Automatic Test Generation for Space

Ulisses Araújo Costa[1], Daniela da Cruz[2], and Pedro Rangel Henriques[3]

1 **VisionSpace Technologies**
Rua Alfredo Cunha, 37, Matosinhos, Portugal
ucosta@visionspace.com
2 **Department of Informatics, University of Minho**
Campus de Gualtar, 4710-057, Braga, Portugal
danieladacruz@gmail.com
3 **Department of Informatics, University of Minho**
Campus de Gualtar, 4710-057, Braga, Portugal
pedrorangelhenriques@gmail.com

arXiv:1206.5104v1 [cs.SE] 22 Jun 2012

Abstract

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.

1998 ACM Subject Classification D.2.5 - Software Engineering, Testing and Debugging

Keywords and phrases Automatic Test Generation, UML/OCL, White-box testing, Black-box testing

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.185

1 Introduction

Since ever, every industry use testing methods to discover problems in early stages of the development process to improve the products quality, and software industry is not an exception. Miller[22] 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*[7] 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 practice 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.

© Ulisses Araújo Costa and Daniela da Cruz and Pedro Rangel Henriques;
licensed under Creative Commons License NC-ND
1st Symposium on Languages, Applications and Technologies.
Editors: Alberto Simões and Ricardo Queirós and Daniela da Cruz; pp. 185–202

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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 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 an 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[8] testing and Black-box[9] 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. First of all an explanation about the Operational Simulator Infrastructure will be provided.

1.2 Operational Simulator Infrastructure

ESA's Operational Simulator called Simulation Infrastructure for the Modeling of SATellites (SIMSAT) is a satellite simulator that model and simulate the behavior of satellites in order to allow operators[1] train more effectively and help them to define the satellites' operational processes.

The simulator consists of operational models of the various internal components of the satellite from their main computer to its payload (instruments aboard the satellite), which interact with each other and thus define the behavior of the satellite. VST has participated in the development of tests to validate the operational simulator. The development of these simulators is based on operating rules simulation of ESA – Simulation Model Portability (SMP)[2], as well as in infrastructure SIMSAT simulation. This standard is infrastructure agnostic of any space specific model, so any other needs of simulation can be used, such as defense, transport, energy, etc.

Here is a brief description of each component in SIMSAT[3]:

SIMSAT Kernel this is a generic simulation infrastructure providing the framework for the running of space systems simulators.

SIMSAT Man-Machine Interface (MMI) this is a generic Graphical User Interface enabling the user interaction with the simulator's components.

---

[1] Operators are responsible for the operation of the satellite after its launch.

[2] SMP is based on the ideas of component-based design and Model Driven Architecture (MDA) as promoted by the Object Management Group (OMG) and is based on the open standards of UML and XML. One of the basic principles is the separation of the platform specific and platform independent aspects of the simulation model. This protects the investments in the model from changes in technology by defining the model in a platform independent way, which can then be mapped into different technologies. Further the SMP specification provides standardised interfaces between the simulation models and the simulation run-time environment for common simulation services as well as a number of mechanisms to support inter-model communication.[1, 2, 3, 4, 5]

[3] More information in: http://www.egos.esa.int/portal/egos-web/products/Simulators/simsat/intro-sim.html

Ground Models this is a family of SIMSAT compatible models enabling a realistic simulations of all ground systems between the spacecraft (or spacecraft model) and the control centre at European Space Operations Centre (ESOC).

Emulator Suite On-board Processor Emulators support the execution in satellite simulators of the real flight software.

Generic Models a set of generic space models that ease the developments of the spacecraft models used in operational simulators.

Ground Systems Test and Validation Applications (GSTV) this is a family of test simulators that are based on the generic simulators infrastructure components listed above and are able to support the different levels of testing of ground infrastructure systems.

Moreover the SIMSAT Kernel is made up of several components[4]:

Scheduler is responsible for the co-ordination and processing of all events within the Simulation Kernel. An event on the schedule identifies an action that needs to be performed at a specified point in simulated time.

Mode Manager is the simulation state machine. The Simulation has a number of operational modes, which control the operation of the simulation.

Time-Manager is responsible for maintaining and providing models and the MMI with the correct simulation-Time. It provides time in four formats, Simulation-Time, Epoch-Time, Zulu-Time and Correlated Zulu-Time. this is a family of SIMSAT compatible models enabling a realistic simulations

Logger supports the recording of Kernel or model events that occur during a simulation. The log in which the current simulation messages are written is called the active log. The logger also provides a view of the simulation event history in an MMI during a simulation session.

