ICSTW 2017

Conference Sponsors

Platinum Sponsor

Gold Sponsors

Silver Sponsors

Bronze Sponsors

Copper Sponsors

2017 IEEE International Conference on Software Testing, Verification and Validation Workshops

Tokyo, Japan

13-17 March 2017

Conference Information

Papers By Session

The 12th Workshop on Testing: Academia-Industry Collaboration, Practice, and Research Techniques (TAIC PART 2017)

1st International Workshop on Testing Extra-Functional Properties and Quality Characteristics of Software Systems (ITEQS 2017)

Runtime Verification for Detecting Suspension Bugs in Multicore and Parallel Software
by Sara Abbaspour Asadollah, Daniel Sundmark, Hans Hansson

Generating Controllably Invalid and Atypical Inputs for Robustness Testing
by Simon Poulding, Robert Feldt

The 12th International Workshop on Mutation Analysis (Mutation 2017)

Message from the Mutation 2017 Chairs
by Jens Krinke, Nan Li, José Miguel Rojas

MutRex: A Mutation-Based Generator of Fault Detecting Strings for Regular Expressions
by Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene

Towards Security-Aware Mutation Testing
by Thomas Loise, Xavier Devroey, Gilles Perrouin, Mike Papadakis, Patrick Heymans

Speeding-Up Mutation Testing via Data Compression and State Infection
by Qianqian Zhu, Annibale Panichella, Andy Zaidman

Applying Mutation Analysis on Kernel Test Suites: An Experience Report
by Iftekhar Ahmed, Carlos Jensen, Alex Groce, Paul E. McKenney

Mutation Patterns for Temporal Requirements of Reactive Systems
by Mark Trakhtenbrot

How Good Are Your Types? Using Mutation Analysis to Evaluate the Effectiveness of Type Annotations
by Rahul Gopinath, Eric Walkingshaw

Reducing Mutants with Mutant Killable Precondition
by Chihiro Iida, Shingo Takada

Finding Redundancy in Web Mutation Operators
by Upsorn Praphamontripong, Jeff Offutt

An Architecture for the Development of Mutation Operators
by Macario Polo Usaola, Gonzalo Rojas, Isyed Rodríguez, Suilen Hernández

Are Deletion Mutants Easier to Identify Manually?
by Vinicius H. S. Durelli, Nilton M. De_Souza, Marcio E. Delamaro

6th International Workshop on Combinatorial Testing (IWCT 2017)

General Message from the IWCT Workshop Chairs
by Dimitris Simos, Rachel Tzoref-Brill

IWCT 2017 Organizers
by Dimitris Simos, Rachel Tzoref-Brill

Test Case Generation & Quality Assessment

A Model for T-Way Fault Profile Evolution during Testing
by D. Richard Kuhn, Raghu N. Kacker, Yu Lei

Mutation Score, Coverage, Model Inference: Quality Assessment for T-Way Combinatorial Test-Suites
by Hermann Felbinger, Franz Wotawa, Mihai Nica

Optimizing IPOG's Vertical Growth with Constraints Based on Hypergraph Coloring
by Feng Duan, Yu Lei, Linbin Yu, Raghu N. Kacker, D. Richard Kuhn

Test Case Generation with Regular Expressions and Combinatorial Techniques
by Macario Polo Usaola, Francisco Ruiz Romero, Rosana Rodríguez-Bobada Aranda, Ignacio García Rodríguez

Applications of Combinatorial Testing: I

Applying Combinatorial Testing to High-Speed Railway Track Circuit Receiver
by Chang Rao, Jin Guo, Nan Li, Yu Lei, Yadong Zhang, Yao Li, Yaxin Cao

Applications of Practical Combinatorial Testing Methods at Siemens Industry Inc., Building Technologies Division
by Murat Ozcan

Using Timed Base-Choice Coverage Criterion for Testing Industrial Control Software
by Henning Bergström, Eduard Paul Enoiu

Modelling

Building Combinatorial Test Input Model from Use Case Artefacts
by Preeti S., Milind B., Medhini S. Narayan, Krishnan Rangarajan

Combinatorial Methods for Modelling Composed Software Systems

by Ludwig Kampel, Bernhard Garn, Dimitris E. Simos

Combinatorial Interaction Testing for Automated Constraint Repair
by Angelo Gargantini, Justyna Petke, Marco Radavelli

A Composition-Based Method for Combinatorial Test Design
by Anna Zamansky, Amir Shwartz, Seri Khoury, Eitan Farchi

Applications of Combinatorial Testing: II

Applying Combinatorial Testing to Data Mining Algorithms
by Jaganmohan Chandrasekaran, Huadong Feng, Yu Lei, D. Richard Kuhn, Raghu Kacker

Combinatorial Testing on Implementations of HTML5 Support
by Xi Deng, Tianyong Wu, Jun Yan, Jian Zhang

