

Formal Verification of OO Programs: the Krakatoa Approach

Claude Marché

INRIA, Orsay, France

FVOOS Winter School, Viinistu, Estonia, January 25-29,
2009



Outline

1 Krakatoa overview

2 Why VC generator

3 Krakatoa to Why

Outline

1 Krakatoa overview

2 Why VC generator

3 Krakatoa to Why

Krakatoa

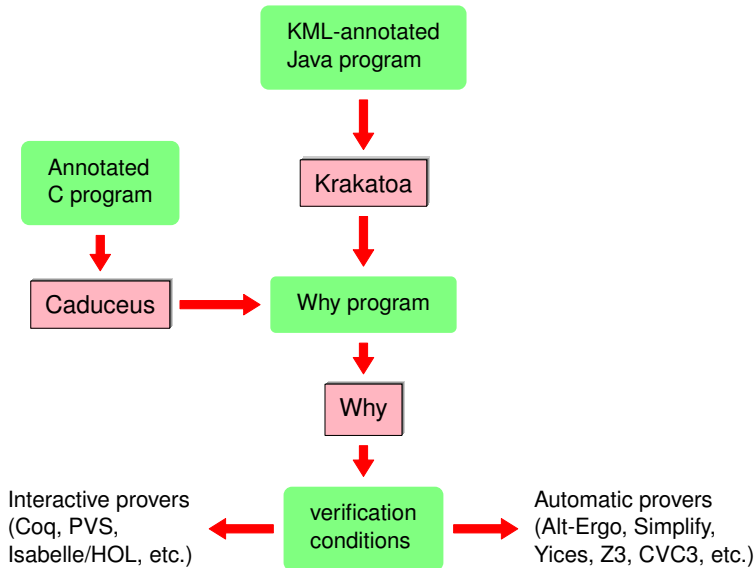
Krakatoa is a deductive verification tool for Java(Card) programs

- Specifications as annotations
- Verification Conditions generated using a weakest precondition calculus (Why VCgen)

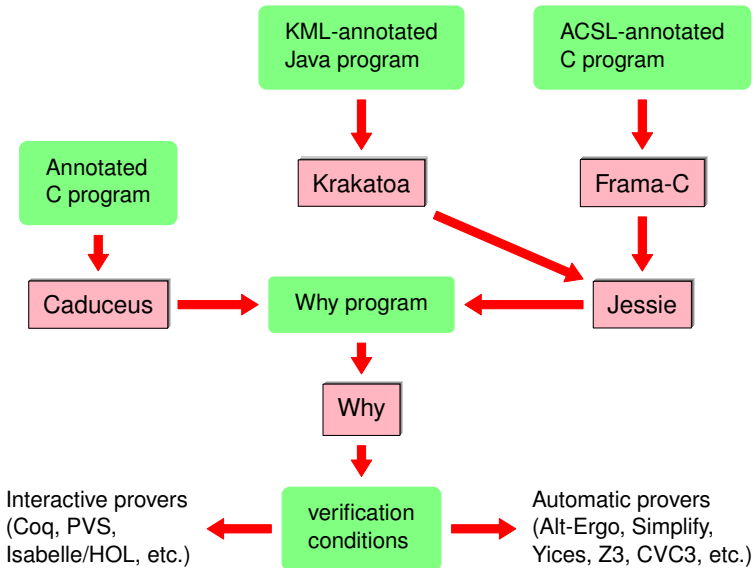
Krakatoa is just one tool of the Why platform which has

- several languages input
- several prover output

Architecture (2004)



Architecture (2008)



First example: isqrt

```
public static int isqrt(int x) {  
    int count = 0, sum = 1;  
    while (sum <= x) {  
        count++;  
        sum = sum + 2*count+1;  
    }  
    return count;  
}
```

DEMO

KML: Krakatoa Modeling Language

- Largely inspired from the JML (*Java Modeling Language*)
- Shares many features with the ACSL (*ANSI/ISO C Specification Language*), the specification language of Frama-C, e.g.:
 - Contracts with named behaviors
 - Algebraic-style specifications

Method Contracts

Simple form:

```
/*@ requires  $R$ ;  
   @ assigns  $A$ ;  
   @ ensures  $E$ ;  
   @*/
```

- R : precondition, supposed to hold in the *pre-state*
Must be checked valid *by the caller*
- E : postcondition, supposed to hold in the *post-state*
- A : set of possibly modified memory locations

Special keywords in E :

- `\result`: denotes the returned value
- `\old(e)`: denotes the value of e in the pre-state.

