The Krakatoa Verification Tool

for JAVA programs

Tutorial and Reference Manual

Version 1.11

Claude Marché

March 17, 2008
INRIA Team-Project *Proval* http://proval.lri.fr
INRIA Saclay - Île-de-France & LRI, CNRS UMR 8623
4, rue Jacques Monod, 91893 Orsay cedex, France

Contents

1 Introduction **5**

2 Tutorial **7**
- 2.1 The basics of the methodology . . . 7
- 2.2 Loop invariants . . . 8
- 2.3 Array accesses . . . 12
- 2.4 Logic predicates . . . 13
- 2.5 Array updates . . . 14
- 2.6 Objects and constructors . . . 14
- 2.7 Calling subprograms, `assigns` clauses . . . 15
- 2.8 Programs with exceptions . . . 16
- 2.9 The Dutch National Flag problem . . . 17
- 2.10 Ghost variables . . . 18

3 Specification Language: Reference **21**
- 3.1 Logic expressions . . . 21
  - 3.1.1 Operator precedence . . . 21
  - 3.1.2 Semantics . . . 23
  - 3.1.3 Typing . . . 23
  - 3.1.4 Integer arithmetic and machine integers . . . 24
  - 3.1.5 Real numbers and floating point numbers . . . 25
- 3.2 Simple contracts . . . 25
- 3.3 Behavior clauses . . . 26
- 3.4 Code annotations . . . 27
  - 3.4.1 Assertions . . . 27
  - 3.4.2 Loop annotations . . . 27
- 3.5 Data invariants . . . 28
- 3.6 Advanced modeling language . . . 28
  - 3.6.1 Logic definitions . . . 28
  - 3.6.2 Hybrid predicates . . . 28
  - 3.6.3 Abstract data types . . . 28
- 3.7 Termination . . . 28
  - 3.7.1 Loop variants . . . 28
- 3.8 Ghost variables . . . 28

4 Appendices **29**
- 4.1 Requirements . . . 29
- 4.2 Installation procedure . . . 29
  - 4.2.1 From the sources . . . 29
  - 4.2.2 Binaries . . . 29

4.3 Summary of features and known limitations 29
4.4 Contacts 30

Bibliography **31**

Index **33**

Chapter 1

Introduction

Krakatoa is a tool for certification of Java programs, annotated using the Java Modeling Language [4] (JML for short), using the Why [1] tool for generating proof obligations.

This version 1.11 of Krakatoa is a major rewriting of the version 0.x family. Major changes have occured including changes in the syntax of annotations.

Chapter 2 is a tutorial to introduce the user step by step to the use of Krakatoa.

Chapter 3 is a reference manual where all options of the tool are described, and also the modeling of Java objects and Java memory heap, that you may encounter if you discharge proofs interactively, e.g. using Coq.

In the appendix you will find various additional informations, including the requirements, a summary of known limitations, and how to get help.

Chapter 2

Tutorial

Before going to the first lesson, we recommend to start by creating a new directory `tutorial`, and code each example in that directory.

2.1 The basics of the methodology

We start by a very simple static method, that computes the maximum of two integers. In the `tutorial` sub-directory, write the following JAVA program into a new file `Lesson1.java`.

```
public class Lesson1 {

    /*@ ensures \result >= x && \result >= y &&
      @    \forall integer z; z >= x && z >= y ==> z >= \result;
      @*/
    public static int max(int x, int y) {
        if (x>y) return x; else return x;
    }
}
```

The comment starting with `/*@` just before the method `max` is a *contract* specifying a intended behavior for `max`. The `ensures` clause introduce a *postcondition*, which is a formula supposed to hold at end of execution of that method, for any value of its arguments. In that formula,
`result` denotes the returned value for that method, hence that formula means three things: (i) the result is greater than or equal to x, (ii) the result is also greater than or equal to y, and (iii) the result is the least of all integers both greater than `x` and `y`.

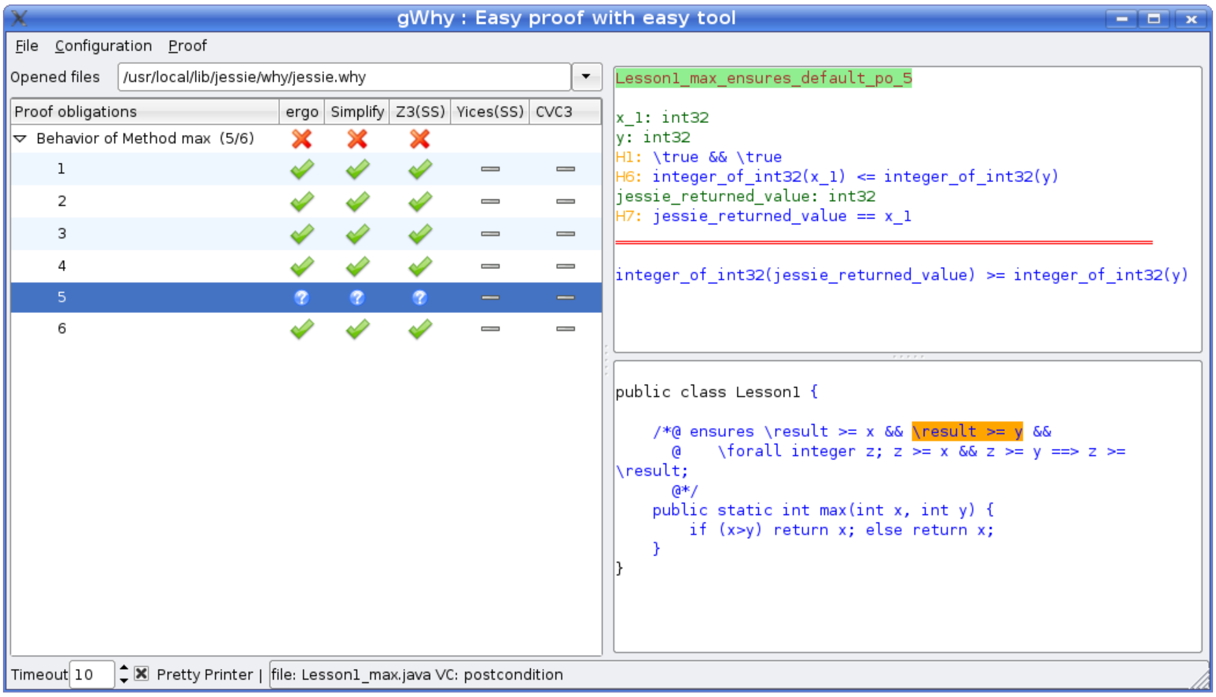
Our aim is to verify that the body of method `max` is a correct implementation, in the sense that it satisfies the contract given. Indeed, we intentionally made a mistake: the second `return` should return `y` instead of `x`.

