The Krakatoa Verification Tool for JAVA programs

Tutorial and Reference Manual

Version 2.34

Claude Marché

March 17, 2014
INRIA Team *Toccata* http://toccata.lri.fr
INRIA Saclay - Île-de-France & LRI, CNRS UMR 8623
Batiment 650, Université Paris-Sud 91405 Orsay cedex, France

Contents

Chapter 1

Introduction

Krakatoa is a tool for certification of Java programs, annotated using the Java Modeling Language [5] (JML for short), using the Why [1] tool for generating proof obligations.

This version 2.34 of Krakatoa is a major rewriting of the version 0.x family. Major changes have occurred 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.

1.1 Important note for version 2.30

The use of the Why2 VC generator is now obsolete, and it is recommended to switch to the Why3 system for specification and VC generation. Why3 must be installed independently of Why2, please instructions given at `http://why3.lri.fr`.

The version of Why3 that is compatible with this version 2.30 of Krakatoa is the version 0.71. Please see `http://krakatoa.lri.fr/` for more details on compatibility between Why2/Krakatoa and Why3.

In this manual, it is assumed that the Why3 VC generator and IDE is in use. The old behavior using the Why2 VC generator and GUI remains possible, using the `gwhy` command just as in version 2.29.

Chapter 2

Tutorial

This chapter provides the basic techniques for specifying a Java source code and prove it correct with Krakatoa. We also recommend to look at the collection of verified programs at URL http://toccata.lri.fr/gallery/krakatoa.en.html to get more hints on how to specify and prove programs with Krakatoa.

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`:

```
krakatoa Lesson1.java
```

