ATGEN: Automatic test data generation using constraint logic programming and symbolic execution

Article *in* Software Testing Verification and Reliability · June 2001

DOI: 10.1002/stvr.225 · Source: DBLP

1 author:

Christophe Meudec
Institute of Technology, Carlow
6 PUBLICATIONS **139** CITATIONS

SEE PROFILE

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.

This paper presents an automatic test data generator based on constraint logic programming and symbolic execution.

After reviewing the symbolic execution technique, approaches for the resolution of the technical difficulties that have so far prevented symbolic execution from reaching its full potential in the area of automatic test data generation are presented. ATGen, an automatic test data generator, based on symbolic execution and that uses constraint logic programming is then discussed.

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. The main technique currently used in the industry for the verification of software is dynamic software testing where the software is actually executed using test data. Complementary, static, approaches to software verification and validation are not discussed here.

Dynamic testing can be supported by automatic tools and three main categories of automation can be distinguished [1, 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 [1].

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 [3] (also called symbolic evaluation technique), as proposed by King [4] more than 20 years ago, has the potential to help with the automation of static and dynamic validation and verification. However, this potential has so far never been fully realized for test data generation purposes due to many technical problems [5].

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. [6, 7]. Other techniques, with a smaller focus, have also been proposed e.g. [8, 9].

After presenting the symbolic execution technique, the traditional technical problems attached to it are reviewed and then an overview of previous work in this area is given.

The general approach used in this work 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 for SPARK Ada, which is presented and discussed before concluding.

2. BACKGROUND

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 [4] to help with the automatic generation of test data for dynamic software verification.

2.1 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.

On this example, using symbolic execution captures exactly and entirely the semantics of the source code. It is performed by associating the assigned variables with a symbolic expression made up of input variables only. Here, the symbolic expressions are denoted by delimiting them using single quotation marks. In the example therefore, `T` is first assigned `'X'`, `X` is then assigned `'Y'` and finally, `Y` is assigned 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 symbolically executed Boolean conditions encountered by the path. The path traversal condition must be satisfiable for the path to be feasible. Infeasible paths are common and, at least for test data generation purposes, 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` two paths are obtained:

Path	Path Traversal Condition	Path Actions
1	'X > Y'	Max = 'X'
2	'Not(X > Y)'	Max = 'Y' and X = 'Y' and Y = 'X'

Many applications of symbolic execution have been identified [10]. In particular we mention, for completeness, that the symbolic execution technique has been successfully applied to program specialization [11] and to static verification problems (e.g. PREfix [12] for the detection of potential run-time errors in C++ programs). These areas are easier to deal with than the automatic generation of test data since unsoundness in the symbolic execution process and in the path feasibility checking can be accommodated. There is also no requirement to generate actual test data.

The focus in this paper is on test data generation. Satisfiable path traversal conditions can be sampled to provide test data that, by construction, will exercise during actual program execution the given path.

2.2 Traditional Difficulties with Symbolic Execution for Test Data Generation

The problems associated with symbolic execution are reviewed next. Two distinct types of difficulties can be distinguished:

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

2.2.1 Technical Problems

These are primarily due to 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 [5]. 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 paths 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 [5].

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 [5] and by Clarke and Richardson in [13].

During the course of the work described here, it has been found 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 [14] (i.e. practical problems associated with the exploitation of the results of the symbolic execution phase for test data generation purposes). To address these problems we have focused on using constraint logic programming as detailed in the next section.

2.2.2 Practical Problems

Most symbolic executors simply generate all the syntactic paths in a program [15] (with special considerations for loops). It has been remarked by Coward in his review of symbolic execution systems [16], that this way of proceeding, besides wasting a lot of effort (because it is a purely syntactic process where feasibility of the intermediate subpaths is not checked during generation), may not be practical since a program may contain more paths that can reasonably be handled. Better, is 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 to exploit fully the potential of the symbolic execution technique it is necessary to be able to check the feasibility of the partial 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.

2.3 Related Work on Automated Test Data Generation from Source Code

Early research tackled the path feasibility problem using linear programming routines and rule-based checks [5, 16, 17, 18].

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 [15], are unlikely to detect many infeasible paths as such.

Using a theorem prover, as illustrated in [14, 19], may allow in some circumstances the handling of arrays where the index is not a constant. However on their own, theorem provers are not suited to the handling of complex arithmetical expressions and cannot generate test data satisfying a particular path traversal condition.

