World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:6, No:4, 2012

Verification and validation for Java classes using Design by Contract. The modular external approach

Darío Ramírez de León, Oscar Chávez Bosquez, and Julián J. Francisco León

Abstract—Since the conception of JML, many tools, applications and implementations have been done. In this context, the users or developers who want to use JML seem surounded by many of these tools, applications and so on. Looking for a common infrastructure and an independent language to provide a bridge between these tools and JML, we developed an approach to embedded contracts in XML for Java: XJML. This approach offer us the ability to separate preconditions, posconditions and class invariants using JML and XML, so we made a front-end which can process Runtime Assertion Checking, Extended Static Checking and Full Static Program Verification. Besides, the capabilities for this front-end can be extended and easily implemented thanks to XML. We believe that XJML is an easy way to start the building of a Graphic User Interface delivering in this way a friendly and IDE independency to developers community wich want to work with JML.

Keywords—Model checking, verification and validation, JML, XML, Java, Runtime Assertion Checking, Extended Static Checking, Full Static Program Verification

I. Introduction

JML was created by Gary T. Leavens [1] and since its conception much work have been done. This drives us between a large amount of tools, duplication of effort & high (collective) maintenance overhead. JML is the most popular Behavioral Interface Specification Language (BISL) for Java [2], [3] supporting Runtime Assertion Checking (RAC), Extended Static Checking (ESC) and Full Static Program Verification (FSPV), in fact, is the only BISL supported by all three of these verification technologies [2], [3].

However, the evolution of Java as programming language and the improvements for JML tools and applications, forces the JML community to keep on date, a hard task which produces many variants and alternatives of these tools and applications.

In this paper we introduce the XJML architecture. As far as we know, it is the first approach to embedding contracts for Java using JML and XML. XJML is built on top of JML4 using the RAC and ESC verification technologies. Also, XJML uses Krakatoa [4], a FSPV tool for Java classes, with JML annotations. Since XJML uses XML, we can deliver a greater uncoupling between Java and JML[1].

The rest of this paper is divided as follows: in the second section we introduce the problem found when we wanted to work with JML and its related verification techniques for Java classes. We focus on the last efforts from the JML community to integrate the three verification techniques: RAC, ESC, and FSPV. Specifically JML4, JMLEclipse and OpenJML [5]. In third section we present our architecture that makes possible embedding contracts in XML using the Java Modeling Language. We called this architecture XJML (X because we can write contracts in XML, and JML for the Java Modeling Language, one underlying language supported in the contracts). XJML already supports RAC, ESC and FSPV, unlike JML4 and JMLEclipse. The fourth section contains preliminary tests and screenshots with XJML 1.0. In the fifth section we present the related work. Last section contains conclusions and future works.

II. The problem

Because there are many tools, applications and implementations working with JML, users and developers who want to use JML seem surounded by many software. To our knowledge, JML4 and JMLEclipse were the last efforts to generate an Integrated Verification Environment (IVE) for JML [2], [3].

Both of them, JML4 and JMLEclipse, aim to support the RAC, ESC, and FSPV verification techniques. Nevertheless, there is no stable release of JML4 nor JMLEclipse, and it seems that JMLEclipse was abandoned. Moreover, it looks that no one else, outside from the JML community has done empirical tests with JML4 or JMLEclipse, according to its main project pages on the Web.

In the other side, OpenJML is the main effort for JML tools, but still under development. So, we propose XJML, mainly motivated for the lack of one stable release of JML4, JMLEclipse and OpenJML.

In this way, we believe that XJML can offer the first stable release, supporting JML and the RAC, ESC and FSPV verification techniques, all this through one XML file, within an architecture that we have called the modular external approach.

III. XJML's Architecture

Since JML is intended for Java, then the architecture for XJML too. The main components for this architecture are:

D. Ramírez de León is with the División Académica de Informática y Sistemas, Universidad Juárez Autónoma de Tabasco, Tabasco, México (e-mail: msc-dar_ram@hotmail.com)

O. Chávez Bosquez is with the División Académica de Informática y Sistemas, Universidad Juárez Autónoma de Tabasco, Tabasco, México (e-mail: oscar.chavez@ujat.mx)

J.J. Francisco León is with the División Académica de Informática y Sistemas, Universidad Juárez Autónoma de Tabasco, Tabasco, México (e-mail: julian.francisco@ujat.mx)

Manuscript received April 19, 2005; revised January 11, 2007.

[1]Unlike JML expressions in comments, annotations, java interfaces, or in jml files

- The class `Preprocessor`, which main function is parse the XML file who contains the contract.
- The abstract class, `Processor`, with abstract methods for compile, run, add tool (RAC, ESC, FSPV) among others. For our tests with JML we've developed JML-Processor, which extends from `Processor`.
- One class, `Compiler` who executes the proper process of compilation, choosing between JML and others specification languages (for this case, we are just working with JML).
- The `AbstractRunner`, one abstract class who defines, simply the `run(String, Class<?>)` mehod. This abstract class was made it to be extended since others, for instance, we've developed `RACRunner`, which only delegates the RAC to the right tool.
- `MyException`, one abstract class inheriting from `java.lang.Exception`. Its function is abstracting the common exceptions (AssertionNotFound, ClassNameNotMatch, ClassScopeNotMatch, MethodNotFound, NotInstanceOf, NumberOfParamNotMatch, ParamNotFound, ReturnTypeNotMatch, WrongAssert) for our architecture.

