This capability would save a lot of manual effort as well as, typically, increase the level of overall coverage achieved.

- Automatic Test Data Generation for Path Domain Testing

Using coverage testing, a particular execution path is only tested once using a single test. It is often necessary however, to generate several tests for a single path in order to detect coincidental correctness [2, 12] or exercise the path using *'extreme'* values (as in boundary analysis [2]).

This can be achieved through analysis of the path actions (e.g. to detect the use of the remainder operator *'rem'* and generate a constraint to distinguish its usage from the modulo operator *'mod'*) or of the definition domain of variables and by adding constraints to the path traversal condition to force the generation of particular values.

This application requires the additional ability to generate constraints depending on the context of execution.

This extra testing has the potential to greatly increase the likelihood that errors will be detected in the program under test.

- Automatic Test Data Generation for Run-Time Errors Testing

Run-time errors occur when something unexpected occurs during the execution of a program (e.g. division by zero, access outside array bounds, variable overflow). They have the potential to crash the operating system.

There are two ways of dealing appropriately with run-time errors:

- Proving that the program is run-time error free;
- Inserting exception handling code to handle run-time errors;

The first approach is sometimes applied to safety critical software where it is acknowledged that preventing run-time errors is better than controlling their effects [1]. This approach falls within the remit of software proving.

Testing software that uses exception handling requires the generation of test inputs which will trigger the run-time error concerned. This can be achieved through specific path traversal conditions generation (e.g. to ensure that the denominator in a division takes zero for value) and sampling for test inputs generation.

To achieve this, the functional requirements for a testing tool are similar to the path domain testing application discussed previously.

Generating test inputs to trigger run-time errors is, of course, also necessary for programs that do not deal with run-time errors appropriately.

- Helping with Software Proving

Software proving is concerned with formal software verification. Symbolic execution is usually used to generate proof requirements involving a formal specification of the program under consideration [22]. Use of assertions for proving interesting properties of the software under consideration is also possible [1]. The proof requirements are then proved, or refuted, independently using a theorem prover. Theorem provers typically require human interventions. The same approach can be used to prove the absence of run-time errors [1].

This is the traditional role of symbolic execution during program proving.

Another angle is to attempt the generation of a test input negating the proof requirement [39]: if a test can be generated the proof need not be undertaken as it is bound to fail. Detecting instantly, at a low cost, that a proof will fail is attractive: commonly, many of the proof requirements attempted are unprovable and time-consuming to deal with.

This is applicable to proving that a program is run-time error free, as the first step should be to try to generate automatically a test triggering a run-time error.

Software Validation
Most of what we have discussed so far, under the software

verification heading, is applicable to software validation except that the tests generation requests would originate from the customer. The specific requirements of software validation however are often overlooked.

For example, it would be attractive to generate tests on a per scenario basis as proposed by the customer. Being able to answer reliably and quickly questions such as *'what happens if such and such variables have such and such values and this loop is taken 14 times?'* would be attractive. The customer may also wish to execute the software under consideration for everyday circumstances (e.g. avoiding extreme values) or in special operational modes (e.g. landing mode in a fly-by-wire software).

While the ability to generate and sample path traversal conditions would be required as before, a new requirement of our testing tool would also be necessary in our view. The obvious way of dealing with such test generation requests using symbolic execution would be through the judicious placing of assertions in the program source code. However, this is cumbersome for large programs. In our view, a higher level of usability, through the development of dedicated Graphical User Interfaces, is necessary to unlock the potential of test data generators for software validation purposes.

Traditional Difficulties with Symbolic Execution

Here we review the problems associated with symbolic execution in general. We can distinguish two distinct types of difficulties:

- Technical difficulties with the symbolic execution technique per se;
- Practical difficulties with the exploitation of the symbolic execution results;

Technical Problems

We can list in this category some features of programming languages that are challenging to deal with.

For example, array references can be problematic where the index is a not a constant but a variable—as is typically the case—as the particular array element referred to is then unknown. Symbolic execution can be performed in these cases with the generation of ambiguous array references in path traversal conditions [12]. The problem then is to decide the satisfiability of the conditions generated.

Loops are also difficult to deal with appropriately. Bounded loops can of course be unfolded as they do not create any new path in the program. Loops which are input variable dependent however, can be executed any number of times. Hence, there is the dilemma of the number of times the body of the loop should be traversed. Typically, symbolic executors generate path traversal conditions with loops executing zero, once or several times. This problem however should be dealt with according to the testing criteria under consideration and the feasibility or not of the current path.

