

Université de la Réunion

UFR Sciences et Technologies

Laboratoire d'Informatique et de Mathématiques

Rapport de stage de master

M2 Informatique

2019 - 2020

Conception, implantation et application d'un outil de test concolique pour CHC/PLC

Auteur : Jean-Emile PELLIER
N° étudiant : 34003552

Encadrants : Pr. Frédéric MESNARD
Pr. Étienne PAYET

Responsable du stage : Pr. Frédéric MESNARD

Période du stage : du 2 janvier au 30 juin 2020

Version du 15 juin 2020

Acknowledgements

Before starting, it should be remembered that a substantial part of the presented work has to be done in the difficult context of the COVID-19 pandemic.
Although the overall quality of my work appears not to be affected by the inherent perturbations, it seems necessary for me to thank all the people who made that possible.
I guess I could not be exhaustive, even if I wanted to be, because they were numerous to have a significant impact on my ability to work both efficiently and serenely.

That is why I will content to be thankful mostly towards the people who are closest to me.
Firstly, I would thank all the members of my family, with a special thought for my mother, for their financial support but also their confidence in my success.
Secondly, I would thank my friends, wherever they are through the world, for their communicative cheerfulness even if the interactions with them are really very rare.
Finally, I would also thank my supervisors Pr. Frédéric MESNARD and Pr. Étienne PAYET for their remarkable efficiency on supervising the stage despite of their respective obligations which were exacerbated because of the health crisis.

Abstract

Concolic testing is a well-known technique to effectively identify program malfunctions. It has been very studied for many years because of its remarkable practical performances. Nevertheless, the main approaches concentrate on imperative programming whilst declarative programming seems to have been neglected. It is a shame because declarative languages are often really easy to parse which makes their semantics easier to be fully considered.
Prolog is an example of declarative language which promotes the use of the logic paradigm. This document has for purpose to present an improvement of a recent concolic testing tool especially designed to fit the needs of logic programming.

Keywords: concolic execution, symbolic execution, software testing, logic programming

---

Résumé

Le test concolique est une technique très populaire permettant d'identifier efficacement certains des dysfonctionnements présents à l'intérieur d'un programme.
Elle a été très étudiée ces dernières années du fait de ses performances pratiques remarquables. Toutefois, les principales approches se concentrent sur la programmation impérative tandis que la programmation déclarative semble avoir été négligée. C'est fâcheux car, les langages déclaratifs étant plus facile à parser, ils disposent d'une sémantique plus simple à manipuler.
Prolog est un exemple de langage déclaratif qui met en avant l'usage du paradigme logique.
Ce document a pour objectif de présenter une amélioration d'un outil de test concolique récent spécialement conçu pour correspondre aux besoins de la programmation logique.

Mots-clés : exécution concolique, exécution symbolique, test logiciel, programmation logique

Introduction

Any developer can make mistakes and the consequences of these can be quite serious. Sometimes it comes from a lack of mastery but more often it comes from a lack of attention. Whether it can be an error in the reasoning or in its implementation, it can lead to an unexpected behaviour or even to a program crash and these situations are never acceptable.

Is there a way to avoid these annoying mistakes?

Fortunately, highly qualified people asked themselves that question and, over the years, they imagined many techniques and tools to help developers to make safer programs.

The safest way to ensure that a program behaves as it should is to prove its correctness [1]. Nevertheless, it can be excessively difficult to check every components of a program by this way. Moreover, the Rice's theorem [2] guarantees the nonexistence of a fully automatic demonstration method because such a method would necessarily imply undecidable properties in its reasoning.

That is why another less safe but more reasonable approach is generally preferred: tests. However, while tests can show efficiently the presence of bugs, they cannot prove their absence. Furthermore, tests can sometimes be difficult to perform while they can often represent up to half of software development costs so they are rarely done properly and bugs appear after deployment.

Thus, it could be interesting to have automated testing tools to reduce the risks effortlessly.

That pleasant solution will be detailed through this document.

Contents

List of Figures

1 An Overview of Software Testing

Software testing [3] is a well-established domain so only a few notions are described here.

A test is a procedure to identify unexpected problematic behaviours in a software product. That behaviour must then be fixed by the developer in order to improve the quality of the software. The procedure is notably important to ensure a software product is stable enough to be deployed.

A test consists in running a program with a selected set of test cases to watch its behaviour. An entity called oracle monitors the execution to check conformity with the given specifications. The oracle [4] can be automatic through mechanisms like contracts or even manual through a user sufficiently motivated to carefully analyze itself the data flow.

An important limitation lies in the difference between defects and failures.
In fact, testing techniques can detect failures but some defects may not necessarily cause failures. In particular, defects in a dead code cannot lead to failures so tests will not be able to detect them.

In order to simplify the testing tasks, it can be interesting to reason with a modular approach. Unit testing consists in splitting a program into units (function blocks) to test them separately. That strategy helps on finding errors in components logic as well as on checking the difficult cases. However, to effectively test the behaviours of a program, in addition to the program itself some extra elements must be written: a test driver, an exploitation code simulating the component environment and finally a code for the functional correctness.
Moreover, well-chosen input test values must be generated then retransmitted to the unit.
That procedure is notably laborious and gives no guarantees on finding every possible behaviours. We will see later that some tools precisely propose an automation of that procedure to simplify it.

A fundamental problem to consider is that testing all possible inputs is generally infeasible. Unfortunately, testing can only establish a proper functioning under specific conditions not generic. The challenge is to choose exclusively test cases that will highlight distinct behaviours in a program. Many code coverage criteria exist in order to evaluate the relevance of a selected set of test cases. Among these multiple criteria [5], here are some of them:

— statements/nodes coverage: the set of test cases cover at least once each instruction
— decisions/branches coverage: the set of test cases cover at least once each option
— paths coverage: the set of test cases cover at least once each path through the graph

Thus, a testing technique takes one criteria and uses it to generate a small test cases sequence maximizing the code coverage to check as much behaviour as possible with minimal effort.