There are other components besides the listed above (i.e., the W3C XML Schema -XSD- which defines the semantic for build contracts with XJML). The full architecture is ilustrated in the figure 1 that shows the component diagram for the XJML's architecture.

A. The XJML Preprocessor Core

The heart of the XJML architecture contains two XML Schema Definition (XSD files) who has the function to express the semantic rules required to build con- tracts with XJML. Besides, there are two Java class. These components are defined as following.

1) Rules-vv-dbc.xsd: An XML schema definition containing the semantic rules required to build contracts with XJML. This XSD import Types.xsd.

2) Types.xsd: This XSD has the definition of the main types used in Rules-vv-dbc.xsd. For instance, the types indicating the scope for one java class (private, protected, public), the primitive types and wrappers valid in Java (void, int, Integer, long, Long, etc.).

Both, Rules-vv-dbc.xsd and Types.xsd plus one XML file (this last is the contract container, validated against Rules-vv-dbc.xsd). This XML file (since now, called, the rules file) is an input parameter for our Preprocessor, so, the rules file is parsed in Preprocessor.

3) Preprocessor.java: This class together with `Processor` are the core of XJML. The main functions of the `Preprocessor` are:

- parse the rules file,
- check the conformance between the rules and the Java file. for example, if we define in the rules file that a method `add(int, int)` must be `private`, return one `int`, and in our Java file, we have `public void add(int)` then the Preprocessor has to report one `MethodNotFoundException` because the Java file is not valid against its rules.
- delegate the compile and run tasks to the right compiler and runner

4) XMLElements.java: This final class only contains attributes `public static final java.lang.String` like `CLASS_TAG`, `METHOD_TAG`, `NAME_ATTR` among others.

B. XJML Processors

The XJML processor package contains the abstract class `Processor`, which has abstract methods to generate preconditions, posconditions and class invariants. To do this, we develope the final class `JMLProcessor`.

1) JMLProcessor.java: The intention of this final class (which extends from `Processor`) is to hide, to the ordinary, non specialized in formal methods, developer, the details of how XJML compile the class, run the verification tools (RAC, ESC and FSPV) and report the results of each tool. By doing this, we believe that XJML can bring closer to the developer the concepts of formal methods, and at the same time, we can offer to the GUI designers one bridge between the front-end (moreover the GUI, remember that one file must exist for each class to verify, this file is the contract, written in XML) and the back-end (the processors, compilers, `Runners`, and so on).

By space limitations, we only focus in the `public boolean compile(java. lang.String)` (inherited from the `Processor` class). You will find all the source code for the `JMLProcessor` and the XJML project on Eclipse in http://sourceforge.net/projects/xjml/.

The `public boolean compile(java.lang. String)` method has the job to compile the class indicated by the input param `java.lang.String`. To do this, we build one array of `java.lang.String`, called `argsCompiler`, containing the arguments for the JML4c. Then, we delegate the compiling task to the `Compiler` class in the XJML Compiler package, passing as arguments the compiler type and the `argsCompiler`. So we have:

```
public boolean compile(String file) {
    String[] argsCompiler = {"java", "−jar", "jml4c.jar", file +
        ".java"};
    return
        mx.ujat.dais.tesis.msc.dario.compilers.Compiler.compile
      (TYPE_COMPILER.JML, argsCompiler);
}
```

Finally, but not least important the `TYPE_COMPILER` is a `public static enum` defined in the `Compiler` class:

```
public static enum TYPE_COMPILER {JML, OTHER}
```

C. The XJML Compiler

This subsystem only contains the `Compiler` component. Nevertheless, the design of XJML does not prevent the creation of new Compilers, so you can extend XJML to use another modeling languages.

