The Why/Krakatoa/Caduceus platform for deductive program verification

Jean-Christophe Filliâtre, Claude Marché

The Why/Krakatoa/Caduceus Platform for Deductive Program Verification*

Jean-Christophe Filliâtre[1,3] and Claude Marché[2,3]

[1] CNRS, Lab. de Recherche en Informatique, UMR 8623, Orsay, F-91405
[2] INRIA Futurs, ProVal, Parc Orsay Université, F-91893
[3] Univ Paris-Sud, Orsay, F-91405

Abstract. We present the Why/Krakatoa/Caduceus set of tools for deductive verification of Java and C source code.

1 Introduction

Why/Krakatoa/Caduceus is a set of tools for deductive verification of Java and C source code. In both cases, the requirements are specified as *annotations* in the source, in a special style of comments. For Java (and Java Card), these specifications are given in the *Java Modeling Language* [1] and are interpreted by the *Krakatoa* tool. For C, we designed our own specification language, largely inspired from JML. Those are interpreted by the *Caduceus* tool. The tools are available as open source software at `http://why.lri.fr/`.

The general approach is to generate *Verification Conditions* (VCs for short): logical formulas whose validity implies the soundness of the code with respect to the given specification. This includes automatically generated VCs to guarantee the absence of run-time errors: null pointer dereferencing, out-of-bounds array access, etc. Then the VCs can be discharged using one or several theorem provers. The main originality of this platform is that a large part is common to

* This research is partly supported by ANR RNTL grant "CAT"

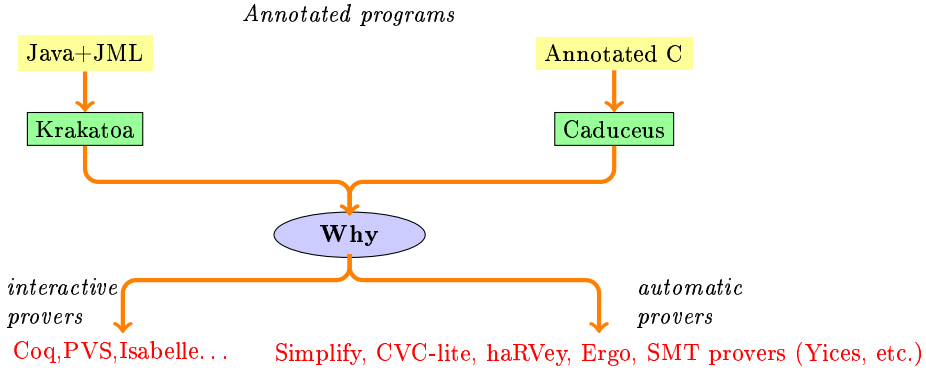

Fig. 1. Platform Architecture

```
typedef struct purse {
  int balance;
} purse;

/*@ requires \valid(p) && s >= 0
  @ assigns p->balance
  @ ensures p->balance ==
  @   \old(p->balance) + s
  @*/
void credit(purse *p,int s) {
  p->balance += s;
}
```

```
/*@ requires \valid(p1) && \valid(p2)
  @   && p1 != p2 && p1->balance == 0
  @ ensures p1->balance == 100
  @*/
void test(purse *p1, purse *p2) {
  credit(p1,100);
  p2->balance = 0;
  return p1->balance;
}
```

Fig. 2. Example of annotated C source code

C and Java. In particular there is a unique, stand-alone, VCs generator called Why, which is able to output VCs in the native syntax of many provers, either automatic or interactive ones. The overall architecture is presented on Figure 1.

Figure 2 shows a short example of annotated C code. Clauses `requires` introduces a precondition, `ensures` a postcondition, and `assigns` specifies the set of modified memory locations. `\valid` is a built-in predicate which specifies that the given pointer can be safely dereferenced, and `\old` denotes the value of the given expression at the function entry. Other kind of annotations include loop invariants and variants. VCs are generated modularly: when calling `credit` from `test`, only the specification of `credit` is used. To make this possible, the `assigns` clause is essential.

2 The Why Verification Condition Generator

The input syntax of Why is a specific language dedicated to program verification. As a programming language, it is a 'WHILE' language which (1) has limited side-effects (only *mutable variables* that cannot be aliased), (2) provides no built-in data type, (3) proposes basic control statements (assignment, `if`, `while`) but also exceptions (throwing and catching). A Why program is a set of functions, annotated with pre- and postconditions. Those are written in a general purpose specification language: polymorphic multi-sorted first-order logic with built-in equality and arithmetic. This logic can be used to introduce abstract data types, by declaring new sorts, function symbols, predicates and axioms.

The VCs generation is based on a Weakest Precondition calculus, incorporating exceptional postconditions and computation of effects over mutable variables [2]. Last but not least, Why provides a multi-prover output as shown on Figure 1. Actually Why can even by used only as a translator from first-order formulas to the syntax of those back-end provers. This translation includes a non-trivial removal of polymorphic sorts when the target logic does not support polymorphism [3].

