

# Algebraic Specifications for Java programs

Claude Marché

INRIA – Project *ProVal*

Université Paris sud, Orsay, France

# What this talk is about ?

- Unsolved problems about abstract datatype modeling, in JML
  - ◆ JML model classes not suitable for static verification
  - ◆ undefinedness
  - ◆ method calls in specifications
  - ◆ quantification over objects
  - ◆ ... [[Leavens & Leino & Müller, 2006](#)]
- How do we deal with that in Krakatoa/Caduceus platform ?

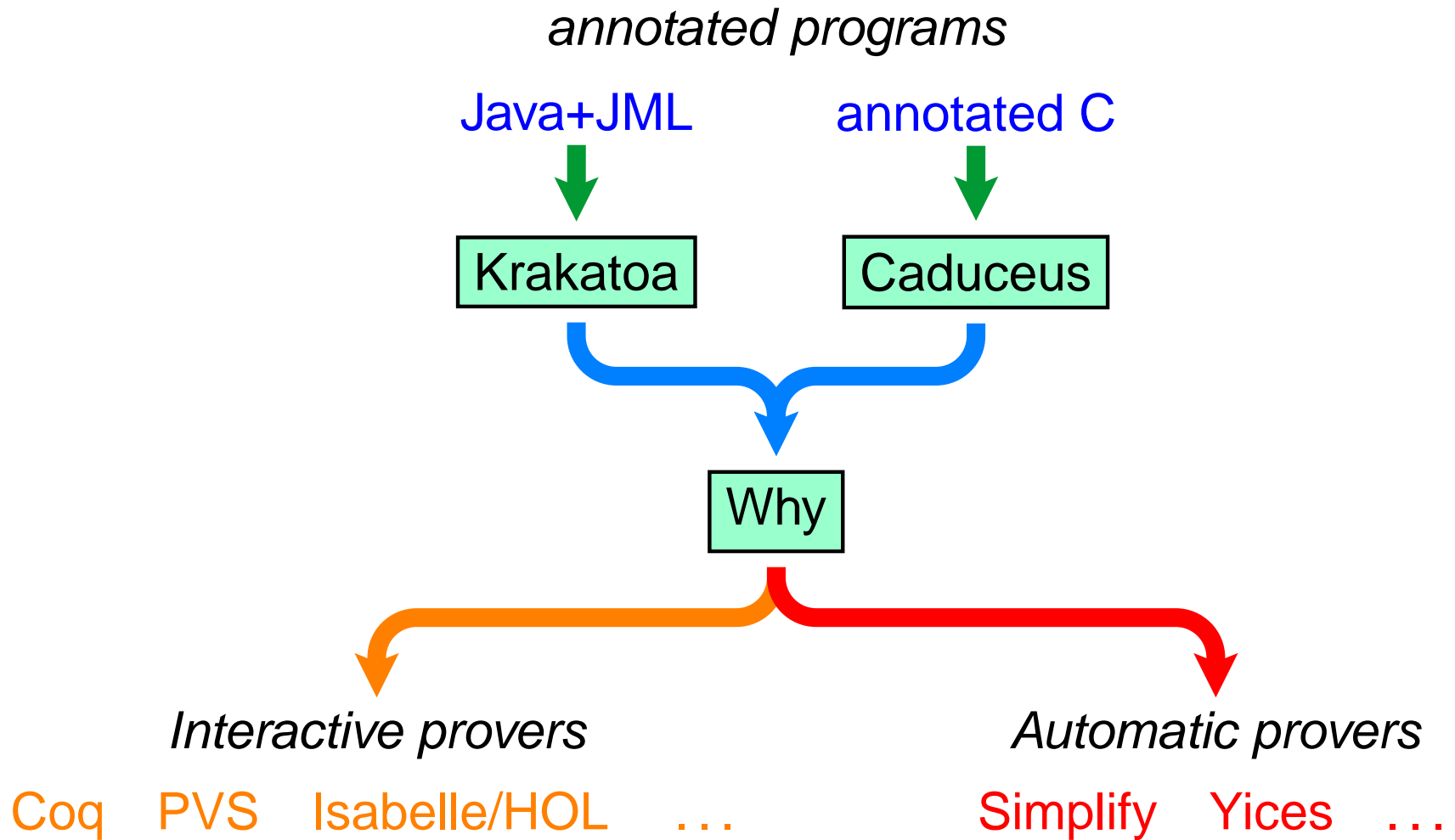
# Context

- ProVal research group develops tools for verification of Java or C source code
- Requirements specified as annotations in the source
- For Java : specifications given in JML (Java Modeling Language) → Krakatoa tool
- For C : home-made specification language → Caduceus tool
- These tools are freely available (open source):