Figure 1 – Test Activities

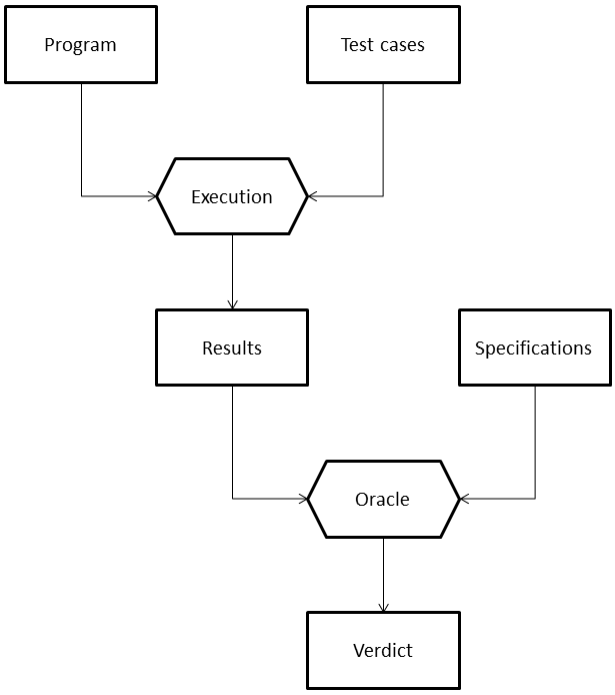
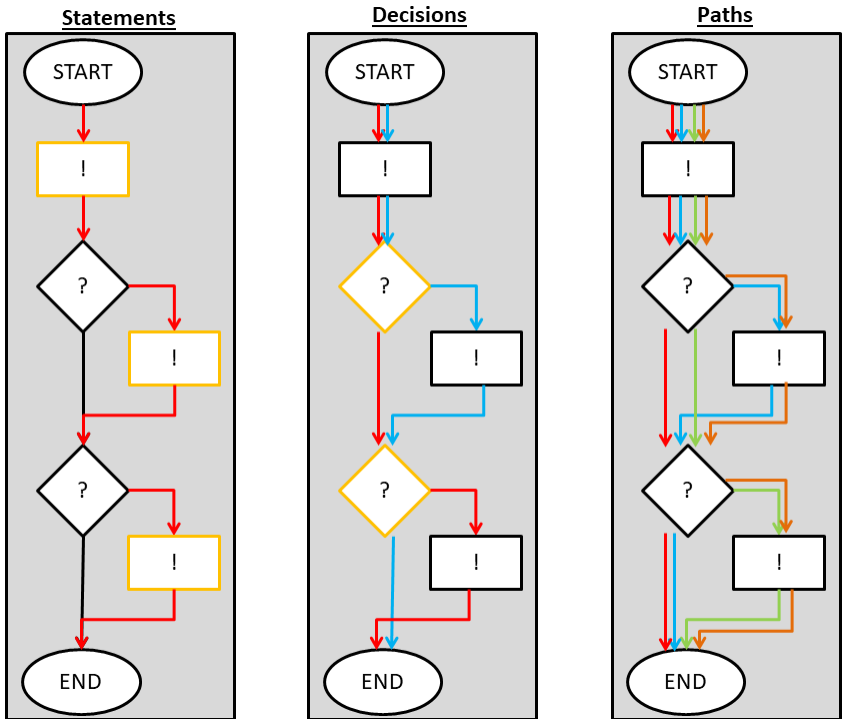


Figure 2 – Coverage Criteria



We will focus on three well-known testing methods: concrete, symbolic, concolic. To detail their functioning, we will consider the following function f written in C:

```
long int f(long int x)
{
    if (x < 0) {
        /* -- block 1 -- */

        return -x;
    }
    else if (x % 2 == 0) {       // x is even
        /* -- block 2 -- */

        if (x == 2020) {
           /* -- block 2.1 -- */

           assert(0); // assertion failure!
        }
        else {
            /* -- block 2.2 -- */

            return x / 2;
        }
    }
    else {                       // x is odd
        /* -- block 3 -- */

        if (x == 2042) {
           /* -- block 3.1 -- */

           assert(0); // assertion failure!
        }
        else {
            /* -- block 3.2 -- */

            return x;
        }
    }
}
```

1.1 Concrete Testing

Principle
Concrete testing consists in running a program with arbitrary values to finally produce results. Since that method is intended to be fairly intuitive, the required values can be chosen randomly. The results are then checked by an automated oracle or a user to know if there is a bug or not.

Algorithm
Assume you have selected a function and you want to test it, you must follow these steps:

1. choose an arbitrary value for each argument
2. call dynamically the function with these arguments
3. ask an oracle if a bug can be found or not
4. quit or go back to step 1

Example
In order to apply the algorithm to the function f, arbitrary values must be chosen for its argument. Let's choose 5 random values: $x_0 = -2171$, $x_1 = -854$, $x_2 = 4602$, $x_3 = 4610$, $x_4 = 6537$.
The following table summarizes the concrete executions for each of the selected values.

index	x	f(x)	bug found?	return block
0	-2171	2171	NO	1
1	-854	854	NO	1
2	4602	2301	NO	2.2
3	4610	2305	NO	2.2
4	6537	6537	NO	3.2

The block 2.1 contains an error but, as the table shows, the selected test cases did not find it. In fact, assuming $sizeof(long) = 4$, the probability to reach randomly the block 2.1 is given by the formula $P = 2^{-8 \times sizeof(long)}$ so this represents 1 chance out of 4,294,967,296.

Remarks
The main advantage of concrete testing is its simplicity: easy to understand and to manipulate. However, that method is terribly ineffective in practise even on very simple examples.
In fact, random selection of test cases implies excessively poor code coverage though behavioural redundancy and very low probability to reach some blocks.

Nevertheless, the approach can be used to guide our intuition early in a software development.

1.2 Symbolic Testing

Principle
Symbolic testing [6] consists in running a program using symbols to finally produce expressions. The expressions can be solved in order to get concrete values leading to every reachable block. An oracle can then proceed to a bug checking through dynamic calls using these concrete values.