3 Krakatoa and Caduceus

The common approach to Java and C source code is to translate them into Why programs. The Why specification language is then used both for the translation of input annotations and for the modeling of Java objects (resp. C pointers/structures). This model of the memory heap is defined by introducing abstract data types together with operations and an appropriate first-order axiomatization. Our heap memory models for C and Java both follow the principle of the Burstall-Bornat 'component-as-array' model [4]. Each Java object field (resp. C structure field) becomes a *Why mutable variable* containing a purely applicative map. This map is equipped with an access function *select* so that $select(f, p)$ denotes the field of the structure pointed-to by p; and an update function *store* so that $store(f, p, v)$ denotes a new map f' identical to f except at position p where it has value v. These two functions satisfy the so-called *theory of arrays*:

$$select(store(f, p, v), p) = p$$
$$p \neq p' \rightarrow select(store(f, p, v), p') = select(f, p')$$

In our example, the translation of the statement `p->balance += s;` from Figure 2 into the Why language is (1) $balance := store(balance, p, select(balance, p) + s)$. The translation of the postcondition `balance == \old(balance)+s` is $select(balance, p) = select(balance@, p) + s$ (where $x@$ denotes the old value of x in Why) and its weakest precondition through (1) is $select(store(balance, p, select(balance, p) + s), p) = select(balance, p) + s$ which is a first-order consequence of the theory of arrays.

4 Past and future work

The heap memory models are original, in particular with the handling of assigns clauses [5], and C pointer arithmetic [6]. Since these publications, many improvements have been made on the platform:

- Improved efficiency, including a separation analysis [**?**].
- More tools, including a graphical interface.
- Support for more provers, e.g. SMT provers (Yices, rv-sat, CVC3, etc.) and Ergo, with encodings of polymorphic sorts as seen above.
- Enhancements of specification languages both for C and Java: ghost variables, axiomatic models
- Specifically to Krakatoa, more support for Java Card source: transactions [7].
- Support for floating-point arithmetic [8].

Several case studies have been conducted: Java Card applets provided by Axalto [9] and Trusted Logic companies; the Schorr-Waite graph-marking algorithm, considered as a challenge for program verification [10]; some avionics embedded code provided by Dassault aviation company, which leaded to an original

analysis of memory separation [?]. Our intermediate first-order specification language was also used to design abstract models of programs [?].

To conclude, our platform is tailored to the proof of advanced behavioral specifications and proposes an original approach based on an intermediate first-order specification language. Its main characteristic is versatility: multi-prover output, multi-source input, on-the-fly generation of first-order models.

Future work includes the development of an integrated user environment. We are also designing an improved support for abstract modeling, by providing (UML-like) higher-level models and refinement. A key issue for the future is also the automatic generation of annotations. Long term perspective is to contribute to Grand Challenge 6 on Verified Software Repository: a key goal for us is to build libraries of verified software.

Acknowledgements Many people have been involved in the design and development of the platform and the case studies: R. Bardou, S. Boldo, V. Chaudhary, S. Conchon, E. Contejean, J.-F. Couchot, M. Dogguy, G. Dufay, N. Guenot, T. Hubert, J. Kanig, S. Lescuyer, Y. Moy, A. Oudot, C. Paulin, J. Roussel, N. Rousset, X. Urbain.

References

1. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer (2004)
2. Filliâtre, J.C.: Verification of Non-Functional Programs using Interpretations in Type Theory. Journal of Functional Programming **13**(4) (2003) 709–745
3. Lescuyer, S.: Codage de la logique du premier ordre polymorphe multi-sortée dans la logique sans sortes. Master's thesis, Master Parisien de Recherche en Informatique (2006)
4. Bornat, R.: Proving pointer programs in Hoare logic. In: Mathematics of Program Construction. (2000) 102–126
5. Marché, C., Paulin-Mohring, C.: Reasoning about Java programs with aliasing and frame conditions. In Hurd, J., Melham, T., eds.: 18th International Conference on Theorem Proving in Higher Order Logics. Volume 3603 of Lecture Notes in Computer Science., Springer (2005)
6. Filliâtre, J.C., Marché, C.: Multi-prover verification of C programs. In Davies, J., Schulte, W., Barnett, M., eds.: 6th International Conference on Formal Engineering Methods. Volume 3308 of Lecture Notes in Computer Science., Seattle, WA, USA, Springer (2004) 15–29
7. Marché, C., Rousset, N.: Verification of Java Card applets behavior with respect to transactions and card tears. In Hung, D.V., Pandya, P., eds.: 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM'06), Pune, India (2006)
8. Boldo, S., Filliâtre, J.C.: Formal Verification of Floating-Point Programs. In: 18th IEEE International Symposium on Computer Arithmetic, Montpellier, France (2007)

9. Jacobs, B., Marché, C., Rauch, N.: Formal verification of a commercial smart card applet with multiple tools. In: Algebraic Methodology and Software Technology. Volume 3116 of Lecture Notes in Computer Science., Stirling, UK, Springer (2004)
10. Hubert, T., Marché, C.: A case study of C source code verification: the Schorr-Waite algorithm. In Aichernig, B.K., Beckert, B., eds.: 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM'05), Koblenz, Germany (2005)
11. Hubert, T., Marché, C.: Separation analysis for deductive verification. Technical report, Université Paris XI (2007) `http://www.lri.fr/~marche/separation.pdf`.