Automatic Test Generation for Object Oriented software

Thomas BRIEN

Sopra Banking Software - R&D

June 16^{th}, 2013

Tutor(s)

Arnaud MAKALA

Résumé

La complexité des systèmes informatiques devient de plus en plus importante, de même que les moyens qui doivent être mis en œuvre pour valider le bon fonctionnement des ces systèmes. Nous nous intéressons ici à l'automatisation de la génération des cas de tests pour les programmes orientés objets. Le travail effectué concerne les programmes java, mais les principes exposés s'appliquent à tout programme orienté objet.
Il existe déjà des outils de génération de tests tels que *Pex* et *Sage*, deux systèmes basés sur du *"fuzzing"* sur boite blanche. Le travail qui suit est un état de l'art des techniques existantes et une proposition de théorie/prototype basé sur les algorithmes d'optimisation sous contraintes et les solveurs SMT.
L'objectif est de proposer une méthode de génération des tests système, sans spécification. C'est-à-dire que l'on veut générer une suite de tests qui assure une non régression du système.
On se penche sur la faisabilité d'un pré-traitement du code permettant d'identifier les chemins d'exécution non définis afin de limiter au maximum l'utilisation du "fuzzing" ou tout autre algorithme de recherche à tâton.
Dans un second temps, on cherche à déduire de ce modèle de données les spécifications de chaque composants (objets). On cherche aussi à montrer qu'il est possible d'appliquer la théorie **ioco** sans introduction de spécification, en raisonnant sur l'environnement de chaque objet.

Mots-clés : génération de tests, génération de données initiales, LTS, ioco, exploration de chemins, solveurs SMT.

Abstract

The complexity of software systems have considerably increased in the past decades, along with the means we must use to validate those software. We focus on the automation of test case generation for object oriented software. Our work is based on java code only, but principles are the same for every object oriented program.
Some tools already handle the test case generation, such as *Pex* or *Sage*, two tools based on *white box fuzzing*. In this paper, we establish a state of the art of test automation techniques and propose a theory/prototype based on constraint optimisation algorithms and SMT solvers.
The following work consists in setting up test suites at the system level, without specification. In other words, we want to generate a test suite that ensures the non-regression of the system.
We work on the feasibility of code pre-treatment allowing a direct identification of undefined paths in order to limit any "fuzzing" or recursive algorithm.
In a second time, we try to deduce from our data structure the specifications for every components (Objects) of the system. We try to find a way to apply the **ioco** theory, with no introduced specifications, based on the environment of every object.

Key-words : test generation, input data generation, LTS, ioco, path exploration, SMT solvers.

Contents

Introduction **5**

State of the art **6**
Common tools and notions 6
Unit tests with JUnit 6
Integration and system tests 6
Mutation testing 6
Test automation 7
Software systems as LTSs 8
Conformance testing 10

Main ideas **11**
Model of the system 11
Generation of initial test data 12
Handle undefined functions 13
extracting test suites with our model 13

Forthcoming work **15**

References **16**
BDD 16
Mutation testing 16
Automation 16
SMT Solvers 16

Introduction

Testing often represents almost 50% of the total costs in software development. The current methodologies, such as *test driven development*, recommend the development of tests as early as possible in the development process. Those techniques have been proved to produce test suits with considerable reduction of the costs, but cannot ensure that the program under test can be trusted in any way. Moreover, testing is a fastidious and repetitive part of the development process. Tests can be build with different objectives: prove that a piece of software is compatible to a given specification, detect bugs, ensure a non-regression of the code during its evolution.

Contribution :
This paper proposes a model for white box testing of object oriented software. We try to built a data structure representing an object oriented software, from which we can efficiently produce a test suite for one or more of its components. The main idea is to use the environment of every object as a specification.

State of the art

Common tools and notions

Unit tests with JUnit

Unit tests are designed to prove the validity of an object.
In a nutshell, apply the process p to an input set A must produce B:

$$A \xrightarrow{p} B$$

A test case can be easily set up by a human, but generate it automatically raises three issues:

- find a correct input
- determine which sets of process to run
- predict the output