Running the verification process can be done by executing the following command, in the directory `tutorial`:

```
gwhy Lesson1.java
```

The command will read the given file, and generate a set of logic formulas call *verification conditions* (abbreviated as VC), which express the validity of the program. The generated formulas are then displayed into the common graphical VC viewer of the Why platform, as shown in Figure 2.1

The left sub-window displays a table where each row corresponds to a VC for method `max`. Each column corresponds to a theorem prover. Running a given prover on the VCs can be acheived by clicking



/usr/local/lib/jessie/why/jessie.why

Proof obligations	ergo	Simplify	Z3(SS)	Yices(SS)	CVC3
Behavior of Method max (5/6)	✗	✗	✗		
1	✓	✓	✓	—	—
2	✓	✓	✓	—	—
3	✓	✓	✓	—	—
4	✓	✓	✓	—	—
5	?	?	?	—	—
6	✓	✓	✓	—	—

```
Lesson1_max_ensures_default_po_5

x_1: int32
y: int32
H1: \true && \true
H6: integer_of_int32(x_1) <= integer_of_int32(y)
jessie_returned_value: int32
H7: jessie_returned_value == x_1

integer_of_int32(jessie_returned_value) >= integer_of_int32(y)
```

```
public class Lesson1 {

    /*@ ensures \result >= x && \result >= y &&
      @    \forall integer z; z >= x && z >= y ==> z >=
\result;
      @*/
    public static int max(int x, int y) {
        if (x>y) return x; else return x;
    }
}
```

Timeout 10 ☒ Pretty Printer | file: Lesson1_max.java VC: postcondition

Figure 2.1: Verification conditions displayed in GWhy

in its name in the first row. On the figure, the first three provers have been run: Alt-Ergo, Simplify and Z3. Their answers are exactly the same: all VCs are valid except the fifth. On the figure, the user has clickde on the corresponding fifth row, whioch is then highlighted in blue. On the upper right part of thw window, the VC is displayed, and in the lower right part, the source code is displayed, with a block highlighted in orange: this block is where the VC originates. On that example, it means that the system is unable to verify that the result of the method is necessarily greater than or equal to `y`. Indeed, this comes from our intentional mistake. If you modify the source code, replacing the last `x` by `y`, then rerun the `gwhy command`, you can run all provers and check that all VCs are true. This means that the method implementation now satisfies its specification.

2.2 Loop invariants

Methods get a bit harder to prove in presence of loops. Below is a contract of a method for computing the square root of an integer (rounded towards zero).

```
/*@ requires x >= 0;
  @ ensures
  @   \result >= 0 && \result * \result <= x
  @   && x < (\result + 1) * (\result + 1);
  @*/
public static int sqrt(int x);
```

The new `requires` introduce a `precondition`. This is formula that is supposed to hold at the beginning of the method call, here we ask for the parameter `x` to be non-negative, otherwise computing its square root would not be possible. The precondition is not something guaranteed by the method itself: it has to be checked by the caller of the method.

The `ensures` clause given states now that (i) the result is non-negative, (i) the square of the result is lessthan or equal to `x`, and (iii) the successor of the result has a square which is greater than `x`. This ensures that the result is indeed the square root rounded towards zero.

To implement such a method, we propose an algorithm based on the following remark: we know that

$$\sum_{i=0}^{k-1} 2i + 1 = k^2$$

so the square root of n is the smallest integer k such that

$$\sum_{i=0}^{k} 2i + 1$$

is greater than n.

Now, add the following to your class `Lesson1`.

```
/*@ requires x >= 0;
  @ ensures
  @   \result >= 0 && \result * \result <= x
  @   && x < (\result + 1) * (\result + 1);
  @*/
public static int sqrt(int x) {
    int count = 0, sum = 1;
    while (sum <= x) {
        count++;
        sum = sum + 2*count+1;
    }
    return count;
}
```

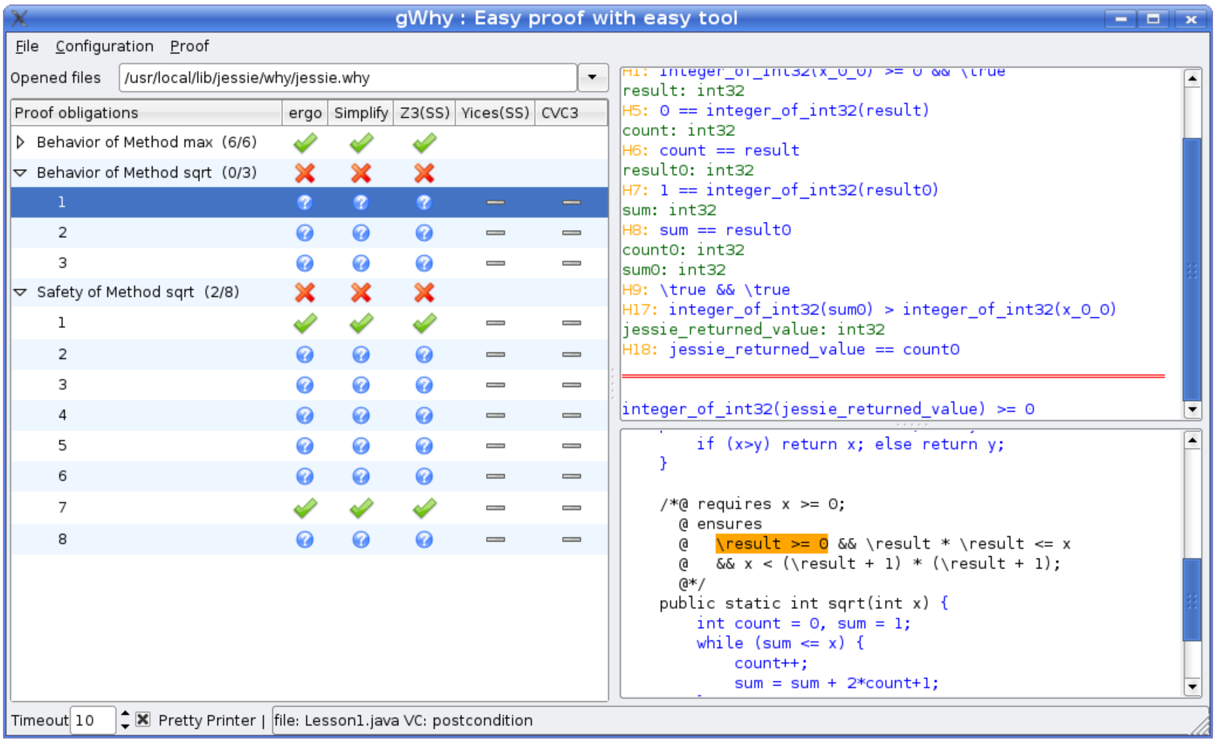
If you generate the VCs now, using

```
gwhy Lesson1.java
```

you will see that almost nothing is provable, as shown on Figure 2.2: the three parts of the postcondition are unproved. (Please ignore the third section “Safety of Method sqrt” for the moment.)