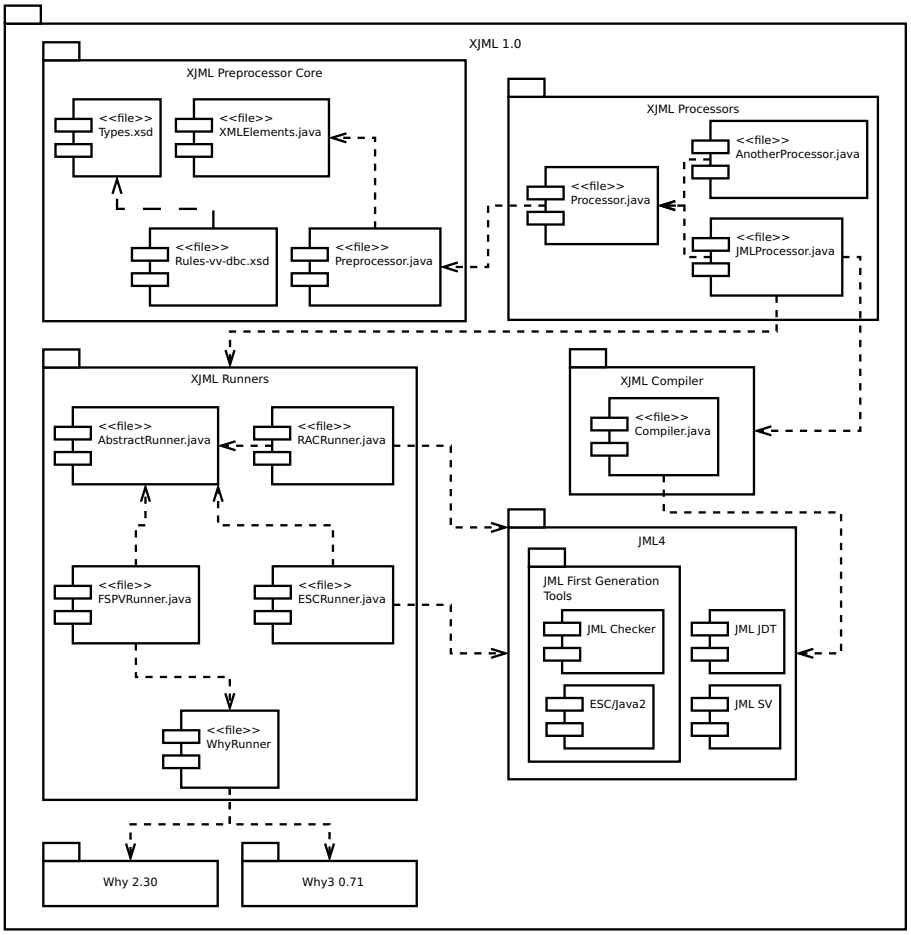

Fig. 1. Diagram component of XJML architecture

1) Compiler.java: Its duty, as you can figure it out, is to perform the compilation process for one Java class and its contract. To do this, the `Compiler` class works like one wrapper for the `jml4c` [6].

The `Compiler` class contains, among others, the method `public static boolean compile(TYPE_COMPILER, java.lang.String[])`. The first argument is the compiler that XJML will invoke. Pro tem, the only supported compiler is `jml4c`. The compilation task is performed through:

```
Process p = Runtime.getRuntime().exec(args, null, new
    File(System.getProperty("user.dir") + "/tmp"))
```

where the `System.getProperty("user.dir") + "/tmp"` is the directory where `jml4c` resides. As you can see, this method must return one `boolean` value, this value will be returned in this way: `return p.waitFor() == 0`.

Last, the `java.lang.String[]` are the arguments passed to the compiler.

D. XJML Runners

This subsystem of XJML is conformed by five components: one `AbstractRunner` and the corresponding runners for RAC (`RACRunner`), ESC (`ESCRunner`), and FSPV (`FSPVRunner`). The remaining runner corresponds to the `WhyRunner`. Of course, there is one Java class for each of them.

The job of these components is to execute the V&V of the Java class, according to its specifications, written in the contract (the XML file).

Note that: RAC is performed through the `jmlrac` tool [7], [8]. ESC is performed through ESC/Java2 [9] and FSPV is performed through the Why platform [10].

IV. Testing XJML 1.0

So far, we have presented the XJML's architecture (figure 2 shows the XJML project in the Eclipse IDE). Now, it's time to show how XJML 1.0 works. For this, we chose the `AccountBank` class presented in [11]; we believe that the `AccountBank` class is analogous to "Hello World" programs, but in the field of Verification and Validation.

The first thing to do, is write the contract for the Java class. We will call this contract `AccountBank.xml` (figure 3), it will be the contract for the `AccountBank` class.

For space limitations we will not include the content of any file from XJML. Instead, you will find the `AccountBank.xml` in http://goo.gl/9I5eV.

Then, if we want verify the `AccountBank` class with XJML 1.0, the second thing to do is modify the class by removing the JML expressions. So, let's check it; our `AccountBank` class (figure 4) to verify with XJML 1.0 will find in http://goo.gl/9iObb.

Lastly, we need the `Main` class (figure 5). In XJML 1.0, this class has the duty to bind the Java class with its respective contract. There are not established rules for this class, except the fact that its `public static void main(java.lang.String[])` method is mandatory. You can download this class through: http://goo.gl/nPux5.

In the `public static void main(java.lang.String[])` method of the `Main` class, you must specify the Java class and its contract.

If you have all these three files, just have to run the `Main` class. So, the figures 6 to 8 present the output for `RACRunner`, `ESCRunner` and `FSPVRunner`, respectively.

V. Related Work

The Java Modeling Language has evolved through the years [12], starting its conception, passing from JML2, JML3, JML4, JML5, JMLEclipse, OpenJML, to our knowledge, the last effort from the JML community to produce an Integrated Verification Environment.

Unfortunately, one first official (non-beta) release of JmlEclipse is targeted to be determined as we can see in [13].

Also exists OpenJML, wich is the next generation of the core JML tools and will support Java 1.7. It is based on OpenJDK but unfortunately the installation fails [14] if the underlying Java VM is not a suitable Java 1.7 VM. Also there's a lot work to do, and the GUI features for its Eclipse plug-in are not yet defined [14].

Regardless the efforts from the JML community to integrate Runtime Assertion Checking, Extended Static Checking and Full Static Program Verification, it seems, and to our knowledge, there is no tool that integrates these verification techniques, and what is more important, there is no tool for JML that allows the user to write contracts for a Java class in a XML file.

We choose the Extensible Markup Language (XML) as the language to write the contract for one Java class in XJML 1.0 because of its important role in the exchange of a wide variety of data [15].