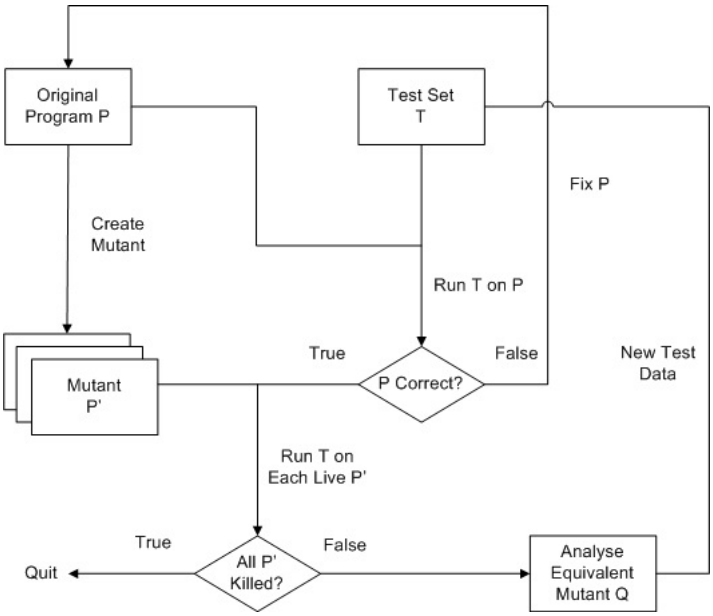
The quality of a test suite is measured in term of coverage:
a set of unit test should cover 100% of the code. It means the programmer in charge of setting up the tests must choose input values that will lead to every possible computation of the *system under test* (SUT).

Integration and system tests

Integration tests aim at validate the behaviour of two subsystems working together. For instance, a part of a program that uses a database. System tests are like master Integration tests, that validates the behaviour of the whole system, where every part communicates with others.

Mutation testing

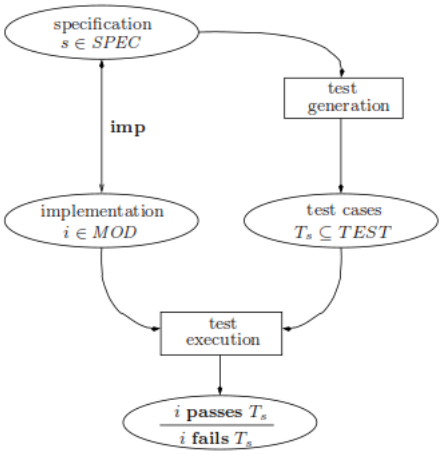
Mutation testing is a method that measures the "quality" of a test suite.
As detailed in [2.1], the method consists in introducing small modification in the system (ex: switch a "+" into a "-") and run the test suite on the modified system, also called the mutant. If the test suite raises an failure, the mutant is "killed".



Of course, a test suite should test every possible action of the system and thus, should kill 100% of mutants. The main issue of this method is the time it consumes. A huge amount of mutant is generated, and a compilation of the code for every possible mutants is most of the time out of question. Current research aims at reducing the amount of generated mutants without reducing the generality of the results.

Test automation

A reference in test automation is *model based testing*.
The starting point of this technique is a *model* of the desired behaviour of the *implementation under test* (IUT). The implementation and its model are confronted to generate a test suite. Then the implementation is run under the generated test suite. If the implementation passes every tests, it implies that it is conform to the model.



Note that the implementation and the model must have the same abstraction level to allow a confrontation.
It is, most of the time, convenient to represent a system as a *labelled transition system* (LTS).

Software systems as LTSs

A common theory for testing is based on *labelled transition systems*, the states of the system are represented by nodes, and the actions produced by or applied on the system are represented by labelled transitions.
An LTS can be formalized as follow:
p = (S, L, T, s_0)
with

- S : a set of states
- s_0 : the initial state
- L : a set of action labels
- $T \in S \times L \times S$: a transition relation

We want to defined basic relation on the path of those systems:

Definition: Let p be an LTS defined above, with q, $q' \in S$ and μ, $\mu_i \in L \cup \{\tau\}$:

$$q \xrightarrow{\mu} q' \quad \Leftrightarrow_{def} \quad (q, \mu, q') \in T$$
$$q \xrightarrow{\mu_1 \dots \mu_n} q' \quad \Leftrightarrow_{def} \quad \exists q_0, \dots, q_n : q = q_0 \xrightarrow{\mu_1} q_1 \xrightarrow{\mu_2} \dots \xrightarrow{\mu_n} q_n = q'$$
$$q \xrightarrow{\mu_1 \dots \mu_n} \quad \Leftrightarrow_{def} \quad \exists q' : q \xrightarrow{\mu_1 \dots \mu_n} q'$$

Definition: Let p be an LTS defined above, with q, $q' \in S$, $a, a_i \in L$, and $\sigma \in L^*$.