Indeed, as usual in Floyd-Hoare logic, to prove a program with a loop it is required to add a loop invariant to have enough information. Finding the right invariant is usually hard, and usually follows from the principle of the algorithm. Here, `count` will increase one by one, whereas `sum` will always be the sum of odd integers between `1` and `2*count+1`, that is `(count+1)*(count+1)`. So we suggest the following

```
public static int sqrt(int x) {
    int count = 0, sum = 1;
    /*@ loop_invariant
      @   count >= 0 && x >= count*count &&
      @   sum == (count+1)*(count+1);
      @ decreases x - sum;
      @*/
    while (sum <= x) {
        count++;
        sum = sum + 2*count+1;
```



Opened files /usr/local/lib/jessie/why/jessie.why

Proof obligations	ergo	Simplify	Z3(SS)	Yices(SS)	CVC3
▷ Behavior of Method max (6/6)	✓	✓	✓		
▽ Behavior of Method sqrt (0/3)	✗	✗	✗		
1	?	?	?	—	—
2	?	?	?	—	—
3	?	?	?	—	—
▽ Safety of Method sqrt (2/8)	✗	✗	✗		
1	✓	✓	✓	—	—
2	?	?	?	—	—
3	?	?	?	—	—
4	?	?	?	—	—
5	?	?	?	—	—
6	?	?	?	—	—
7	✓	✓	✓	—	—
8	?	?	?	—	—

```
H1: integer_of_int32(x_0_0) >= 0 && \true
result: int32
H5: 0 == integer_of_int32(result)
count: int32
H6: count == result
result0: int32
H7: 1 == integer_of_int32(result0)
sum: int32
H8: sum == result0
count0: int32
sum0: int32
H9: \true && \true
H17: integer_of_int32(sum0) > integer_of_int32(x_0_0)
jessie_returned_value: int32
H18: jessie_returned_value == count0

integer_of_int32(jessie_returned_value) >= 0
```

```
        if (x>y) return x; else return y;
    }

    /*@ requires x >= 0;
      @ ensures
      @   \result >= 0 && \result * \result <= x
      @   && x < (\result + 1) * (\result + 1);
      @*/
    public static int sqrt(int x) {
        int count = 0, sum = 1;
        while (sum <= x) {
            count++;
            sum = sum + 2*count+1;
```

Timeout 10 ☒ Pretty Printer | file: Lesson1.java VC: postcondition

Figure 2.2: Verification conditions for `sqrt`

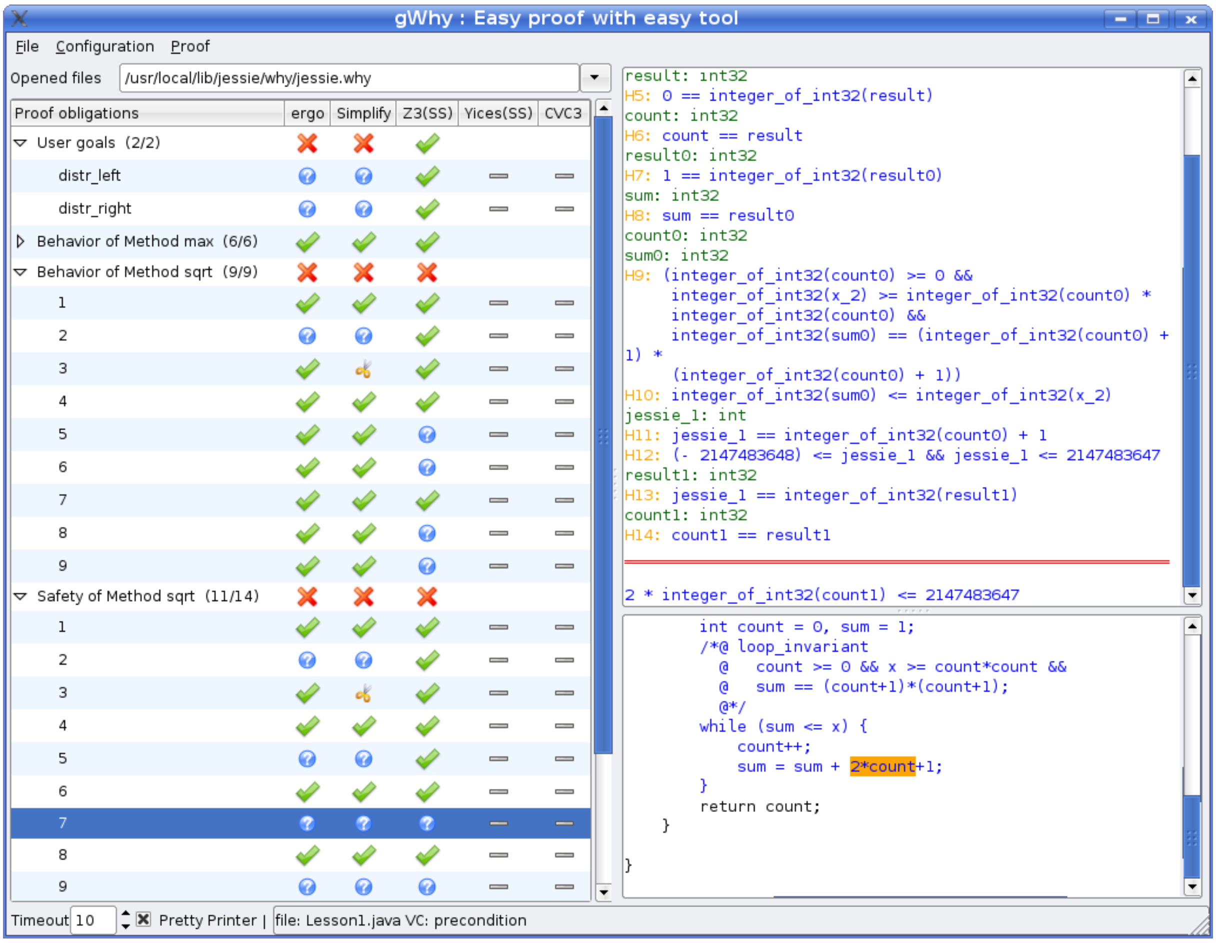
```
        }
        return count;
}
```