Recent work by Offut et al. [20] is entirely based on constraint satisfaction using a dynamic domain reduction procedure. This technique while sufficient for constraints over integers is not suitable for constraints involving floating point numbers. Further, it does not generalise to other constraints that are encountered when dealing with programming language, primarily, array referencing with unknown index. As is described next, Constraint Logic Programming provides a flexible framework for the integration of many constraint resolution techniques, including domain reduction for finite domain variables.

To conclude, many separate techniques have been employed in previous attempts at determining path feasibility. The sheer complexity of most path traversal conditions however 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 [5] and no test data generation facility is provided [19].

3. ON THE USE OF CONSTRAINT LOGIC PROGRAMMING

The approach described here 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 as framework. Two overriding concerns underpin this approach: to avoid a combinatorial explosion of states and to enlarge the typical programming language subset that can be efficiently tackled.

3.1 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 makes it possible to check by following the control flow of the program under consideration using symbolic execution, the feasibility of the subpaths selected to fulfil a given testing criteria.

This approach avoids the heavy overhead engendered by the investigation of subpaths that are infeasible or lead to unnecessary tests.

Additionally, the means for the automatic sampling of satisfiable path traversal conditions must also be provided to generate actual test data.

While this approach is not a new proposal [5], its successful implementation had, to date, been elusive. Constraint Logic Programming, described next, has allowed such an implementation.

3.2 Constraint Logic Programming

As seen, it is necessary to check the satisfiability of large algebraic expressions. I.e. given an algebraic expression, along with the variables involved and their respective domains, it must be shown 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, the problem can be reduced to a Constraint Satisfaction Problem (CSP): it entails a search of the domains of variables for solutions to a fixed finite set of constraints.

3.2.1 Constraint Satisfaction Problems (CSPs)

CSPs (see [21] 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 [22].

To implement the kind of solver required here, e.g. able to work with non-linear constraints over floating point numbers and integers, it is possible to 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, it would be better to concentrate on the '*what*' rather than the '*how*', i.e. to be more focused on the problem of combining the heuristics to implement the kind of constraints solver required here 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 [23], over procedural programming have long been recognized [24]: 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: this already makes Prolog an ideal choice for implementing a symbolic executor that follows the control flow graph of the program under consideration according to a given testing criteria and backtracks whenever unsatisfiability of the current subpath is detected by a purpose built constraints solver.

However, Prolog suffers from a general lack of facilities for expressing mathematical relationships between objects: 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.

3.2.2 Constraint Logic Programming

Constraint Logic Programming (CLP), as introduced by Jaffar and Lassez [25], reviewed by Colmerauer [26] and discussed in [27], 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 of CSPs. When the decision procedure is incomplete—e.g. for non-linear arithmetic constraints—the problematic constraints are suspended, it is also said 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. This delaying mechanism can be reused to implement resolution mechanisms for constraints that occur in path traversal conditions during symbolic execution: e.g. constraints over array elements whose index is unknown can be delayed until the index is further constrained.

The labeling mechanism further constrains the system of constraints according to some value choosing 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 (as would be required for test data generation).

A simple example of constraint resolution involving integers is given in [10].

CLP languages are ideal for implementing a customised constraints solver able to detect unsatisfiability in path traversal conditions generated during symbolic execution. They also allow the implementation through a labeling mechanism of sampling strategies for satisfiable path traversal condition to generate actual tests data. Their in-built resolution mechanisms (typically over integers and rationals) remove most of the needed development effort and can be further combined and extended using logic programming and a delaying mechanism.

In fact, CLP languages allow the rapid development of efficient, dedicated, constraints solvers.

4. ATGen: AN AUTOMATIC TEST DATA GENERATOR

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

ATGen is implemented using the ECLiPSe [28] constraint logic programming environment and consists over roughly 5000 lines of commented Prolog code. 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 resolution mechanisms.

The current area of application of ATGen is the automatic generation of test data to achieve 100% decision coverage for programs written in SPARK Ada. In decision testing [29, 2] the aim is to test all decision outcomes in the program. Decisions are Boolean expressions controlling the flow of execution in the program such as in conditional constructs and loops. Discounting infeasible decision outcomes [29] the aim is to generate a test data suite achieving 100% decision coverage.