Visualization manager is responsible for making the values of both model and Kernel data items available for display in an MMI.

State-vector manager is responsible for the saving and restoring of the state of the simulation. Its main purpose is to allow the Simulation State, at any point in the simulation, to be saved. This allows the user to return to an earlier simulation scenario.

Command handler is responsible for the reception and execution of Kernel and user defined commands.a set of generic space models that ease the developments of the spacecraft models used in operational.

Command procedure interpreter is responsible for the interpretation of command procedures. A command procedure contains Kernel and User defined simulator commands and supports a procedural language to control the flow of these commands. The execution of command procedures is controlled directly from the MMI.

Right now, to be able to perform tests in the Operational Simulator, in order to validate SIMSAT, VST Engineers need to write scripts that perform simulations and validate the results using GUI interfaces (SIMSAT MMI). This job can be tedious and difficult to replicate.
So a first solution will have to go through a preliminary study of the tools that currently exist with which we can generate tests automatically. By studding these tools we do not hope to find the perfect solution, but combine techniques to obtain an optimal solution to improve VST work.

[4] More information in: http://www.egos.esa.int/portal/egos-web/products/Simulators/SIMSAT/

1.3 White-box vs Black-box testing

In this subsection is discussed the two most common approaches for testing: White-box and 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. The 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[8, 24]; Select paths to achieve complete branch coverage[28, 8]; or Select paths to achieve predicate coverage[8, 24].

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 is always finite[27].

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, Constraint-based, Grammar-based and Random-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 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, we do not need the implementation to start the development of test cases. The only thing needed is the functional contracts and/or oracles[5] for each function/method.
Since the 90's there have been some effort into using specifications to try to generate test

[5] A test oracle determines whether or not the results of a test execution are correct[26].

cases such as Z specifications [19, 30], UML statecharts[25],VDM[6] or ADL specifications[29]. 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[6].

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[7].

Smartesting CertifyIt is a commercial tool that generates test cases from a functional model, as UML[8].

T-Vec is a commercial tool that generates test cases from modeling tools available from T-VEC or third-party vendors[9].

Rational Tau is an IBM commercial tool that provides automated error checking, rules-based model checking, and a model-based explorer using UML[10].

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

2.1.1 Spec Explorer

This is a Microsoft model-based testing that uses one software modeling languages, the AsmL (Abstract State Machine Language). This modeling language provides the foundations of the Spec Explorer[11] tool and Spec# that is a formal language for API contracts (influenced by JML, AsmL, and Eiffel), which extends C# with constructs for non-null types, pre-conditions, post-conditions, and object invariants[12]. These tool is already available to users and is in a very mature phase.

The user of Spec Explorer writes a model of the system and sets the possible values for some properties in his code, furthermore the user also provides a scenario. These scenarios are simple sets of calls to methods without their parameters (remember that this is Spec Explorer job). Then Spec Explorer will generate a visual graph where each node represents a state of the system and the arrows represent a call to some method. It searches throw all possible sequences of methods invocation that do not violate the contracts (pre, pos conditions) and that are relevant to a user-specified set of test properties. After that we can generate from this visual graphs the unit tests (the arrows) and the test cases (a graph).

2.1.2 JMLUnit

JMLUnit[15] is a tool that automates the generation of oracles for JAVA testing classes. This tool monitors the specified behavior of the method being tested to decide whether the test passed or failed. This monitoring is done using the formal specification language runtime assertion checker. The main idea behind these tools is to translate the pre- and post-conditions methods into the code of the testing method.

[6] See more at: `http://www.conformiq.com/products.php`
[7] See more at: `http://www.all4tec.net/index.php/All4tec/matelo-product.html`
[8] See more at: `http://www.smartesting.com/index.php/cms/en/product/certify-it`
[9] See more at: `http://www.t-vec.com/`
[10] See more at: `http://www-01.ibm.com/software/awdtools/tau/`
[11] See more at: `http://research.microsoft.com/en-us/projects/specexplorer/`
[12] See more at: `http://research.microsoft.com/en-us/projects/specsharp/`

The pre-conditions became the criteria for selecting test inputs, and the post-conditions provided the properties to check for test results. So, the post-conditions became the test oracles.
This tool uses the JML[12] specification language to annotate JAVA methods code with pre- and post-conditions and automatically generate JUnit test classes from JML specifications.