$q \overset{\epsilon}{\Rightarrow} q' \quad \Leftrightarrow_{def} \quad q = q'$ or $q \xrightarrow{\tau \ldots \tau} q'$

$q \overset{a}{\Rightarrow} q' \quad \Leftrightarrow_{def} \quad \exists q_1, q_2 : q \overset{\epsilon}{\Rightarrow} q_1 \xrightarrow{a} q_2 \overset{\epsilon}{\Rightarrow} q'$

$q \overset{a_1 \ldots a_n}{\Longrightarrow} q' \quad \Leftrightarrow_{def} \quad \exists q_0, \ldots, q_n : q = q_0 \overset{a_1}{\Longrightarrow} q_1 \overset{a_2}{\Longrightarrow} \ldots \overset{a_n}{\Longrightarrow} q_n = q'$

$q \overset{\sigma}{\Rightarrow} \quad \Leftrightarrow_{def} \quad \exists q' : q \overset{\sigma}{\Rightarrow} q'$

Definition: We also define the parallel synchronization of two LTSs by the operator $\| : LTS(L) \times LTS(L) \rightarrow LTS(L)$, which is defined by the following inference rules:

$$u \xrightarrow{a} u', p \not\xrightarrow{a} p', a \in L \quad \vdash \quad u \| p \xrightarrow{a} u' \| p$$
$$u \not\xrightarrow{a} u', p \xrightarrow{a} p', a \in L \quad \vdash \quad u \| p \xrightarrow{a} u \| p'$$
$$u \xrightarrow{a} u', p \xrightarrow{a} p', a \in L \quad \vdash \quad u \| p \xrightarrow{a} u' \| p'$$

The previous definitions are the base of LTS logic, and help reasoning on labelled path.
The parallel synchronization is an operation used for test runs, at the end of the testing process.

The ioco relation (cf.[3.1]) :

The **ioco** relation, or **ioco** theory, is a relation that check the conformance of an implementation, modelled by an $ioTS$, to a specification, modelled by an LTS. The following definition and notations are a brief presentation of the **ioco** theory, containing the minimum information we need to write: i **ioco** s

Definition: Let p be a state in a transition system, and let P be a set of states, then

$init(p) =_{def} \{\mu \in L \cup \{\tau\} \mid p \xrightarrow{\mu}\}$

$traces(p) =_{def} \{\sigma \in L^* \mid p \overset{\sigma}{\Rightarrow}\}$

p **after** $\sigma =_{def} \{p' \mid p \overset{\sigma}{\Rightarrow} p'\}$

$out(p) =_{def} \{x \in L_U \mid p \xrightarrow{x}\} \cup \{\delta \mid \delta(p)\}$

$out(P) =_{def} \bigcup \{out(p) \mid p \in P\}$

Definition: Straces(p) are the *suspension traces* of the process p

$L_\delta =_{def} L \cup \{\delta\}$

$p_\delta =_{def} \langle Q, L_I, L_U \cup \{\delta\}, T \cup T_\delta, q_0 \rangle$

Straces(p) $=_{def} \{ \sigma \in L_\delta^* \mid p_\delta \overset{\sigma}{\Rightarrow} \}$

Definition: Let $i \in IOTS(L_I, L_U)$, $s \in LTS(L_I \cup L_U)$, then:
i **ioco** $s =_{def} \forall \sigma \in Straces(s) : out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma)$

Conformance testing

At this point, we have a specification, an implementation modelled by an LTS and a theory of conformance. We only need an algorithm capable of abstracting the specification into a set of test cases to decide weather our system is conform to its specification or not.

Definition:
(1) A *test case* t is a 5-tuple $\langle S, L, T, v, s_0 \rangle$, such that $\langle S, L, T, s_0 \rangle$ is a deterministic labelled transition system with finite behaviour, and v : $S \rightarrow \{\mathbf{fail}, \mathbf{pass}\}$ is a *verdict function*. The class of test cases over actions in L is denoted by $LTS_t(L)$. Definitions applicable to $LTS(L)$ are extended to $LTS_t(L)$ by defining them over the underlying labelled transition system.

(2) A *test suite* T is a set of test cases: $T \in P(LTS_t(L))$, where $p(LTS_t(L))$ is the powerset of $LTS_t(L)$, i.e., the set of all possible subsets of $LTS_t(L)$.