Algorithm
Assume you have selected a function and you want to test it, you must follow these steps:

1. select arbitrarily an instruction block within the function
2. find the path conditions leading to the block by tracking the conditional instructions
3. if there is another block to reach then go to step 1 else quit

Example
In order to apply the algorithm to the function f, blocks exploration order must be chosen. Let's simply choose the linear order: 1, 2, 2.1, 2.2, 3, 3.1, 3.2
The following table summarizes the induced symbolic execution.

index	block	path conditions	bug found?	solution
0	1	$x < 0$	NO	$x = -1$
1	2	$x \geq 0 \wedge is\_even(x)$	NO	$x = 0$
2	2.1	$x \geq 0 \wedge is\_even(x) \wedge x = 2020$	YES	$x = 2020$
3	2.2	$x \geq 0 \wedge is\_even(x) \wedge x \neq 2020$	NO	$x = 0$
4	3	$x \geq 0 \wedge \neg is\_even(x)$	NO	$x = 1$
5	3.1	$x \geq 0 \wedge \neg is\_even(x) \wedge x = 2042$	NO	$\varnothing$
6	3.2	$x \geq 0 \wedge \neg is\_even(x) \wedge x \neq 2042$	NO	$x = 1$

According to the table, no blocks has been forgotten even if the block 3.1 could not be explored. In particular, a value to reach to the block 2.1 has been generated and a bug has been identified. However, another similar bug remains undetected in the block 3.1 but anyway it is a dead code.

Remarks
The main advantage of symbolic testing is its ability to explore every reachable instruction blocks. For that reason, it benefits from excellent code coverage and it can find a large number of bugs. However, that strategy is quite complex and expensive: it can be difficult to implement properly while the numerous symbolic constraints produced during the execution can often be hard to solve.

Contrary to concrete testing, that approach seems to fit the testing needs of leading software.

1.3 Concolic Testing

Principle
Concolic testing [7] consists in running a program with symbolically chosen values to make results. It must be seen as a hybrid method: "concolic" is a concatenation of "concrete" and "symbolic". Contrary to concrete testing, only one arbitrary value is needed because the others are computed.

Algorithm
Assume you have selected a function and you want to test it, you must follow these steps:

1. choose an arbitrary value for each argument
2. call dynamically the function with these arguments
3. ask an oracle if a bug can be found or not
4. find the path conditions leading to the reached block by tracking the conditional instructions
5. if it exists a last path condition not already negated then negate it else quit
6. use a SMT solver on the path constraints to get new values satisfying the constraints
7. if the SMT solver cannot produce argument values then go to step 5 else go to step 2

Example
In order to apply the algorithm to the function f, an initial value must be chosen for its argument. Let's choose randomly the initial value: $x_0 = 123$.
The following table summarizes the induced concolic execution.

index	old path constraints	solution	block	new path constraints	bug found?
0	$\top$	$x = 123$	3.2	$x \geq 0 \wedge \neg is\_even(x) \wedge x \neq 2042$	NO
1	$x \geq 0 \wedge \neg is\_even(x) \wedge x = 2042$	$\varnothing$	-	-	NO
2	$x \geq 0 \wedge is\_even(x)$	$x = 100$	2.2	$x \geq 0 \wedge is\_even(x) \wedge x \neq 2020$	NO
3	$x \geq 0 \wedge is\_even(x) \wedge x = 2020$	$x = 2020$	2.1	$x \geq 0 \wedge is\_even(x) \wedge x = 2020$	YES
4	$x < 0$	$x = -1$	1	$\bot$	NO

As the table shows, the execution succeeded in checking all the blocks like the symbolic one. However, only the required path constraints have been generated during the concolic execution.

Remarks
Concolic testing cumulates the advantages of concrete testing and symbolic testing.
It benefits from excellent code coverage and limits the computational cost using explicit values. However, the reasoning remains complex:

— a nondeterministic behaviour will lead to an uncertain path
— an imprecise symbolic representation will induce an approximate reasoning
— a too complex constraint will imply a laborious or impossible resolution

Better than symbolic testing, that approach is more likely to be used on software testing tasks.

2 State of the Art

Concolic testing is a powerful testing technique that has been studied for a very long time. A founding paper published in 1976, J.C. King's "Symbolic Execution and Program Testing" [8], has laid the foundations of symbolic testing and has provided a first testing tool named EFFIGY.

Many years later, in 2005 two more articles came to enrich it by establishing concolic testing, P. Godefroid's "DART: Directly Automated Random Testing" [9] with a second tool named DART and K. Sen's "CUTE: A Concolic Unit Testing Engine for C" [10] using a third tool named CUTE.

Recently, concolic testing has become even more popular because of the appearing of constraint solvers called SMT solvers which are enough powerful yet to deal with these of a concolic execution.

2.1 Symbolic Execution and Program Testing

2.1.1 An Exhaustive Testing Approach

Several techniques exist to produce reliable programs even without using formal specifications. However, even if the works are promising, it is really difficult to move from theory to practice.

Proofs and tests are two extreme approaches: while the first require a rigorous and laborious global program analysis, the second is simpler because it only needs some well-chosen test samples. Unfortunately, tests can only check chosen samples but not the global program contrary to proofs.

Despite the intrinsic limitation, a vast majority of verification techniques are based on testing. The idea is to ensure valid results are produced when a program is called with expected inputs. Among the various existing criteria to select test samples, randomization is so by far the worst.

Symbolic testing propose an innovative intermediate approach between proofs and tests by generation of sample classes using the control flow information to differentiate them.
A symbolic execution is closely linked to normal program execution but it brings a significant advantage in test and debugging because it can represent infinity of execution classes.

2.1.2 Extensions to Deal with Symbols

Ordinary programming languages are not natively able to handle symbols as easily as values. It is necessary to think about some extensions in order to proceed to a symbolic execution.

An ideal symbolic execution can be characterized by:
— an arbitrary integer magnitude ignoring the machine word size restrictions (16/32/64 bits)
— an infinite execution tree so a program should never be hindered in its decision making
— an unlimited resolution ability which means any conditional instruction should be decidable

A programming language suitable for a symbolic execution must have an appropriate semantics for its functions to accept arbitrary symbols as inputs and formulas as outputs.