Combinatorial Testing on MP3 for Audio Players
by Shaojiang Wang, Tianyong Wu, Yuan Yao, Beihong Jin, Liping Ding

Poster Session

Finding Minimum Locating Arrays Using a SAT Solver
by Tatsuya Konishi, Hideharu Kojima, Hiroyuki Nakagawa, Tatsuhiro Tsuchiya

Test Optimization Using Combinatorial Test Design: Real-World Experience in Deployment of Combinatorial Testing at Scale
by Saritha Route

4th International Workshop on Software Test Architecture (InSTA 2017)

Messages from the InSTA 2017 Chairs
by Satoshi Masuda

Research

Analysing Test Basis and Deriving Test Cases Based on Data Design Documents
by Tsuyoshi Yumoto, Tohru Matsuodani, Kazuhiko Tsuda

Improvement of Description for Reusable Test Type by Using Test Frame
by Keiji Uetsuki, Mitsuru Yamamoto

Emerging

Suggestion of Practical Quantification Measuring Method of Test Design Which Can Represent the Current Status
by Sunil Chon, Jihwan Park

Software Testing Design Techniques Used in Automated Vehicle Simulations
by Satoshi Masuda

Closing the Gap between Unit Test Code and Documentation
by Karsten Stöcker, Hironori Washizaki, Yoshiaki Fukazawa

Test Conglomeration - Proposal for Test Design Notation Like Class Diagram
by Noriyuki Mizuno, Makoto Nakakuki, Yoshinori Seino

Defining the Phrase "Software Test Architecture" Emerging Idea
by Jon D. Hagar

13th Workshop on Advances in Model Based Testing (A-MOST 2017)

Message from the A-MOST 2017 Chairs
by Paolo Arcaini, Xavier Devroey, Shuai Wang

Functional MBT

Mutation-Based Test-Case Generation with Ecdar
by Kim G. Larsen, Florian Lorber, Brian Nielsen, Ulrik M. Nyman

Reducing the Concretization Effort in FSM-Based Testing of Software Product Lines
by Vanderson Hafemann Fragal, Adenilso Simao, André Takeshi Endo, Mohammad Reza Mousavi

Property-Based Testing with External Test-Case Generators
by Bernhard K. Aichernig, Silvio Marcovic, Richard Schumi

Non-Functional MBT

Planning-Based Security Testing of the SSL/TLS Protocol
by Josip Bozic, Kristoffer Kleine, Dimitris E. Simos, Franz Wotawa

Towards Decentralized Conformance Checking in Model-Based Testing of Distributed Systems
by Bruno Miguel Carvalhido Lima, João Carlos Pascoal Faria

Pattern-Based Usability Testing
by Fernando Dias, Ana C. R. Paiva

10th IEEE International Conference on Software Testing, Verification and Validation - Posters Track (ICST 2017 Posters)

A Mechanism of Reliable and Standalone Script Generator on Android
by Kuei-Chun Liu, Yu-Yu Lai, Ching-Hong Wu

EarthCube Software Testing and Assessment Framework
by Emily Law

Using Model-Checking for Timing Verification in Industrial System Design
by Laurent Rioux, Rafik Henia, Nicolas Sordon

Challenges of Operationalizing Spectrum-Based Fault Localization from a Data-Centric Perspective
by Mojdeh Golagha, Alexander Pretschner

Towards a Gamified Equivalent Mutants Detection Platform
by Thomas Laurent, Laura Guillot, Motomichi Toyama, Ross Smith, Dan Bean, Anthony Ventresque

Cloud API Testing
by Junyi Wang, Xiaoying Bai, Haoran Ma, Linyi Li, Zhicheng Ji

Automated A/B Testing with Declarative Variability Expressions
by Keisuke Watanabe, Takuya Fukamachi, Naoyasu Ubayashi, Yasutaka Kamei

Weighting for Combinatorial Testing by Bayesian Inference
by Eun-Hye Choi, Tsuyoshi Fujiwara, Osamu Mizuno

Impact of Static and Dynamic Coverage on Test-Case Prioritization: An Empirical Study
by Jianyi Zhou, Dan Hao

BDTest, a System to Test Big Data Frameworks
by Alexandre Langeois, Eduardo Cunha De Almeida, Anthony Ventresque

What You See Is What You Test - Augmenting Software Testing with Computer Vision
by Rudolf Ramler, Thomas Ziebermayr

Framework for Model-Based Design and Verification of Human-in-the-Loop Cyber-Physical Systems
by Filip Cuckov, Grant Rudd, Liam Daly

Automated Test Case Generation from OTS/CafeOBJ Specifications by Specification Translation
by Ryusei Mori, Masaki Nakamura

An architecture for the development of mutation operators

Macario Polo Usaola
Dept. of Information Systems and Technologies
University of Castilla-La Mancha
Ciudad Real, Spain

Gonzalo Rojas, Isyed Rodríguez, Suilen Hernández
Dept. of Computer Science
University of Concepción
Concepción, Chile

