# Automatic Test Generation for Space

Ulisses Araújo Costa
University of Minho

The European Space Agency (ESA) uses an engine to perform tests in the Ground Segment infrastructure, specially the Operational Simulator. This engine uses many different tools to ensure the development of regression testing infrastructure and these tests perform black-box testing to the C++ simulator implementation. VST (Visionspace Technologies) is one of the companies that provides these services to ESA and they need a tool to infer automatically tests from the existing C++ code, instead of writing manually scripts to perform tests. With this motivation in mind, this paper explores automatic testing approaches and tools in order to propose a system that satisfies VST needs.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms: Languages, Verification

Additional Key Words and Phrases: Automatic Test Generation, UML/OCL, White-box testing, Black-box testing

## 1. INTRODUCTION

Since ever, every industry use testing methods to discover problems in early stages of development process to improve the products quality and software industry is not an exception. Miller[Miller 1981] describe the utility of software testing as:

> The general aim of testing is to affirm the quality of software systems by systematically exercising the software in carefully controlled circumstances.

In the most recent period of software history the integration of software testing as an important step in the process of software development opened up to the origin of *xUnit*[Beck 1989] tools and Agile software development. Also, ESA started to use manual written tests as a part of their software development processes.
Using manual written tests is tedious, time consuming and error-prone. Lots of functions/methods need full code coverage and this technique leads to incomplete test suites; as it is hard to create tests that cover specific code paths, many hidden bugs can be left. Many times a supervision leaded by the developer is needed to assure that the right paths in the code are being tested, specially regarding black-box testing.
Nowadays we start to observe a rapid increase in the automatic test generation field.

### 1.1 Goals

This document correspond to the first milestone in the author's dissertation (developed under a partnership agreement between UM and VST) aimed at producing a

ucosta@visionspacetech.com
Departamento de Informática, Universidade do Minho Master PreThesis Reports 2012

tool that is able to automatically generate interesting testcases for the C++ ESA's Operational Simulator.
This document reviews the most studied techniques and the tools that implement them in order to choose the best set of suitable techniques to incorporate in a automatic testing generator to the Ground Segment infrastructure, specially the Operational Simulator at ESA.
Two different techniques emerge for different purposes, Structural Techniques and Functional Techniques, known respectively as White-box testing and Black-box[Beizer 1995] testing. Functional testing is the most common at ESA, because of the calculation complexity behind the Operational Simulators.
A brief discussion will be presented regarding White-box testing vs. Black-box testing and then some automatic generation techniques will be discussed in more detail. Furthermore the potential of the described tools will be explained, and how they can help on solving the problem VST has nowadays.

### 1.2 White-box vs Black-box testing

In White-box testing the tester needs to understand the internals of the code to be able to write tests for it. The goal of selecting test cases that test specific parts of the code is to cause the execution of specific spots in the software, such as statements, branches or paths. This technique consists in analyzing statically a program, by reading the program code and using symbolic execution techniques to simulate abstract program executions in order to attempt to compute inputs to drive the program along specific execution paths or branches, without ever executing the program. Control Flow based testing approach can be useful to analyze all the possible paths in the code and write unit tests to cover multiple paths. CFG (Control Flow Graph) of the program can be built, test inputs can be generated to make any path execute regarding a given criterion: Select all paths; Select paths to achieve complete statement coverage[Beizer 1990; Ntafos 1988]; Select paths to achieve complete branch coverage[Roper 1994; Beizer 1990]; or Select paths to achieve predicate coverage[Beizer 1990; Ntafos 1988].

Data Flow Testing is designed into looking at the life cycle (creation, usage and destruction) of a particular piece of data and observe how it is used along the CFG, this ensures that the number of paths are always finite[Rapps and Weyuker 1985].

Opposite to White-box testing, Black-box testing is based on functionality, so the tester observes a system based on its functional contracts and writes the pairs of inputs and the expected outputs. This approach is used for unit testing of single methods/functions, integration testing of combinations of the methods/functions, or even final system testing.