A symbolic execution succeed in exploring every paths using a notion of path conditions. Path conditions are a Boolean expression retracing the successive constraints leading to a block. In other words, path conditions simply are a Boolean conjunction of block conditional instructions.

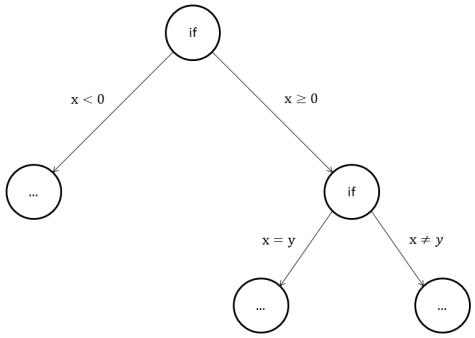
2.1.3 Symbolic Execution Tree

A symbolic execution can be succinctly visualized with a tree describing the path constraints. The tree is made of nodes representing the instructions and arcs representing the transitions. A symbolic execution tree has the following properties:

— it is a binary tree knowing every conditional structure can be translated into an if-else one
— a leaf necessarily describes a path reached by at least one concrete execution
— the terminal leaves has distinct path conditions

```
void f(int x, int y)
{
    if (x < 0) {
        ...
    }
    else {
        if (x == y) {
            ...
        }
        else {
            ...
        }
    }
}
```

Figure 3 – A Symbolic Execution Tree



2.1.4 EFFIGY

EFFIGY is a testing tool to perform symbolic executions in an interactive debugging system. It proposes basic features to easily test and debug any program written in a custom PL/I syntax. An exhaustive test manager is available to allow the user to explore each of the alternative paths. A program verifier can be used with user-defined assertions to generate verification conditions. A range of interactive debugging features is provided: tracing, breakpoint, state saving.

As part of EFFIGY, a concrete execution can be seen as a particular case of symbolic execution so, as well as symbolic calls, concrete calls or mixed ones are allowed too.

2.1.5 Program Correctness

In 1967, Robert W. Floyd has suggested a method to prove the correctness of a given program. That method consists in taking an input predicate and an output predicate then connects the two. The proof stems from the fact that the connection is established through a symbolic execution. In fact, if the input predicate is valid we must valid the output predicate following path conditions. For that reason, there is an obvious link between program correctness and symbolic execution. Moreover, where the method needs inductive predicates when an absolute correctness is impossible, symbolic execution needs them to deals with the lack of exhaustiveness in infinite execution trees.

2.1.6 Underlying Limitations

The ideal symbolic execution assumptions are unrealistic and pose some practical problems. There is notably a gap between predicate truth and an automatic prover ability to establish it. Moreover, there is not always an acceptable simplification in the context of the conflict between discrete arithmetic and infinite precision continuous real numbers.

EFFIGY is not able to deal with variable storage referencing because it is a complex problem.

2.2 DART: Directly Automated Random Testing

2.2.1 An Automated Testing Approach

DART is an automated testing tool that tries to explore dynamically alternative branches. It proposes, for C programs, an approach that combines symbolic testing with concrete testing.

DART works by performing a directed search through a program instrumented into RAM. Starting with a random input, it then computes a vector for the next execution resulting from the symbolic constraint resolution to enforce a new path exploration until there are no more.

DART deals with expressions whose symbolic variables are represented by memory addresses. An instrumented program handles memory using statements: conditional, assignation, abort, halt. As much as possible, DART will try to explore all execution paths through these mechanisms:
— it maintains symbolic memory linking between addresses and expressions
— it gets values using their addresses to evaluate expressions if necessary
— it reloads the same program many times with different inputs

From a little static analysis, DART can find dynamically initialized memory positions and identify interfaces from where a program gets its inputs: variables, external functions, etc.
DART distinguishes 3 kinds of functions that it can trivially call:
— program functions (internal)
— external functions (environment dependant)
— library functions (program driven)

DART generates a non-deterministic (random) test driver to simulate a generic environment. The driver is simply a C program which randomly initializes the instrumented program like that:
— a main function initializes external variables and toplevel function arguments
— the external functions are simulated by returning random values

Analyzing a program, DART can behave as follows:
— if the program throws an exception then an error has been found
— if DART affirms "bug found" then an input leads to abort
— if DART silently terminates then no input leads to abort and all paths have been checked
— DART does not terminate

2.2.2 Benefits

Even if DART a has limited completeness on linear constraints, it can reason on dynamic data. Thus, it can use concrete values to get around theorem prover limitations on symbolic values. Randomizing helps it where automatic reasoning is difficult or impossible.

Here are some examples of its capabilities:
— it can reason with the "malloc" function by randomizing its result
— it can find an input satisfying a strongly dynamic constraint like instance->member == 0
— it can find an input satisfying a non-linear constraint like x * x * x > 0

While a simple code inspection using static analysis can lead to warnings and false alarms, DART rather dynamically identifies all executions leading to sure errors.
Nevertheless, DART can be seen as a complement to a static analysis inspection because tests can be computationally expensive while the efficiency of dynamic test cases generation can be limited.

DART can test every C program without needing an extra test driver or an exploitation code.

2.3 CUTE: A Concolic Unit Testing Engine for C

2.3.1 An Improved Automated Testing Approach

Similar to DART, CUTE is an automated testing tool but focusing on dynamic data structures. It can be seen as an arithmetic and pointer constraint solver allowing generation of test inputs. Its solver is based on an optimization of lp_solve which is a linear arithmetic constraint solver.

CUTE was made for unit testing but a test unit can have many functions so it necessary ask the user to specify explicitly which one must be used to generate test inputs.
The function takes as input a memory graph summarizing every reachable memory destination. CUTE automatically generates a main function to initialize the previously selected function then the resulting program can finally be executed and instrumented into RAM.