Method Contracts

General form of contracts:

```
/*@ requires  $R$ ;  
  @ behavior  $b_1$ :  
  @ ...  
  @ behavior  $b_2$ :  
  @ ...  
@*/
```

b_1, b_2, \dots is a set of named behaviors.

Normal behaviors

A *normal* behavior has the form:

```
/ * @ ...  
@ behavior b:  
@   assumes A;  
@   assigns L;  
@   ensures E;  
@ */
```

It states that:

- In the post-state: $\text{old}(A) \implies E$ holds
- If A holds in the pre-state: each memory location not in L is unmodified.

Exceptional behaviors

An *exceptional* behavior has the form

```
/*@ ...  
  @ behavior b:  
  @   assumes A;  
  @   assigns L;  
  @   signals (Exc x) E;  
  @*/
```

States the same properties as a normal behavior, but in the case the method returns abruptly with exception *Exc*.

Class invariants

A *class invariant* is declared at the level of class fields, with a name:

```
/*@ invariant id: e;
```

- **Warning:** Krakatoa handles those invariants just as “macros” for pre- and post-conditions of methods

Example: Toy Electronic Purse

```
public class Purse {  
  
    private int balance;  
    /*@ invariant balance_positive:  
        @ balance > 0;  
    @*/  
  
    /*@ requires amount > 0;  
        @ assigns balance;  
        @ ensures balance == amount;  
    @*/  
    public Purse(int amount) {  
        balance = amount;  
    }  
}
```

Example: Toy Electronic Purse

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

Example: Toy Electronic Purse

```
/*@ requires s >= 0;
   @ assigns balance;
   @ ensures s < \old(balance) &&
   @   balance == \old(balance) - s;
   @ behavior amount_too_large:
   @   assigns \nothing;
   @   signals (NoCreditException)
   @     s >= \old(balance) ;
   @*/

public void withdraw(int s)
    throws NoCreditException {
    if (s < balance)
        balance = balance - s;
    else
        throw new NoCreditException();
}
```


Statement Annotations

- Assertions:

```
//@ assert p;
```

\rightsquigarrow p holds at this program point.

- Loop annotations:

```
/*@ loop_invariant I  
   @ for b: loop_invariant Ib;  
   @ loop_variant V;  
   @*/
```

- I holds at loop entry
- I preserved by any iteration of the loop body
- I_b , same as I , but under `assumes A` of behavior b .
- V integer expression which decrease at each loop iteration, and remains non-negative

Logic definitions

- KML does not allow *pure* methods in annotations.
- But allows to declare new logic functions and predicates:

```
//@ logic  $\tau$  id( $\tau_1$   $x_1, \dots, \tau_n$   $x_n$ ) =  $e$  ;
```

```
//@ predicate id( $\tau_1$   $x_1, \dots, \tau_n$   $x_n$ ) =  $p$  ;
```

- Types τ and τ_i : Java types or purely logic types: integer, real, etc.
- Lemmas:

```
//@ lemma id:  $p$ ;
```

Example: Sorting Algorithm

- We want to express the behavior: resulting array is in increasing order
- We declare a predicate `sorted` for that purpose

DEMO

Hybrid predicates

- `sorted` is a *hybrid* predicate: it depends on some memory state.
- More generally, we can have predicates depending on *several* memory states, by attaching several labels.

```
/*@ logic  $\tau$  id\{L_1, \dots, L_k\}(\tau_1 x_1, \dots, \tau_n x_n) =
```

@ **e**;

@*/

```
/*@ predicate id\{L_1, \dots, L_k\}(\tau_1 x_1, \dots, \tau_n x_n) =
```

@ **p**;

@*/

Inductive predicates

```

/*@ inductive  $P(\tau_1 x_1, \dots, \tau_n x_n)$  {
  @ case  $C_1$  :  $p_1 i$ ;
  . . .
  @ case  $C_k$  :  $p_k i$ ;
  @ }
/*@ */

```

- P is the least fixpoint of the cases
- each proposition p_i must have the form

$$\forall y_1, \dots, y_m, \\ h_1 \implies \dots \implies h_l \implies P(t_1, \dots, t_n)$$

where P occurs only positively in hypotheses h_1, \dots, h_l .

