Specification, Verification, and Proofs for Java

Erik Poll

Digital Security

Radboud University Nijmegen

presented at FOSAD'2009, Bertinoro, Italy
supported by the EU IST project Mobius

Overview

- Context - Proof Carrying Code (PCC)
- The JML specification language for Java
- The Mobius PCC infrastructure for Java
- Applications & case studies

The problem

Is it ok to run this *untrusted, mobile* code on my phone?

mobile code is an oxymoron, all code is mobile

Potential problems

The untrusted code may

- damage the system or information stored on the system (integrity, availability)
- use up *limited* resources (availability)
  - CPU, (persistent) memory, ..
- consume *billable* resources ($$$)
  - SMS text messages, phone calls, bandwidth
- reveal confidential information
  - phonebook, location, camera, diary, credit card no, ... (confidentiality)

Potential solutions

1. trusting the code producer
   - eg. using digital signatures
1. baby-sitting/monitoring the application
   - runtime
   - eg. by OS or Java sandbox
1. static analysis/formal verification
   - compile-time (or load time)
   - eg. type checking, formal program verification

For 2 and 3 we need security requirements or security policy, for 1 we don't

- coming up with this is hard!

1. How to trust a code producer?

- Use a public key infrastructure (PKI) & digital signatures
- Certification of the code producers
  - ISO9000, CMM, Visa Certified, MasterCard Secured, ...
- Certification of the actual code
  - eg Common Criteria, ITSEC
- Common interests with code producer
  - telco operator will feel the pain of customer security problems (eg. cost of helpdesk, lost customers & reputation)
  - Possibility to sue the code producer could make a real difference!
- Not just a problem for end user, but eg. also for company buying/integrating software from third party

2. Baby-sitting / runtime monitoring

Examples

- Operating system access control
- Java sandbox
- runtime typechecks, eg by VM for array bounds
- reference monitors
- security automata
- ...

2. Baby-sitting / runtime monitoring

Pros

+ relatively simple, and hence trustworthy
+ works for many properties

Cons

- some runtime overhead
- only catches problems at the very last moment
  - can be annoying: eg think of game that is only allowed to use 1Mbyte of memory.
- limits to what can be enforced
  - eg information flow policies

Involving the end user in the process – by security pop-ups – has only very limited value

3. Verification

In the broad sense

- to guarantee that an application meets some security policy
- prior to execution, eg. at compile time or load time

Examples:

- type systems
  - eg byte code verifier (bcv)
- formal program verification
  - formally verify that program meets some (partial) specification, expressed in formal specification language, by logic reasoning
  - type systems are simple & highly successful forms of program verification

Examples of "verification"

- source code analysis/static checkers/code scanners

  looking for bugs or suspicious code,

  - by simple grep-ing or
    - eg looking for *gets* in C code
  - or something more sophisticated
    - dataflow analysis, control flow analysis,...

  Eg RATS, ITS4E, FindBugs, PMD, PREfast, Fortify, jTest...

3a. Typing

Pros

+ simple and widely used
+ accepted by programmers
+ catches problems early (at compile/load time)

Cons

- limited expressivity

3b. Formal program verification

Pros

+ no runtime overhead
+ catches problems early (at compile/load time)
+ very expressive
  - expressing something interesting correctly may be hard!

    ("Every advantage has a disadvantage", Johan Cruijff)

Cons

- complicated formal infrastructure needed as foundation
- huge TCB, including a theorem prover + logical theories for it
- lots of work

Ingredients for program verification

- formalisation of programming language that allows reasoning
  - eg operational semantics or program logic
  - incl. any APIs used
- specification language
  - to express properties of interest
- policy
  - that we want a particular program to meet
- proof
  - that a particular program meets a policy
- a theorem prover
  - to do all this in...

