Deductive Verification of Java programs with Algebraic Modeling and Multi-Prover Backend: the Why/Krakatoa platform

Claude Marché Christine Paulin
Jean-Christophe Filliâtre
Nicolas Rousset Xavier Urbain
ProVal project - http://proval.lri.fr
INRIA-Futurs & Université Paris-Sud 11
Orsay, France

January 20th, 2007

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
INRIA FUTURS
LABORATOIRE DE RECHERCHE EN INFORMATIQUE

Outline

Outline

Context

- ProVal research group develops tools for *Deduction Verification* of Java and C source code
- Requirements: specified as *annotations* in the source
- For Java (Card): specifications given in *JML (Java Modeling Language)* $\rightsquigarrow$ *Krakatoa* tool
- For C: home-made specification language $\rightsquigarrow$ *Caduceus* tool
- Generation of *Verification Conditions*, to be discharged by *theorem provers*
- Originality: *common platform* for C and Java

JML toy example

- JML class invariants
- JML method behaviors: pre- and post-conditions

```
class Purse {

    int balance;
    //@ invariant balance >= 0;

    /*@ normal_behavior
      @   requires s >= 0;
      @   assigns balance;
      @   ensures balance == \old(balance)+s;
      @*/
    public void credit(int s) {
        balance += s;
    }
```

Toy example (cont.)

- JML exceptional behaviors

```
/*@ behavior
  @   requires s >= 0;
  @   assigns balance;
  @   ensures s <= \old(balance) &&
  @     balance == \old(balance) - s;
  @   signals (NoCreditException)
  @      s > \old(balance) &&
  @      balance == \old(balance);
  @*/
public void withdraw(int s) throws NoCreditException {
    if (balance >= s) balance -= s;
    else throw new NoCreditException();
}
```

JML tools

- *Runtime assertion checking*: JML RAC
- *Static verification*:
  - ESC/Java
  - Several others: LOOP, Jack, KeY, Jive, Bogor. . . Krakatoa
    - Common goal: prove advanced functional behaviors
    - Why so many tools ? hard problems, many challenges: http://www.cs.ru.nl/~woj/esfws06/
- Other tools: testing, symbolic execution. . .

Outline

Demo: toy example (cont.)

- A buggy example

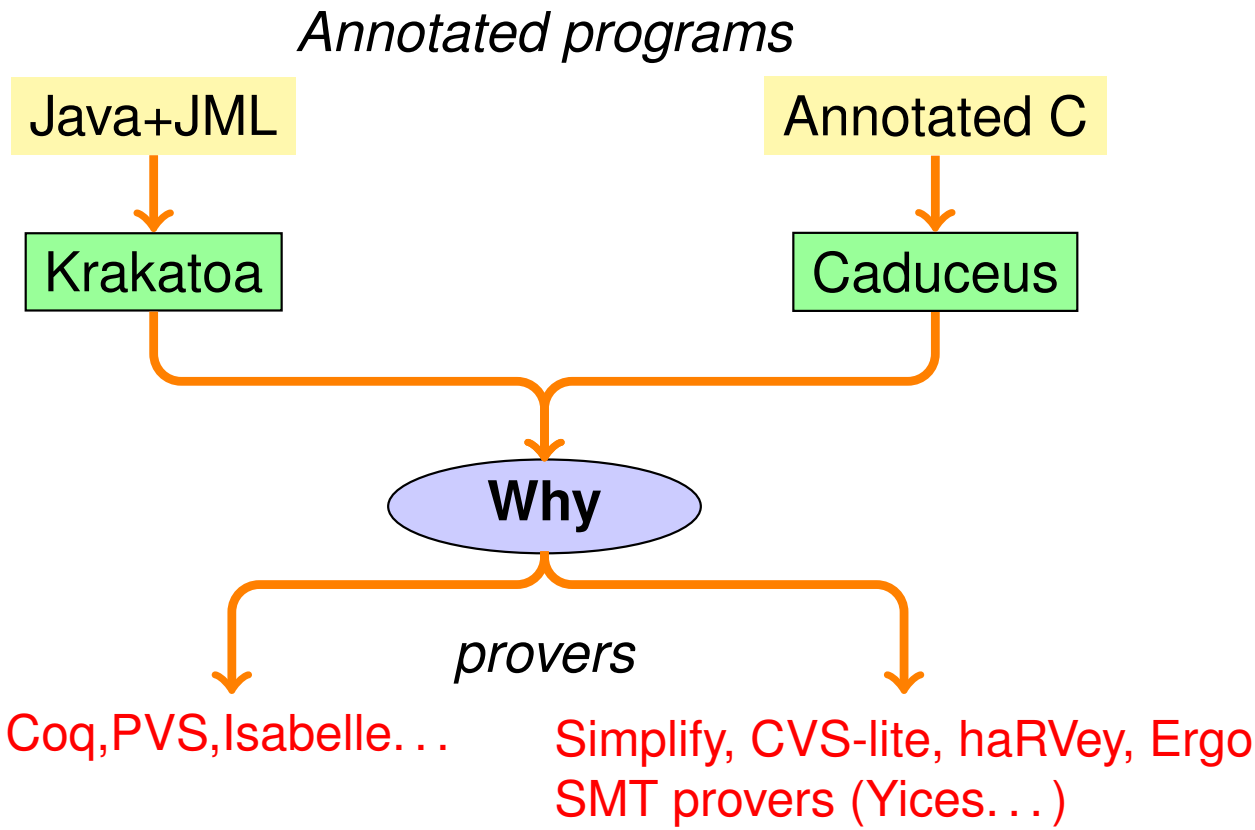
```
/*@ normal_behavior
  @   requires p1.balance == 100;
  @   ensures \result == 150;
  @*/
public static int test(Purse p1, Purse p2) {
    p1.credit(50);
    p2.withdraw(100);
    return p1.balance;
}
```