This document is organized as follows. In Section 2 the important testing approaches in use—Specification-based testing and Constraint-based generation—are briefly revisited and, for each one, the most relevant tools are identified. In Section 3 some of the tools referred are experimented in order to be compared. Our proposal for a test generation system is introduced in Section 4. The document is concluded in Section 5.

## 2. TESTING TOOLS APPROACHES

In this section, a study of the most recent tools that use Specification-based or Constraint-based tests generation approaches for the most popular languages - C, JAVA and C# will be presented.

### 2.1 Specification-based Generation Testing

Specification Based Testing also known as Model Based Testing refers to the process of testing a program based on what its specification or model says its behavior should be. In particular, can be generated test cases based on the specification of the program's behavior, without seeing an implementation of the program. So this clearly a way of Black-box testing.
With this technique the testing phase and development phase can be started in parallel, don't need the implementation to start the development of test cases. The only thing needed is the functional contracts and/or oracles for each function/method.
Since the 90's there have been some effort into using specifications to try to generate test cases such as Z specifications [Horcher 1995; Stocks and Carrington 1996], UML statecharts[Offutt and Abdurazik 1999],VDM[Aichernig 1999] or ADL specifications[Sankar et al. 1994]. These specifications typically do not consider structurally complex inputs and these tools do not generate junit test cases. Nowadays there are some tools out there that can perform Specification-based Testing approach:

*Conformiq.* is a commercial Tool Suite that generates human-readable test plans and executable test scripts from Java code, state charts and UML[1].

*MaTeLo.* stands for Markov Test Logic and is a commercial tool that generates test sequences from a collection of states, transitions, classes of equivalence, types, sequences, global variables and test oracles using their user interface[2].

*Smartesting CertifyIt.* is a commercial tool that generates test cases from a functional model, as UML[3].

*T-Vec.* is a commercial tool that generates test cases from modeling tools available from T-VEC or third-party vendors[4].

*Rational Tau.* is an IBM commercial tool that provides automated error checking, rules-based model checking, and a model-based explorer using UML[5].

The relevant ones or the recent open-source ones will be discussed.

2.1.1 *Korat.* Korat[Boyapati et al. 2002] is a mature framework for automated testing structurally complex inputs of JAVA programs. Given a formal specification for a method, Korat[6] uses the method precondition to automatically generate all (nonisomorphic) test cases up to a given small size. Korat then executes the method on each test case, and uses the method postcondition as a test oracle to check the

[1]See more at: `http://www.conformiq.com/products.php`
[2]See more at: `http://www.all4tec.net/index.php/All4tec/matelo-product.html`
[3]See more at: `http://www.smartesting.com/index.php/cms/en/product/certify-it`
[4]See more at: `http://www.t-vec.com/`
[5]See more at: `http://www-01.ibm.com/software/awdtools/tau/`
[6]See more at: `http://korat.sourceforge.net/`

correctness of each output.
To be able to generate test cases for a method, Korat uses a predicate and a bound on the size of its inputs, Korat generates all (nonisomorphic) inputs for which the predicate returns *true*. Korat generates all the possible input spaces regarding the predicate and monitor the predicate's executions to be able to prune large portions of the search space.

The writing of a predicate is done using JAVA language and in most cases can be written the first thing that cames to programmer's head to restrict the input space. But for more complex structures it is better to understand how the matching algorithm work to be able to write a fast verifiable predicate.
Unfortunately the test derivation tool using Korat (that also uses JML) is not available to the public.

## 2.2 Constraint-based Generation Testing

Constraint Based Testing[DeMillo and Offutt 1991] can be used to select test cases satisfying specific constraints by solving a set of constraints over a set of variables. The system is described using constraints and these can be solved by SAT solvers. Constraint programming can be combined with symbolic execution, regarding this approach a program is executed symbolically, collecting data constraints over different paths in the CFG, and then solving the constraints and producing test cases from there. This is clearly a White-box testing approach. There are some tools out there, like:

*Euclide.* for verifying safety properties over C code using ACSL annotations, CPBPV for program verification.

*OSMOSE.* a tool that uses concolic execution and path-based techniques over machine code.

*GATeL.* for Lustre language to generate test sequences[7].