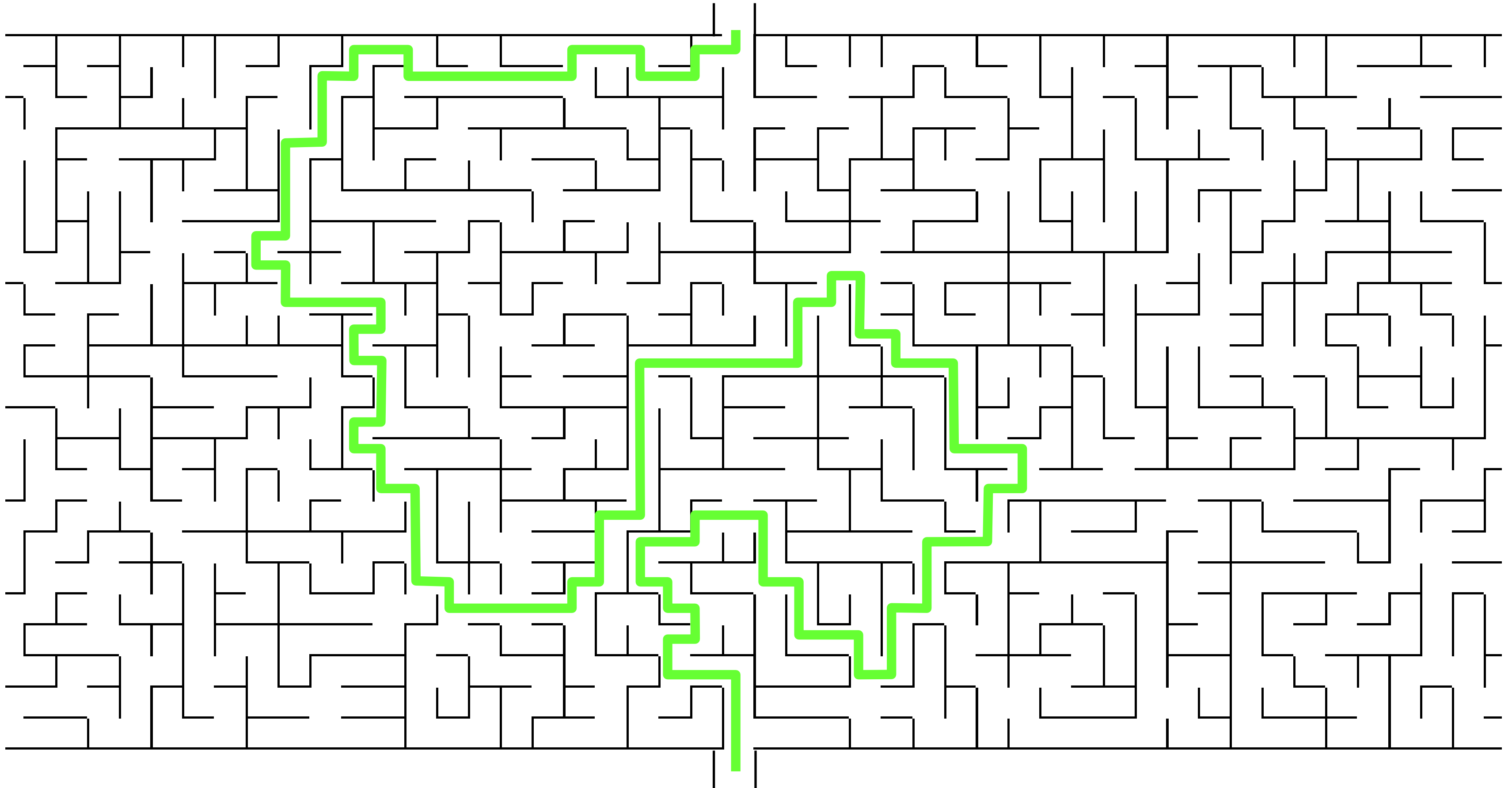
Verifying downloaded code before running it is a lot of work...

Proof-Carrying Code (PCC)

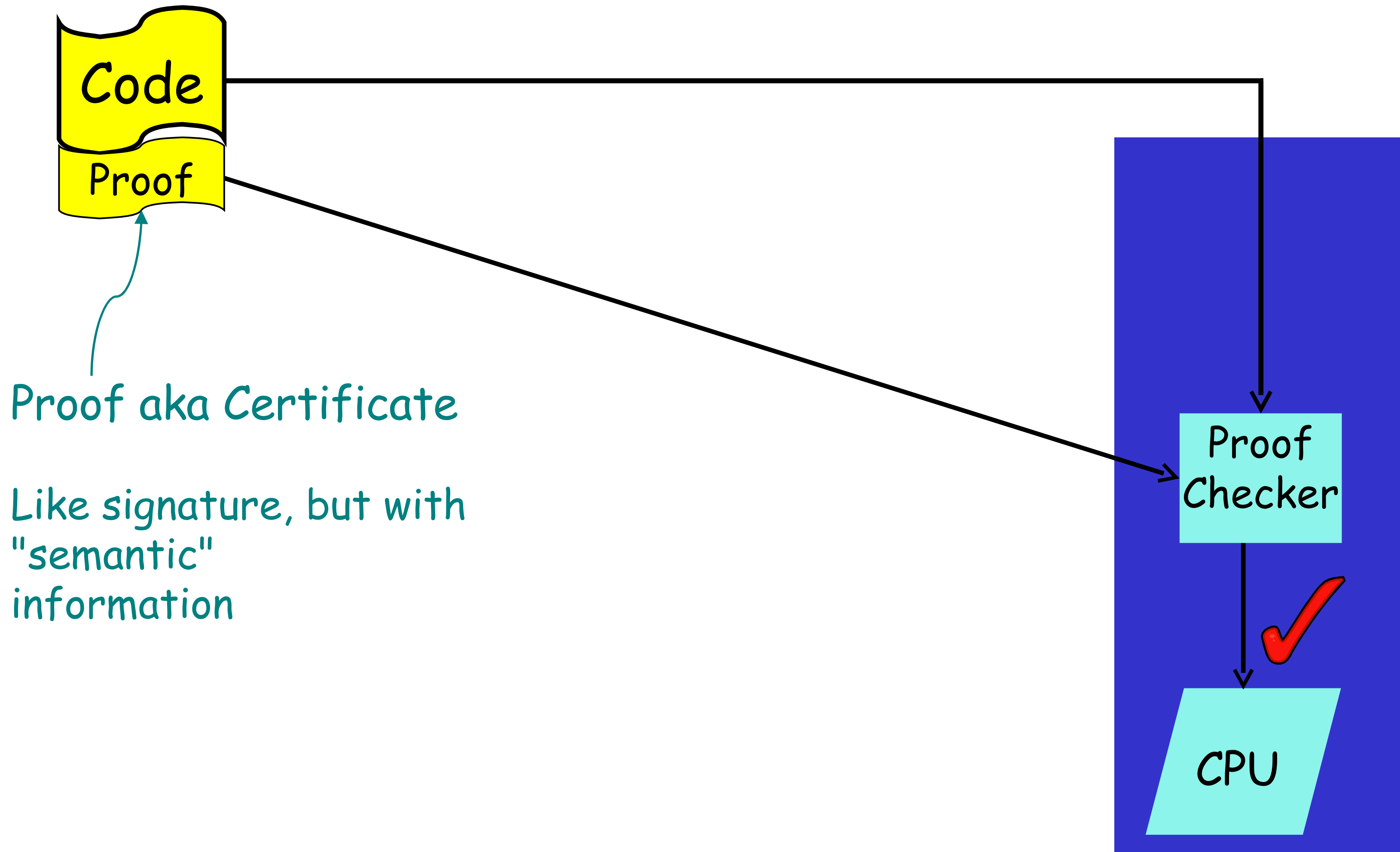
- Introduced by George Necula and Peter Lee
  - 'Safe Kernel Extensions Without Runtime checking', OSDI'96
- A way to make program verification workable in practice, by
  - reducing the amount of work for the code consumer
  - reducing the size of the TCB
- Original application: certifying that binaries are memory-safe
  - no access out of array bounds, no reading of uninitialised memory