***Abstract*— This paper introduces an abstract specification of mutation operators that (1) we have used to create traditional operators and, (2) we are currently using to define and implement mutation operators for context-aware, mobile applications that come from a list of common errors reported by three companies. This specification describes the structure and behavior of mutation operators at a high abstraction level, thus supporting the specification of new mutation operators according to the evolving state of the art of context-awareness. The paper also gives some notes about *BacterioWeb*, a web-based mutant tool with an execution engine for Android applications.**

***Keywords*— mutant generation; context-awareness; mutation operators design; mutation operators architecture.**

I. INTRODUCTION

Software Testing evolves as new technological features are incorporated by software systems. The rapid evolution of ubiquitous computing has extended the set of aspects for which quality must be assured. In particular, a relevant factor in developing mobile applications is their sensibility to changes in the context they are executed [1].

In this scenario, the adoption of Mutation Analysis increasingly demands the definition of specific mutation operators for these new features. It is desirable that these new operators can be added to the mutation environment for reproducing those new errors that appear over time.

The main contributions of this paper are: (1) the development of a hierarchical architecture for mutation operators that minimizes the dependence of external libraries and that facilitates the implementation of new operators; (2) *BacterioWeb*, a new mutation tool that performs all the mutation tasks in the web; (3) some mutation operators specifically designed for reproducing some common context-aware errors reported by mobile applications developers; (4) a comparison of the proposed architecture with those in other tools.

Migrating *Bacterio* to the web holds several important advantages: (1) projects can be shared amongst testers; (2) operators implemented by a developer are automatically available in all the testing projects; (3) for mobile testing, emultaro and devices are connected to the server and, thus, testers do not need to have all the target devices connected to their computers.

The remainder of this paper is structured as follows: Section II describes the background of our research; Section III introduces the proposed architecture, by describing the adopted mutation strategy and the main components of the mutation operators' architecture, including their structure and behavior; Section IV describes two examples of new context-aware operators, defined from the introduced architecture; Section V compares the architectural design of our operators with those in other mutation tools. Finally, Section VI presents some conclusions and future work perspectives.

II. BACKGROUND

Mutation operators insert faults in the system under test that should be similar to those that programmers unintentionally introduce into their systems. Software evolution has led to the proposal and development of multiple operators for all kind of testing levels, programming languages, paradigms and platforms. Thus, for example, first works about mutation testing targeted individual functions and methods of Fortran programs, in a kind of unit testing [2]; later, mutation operators for integration testing were developed [3], [4]; Ma and Offutt [5] proposed specific operators for object orientation, and implemented them in MuJava; Reales et al. [6] defined 58 mutation operators for testing multi-class systems at the integration and system levels; different authors have proposed specific operators for several programming languages (C [7], C# [8], C++ [9], Python [10] or PHP [11]); there are also mutation operators for other contexts: for relational databases [12], the ATL model-transformation language [13] or BPEL [14]. Jia and Harman [15] cite works about the application of mutation to state machines, Estelle specifications, Petri nets, network protocols, security policies and web services.

The variety of works is as wide the variety of systems, platforms and environments. Mutation operators for a certain type of system, paradigm or language are responsible of inserting the common faults that programmers and developers commit when they build the system: the *virtual modifier insertion* operator for C++ [9], for example, has nothing to do with a BPEL or a PHP specification.

Mobile software has recently received the attention of mutation researchers: Deng et al. [1] propose eleven mutation operators for testing several characteristics of Android apps, although they explicitly left for the future the implementation of others, specially those related to the context-awareness.

As developers of mutation tools, we are also concerned with the development of mutation operators for mobile apps. Both *testooj* [16] and *Bacterio* [17] deal with Java applications. Whilst *testooj* takes the mutants generated by MuJava [5] as input, *Bacterio* includes a mutant generator that in-

978-1-5090-6676-6/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSTW.2017.31

serts the faults directly in the bytecode. We are now building *BacterioWeb*, a version of *Bacterio* that runs on the web and performs all the mutation testing tasks on the server: from the mutant generation to the result analysis, and through the test case execution.

BacterioWeb is being developed almost from scratch, bearing in mind the goal of keeping quite easy the implementation and addition of new mutation operators. Furthermore, in this new tool we are focused on the mutation testing of mobile applications: the user interface of *BacterioWeb* offers the tester a list of the mobile devices and emulators that are available in the server. Then, the server runs the test cases against the app under test on the selected device or emulator.

III. ARCHITECTURE OF OPERATORS

Android developers usually write most of the application code in Java, commonly with Android Studio. The deployment of an application onto a mobile device requires several steps: (1) a translation of *.java* files to *.class;* (2) a new translation of each *.class* into a *.dex* file (which is the machine language understood by the Android Runtime and that is compatible with Dalvik, the virtual machine used before Android 5.0); and (3) the packaging of the *.dex* and other resource files into an *.apk* file that holds application.