Demo

Remarks

- Krakatoa generates VCs for both
  - *Safety properties*: no NullPointerException, no ArrayIndexOutOfBounds, no DivisionByZero
  - Method calls: *precondition* is satisfied
  - *Methods post-conditions* are valid
  - *Class invariants* are preserved (beware: challenging issues)
- *Modular Approach*:
  - for each method call: only its specification is seen

Platform Architecture



Platform characteristics

- Shared *intermediate language*: Why language
- Only one *VCG (Verification Condition Generator)*: Why tool
- Several provers as output:
  - allows both
    - *automatic proving* and
    - *interactive proof contruction*

  for discharging VCs

Why tool

- Multi-prover output
- Why language :
  - programming language: a WHILE language, tailored to VC generation generation, with
    - limited side-effects: only *mutable variables*
    - *no data types*
    - basic control statements + `throw, try/catch`
  - program = set of functions, annotated with pre- and post-conditions
  - specification language: multi-sorted (polymorphic) *first-order logic*, with *built-in arithmetic*
- VC generation based on a *Weakest Precondition calculus*, incorporating *exceptional post-conditions*, and computation of *effects over mutable variables*.

Why as intermediate language

- Common approach to Java and C:

  translation into Why programs

- Why specification language used both for
  - translation of input annotations
  - modeling Java objects (resp. C pointers/structures) and heap memory.
- Modeling in Why: *algebraic specifications*
  - introducing functions and predicates
  - stating axioms

Heap memory model

Principle: Burstall-Bornat 'component-as-array' model

Java	Why
`balance += s;`	balance := *upd*(balance,this, *acc*(balance,this)+s)

- Each *Java field* becomes a *Why mutable variable* of type *'functional array'*
- $acc(f, o)$ denotes f at index o $\rightarrow$ *encodes o.f*
- $upd(f, o, v)$ denotes *functional update*
- Theory of arrays:

$$\begin{aligned} acc(upd(f, o, v), o) &= v \\ o \neq o' \rightarrow acc(upd(f, o, v), o') &= acc(f, o') \end{aligned}$$

Heap memory model in Krakatoa

- Similar encoding for Java arrays
- Objects hierarchy modeled by a predicate *instanceof* with axioms.
-
- A theory for modeling *assigns* clauses [TPHOLs'05]
- Approximately 500 lines of Why specifications, + additional axioms generated on-the-fly for each Java program

Java: contributions

- Krakatoa tool publicly available
- A specific modeling of Java/JML, in particular for `assigns` clauses [TPHOLs'05]
- Java Card transactions [SEFM'06]
  - on-the-fly generation of interpretation of *beginTransaction()*, *commitTransaction()*...
- Case studies:
  - PSE applet provided by Axalto [AMAST'04]
  - Demoney Applet provided by Trusted Logic

C programs: contributions

- Caduceus tool publicly available
- An original modeling of heap memory and pointer arithmetic [ICFEM'04]
- Original support of floating-point programs [ARITH'07]
- Case studies:
  - Schorr-Waite graph-marking algorithm [SEFM'05],
  - Avionics embedded code provided by Dassault aviation company
    $\rightsquigarrow$ A original analysis of memory separation [submitted]