From now, we assume that we have an algorithm capable of generating test cases based on a specification, and a theory, or *implementation*. Such algorithm can be represented by the following function:

$$gen_{\mathbf{imp}} : LTS(L) \rightarrow P(LTS_t(L))$$

For instance, if we confront a specification $s \in LTS(L)$ to the *ioco* theory, we obtain a test suite $T = gen_{\mathbf{ioco}}(s)$

As detailed in [3.5], it is now easy to prove that an implementation i passes a test case t:
i **passes** $t =_{def} \forall \sigma \in L^* : t \| i$ **after** σ **deadlocks** *implies* v(t **after** σ) = **pass**

Main ideas

We want to apply a **ioco**-like theory to white-box testing. In this chapter, we consider systems without recursive or asynchronous functions.

Model of the system

We are reasoning at the function level.
Every function can be represented as a labelled transition system. The set of labels is $L = I \cup U \cup A \cup D$ where :

- I is a set of *inputs* label
- U is a set of *outputs* label
- A is a set of *actions* label
- F is a set of *formula* label

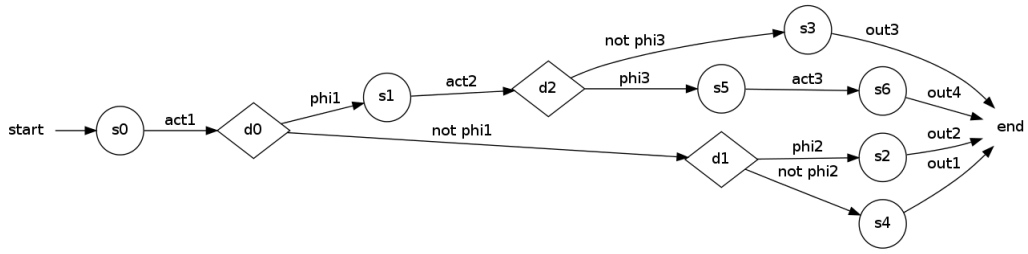
And the set of states is $S = C \cup D$
where C is the set of common states and D is the set of states preceding a *decision*. We call *parent* of a label $f \in F$ a node $d \in D$ such as:

$$\exists s \in S,\ s \in init(d)$$

The following graph represent a example function where:

- $I = \emptyset$
- $U = \{out1, out2, out3, out4\}$
- $A = \{act1, act2, act3\}$
- $F = \{phi1, phi2, phi3\}$

Figure 1: graph of a simple function



NB: The set of decisions label is associated with a set of state $\{d0, d1, d2\}$. Every transition from those states is a decision label i.e. **a first order formula**. We can also notice that decision states are the only multiple output states, and for a state d with n labels associated (every label corresponding to a FO formula ϕ_i, $i \in [0, n]$)

Lemma: 1 *For a given set of inputs, the system is deterministic:*
$\forall i, j \in [0, n], i \neq j \Rightarrow \neg(\phi_i \wedge \phi_j)$
(i.e. a given input set will activate a unique path at every run.)

Lemma: 2 *Every set of input leads to a computation:*
$\forall i \in [0, n]$

$$\neg(\bigvee_{\substack{j=0 \\ j \neq i}}^{n} \phi_j) \Rightarrow \phi_i$$

Rq: In our prototype, a decision node has two labels maximum. This is due to parsing issues of java code. Manage multiple decision node linked to each other is less complex than processing the code to obtain a single, node with multiple FO formulas.

Anyway, every possible trace is stored and associated with its condition of application. For every function p, we link a domain to a trace:
$\forall i \in Card(traces(p))$
$\phi_i \leftrightarrow trace_i$

Generation of initial test data

At first, we assume that our system is only composed of well defined functions. In other words, for a given set of inputs, the system is deterministic and the states of each variables can be traced.

A trace is accessible if we compute inputs from a specific domain. The program itself is a classification tool, and we need to test every class of inputs to

ensure the correctness of the system.

Generate a population of test objects respecting the classification condition is made by constraint programming. We use *choco*, a constraint programming library described in [4.2].

Previously, we have seen that every trace could be linked to a *first order formula* which variables can be mapped to a set of inputs. It is now possible to extract the conditions concerning an object, and set its attributes with *choco*.