2.1.3 TestEra

TestEra[21] can be used to perform automated specification-based testing of JAVA programs. This framework requires as input a JAVA method, a formal specification[13] of the pre and post-conditions of that method, and a bound that limits the size of the test cases to be generated.
With the pre-condition it automatically generates all non-isomorphic test inputs up to the given bound. It executes the method on each test input, and uses the method post-condition as an oracle to check the correctness of each output. This tool uses Alloy's[14] SAT system to analyze first-order formulae. The authors claim that have used TestEra to check several JAVA programs including an architecture for dynamic networks, the Alloy-alpha analyzer, a fault-tree analyzer, and methods from the JAVA Collection Framework.

2.1.4 Korat

Korat[11] is a mature framework for automated testing structurally complex inputs of JAVA programs. Given a formal specification for a method, Korat[15] uses the method pre-condition to automatically generate all (non-isomorphic) test cases up to a given small size. Korat then executes the method on each test case, and uses the method post-condition as a test oracle to check the 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 (non-isomorphic) 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[18] 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

---

[13] Specifications are first-order logic formulae.
[14] Alloy is a first-order declarative language based on sets and relations. The Alloy Analyzer is a fully automatic tool that finds instances of Alloy specifications: an instance assigns values to the sets and relations in the specification such that all formulae in the specification evaluate to true.
[15] See more at: `http://korat.sourceforge.net/`

CFG, and then solving the constraints and producing test cases from there. 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[16].

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

2.2.1 Pex

Pex[32] 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[33]. Pex uses the theorem prover and constraint solver Z3[17] 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. These 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 code coverage.
Pex can be used in a project, class or method (which makes it a very helpful and versatile tool). After the analysis process the "Pex Explorarion Results" shows the $input \times output$ 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 static analysis[34], it uses constraint logic programming to generate the Test-cases. PathCrawler[18] 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.

2.3 Grammar-based Generation Testing

In this approach inputs to a system under test are defined by a context-free grammar. The language of the grammar contains all possible test cases. Using this approach to describe the syntax of the input to the system under test proves to be very helpful to test network protocols[31, 20] and parsers and compilers[13, 14].

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

2.3.1 ASTGen

ASTGen[17] is a JAVA framework that automates testing of refactoring engines: generation of test inputs and checking of test outputs. The main technique is an iterative generation of structurally complex test inputs. ASTGen[19] allows developers to write imperative generators whose executions produce input programs for refactoring engines. More precisely, ASTGen offers a library of generic, reusable, and composable generators that produce abstract syntax trees (ASTs).
So, ASTGen ensures the production of test inputs instead of the developer produce them. The developer needs to write a generator whose execution produces thousands of programs with structural properties that are relevant for the specific refactoring being tested. This tool has found 21 bugs in Eclipse and 26 bugs in Netbeans applications.

2.4 Random-based Generation Testing

In the random testing approach, test inputs are selected randomly from the input domain of the system. To have a random testing suite first we must identify the input domain, after that select test inputs independently from the domain, then the system under test is executed on these inputs, the results are compared to the system specification, an oracle.
Random testing gives us an advantage of easily estimating software reliability from test outcomes. Test inputs are randomly generated according to an operational profile, and failure times are recorded. The data obtained from random testing can then be used to find bugs or non expected behaviors.

The main problem regarding random generation is the problem of the coverage, it is possible that it will not be broad enough. And furthermore it can be too sparse to actually test specifics parts of the program. Either way, this technique proves to be very effective for testing compilers.

2.4.1 Csmith

Csmith[36] is a black-box random tests generator that is able to generate C programs conform to the C99[20] standard. This is a very recent tool that already discover more than 195 bugs in LLVM and 79 bugs in GCC. With Csmith we are able to generate random programs with unambiguous meanings (undefined behavior or unspecified behavior). Does not attempt to generate terminating program, so they use timeouts for long time consuming generated programs. And the main supported features right now are: Arithmetic, logical, and bit operations on integers, Loops, Conditionals, Function calls, Const and volatile, Structs and Bitfields, Pointers and arrays, Goto, Break and continue. The generation of code regarding this features can be tuned using the command line program.

2.4.2 QuickCheck for JAVA

QuickCheck was originally a combinator library for the Haskell[21] programming language[16]. Later on QuickCheck philosophy spread to other programming languages like: JAVA, Erlang, Perl, Ruby and JavaScript.
QuickCheck works by generating high amounts of data (within the method domain) and

[19] See more at: `http://mir.cs.illinois.edu/astgen/`
[20] See more at: `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`
[21] See more at haskell.org

checking it against a given property, it is expected to create a wide range of the input domain, thus increasing the chances of giving more test coverage.