If you rerun the GWhy tool and the provers, you will see that the sixth VC, which amounts to show that the property `sum == count * count` is preserved by the loop, is not proved by any provers. It does not mean it is not true: in that case, it illustrates the incompleteness of the method, since checking validity of first-order formulas is undecidable. Showing that `sum == count * count` is preserver by a loop iteration means to prove that assuming `sum == count * count` then `sum' == count' * count'` if `count' == count+1` (instruction `count++`) and `sum' == sum + 2*count' + 1`. Proving this requires to simplify the formulas by applying property of distributivity of multiplication over addition. It appears that automatic provers are usually quite dumb with multiplication. To help them, it is possible to introduce *lemmas* as hints. This is done as follows:

```
/*@ lemma distr_right:
  @   \forall integer x,y,z; x*(y+z) == (x*y)+(x*z);
  @*/

/*@ lemma distr_left:
  @   \forall integer x,y,z; (x+y)*z == (x*z)+(y*z);
  @*/


public class Lesson1 {
```



/usr/local/lib/jessie/why/jessie.why

Proof obligations	ergo	Simplify	Z3(SS)	Yices(SS)	CVC3
User goals (2/2)	✗	✗	✓		
distr_left	?	?	✓	—	—
distr_right	?	?	✓	—	—
Behavior of Method max (6/6)	✓	✓	✓		
Behavior of Method sqrt (9/9)	✗	✗	✗		
1	✓	✓	✓	—	—
2	?	?	✓	—	—
3	✓	✂	✓	—	—
4	✓	✓	✓	—	—
5	✓	✓	?	—	—
6	✓	✓	?	—	—
7	✓	✓	✓	—	—
8	✓	✓	?	—	—
9	✓	✓	?	—	—
Safety of Method sqrt (11/14)	✗	✗	✗		
1	✓	✓	✓	—	—
2	?	?	✓	—	—
3	✓	✂	✓	—	—
4	✓	✓	✓	—	—
5	?	?	✓	—	—
6	✓	✓	✓	—	—
7	?	?	?	—	—
8	✓	✓	✓	—	—
9	?	?	?	—	—

```
result: int32
H5: 0 == integer_of_int32(result)
count: int32
H6: count == result
result0: int32
H7: 1 == integer_of_int32(result0)
sum: int32
H8: sum == result0
count0: int32
sum0: int32
H9: (integer_of_int32(count0) >= 0 &&
     integer_of_int32(x_2) >= integer_of_int32(count0) *
     integer_of_int32(count0) &&
     integer_of_int32(sum0) == (integer_of_int32(count0) +
1) *
     (integer_of_int32(count0) + 1))
H10: integer_of_int32(sum0) <= integer_of_int32(x_2)
jessie_1: int
H11: jessie_1 == integer_of_int32(count0) + 1
H12: (- 2147483648) <= jessie_1 && jessie_1 <= 2147483647
result1: int32
H13: jessie_1 == integer_of_int32(result1)
count1: int32
H14: count1 == result1

2 * integer_of_int32(count1) <= 2147483647
```

```
        int count = 0, sum = 1;
        /*@ loop_invariant
          @   count >= 0 && x >= count*count &&
          @   sum == (count+1)*(count+1);
          @*/
        while (sum <= x) {
            count++;
            sum = sum + 2*count+1;
        }
        return count;
    }

}
```

file: Lesson1.java VC: precondition

Figure 2.3: Loop invariant VCs for `sqrt`

Rerunning again GWhy and the provers results in Figure 2.3. No prover is able to prove every VCs, but combining results of the three provers, each VCs is proved.

Safety

Let's now look at the "Safety" section of VCs. This section contains VCs for checking that no runtime error may occur during execution. These includes:

- Division by zero
- Arithmetic Overflow
- Null pointer dereferencing
- Out-of-bounds arry access

On this example, arithmetic overflow is the concern. On Figure 2.3, the focus is put on VCs 7, located at expression `2*count` in the source code. Looking at the upper right, below the red line is the goal to prove, which is `2 * integer_of_int32(count1) <= 2147483647`. The last constant is $2^{31} - 1$, which is the maximal integer representable in the `int` type. So this goal amounts to prove that `count`, since as a mathematical integer, multiplied by 2, does not exceed $2^{31} - 1$. Indeed it is not provable, because it is not always true: if the value of x is too large, arithmetic overflow may occur.

It is possible to find a bound for `x`, to be added in the precondition, together with appropriate bounds for `sum` and `count` to be added as loop invariants, to prove the absence of arithmetic overflow. This is left as an exercice. In some situation, one may want to simply ignore possible arithmetic overflow. This is done by added the following *pragma* at the beginning of the file:

```
//@+ CheckArithOverflow = no
```

You may try this now, and checks that all remaining VCs are proved.

Termination

Since the `sqrt` method body contains a `while` loop, it is possible that the method execution does not terminate for some of the input. This is not check by default, but if you want to check termination, you can add another clause to the loop annotation: a `decreases` clause. An integer expression must follow, whic is supposed to decreases at each loop iteration, and remaining nonnegative. On the `sqrt` example, that `x-sum` is decreasing, so you can add:

```
/*@ loop_invariant
  @   ...
  @ decreases x - sum;
  @*/
while (sum <= x) {
   ...
```

You can check again that this is porved automatically.

2.3 Array accesses

In this lesson, we now consider arrays. You should now create a new file `Arrays.java`. Here is the specification of a method for computing the maximum element of an array of ints, given as argument:

```
//@+ CheckArithOverflow = no

public class Arrays {

    /*@ requires t != null && t.length >= 1;
      @ ensures
      @   0 <= \result < t.length &&
      @   \forall integer i; 0 <= i < t.length ==> t[i] <= t[\result];
      @*/
    public static int findMax(int[] t);

}
```

The precondition is necessary to ensure that `t` is a non-empty array, so that the maximum exists. An alternative would have been to raise an exception in that case, but exceptions will be considered later. We also add the pragma for avoiding arithmetic overflow checking for simplicity, although in that case there will be no overflow problem.

We propose the following code for that method:

```
  public static int findMax(int[] t) {
      int m = t[0];
      int r = 0;
      /*@ loop_invariant
        @   1 <= i && i <= t.length && 0 <= r && r < t.length &&
        @   m == t[r] && \forall integer j; 0<=j && j<i ==> t[j]<=t[r];
        @ decreases t.length-i;
        @*/
      for (int i=1; i < t.length; i++) {
          if (t[i] > m) {
              r = i;
              m = t[i];
          }
      }
      return r;
  }
```

The loop invariant ensures that `r` is always the index of the maximum element of `t` between `0` and `i-1`. We put also as invariant the facts that `r` and `i` remain in the bounds of `t`.

