Properties Preservation during Transformation*

Daniela da Cruz[1], Jorge Sousa Pinto[1], and Pedro Rangel Henriques[1]

University of Minho - Department of Computer Science,
CCTC - Centro de Ciências e Tecnologias da Programação
Campus de Gualtar, 4715-057, Braga, Portugal
{danieladacruz,jsp,prh}@di.uminho.pt

Abstract. To prove the correctness of a program (written in a high level programming language) with respect to a specification (a set of proof obligations) does not assure the correctness of the machine code that the end-user will run after compilation and deployment phases. The code generated by the compiler should be verified again to guarantee that its correctness was preserved, and then that it can be executed in safety.

In the context of a Ph.D. work in the area of software analysis and transformation, we are looking for a suitable approach to prove that the software properties (validated at source level) are kept during translation.

In this position paper we introduce our architectural proposal, and discuss the platform and we are building for Java+JML on the top of Eclipse.

KEYWORDS: program verification and validation, proof carrying code, software analysis and transformation.

1 Introduction

The intention of this position paper is twofold. On one hand we want to bring to discussion our idea of a suitable platform for the generation and verification of *proof-carrying code* (PCC), that we will use to assure that a transformation process keeps a set of proof obligations. On the other hand, we want to make a note on the actual testbed that we are developing to experiment ideas in this context; that framework is based on Java as programming language and JML (Java modeling language) as the specification language used to set the invariants, pre- and post-conditions to be complied by the program. Only a subset of both languages is being considered.

Before the detailed discussion of our platform, we include in the rest of this section the basic concepts and definitions that constitute the theoretical background in this research area.

* This work is supported by a MAPi/FCT Ph.D. grant nu. SFRH/BD/33231/2007.

Verification Conditions. Our context is the following: the behavior of a program is specified through the use of pre-conditions and post-conditions. Programs are verified against their specifications using some program logic, typically some variant of Hoare Logic [4].

A popular architecture for a program verification system is based on two stages: a first component, called a *Verification Condition Generator* (VCGen) uses a program logic to generate a set of first-order proof obligations, the verification conditions, that are then passed on to a generic theorem prover. This is a more flexible approach than using a dedicated proof tool for reasoning directly with the program logic. The two-stage approach allows, for instance, to generate proof obligations for more than one theorem prover, and is more easily adaptable to modifications in the programming language, since the VCGen is in general a much simpler component than a theorem prover.

Proof-Carrying Code. PCC [7] is a program verification technology based on the generation of verification conditions from *compiled*, rather than source code. The novelty of the approach is that the application's executable code contains itself the formal proof that should be verified by the host system. It is appropriate in situations in which the client who executes the code has no access to the sources, or even if it has, does not trust the compiler to be correct. Also the client must be sure that the binary received is undamaged. So, it is not only a problem of the compilation process, but also the deployment of the executable code itself cannot be trusted. The executable code comes equipped with a proof certificate that testifies that the program conforms to its specification (annotated in the code). Both are generated by the code provider, but the key idea is that the client can generate the verification conditions and then mechanically check whether the given certificate successfully discharges those conditions. For this the client needs only trust relatively simple components – a low-level VCGen and a proof checker.

The standard way to generate the certificate is through *certifying compilation* [8]; however, Barthe and colleagues have proved that non-optimizing compilation of a subset of Java *preserves verification conditions* [2], i.e. a source-level VCGen and a bytecode-level VCGen will generate exactly the same proof obligations to be discharged. The certificate can be generated as the result of a standard bytecode-level program verification process. Notice that for the proofs at source-level the certificate is generated according to a source-level program verification process and so, different from the previous one. The present work takes precisely this point of view.

The Java Modelling Language. JML [6] is a standard annotation language for Java programs that supports the so-called *design by contract* approach to software development. It allows programs to be annotated with preconditions, postconditions, frame conditions, invariant properties, and model-level fields and methods.

Its strength come from using the same syntax for expressions as the source language (making specification tasks available to programmers in general), and

from being a federated effort: the design-by-contract approach to software development is supported by a number of tools for different tasks, including dynamic checkers, unit test generators, and even applications that can be used to help write specifications.

A number of existing program verification systems are based on JML as specification language; these are based on either (i) a translation of Java+JML programs into some simpler intermediate language, composed with a VCGen for this language [3, 1]; or (ii) a complex Hoare-style Logic for a subset of sequential Java [9, 5]. However, to the best of our knowledge, no direct definition of a VCGen for the full language has been published.

The Satisfiability Modulo Theories Library, SMT-Lib [10], is a library of benchmarks for *Satisfiability Modulo Theories*. SMT deals with the satisfiability of logical formulas with respect to combinations of background theories (expressed in classical first-order logic with equality) for which specialized decision procedures exist (such as, for instance, the theory of lists, of arrays, linear arithmetic, and so on). We elect SMT-Lib's concrete syntax to represent first-order proof obligations, with all the inherent advantages of using a standard language used by many theorem provers.