4.1 Overview

The symbolic execution follows the control flow of the program under consideration according to the testing coverage criteria chosen and the feasibility analysis of the current subpath traversal condition: infeasible or redundant subpaths are immediately abandoned and the symbolic execution backtracks in an ordinary Prolog manner. The feasibility analysis is performed using a bespoke constraints solver implemented using Constraint Logic Programming. The path traversal condition of suitable paths is sampled to generate test data. This entire process is repeated, through backtracking, until the testing coverage criteria is fulfilled. It is to be noted that Ada floating point variables are modeled using infinite precision rationals.

4.2 Detailed Architecture Description

The overall architecture of ATGen is illustrated in Figure 1 where rectangles represent processes and bubbles represent inputs and outputs.

4.2.1 SPARK Ada Program

SPARK Ada [15] 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.

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 [15]) to perform various static program analysis tasks (such as data flow analysis [15]). ATGen however discards any SPARK annotations and only considers the Ada constructs for test data generation purposes.

ATGen handles most of the 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. Due to development time constraints ATGen does not handle programs scattered over multiple files and some Ada entities attributes. ATGen however extends some aspects of SPARK Ada to include for example functions with side effects.

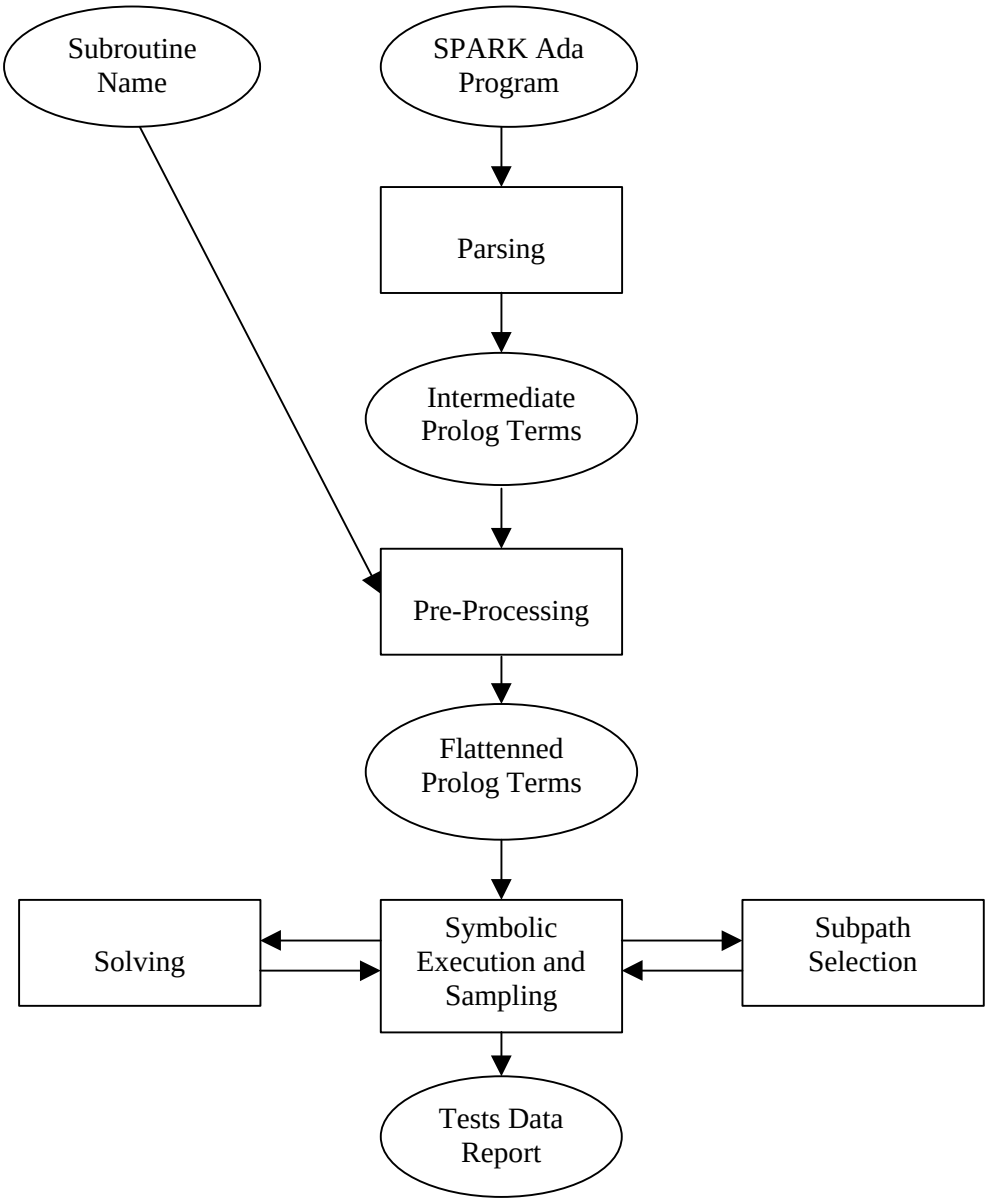

Figure 1. High Level Architecture of ATGen

4.2.2 Parsing

Parsing is handled by a program written in C that transforms the SPARK Ada code into a list of Prolog terms. This transformation is purely syntactical (e.g., the first letter of variables is put into upper case). The parser is generated using the Flex and Bison tools [30, 31]. The parser generates an intermediate Prolog terms file.

4.2.3 Intermediate Prolog Terms

As an example, the parsed version of the beginning of the body of `precedes`, our next example, is given below.

```
body(stmts([if_stmt([
if_clause(cond(1,field(D1,year)<field(D2,year)),stmts([assign(P,true)])),
if_clause(cond(2,field(D1,year)>field(D2,year)),stmts([assign(P,false)]))
, ...
```

4.2.4 Subroutine Name

This simply consists of the name of the subroutine for which tests must be generated. ATGen performs inter-procedural test data generation so that it will attempt to cover all the decisions of any subroutine called by the initial subroutine. In the final tests data report, the test input data will consist of all the parameters and global variables read by the subroutine. The test output data will consist of all the parameters and global variables assigned by the subroutine. The initial subroutine can be at any level of nesting within the input program.