As other tools, *BacterioWeb* also introduces mutants in the Java bytecode of the SUT. We use ASM, a powerful API to directly manipulate the bytecode produced by the Java compiler [18]. With ASM, a *.class* file can be loaded into a *ClassNode,* an object that wraps the class, holds all the information required to know the wrapped class details and offers all kind of operations to manipulate it. Thus, a *ClassNode* has the collections of fields and methods in two respective lists of *FieldNode* and *MethodNode*. Besides other information (*name, annotations, exceptions...)*, every *MethodNode* has its bytecode instructions in a *InsnList* object, which implements a doubly-linked list of *AbstractInsnNode* objects (Fig. 1).

A. Mutable instructions and mutant generation

The behavior of a mutant generator may consist in going through every mutation operator and asking it to get the mutants of the class to mutate.

Supposing (for the shake of clarity) that only constructors and methods can be mutated, the operator goes through every operation in the class and, for each operation, it goes in turn over all its instructions to determine whether it can or cannot mutate the method.

Fig. 2 shows the pseudocode of a possible implementation of a *generateMutants(c : Class)* method that belongs to the *Operator* class: as observed, it adds to a *mutableMethods* collection all the methods in *c* that it can mutate. For every mutable method, it calls an additional *mutate(c : Class, m : Method)* function, that applies the mutation operator to the method passed as parameter.

The behavior described in the pseudocode of Fig. 2 is common for all the mutation operators: thus, even though the *Operator* class must be abstract (because the change implementation obviously depends on the self operator), this operation may be concrete.

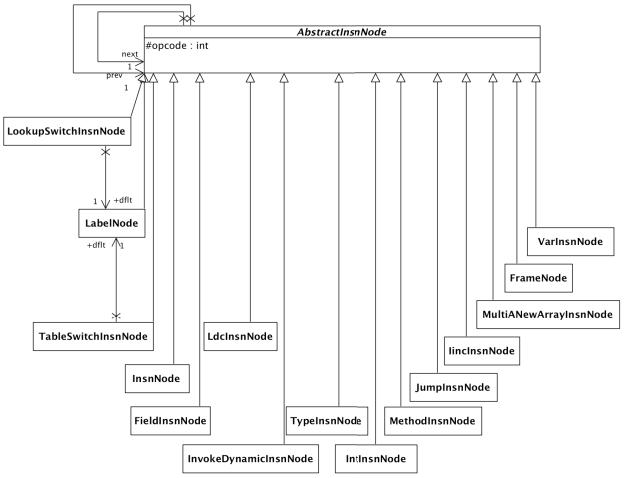

Fig. 1. Class hierarchy of instructions

The function called by *generateMutants* (i.e., *mutate(c : Class, m : Method)*) goes over the instructions of *m* and gets the corresponding mutants: if the operator can produce *p* mutants for a given instruction and there are *q* mutable instructions in the method, the operator must generate *p* x *q* mutants.

```
let be c the class to mutate
let be mutants = ∅
let be mutableMethods = ∅

for each method m in c
   if m is mutable then
      mutableMethods = mutableMethods ∪ { m }
   end
end
for each method m in mutableMethods
   mutants = mutants ∪ mutate(c, m)
end
```

Fig. 2. Pseudocode of *Operator::generateMutants(c : Class)*

Thus, for each mutable instruction in *m, mutate(c, m)* calls *mutate(c, m, instruction)*, that:

(1) Gets the list of changes applicable to the *instruction* passed.
(2) For each *change*, performs the mutation by calling *performMutation(method, instruction, change).*

Obviously, both getting the list of changes and performing the mutation depend on the concrete operator: for example, the application of UOI (*Unary Operator Insertion)* to *this.i=v;* returns *this.i=-v, this.i=v++, this.i=v--, this.i=++v* and *this.i=--v,* whereas AOR (*Arithmetic Operator Replacement)* would return nothing.

B. Defining the operators architecture

Since our goal is to define a reusable architecture to easily implement mutation operators, we have defined an abstract *Operator* class that holds as many concrete methods as possible. In Fig. 3:

- Each operator has two fields: the class file name

(which is used to process its bytecode with ASM) and the *family*, which is used to group the operators by categories in the web user interface. Some values of the *family* field can be "Traditional" (in the sense of the classification given in [5]) or "Sensors" (meaning that the operator is designed to sensors).