The Bytecode Modeling Language, BML [11], is a notation for formally specifying the behavior and interfaces of Java classes and methods at byte-code level in the form of annotations.

BML has basically the same syntax as JML with two exceptions:

1. Specifications are not written directly in the program code, they are added as special attributes to the byte code; and
2. the grammar for expressions only allows byte-code expressions.

2 Our Approach, the proposed Architecture

In [13], *Colby et al* proposed an approach based on certifying compilers; their approach, by compiling in a completely automatic way high-level source programs into optimized PCC binaries, was an evolution of the semi-automatic theorem-proving techniques presented earlier [8], and led to a more practical PCC technology.

As explained above, we adopt here Barthe's approach. So, the difference between Colby's approach and ours relies on the fact that we build up a *front-end* (the so-called Verification Condition Generator) between the specification languages and the proof systems (at both sides, the source code producer and the user of the executable code).

In concrete term we are building a PCC platform for Java programs, using JML and BML, respectively as source and machine level specification languages. The process is basically split into four major steps:

– Translation of JML annotations into a set of proof obligations (PO's) written in SMT language;

– Compilation of the Java programs into Java byte-code, using an existing compiler (javac);
– Translation of JML annotations into BML annotations that are *weaved* into the generated Java Bytecode using *code instrumentation techniques*, as will be referred below.
– Translation of BML annotations into proof obligations (PO's) also specified in SMT language.

The generic architecture depicted in Fig. 1 describes precisely the steps above; VCgen_s is the VCgen applied to the source code and VCGen_b is the VCgen applied to the byte-code.

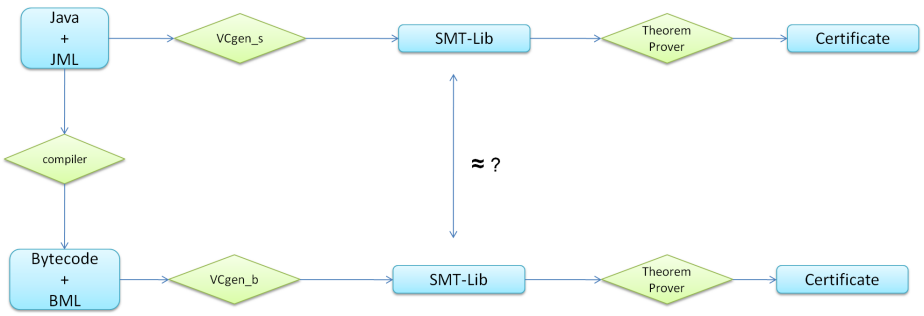

Fig. 1. The architecture of our PCC implementation

The theorem prover used in each side will be a first order logic (Hoare logic) capable of interpret the SMT language.

Below, an example is given for the first step referred above. Listing 1.1 shows a Java program with `max` method (that returns the maximum value of two integers) annotated with JML — the annotation defines a post-condition. Listing 1.2 shows the result of translating the annotations into the SMT language.

Listing 1.1. `Max` method annotated with JML

```
1  public class Operations {
2
3      /*@ ensures \result >= x && \result >= y &&
4        @ \forall integer z; z >= x && z >= y ==> z >= \result;
5        @*/
6      public static int max(int x, int y) {
7          if (x>y) return x;
8          else  return y;
9      }
   }
```

Listing 1.2. Proof obligations generated for the `max` example

```
(forall (?x c_unsorted)
(forall (?y c_unsorted)
(implies true
(and
(implies
(> (integer_of_int32 (c_sort int32 ?x))
  (integer_of_int32 (c_sort int32 ?y)))
(and (> (integer_of_int32 (c_sort int32 ?x))
  (integer_of_int32 (c_sort int32 ?x)))
(and (> (integer_of_int32 (c_sort int32 ?x))
  (integer_of_int32 (c_sort int32 ?y)))
(forall (?z c_unsorted)
(implies (and (> (integer_of_int32 (c_sort int32 ?z))
  (integer_of_int32 (c_sort int32 ?x)))
(> (integer_of_int32 (c_sort int32 ?z))
  (integer_of_int32 (c_sort int32 ?y))))
(> (integer_of_int32 (c_sort int32 ?z))
  (integer_of_int32 (c_sort int32 ?x)))))))
(implies
(not (> (integer_of_int32 (c_sort int32 ?x))
  (integer_of_int32 (c_sort int32 ?y))))
(and (> (integer_of_int32 (c_sort int32 ?y))
  (integer_of_int32 (c_sort int32 ?x)))
(and (> (integer_of_int32 (c_sort int32 ?y))
  (integer_of_int32 (c_sort int32 ?y)))
(forall (?z c_unsorted)
(implies (and (> (integer_of_int32 (c_sort int32 ?z))
  (integer_of_int32 (c_sort int32 ?x)))
(> (integer_of_int32 (c_sort int32 ?z))
  (integer_of_int32 (c_sort int32 ?y))))
(> (integer_of_int32 (c_sort int32 ?z))
  (integer_of_int32 (c_sort int32 ?y)))))))))))))
```