4.2.5 Pre-Processing

In order to be able to perform inter-procedural test data generation, the pre-processor, written in Prolog, transforms the intermediate Prolog terms representing the program into a flat structure. That is, nested subroutines are extracted and listed on their own. To preserve the original semantics of the program, parameters are added reflecting the variables that are were visible at the definition point of the subroutine and calls to the subroutine are updated to include these parameters.

The name of the sub-routine to test is used to generate a virtual test harness which sets up the declarations needed and a query that calls the initial subroutine which will be used by the symbolic executor to start the analysis.

4.2.6 Flattened Prolog Terms

The flattened Prolog terms produced by the pre-processing phase are dynamically compiled as a Prolog program and are thus available to the symbolic execution phase through Prolog queries.

4.2.7 Subpath Selection

Currently this part is the least developed of ATGen. It only indicates to the symbolic executor if a condition has already been covered by a test. It is also used to produce the final coverage score in the tests data report. Due to this simplicity, ATGen can in the presence of infeasible subpaths attempt to satisfy them by following a different subpath which in the presence of unbounded loops has for consequence to induce an infinite futile search. A way to deal with this situation and, in general, more efficient guided search will have to be found during future work on ATGen.

4.2.8 Symbolic Execution

The first task of the symbolic execution phase is to initialise all the variables needed by the subroutine under consideration and to initiate an initial call using the virtual test harness

produced during pre-processing. The flattened subroutine is then retrieved by a simple Prolog query. Before tackling the body of the subroutine, the following activities take place: the parameters are matched and the local declarations are processed. The core of the symbolic execution process is of course the handling of statements.

During an assignment the symbolic value of the assigned variable is updated to the assigned expression expressed in terms of input variables only. Subroutine calls are unfolded in situ with parameter matching. Basic block statement proceeds without creating choice points in the Prolog execution.

The following procedure is followed whenever a condition is encountered in the code.