2.4 Logic predicates

The following example is the same is the previous one. We just introduce a logic predicate to avoid the writing of similar formulas.

```
/*@ predicate is_max(int[] t,integer i,integer l) {
  @   0 <= i < l &&
  @       \forall integer j; 0 <= j < l ==> t[j] <= t[i] }
  @*/

public class Arrays {

    /*@ requires t != null && t.length >= 1;
      @ ensures is_max(t,\result,t.length);
      @*/
    public static int findMax2(int[] t) {
        int m = t[0];
        int r = 0;
        /*@ loop_invariant
          @   1 <= i <= t.length && m == t[r] && is_max(t,r,i) ;
          @ decreases t.length-i;
          @*/
        for (int i=1; i < t.length; i++) {
            if (t[i] > m) {
                r = i;
                m = t[i];
            }
        }
        return r;
    }

}
```

2.5 Array updates

The following method shifts for one cell to the right the contents of an array. It illustrates the `\old` and the `\at` constructs.

```
public class Arrays {

    /*@ requires t != null;
      @ ensures
      @   \forall integer i; 0 < i < t.length ==> t[i] == \old(t[i-1]);
      @*/
    public static void shift(int[] t) {
        /*@ loop_invariant
          @   j < t.length &&
          @   (\forall integer i; 0 <= i <= j ==> t[i] == \at(t[i],Pre)) &&
          @   (\forall integer i;
          @          j < i < t.length ==> t[i] == \at(t[i-1],Pre));
          @ decreases j;
          @*/
        for (int j=t.length-1 ; j > 0 ; j--) {
            t[j] = t[j-1];
        }
    }

}
```

2.6 Objects and constructors

It is time to consider true JAVA objects. Let's consider the following class that implements a very simple electronic purse.

```
class NoCreditException extends Exception {

    public NoCreditException();

}

public class Purse {

    public int balance;

    //@ invariant balance_non_negative: balance >= 0;

    /*@ ensures balance == 0;
```

```
      @*/
    public Purse() {
        balance = 0;
    }

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

    /*@ requires s >= 0 && s <= balance;
      @ ensures balance == \old(balance) - s;
      @*/
    public void withdraw(int s) {
        balance -= s;
    }

}
```

2.7 Calling subprograms, `assigns` clauses

```
    /*@ public normal_behavior
      @   ensures \result == 150;
      @*/
    public static int test1() {
        Purse p = new Purse();
        p.credit(100);
        p.withdraw(50);
        p.credit(100);
        return p.balance;
    }

    /*@ public normal_behavior
      @   ensures \result == 150;
      @*/
    public static int test2() {
        Purse p1 = new Purse();
        Purse p2 = new Purse();
        p1.credit(100);
        p2.credit(200);
        p1.withdraw(50);
        p2.withdraw(100);
        return p1.balance+p2.balance;
    }
```

These tests cannot be proved, the `credit` and `withdraw` methods must be given `assigns` clauses as follows:

```
/*@ assigns balance;
  @ ensures balance == 0;
  @*/
Purse() {
    balance = 0;
}

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

/*@ requires 0 <= s <= balance;
  @ assigns balance;
  @ ensures balance == \old(balance) - s;
  @*/
public void withdraw(int s) {
    balance -= s;
}
```

2.8 Programs with exceptions

```
class NoCreditException extends Exception {

    /*@ assigns \nothing;
      @*/
    NoCreditException() {}

}

class 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 withdraw2(int s) throws NoCreditException {
        if (balance >= s) {
            balance = balance - s;
        }
        else {
            throw new NoCreditException();
        }
    }

}
```

2.9 The Dutch National Flag problem

```
//@ predicate is_color(int c) { Flag.BLUE <= c && c <= Flag.RED }

/*@ predicate is_color_array{L}(int t[]) {
  @   t != null &&
  @   \forall int i; 0 <= i && i < t.length ==> is_color(t[i])
  @ }
  @*/

/*@ predicate is_monochrome{L}(int t[],integer i, integer j, int c) {
  @   \forall int k; i <= k && k < j ==> t[k] == c
  @ }
  @*/



class Flag {

    public static final int BLUE = 1, WHITE = 2, RED = 3;

    int t[];
    //@ invariant t_non_null: t != null;
    //@ invariant is_color_array_inv: is_color_array(t);

    /*@ requires 0 <= i <= j <= t.length ;
      @ behavior decides_monochromatic:
      @   ensures \result <==> is_monochrome(t,i,j,c);
      @*/
    public boolean isMonochrome(int i, int j, int c) {
            /*@ loop_invariant i <= k &&
          @   (\forall int l; i <= l && l < k ==> t[l]==c);
              @ decreases j - k;
          @*/
        for (int k = i; k < j; k++) if (t[k] != c) return false;
        return true;
    }

    /*@ requires 0 <= i < t.length && 0 <= j < t.length;
      @ behavior i_j_swapped:
      @   assigns t[i],t[j];
      @   ensures t[i] == \old(t[j]) && t[j] == \old(t[i]);
    @*/
  private void swap(int i, int j) {
      int z = t[i];
      t[i] = t[j];
      t[j] = z;
  }

  /*@ behavior sorts:
    @   assigns t[..];
    @   ensures
    @     (\exists int b,r; is_monochrome(t,0,b,BLUE) &&
    @                       is_monochrome(t,b,r,WHITE) &&
    @                       is_monochrome(t,r,t.length,RED));
    @*/
  public void flag() {
      int b = 0;
      int i = 0;
      int r = t.length;
      /*@ loop_invariant
        @   is_color_array(t) &&
        @   0 <= b && b <= i && i <= r && r <= t.length &&
        @   is_monochrome(t,0,b,BLUE) &&
        @   is_monochrome(t,b,i,WHITE) &&
        @   is_monochrome(t,r,t.length,RED);
        @ decreases r - i;
        @*/
      while (i < r) {
          switch (t[i]) {
          case BLUE:
              swap(b++, i++);
              break;
          case WHITE:
              i++;
              break;
          case RED:
              swap(--r, i);
              break;
          }
      }
  }
}
```