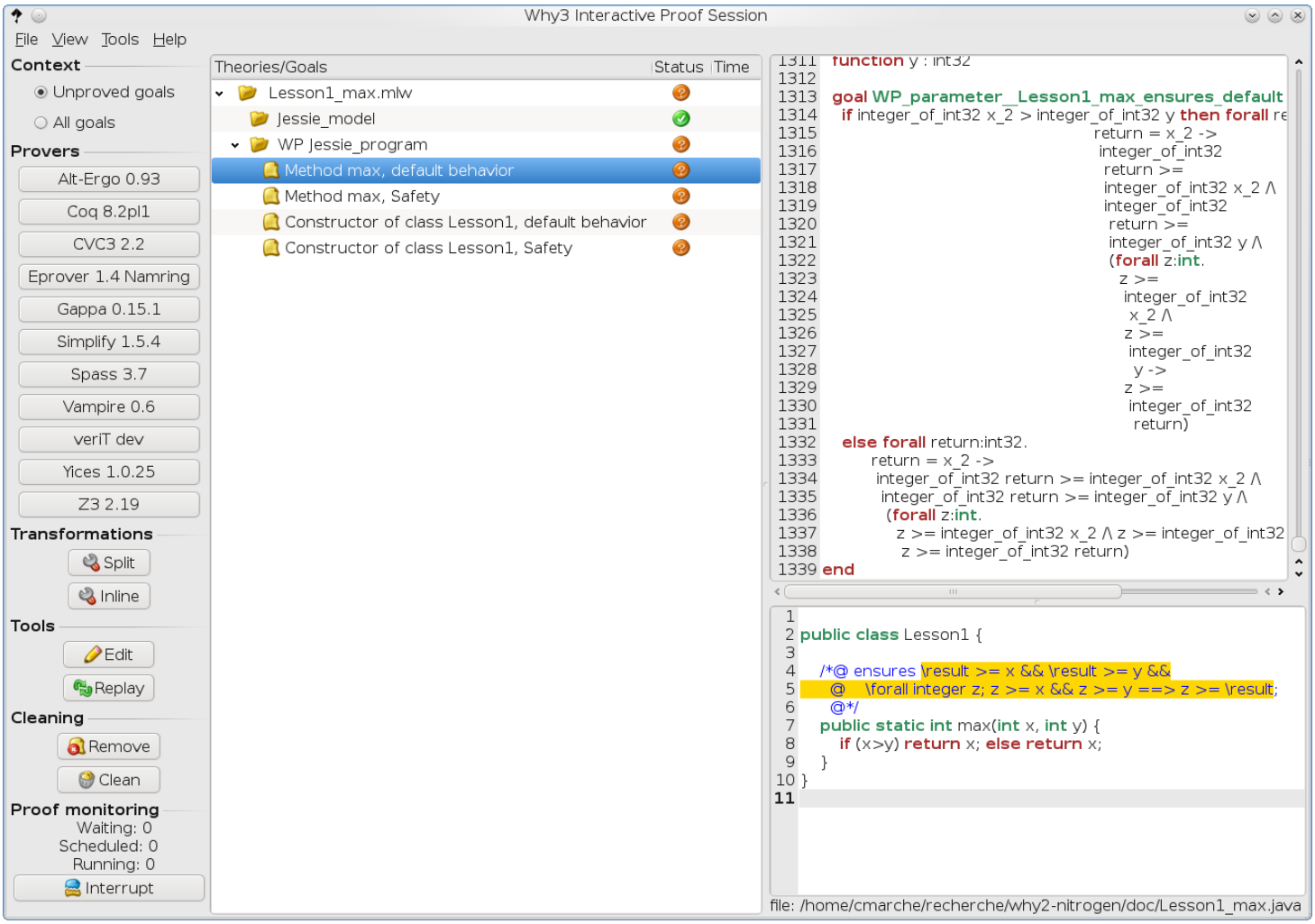

Figure 2.1: Verification conditions displayed in Why3 IDE

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 graphical interface of the Why3 platform, as shown in Figure 2.1

Please refer to Why3's user manual for details on the use of this interface. We only remind here the main features.

The column on the left is a tool bar. The large window in the middle is the tree view of all generated VCs. There are 4 VCs generated from this program, given as sub-rows of the row called "WP Jessie program". Generally speaking, there is one VCs for each behavior of each method or constructor. Safety is one of these behaviors, which includes checking absence of null dereferencing, absence of out-of-bounds array acces, absence of division by zero, and other kind of possible runtime errors like that. In this tree view, the first VC has been selected was selected, by clicking on the corresponding row. The corresponding specification frim the source code is shown on the bottom right part: it is here the post-condition we gave to method max. The top right part of the window displays the logical formula associated to the VC.

The left toolbar contains a series of buttons for each prover that was detected when you run "why3config –detect". A given prover can be run on a given goal, or a set of goals by selecting the wanted part of the tree. In our example, we can select the row "WP Jessie program" and click e.g on