CUTE instruments a program by running it concretely but also symbolically.
The symbolic execution follows the concrete execution path replacing by concrete values every symbolic expression the constraint solver is not able to deal with.
An instrumented program maintains 2 symbolic states at runtime:
— memory positions to symbolic arithmetic expressions
— memory positions to symbolic pointer expressions
Arithmetic expressions are linear combinations of symbolic variables and integer constants.
Pointer expressions differs from arithmetic ones since they do not allow the same operations.

Trying to solve path conditions, CUTE initializes each integer variable with a random value. When a pointer is required, CUTE preferentially choose a null pointer if possible or alternatively it instantiates a new pointer and recursively initializes member variables and pointers.

Because physical addresses changes from one execution to another, CUTE rather uses logical addresses to deal with pointer constraints in order to preserve the dynamic structures it handles. CUTE keeps a trace of the input graph as a logical input map linking logical addresses to primitive values: the map represent symbolically the input memory graph at the beginning of the execution. The logical input map is an integer sequence where an element can be a logical address or a value. It represents each of the inputs (including memory graphs) through a symbolic variable set.

CUTE performs the following tasks:
— it instruments the code under test in order to explore alternative paths
— it builds a logical input map for that code
— it iteratively executes the code as follows:

1. it builds from the map a concrete input graph and two symbolic states (pointer/primitive)
2. it runs the code on the graph collecting the input constraints leading to the same path
3. it negates one of the constraints and solves the constraint system to get another map

2.3.2 Benefits

Contrary to DART, CUTE can efficiently manipulates dynamic data structures while testing. It demonstrates that it is feasible and scalable to approximate a symbolic execution in order to test programs implying complex dynamic data structures.

CUTE has a mighty method to build dynamic structures by incremental node adding/removing.

3 SMT Solvers

The SMT (Satisfiability Modulo Theories) problem [11] is a decision problem which can be seen as a generalization of the well-known SAT problem: both consist in determining if there is a way to satisfy a Boolean formula by identifying assignations for each of its variables.

Solvers have been made to solve this kind of problems in a automated way.
While a SAT solver can only find suitable assignations for Boolean variables, a SMT solver can do much better by finding assignations for all kinds of variables with respect to background theories. Among these theories, there are integers, real numbers and even data structures like lists or arrays.

A SMT instance is a formula written using first-order logic where both function and predicate symbols can have additional interpretations corresponding to the underlying theories.
Internally, SMT solvers can cleverly translate a SMT instance into a SAT one to forward it to a SAT solver in order to benefits from optimized algorithms like DPLL during the resolution tasks.

Most of the common SMT approaches work well with decidable theories but some problems may involve undecidable theories so if a solution exists it does not necessary means it can be found.

Many powerful SMT solvers exist: CVC4, OpenSMT, VeriT, Yices, Microsoft's Z3, etc.
Beyond their singularities, most of them have a C/C++ API while they seems to unanimously consider the SMTLIB syntax as a common encoding format for solver-specific languages.

SMT solvers are very helpful in software verification requiring proofs of program correctness as well as in software testing in regards to the computational needs of symbolic executions.
An interesting verification technique consists in translating preconditions, postconditions, loop conditions, and assertions into SMT formulas in order to determine if all properties can hold.

3.1 An Introducing MOOC

An introducing MOOC entitled "Automated Reasoning: satisfiability" is accessible through the following link https://www.coursera.org/learn/automated-reasoning-sat.

After a quick reminder of certain fundamental notions of logic and software verification, it introduces an immediate concrete application of these notions: SMT solvers.
The course shows how to use satisfiability (SAT/SMT) verification tools to solve a wide range of practical problems and, while some examples illustrates the importance of these tools, some of the background theories are described to clarify their underlying functioning.

That course invites us to experiment by ourselves on several examples so it can be a good way to understand the functioning of SMT solvers and to learn to use them properly.

3.2 Microsoft's Z3

Z3 Theorem Prover [12] is an efficient cross-platform SMT solver designed by Microsoft.
Z3 can handle a wide range of theories: (linear/nonlinear) arithmetic, fixed-size bit-vectors, datatypes, extensional arrays, sequences, strings, uninterpreted functions, quantifiers, etc.
The user can communicate with Z3 using assertions written in a Z3-specific variant of SMTLIB2.
Several well-chosen simple Z3 script examples are given in annex 1.

Many verifiers use Z3 as a component to check assertions like Dafny via contract programming.

4 Logic Programming with Prolog

Prolog [13] is an interpreted programming language implementing the logic paradigm [14].
Based on formal logic, it intends to express programs with relations using facts and rules.
Interpreters have an internal mechanism called the SLD resolution which is in charge of running programs by establishing links between relations to determine the truth value of an initial query.
That deduction process makes the language natively suitable for tasks requiring expert systems.

Among the other programming languages, Prolog stands out with interesting characteristics:
— nondeterminism: an initial query can produce multiple solutions
— reversibility: a predicate argument can be indifferently seen as an input or as an output
— reflectivity: a Prolog program is able to alter the structure of the language itself

Many distinct Prolog implementations exist: GNU Prolog, SICStus Prolog, SWI-Prolog, etc.
Each of them can notably differ from another on built-in predicates as well as syntax or semantics.

4.1 An Unusual Paradigm

Every programming language is generally based on one or more programming paradigms.
These paradigms guide developers while writing code by adding constraints to its reasoning.

There are two big families of programming paradigms: imperative and declarative.
While with imperative languages a developer must specify precisely how its program must proceed, it only has to gives some declarations to guide an execution mechanism with declarative languages.

Logic programming can be seen as a subset of the declarative paradigm.
Its approach roughly consists in describing a logic reasoning to represent knowledge from raw data.
While terms allow labelling raw data, clauses allow expressing relations between these terms.
Comparatively to imperative programming, logic programming is demanding on the following:
— immutability: a variable can only be assigned once
— pattern-decisionality: a condition should be defined via pattern matching
— non-iterativity: a loop must be implemented recursively

4.2 Terms

Prolog is an untyped language so it considers a single generic data type named term.
However, Prolog can discriminate atoms, numbers, variables, compound terms, lists and strings.
Its permissive type system allows the user to build enumerations, nested lists or even exotic types.