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 a simple example will be used 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 1.

Table 1 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).

```
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 gives 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

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 was 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. No pre-condition was written, so PathCrawler does not know that this is a pre-condition for *BinarySearch* function: $lo \leq \ x < hi$. In Table 2 is shown some of the test inputs generated for this example.

Table 2 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 nth 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, we used the same examples shown 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 3, with the inputs and outputs that Pex found to ensure a full coverage of the code.

Table 3 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 4.

Now we have a more complex example, a function that returns the year of the nth day after 1980-01-01. Pex was able to generate some important test cases, but it has reached the limit amount of time to calculate interesting paths in the code, this boundary prevents Pex from getting stuck when the program goes into an infinite loop.

```
1  public class Program {
```

Table 4 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	

```
2     private static bool IsLeapYear(int year) {
3       return (year % 4 == 0) && ((year % 100 != 0) || (year % 400 == 0));
4     }
5     public static void FromDayToYear(int day, out int year) {
6       year = 1980;
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    }
19  }
```

Pex was unable to discover the year for day 366 and 7671 as we can see in Table 5. This problem occurred because Pex by default has a maximum number of conditions, this avoids never ending functions and still has a result from Pex. In this particular case we could increment the number of $MaxConditions$: $[PexMethod(MaxConditions = 10000)]$.

Table 5 Output Table for $FromDayToYear$ method using Pex

Result	day	out year	Output/Exception
✓	0	1980	
✓	367	1981	
⚠	366		path bounds exceeded
✓	1023	1982	
✓	2561	1987	
✓	7874	2001	
⚠	7671		path bounds exceeded

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;
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 ($\Subset$ and $\Subset\!\!\subseteq$, these symbols are used to define the $LinkedList$ invariants in Figure 1):

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

We took the properties described in Figure 1 and use them to restrict the generation of structures as we can see in the following Java implementation code. Note that we using short-circuiting, so we return $false$ as soon as we can. This way Korat will be able to generate faster the instances matching our criteria.

$$\langle \forall\ l : l \in LinkedList\ \ : Head(l) \equiv null \vee Tail(l) \equiv null \Leftrightarrow size(l) \equiv 0\ \rangle \quad (1)$$

$$\langle \forall\ l : l \in LinkedList\ \ : Tail(l).Next \equiv null\ \rangle \quad (2)$$

$$\langle \forall\ l : l \in LinkedList\ \ : Head(l).Prev \equiv null\ \rangle \quad (3)$$

$$\langle \forall\ l : l \in LinkedList\ \ : size(l) \equiv 1 \Leftrightarrow Head(l) \equiv Tail(l)\ \rangle \quad (4)$$

$$\langle \forall\ l : l \in LinkedList\ \ : \langle \forall\ e_1, e_2 : \{e_1, e_2\} \Subset\!\!\subseteq l\ \ : \langle \exists\ e : e \Subset l\ \ : e_1.Next \equiv e \wedge e_2.Prev \equiv e\ \rangle\ \rangle\ \rangle \quad (5)$$

$$\langle \forall\ l : l \in LinkedList\ \ : \langle \forall\ e_1, e_2 : \{e_1, e_2\} \Subset\!\!\subseteq l\ \ : e_1 \equiv e_2 \Rightarrow i(e_1) \equiv i(e_2)\ \rangle\ \rangle \quad (6)$$

Figure 1 Invariants for class *LinkedList*