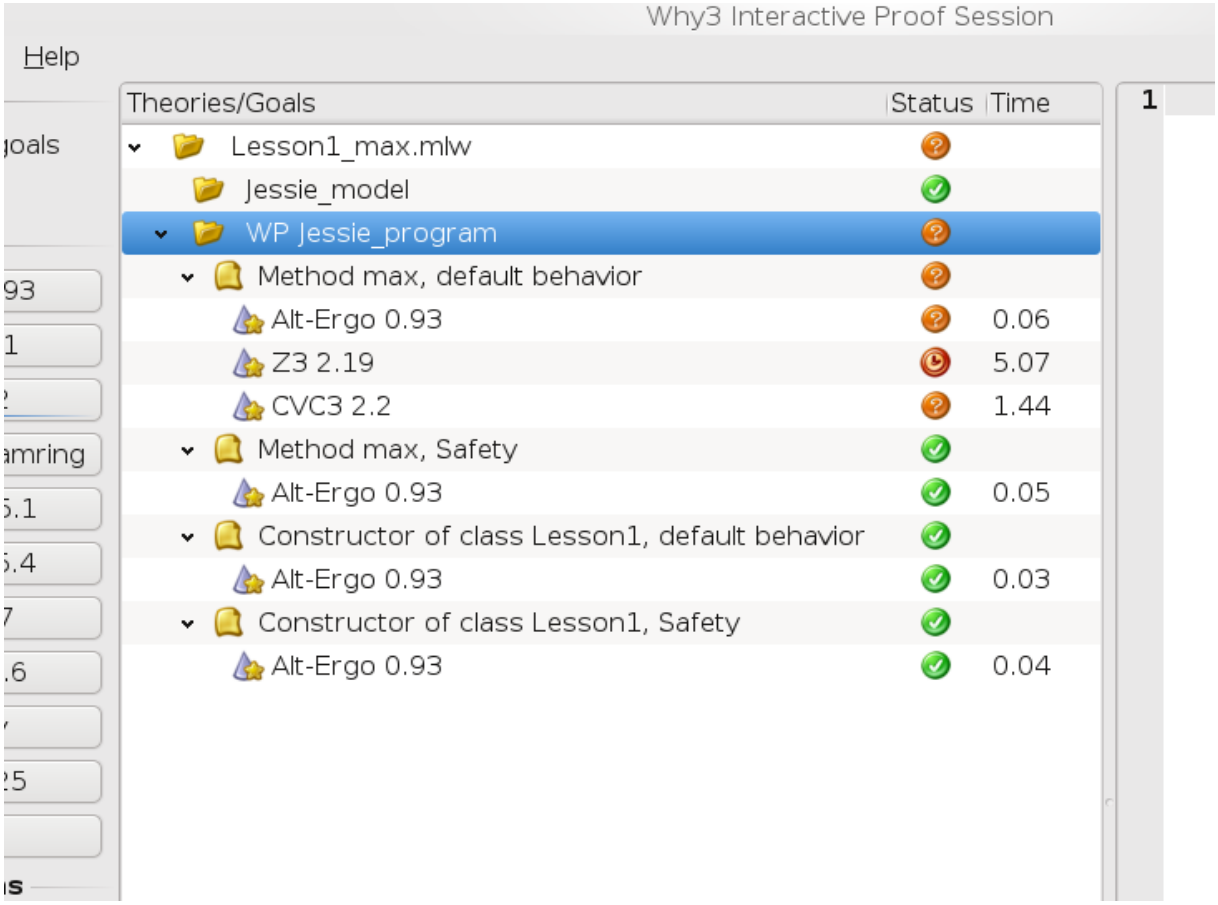

Figure 2.2: Provers run on verification conditions

Alt-Ergo. Clicking further on other provers like Z3 or CVC3, will run these on the VCs that are not yet proved (see Why3 manual for details). The result is displayed on Figure 2.2.

The VC for the post-condition of max is not proved. To investigate further, and since it is a conjunction of several propositions, it is a good idea to split it in parts. This is done by selecting the VC and clicking on the "split" button. This results into two subgoals for that VC. Again, we can click on provers Alt-Ergo, Z3 and CVC3 to check these subgoals. The second remains unproved. It can be splitted in parts again, which this time results into 3 subgoals, and provers can be run again on them. The result is displayed on Figure 2.3.