Example: Sorting Algorithm

- New behavior: the resulting array is a permutation of the original one
- We declare `permut` as an inductive, two-state predicate

DEMO

Axiomatic blocks

```
axiomatic Power {  
  logic real lpower(real x, integer n);  
  axiom power_zero:  
    \forall real x; lpower(x,0) == 1.0;  
  axiom power_one:  
    \forall real x; lpower(x,1) == x;  
  axiom power_sum: ...  
}
```

- type names
- profiles for predicates and logic functions
- axioms they satisfy

Warning! no guarantee of consistency

Outline

1 Krakatoa overview

2 Why VC generator

3 Krakatoa to Why

Why: a Verification Condition Generator

Why is a *verification condition generator* for a language with

- variables containing pure values, no alias
(~ Hoare-logic language)
- basic control structures: tests, (infinite) loops
- exceptions
- (possibly recursive) functions
- polymorphic first-order logic with equality and arithmetic

Why is similar to Boogie (SPEC# project)

Why is also responsible for *translating* verification conditions
to the *native logics* of all provers

Basic Idea

makes program verification

- prover-independent but prover-aware
- language-independent

so that we can use it to verify C, Java, etc. programs with
HOL provers but also with FO decision procedures

The essence of Hoare logic: assignment rule

$$\{ P[x \leftarrow E] \} x := E \{ P \}$$

- 1 absence of aliasing
- 2 side-effects free *E* *shared* between program and logic

Data types

Any purely applicative data type from the logic can be used in programs

Example: a data type `int` for integers with constants `0`, `1`, etc. and operations `+`, `*`, etc.

The pure expression `1+2` belongs to both programs and logic

Only one data structure:

- *reference* (mutable variable) containing *only pure values*
- no alias allowed between two different references

ML syntax

No distinction between expressions and statements

⇒ less constructs

⇒ less rules

| | |
|-----------------|--|
| dereference | $!x$ |
| assignment | $x := e$ |
| local variable | $\text{let } x = e_1 \text{ in } e_2$ |
| local reference | $\text{let } x = \text{ref } e_1 \text{ in } e_2$ |
| conditional | $\text{if } e_1 \text{ then } e_2 \text{ else } e_3$ |
| loop | $\text{loop } e$ |

syntactic sugar, e.g.:

sequence $e_1; e_2 \equiv \text{let } _ = e_1 \text{ in } e_2$

Annotations

Assertions, post-conditions:

- `assert { p }; e`
- `e { p }`

Examples:

- `assert {! $x > 0$ }; $1/!x$`
- `! $x + 1$ {result =! $x + 1$ }`
- `$x := !x + 1$ {! $x = x@ + 1$ }`

Loop invariant and variant:

- `loop e {invariant p variant t }`

Functions

A function declaration introduces a *precondition*

- $\text{fun } (x : \tau) \rightarrow \{p\} e$

Example:

$$\text{fun } (x : \text{int ref}) \rightarrow \{!x > 0\} x := !x - 1 \{!x \geq 0\}$$

Function call:

- $e_1 e_2$: e_1 applied to e_2
- sugar: $(e e_1 e_2 \cdots e_n)$: e applied to e_1, \dots, e_n

Exceptions

new constructs

- $\text{raise } (E\ e) : \tau$
- $\text{try } e_1 \text{ with } E\ x \rightarrow e_2 \text{ end}$

The notion of postcondition is extended

```
if  $x < 0$  then raise Negative else sqrt  $x$ 
{ result  $\geq 0$  | Negative  $\Rightarrow x < 0$  }
```


Example: isqrt

exception Break of unit

```
let isqrt= fun (x:int) ->
  { x >= 0 }
  let count = ref 0 in
  let sum = ref 1 in
  try
    loop
      if sum > x then raise Break;
      count := !count + 1;
      sum := !sum + 2 * !count + 1;
      { invariant count >= 0
        and x >= sqr(count)
        and sum = sqr(count+1)
        variant x - sum }
  with Break -> void;
  !count
  { result >=0 and sqr(result) <= x
    and x < sqr(result + 1) }
```