In this sense, PiXL (Protocol Interchange using XML Languages) [16] is a clear example of how the XML technologies and standards can be used for the integration of analysis tools. However, PiXL has not support for JML.

So, we cay say that XJML is the first approach for JML that allows to write contracts for one Java class in one XML file, supporting RAC, ESC and FSPV.

VI. Conclusion

The Java Modeling Language (JML) is perhaps the best known modeling language for Java classes. There is a wide community of JML experimenting with it, and their last efforts are focused in build one Integrated Verification Environment (IVE) [2], where the verification techniques: Runtime Assertion Checking (RAC), Extended Static Checking (ESC) and Full Static Program Verification (FSPV), can be integrated.

Of these three verification techniques, FSPV it seems the harder to be integrated and the last effort (OpenJML) are not ready yet. We believe that XJML is the first in its type, where the JML expressions are not in the Java class. Instead, these expressions can be embedded in one XML file. Furthermore, XJML to our knowledge, is the first tool that bring together RAC (using `jmlrac`, from JML4), ESC (with ESC/Java2) and FSPV (where unlike JML4 and JMLEclipse, XJML uses the Why platform).

We are working with experiments using the `AccountBank` class. These experiments must use RAC, ESC and FSPV techniques, through, of course `RACRunner`, `ESCRunner` and `FSPVRunner` of XJML 1.0. After this, we are going to make sure that there are no alteration between our results and the results of each verification technique running separately.

XJML 1.0 only supports preconditions, posconditions and class invariants, so we hope that in the next updates of XJML we can offer support for more JML expressions.

The third thing to do is to build a Graphic User Interface (GUI) for XJML. Then, we are thinking in one second release of XJML.

We hope that in XJML 2.0, the user does not need to write the `Main` class (responsible for bind the Java class and its contract). Instead, the GUI will make easier this job, hiding this class to the user.

Finally, we hope that XJML will be useful, for the JML community and for all the Java programers.

Acknowledgment

This work has been supported by the project *Development of a technological infrastructure to support the Human Rights Commission of the state of Tabasco using advanced software tools to care for vulnerable groups*, thanks the Fondos Mixtos, through the Consejo Nacional de Ciencia y Tecnología (CONACYT). The number of project was TAB–2008–C03–95740.

We also thank to Gary T. Leavens, Kuat Yessenov, Greg Dennis, and Homero Alpuin, for all their support, email responses, suggestions, and more.