4.3 Clauses

A Prolog program describes a logical reasoning implying terms by means of clauses.
While a rule is a clause which requires predicate calls to be validated, a fact is an always true rule.

4.4 SLD Resolution

In order to establish the truth value of an initial query while retrieving its potential solutions, Prolog has a resolution mechanism based on pattern matching during a depth-first tree traversal.
In fact, a tree is build when an initial query requires predicate calls which again imply other calls.
While calling, an unification is performed by pattern matching on the given predicate arguments to identify a corresponding clause and, because the pattern matching does not necessarily select a single clause, a backtracking can be performed to retrieve all the possible solutions.

5 A Concolic Testing Tool for Logic Programs

Concolic testing has been very studied for years in scientific literature with imperative programs whilst it has been less explored for declarative programs despite their often simpler semantics. Indeed, because an execution mechanism takes care of the complex tasks, a declarative language can have a very basic but powerful semantics which helps on designing suitable testing tools.

A Belgian student named Sophie Fortz had recently tried to design a concolic testing tool specifically for logic programming by implementing the algorithm with SWI-Prolog and Z3. Through her master thesis [15], she had carefully detailed her globally functional approach.

5.1 A Specific Concolic Algorithm

Logic programming has specificities requiring some adjustments in its concolic algorithm. In particular, Sophie Fortz had notably mentioned a collaborative work on:

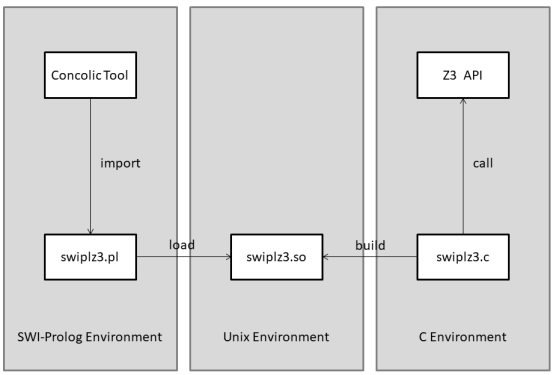
— choice coverage: a variant of path coverage specific to nondeterministic programming
— selective unification problem: a fundamental problem inherent in the SLD resolution
— concolic semantics: an execution semantics fully adapted to logic programming

These various works had finally led to an implementable specific concolic testing procedure.

5.2 An Interface between SWI-Prolog and Z3

In order to use Z3 in combination with SWI-Prolog, an interface has to be implemented. It turns out that SWI-Prolog has a foreign language interface with C while Z3 has a C API so it is eventually possible to build a bridge between SWI-Prolog and Z3 through C.
However, such a bridge only allows SWI-Prolog to use C primitives mirroring some Z3 features. The following diagram depicts the functioning of the induced swiplz3 module:

Figure 4 – The Swiplz3 Module



According to concolic testing requirements, Sophie Fortz's primitives provide formula checking and model retrieving but they also allow Prolog terms handling among other noticeable features.

5.3 A Promising Implementation

Sophie Fortz's tool is available there: https://github.com/sfortz/Pl_Concolic_Testing/
Although her concolic tool implementation should only be considered as proof of concept because of its inability to deal with benchmarks, it represents a promising outcome for logic programming.

6 An Improvement of the Existing Tool

Most of my work consisted in improving the previously described Sophie Fortz's concolic tool. Although the tool was well thought, it was incapable of handling a large majority of programs. An in-depth code analysis of the tool revealed some irregularities justifying unstable behaviours. I was able to fix some bugs by restructuring the tool and by implanting self-made diagnostic tools.

6.1 A Renewed Conception

Sophie Fortz's conception was not necessarily bad but it was really hard to understand and maintain so I rethought the conception for it to be more intuitive by modularizing it properly. Moreover, conception choices have been reviewed to relax the initial implementation constraints.

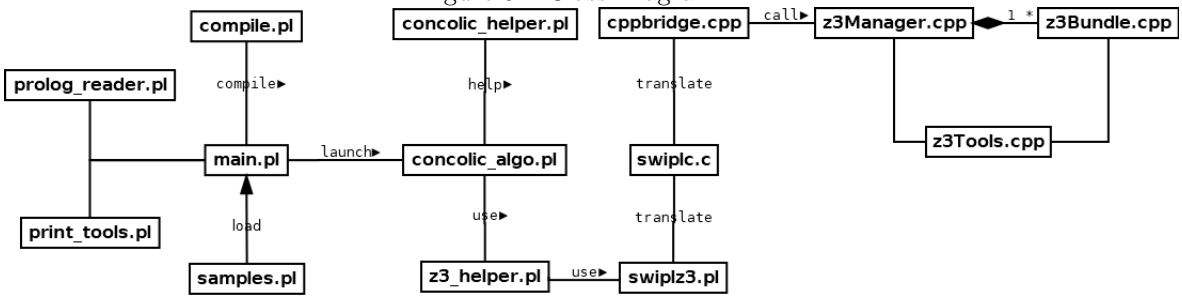
6.1.1 From C to C++

Originally, the use of C was motivated by building a bridge between SWI-Prolog and Z3. However, because of intrinsic C constraints, many global variables representing multidimensional raw arrays appeared in the C code so I replaced them all by class instance variables using C++. Because C++ is a superset of C with notably encapsulation capabilities, it was possible to clean the existing code while functionally preserving each of the already implemented features. Henceforth, C cohabits with C++ within the swiplz3 module through an internal syntactic bridge.

6.1.2 Another Design Model

The following class diagram summarizes as precisely as possible the newly chosen conception.