2.10 Ghost variables

The following example illustrates code instrumentation using ghost variables.

```
/* complements for non-linear integer arithmetic */

/*@ lemma distr_right:
  @   \forall integer x,y,z; x*(y+z) == (x*y)+(x*z);
  @*/

/*@ lemma distr_left:
  @   \forall integer x,y,z; (x+y)*z == (x*z)+(y*z);
  @*/

/*@ lemma distr_right_minus:
  @   \forall integer x,y,z; x*(y-z) == (x*y)-(x*z);
  @*/

/*@ lemma distr_left_minus:
  @   \forall integer x,y,z; (x-y)*z == (x*z)-(y*z);
  @*/

/*@ lemma mul_comm:
  @   \forall integer x,y; x*y == y*x;
  @*/

/*@ lemma mul_assoc:
  @   \forall integer x,y,z; x*(y*z) == (x*y)*z;
  @*/

/*@ predicate divides(integer x, integer y) {
  @   \exists integer q; y == q*x
  @ }
  @*/

/*@ lemma div_mod_property:
  @  \forall integer x,y;
  @    x >=0 && y > 0 ==> x%y  == x - y*(x/y);
  @*/

/*@ lemma mod_property:
  @  \forall integer x,y;
  @    x >=0 && y > 0 ==> 0 <= x%y && x%y < y;
  @*/

/*@ predicate isGcd(integer a, integer b, integer d) {
  @   divides(d,a) && divides(d,b) &&
  @     \forall integer z;
  @     divides(z,a) && divides(z,b) ==> divides(z,d) }
  @*/

/*@ lemma gcd_zero :
  @   \forall integer a; isGcd(a,0,a) ;
  @*/

/*@ lemma gcd_property :
  @   \forall integer a, b, d, q;
  @       b > 0 && isGcd(b,a % b,d) ==> isGcd(a,b,d) ;
  @*/

class Gcd {

    /*@ requires x >= 0 && y >= 0;
      @ behavior resultIsGcd:
      @   ensures isGcd(x,y,\result) ;
      @ behavior bezoutProperty:
      @   ensures \exists integer a,b; a*x+b*y == \result;
      @*/
    int gcd(int x, int y) {

        //@ ghost integer a = 1, b = 0, c = 0, d = 1;

        /*@ loop_invariant
          @     x >= 0 && y >= 0 &&
          @     (\forall integer d ;  isGcd(x,y,d) ==>
          @         isGcd(\at(x,Pre),\at(y,Pre),d)) &&
          @     a*\at(x,Pre)+b*\at(y,Pre) == x &&
          @     c*\at(x,Pre)+d*\at(y,Pre) == y ;
          @ decreases y;
          @*/
        while (y > 0) {
            int r = x % y;
            //@ ghost integer q = x / y;
            x = y;
            y = r;
            //@ ghost integer ta = a, tb = b;
            //@ ghost a = c;
            //@ ghost b = d;
            //@ ghost c = ta - c * q;
            //@ ghost d = tb - d * q;
        }

        return x;
    }

}
```

Chapter 3

Specification Language: Reference

3.1 Logic expressions

We present the language of expressions one can use in annotations. These are called *logic expressions*.

This language is essentially the standard multi-sorted first-order logic. It is a two-valued logic language, made of *propositions* with standard first-order connectives, built upon a language of atoms called *terms*.

As far as possible, the syntax of Java expressions is reused: conjunction is denoted as `&&`, disjunction as `||` and negation as `!`. Additional connectives are `==>` for implication, `<==>` for equivalence. Universal quantification is denoted by $\texttt{\textbackslash forall} \tau\, x_1, \ldots, \tau\, x_n ; e$ and existential quantification by $\texttt{\textbackslash exists} \tau\, x_1, \ldots, \tau\, x_n ; e$.

Terms are built on

- integer and real arithmetic
- Java array access and field access
- Java cast
- user-defined logic types, and user-defined predicate and logic functions, described in Section 3.6

In essence, terms correspond to pure Java expressions, with additional constructs that we will introduce progressively, except that method and constructor calls are not allowed.

Figure 3.1 presents the grammar for the basic construction of logic expressions.

Basic additional constructs are as follows:

Conditional $c ? e_1 : e_2$. There is a subtlety here: the condition may be either a boolean term or a proposition. In case of a proposition, the two branches must be also proposition, so that this construct acts as a connective with the following semantics: $c ? e_1 : e_2$ is equivalent to $(c \texttt{==>} e_1) \texttt{\&\&} (\texttt{!} c \texttt{==>} e_2)$

Consecutive comparison operators the construct $t_1\ relop_1\ t_2\ relop_2\ t_3 \cdots t_k$ with several consecutive comparison operators is a shortcut for $t_1\ relop_1\ t_2$ `&&` $t_2\ relop_2\ t_3$ `&&` $\cdots$. Nevertheless, it is required that the $relop_i$ operators must be in the same “direction”, *i.e.* they must all belong either to $\{<, <=, ==\}$ or to $\{>, >=, ==\}$. For example, expressions `x < y > z` and `x != y != z` are forbidden.

3.1.1 Operator precedence

The precedence of Java operators is conservatively extended with additional operators, as shown Figure 3.2. In this table, operators are sorted from highest to lowest priority. Operators of same priority are presented on the same line.

```
              rel-op  ::=  == | != | <= | >= | > | <

              bin-op  ::=  + | - | * | / | %
                        |  && | || |                      boolean operations
                        |  & | |                          bitwise operations
                        |  ==>                            boolean implication
                        |  <==>                           boolean equivalence

            unary-op  ::=  + | -                          unary plus and minus
                        |  !                              boolean negation
                        |  ~                              bitwise complementation

               lexpr  ::=  true | false
                        |  integer                        integer constants
                        |  real                           real constants
                        |  id                             variables
                        |  unary-op lexpr
                        |  lexpr bin-op lexpr
                        |  lexpr (rel-op lexpr)+          comparisons, see remark below
                        |  lexpr [ lexpr ]                array access
                        |  lexpr . id                     object field access
                        |  ( type-expr ) lexpr            cast
                        |  id ( lexpr (, lexpr)* )        function application
                        |  ( lexpr )                      parentheses
                        |  lexpr ? lexpr : lexpr
                        |  \forall binders ; lexpr        universal quantification
                        |  \exists binders ; lexpr        existential quantification

             binders  ::=  type-expr variable-ident
                           (, type-expr variable-ident)*

           type-expr  ::=  logic-type-expr | Java-type-expr

     logic-type-expr  ::=  built-in-logic-type
                        |  id                             logic type identifier

 built-in-logic-type  ::=  integer | real

      variable-ident  ::=  id | variable-ident []
```

Figure 3.1: Grammar of logic expressions

class	associativity	operators
selection	left	`[···] .`
unary	right	`! ~ + -` (cast)
multiplicative	left	`* / %`
additive	left	`+ -`
shift	left	`<< >> >>>`
comparison	left	`< <= > >=`
comparison	left	`== !=`
bitwise and	left	`&`
bitwise xor	left	`^`
bitwise or	left	`|`
connective and	left	`&&`
connective xor	left	`^^`
connective or	left	`||`
connective implies	right	`==>`
connective equiv	left	`<==>`
ternary connective	right	`··· ? ··· : ···`
binding	left	`\forall \exists`

Figure 3.2: Operator precedence

3.1.2 Semantics

The semantics of logic expressions is based on mathematical first-order logic (`http://en.wikipedia.org/wiki/First_order_logic`), thus it is a 2-valued logic with only total functions. Consequently, expressions are never "undefined".