Handle undefined functions

We call *undefined function* a function for which we cannot access the source code. Those functions can be seen as black boxes, with no link between the inputs and the outputs.
Let I be a set of input given to f, an undefined, but surjective, function.
Let O be the set of output returned from f for I.
Let S be the set of states of a process p containing the undefined function, and p, $p' \in S$ such as:
$p \xrightarrow{f} p'$
Let p_0 be the initial state and $p \stackrel{\sigma}{\Rightarrow} p'$. We must run an new study on $p_{|p'}$ the LTS p reduced to the states p **after** σ.
Another method would be to replace every input in I by an undefined function in the FO formulas of the decision states. Note that SMT solvers can handle undefined functions.

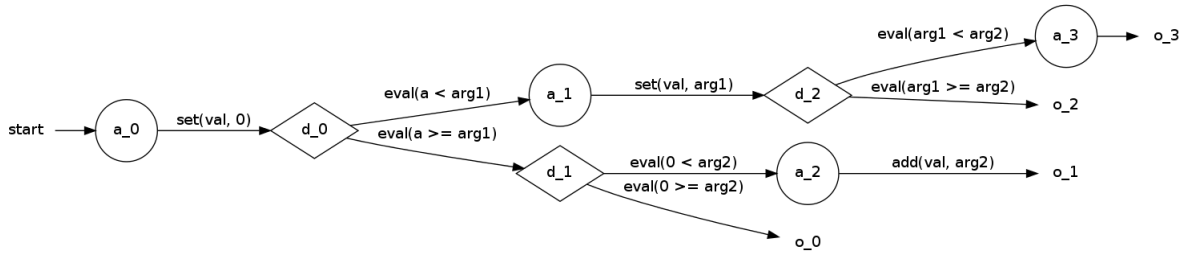
Extracting test suites with our model: example

Our analysis is based on the java source code.

The full process on a simple function belonging to a java class Example:

```
public int example(int arg1, int arg2){
                int val = 0;
                if(this.a<arg1){
                        val = arg1;
                }
                if(val<arg2){
                        val += arg2;
                }
                return val;
        }
```

From this function, we produce the following graph:



For every run, we store every condition with previous action impact taken to account. For instance, the run $a_0.d_0.d_1.a_2.o_1$ is linked to:
$\phi_1 := a >= arg1 \wedge arg2 > 0$
With our model, the constraint satisfier will generate data to satisfy any condition previously stored and run every possible path: ex:
Example ex = new Example(3);
ex.example(1, 2);
This code will activate the trace $a_0.d_0.d_1.a_2.o_1$

And so on until every trace is activated.

Forthcoming work

In its current state, our work doesn't handle recursive or asynchronous systems.

The case of undefined function will be solved by *fuzzing*, or data generation using *genetic algorithm*.
A study should be made to determine the less time consuming technique, but such experiences will require considerable programming efforts.

Moreover, we are not yet capable of generating specification automatically. A system of data-dependency analysis is currently in development, but not up and running by now.
After the specification generation, it would be interesting to focus on the adaptation of a *ioco-like* theory that would consider $LTSs$ with inputs, outputs **and variables**. Reducing the current white box abstraction to a classic $ioLTS$ is a loss of information.

Eventually, it would be interesting to focus on the feasibility of an automatic recognition of *input* and *output* functions.
For instance, our model will not be able to set up high quality *integration test suites*. We must introduce *write/read* notions, so that we can check push/pull functionalities between systems.

References

BDD

[1.1] *Testing by Contract - Combining Unit Testing and Design by Contract*
Per Madsen (madsen@cs.auc.dk) Institute of Computer Science, Aalborg University Fredrik Bajers Vej 7, DK-9220 Aalborg, Denmark

Mutation testing

[2.1] *Composants objets fiables : une approche pragmatique*
Daniel Deveaux* - Régis Fleurquin* - Patrice Frison* Jean-Marc Jézéquel** - Yves Le Traon**
* Laboratoire VALORIA (Aglae) UBS - IUP de Tohannic - Rue Mainguy 56000 VANNES
** IRISA-CNRS (Pampa) Université Rennes 1 - Campus de Beaulieu 35042 RENNES

Automation

[3.1] *Test Generation with Inputs, Outputs and Repetitive Quiescence*, Jan Tretmans - Tele-Informatics and Open Systems Group, Department of Computer Science - University of Twente.

[3.2] Bertrand Meyer, Ilinca Ciupa, Andreas Leitner and Lisa (Ling) Liu, *Automatic Testing of Object-Oriented Software*, in SOFSEM 2007

[3.3] Hans-Gerhard Gross and Arjan Seesing, *A Genetic Programming Approach to Automated Test Generation for Object Oriented Software*, Report TUD-SERG-2006-017

[3.4]*Higher-Order Test Generation* - Patrice Godefroid - Microsoft Research

[3.5]*Conformance testing with labelled transition systems: Implementation relations and test generation* - Jan Tretmans - Tele-Informatics and Open Systems Group, Department of Computer Science - University of Twente.