```
1  public boolean repOK() {
2    if(Head == null || Tail == null)
3      return size == 0;
4    if(size == 1) return Head == Tail;
5    if(Head.Prev != null) return false;
6    if(Tail.Next != null) return false;
7    LinkedListElement<T> last = Head;
8    Set visited = new HashSet();
9    LinkedList workList = new LinkedList();
10   visited.add(Head);
11   workList.add(Head);
12   while (!workList.isEmpty()) {
13     LinkedListElement<T> current = (LinkedListElement<T>) workList.
           removeFirst();
14     if (current.Next != null) {
15       if (!visited.add(current.Next))
16             return false;
17       workList.add(current.Next);
18       if(current.Next.Prev != current) return false;
19       last = current.Next;
20     }
21   }
22   if(last != Tail)
23     return false;
24   return (visited.size() == size);
25 }
```

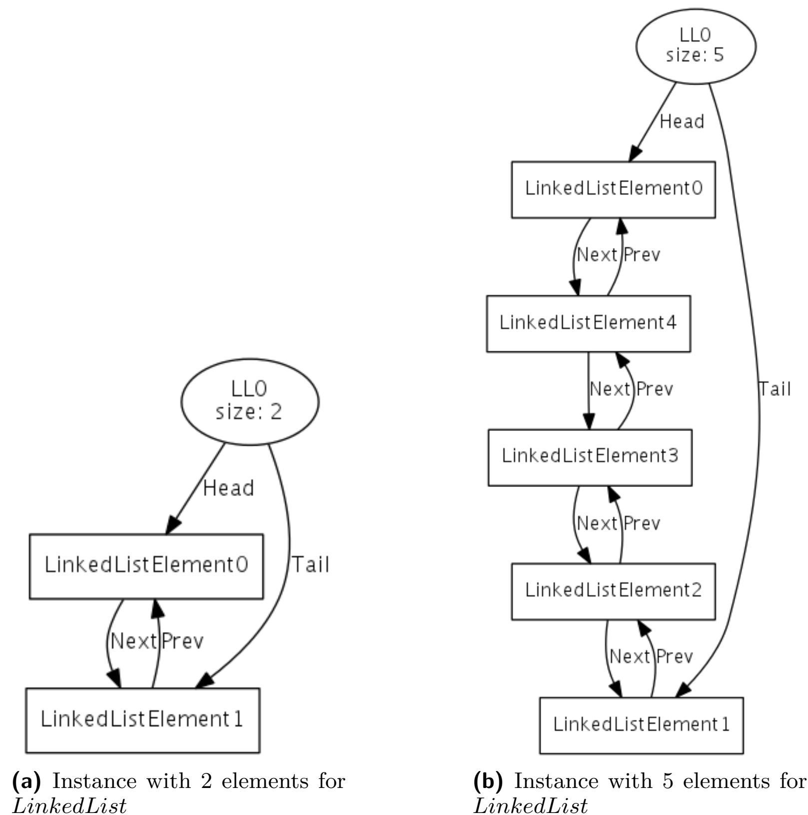
The last step was defining the finitization method, this way we tell Korat how to bound the input space.