The basic observation behind PCC

Finding proof is hard, but checking proof is a lot simpler



Proof-Carrying Code (PCC)



PCC

Pros

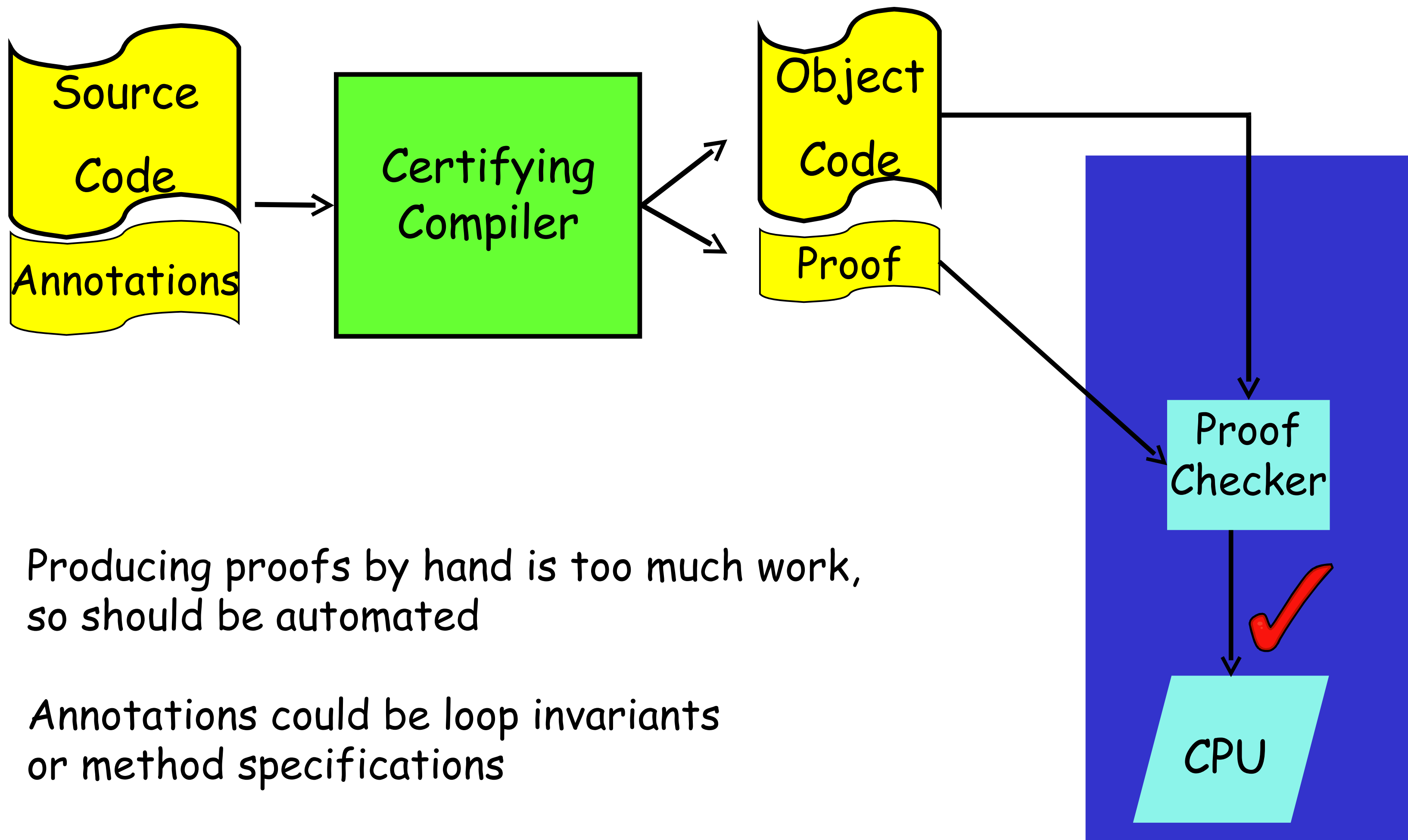
+ very expressive
  - though first examples look at relatively weak properties, namely memory safety
+ no runtime overhead
+ catches problems early (at compile/load time)
+ smaller (but still large) TCB than program verification
  - proof *checker* instead of theorem *prover*

Cons

- complicated formal infrastucture needed as foundation
- proof terms can be huge (100s times the code)

-/+ less work for code consumer, more for code producer
but we can mitigate this by certifying compiler