- Since we want to give the tool a plugin architecture (i.e., new operators can be added, loaded at applied at runtime), the class constructor is protected and is not visible from the outside. To instantiate and load the operators, the tool will look for all the concrete specializations of *Operator* and, over every one, it will call its constructor with a reflective call to its *newInstance* method (inside the *java.lang.Class*).
- *getName* returns the class operator name, and it is the acronym shown in the user interface. For example, if the *AOR* operator is implemented in the *AOR.class* file, it reflectively returns the "AOR" string.
- *getDescription* is abstract, because it returns a textual description of the operator. For AOR, for example, it returns "Arithmetic Operator Replacement".
- Both *mutate* methods implement the tasks described in the previous subsection, and they are concrete.
- *instructionIsMutable* is abstract, since its implementation depends on the concrete operator.
- *performMutation* modifies the method and instruction whose indexes are passed as parameters. The change may be a single instruction (substituting machine instructions *IADD* by *ISUB,* for example) or a list of instructions: thus, the third parameter is a list of instructions (i.e., an instance of *InsnList).* This is the method that actually builds up each mutant, returning it as a *ClassNode* object with its bytecode.

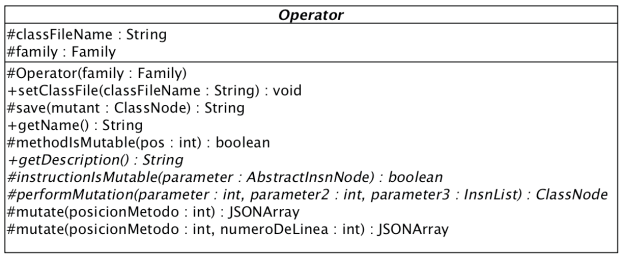

Fig. 3. Structure of the abstract *Operator*

As we pointed out, some operators, such as AOR, consist in the simple substitution of one bytecode instruction by another one, whereas others require the insertion of bytecode lines at a specific position of the method's *InsnList.* For this, *Operator* has the two direct, abstract specializations shown in Fig. 4 (*InsertionOperator* and *ReplacementOperator)*. This figure shows also the operators hierarchy for four of the "traditional" mutation operators implemented in *Bacterio Web:* AOR, ABS (*Absolute value insertion)*, ROR (*Relational Operator Replacement)* and *UOI* (which is implemented in several classes).

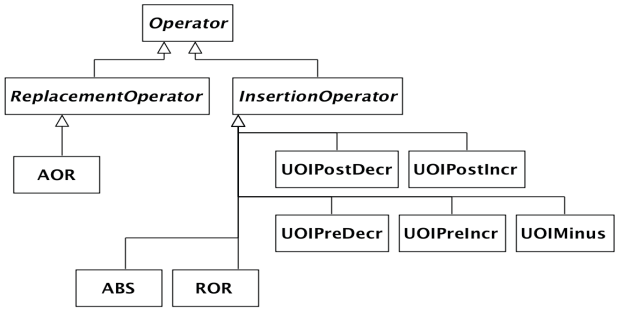

Fig. 4. Design of four of the *Traditional* operators

IV. ADDITION OF CONTEXT-AWARE OPERATORS

Taking advantage of the proposed architecture, we have defined and implemented several mutation operators for common context-awareness errors. These errors (TABLE I) come from the experience of three companies that are involved with us in a private research & development project. These errors will drive the definition and implementation of new operators.

The number and nature of the errors may grow up, which is one of the main reasons for (1) developing the hierarchy-based mutation operators design and (2) trying our best to keep the implementation of the actual mutation operators as simple as possible.

TABLE I. Categories and errors identified

Category	Errors
User interface	14
Connectivity	5
Screen orientation	6
Sensors	8
Interaction with other apps	2
Internal interaction	12
Database	6
Other errors	7
Total	60

A. A replacement operator

In Android, data from sensors are received in classes implementing the *android.hardware.SensorEventListener* interface, that offers the *onSensorChanged(SensorEvent event)* method. A *SensorEvent* holds the sensor measures in an array of floats (the *values* field is final and, therefore, cannot be changed), a reference to the source *Sensor,* the measure accuracy and the timestamp. This section describes some operators for sensors and explains how to arrive to the final implementation of one of them.

Usually, the app requests the use of a *Sensor* by means of the *getDefaultSensor(int sensorId)* operation in a *SensorManager* instance. In turn, this instance is recovered by the *getSystemService(int)* method of the app's *Context.* The sensor speed is set up with the *registerListener* method. When the sensor is no longer needed, it must be released

with a call to *unregisterListener(SensorListener)* in *SensorManager*. This common cycle of operations is summarized in Fig. 5. Some of the specific errors reported for Sensors appear in TABLE II.