Procedure and function calls can be handled by in-lining the sub-program code each time it is encountered or symbolically executing it once and using the results at each invocation [12].

Other characteristics of structured programming languages, which are difficult to deal with using symbolic execution, are dynamic memory allocation, pointers (especially pointer arithmetic as is allowed in the C programming language) and recursion.

Many of the technical problems faced by symbolic executors have been discussed by Coward in [12] and by Clarke and Richardson in [6].

It is our view that, although the generation of symbolic expressions along a given path in a program is not without technical difficulties, most of the restrictions usually imposed by symbolic executors on the source language that can be handled originate from the limitations of the techniques used for path feasibility analysis and test data generation [26] (i.e. practical problems associated with the exploitation of the results of the symbolic execution phase).

Practical Problems

Most symbolic executors simply generate all the syntactic paths in a program [1] (with special considerations for loops). It has been remarked by Coward in his review of symbolic execution systems [11], that this way of proceeding, besides wasting a lot of effort (because it is a purely syntactic process where feasibility of the intermediate paths is not checked during generation), may not be practical since a program may contain more paths that can reasonably be handled. Better, would be to integrate a path selection strategy within the symbolic executor to generate as few conditions as is necessary to achieve a particular testing criterion.

Further, and as we have seen, to exploit fully the potential of the symbolic execution technique it is necessary to be able to check the feasibility of the path traversal conditions generated and, for testing purposes at least, to be able to generate actual test data for feasible paths. Unfortunately, and as highlighted in the next section, the complexity of the path traversal conditions generated have, to date, proved too high to be tackled efficiently and automatically.

In our view, it is that fundamental problem that has hindered the wider use, and further development, of verification and validation tools based on symbolic execution rather than the perceived technical problems traditionally associated with the symbolic execution technique per se.

Related Work

Early research tackled the path feasibility problem using linear programming routines and rule-based checks [12, 11, 36, 7].

The problem with this approach is the inflexibility of the resulting tools. It may work well for conjunctions of linear conditions over integers, but separate techniques need to be used for, say, non-linear conditions over floating point numbers.

Furthermore, path traversal conditions typically are logical expressions over a mix of Boolean, integer, floating point number and enumeration variables organized in arrays and records: these cannot be solved using a single resolution strategy. At best, a lot of preprocessing needs to be performed before submitting subsets of a condition to a particular constraint resolution technique.

Syntactic simplification rules, while of value towards the representation of traversal conditions in a simplified form [1], are unlikely to detect many infeasible paths as such.

Using a theorem prover, as illustrated in [26, 19], may allow the handling of arrays where the index is not a constant. However, while using axiomatic rules for proving that a particular set of symbolic expressions over arrays is unsatisfiable may be suitable, it cannot be applied to linear expressions over, say, integers. Also, on their own, theorem proving tools are not suited for generating test data satisfying a particular path traversal condition.

So, while many separate techniques have been employed in previous attempts at determining path feasibility, the sheer complexity of most path traversal conditions has meant that, in practice, the underlying language on which symbolic execution is applied must be simplified and that the complexity of the path traversal conditions must be low for the approach to succeed (e.g. linear expressions over either integer or floating point variables but not mixed conditions where floating point and integer variables are used).

Therefore, the source language typically handled by testing tools based on symbolic execution is a small subset of its original [12] and no test data generation facility is provided: the tool only performs path feasibility analysis on all the syntactic paths [19].

3 OUR APPROACH

Our underlying approach centers on the tighter integration of the different sub-systems making up a test data generator based on symbolic execution, by using a constraint logic programming language. We have two overiding concerns: reducing the amount of wasted effort during generation of the symbolic expressions and enlarging the typical programming language subset that can be efficiently tackled by symbolic execution. Coincidentally, we are, in effect, taking further the general ideas presented by Hamlet in [21] for the rapid implementation of general testing tools.

Closer Integration

To avoid the generation of many paths with unsatisfiable path traversal conditions, and of paths which are not required for the fulfillment of the testing criteria under consideration, it is necessary to integrate the following, traditionally separate, elements of a test data generator:

- Symbolic executor;
- Path selector;
- Path feasibility analyzer;

Doing so would make it possible to check, during their symbolic execution, the feasibility of required paths only. Thus, we would avoid the heavy overhead engendered by the generation of unnecessary or infeasible paths.