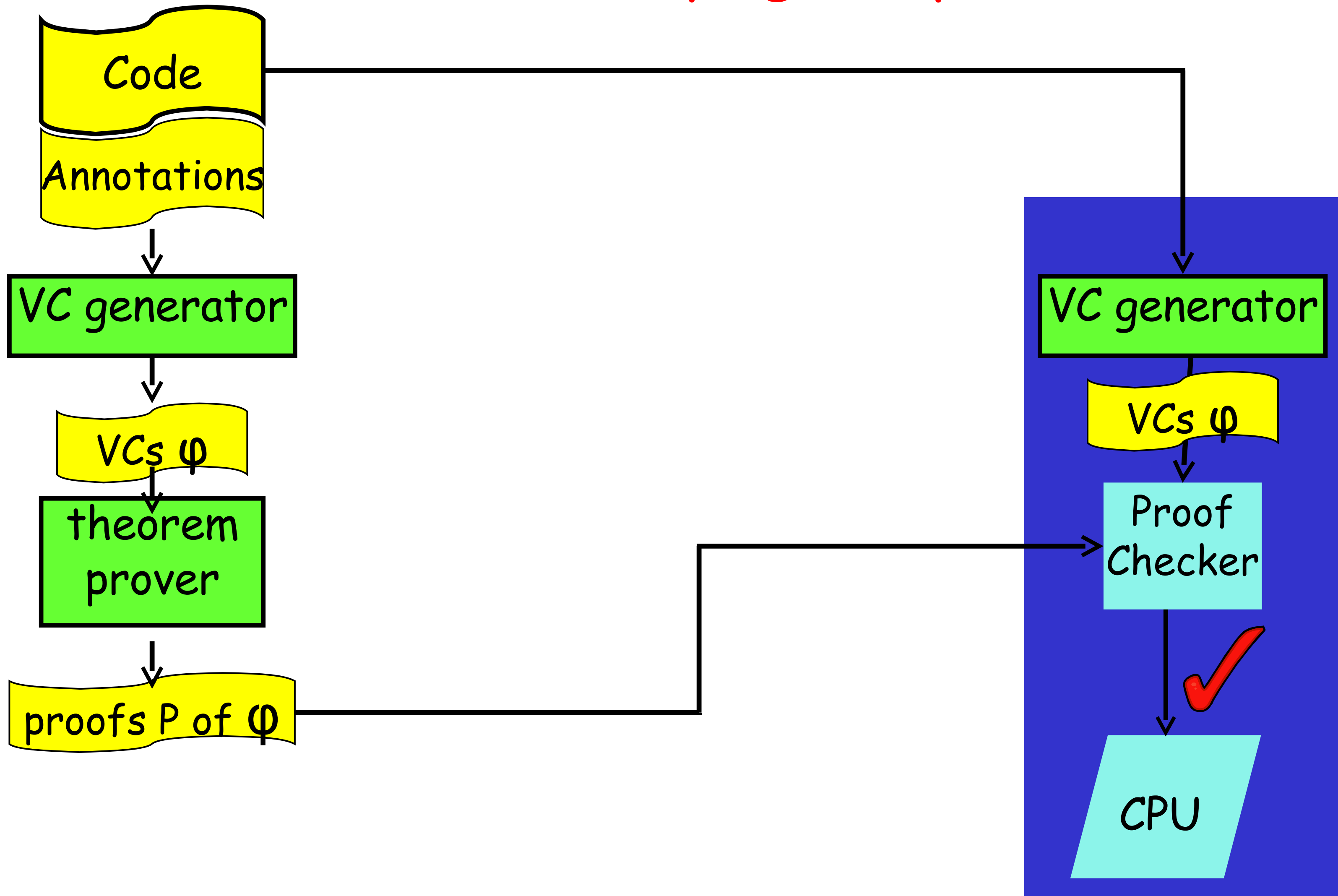
PCC using a certifying compiler



Producing proofs by hand is too much work, so should be automated

Annotations could be loop invariants or method specifications

inside certifying compiler



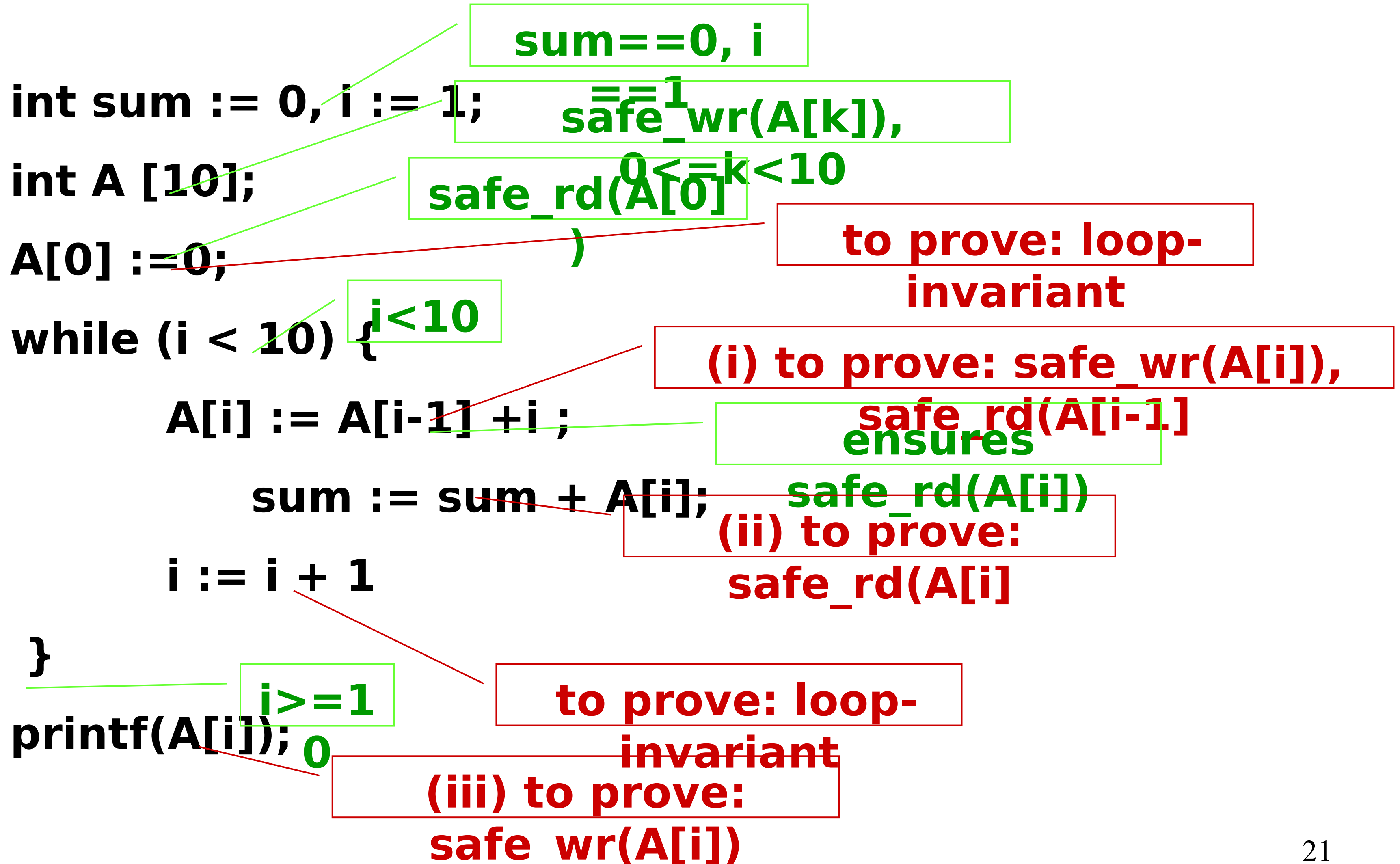
An example: proving memory safety

```
int sum := 0, i := 1;
int A [10];
A[0] :=0;
while (i < 10) {
        A[i] := A[i-1] +i ;
            sum := sum + A[i];
        i := i + 1
 }
printf(A[i]);
```

is this program memory safe?

ie. does it (a) stay inside array bounds and (b) not read uninitialised data?

An example



PCC successes

- The Touchstone Java compiler produces machine code for Intel x86 with certificates that these are memory safe.
  [Colby, Lee, Necula, Blua, Plesko, Kline. *A certifying compiler for Java.* PLDI'00]

- Sun's KVM (JVM for embedded devices) uses certificates for lightweight bytecode verification
  [Eva Rose, Lightweight Bytecode Verification, Journal of Automated Reasoning, 2003]
  - bvc requires computation of a fixpoint for the dataflow analysis
  - *lightweight* bcv supplies the fixpoint in form of partial type annotations
    - instead of *computing* a fixpoint, we only have to *check* that the given fixpoint is a fixpoint
  - aka Abstraction Carrying Code (ACC)

TCB (Trusted Computing Base)

We don't have to trust

- the compiler
- the annotations
- the prover

We do have to trust

- the VCGen
- the proof checker
- the CPU

For example, Touchstone's VCGen is 23,000 lines of C...

To reduce the TCB, one could try to formally verify the VCGen or the proof checker

Reducing the TCB

- Foundational PCC: get rid of the VCGen and give correctness proofs wrt an formal operational semantics

  [Andrew Appel and Amy Felty, A semantic model of typed and machine instructions for PCC, POPL'00]