Here two tools will be explained, one proprietary and other academic.

### 2.2.1 *Pex.*

Pex[Tillmann and De Halleux 2008] is an automatic white-box test generation tool for .NET. Starting from a method that takes parameters, Pex performs path-bounded model-checking by repeatedly executing the program and solving constraint systems to obtain inputs that will steer the program along different execution paths. This uses the idea of dynamic symbolic execution[Tillmann and Schulte 2006]. Pex uses the theorem prover and constraint solver Z3[8] to reason about the feasibility of execution paths, and to obtain ground models for constraint systems.
Pex came with Moles that helps to generate unit tests. This tools together are able to understand the input (by analyzing branches in the code: declarations, all exceptions throws operations, if statements, asserts and .net Contracts). With this information Pex uses Z3 constraint solver to produce new test inputs which exercise diferent program behavior.
The result is an automatically generated small test suite which often achieves high

[7]See more at: `http://www-list.cea.fr/labos/gb/LSL/test/gatel/index.html`
[8]See more at: `http://research.microsoft.com/en-us/um/redmond/projects/z3/`

code coverage.
Pex can be used in a project, class or method (which make it a very helpful and versatile tool). After the analysis process the "Pex Explorarion Results"shows the *input* × *output* pairs selected for each test case for the method, here it also shows the percentage of the test coverage.

2.2.2 *PathCrawler.* This is an academic tool based on dynamic and statical analysis[Williams et al. 2005], it uses constraint logic programming to generate the Test-cases. PathCrawler[9] executes an instrumented function for each function under test with the generated inputs, it preserves this information to not cover the same path.
This tool supports assertions in any point in the code and pre-conditions regarding the input values.

## 3. USING THE TOOLS

After introducing the theory and the techniques that support each tool, some of the tools will be demonstrated in action, resorting to small but illustrative examples on how each tool can help us to find good test cases.

### 3.1 PathCrawler

Concerning the first case it will be used a simple example based on a function that performs a multiplication, creating a simple branch on the code.

```
1  typedef struct s {
2      int x;
3      int y;
4  }Point;
5
6  int Multiply(Point p) {
7      if(p.x * p.y == 42) return 1;
8      else return 0;
9  }
```

Pointers were tried instead of coping the structure as a parameter to *Multiply* function, but PathCrawler was not able to run.

Nevertheless, PathCrawler was able to give a full coverage for this simple function as you can see in Table I.

Table I. Output Table for *Multiply* function using PathCrawler

| Result | p | return value |
|---|---|---|
| ✓ | Point{x=1,y=42} | 1 |
| ✓ | Point{x=177407,y=109471} | 0 |

Regarding our second example a function that performs a binary search in order to find if a number is in a given range (between two bounds).

[9]See more at: `http://www-list.cea.fr/labos/gb/LSL/test/pathcrawler/index.html`

```
1  int BSearch(int x, int n) {
2      return BinarySearch(x, 0, n);
3  }
4
5  int BinarySearch(int x, int lo, int hi) {
6      while (lo < hi) {
7          int mid = (lo+hi)/2;
8          pathcrawler_assert(mid >= lo && mid < hi);
9          if (x < mid) { hi = mid; }
10                 else { lo = mid+1; }
11     }
12     return lo;
13 }
```

A function that PathCrawler give to us has been used: *pathcrawler_assert*, this function can be used at any location in the program under test, and will force PathCrawler to generate test cases to cover both the case where its argument is true and the case where it is is false. This feature may be seen as another way to write an oracle.
The results were interesting: 31 covered paths and 44 infeasible paths and the test as interrupted by PathCrawler, because PathCrawler reach the maximal test session time (the user can increase this number, but for this example is left the default value).
A further analysis of the results demonstrated that 28 out of the 44 infeasible paths discovered appeared when PathCrawler tried to do the assertion in line 8. Was not written any pre-condition, so PathCrawler does not know that this is a pre-condition for *BinarySearch* function: $lo \leq\ x < hi$. In Table II is shown some of the test inputs generated for this example.