Outline

Design choices

- Krakatoa, 2003:
  - ad-hoc interpretation of *pure* methods
- Underlying Why logic:
  - multi-sorted first order logic
  - one may declare sorts, logical functions, predicates, axioms.
- Idea:
  - use this logic for describing models of programs
  - $\rightsquigarrow$ algebraic specifications of models

Design choices

- Caduceus, 2004:
  - allow first-order modeling at C source level
  - used for:
    - linked-list in-place reversal in C [Filliâtre & Marché, ICFEM 2004]
    - Schorr-Waite graph traversal in C [Hubert & Marché, SEFM 2005]
- Krakatoa, 2006:
  - allow first-order modeling similarly
  - but JML models are OO, not algebraic
  - so *Krakatoa now diverges from JML*: allows algebraic models (recent work, still in progress)

Example

- General-purpose class for finite sets of integers
- Basic interface :

```
class IntSet {

    // checks whether n belongs to this
    public boolean mem(int n);

    // adds n to this
    public void add(int n);
}
```

Algebraic model (1)

- Sort, functions, predicates:

```
/*@ logic type intset;
  @ logic intset emptyset();
  @ logic intset singleton(int n);
  @ logic intset union(intset s1, intset s2);
  @ predicate in(int n,intset s);
  @*/
```

Algebraic model (2)

- Axiomatization:

```
/*@ axiom in_empty :
  @  (\forall int n; !in(n,emptyset()));
  @
  @ axiom in_singleton :
  @  (\forall int n,k;
  @   in(n,singleton(k)) <==> n==k;
  @
  @ axiom in_union :
  @  (\forall int n; \forall intset s1,s2 ;
  @   in(n,union(s1,s2)) <==>
  @       in(n,s1) || in(n,s2);
  @
  @ axiom intset_ext :
  @  (\forall intset s1,s2 ;
  @   (\forall int n ; in(n,s1) <==> in(n,s2))
  @   ==> s1==s2) ;
  @*/
```

Specification of IntSet class

Introducing a *model* field:

```
class IntSet {
 //@ model intset my_model;
```

Methods' behaviors:

```
 //@ ensures \result <==> in(n,my_model)) ;
 public boolean mem(int n);

 /*@ assigns my_model;
   @ model ensures
   @  my_model == union(\old(my_model),singleton(n)) ;
   @*/
 public void add(int n);
}
```

An implementation

By a sorted array

```
class IntSet {

  int size;
  int t[];

  /*@ invariant
    @   t != null &&
    @   0 <= size && size <= t.length &&
    @   (\forall int i,j;
    @      0 <= i && i <= j && j < size;
    @      t[i] <= t[j]);
    @*/

}
```

Relating models and implementation

```
/*@ predicate IntSet_models(intset s, IntSet this) {
  @    this != null &&
  @     array_models(s,this.t,0,this.size-1)
  @ }
  @
  @ predicate array_models(intset s, int t[],
  @                        int i, int j) {
  @   t != null &&
  @   0 <= i && j < t.length &&
  @   (\forall int n; in(n,s) <==>
  @      (\exists int k;
  @         i <= k && k <= j ; n==t[k]))
  @ }
  @*/

//@ invariant IntSet_models(my_model,this);
```

Demo

Refinement property

- Proof of `add` post-condition: closure of the diagram

$$\begin{array}{lccc} \textit{abstract level}: & \text{my\_model} & \xrightarrow{add} & \text{my\_model'} \\ & \updownarrow I & & \updownarrow I \\ \textit{concrete level}: & \text{t,size} & \xrightarrow{add} & \text{t',size'} \end{array}$$

- It is a *refinement* property.
- Comparison with the B approach in progress.

Outline

Summary

- Prototype tools, available for experimentation (open source):

  `http://krakatoa.lri.fr`
  `http://why.lri.fr`

- Tailored to the proof of advanced behaviors
- Originality: intermediate first-order specification language for:
  - modeling Java memory heap
  - modeling Java Card transactions
  - user-defined abstract models of programs
- *Versatility*: multi-prover output, multi-source input, allows on-the-fly generation of model

Perspectives

- GUI in the Eclipse framework
- Improved support for abstract modeling: refinement, higher-level models (UML)
- Automatic annotation generation
- Analysis of memory separation

Long term:

- Contribute to *Grand Challenge 6, Verified Software Repository*
- Key goal: *libraries* of verified software