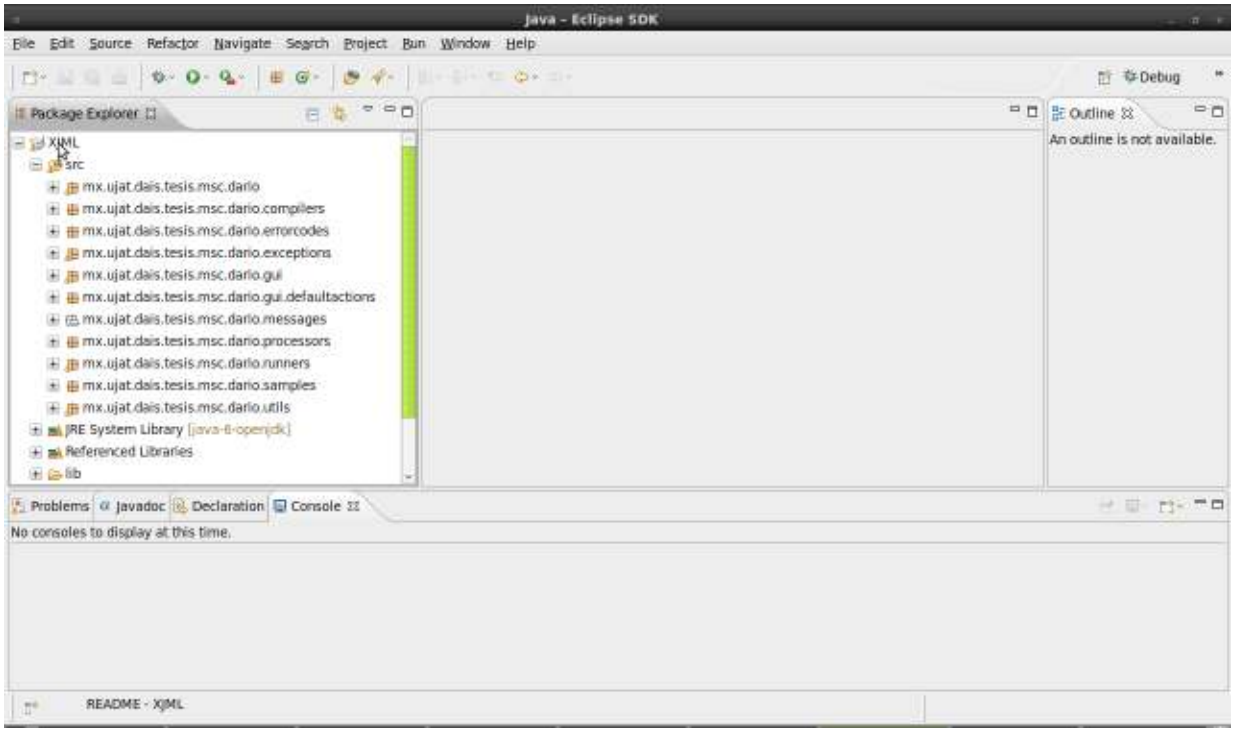

Fig. 2. The XJML 1.0, in Eclipse IDE

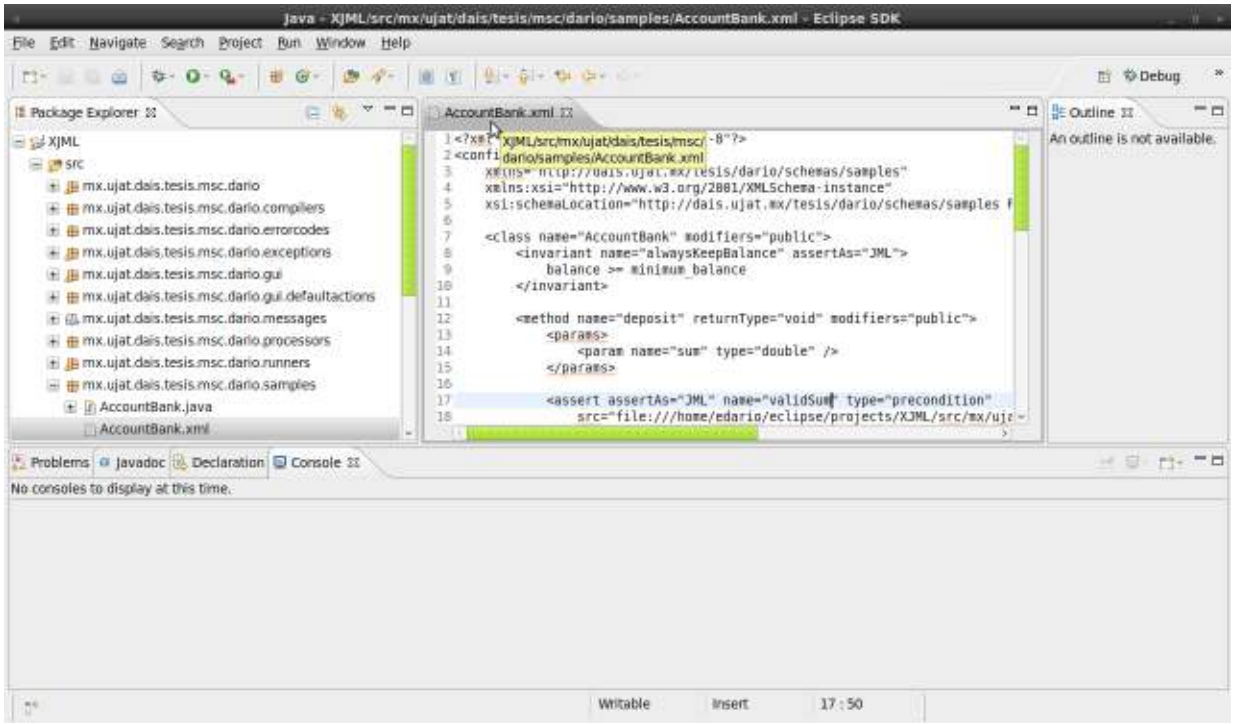

Fig. 3. The `AccountBank` contract

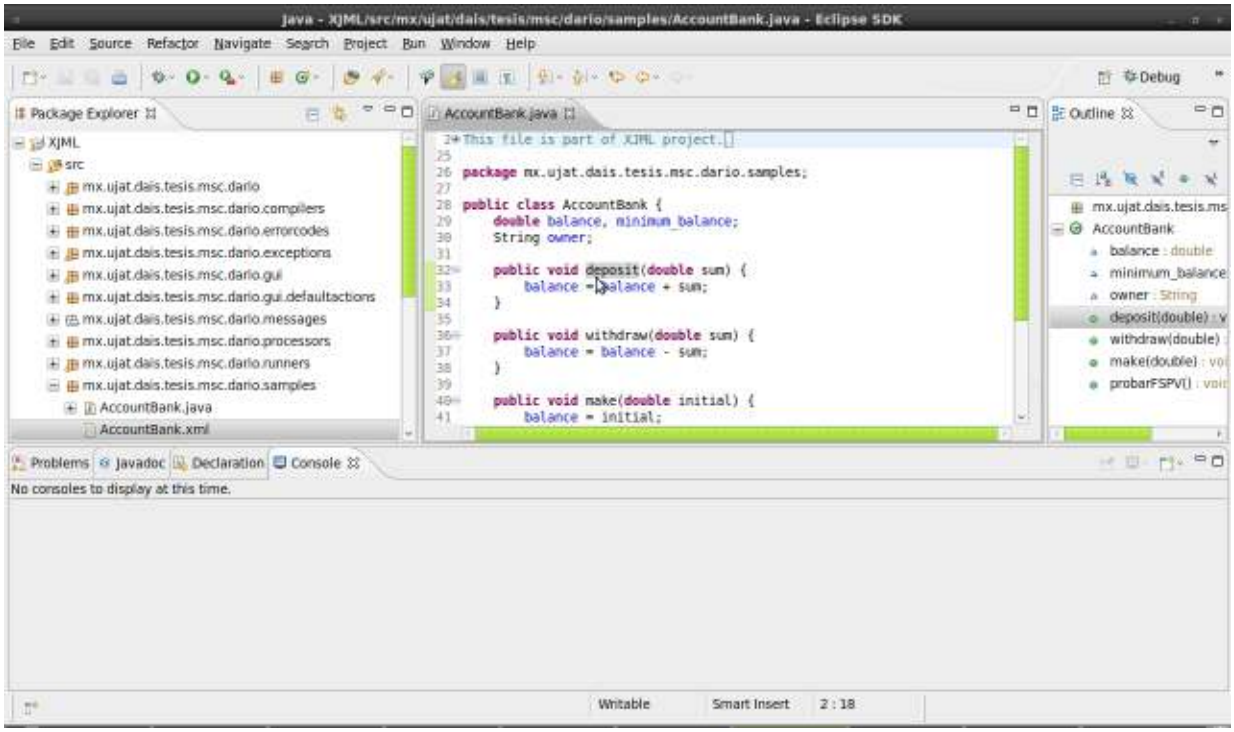

Fig. 4. The `AccountBank` class

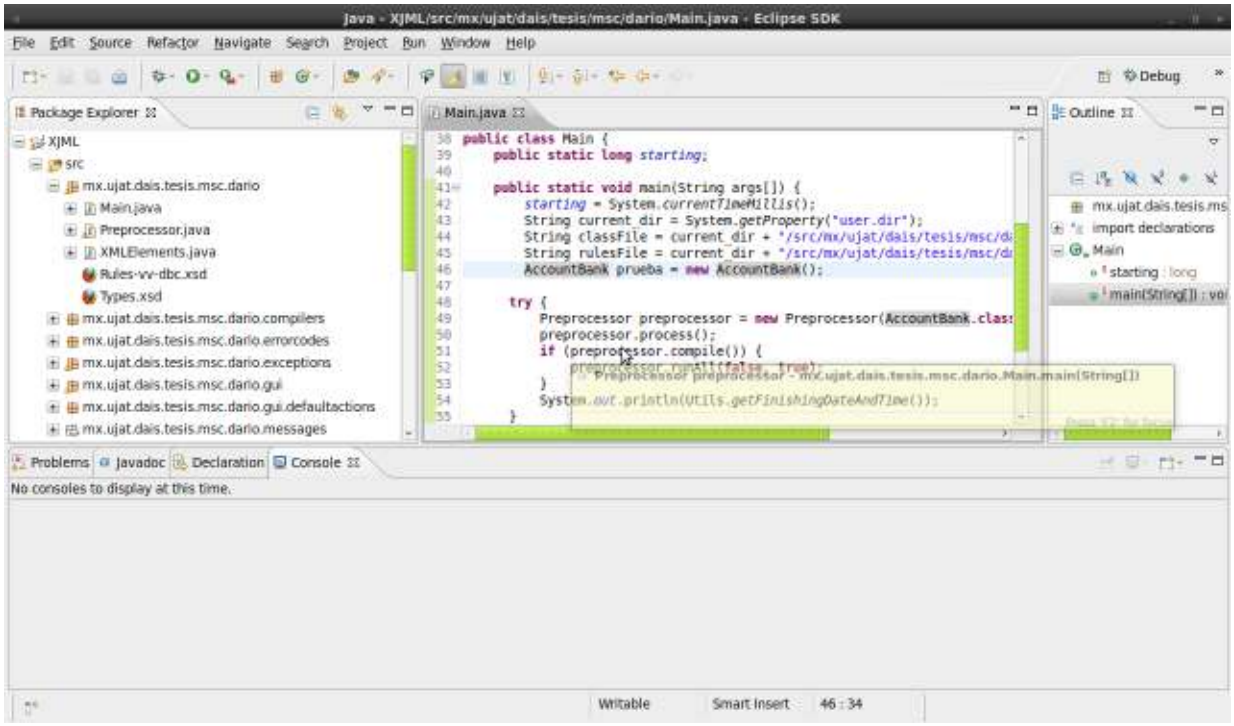

Fig. 5. Binding the Java class with its contract: the `Main` class

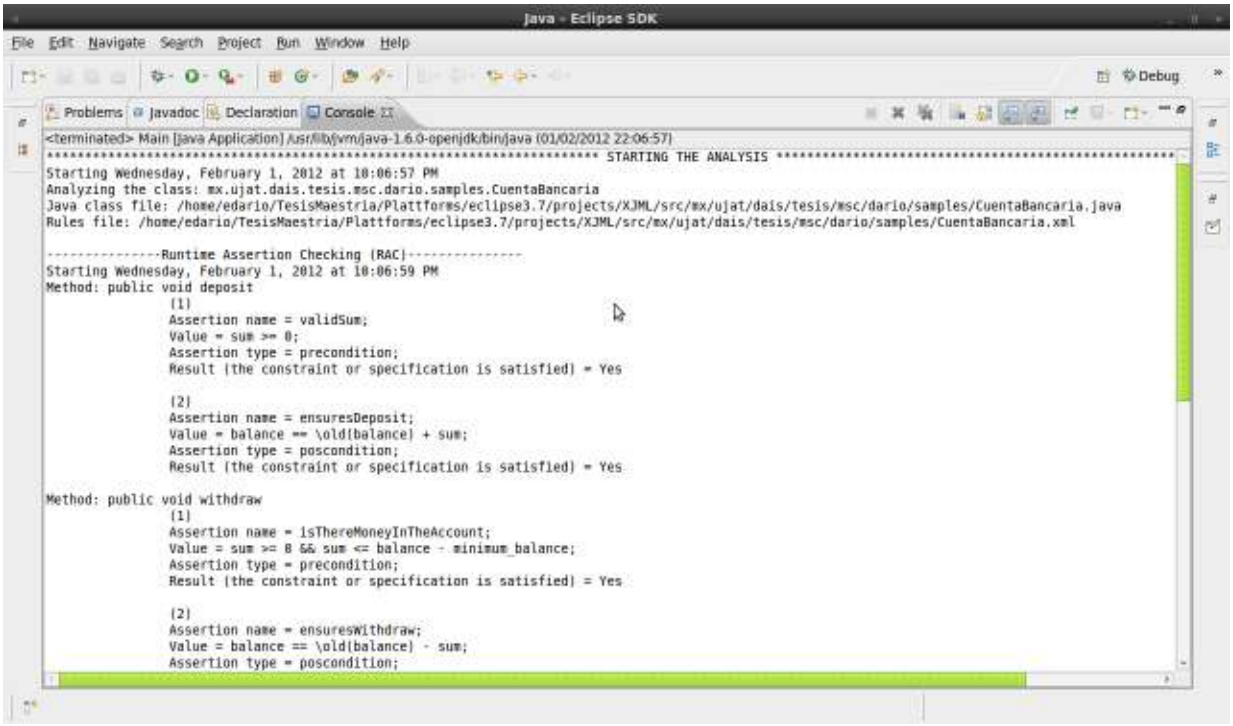

Fig. 6. Running RAC through XJML 1.0

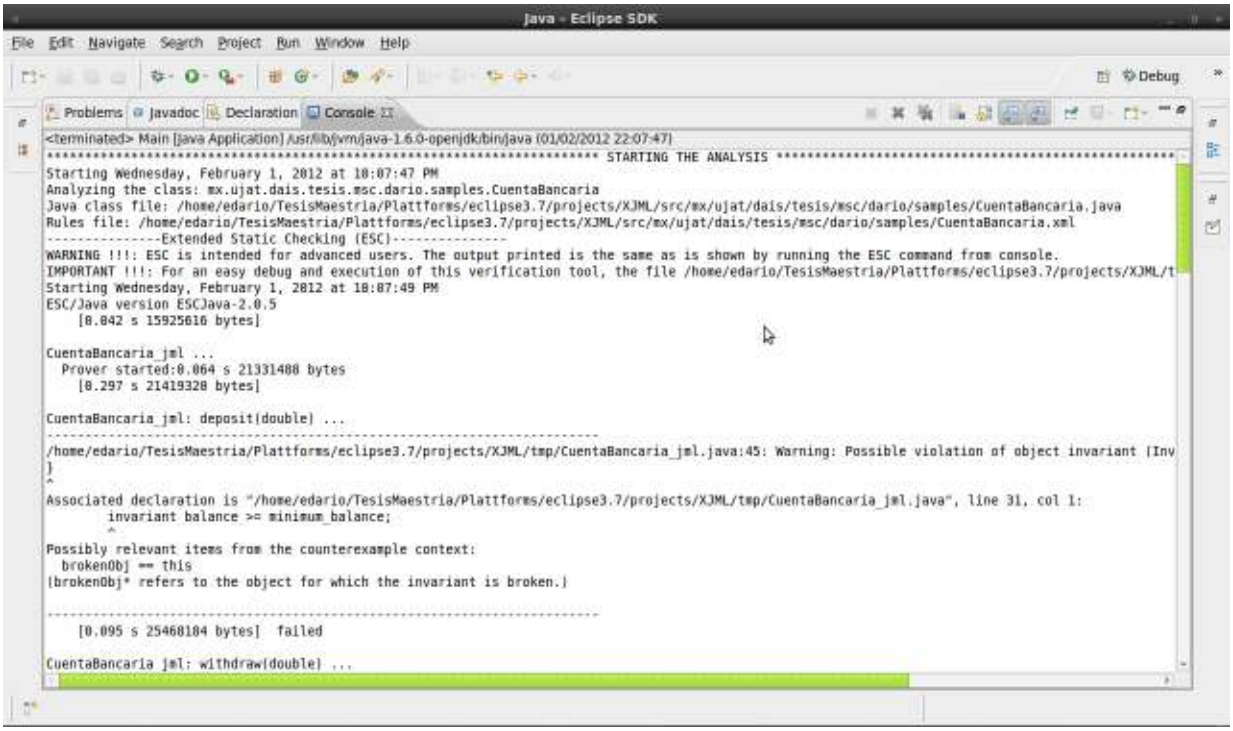

Fig. 7. Running ESC through XJML 1.0

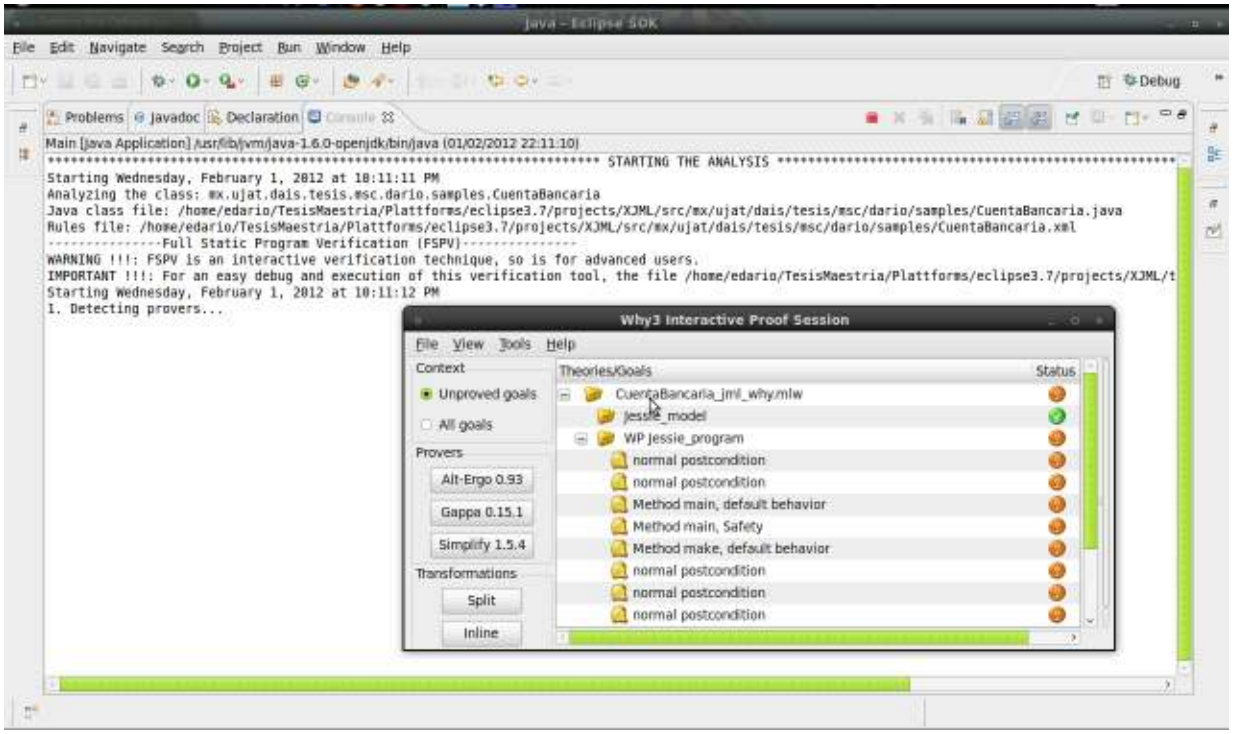

Fig. 8. Running FSPV through XJML 1.0

REFERENCES

[1] Gary T. Leavens's Home Page, http://www.cs.ucf.edu/~leavens/homepage.html