While this approach is not a new proposal [12], its successful implementation has, to date, been elusive.

Additionally, we must also provide the means for the automatic sampling of satisfiable path traversal conditions so as to generate actual test data for the, known feasible, selected paths.

Constraint logic programming is the paradigm that has allowed us to realize these aspirations.

Use of Constraint Logic Programming

As we have seen, we want to check the satisfiability of algebraic expressions. I.e. given an algebraic expression, along with the variables involved and their respective domains, we must show that there exists an instantiation of the variables which reduces the expression to true. In effect, an algebraic expression constrains its variables to a particular set of values from their respective domains. If any of the sets are empty, the assertion is reduced to false and is said to be unsatisfiable. Thus, an algebraic expression is a system of constraints over its variables.

Hence, we have a Constraint Satisfaction Problem (CSP): we want to search the variable domains for solutions to a fixed finite set of constraints.

Constraint Satisfaction Problems (CSPs)

CSPs (see [17] for an informal introduction) are in general NP complete and a simple '*generate and test*' strategy, where a solution candidate is first generated then tested against the system of constraints for consistency, is not feasible. Constraint satisfaction problems have long been researched in artificial intelligence and many heuristics for efficient search techniques have been found. For example, linear rational constraints can be solved using the well-known simplex method [13].

To implement the kind of solver we require, e.g. able to work with non-linear constraints over floating point numbers and integers, we could implement these heuristics by writing a specialized program in a procedural language (such as C, or using an existing solving routines library). Nevertheless, although the heuristics are readily available, this approach would still require a substantial amount of

effort and the resulting program would be hard to maintain, modify and extend. Ideally, we would like to concentrate on the '*what*' rather than the '*how*', i.e. we are more interested in the problem of combining the heuristics rather than in implementing the internal mechanism of each individual heuristic search technique.

The advantages of logic programming, mainly under the form of the Prolog programming language [4], over procedural programming have long been recognized [21]: the '*what*' and the '*how*' are more easily separated since Prolog is based on first order predicate logic and has an in-built resolution computation mechanism. However, Prolog's relatively poor efficiency when compared to procedural languages has hindered its general acceptance.

For CSPs, however, Prolog is still the language of choice. Searches are facilitated by its in-built depth-first search procedure and its backtracking facilities. However, even in this area Prolog suffers from a general lack of facilities to express complex relationships between objects (terms): the semantics of objects has to be explicitly coded into a term. This is the cause of the perceived poor mathematical handling capabilities of Prolog when compared with its other facilities: only instantiated mathematics can be dealt with readily. Further, the basic in-built depth-first strategy tends to lead to a '*generate and test*' approach to most problems: specialized heuristics must be implemented to prune the search space.

Constraint Logic Programming

Constraint Logic Programming (CLP), as introduced by Jaffar and Lassez [25], reviewed by Colmerauer [10] and discussed in [9], alleviates these shortfalls by providing richer data structures on which constraints can be expressed and by using constraint resolution mechanisms (also known as decision procedures) to reduce the search space. When the decision procedure is incomplete—e.g. for non-linear arithmetic constraints—the problematic constraints are suspended, we also say delayed, until they become linear. Non-linear arithmetic constraints can become linear whenever a variable becomes instantiated (or bound). This can happen when other constraints are added to the system of constraints already considered or during labeling.

The labeling mechanisms further constrain the system of constraints according to some strategy. It can be viewed as a process to make assumptions about the system of constraints under consideration. It is a very powerful mechanism and it is used to awaken delayed constraints or to generate a solution to an already known satisfiable system of constraints.

To deal with non-linear problems, the labeling strategies used are critical to the overall efficiency of the solver. A discussion of constraint satisfaction using CLP can be found in [24].

We now give an example of constraint resolution involving integers.

In Prolog the equality:

```
X*X + Y = 10
```

results in failure, since in Prolog equality only holds between syntactically identical terms and `X` is just a variable of no particular type. Using a CLP language however, it is possible to code the semantics of `X` and `Y` as being integer-like and constrain them such that `X*X + Y = 10` holds. The constraint resolution mechanism will detect the constraint as non-linear and reduce it as follows:

```
X = X, Y = Y
X*X + Y = 10 is delayed
```

I.e. the system of constraints is satisfiable (subject to consideration of the delayed constraints) and a simplified version is internally held. A labeling strategy must impose further constraints on either `X` or `Y` for the satisfiability of the system of constraints to be confirmed.