```
1  public static IFinitization finLL(int nodesNum, int minSize, int
       maxSize) {
2    IFinitization f = FinitizationFactory.create(LL.class);
3    IObjSet nodes = f.createObjSet(LinkedListElement.class, nodesNum,
         true);
4    f.set("Head", nodes);
5    f.set("Tail", nodes);
6    f.set("size", f.createIntSet(minSize, maxSize));
7    f.set("LinkedListElement.Next", nodes);
8    f.set("LinkedListElement.Prev", nodes);
9    return f;
10 }
```

The properties in Figure 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 shown 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 too simple to perform more interesting tests. The *LinkedList* class was written in C# with many management methods implemented (Add, Remove, Find,. . .).



(a) Instance with 2 elements for *LinkedList*

(b) Instance with 5 elements for *LinkedList*

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

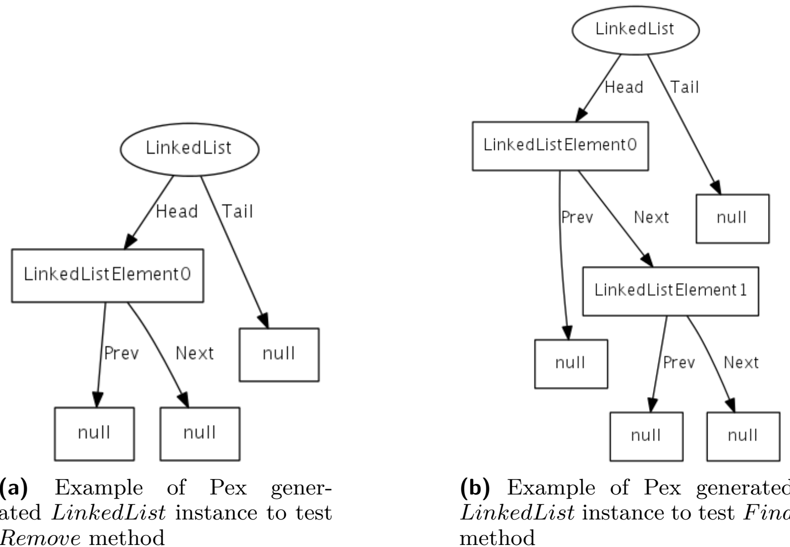
Pex generated very simple *LinkedList*'s structures to perform automatic test generation for each implemented method. The problem is that the generated structures does not meet the properties about Doubly Linked Lists as it can be seen in Figure 3. Concerning Korat, this is The tool to generate complex data structures. The freely available part of Korat show potential in expressing rules to hedge the automatic generation of data structures.
In Table 6 we can see a brief comparison between all the experimented and mentioned tools, a more detailed conclusion is addressed in Chapter 5.

4 Generate Tests from Code+OCL

Since the Operational Simulator code is not familiar to us, regarding its implementation, 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 [10] 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[35] test generation framework with two test prespectives, functional and 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

Name	Target Language	Black/White-box	Additional Input	Output	Comments
PathCrawler	C	White-box (symbolic execution)	Test vectors	Constraints about the executed paths	Too Complex
Pex	C#	White-box (symbolic execution)	–	Unit Tests	Poor generated data instances (objects)
Korat	JAVA	Black-box	Invariants written in JAVA	Graphical form of data structures (using Alloy-GraphViz)	Powerful generating valid data instances

Table 6 Comparison of experimented and mentioned tools



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

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

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

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 incapability 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 post-conditions[23] and interesting safety conditions from C source code.

References

1. European Space Agency. In *ECSS-E-TM-40-07 Volume 1A - Simulation modelling platform - Volume 1: Principles and requirements*, 2011 January.
2. European Space Agency. In *ECSS-E-TM-40-07 Volume 2A - Simulation modelling platform - Volume 2: Metamodel*, 2011 January.
3. European Space Agency. In *ECSS-E-TM-40-07 Volume 3A - Simulation modelling platform - Volume 3: Component model*, 2011 January.
4. European Space Agency. In *ECSS-E-TM-40-07 Volume 4A - Simulation modelling platform - Volume 4: C++ Mapping*, 2011 January.
5. European Space Agency. In *ECSS-E-TM-40-07 Volume A5 - Simulation modelling platform - Volume 5: SMP usage*, 2011 January.
6. Bernhard K. Aichernig. Automated black-box testing with abstract vdm oracles. In *Computer Safety, Reliability and Security: proceedings of the 18th International Conference, SAFECOMP'99*, pages 250–259. Springer, 1999.
7. Kent Beck. Simple Smalltalk Testing: With Patterns. Technical report, First Class Software, Inc., 1989.
8. Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.
9. Boris Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., New York, NY, USA, 1995.
10. Mohammed Benattou, Jean-Michel Bruel, and Nabil Hameurlain. Generating test data from ocl specification, 2002.
11. Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on java predicates. In *IN PROC. INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS (ISSTA*, pages 123–133. ACM Press, 2002.
12. Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of jml tools and applications, 2003.
13. C. J. Burgess. The automated generation of test cases for compilers. *Sofwtware testing, Verification and Reliability*, 4(2):81–99, June 1994.
14. C J Burgess and M Saidi. The automatic generation of test cases for optimizing fortran compilers. *Information and Software Technology*, 38(2):111–119, 1996.
15. Yoonsik Cheon, Yoonsik Cheon, Gary T. Leavens, and Gary T. Leavens. The jml and junit way of unit testing and its implementation. Technical report, 2004.

16 Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, September 2000.

17 Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 185–194, New York, NY, USA, 2007. ACM.

18 Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 17(9):900–910, 1991.

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

20 R. Kaksonen and Valtion teknillinen tutkimuskeskus. *A functional method for assessing protocol implementation security*. VTT publications. Technical Research Centre of Finland, 2001.

21 Sarfraz Khurshid and Darko Marinov. Testera: Specification-based testing of java programs using sat. *Automated Software Engg.*, 11:403–434, October 2004.

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

23 Yannick Moy. *Automatic Modular Static Safety Checking for C Programs.* PhD thesis, Université Paris-Sud, 2009.

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

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

26 Dennis Peters. Generating a test oracle from program documentation, 1995.

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

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

29 Sriram Sankar, Roger Hayes, Sriram Sankar, and Roger Hayes. Specifying and testing software components using adl, 1994.

30 Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Trans. Softw. Eng.*, 22:777–793, November 1996.

31 Oded Tal, Scott Knight, and Tom Dean. Syntax-based vulnerability testing of frame-based network protocols. In *PST'04*, pages 155–160, 2004.

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

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

34 Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. Pathcrawler: automatic generation of path tests by combining static and dynamic analysis. In *In: Proc. European Dependable Computing Conference. Volume 3463 of LNCS (2005) 281–292*, pages 281–292. Springer. ISBN, 2005.

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

36 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. *SIGPLAN Not.*, 46:283–294, June 2011.