This design choice has to be emphasized because it is not straightforward, and specification writer should be aware of that. The issues are shared with JML. A comprehensive list of issues has been compiled by Patrice Chalin [2, 3].

The choice of having only total functions allows to write for example the term `1/0`, or `p.f` when p is null, or `t[n]` where n is outside the array bounds. In particular, the predicates

$$\begin{array}{rcl} 1/0 & == & 1/0 \\ p.f & == & p.f \end{array}$$

are true, since they are instances of the general axiom $\forall x, x{==}x$ of first-order logic.

So, it is up to the writer of specification to take care of writing consistent assertions.

3.1.3 Typing

The language of logic expressions is typed (as for *multi-sorted* first-order logic). Types are either Java types or *logic types* defined as follows:

- "Mathematical" types: `integer` for unbounded, mathematical integers; `real` for real numbers.
- Logic types introduced by specification writer (see Section 3.6).

There are implicit coercions for numeric types:

- integer types `char`, `byte`, `short`, `int` and `long` are all subtypes of type `integer`,
- `integer` is itself a subtype of type `real`,

- types `float` and `double` are subtypes of type `real`.

Notes:

- There is a distinction between booleans and propositions. An expression like $x < y$, in a term position returns a boolean, and is also allowed in a proposition position.
- Quantification can be made over any type: logic types and any Java types. Quantification over objects must be carefully designed, regarding the memory state where field accesses are done: see Section 3.1.4 and Section 3.6.2.

3.1.4 Integer arithmetic and machine integers

The following integer arithmetic operations apply to *mathematical integers*: addition, subtraction, multiplication, unary minus, division and modulo. The value of a Java variable of an integer type is promoted to a mathematical integer. As a consequence, there is no such thing as "overflow" in logic expressions.

For division and modulo, the results are not specified if divisor is zero, otherwise if q and r are the quotient and the remainder of n divided by d then:

- $|d.q| \leq |n|$, and $|q|$ is maximal for this property ;
- q is zero if $|n| < |d|$;
- q is positive if $|n| \geq |d|$ and n and d have the same sign ;
- q is negative if $|n| \geq |d|$ and n and d have the opposite signs ;
- $q.d + r = n$;
- $|r| < |d|$;
- r is zero or has the same sign as d.

Example 3.1 *The following examples illustrates the results of division and modulo depending on signs of arguments:*

- `5/3` *is 1 and* `5%3` *is 2*
- `(-5)/3` *is -1 and* `(-5)%3` *is -2*
- `5/(-3)` *is -1 and* `5%(-3)` *is 2*
- `(-5)/(-3)` *is 1 and* `(-5)%(-3)` *is -2*

Hexadecimal and octal constants

Hexadecimal and octal constants always denote non-negative integers.

Casts and overflow

In logic expressions, casting operations from mathematical integers towards a Java integer type t (among `char`, `byte`, `short`, `int` and `long`) is allowed, and is interpreted as follows: the result is the unique value of the corresponding type that is congruent to the mathematical result modulo the cardinal of this type.

Example 3.2 `(byte)1000` *is* $1000 \bmod 256$ i.e. -24.

If one wants to express, in the logic, the result of the Java computations of an expression, one should add all necessary casts. For example, the logic expression which denotes the result of Java computation of $x * y + z$ is `(int)((int)(x*y)+z)`.

Remark: implicit casts from integers to Java integer type are forbidden.

Quantification

Quantification can be either on mathematical `integer` or bounded types `short`, `char`, etc. In the latter case, quantification corresponds to integer quantification over the corresponding interval.

Example 3.3 *The formula*

```
\forall byte b; b <= 1000
```

is valid since it is equivalent to

```
\forall integer b; -128 <= b <= 127 ==> b <= 1000
```

Bitwise operations

Like arithmetic operations, bitwise operations apply to any mathematical integers: any mathematical integer as a unique infinite 2-complement binary representation with infinitely many 0 (for non-negative numbers) or 1 (for negative numbers) on the left. Then bitwise operations apply to these representation.

Example 3.4

- `7 & 12` = $\cdots 00111$`&`$\cdots 001100 = \cdots 00100$ = *4*
- `-8 | 5` = $\cdots 11000$`|`$\cdots 00101 = \cdots 11101$ = *-3*
- `~5` = `~`$\cdots 00101 = \cdots 111010$ = *-6*
- `-5 << 2` = $\cdots 11011$<<2 = $\cdots 11101100$ = *-20*
- `5 >> 2` = $\cdots 00101$>>2 = $\cdots 0001$ = *1*
- `-5 >> 2` = $\cdots 11011$>>2 = $\cdots 1110$ = *-2*

3.1.5 Real numbers and floating point numbers

Floating-point constants and operations are interpreted as mathematical real numbers. A Java variable of type float or double is implicitly promoted to a real. Integers are promoted to reals if necessary.

Example 3.5 $2 * 3.5$ *denotes the real number 7*

Comparisons operators are interpreted as real operators too.

3.2 Simple contracts

A simple contract is an annotation that can be given to constructors and methods (not depending on whether they are given a body or are abstract). It is a contract between the caller and the callee, made of a precondition and a postcondition:

- Precondition:
  - The caller is responsible for ensuring it before call
  - The callee assumes it at entrance of its body
- Postcondition:
  - The callee must guarantee it at normal exit
  - The caller can then assume it after the call

contract ::= (*requires-clause* | *assigns-clause* | *ensures-clause*)*

requires-clause ::= `requires` *proposition* `;`

assigns-clause ::= `assigns` *locations* `;`

locations ::= *location* (`,` *location*)* | `\nothing`

ensures-clause ::= `ensures` *proposition* `;`

Figure 3.3: Grammar of simple contracts

Additionally, a contract can be completed with a *frame clause* which describes side-effects, and acts as a postcondition.