During labeling, a not so efficient strategy would select `Y` to be sampled first. The sampling strategy would then instantiate `Y` with, say, `2` thus awaking and simplifying the delayed constraint to `X*X = 8`. This constraint would have to be delayed. The labeling strategy now attempts to instantiate `X` repeatedly without success (because failure in this case actually occurs on the entire definition domain of `X`) which induces backtracking in the traditional, logic programming, manner. Eventually `Y` is instantiated to another value, say `1`, thus reducing the constraint store to:

```
X = X, Y = 1
X*X = 9 is delayed
```

The labeling mechanism now attempts to awake the delayed constraint: this can only be achieved through instantiating `X`. Eventually `X` will be instantiated with `-3` or `3` and the system of constraints will be declared satisfiable and the sample, say, `X = 3, Y = 1` will also be available. To succeed, this labeling strategy has generated thousands (if the initial domain of the variables is of that order) of futile assumptions.

A more efficient labeling strategy would recognize that `X` is the variable on which the linearity of the delayed constraint depends and attempt to constrain it first. Any value in the domain of `X` will make the delayed constraint linear thus allowing the constraint resolution mechanism of the underlying solver to detect satisfiability directly. E.g. `X = -5` will awaken the delayed constraint and the solver would directly yield:

```
X = -5, Y = -15
```

This latest labeling strategy is adequate as only one assumption is made to yield a positive outcome.

It is this general approach that we have customized to be applicable to path traversal conditions as generated during symbolic execution.

CLP languages are ideal for our purpose as their in-built resolution mechanism removes most of the needed development effort and still offer the flexibility of logic programming. In fact, they allow the rapid development of efficient, dedicated, constraints solvers.

4 ATGen: AN AUTOMATIC TEST DATA GENERATOR

ATGen is our prototype testing tool implemented using the underlying approach outlined in the previous section.

The particular constraint logic programming environment used to implement ATGen is ECLiPSe [15]. ECLiPSe is a Prolog based system that serves as a platform for integrating various logic programming extensions, in particular CLP. ECLiPSe is distributed with many valuable libraries implementing various constraint solvers (over integers, rational numbers, sets etc.). Other similar environments may well be as equally suited.

Current Area of Application

While it should be clear that our approach is general enough to be applied to many structured programming languages for a variety of purposes we have chosen the initial area of application to be as compelling as possible.

Hence, the current area of application of ATGen is the automatic generation of test data to achieve 100% decision coverage of SPARK Ada programs.

Decision Testing

In decision testing [5, 2] the aim is to test all decision outcomes in the program. Typical decisions are Boolean expressions controlling the flow of execution in the program such as in conditional constructs and loops. Discounting infeasible decision outcomes [5] we aim to generate a test data suite achieving 100% decision coverage.

Note that the current implementation of our path selection strategy is not aimed towards producing the smallest test set possible but towards the fastest generation of a set of tests achieving coverage. Thus, some redundant paths may well be generated.

SPARK Ada

SPARK Ada [1] is a subset of the Ada programming language designed in particular for the development of high integrity software. It is the most popular Ada subset for safety critical software.

Briefly, the following Ada features are excluded from SPARK Ada: concurrency, dynamic memory allocations, pointers, recursion, and interrupts. The reader is referred to Barnes [1] for a complete definition of SPARK Ada.

Formally SPARK Ada is not just a subset of Ada, as it also requires additional annotations to give extra information about the program. This extra information can then be handled by SPARK analysis Tools (such as the SPARK Examiner [1]) to perform various static program analysis tasks (such as data flow analysis [1]).

We however discard any SPARK annotations and only consider the Ada constructs for our test data generation purposes (ATGen however could easily be adapted to handle FDL—Functional Description Language—constructs making up the outputs of the SPARK analysis tools [1]).

ATGen handles the entire SPARK Ada subset including Boolean, integer, floating point (represented using infinite precision rational numbers), enumeration types, records, multi-dimensional arrays, all loop constructs, functions and procedures calls. Further, there are no technical reasons why we may not extend the Ada subset currently handled by ATGen beyond SPARK.

Overall Structure

ATGen is composed of a pre-parser written in C and of roughly 4200 lines of commented Prolog code divided in seven modules.