Table II. Output Table for *BSearch* function using PathCrawler

| Result | x | n | return value |
|---|---|---|---|
| ✓ | -189424 | -140714 | 0 |
| ✓ | 157819 | 0 | 0 |
| ✓ | 1 | 1610612736 | 2 |
| ✓ | 2 | 805306368 | 3 |
| ✓ | 11 | 1610612736 | 12 |

PathCrawler was tried with the following function that calculates the year of the $n^{th}$ day after 1980-01-01.

```
1  int IsLeapYear(int year) {
2    return (year % 4 == 0) && ((year % 100 != 0) || (year % 400 == 0)
         );
3  }
4  int FromDayToYear(int day) {
5    int year = 1980;
6
7    while (day > 365) {
8      if (IsLeapYear(year)) {
9        if (day > 366) {
```

```
10          day -= 366;
11          year += 1;
12        }
13      } else {
14        day -= 365;
15        year += 1;
16      }
17    }
18    return year;
19  }
```

The result was unexpectedly *unknown*. PathCrawler was unable to trace even one path in our code, the number of $k$-path's could be increased but with no success for this example.

### 3.2 Pex

Regarding Pex is used the same examples showed previously adapted to C# language. Because C# is a more expressive language than C our examples will be improved with some other OO and C# specific features like Exceptions and Debug.Assert calls. In fact Pex can also support a lot more features that are present in C# language like .NET Contracts and many more.
This is the simple implementation of a 2D *Point* class that has been created to have special behavior, under a certain condition $x \times y \equiv 42$ it is supposed to throw an exception.

```
1  public class Point {
2    public readonly int X, Y;
3    public Point(int x, int y) { X = x; Y = y; }
4  }
5
6  public class Multiply {
7    public static void multiply(Point p) {
8      if (p.X * p.Y == 42)
9          throw new Exception("hidden bug!");
10   }
11 }
```

So, as was described earlier, Pex will try to generate such input as it is possible (in a given amount of time) to traverse all the paths inside the code. The output table can be seen in Table III, with the inputs and outputs that Pex found to ensure a full coverage of the code.

Table III. Output Table for *multiply* method using Pex

| Result | p | Output/Exception | Error Message |
|---|---|---|---|
| ✗ | null | NullReferenceException | Object ref. not set to an instance of an object. |
| ✓ | new Point{X=0,Y=0} | | |
| ✗ | new Point{X=3,Y=14} | Exception | hidden bug! |

Pex was successful to reach the *Exception* path inside the code. Of course this is not always possible, since sometimes the functions inside the *if* statement does not have inverse function.

Pex can also be very helpful checking assertions and contracts in .net code. A binary search algorithm was written and an assertion was also written in the middle of our code.

```
1  public class Program {
2    public static int BSearch(int x, int n) {
3      return BinarySearch(x, 0, n);
4    }
5    static int BinarySearch(int x, int lo, int hi) {
6      while (lo < hi) {
7        int mid = (lo+hi)/2;
8        Debug.Assert(mid >= lo && mid < hi);
9        if (x < mid) { hi = mid; } else { lo = mid+1; }
10     }
11     return lo;
12   }
13 }
```

Pex was able to generate an input that could not pass in the assertion inerted in our code, as can be seen in Table IV.

Table IV. Output Table for *BSearch* method using Pex

| Result | x | n | result | Output/Exception |
|---|---|---|---|---|
| ✓ | 0 | 0 | 0 | |
| ✓ | 0 | 1 | 1 | |
| ✓ | 0 | 3 | 1 | |
| ✗ | 1073741888 | 1719676992 | | TraceAssertionException |
| ✓ | 1 | 6 | 2 | |
| ✓ | 50 | 96 | 51 | |