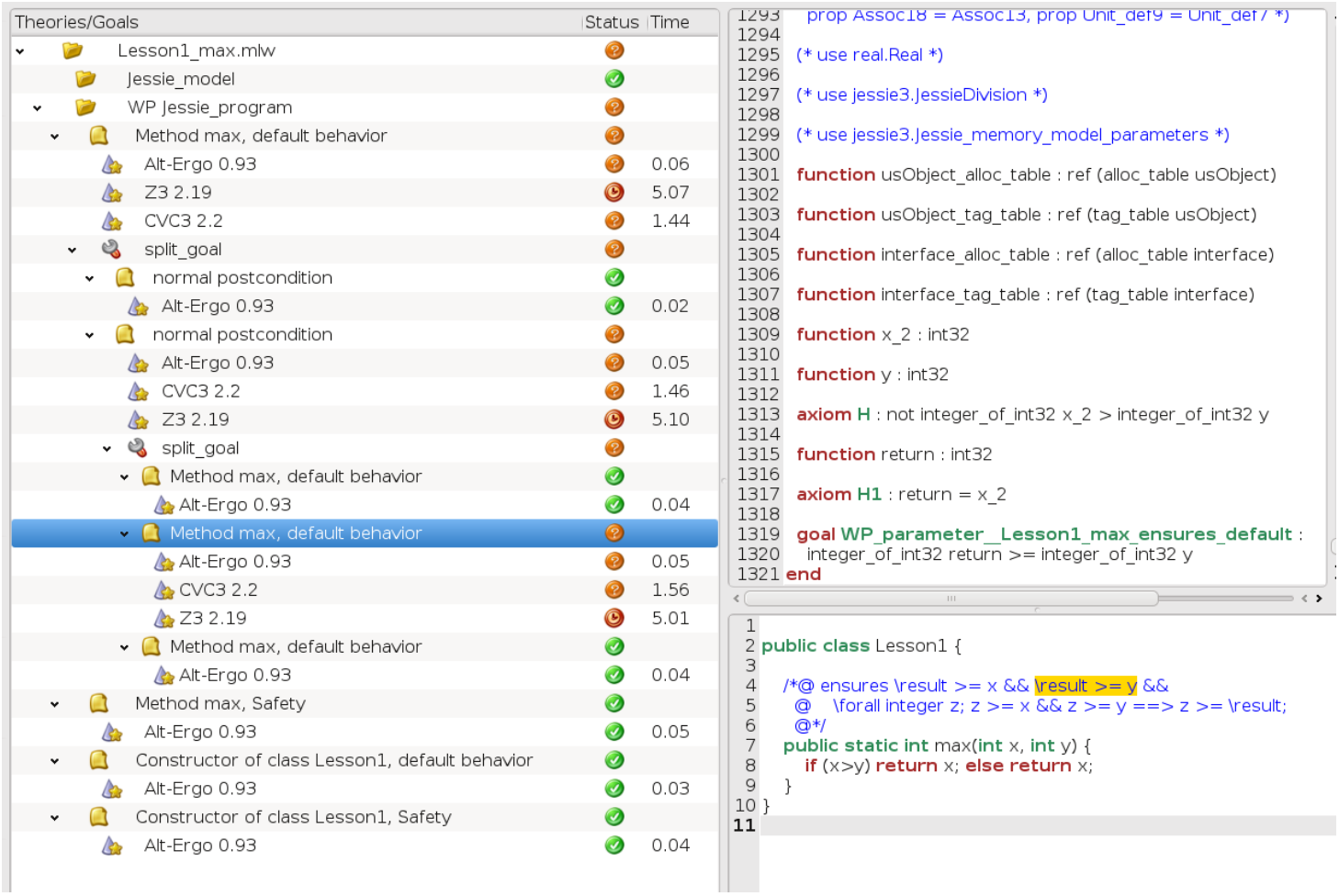

Figure 2.3: Splitting verification condition in parts

The subgoal that remains unproved was selected, and the corresponging spec is shown on the bottom left: it is the part `\result >= y` which is not valid. Indeed, this comes from our intentional mistake; the result s not necessarily greater or equal to y in the second branch. It is time to fix the source code. You can quit the graphical interface, and replace with your favorite editor the last `x` by `y`. If you rerun the `krakatoa` command as above, you will see that the state of your proof session was recorded and thus see all the splitting you made earlier. You notice also that most of the goals are marked as "obsolete" meaning that the results recorded in your proof sessions are not accurate any more. It is the right time to use the "replay" button of the interface to rerun prover on all obsolete proofs. Then click on the remaining VC and on a prover like Alt-ergo, to see that this time the proof is OK. Indeed, the splittings that you made to reach this state are not necessary anymore; select the first split and the "remove" button, then restart provers on the first VC, it should be proved. 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, (ii) the square of the result is less than 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

```
krakatoa Lesson1.java
```

you will see the VC for the post-condition, as shown on Figure 2.4 is unproved. (Please ignore the VC "Safety" for the moment.)