1. The subpath selection process indicates whether the current condition and its negation have already been covered.
2. If the condition (resp. the logical negation of the condition) has already been successfully covered (i.e. a test case that traverse it by construction has been generated) then the negation of the condition (resp. the condition) is chosen first. If both the condition and its negation have already been covered by a test case then we choose one at random. This approach focuses the symbolic executor on the areas of the control flow graph of the subroutine under consideration which have yet not been covered, without excluding the re-coverage of already covered expressions as they may lead to yet uncovered subsequent conditions. This approach allows us to deal with all loops in the code under consideration but may, of course, lead to very long searches.
3. The condition chosen is expressed in terms of input variables only (i.e. it is symbolically executed).
4. The condition is added to the current constraint store through the solver. If the solver fails to add the constraint then the current path traversal condition is unsatisfiable. This can be detected if contradictory linear constraints over integer or rational numbers exist. In case of failure the symbolic execution backtracks in an ordinary Prolog manner: it undoes all its previous actions up to the last decision point and then proceeds forward again. If successful then the current subpath may be traversable and the symbolic execution continues forward.
5. On reaching the end of the subroutine under consideration, all the input variables involved in the current path traversal condition are labeled using a variety of strategies depending on their domain. The labeling process attempts to give values to the variables while respecting the current constraint store (i.e. the values satisfy the path traversal condition). Boolean, integer and enumeration variables are labeled using an exhaustive search of their domain if necessary. At this level two modes of failure can be distinguished:
    - Failure because the path traversal condition is actually unsatisfiable (in which case we backtrack automatically up to the last decision point in the symbolic execution process), this can occur for example because of contradictory non-linear constraints over integers.
    - Failure because the labeling of rational variables was unsuccessful (this may or not be because the path traversal condition is actually unsatisfiable). Whenever this occurs it can be said that ATGen has failed to establish the feasibility or otherwise of a path traversal condition. This failure is indicated in the report file and backtracking occurs in a conservative manner. The search will try to reach the conditions not yet covered using a different path.

Much of the simplicity of the design of our symbolic execution implementation is due to the

backtracking facility of Prolog. Once no more decision point remains in the Prolog execution of the subroutine under consideration or if all decisions have been covered the symbolic execution terminates.

4.2.9 Solving

The solver, implemented using ECLiPSe, is composed of the following components:

- fd, a constraint solver over integers provided as a library by ECLiPSe (which uses domain propagation techniques) adapted to handle additional Ada operators (e.g. `mod`, `rem`, `power to`);
- clpq, a constraint solver over infinite precision rationals provided as a library by ECLiPSe (used in ATGen to model Ada floating point variables);
- a bespoke bridge between fd and clpq to handle mixed constraints;
- custom extensions to handle constraints over arrays, records and enumeration literals. These make extensive use of ECLiPSe delaying mechanism to handle constraints which cannot be resolved immediately (e.g. array access with unknown index);

For lack of space, it is not possible here to discuss in details the implementation of the solver. It is hoped that the solver will soon be available for release as a stand-alone library useful for general program analysis purposes.

4.2.10 Tests Data Report

The report file contains a list of entries each of which consisting of the test data generated along with the path covered, the expected result, the input and output variables domains along that path. An overall coverage score is also given.

5. INITIAL RESULTS

ATGen has been tested, by translating to SPARK Ada, programs from the literature on symbolic execution and test data generators, using the examples from the definition of SPARK Ada [15], adapting open source Ada programs [32] and also using made up examples. However, it has proved impossible at this stage to experiment with ATGen using industrial SPARK Ada code as such code is not usually made available even for research due to its safety critical aspects.

ATGen capabilities can however be illustrated on a made up example which goes beyond, in terms of complexity, other examples found in the literature on symbolic execution and tests data generators.

```
  type year_t is range 1900 .. 3000;
  type day_t is range 1..31;
  type month_t is (jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec);
  type date_t is
    record
      day : day_t;
      month : month_t;
     year : year_t;
    end record;
  type index is range 1..4;
  type list is array(index) of date_t;

function precedes(d1, d2 : date_t) return boolean
is
  p : boolean;
begin
  if d1.year < d2.year then p := T;                --decision 1
  elsif d1.year > d2.year then p := F;             --decision 2
```

```
  elsif d1.month < d2.month then p := T;          --decision 3
  elsif d1.month > d2.month then p := F;          --decision 4
  elsif d1.day < d2.day then p := T;              --decision 5
  elsif d1.day > d2.day then p := F;              --decision 6
  else p := F;
  end if;
  return p;
end precedes;

procedure InsertionSort(L: in out list) is
  firstunsorted, place : index;
  current : date_t;
  found : boolean;
begin
for firstunsorted in index range index'succ(index'first) .. index'last loop
-- above is decision 7
  if precedes(L(firstunsorted), L(index'pred(firstunsorted))) then --decision 8
    place := firstunsorted;
    current := L(firstunsorted);
    loop                                          --decision 9
      place := index'pred(place);
      L(index'succ(place)) := L(place);
      if place = index'first then                 --decision 10
        found := T;
      else
        found := precedes(L(index'pred(place)), current);
      end if;
      exit when found;                            --decision 11
    end loop;
    L(place) := current;
  end if;
end loop;
end InsertionSort;
```