## 3.3 Korat

Like was explained before, Korat generates a graphical representation of the structure instances that validates the property *repOK*. This property was written using JAVA code.
In order to test the freelly available version of Korat, a Doubly Linked List structure was created in JAVA.

```
1  public class LinkedList<T> {
2    public static class LinkedListElement<T> {
3      public T Data;
4      public LinkedListElement<T> Prev;
5      public LinkedListElement<T> Next;
6    }
7    private LinkedListElement<T> Head;
8    private LinkedListElement<T> Tail;
```

$$\langle \forall\ l : l \in LinkedList\ : Head(l) \equiv null \vee Tail(l) \equiv null \Leftrightarrow size(l) \equiv 0 \rangle \quad (1)$$
$$\langle \forall\ l : l \in LinkedList\ : Tail(l).Next \equiv null \rangle \quad (2)$$
$$\langle \forall\ l : l \in LinkedList\ : Head(l).Prev \equiv null \rangle \quad (3)$$
$$\langle \forall\ l : l \in LinkedList\ : size(l) \equiv 1 \Leftrightarrow Head(l) \equiv Tail(l) \rangle \quad (4)$$
$$\langle \forall\ l : l \in LinkedList\ : \langle \forall\ e_1, e_2 : \{e_1, e_2\} \subseteq\!\!\subseteq l\ : \langle \exists\ e : e \in\!\!\in l\ : e_1.Next \equiv e \wedge e_2.Prev \equiv e\ \rangle\ \rangle \quad (5)$$
$$\langle \forall\ l : l \in LinkedList\ : \langle \forall\ e_1, e_2 : \{e_1, e_2\} \subseteq\!\!\subseteq l\ : e_1 \equiv e_2 \Rightarrow i(e_1) \equiv i(e_2)\ \rangle\ \rangle \quad (6)$$

Fig. 1. Invariants for class *LinkedList*

```
9     private int size;
10  }
```

Now the *repOK* predicate method must be defined. This predicate method will check that the tree doesn't have any cycles and that the number of nodes traversed from root matches the value of the field size. First was defined the properties about this data structure. The most relevant ones are property 5 in Figure 1 that ensures the structure and property 6 that ensures our doubly linked list does not have repeated elements.
Consider $e, e_1, e_2 \in LinkedListElement$ and $i$ the index function: $i : LinkedListElement \rightarrow int$, that receives an element of *LinkedList* and returns the position of that element in the structure. Consider also three new functions:

(1) $Head(l)$ being $l$ of type *LinkedList* and meaning in Java code $l.Head$.
(2) $Tail(l)$ being $l$ of type *LinkedList* and meaning in Java code $l.Tail$.
(3) $size(l)$ being $l$ of type *LinkedList* and meaning in Java code $l.size$.

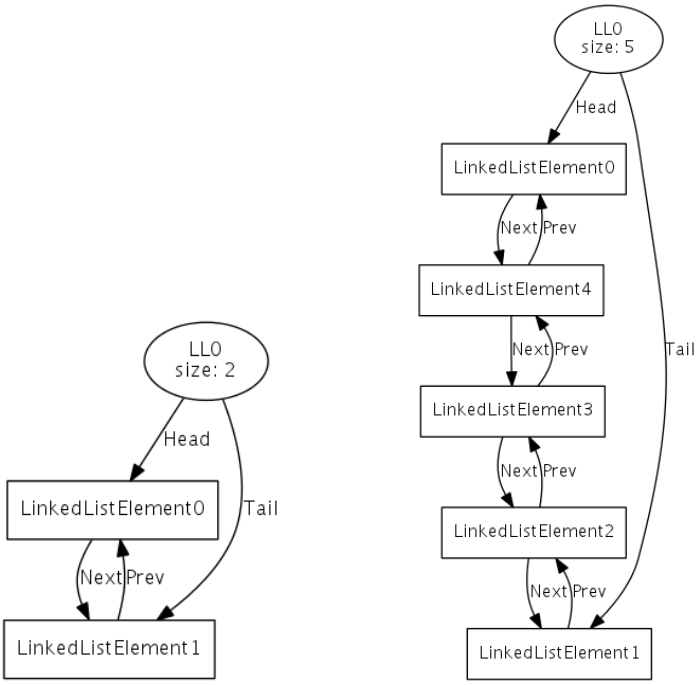
As a matter of avoiding verbosity two symbols were defined:

(1) $a \in\!\!\in l$ being $a$ of type *LinkedListElement* and meaning that $a$ is an element of the *LinkedList* $l$.
(2) $\{a, \ldots, z\} \subseteq\!\!\subseteq l$ meaning $a \in\!\!\in l \wedge \ldots \wedge z \in\!\!\in l$.