  – this allows arbitrary properties to be certified, not just those that VC generation can deal with

    - eg non-interference

- Reflective PCC: prove the correctness of some executable checker wrt the operational semantics

  – reducing certificate size

  – faster checking of certicates

Mobius project

Goals include

- certified PCC infrastructure for sequential Java, incl.
  - formal operational semantics in Coq: Bicolano
  - certified executable checkers wrt this semantics
  - not just traditional PCC for fixed safety properties, but certification of arbitrary (eg functional) properties :
    - using the Java specification language JML and its bytecode counterpart BML
- Java midlets (ie mobile phone) for case studies

More Mobius-related talks, by German Puebla and David Pichardie

Overview

- Context & Proof Carrying Code (PCC)
- The JML specification language for Java
- The Mobius PCC infrastructure for Java
- Applications & case studies

JML

JML (Java Modeling Language)

- formal specification language for sequential Java by Gary Leavens et. al.
  - to specify behaviour of Java classes
  - to record detailed design decisions

  by adding annotations to Java source code in Design-By-Contract style, using eg. pre/postconditions and invariants

- Design goal: meant to be usable by any Java programmer

Lots of info on http://www.jmlspecs.org

Example

```
public class ePurse{
 private int balance;
 //@ invariant 0 <= balance && balance < 500;

 //@ requires amount >= 0;
 //@ ensures balance <= \old(balance);
  public debit(int amount) {
    if (amount > balance) {
          throw (new BankException("No way"));}
    balance = balance - amount;
 }
```

To make JML easy to use

- JML annotations added as special Java comments, between `/*@ .. @*/` or after `//@`