On the insertion sort example above ATGen generates a set of test data that achieves 100% decision coverage in under 10 seconds on average on a 450 MHz Pentium III machine. Note that the subroutine `Precedes` is indirectly fully tested via `InsertionSort`. A typical set of test data for `InsertionSort` is given below:

```
Path 1: (7, T), (1, T), (8, T), (9, T), (10, T), (11, T), (7, T), (1, F), (2, F),
(3, T), (8, T), (9, T), (10, F), (1, T), (11, T), (7, T), (1, T), (8, T), (9, T),
(10, F), (1, F), (2, T), (11, F), (10, F), (1, F), (2, F), (3, F), (4, F), (5, T),
(11, T), (7, F)]
TEST INPUT DATA L=[(9, feb, 2857),(3, jan, 2854),(4, jan, 2857),(15, jan, 2854)]
TEST ORACLE DATA L=[(3, jan, 2854),(15, jan, 2854),(4, jan, 2857),(9, feb, 2857)]

PATH 2: (7, T), (1, T), (8, T), (9, T), (10, T), (11, T), (7, T), (1, F), (2, F),
(3, T), (8, T), (9, T), (10, F), (1, T), (11, T), (7, T), (1, T), (8, T), (9, T),
(10, F), (1, F), (2, T), (11, F), (10, F), (1, F), (2, F), (3, F), (4, F), (5, F),
(6, F), (11, F), (10, T), (11, T), (7, F)]
TEST INPUT DATA L=[(28, feb, 2968),(29, jan, 2733),(7, jan, 2968),(29, jan, 2733)]
TEST ORACLE DATA L=[(29, jan, 2733),(29, jan, 2733),(7, jan, 2968),(28, feb,
2968)]

PATH 3: (7, T), (1, T), (8, T), (9, T), (10, T), (11, T), (7, T), (1, F), (2, F),
(3, T), (8, T), (9, T), (10, F), (1, T), (11, T), (7, T), (1, T), (8, T), (9, T),
(10, F), (1, F), (2, T), (11, F), (10, F), (1, F), (2, F), (3, F), (4, F), (5, F),
```

```
(6, T), (11, F), (10, T), (11, T), (7, F)]
TEST INPUT DATA L=[(3, feb, 2037),(30, jan, 2006),(4, jan, 2037),(23, jan, 2006)]
TEST ORACLE DATA L=[(23, jan, 2006),(30, jan, 2006),(4, jan, 2037),(3, feb, 2037)]

PATH 4: (7, T), (1, T), (8, T), (9, T), (10, T), (11, T), (7, T), (1, F), (2, F),
(3, T), (8, T), (9, T), (10, F), (1, T), (11, T), (7, T), (1, T), (8, T), (9, T),
(10, F), (1, F), (2, T), (11, F), (10, F), (1, F), (2, F), (3, F), (4, T), (11,
F), (10, T), (11, T), (7, F)]
TEST INPUT DATA L=[(7, feb, 2470),(4, feb, 1976),(23, jan, 2470),(1, jan, 1976)]
TEST ORACLE DATA L=[(1, jan, 1976),(4, feb, 1976),(23, jan, 2470),(7, feb, 2470)]
```

To execute the test data, a test driver should be written to set up the necessary calling context for the subroutine under investigation and collect the actual results. The calling context, including global variable values if necessary, is entirely and automatically provided by ATGen. The automatic generation of a test harness from ATGen outputs is currently under development.

Two conclusions are drawn on ATGen's capabilities from the above example.

- The search for uncovered decisions in the control flow graph of the subroutine under consideration is random. There are therefore wide performance variations between successive runs, in particularly for programs with loops, although they are always successful.
- The bespoke solver is inefficient as it induces more choice points during the satisfiability checking of subpaths than are logically necessary. This results in much fruitless backtracking which makes up most of the execution time of the test data generation process.

On the example above, because of the above points, the run time degrades rapidly with the size of the array in a manner that remains to be investigated.

A complementary example illustrating ATGen capabilities for programs with input variable dependent loops is given in [9].

6. CONCLUSIONS

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

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

This 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 for test data generation is unique to the work described here.

The initials results are promising and further work will center on improving the efficiency of the search algorithms used and the solver. Up-to-date information on the ATGen project is available on [33].