Figure 5 – Class Diagram



All the elements appearing in the diagram above are detailed below:
— main.pl: the concolic tool main entry point
— samples.pl: a loader to test the samples which are defined within the file "samples.txt"
— compile.pl: a compiler to transform the concolic tool to a platform-dependant executable
— concolic_algo.pl: the implementation of the concolic algorithm
— concolic_helper.pl: some tools to assist the concolic algorithm
— prolog_reader.pl: some tools to handle user inputs and files
— print_tools.pl: some tools to print pretty things
— z3_helper.pl: a specific intermediate for the concolic algorithm to use z3 primitives
— swiplz3.pl: a generic intermediate to call z3 primitives using SWI-Prolog
— swiplc.c: a bridge between C and SWI-Prolog through a foreign language interface
— cppbridge.cpp: a syntactic bridge between C and C++
— z3Bundle.cpp: a data bundle which wrap a Z3 context
— z3Manager.cpp: a singleton which manages Z3 bundles
— z3Tools.cpp: some generic utilities to deal with Z3 more easily

6.2 Diagnostic Tools to Improve Internal Stability

As we have seen before, our concolic testing tool requires a swiplz3 module to work with Z3. Thus, the project is made of two main components so it is preferable to test them separately. That is the reason why I decided to make diagnostic tools which focus on the swiplz3 module.

6.2.1 A Log File to Track Internal Interactions

Inside the swiplz3 module, I added a log file to chronologically track every primitive call. This means, each time SWI-Prolog interacts with Z3, interaction details appear in the file "swiplz3.log". That file is written using the following intuitive format:

```
primitive_function_name
  io_mode var_name: var_type = var_value
```

Importantly, consecutive loading of samples in a single concolic tool terminal is not recommended here because induced traces would accumulate in a same huge file without any delimitation. In fact, the file is based on a singleton life cycle while no primitive provides a delimitation signal.

The log file helps to find invalid primitive arguments and more specifically ill-formed assertions by providing a viewable representation of internal interactions between SWI-Prolog and Z3.

6.2.2 A Tester to Simulate Internal Interactions

Besides the log file, I made "swiplz3_tester.pl" to individually test each implemented feature. That tester is an interpreter which bypasses the concolic tool to directly interacts with swiplz3. That interpreter can translate a SMTLIB2 script file into Z3 instructions via swiplz3 primitives. Then, Z3 processes them and its results are displayed on the standard output.
Because the primitives are currently very limited, only the following instructions can be parsed:

- ($push) creates a new scope
- ($pop _) removes the last scope
- ($mk_int_vars variable_names...) declares multiple integer variables
- ($mk_term_type (term_names...) need_int need_list) defines the Term data type
- ($mk_term_vars variable_names...) declares multiple term variables
- ($assert_int_string formula) asserts a string formula implying integers
- ($assert_term_string formula need_int need_list) asserts a string formula implying terms
- ($check) checks if a model exist
- ($print_model) prints a model (if any)
- ($get_model_intvar_eval variable_names...) defines integer variables using a model
- ($get_model_termvar_eval variable_names...) defines term variables using a model
- (declare-const name Type) ≡ ($mk_int_vars name) or ($mk_term_vars name) or ø
- (push) ≡ ($push)
- (pop) ≡ ($pop _)
- (check-sat) ≡ ($check)
- (reset) resets the current Z3 context

Importantly, instructions starting with '$' correspond to the swiplz3 primitives so they cannot be recognized by any other SMTLIB2 parser unlike the other instructions which are standardized.

The project contains a few examples of parsable scripts which demonstrate the tester abilities.

Coupled with the log file, the swiplz3 tester helps not only while reproducing and fixing bugs within the existing swiplz3 primitives but also while implementing new experimental features.

7 Some Lines of Thought to Deepen

Because of a lack of time, the following elements have not been fully explored yet but they seem interesting and promising enough to be addressed in the context of subsequent works.

7.1 Constraint Logic Programming

Constraint logic programming [16] extends logic programming to face constraint satisfaction.
A constraint logic program allows constraints within its clauses in addition to predicate calls.
For example, a constraint could be a comparison of two arithmetic expressions implying integers.
A constraint belongs to a domain: $\mathcal{H}$(terms), $\mathcal{FD}$(bounded integers), $\mathcal{R}$(reals), $\mathcal{B}$(booleans), etc.
The domain defines which background theory must be used in order to solve the constraint.

When a constraint logic program runs, a resolution is performed similarly to a regular logic one, except the browsed constraints that are accumulated into a set called constraint store.
A backtracking will be enforced as soon as the constraint store is detected unsatisfiable.

7.1.1 A Meta-interpreter as a Constraint Translator

Because of its reflectivity, Prolog allows meta-interpreters to reason from any of its structures.
Thus, it should be possible to make a meta-interpreter which could translate every encountered Prolog constraint during the SLD resolution into a Z3 constraint through swiplz3 primitives.

Unfortunately, the idea is actually suspended because it requires time to be implemented.

7.2 Datatypes Management

Because Prolog is an untyped language, the concolic tool only defines a Term type within Z3.
However, some Prolog variants has types which are defined via the Hindley-Milner type system [17].
Thus, there is a basement to consider a concolic tool extension to deal with more elaborated types.

7.2.1 Typed-Prolog

Typed-Prolog is a library for Prolog which allows type declarations through code annotations.

I typed benchmarks using it: https://github.com/34003552/Typed_Prolog_Benchmarks
A manual attempt of defining suitable Z3 types for the file "deriv.pl" is given in annex 2.
Despite this promising attempt, there is still a lot to do before automating Z3 type generation.

7.3 Mercury

Mercury [18] is a logic programming language similar to Prolog but it is much more efficient.
Among its interesting specificities, there are strong types, strong modes and a determinism system.
Mercury does not allow extra-logical Prolog statements such as !(cut) and imperative input/output.

Despite its notable characteristics, the Mercury's syntax remind a lot that of Typed-Prolog.
Thus, once the concolic testing tool will be able to deal with Typed-Prolog, it could be interesting to consider Mercury with the aim of extending compatibility to other logic programming languages.

7.4 Concolic Tool Scope Widening

Although our concolic tool has been designed exclusively for logic programming, it could be interesting to think about a translation method to extend its compatibility to other paradigms.
In fact, even if it already exists similar tools for imperative programs notably, reasoning from a logical representation whose the semantics is simpler could interestingly improve performances.

Conclusion

Because of awkward bugs, scientific literature interested in software testing since a long time. Several testing approaches have been explored but only a few of them showed interesting results. While concrete testing has been proven ineffective, symbolic testing has shown promising results.