1) Sensor request:

```
SensorManager sm =
  (SensorManager) ctx.getSystemService(Context.SENSOR_SERVICE);

Sensor sensor =
    sm.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);

sm.registerListener(this, sensor,
        SensorManager.SENSOR_DELAY_NORMAL);
```

2) Use of the sensor via asynchronous calls to:

```
public void onSensorChanged(SensorEvent event) {
        ... // Deal with the measure
}
```

3) Sensor release:

```
sm.unregisterListener(this);
```

Fig. 5. Typical cycle of a sensor use

TABLE II. Some errors for sensors

Method	Error
getDefaultSensor	a wrong sensor is requested
getDefaultSensor	the device has not the requested sensor
registerListener	wrong speed
onSensorChanged	*values* are used in a wrong order. i.e.: *{z, x, y}* instead of *{x, y, z}*
onSensorChanged	*values* are incorrectly interpreted, i.e.: *{- x, -y, -z}* instead of *{x, y, z }*
onSensorChanged	the sensor sends a *null* measurement

The first error (wrong sensor requested) appears when the programmer asks for a sensor different than the one needed (i.e., *TYPE_LINEAR_ACCELERATION* instead of *TYPE_ACCELEROMETER)*. Since the change to reproduce this error just consists in substituting the value of the constant representing the sensor, the corresponding mutation operator for this error is a specialization of *ReplacementOperator* that must be applied when the *getDefaultSensor* method is called. In bytecode:

```
ICONST_1
INVOKEVIRTUAL
 android/hardware/SensorManager.getDefaultSensor
 (I)Landroid/hardware/Sensor;
```

The first instruction (*ICONST_1)* refers the constant value of the *TYPE_ACCELEROMETER:* it can be another constant (up to *ICONST_5)* or a *BIPUSH value* instruction for higher values. The second one is the call to the method.

Therefore, the mutation operator will be applied before the second instruction. If the value loaded before this method is a constant (from *ICONST_0* to *ICONST_5),* this value will be changed by a different one. If it is a higher value, it will be changed by *ICONST_1*.

The implementation of the operator is straightforward: besides the implementation of three simple methods (the constructor, *getDescription* and *performMutation),* the main difficulties are in *instructionIsMutable* and in *getChanges:*

- *instructionIsMutable* returns true only when the instructions is a call to *Sensor::getDefaultSensor.*
- *getChanges* returns a *InsnList* with just one instruction, which is the new *InsnNode*.

B. An insertion operator

One of the reported errors with respect to the use of the screen comes from unexpected events from the user, such as touching twice on a widget or rotating the device. This event produces a change of the coordinates where the user touches: the screen center, instead of being *(x, y),* passes to be *(y, x)*.

Depending on the operation where the event is collected, the data in the event may arrive in a different type of object. For example, the *onTouchEvent* method of the *View* class receives a *MotionEvent* object as its only parameter.

To simulate the change of coordinates, it is enough to add a call to the *setLocation(float, float)* method as the first line of any implementation of *onTouchEvent,* interchanging the coordinates with *getX* and *getY:*

```
public boolean onTouchEvent(MotionEvent event) {
     event.setLocation(event.getY(), event.getX());
                ...
}
```

The insertion of this single Java statement requires the insertion of six new bytecode instructions. We must redefine *methodIsMutable* method (which has a default implementation in *Operator,* see Fig. 3) and *instructionIsMutable:*

- *methodIsMutable* returns *true* only for the *onTouchEvent(MotionEvent)* of the *View* class.
- Since the mutation is introduced as the first instruction of the method, this operator requires a boolean field *isFirstInstruction,* in such way that *instructionIsMutable* returns true only the first time is executed.

Fig. 6 shows the inclusion of these two operators in the architecture.

V. COMPARISON WITH OTHER TOOLS

Other tools make also use of inheritance for implementing their mutation operations, and all of them require external libraries to manipulate the bytecode. Both inheritance and external libraries increase the system coupling.

Coupling "is a qualitative measure of the degree to which classes are connected to one another" [19]. The coupling may be better or worse depending on the impact that a change in a part of the system has on the others. Lethbridge and Laganiére define several coupling categories [20]. Inheritance introduces *Content coupling*, probably the most dangerous of all, since the structure and behavior of all subclasses have a complete dependence on all their ancestors; thus, the modification of a superclass affects all its descendants. If the superclass is implemented in a third-party component, then the evolution of our system becomes completely dependent on the evolution of such external system.

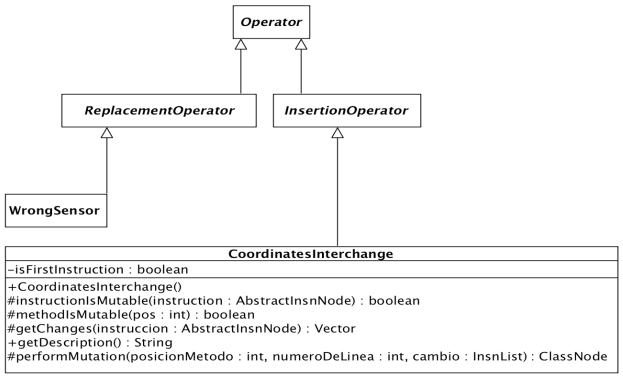

Fig. 6. Addition of *CoordinatesInterchange* as an *InsertionOperator*

Type use is a "not so bad" type of coupling. It occurs when "component *A* uses a data type defined in component *B"* [19]. If *B* is the external library and this does not evolve according to *A's* requirements, *A* must be modified, maybe with the substitution of *B* by a new library. This type of coupling is better than content coupling because the structure and behavior of *A* is actually implemented in *A* itself, being under the control of *A's* developer.