REFERENCES

1. Ould, M.A. Testing—A Challenge to Method and Tool Developers, *Software Engineering Journal*, 6(2):59-91, Mar. 1991.
2. Beizer, B. *Software Testing Techniques*, 2nd Edition, Van Nostrand Reinhold, ISBN 0-442-20672-0, 1990.
3. Meudec, C. Revisiting Symbolic Execution, *In Proceedings of The Institute of Technology 4th Computing and Science Colloquium*, Letterkenny Ireland, May 1999.
4. King, J.C. Symbolic Execution and Program Testing, *Commun. ACM*, 19(7):385-394, 1976.
5. Coward, P.D. Symbolic Execution and Testing, *Information and Software Technology*, 33(1):53-64, Jan./Feb. 1991.

6. 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.
7. Tracey, N., Clark, J., and Mander, K. Automated Program Flaw Finding using Simulated Annealing. In *Proceedings ISSTA'98*, pp. 73-81.
8. Korel, B. Automated Test Data Generation for Programs with Procedures, In *Proceedings ISSTA'96*, pp. 209-215.
9. Demillo, R., and Offutt, A. Constraint-Based Automatic Test Data Generation. *IEEE Trans. Software Eng.*, 17(9):900-910, 1991.
10. Meudec, C. ATGen: Automatic Test Data Generation using Constraint Logic Programming, *In Proceedings IEEE/ACM First International Workshop on Automated Program Analysis, Testing and Verification*, 22nd International Conference on Software Engineering (ICSE 2000), June 2000, pp. 22-31.
11. Danvy, O., Glück, R., and Thiemann, P., (eds). Partial Evaluation, *Lecture Notes in Computer Science*. Vol. 1110, 1996.
12. Cop. Prefixco. PREfix Automated Code Reviewer. *Available at "http://www.prefixco.com/"*, 2001.
13. Clarke, L.A., and Richardson, D.J. Applications of Symbolic Evaluation, *J. Systems Software*, 5:15-35, 1985.
14. 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.
15. Barnes, J. *High Integrity Ada: The SPARK Approach*, Addison-Wesley, ISBN 0-201-17517-7, 1997.
16. Coward, P.D. Symbolic Execution Systems—A Review, *Software Engineering Journal*, 3(6):229-239, Nov. 1988.
17. 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.
18. 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.
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. Offutt, A.J., Jin, Z., and Pan, J. The Dynamic Domain Reduction Procedure for Test Data Generation, *Software Practice and Experience*, 29(2): 167-193, 1999.
21. Freuder, E. The Many Paths to Satisfaction, In *Proceedings ECAI'94 Workshop on Constraint Processing*, Amsterdam, Aug. 1994.
22. Dantzig, G.B. *Linear Programming and Extensions*, Princeton University Press, Princeton, New Jersey, 1963.
23. Bratko, I. *Prolog Programming for Artificial Intelligence*, 2nd Edition, Addison-Wesley, ISBN 0-201-41606-9, 1990.
24. Hamlet, D. Implementing Prototype Testing Tools, *Software Practice and Experience*, 25(4):347-371, Apr. 1995.
25. Jaffar, J., and Lassez, J-L. Constraint Logic Programming, In *Proceedings POPL'87*, pp. 111-119, Munich, Jan. 1987, ACM Press.
26. Colmerauer, A. An Introduction to Prolog III, *Commun. ACM*, 33(7):69-90, Jul. 1990.
27. Cohen, J. Constraint Logic Programming Languages, *Commun. ACM*, 33(7):52-68, Jul. 1990.
28. ECLiPSe Release 5.0, Imperial College London, 2000, *http://www.icparc.ic.ac.uk/eclipse/*
29. British Computer Society Specialist Interest Group in Software Testing (BCS SIGIST), *Standard for Software Component Testing*, Working Draft 3.3, 1997.
30. Paxson V. Flex: A Fast Scanner Generator, Edition 2.5, *The University of California, Berkeley*, Mar. 1995.
31. Donnelly C. and Stallman R. Bison: The YACC-Compatible Parser Generator, Version 1.25, *The Free Software Foundation*, Nov. 1995.
32. The Ada Source Code Treasury, *http://www.adapower.com/adacode.html*, 2001.
33. Meudec, C. The ATGen Project Homepage, *http://members.tripod.co.uk/meudec/atgen/*, 2001.