- JML specs can be in .java files, or in separate .jml files

- Properties specified using Java expressions, extended with some operators

  `\old( ), \result, \forall, \exists, ==> , ..`

  and some keywords

  `requires, ensures, invariant, ....`

Exceptional postconditions: signals

```
//@ requires amount >= 0;
//@ ensures balance <= \old(balance);
//@ signals (BankException) balance == \old(balance);
 public debit(int amount) {
   if (amount > balance) {
         throw (new BankException("No way"));}
   balance = balance - amount;
}
```

assert and loop_invariant

Inside method bodies, JML allows

- assertions

```
/*@ assert (\forall int i; 0<= i && i< a.length;
                          a[i] != null );
@*/
```

- loop invariants

```
/*@ loop_invariant 0<= n && n < a.length &
            (\forall int i; 0<= i & i < n;
                          a[i] != null );
@*/
```

Tool support: runtime assertion checking

- implemented in JMLrac, with JMLunit extension
- annotations provide the test oracle:
  - any annotation violation is an error,
    except if it is the initial precondition
- Pros
  - Lots of tests for free
  - Complicated test code for free, eg for
    ```
    signals (Exception) balance == \old(balance);
    ```
    and even for `\forall` if domain is finite
  - More precise feedback about root causes
    - eg "Invariant X violated in line 200" after 10 sec instead of "Nullpointer exception in line 600" after 100 sec

Hence testing can be largely automated, simply by throwing random inputs at the code

Tool support: compile time checking

- extended static checking

  automated checking of simple specs, deliberately sacrificing soundness

  - ESC/Java(2)

- program verification tools

  sound, interactive checking of arbitrarily complex specs

  - KeY, Krakatoa, JACK, Jive, LOOP, JML2BML;BMLVCGEN,...

In practice, each tool support its own subset of JML...

Related work

- Spec# for C#
  by Rustan Leino & co at Microsoft Research

- SparkAda for Ada
  by Praxis High Integrity System
  Commercially used!

Towards a usable, formal specification language for Java?

- Designing a specification language for Java involves
  - lots of details and subtle semantics issues
    - even for apparently simple notions
  - lots of features that seem to be needed

Exercise: JML specification for arraycopy

```
/*@ requires ... ;
     ensures ... ;
  @*/
static void arraycopy (int[] src,  int srcPos,
                       int[] dest, int destPos,
                       int len)
 throws NullPointerException,
        ArrayIndexOutOfBoundsException;
```

Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array.

Exercise: JML specification for arraycopy

```
/*@ requires src != null && dest != null &&
             0 <= srcPos && srcPos + len < src.length &&
             0 <= destPos && srcPos + len < dest.length;

    ensures (\forall int i; 0 <= i && i < len;
                 dest[dstPos+i] == src[srcPos+i] ) &&
              (* rest unchanged *)
  @*/
 static void arraycopy (int[] src,  int srcPos,
                        int[] dest, int destPos,
                        int len);
```

Exercise: JML specification for arraycopy

```
/*@ requires src != null && dest != null &&
             0 <= srcPos && srcPos + len < src.length &&
             0 <= destPos && srcPos + len < dest.length;

    ensures (\forall int i; 0 <= i && i < len;
                 dest[dstPos+i] == \old(src[srcPos+i])) &&
              (* rest unchanged *)
  @*/
 static void arraycopy (int[] src,  int srcPos,
                        int[] dest, int destPos,
                        int len);
```

Exercise: JML specification for arraycopy

```
/*@ requires ...

     ensures (\forall int i; 0 <= i && i < len;
                   dest[dstPos+i] == \old(src[srcPos+i])) &&
               (* rest unchanged *)
   @*/
 static void arraycopy (int[] src,  int srcPos,
                        int[] dest, int destPos,
                        int len);
```

We don't have to write `\old(len)` and `\old(dest)[\old(dstPos)+1]` in the postcondition, because all parameters are implicily \old() in JML postconditions

Defaults and conjoining specs

- Default pre- and postconditions
  ```
  //@ requires true;
  //@ ensures true;
  ```
  can be omitted

- ```
  //@ requires P
  //@ requires Q
  ```
  means the same as
  ```
  //@ requires P && Q;
  ```
  ```

Default signals clause?

```
//@ requires amount >= 0;
//@ ensures balance <= \old(balance);
public debit(int amount) throws BankException
```

- Can debit throw a BankException, if precondition holds?
  YES
- Can debit throw a NullPointerException, if the precondition holds?
  NO. Unlike Java, JML *only* allows method to throw unchecked exceptions explicitly mentioned in throws-clauses!
- Methods *are* always allowed to throw Errors

Default signals clause?

- For a method

```
//@ public void m throws E1, ... En { ... }
```

  the default is

```
//@ signals (E1) true;
      ...
//@ signals (En) true;
//@ signals_only E1, ... En;
```

- Here

```
//@ signals_only E1, ... En;
```

  is shorthand for

```
/*@ signals (Exception e)
       \typeof(e) <: E1 || ... || \typeof(e) <: En;
  @*/
```

Specifying exceptional behaviour is tricky!

- Beware of the difference between
  1. if P holds then exception E *must* be thrown
  2. if P holds then exception E *may* be thrown
  3. if exception of type E is thrown then P will hold
     (in the poststate)

     This is what **`signals`** specifies

- Most often we just want to rule out exceptions
  - and come up with preconditions and invariants to do this

- Ruling out exceptions also helps with certified analyses for PCC, as it rules out many execution paths

requiring & ruling out exceptions