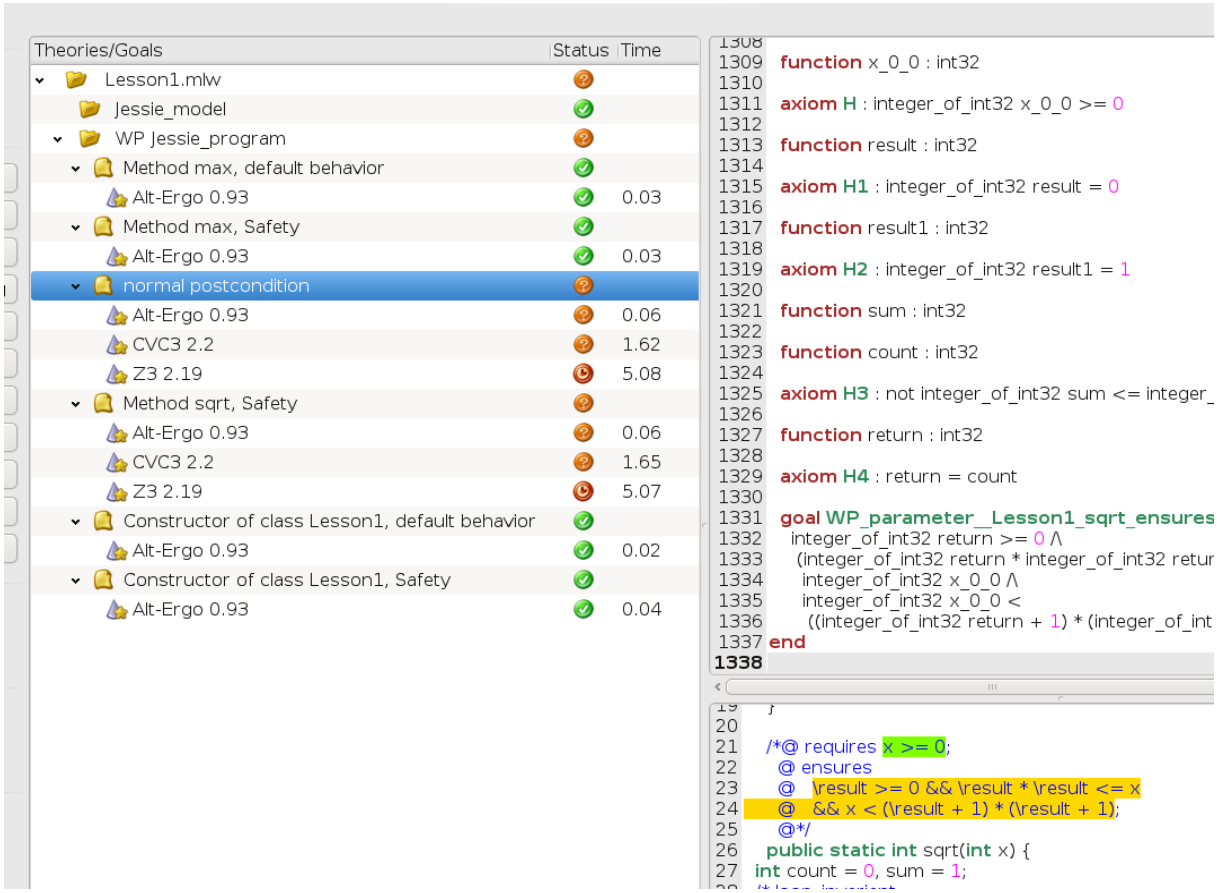

Figure 2.4: Verification conditions for `sqrt`

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);
      @*/
    while (sum <= x) {
        count++;
        sum = sum + 2*count+1;
    }
    return count;
}
```

If you rerun the krakatoa tool, you should now see that CVC3 and Z3 are able to prove the normal behevior of the method.

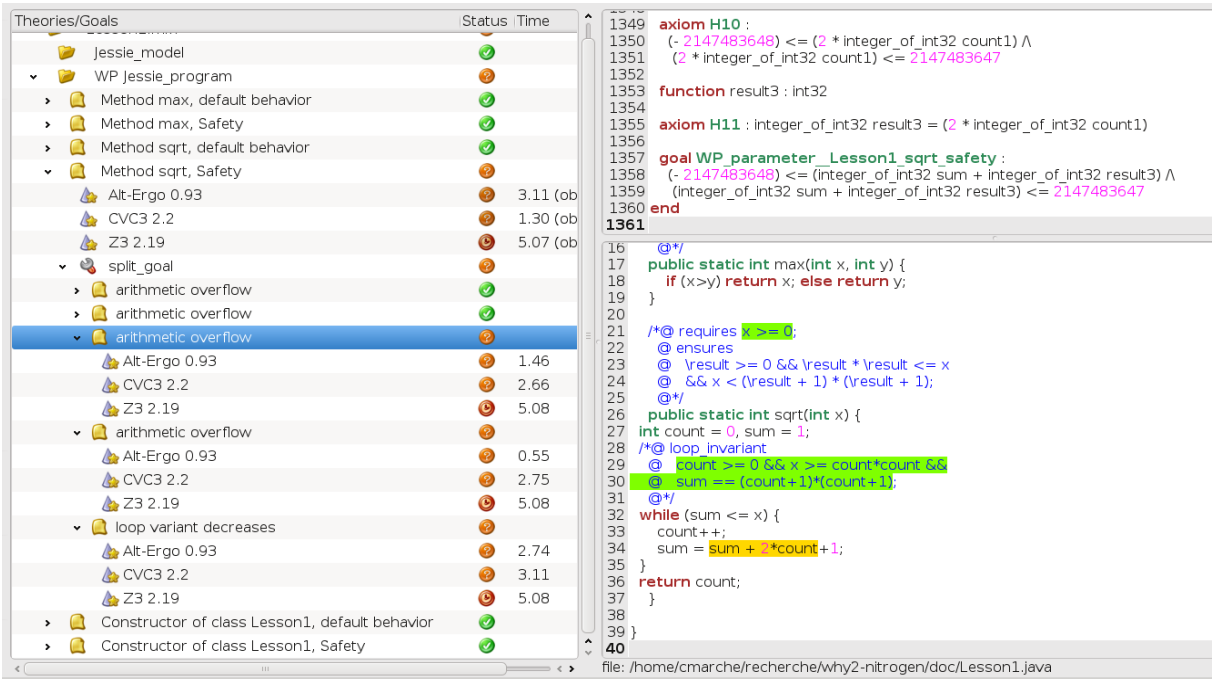

Figure 2.5: Safety VCs for `sqrt`

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 array access

To understand why safety is not proved, let's first split the goal and try provers on sub-goals. On this example, arithmetic overflow is the concern. On Figure 2.5, the focus is put on the VC corresponding to expression `sum+2*count` in the source code (bottom right windows). Looking at the top right window, the goal to prove, which is