Modularity

A function declaration extends the ML function type with a *precondition*, an *effect* and a *postcondition*

$$f : x : \tau_1 \rightarrow \left\{ \begin{array}{l} p \\ \tau_2 \\ \text{reads } x_1, \dots, x_n \\ \text{writes } y_1, \dots, y_m \\ \text{raises } E_1, \dots, E_k \\ \{q \mid E_1 \Rightarrow q, \dots, E_k \Rightarrow q\} \end{array} \right\}$$

Examples:

swap : $x : \text{int ref} \rightarrow y : \text{int ref} \rightarrow$
 $\{ \} \text{ unit writes } x, y \{ x = y@ \wedge y = x@ \}$

div : $x : \text{int} \rightarrow y : \text{int} \rightarrow$
 $\{ \} \text{ int raises Negative } \{ \dots \mid \text{Negative} \Rightarrow y = 0 \}$

Typing judgment

$$\Gamma \vdash e : (\tau, \epsilon)$$

Rules given in the lecture notes

Important: *typing excludes aliasing*:

```
let incr = fun(x : int ref)(y : int ref) →  
    {x :=!x + 1; y :=!y + 1 {x = x@ + 1 ∧ y = y@ + 1}}  
let _ = (incr z z)
```

↪ **error**: application creates an alias

Semantics

Call-by-value semantics, with left to right evaluation

Big-step operational semantics given in the lecture notes

Proof Rules: Weakest Preconditions

- $wp(e, q)$: the *weakest precondition* for program e and postcondition q
- *Property*: If $wp(e, q)$ holds, then e terminates and q holds at the end of execution (and all inner annotations are verified)
- The correctness of an annotated program e is thus $wp(e, \text{True})$

Definition of $wp(e, q)$

We actually define $wp(e, q; r)$ where

- q is the “normal” postcondition
- $r \equiv E_1 \Rightarrow q_1; \dots; E_n \Rightarrow q_n$ is the set of “exceptional” post.

WP rules: excerpt

- pure expressions:

$$wp(u, q; r) = q[result \leftarrow u]$$

- assignment:

$$wp(x := e, q; r) = wp(e, q[! x \leftarrow result]; r)$$

- if statement:

$$wp(\text{if } e_1 \text{ then } e_2 \text{ else } e_3, q; r) = \\ wp(e_1, \text{if } result \text{ then } wp(e_2, q; r) \text{ else } wp(e_3, q; r);$$

Note: this is the “natural” WP rule for if. There exists an “efficient” one [Leino03]

Exceptions

- if e is pure:

$$wp(\text{raise } (E e) : \tau, q; r; E \Rightarrow r_E) = r_E$$

- in general:

$$wp(\text{raise } (E e) : \tau, q; r; E \Rightarrow r_E) = wp(e, r_E; r)$$

- exception catching:

$$wp(\text{try } e_1 \text{ with } E x \rightarrow e_2 \text{ end}, q; r) = \\ wp(e_1, q; E \Rightarrow wp(e_2, q; r)[x \leftarrow \text{result}]; r)$$

Annotations

$$wp(\text{assert } \{p\}; e, q; r) = p \wedge wp(e, q; r)$$

$$wp(e \{q'; r'\}, q; r) = wp(e, q' \wedge q; r' \wedge r)$$

Loops

- loop without variant:

$$wp(\text{loop } e \{ \text{invariant } p \}, q; r) = \\ p \wedge \forall \omega. p \Rightarrow wp(e, p; r)$$

where ω = the variables (possibly) modified by e

- loop with variant: see lecture notes.

Functions

- Function call: first simplified as

$$e_1 \ e_2 \equiv \text{let } x_1 = e_1 \text{ in let } x_2 = e_2 \text{ in } x_1 \ x_2$$

- then, assuming $x_1 : (x : \tau) \rightarrow \{p'\} \tau' \in \{q'\}$

$$\begin{aligned} wp(x_1 \ x_2, q) = \\ p'[x \leftarrow x_2] \wedge \forall \omega. \forall \text{result}. (q'[x \leftarrow x_2] \Rightarrow q)[t@ \leftarrow t] \end{aligned}$$

Outline

1 Krakatoa overview

2 Why VC generator

3 **Krakatoa to Why**

Krakatoa to Why

Delayed to lecture 2

End of lecture 1

- **Krakatoa installation, exercises:**
`http://krakatoa.lri.fr/ws`
- **Try to solve exercises before the lab session**
- **Challenge: solve the “mystery code” exercise**