```
/*@  requires amount <= balance;
      ensures ...;
      signals (Exception) false;
   also
     requires amount > balance;
      ensures false;
      signals (BankException) ...;
  @*/

 public debit(int amount) throws BankException
```

requiring & ruling out exceptions

```
/*@ normal_behavior
      requires amount <= balance;
      ensures ...;
   also
    exceptional_behavior
     requires amount > balance;
      signals (BankException) ...;
  @*/
 public debit(int amount) throws BankException
```

requiring & ruling out exceptions

or simply

```
/*@ requires amount <= balance;
     ensures ...;
   @*/
  public debit(int amount) // throws BankException
```

Effectively a `normal_behavior`, since there is no throws clause

Ruling out exceptions, esp. RuntimeExceptions, as much as possible is the natural thing to do - and a good bottom line specification

Visibility and spec_public

The standard Java visibility modifiers (public, protected, private) can be used on invariants and method specs, eg

```
//@ private invariant 0 <= balance;
```

Visibility of fields can be loosened using the keyword spec_public, eg

```
public class ePurse{
   private /*@ spec_public @*/ int balance;

   //@ ensures balance <= \old(balance);
   public debit(int amount)
```

allows private field to be used in (public) spec of **debit**

Of course, this exposes implementation details, which is not nice...

Dealing with undefinedness

- Using Java syntax in JML annotations has a drawback
  - what is the meaning of

    `//@ requires !(a[3] < 0);`

    if `a.length == 2` ?
- How to cope with Java expressions that throw exceptions?
  - runtime assertion checker can report the exception
  - program verifier can treat `a[3]` as unspecified integer

- Moral: write protective specifications, eg

  `//@ requires a.length > 4 && !(a[3] < 0);`

non_null

- Lots of invariants and preconditions are about reference not being null, eg

```
int[] a; //@ invariant a != null;
```

- Therefore there is a shorthand

```
/*@ non_null @*/ int[] a;
```

- But, as most references are non-null, JML adopted this as default, and only *nullable* fields, arguments and return types need to be annotated, eg

```
/*@ nullable @*/ int[] b;
```

- JML will move to adopting JSR308 Java tags for this

```
@Nullable int[] b;
```

pure

Methods without side-effects that are guaranteed to terminate can be declared as pure

```
/*@ pure @*/ int getBalance (){
    return balance;
};
```

Pure methods can be used in JML annotations

```
//@ requires amount < getBalance();
public debit(int amount)
```

Subtle semantic issues:

- is pure method allowed to allocate & modify new memory?
- is a constructor pure, if it only initialises its newly allocated memory?

Yes, but disallowing such 'weakly pure' methods may simplify life [Adam Darvas and Peter Muller, Reasoning About Method Calls in JML Specications, Journal of Object Technology, 2006]

assignable (aka modifies)

For non-pure methods, frame properties can be specified using assignable clauses, eg

```
/*@   requires amount >= 0;
      assignable balance;
         ensures balance == \old(balance) - amount;
  @*/
void debit()
```

says **debit** is *only* allowed to modify the **balance** field

- NB this does *not* follow from the postcondition
- Assignable clauses are needed for modular verification!
- Still, these static frame conditions are not the last word on the subject...

assignable

The default assignable clause is

```
//@ assignable \everything;
```

Pure methods are

```
//@ assignable \nothing;
```

Pure constructors are

```
//@ assignable this.*;
```

Reasoning in presence of late binding

Late binding (aka dynamic dispatch) introduces a complication in reasoning:

which method specification do we use to reason about

....; x.m(); ....

if we don't know the dynamic type of x?

Solutions:

1. do a case distinction over all possible dynamic types of x,
   - ie. x's static type A and all its subclasses

   Obviously not modular!
1. insist on behavioural subtyping:
   - use spec for m in class A and require that specs for m in subclasses are stronger or identical

Behavioural subtyping & substitutivity

- The aim of behavioural subtyping aims to ensure the principle of subsitutivity:

  "substituting a subclass object for a parent object will not cause any surprises"

- Well-typed OO languages already ensure this in a weak form, as soundness of subtyping:

  "substituting a subclass object for a parent object will not result in 'Method not found' errors at runtime"

behavioural subtyping

Two ways to achieve behavioural subtyping

1. For any method spec in a subclass, prove that it is implies the spec for that method in the parent class
   - ie prove that the precondition is *weaker* !

     and the postcondition is stronger
1. Implicitly conjoin method spec in a subclass with method specs in the parent class
   - called specification inheritance, which is what JML uses
   - this guarantees that resulting precondition is weaker, and the resulting postcondition is stronger

Specification inheritance for method specs

Method specs are inherited in subclasses, and required keyword `also` warns that this is the case

```
class Parent {
    //@ requires i >=0;
    //@ ensures  \result >= i;
    int m(int i) {...}
}
class Child extends Parent {
    //@ also
    //@  requires i <= 0;
    //@  ensures  \result <= i;
    int m(int i) {...}
}
```

Effective spec of **m** in Child:

```
requires true;
ensures
  (i>=0 ==> result>=i)
&&
  (i<=0 ==> result<=i);
```

Avoiding behavioural subtyping

Sometimes you have to specify something not to be necessarily inherited by subclasses (unfortunately..)