Due to these risks (ASM is an external library), the development of operators in *BacterioWeb* uses *Type use* coupling (Fig. 7) and the dependence on changes of ASM is not so strong as with *Content coupling*. The "old" *Bacterio* used ASM too. Although it had no dependence on hierarchy from classes in ASM, we have taken advantage of the lessons learned during its development in this new implementation.

Next sections review the operators in two other tools.

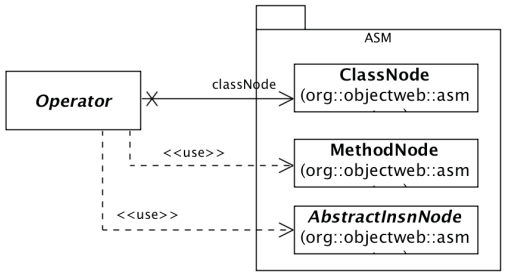

Fig. 7. Dependencies of our *Operator* with respect to ASM

A. MuJava

μJava [5] was first released in 2003, although it has undergone several improvements over time. It performs mutation at bytecode level using *OpenJava* [21], a library to manipulate Java bytecode that, according to its webpage, has not been updated since 2007.

Fig. 8 shows a partial view of the hierarchy of mutation operators in MuJava: *ABS* and *ABS_AOR_LCR_ROR_UOI* are "traditional" operators, whilst *IHD* is an object-oriented operator. As observed, there is a strong coupling by inheritance of all the operators with respect to *OpenJava*: note that all the classes in the *MuJava* core (those with no package name in Fig. 8) are descendant of classes included in this external library. Thus, the discontinuation in the development of OpenJava seriously threatens the evolution of this interesting mutation tool.

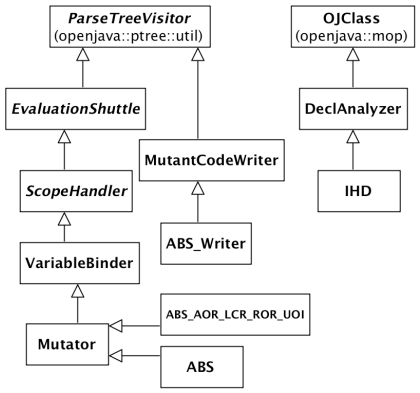

Fig. 8. Operators' architecture in *MuJava*

B. Javalanche

Javalanche is another tool developed by Schuler and Zeller [22]. As *BacterioWeb,* it uses the ASM library to insert the changes in the bytecode. As *MuJava*, its operators are also specializations of classes in the external library. In spite of ASM is evolving from 2002 to 2016, the github site of *Javalanche* has no commits since march 2012.

C. Pitest

This tool [23] also performs mutation at bytecode level using ASM. Pitest is Maven-based and, thus, is not explicitly invoked by the tester, but it is added as a plugin to the *pom.xml* project file. As most tools, it also has *Content coupling* by inheritance with ASM classes. As an example, Fig. 9 illustrates that how the *GregorMutationEngine* uses a *GregorMutater* that, in turn, uses a *MutatingClassVisitor* that directly inherits from the ASM *ClassVisitor*. Anyway, most of the core *Pitest* classes are decoupled from ASM via a very strong use of wrappers, and the use of interfaces probably allows the development of independent mutation engines.

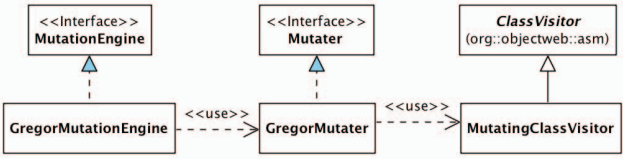

Fig. 9. A sample of content coupling in *Pitest*

VI. CONCLUSIONS AND FUTURE WORK

This paper has presented the architecture of mutation operators we are developing for *BacterioWeb.* Its goal is to make easy the development of new operators, as well as to reduce the dependence of external libraries. The class hierarchy has been carefully designed to minimize the code required in the implementation of new operators.

The architecture supports the implementation of classic operators, but also of others for specific methods by means of the *methodIsMutable* operation defined in *Operator,* the root of the hierarchy. Overriding this method makes

possible to build particular operators for specific operations of the system under test that, for example, can be identified by its name. *methodIsMutable*, together to *instructionIsMutable,* are specially useful for the context-awareness characteristics of mobile applications, which is one of our current research areas. On the other side, the use of reflection for recovering the implemented mutation operators gives a plugin architecture to the mutant generation module of *BacterioWeb:* operators can be written and added on the fly, and the tool is able to apply them immediately.