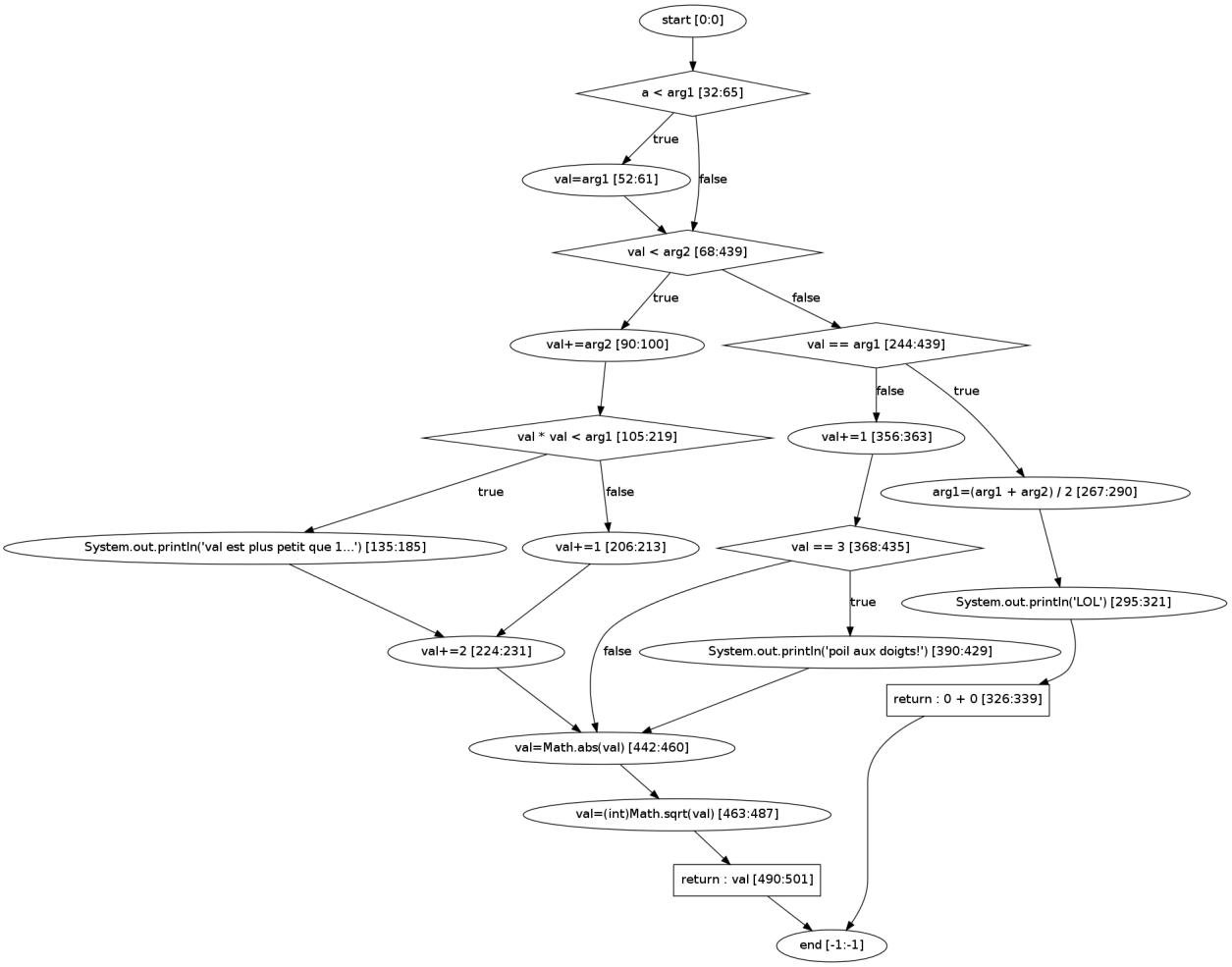
SMT Solvers and constraint satisfaction

[4.1] Yices - SMT solver (and smt-lib)

[4.2] *choco*: an Open Source Java Constraint Programming Library - Ecole des Mines de Nantes

ANNEXE

A graph automatically generated by our prototype from a java function:



This figure was generated from our function interpreter, using *graphviz*.