[2] P. Chalin, P. R. James, and G. Karabotsos: JML4: Towards an Industrial Grade IVE for Java and Next Generation Research Platform for JML. In: Proceedings of the International Conference on Verified Software: Theories, Tools, Experiments (VSTTE). Toronto, Canada, Oct. 6-9 (2008)
[3] P. Chalin, P. R. James, and G. Karabotsos: An Integrated Verification Environment for JML: Architecture and Early Results. In: Proceedings of the Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS), Cavtat, Croatia, Sept. 3-4 (2007)
[4] Krakatoa and Jessie: verification tools for Java and C programs, http://krakatoa.lri.fr/
[5] OpenJML jmlspecs, http://sourceforge.net/apps/trac/jmlspecs/wiki/OpenJml
[6] JML4c, http://www.cs.utep.edu/cheon/download/jml4c/index.php
[7] JMLRAC, http://www.eecs.ucf.edu/~leavens/JML2/docs/man/jmlrac.html
[8] Yoonsik Cheon, Gary T. Leavens: A runtime assertion checker for the Java Modeling Language (JML). In: Proceedings of the International Conference On Software Engineering Research and Practice (SERP 02), Las Vegas, Nevada, USA, June 24-27 (2002)
[9] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata: Extended Static Checking for Java. In Programming Language Design and Implementation (PLDI) 2002 forum. http://research.microsoft.com/en-us/um/people/leino/papers/krml103.pdf
[10] Why platform, http://why.lri.fr/
[11] C. Oriat. Jartege: A Tool for Random Generation of Unit Tests for Java Classes In Quality of Sofware Architectures and Software Quality, 2nd International Workshop of Software Quality SOQUA05, LNCS 3712, pages 242-256, Erfurt, Germany, Sept 2005.
[12] JML Tools Review & Evaluation, http://www.cs.colorado.edu/~bec/courses/csci5535-s10/slides/grosshans-lewis-prazen.2up.pdf
[13] JMLEclipse - jmlspecs, http://sourceforge.net/apps/trac/jmlspecs/wiki/JmlEclipse
[14] The OpenJML User Guide, page 13, http://jmlspecs.sourceforge.net/OpenJMLUserGuide.pdf
[15] Extensible Markup Language (XML), http://www.w3.org/XML/
[16] del Mar Gallardo, M., Martínez, J., Merino, P., Nuñez, P. and Pimentel, E.: PiXL: Applying XML Standards to Support the Integration of Analysis Tools for Protocols. In Science of Computer Programming Journal, Volume 65, Issue 1, 57-69 (March 2007). http://www.lcc.uma.es/~pedro/publications/pixl_gallardo.pdf