The pre-parser transforms* the SPARK code into a list of Prolog facts. This transformation is purely syntactical (e.g. the first letter of variables is put into upper case, labels for conditions are automatically generated). For interest, we give below the parsed version of the second loop of `quotient`, our next example.

```
while(cond(C2, T <> D), stmts([
  assign(Q, Q * 2),
  assign(T, T / 2),
  if(cond(C3, T <= R), then(stmts([
    assign(R, R - T),
    assign(Q, Q + 1)
  ])),
  elsifs([]), else(stmts([])))
]))
```

This intermediate representation of the SPARK source code is compiled into ATGen as an ordinary Prolog program.

The symbolic execution of the program under consideration is directed according to the testing coverage criteria chosen and the feasibility analysis of the current subpath: infeasible or redundant subpaths are immediately abandoned and the system backtracks in an ordinary Prolog manner. The path traversal condition of suitable paths is sampled to generate test data. This entire process is repeated, through backtracking, until the testing coverage

* this transformation is still partially performed manually, but a parser is nearing completion using a YACC-like parser generator.

criteria is fulfilled.

Below we give a rough estimate of where the development effort was spent.

- 30% solving activities. Mostly concerned with customization and extension of the solving capabilities provided with ECLiPSe;
- 30% symbolic execution per se. Dealing with sub-program calls, iterative and conditional constructs, assignments;
- 20% source language features manipulation. Mainly concerned with the data structures of the source language (arrays, record, enumeration types);
- 10% labeling. Implementing overall and data type specific labeling strategies;
- 5% path selection strategy implementation;
- 5% test report generation. In particular printing of arrays, output domains;

The reader can infer from the above what effort would be involved in extending ATGen (for another language, a new coverage measure, improved labeling strategies etc.)

Characteristics and Limitations
We list below some interesting aspects of ATGen.

- The tests generated for coverage testing are designed not to generate run-time errors (including avoiding internal overflow of expressions);
- Using ATGen, actual test execution becomes unnecessary since the actual test output is provided along side the test inputs. However, it may still be necessary, to comply with the independent verification requirement of safety critical systems for example, to actually execute the test generated;
- Annotation of the SPARK source code is not needed;
- ATGen can be used for integration testing purposes [23] which is maybe the area where the manual design of test data is the most difficult;
- ATGen itself need not be of high integrity: the actual level of code coverage achieved can be checked using a third party tool;
- Sometimes path feasibility will be too time consuming to infer (mainly when the path traversal condition involves complex non-linear relations between floating point variables). Better heuristics for labeling and advances in CLP languages in general should reduce this problem in the future. Currently ATGen in such situations issues a warning message indicating which path has been considered infeasible by default;

Example
We reproduce, verbatim, an example given in [18] to demonstrate the problems associated with symbolic execution:

```
procedure quotient(n: Some_Integer,
                   d: Some_Integer) is
-- calculate quotient and
-- remainder of the integer
-- division of n by d, (n>0, d>0)
  q: Some_Integer := 0;
  r: Some_Integer := n;
  t: Some_Integer := d;
begin
  while r >= t loop           --C1
    t := t * 2;
  end loop;
  while t /= d loop           --C2
    q := q * 2;
    t := t / 2;
    if t <= r then            --C3
      r := r - t;
      q := q + 1;
    end if;
  end loop;
-- manipulate r and q;
end quotient;
```

The difficulties with `quotient` are that both loops are input variable dependent and that the second loop must be executed exactly the same number of times as the first loop was for the path under consideration to be actually feasible. Further, the path traversal conditions generated involve non-linear arithmetic.

A typical ATGen output for the procedure `quotient` using the decision coverage testing criteria is given below.

Path: `C1 false, C2 false`
Test Data: `D = 10084, N = -20016`
Test Result: `T = 10084, R = -20016, Q = 0`

Path: `C1 true, C1 true, C1 false, C2 true, C3 true, C2 true, C3 false, C2, false`
Test Data: `D = 5836, N = 12905`
Test Result: `T = 5836, R = 1233, Q = 2`

We make several remarks about this result:

- The above result is generated in under 1.5 seconds on average using a 450MHz Pentium III based machine;
- ATGen is non-deterministic: the actual paths and test data generated may differ on subsequent runs;
- More information, than we have space here for, per path is available (such as the actual path traversal condition);
- The second path generated actually makes the first one

redundant for decision coverage purposes;

- The path traversal condition for the second path is: `N >= D and N >= D*2 and not(N >= D*2*2) and D*2*2 /= D and D*2*2/2 <= N and D*2*2/2 <> D and not(D*2*2/2/2 <= N - D*2*2/2) and not(D*2*2/2/2 <> D)`

5 FUTURE WORK

Below are our plans for future work on ATGen.