Note that post-conditions of simple contracts only concern normal exit: exiting with exceptions can be specified using additional *behavior clauses* described in Section 3.3

The syntax of simple contracts syntax is given Figure 3.3. Let's consider a simple contract of the following generic form:

```
/*@ requires $P_1$;
  @ requires $P_2$;
  @ assigns $L_1$;
  @ assigns $L_2$;
  @ ensures $E_1$;
  @ ensures $E_2$;
  @*/
```

the semantics of such a contract can be rephrased as follows:

- The caller must guarantee that the callee is called in a state (called *prestate*) where the property $P_1 \&\& P_2$ holds.
- When the callee returns normally, the property $E_1 \&\& E_2$ must hold in the corresponding *poststate*.
- All memory locations of the prestate that do not belong to the set $L_1 \cup L_2$ remain allocated and are left unchanged in the poststate.

Thus, the contract above is equivalent to the following simplified one:

```
/*@ requires $P_1 \&\& P_2$;
  @ assigns $L_1, L_2$;
  @ ensures $E_1 \&\& E_2$;
  @*/
```

The multiplicity of clauses are proposed mainly to improve readibility. Also, if no clause **`requires`** is given, it defaults to requiring 'true', and similarly for **`ensures`** clause. Giving no **`assigns`** clause means that side-effects are not specified: it potentially modifies everything.

3.3 Behavior clauses

A simple contract can be augmented with *behaviors*.

A behavior without throws clause as the form

```
behavior id :
  assumes A ;
  assigns  L ;
  ensures  E ;
```

The semantics of such a behavior is as follows. The callee guarantees that if it returns normally, then in the post-state:

- `\old`$(A) \Rightarrow E$ holds
- If `\old`(A) holds, each location of the pre-state not in L remains allocated and unchanged in the post-state

A behavior with a throws clause as the form

```
behavior id :
  assumes A ;
  throws Exc ;
  assigns  L ;
  ensures  E ;
```

The semantics of such a behavior is as follows. The callee guarantees that if it exits with exception Exc, then in the post-state:

- `\old`$(A) \Rightarrow E$ holds
- If `\old`(A) holds, each location of the pre-state not in L remains allocated and unchanged in the post-state

Notice that in E, `\result` is bound to the exception object thrown.

3.4 Code annotations

3.4.1 Assertions

- Additional clause:

  ```
  //@ assert prop
  ```

  specify that the given property is true at the corresponding program point

3.4.2 Loop annotations

- Additional clause for loops:

  ```
  /*@ loop_invariant prop
    @ loop_assigns tset
    @*/
  ```

  specify that

  - the given property is an inductive loop invariant for the loop:
    - true at loop entrance
    - preserved by loop body
  - side-effect of the loop are included in the given tset

3.5 Data invariants

A class invariant is a property attached to a class. This property is supposed to hold on any object of that class. More precisely, it holds at method entrance and exit, and at the exit of constructor.

3.6 Advanced modeling language

3.6.1 Logic definitions

- New logic functions and predicates can be defined

3.6.2 Hybrid predicates

- Logic functions and predicates may depend on memory heap
- In such a case, logic labels are required:
- Indeed, with only one label, the following is allowed too:

3.6.3 Abstract data types

New logic data types (i.e. immutable) can be introduced abstractly by giving

- a type name
- the profile of logic functions and predicates operating on it
- a set of axioms

3.7 Termination

3.7.1 Loop variants

- A loop can be annotated with a *variant*: an integer expression that decreases at each iteration

3.8 Ghost variables

Chapter 4

Appendices

4.1 Requirements

Compiling from sources requires Objective Caml compiler, version 3.09 or higher.

External theorem provers must be installed. Alt-Ergo is available from http://alt-ergo.lri.fr, simplify is available from (see why web page), Z3 ia available from ..., Yices from ..., CVS3 from ...

For using the Coq intereactive prover, you need Coq version 8.0 or higher.

4.2 Installation procedure

4.2.1 From the sources

Get a copy of sources at the web page http://why.lri.fr/.

Decompress the archive in a directory of your choice.

Run commands

```
./configure
make
make install
```

4.2.2 Binaries

Please look at the web page http://why.lri.fr/ for binaries for popular architectures. Krakatoa is distributed as part of the Why debian package, available on standard repositories of debian-based distributions.

4.3 Summary of features and known limitations

- Unsupported kind of statements in translation: ...
- exception NullPointerException and ArrayOutOfBoundsException are required NOT to be thrown, and consequently should not be catched.
- For recursive or mutually recursive methods, only partial correctness is guaranteed.

4.4 Contacts

The web page for Krakatoa is at URL `http://krakatoa.lri.fr/` and the wbe page for the Why platform in general is at `http://why.lri.fr`.

For general questions regarding the use of the tool, please use the Why mailing list. You need to subscribe to the list before sending a message to it. To subscribe, follow the instructions given on page `http://lists.gforge.inria.fr/cgi-bin/mailman/listinfo/why-discuss`

For bug reports, please use the bug tracking system at `https://gforge.inria.fr/tracker/?atid=4012&group_id=999&func=browse`. For security reasons, you need to register before submitting a new bug. Please create an account there, where you can put "ProVal" for the required field "INRIA Research Project you work with or work in".

In case if the mailing list above is not appropriate, you can contact the authors directly at the following address: `mailto:ClaudedotMarcheatinriadotfr`.

Bibliography

[1] The Why verification tool. `http://why.lri.fr/`.

[2] Patrice Chalin. Reassessing JML's logical foundation. In *Proceedings of the 7th Workshop on Formal Techniques for Java-like Programs (FTfJP'05)*, Glasgow, Scotland, July 2005.

[3] Patrice Chalin. A sound assertion semantics for the dependable systems evolution verifying compiler. In *Proceedings of the International Conference on Software Engineering (ICSE'07)*, pages 23–33, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[4] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, Iowa State University, 2000.

Index

arithmetic overflow, 11
`assigns` clause, 15
assigns-clause
    non-terminal, 26
`\at` construct, 14

behavior, 16
`behavior` declaration, 16
bin-op
    non-terminal, 22
binders
    non-terminal, 22
built-in-logic-type
    non-terminal, 22

Coq, 29

`decreases` clause, 12

`ensures` clause, 7
ensures-clause
    non-terminal, 26

`ghost` declaration, 18
ghost variables, 18
GWhy, 7
`gwhy` command, 7

Java, 1, 7, 14

`lemma` declaration, 11
lemmas, 11
lexpr
    non-terminal, 22
locations
    non-terminal, 26
logic predicates, 13
logic-type-expr
    non-terminal, 22
loop invariant, 8
`loop_invariant` clause, 8

`\old` construct, 14

postcondition, 7
precondition, 8
`predicate` declaration, 13

`requires` clause, 8
requires-clause
    non-terminal, 26

safety, 11

termination, 12
`throws` clause, 16
type-expr
    non-terminal, 22

unary-op
    non-terminal, 22

variable-ident
    non-terminal, 22