Although it has not been the focus of this paper, the development of this web version of *Bacterio* is also a challenge. In fact: (1) it changes the way of dealing with the project under test, that is hosted in the server and can be shared with other testers; (2) WebSockets give the user feedback about the execution advance; (3) testers may also share mutation operators; (4) testers do not need to deal with mobile devices or emulators, which are connected to the server; (5) we will need to face performance problems when different testers execute several test suites of different projects, maybe with cloud approach and parallel execution of mutants [24].

VII. Acknowledgments

This work has been developed within the GINSENG Project, TIN2015-70259-C2-1-R, Fondo Europeo de Desarrollo Regional & Ministerio de Economía y Competitividad.

References

[1] L. Deng, J. Offutt, P. Ammann, and N. Mirzaei, "Mutation operators for testing Android apps," *Inf. Softw. Technol.*, vol. 81, pp. 154–168, Jan. 2017.

[2] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[3] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, "Interface Mutation: an approach for integration testing," *IEEE Trans. Softw. Eng.*, vol. 27, no. 3, pp. 228–247, 2001.

[4] S. Ghosh and A. P. Mathur, "Interface mutation," *Softw. Test. Verification Reliab.*, vol. 11, no. 4, pp. 227–247, 2001.

[5] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: an automated class mutation system: Research Articles," *Softw Test Verif Reliab*, vol. 15, no. 2, pp. 97–133, Jun. 2005.

[6] P. Reales, M. Polo, and J. Offutt, "Mutation at the multi-class and system levels," *Sci. Comput. Program.*, vol. 78, no. 4, pp. 364–387, 2012.

[7] E. S. Hiralal Agrawal Richard A.DeMillo, Bob Hathaway, William Hsu, Wynne Hsu, E. W.Krauser, R. J.Martin, Aditya P.Mathur, "Design of Mutant Operators for the C Programming Language," Purdue University, Mar. 1989.

[8] A. Derezińska, "Advanced mutation operators applicable in C# programs," in *Software Engineering Techniques: Design for Quality*, K. Sacha, Ed. Springer US, 2007, pp. 283–288.

[9] P. Delgado-Pérez, I. Medina-Bulo, J. J. Domínguez-Jiménez, A. García-Domínguez, and F. Palomo-Lozano, "Class mutation operators for C++ object-oriented systems," *Ann. Telecommun. - Ann. Télécommunications*, pp. 1–12, Sep. 2014.

[10] A. Derezińska and K. Hałas, "Analysis of Mutation Operators for the Python Language," in *Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX. June 30 – July 4, 2014, Brunów, Poland*, W. Zamojski, J. Mazurkiewicz, J. Sugier, T. Walkowiak, and J. Kacprzyk, Eds. Springer International Publishing, 2014, pp. 155–164.

[11] P. Brady, "Halleck45/MutaTesting," *GitHub*. [Online]. Available: http://bit.ly/2jCZrx5. [Accessed: 04-Sep-2014].

[12] J. Tuya, M. J. Suárez-Cabal, and C. de la Riva, "Mutating database queries," *Inf. Softw. Technol.*, vol. 49, no. 4, pp. 398–417, Apr. 2007.

[13] J. Troya, A. Bergmayr, L. Burgueño, and M. Wimmer, "Towards systematic mutations for and with ATL model transformations," in *8th Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW)*, 2015, pp. 1–10.

[14] A. Estero-Botaro, F. Palomo-Lozano, and I. Medina-Bulo, "Quantitative Evaluation of Mutation Operators for WS-BPEL Compositions," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, Washington, DC, USA, 2010, pp. 142–150.

[15] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, 2011.

[16] M. Polo, S. Tendero, and M. Piattini, "Integrating techniques and tools for testing automation: Research Articles," *Softw Test Verif Reliab*, vol. 17, no. 1, pp. 3–39, Mar. 2007.

[17] P. Reales and M. Polo, "Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases," in *28th IEEE Int. Cong. on Software Maintenance (ICSM)*, 2012, pp. 646–649.

[18] E. Bruneton, R. Lenglet, and T. Coupaye, "ASM: a code manipulation tool to implement adaptable systems."

[19] R. S. Pressman, *Software Engineering: A Practitioner's Approach*. New York: McGraw-Hill, 2009.

[20] T. Lethbridge and R. Laganiere, *Object-Oriented Software Engineering: Practical Software Development Using UML and Java*, Edición: 2. London: McGraw-Hill, 2004.

[21] "OJ : An Extensible Java." [Online]. Available: http://www.csg.ci.i.u-tokyo.ac.jp/openjava/. [Accessed: 01-Dec-2016].

[22] D. Schuler and A. Zeller, "Javalanche: efficient mutation testing for Java," 2009, p. 297.

[23] Henry Coles, "Pitest." [Online]. Available: http://pitest.org/. [Accessed: 01-Dec-2016].

[24] P. Reales and M. Polo, "Parallel mutation testing," *Softw. Test. Verification Reliab.*, vol. 23(4), pp. 315–350, Mar. 2012.