Demonstrating Practicality*

We must better demonstrate the practicality of ATGen for real world testing applications. Therefore, while we need to evaluate ATGen using automatically generated code (as in [18]), our main motivation should be to seek actively real world software testing problems in the best engineering research tradition [35].

Increasing Usability

If the potential of symbolic execution is to be realized, ATGen must provide better ways for the vast amount of information that can be generated to be channeled efficiently to the human testers. Further, the execution of the test data generator should be easy to parameterize to facilitate software validation activities.

We believe that this requires the development of a Graphical User Interface (GUI) with facilities to visualize and manipulate the control flow organization of the software under consideration. We have started work on this aspect.

We remark that program analysis tools in general, need to allow greater interaction with the user to expand successfully in a software engineering environment [31].

Increasing Versatility

We would like to increase the number of test coverage criteria handled by ATGen (in particular MCDC [5], Modified Condition Decision Coverage, which is required for airborne systems [38]), and expand into data flow testing [5, 16].

Larger subsets of Ada than SPARK may also be considered as well as other languages (e.g. Java [20] or C).

In addition, the application of ATGen in the area of run-time errors testing deserves further investigations.

Increasing Performance

While we have been pleasantly surprised by the overall performance of ATGen in terms of execution time and capability at detecting immediately infeasible paths, we recognize that further improvements may well be necessary. In particular we need to investigate better labeling strategies including moved based heuristics such as Hill climbing, Simulated Annealing and Tabu search [15, 37].

6 SUMMARY

ATGen automatically generates test data for total decision coverage of SPARK Ada programs.

It implements our general approach for solving the traditional problems associated with the symbolic execution technique.

Our approach is centered on tighter integration of the various components making up a test data generator using constraint logic programming. This use of constraint logic programming is, to our knowledge, unique to our work.

We have presented our plans for future work and are confident that ATGen will be successfully applied on real world software testing problems in the near future.

ACKNOWLEDGEMENTS

Some preliminary results, concerning the work presented, were obtained while the author was employed by the University of York, UK.

REFERENCES

1. Barnes, J. *High Integrity Ada: The SPARK Approach*, Addison-Wesley, ISBN 0-201-17517-7, 1997.
2. Beizer, B. *Software Testing Techniques*, 2nd Edition, Van Nostrand Reinhold, ISBN 0-442-20672-0, 1990.
3. Bicevskis, J., Borzovs, J., Straujums U., Zarins A., and Miller, E.F. SMOTL—A System to Construct Samples for Data Processing Program Debugging, *IEEE Trans. Software Eng.* 15:60-66, 1979.
4. Bratko, I., *Prolog Programming for Artificial Intelligence*, 2nd Edition, Addison-Wesley, ISBN 0-201-41606-9, 1990.
5. British Computer Society Specialist Interest Group in Software Testing (BCS SIGIST), *Standard for Software Component Testing*, Working Draft 3.3, 1997.
6. Clarke, L.A., and Richardson, D.J. Applications of Symbolic Evaluation, *J. Systems Software*, 5:15-35, 1985.
7. Clarke, L.A., Richardson, D.J., and Zeil, S.J. Team: A Support Environment for Testing, Evaluation and Analysis, In *Proceedings Software Engineering Symposium of Practical Software Development*, pp. 153-162, Nov. 1988.
8. Coen-Porisini, A., De Paoli, F., Ghezzi, C., and Mandrioli, D. Software Specialization Via Symbolic Execution, *IEEE Trans. Software Eng.*, 17(9):884-899, Sep. 1991.
9. Cohen, J. Constraint Logic Programming Languages, *Commun. ACM*, 33(7):52-68, Jul. 1990.

* It is hoped that further results will be available in time for the workshop as well as a tool prototype.

10. Colmerauer, A. An Introduction to Prolog III, *Commun. ACM*, 33(7):69-90, Jul. 1990.

11. Coward, P.D. Symbolic Execution Systems—A Review, *Software Engineering Journal*, 3(6):229-239, Nov. 1988.

12. Coward, P.D. Symbolic Execution and Testing, *Information and Software Technology*, 33(1):53-64, Jan./Feb. 1991.

13. Dantzig, G.B. *Linear Programming and Extensions*, Princeton University Press, Princeton, New Jersey, 1963.

14. Demillo, R., and Offutt, A. Constraint-Based Automatic Test Data Generation. *IEEE Trans. Software Eng.*, 17(9):900-910, 1991.