The properties in Fig. 1 were taken and used to restrict the generation of structures using Java. So the *repOK* method that receives a *LinkedList* structure and returns *Bool* whenever this structure follows the invariants in 1 was defined. Using this specification Korat generated the 2 structures showed in Figure 2. In Figure 2a with 2 elements and in Figure 2b an instance with 5 elements.

### 3.4 Summary

After the experimental study of the selected tools, reported in the previous subsections, it was found that PathCrawler and Pex have different approaches regarding testcase generation. PathCrawler seems to be a very efficient tool to discover multiple infeasible paths in C code, because it uses a mix between static and dynamic analysis. When it finds a suitable input for a function it tries to execute collecting all the executed paths in the code. Pex on the other side just uses static execution and it is very efficient discovering all the feasible paths in C# methods. Pex was also used to perform testcase generation in C# classes, but the generated instances are




(a) Instance with 2 elements for *LinkedList*

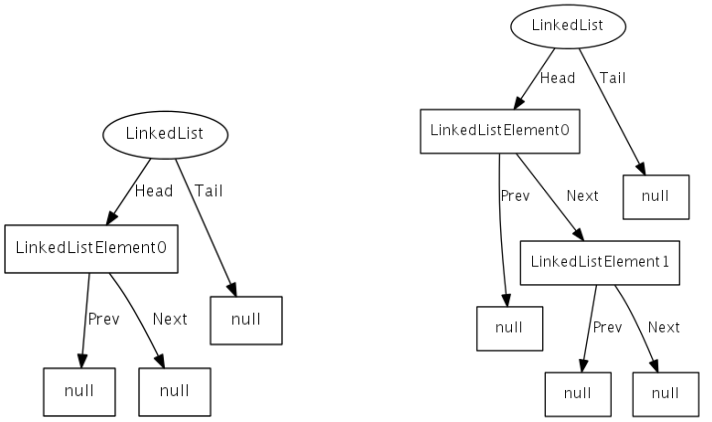
(b) Instance with 5 elements for *LinkedList*

Fig. 2. Examples of generated instances from Korat for *LinkedList* class.

too simple to perform more interesting tests. The *LinkedList* class was written in C# with many management methods implemented (Add, Remove, Find,...). Pex generated very simple *LinkedList*'s structures to perform automatic test generation for each implemented method. The problem is that the generated structures does not meet the properties about Doubly Linked Lists as it can be seen in Figure 3. Concerning Korat, this is The tool to generate complex data structures. The freely available part of Korat show potential in expressing rules to hedge the automatic generation of data structures.

## 4. GENERATE TESTS FROM CODE+OCL

Since the Operational Simulator code is not familiar to us, it was decided to start solving this problem by inferring the UML+OCL from the existing code to be able to work on a more abstract level rather than the implementation. The idea is to extract tests from the inferred OCL, using the Partition Analysis described in [Benattou et al. 2002] and at the same time generate tests directly from the code, using symbolic execution to complement the specification-based generation from OCL. The main goal is to extract as many tests as possible from a model and from the implementation to provide information to a feedback loop[Xie and Notkin 2003] test generation framework with two test prespectives, functional and




(a) Example of Pex generated *LinkedList* instance to test *Remove* method

(b) Example of Pex generated *LinkedList* instance to test *Find* method

Fig. 3. Examples of generated instances from Pex for *LinkedList* class.

structural, and from there be able to get a more refined set of tests.
A combination of both, symbolic execution from Pex and complex data generation from Korat, it will be designed and implemented to generate more interesting inputs for the methods under testing.