`http://krakatoa.lri.fr`

`http://caduceus.lri.fr`

# Platform architecture



# Platform characteristics

- Multi-prover output
- Why intermediate language :
  - ◆ programming language: not important here
  - ◆ specification language
- Why specification language both for
  - ◆ expressing verification conditions
  - ◆ modeling Java objects, C pointers/structures, heap memory
  - ◆ expressing models of programs

# Why specification language

- Is “compatible” with target provers
- Features:
  - ◆ first-order logic
  - ◆ sorted
  - ◆ equality
  - ◆ built-in: booleans; integer and real arithmetic
- Modeling is done by
  - ◆ introducing functions and predicates
  - ◆ stating axioms
- Thus: **Algebraic specifications**

# Design choices

- Krakatoa, 2003:
  - ◆ ad-hoc interpretation of pure methods
- Caduceus, 2004:
  - ◆ allow first-order modeling at C source level
  - ◆ linked-list in-place reversal in C [Filliâtre & Marché, ICFEM 2004]
  - ◆ Schorr-Waite graph traversal in C [Hubert & Marché, SEFM 2005]
- Krakatoa, 2006:
  - ◆ allow first-order modeling similarly
  - ◆ incompatible with JML OO models

# Toy example

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

```
class IntSet {  
  
    // checks whether n belongs to this  
    public boolean mem(int n);  
  
    // adds n to this  
    public void add(int n);  
}
```

# Algebraic model (1)

- Sort, functions, predicates:

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

# Algebraic model (2)

## ■ Axiomatization:

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

# Specification of IntSet class

Introducing a “hybrid” predicate:

```
//@ predicate models(intset s, IntSet this);
```

Methods' behaviors:

```
class IntSet {  
  /*@ ensures (\forall intset s; models(s,this) ;  
    @      \result <==> in(n,s)) ;    */  
  public boolean mem(int n);  
  
  /*@ ensures  
    @      (\forall intset s; \old(models(s,this)) ;  
    @      models(union(s, singleton(n)), this)) ;  
  @*/  
  public void add(int n);  
}
```

# An implementation

By a sorted array

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

# Relating models and implementation

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

# Remarks

- Predicate definitions involve Java constructs:
  - ◆ object references
  - ◆ field and array accesses
- They depend on the state of the heap
- In Why specification language, hence in target provers they have extras arguments:
  - ◆ arrays corresponding to each field involved
  - ◆ 'component as array' model

# Delayed realization

- If necessary, logic functions and predicates may be implemented only into the target prover
- Example : linked-list model

```
/*@ predicate path(list l, ListNode n1,  
    @           ListNode n2)  
    @   reads n1.next;  
@*/
```

realized by an induction definition in Coq [Filliâtre & Marché, ICFEM 2004]

- Similar for graph reachability [Hubert & Marché, SEFM 2005]
- Similar idea in [Charles, FTfJP 2006]: ‘native’ specifications

# Pros

- Avoid many issues with OO specifications:
  - ◆ no undefinedness issue
  - ◆ no method calls, no need for pure methods
  - ◆ ...
- Suitable for provers
- May be more suitable for specifying libraries
- May allow natural use of Z, B, ... specifications

# Cons

- The dark side: hybrid predicates:
  - ◆ `reads` clause quite ad-hoc
  - ◆ semantics given using the underlying model of Java objects and heap memory
- Incompatible with JML, with RAC

# Conclusions

- For our platform, algebraic specifications provide a very useful alternative to JML OO models
- Case study in progress: behavioral properties of Java Card applets (with Nicolas Rousset, Gemalto)
- Open questions:
  - ◆ how general is this approach?
  - ◆ could it be added to JML standard?