Code instrumentation is a mechanism that allows modules of programs to be completely rewritten at runtime. With the advent of virtual machines, this type of functionality is becoming more interesting because it allows the introduction of new functionality after an application has been deployed, easy implementation of aspect-oriented programming, performing security verifications, dynamic software upgrading, among others.

The value added by our proposal at this point is the *weaving* of the BML annotations into Java Bytecode files, after the compilation of Java source programs by a traditional compiler that does not need to be modified.

Two of the most important libraries in this area of code instrumentation for Java are BCEL [14], which provides a high-level API for manipulating Java Bytecode, and JOIE [12] that also allows Java objects to be instrumented.

The VCGen algorithms implemented will basically follow [2] – we will be able to prove, empirically, that the PO's generated at both levels are equivalent. In

a second phase of this project, we will extend the algorithms to richer subsets of Java, and our implementation will be an invaluable tool for identifying PO-preserving extensions.

3 Conclusion

In this article we have presented an approach to implementing a source-level PCC system; in our proposal we build a front-end (a Verification Condition Generator) between the specification languages (Java and JML) and the proof systems, that allows for the interactive construction of proofs of properties of programs at source-level, and then replicate this process at bytecode level.

At the present moment we are in the first phase of the process: the translation of the JML annotations into the proof obligations in SMT language. To implement this step we are using a well known compiler generator — ANTLR [15]. To have a full recognizer of the Java language with JML annotations we took an existing Java grammar and used a code instrumentation like technique. That is, starting from the original Java grammar we studied where the JML annotations could appear and introduced the corresponding fragments of JML grammar at each point.

The next step will be the development of a new Java-JML compiler that will reuse the Java compiler and translate each JML fragment into a BML fragment that, using again code instrumentation, will be weaved into the Java byte-code.

After that we will work on the low-level (or machine level) VCGen in order to finish the testbed that will allow us to make some experimentations in the area of theorem proving. For the limited subset of Java used in [2], proof obligations are guaranteed to be the same at both levels; we intend to further extend the VCGen algorithms to richer languages while still guaranteeing the preservation of proof obligations. Our testbed will help us in this task by allowing us to immediately recognize the features that violate this preservation.

The final objective of this research project will be to understand how this approach to PCC can be applied to a broader area that is under our present interest: generic program transformations.

References

1. Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.
2. Gilles Barthe, Tamara Rezk, and Ando Saabas. Proof obligations preserving compilation. In Theodosis Dimitrakos, Fabio Martinelli, Peter Y. A. Ryan, and Steve A. Schneider, editors, *Formal Aspects in Security and Trust*, volume 3866 of *Lecture Notes in Computer Science*, pages 112–126. Springer, 2005.

3. Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.
4. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.
5. Bart Jacobs and Erik Poll. A logic for the java modeling language jml. In Heinrich Hußmann, editor, *FASE*, volume 2029 of *Lecture Notes in Computer Science*, pages 284–299. Springer, 2001.
6. Gary T. Leavens, Clyde Ruby, K. Rustan, M. Leino, Erik Poll, and Bart Jacobs. Jml: notations and tools supporting detailed design in java. In *OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 105–106, New York, NY, USA, 2000. ACM.
7. G. C. Necula. Proof-carrying code. In *Proceedings of POPL'97*, pages 106–119. ACM Press, 1997.
8. George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *PLDI*, pages 333–344, 1998.
9. Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential java. In S. Doaitse Swierstra, editor, *ESOP*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 1999.
10. Clark Barrett, Silvio Ranise, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2008.
11. Lilian Burdy, Marieke Huisman, and Mariela Pavlova. Preliminary design of BML: A behavioral interface specification language for Java bytecode. In *Fundamental Approaches to Software Engineering (FASE 2007)*, volume 4422 of *Lecture Notes in Computer Science*, pages 215–229. Springer-Verlag, 2007.
12. Geoff Cohen, Jeff Chase, and David Kaminsky. Automatic program transformation with JOIE. In *1998 USENIX Annual Technical Symposium*, pages 167–178, 1998.
13. Christopher Colby, Peter Lee, and George C. Necula. A proof-carrying code architecture for java. In *Computer Aided Verification*, pages 557–560, 2000.
14. M. Dahm. Byte code engineering with the bcel api, 2001.
15. Terence Parr. *The Complete Antlr Reference Guide*. Pragmatic Bookshelf, 2007.