## 5. CONCLUSION

Looking for an efficient solution to automatically generate complete test sets for complex and critical C++ software, the state-of-the-art approaches in the area were studied and along the document some tools were introduced from methodological and experimental perspectives. Pex has proved to be a very powerful tool, aimed at offering a full coverage. However, the uncapability for generating calling-methods sequences was a bit disappointing. With Microsoft's SpecExplorer we can already manually call sequences of methods; maybe a combination of this feature with Pex would make Pex a perfect all-in-one testing tool regarding .NET automatic testing tools. Concerning Korat, the expected improvement is just to write the invariants for a class instead of the *repOK* method, or maybe infer these invariants from the existing code. Writing the *repOK* method for very complex data structures requires some previous experience with Korat, but we think this is not a weakness, since the tester quickly gets used to write the *repOK* method in Korat. The only problem is that right now we can not fully automate the process without human help.

Considering the studied tools and thinking about a full automated test generation tool, a clever composition among between Pex to ensure the maximum possible coverage, Korat to generate all the valid data structures and an automatic tool to generate calls to methods combinations would be the perfect tool.

At the end, it was proposed an approach based on the inference of tests from a

Code+OCL. Concerning the OCL inference from C++ code, work will now be done on a tool that implements it. For that purpose, Frama-C will be explored, as it is well known that this tool is able to infer pre- and postconditions[Moy 2009] and interesting safety conditions from C source code.

## REFERENCES


Aichernig, B. K. 1999. Automated black-box testing with abstract vdm oracles. In *Computer Safety, Reliability and Security: proceedings of the 18th International Conference, SAFECOMP'99*. Springer, 250–259.

Beck, K. 1989. Simple Smalltalk Testing: With Patterns. Tech. rep., First Class Software, Inc.

Beizer, B. 1990. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA.

Beizer, B. 1995. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., New York, NY, USA.

Benattou, M., Bruel, J.-M., and Hameurlain, N. 2002. Generating test data from ocl specification.

Boyapati, C., Khurshid, S., and Marinov, D. 2002. Korat: Automated testing based on java predicates. In *IN PROC. INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS (ISSTA*. ACM Press, 123–133.

DeMillo, R. A. and Offutt, A. J. 1991. Constraint-based automatic test data generation. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING 17*, 9, 900–910.

Horcher, H.-M. 1995. Improving software tests using z specifications. In *In Proc. 9th International Conference of Z Users, The Z Formal Specification Notation*. Springer-Verlag, 152–166.

Miller, F. 1981. *Introduction to Software Testing Technology*. Tutorial: Software Testing and Validation Techniques. IEEE Computer Society Press, 4–16.

Moy, Y. 2009. Automatic modular static safety checking for c programs. Ph.D. thesis, UniversitÃľ Paris-Sud.

Ntafos, S. C. 1988. A comparison of some structural testing strategies. *IEEE Trans. Softw. Eng. 14*, 868–874.

Offutt, J. and Abdurazik, A. 1999. Generating tests from uml specifications. In *Proceedings of the 2nd international conference on The unified modeling language: beyond the standard*. UML'99. Springer-Verlag, Berlin, Heidelberg, 416–429.

Rapps, S. and Weyuker, E. 1985. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering 11*, 367–375.

Roper, M. 1994. *Software Testing*. The International Software Engineering Series. McGraw-Hill.

Sankar, S., Hayes, R., Sankar, S., and Hayes, R. 1994. Specifying and testing software components using adl. Tech. rep.

Stocks, P. and Carrington, D. 1996. A framework for specification-based testing. *IEEE Trans. Softw. Eng. 22*, 777–793.

Tillmann, N. and De Halleux, J. 2008. Pex: white box test generation for .net. In *Proceedings of the 2nd international conference on Tests and proofs*. TAP'08. Springer-Verlag, Berlin, Heidelberg, 134–153.

Tillmann, N. and Schulte, W. 2006. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Software 23*, 2006.

Williams, N., Marre, B., Mouy, P., and Roger, M. 2005. Pathcrawler: automatic generation of path tests by combining static and dynamic analysis. In *In: Proc. European Dependable Computing Conference. Volume 3463 of LNCS (2005) 281âĂŞ292*. Springer. ISBN, 281–292.

Xie, T. and Notkin, D. 2003. Mutually enhancing test generation and specification inference. In *In Proc. 3rd International Workshop on Formal Approaches to Testing of Software, volume 2931 of LNCS*. 60–69.