Over the years, concolic testing has demonstrated its efficiency with imperative programs. That powerful testing method cleverly combines concrete and symbolic testing in order to obtain excellent code coverage with minimal effort via an automated generation of relevant test cases.

Many concolic testing tools like DART or CUTE have been implemented then improved to fit as precisely as possible the specific semantics of different popular programming languages. Since the arrival of SMT solvers, these tools has significantly become even more potent.

Logic programming had been forgotten but Prolog now has its dedicated concolic testing tool. Because Prolog is a declarative language, it has unique characteristics which required adjustments. While researchers worked on its specificities, Sophie Fortz implemented it and I improved it. The current version of the improved concolic testing tool should be available on GitHub soon.

Although the presented concolic tool is not scalable yet, it seems to have a promising future. Indeed, it is only a question of time before the suggested improvement ideas to be implemented.

Annex 1: Z3 Script Examples

Z3 can be used as a SAT solver:

```
(declare-fun a () Bool)
(declare-fun b () Bool)
(assert (= (not (and a b)) (or (not a) (not b))))
(check-sat)
```

The above script would like to know if it exists two Booleans a and b satisfying $\neg(a \wedge b) \equiv (\neg a \vee \neg b)$

```
sat
```

Z3 answers "sat" so it means it exists two Booleans a and b satisfying $\neg(a \wedge b) \equiv (\neg a \vee \neg b)$

Z3 can be used as an equation solver:

```
(declare-const a Int)
(declare-const b Int)
(assert (= (+ a b) 20))
(assert (= (+ a (* 2 b)) 10))
(check-sat)
(get-model)
```

The above script would like to know the value of two integers a and b satisfying $\begin{cases} a + b = 20 \\ a + 2 \times b = 10 \end{cases}$

```
sat
(model
  (define-fun b () Int -10)
  (define-fun a () Int 30)
)
```

Z3 answers "sat" and gives a model suggesting that (a, b) = (30, -10) satisfies $\begin{cases} a + b = 20 \\ a + 2 \times b = 10 \end{cases}$

Annex 2: A type translation from the Typed-Prolog file "deriv.pl" into a Z3 script

```
(declare-datatypes ()
    (
        ; ------------ generic term type ------------
        (Term
            ; generic constructors
            (term_from_int (term_as_int Int))
            (term_from_list (term_as_list (List Term))) ; unused!

            ; user-defined predicates
            (expr_var (expr_var_arg_0 ExprVar))
            (data (data_arg_0 Empty))
            (benchmark (benchmark_arg_0 Empty) (benchmark_arg_1 QuadExpr))
            (ops8 (ops8_arg_0 Expr))
            (divide10 (divide10_arg_0 Expr))
            (log10 (log10_arg_0 Expr))
            (times10 (times10_arg_0 Expr))
            (d (d_arg_0 Expr) (d_arg_1 ExprVar) (d_arg_2 Expr))
        )

        ; ------------ user-defined types ------------

        ; :- type empty.
        (Empty)

        ; :- type expr_var ---> x ; y ; z.
        (ExprVar x y z)

        ; :- type expr ---> i(integer) ; v(expr_var) ; expr + expr ; expr - expr ;
                            expr * expr ; expr / expr ; ^(expr, integer) ; -expr ;
                            exp(expr) ; log(expr).
        (Expr
            (i (i_arg_0 Int))
            (v (v_arg_0 ExprVar))
            (+ (+_arg_0 Expr) (+_arg_1 Expr))
            (- (-_arg_0 Expr) (-_arg_1 Expr))
            (* (*_arg_0 Expr) (*_arg_1 Expr))
            (/ (/_arg_0 Expr) (/_arg_1 Expr))
            (^ (^_arg_0 Expr) (^_arg_1 Int))
            (- (-_arg_0 Expr))
            (exp (exp_arg_0 Expr))
            (log (log_arg_0 Expr))
        )

        ; :- type quad(T0,T1,T2,T3) ---> quad(T0,T1,T2,T3).
        ; :- type quad_expr ---> quad(expr, expr, expr, expr).
        (QuadExpr
            (quad (quad_arg_0 Expr) (quad_arg_1 Expr) (quad_arg_2 Expr) (quad_arg_3 Expr))
        )
    )
)
```

References

[1] Correctness. https://en.wikipedia.org/wiki/Correctness_(computer_science) .

[2] Rice's theorem. https://en.wikipedia.org/wiki/Rice%27s_theorem .

[3] Software testing. https://en.wikipedia.org/wiki/Software_testing .

[4] Test oracle. https://en.wikipedia.org/wiki/Test_oracle .

[5] Code coverage. https://en.wikipedia.org/wiki/Code_coverage .

[6] Symbolic execution. https://en.wikipedia.org/wiki/Symbolic_execution .

[7] Concolic testing. https://en.wikipedia.org/wiki/Concolic_testing .

[8] J. c. king's paper. https://yurichev.com/mirrors/king76symbolicexecution.pdf .

[9] P. godefroid's "dart". https://web.eecs.umich.edu/~weimerw/2011-6610/reading/p213-godefroid.pdf .

[10] K. sen's "cute". http://mir.cs.illinois.edu/marinov/publications/SenETAL05CUTE.pdf .

[11] Satisfiability modulo theories. https://en.wikipedia.org/wiki/Satisfiability_modulo_theories .

[12] Z3 theorem prover. https://en.wikipedia.org/wiki/Z3_Theorem_Prover .

[13] Prolog. https://en.wikipedia.org/wiki/Prolog .

[14] Logic programming. https://en.wikipedia.org/wiki/Logic_programming .

[15] S. fortz's thesis. https://researchportal.unamur.be/files/41563685/2019_FortzSophie_memoire_V2.pdf .

[16] Constraint logic programming. https://en.wikipedia.org/wiki/Constraint_logic_programming .

[17] Hindley-milner type system. https://en.wikipedia.org/wiki/Hindley%E2%80%93Milner_type_system .

[18] Mercury. https://en.wikipedia.org/wiki/Mercury_(programming_language) .