15. ECLiPSe Release 4.2, Imperial College London, 1999, *http://www.icparc.ic.ac.uk/eclipse/*

16. Frankl, P.G., and Weyuker, E.J. An Applicable Family of Data Flow Testing Criteria, *IEEE Trans. Software Eng.*, 14(10):1483-1498, Oct. 1988.

17. Freuder, E. The Many Paths to Satisfaction, In *Proceedings ECAI'94 Workshop on Constraint Processing*, Amsterdam, Aug. 1994.

18. Gallagher, M.J., and Narasimhan, V.L. ADTEST: A Test Data Generation Suite for Ada Software Systems, *IEEE Trans. Software Eng.*, 23(8):473-484, Aug. 1997.

19. Goldberg, A., Wang, T.C., and Zimmerman, D. Applications of Feasible Path Analysis to Program Testing, In *Proceedings ISSTA'94* (Seattle, WA, Aug. 1994)

20. Gosling, J., Joy, B., and Steele, G. *The Java Language Specification*, Technical Report, Sun Microsystems, Aug. 1996.

21. Hamlet, D. Implementing Prototype Testing Tools, *Software Practice and Experience*, 25(4):347-371, Apr. 1995.

22. Hantler, S.L., and King, J.C. An Introduction to Proving the Correctness of Programs, *ACM Computing Surveys*, 8(3), pp. 331-353, September 1976.

23. Harrold, M.J., and Soffa, M.L. Selecting and Using Data for Integration Testing, *IEEE Software*, 58-65, Mar. 1991.

24. Hentenryck, P.V. Constraint Satisfaction using Constraint Logic Programming, *Artificial Intelligence*, 58:113-159, 1992.

25. Jaffar, J., and Lassez, J-L. Constraint Logic Programming, In *Proceedings POPL'87*, pp. 111-119, Munich, Jan. 1987, ACM Press.

26. Jasper, R., Brennan, M., Williamson, K., Currier, B., and Zimmerman, D. Test Data Generation and Feasible Path Analysis, In *Proceedings ISSTA'94* (Seattle, WA, Aug. 1994) 95-107.

27. King, J.C. Symbolic Execution and Program Testing, *Commun. ACM*, 19(7):385-394, 1976.

28. King, J.C. Program Reduction Using Symbolic Execution, *SIGSOFT Software Engineering Notes*, 6(1):9-14, 1981.

29. Kneuper, R. Symbolic Execution: a Semantic Approach, *Science of Computer Programming*, 16(3), pp. 207-249, Oct. 1991.

30. Korel, B. Automated Test Data Generation for Programs with Procedures, In *Proceedings ISSTA'96*, pp. 209-215.

31. Le Métayer, D. Program Analysis for Software Engineering: New Applications, New Requirements, New Tools, *ACM SIGPLAN Notices*, 32(1):86-88, Jan. 1997.

32. Meudec, C. Tests Derivation from Model Based Formal Specifications, In *Proceedings 3rd Irish Workshop in Formal Methods*, Galway, Jul. 1999, http://www.ewic.org.uk/ewic/.

33. Muller, G., and Schultz, U.P. Harissa: A Hybrid Approach to Java Execution, *IEEE Software*, 44-51, Mar./Apr. 1999.

34. Ould, M.A. Testing—A Challenge to Method and Tool Developers, *Software Engineering Journal*, 6(2):59-91, Mar. 1991.

35. Parnas, D.L. On ICSE's "Most Influential Papers", *ACM Software Eng. Notes*, 20(3), 1995.

36. Ramamoorthy, C.V., Siu-Bun, F.H., and Chen, W.T. On the Automated Generation of Program Test Data, *IEEE Trans. Software Eng.*, 2(4):293-300, Dec. 1976.

37. Rayward-Smith, V.J., and al. Editors. *Modern Heuristic Search Methods*, Wiley, 1996.

38. RTCA, *Software Considerations in Airborne Systems and Equipment Certification Guideline*, Radio Technical Commission for Aeronautics, Number DO-178A, Mar. 1985.

39. Tracey, N., Clark, J., and Mander, K. Automated Program Flaw Finding using Simulated Annealing. In *Proceedings ISSTA'98*, pp. 73-81.