```
(- 2147483648) <= (integer_of_int32 sum + integer_of_int32 result3)
               <= 2147483647
```

The constant 2147483648 is 2^{31}. This goal amounts to prove that the mathematical result of `sum + 2*count` fits into the `int` type. 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 exercise. 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 the "arithmetic overflow" VCs disappear.

Termination

There is another VC that remains to prove, named "loop variant decreases". Indded, since the `sqrt` method body contains a `while` loop, it is possible that the method execution does not terminate for some of the input. To prove termination, you can add another clause to the loop annotation: a `loop_variant` clause. An integer expression must follow, which is supposed to decreases at each loop iteration, and remaining non-negative. On the `sqrt` example, that `x-sum` is decreasing, so you can add:

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

You can check again that this is proved 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];
        @ loop_variant 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. See also http://toccata.lri.fr/gallery/krakatoa.en.html.

```
/*@ predicate is_max{L}(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) ;
          @ loop_variant 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. See also http://toccata.lri.fr/gallery/krakatoa.en.html.

```
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));
          @ loop_variant 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. See also http://toccata.lri.fr/gallery/krakatoa.en.html.

```
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

See also http://toccata.lri.fr/gallery/krakatoa.en.html.

```
    /*@ 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

See also http://toccata.lri.fr/gallery/krakatoa.en.html.

```
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

See also http://toccata.lri.fr/gallery/krakatoa.en.html.

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

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

/*@ predicate is_monochrome{L}(int t[],integer i, integer j, int c) =
  @   \forall integer 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 integer l; i <= l && l < k ==> t[l]==c);
              @ loop_variant 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 integer 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);
          @ loop_variant 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. See also http://toccata.lri.fr/gallery/krakatoa.en.html.

```
/* 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 ;
      @ loop_variant 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

This chapter is the reference for the specification language. It is unfortunately largely incomplete, but for missing parts and explanations we refer to the ACSL Reference Manual [2], which is very close.

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 `\forall`τ $x_1, \ldots, \tau$ x_n; e and existential quantification by `\exists`τ $x_1, \ldots, \tau$ 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.

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*$^+$ (`,` *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

3.1.1 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.

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 [3, 4].

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

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

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.

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

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

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 P1;
  @ requires P2;
  @ assigns L1;
  @ assigns L2;
  @ ensures E1;
  @ ensures E2;
  @*/
```

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 P1&&P2;
  @ assigns L1,L2;
  @ ensures E1&&E2;
  @*/
```

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 normal behavior 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

An exceptional behavior clause as the form

```
behavior id :
  assumes A ;
  assigns L ;
  signals (Exc) 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, using the clause `loop_variant`.

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. See the page http://why.lri.fr/provers.html

For using the Coq interactive 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 caught.

4.4 Contacts

The webpage for Krakatoa is at URL http://krakatoa.lri.fr/ and the webpage 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] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.4*, 2009. http://frama-c.cea.fr/acsl.html.

[3] 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.

[4] 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.

[5] 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