```
public class Object {
    //@ ensures  \result == (this == o);
    public boolean equals(Object o) {...}
    ...
}
```

Trick to do this:

```
ensures  \typeof(this) == \type(Object)
          ==>
          \result == (this == o);
```

Specification inheritance for invariants

Invariants are inherited in subclasses, eg in

```
class Parent {
    //@ invariant invParent;
    ...
}

class Child extends Parent {
    //@ invariant invChild;
    ...
}
```

the invariant for the Child is `invChild && invParent`

JML
invariants

The semantics of invariants

- Basic idea:
  - Invariants have to hold on method entry and exit
  - but may be broken temporarily during a method

- NB invariants also have to hold if an exception is thrown!

- But there's more to it than that...

The callback problem

```
class A {
  int i;
  int[] a;
  B b;
  //@ invariant 0<=i && i<
    a.length;

  void inc() {a[i]++; }

  void break() {
    int oldi = i; i = -1;
    b.m(); i = oldi;
  }
```

```
class B {
 A a;

 void m() {
    a.inc(); // possible callback
 }

}
```

invariant temporarily broken

What if b.m() does a callback on **inc** of that *same* A object, while its invariant is broken...

The semantics of invariants

- An invariant can be temporarily broken during a method, but – because of the possible callbacks - it has to hold when any other method is invoked.

- Worse still, one object could break another object's invariant...

- visible state semantics

  all invariants of *all* objects have to hold in *all* visible states, ie. entry and exit points of methods

Problems with invariants

- The visible state semantics is *very* restrictive
  - eg, a constructor cannot call out to other methods before it has established the invariant

It can be loosened in an ad-hoc manner by declaring methods as **`helper`** methods

  - **`helper`** methods don't require or ensure invariants
  - effectively, you can think of them as in-lined

- The more general problem: how to cope with invariants that involve multiple (or aggregate) objects
  - still an active research area...
  - one solution is to use some notion of object ownership

universes & relevant invariant semantics

Current JML approach to weakening visible state semantics for invariants

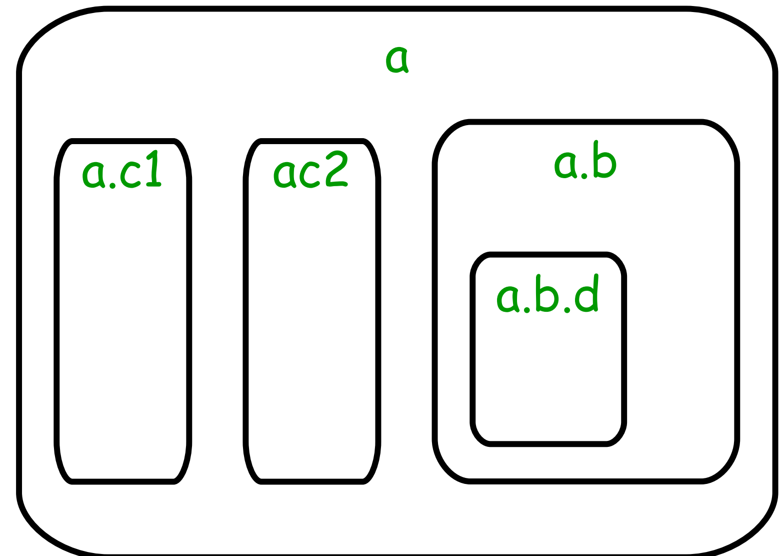
- universe type system
  - enforces hierachical nesting of objects

- relevant invariant semantics
  - invariant of outer objects may be broken when calling methods in inner objects

universes & relevant invariant semantics

```
class A {
  //@ invariant invA;
  /*@ rep @*/ C c1, c2;
  /*@ rep @*/ B b;
}

class B {
   //@ invariant invB;
   /*@ rep @*/ D d;
}
```



- invariants should only depend on owned state
- an object's invariant may be broken when it invokes methods on sub-objects

The problems with invariants

Alternative approaches to coping with invariants

- the Boogie methodology
- explicitly tracking & specifying dependencies on invariants
- dynamic frames
- separation logic
- ...

Composing objects to construct bigger objects is a (the?) core idea of OO, but real OO languages don't make any guarantees

every object is somewhere on the heap, and can refer to all other objects...

Overview

- Context & Proof Carrying Code (PCC)
- The JML specification language for Java
- The Mobius PCC infrastructure for Java
- Applications & case studies

Mobius PCC infrastructure

Mobius project

- certified PCC for sequential Java
- basis for everything
  - formal operational semantics in Coq: Bicolano
- certified executable checkers
  - for *specific* safety properties
  - eg talk by David Piccardie earlier today
- certified Verification Condition Generator (VCGen)
  - for *arbitrary* properties expressible in JML
    - or its bytecode counterpart BML

Overview in [Gilles Barthe et al. The MOBIUS Proof Carrying Code Infrastructure (An overview), FMCO'2007]

The Coq theorem prover

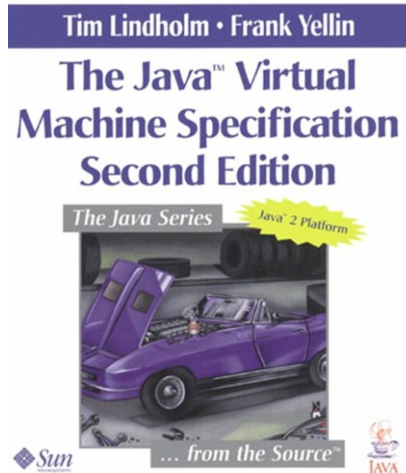
- Coq is a mechanical proof assistant based on higher order type theory
- This type theory allows
  - definition of mathematical objects & concepts
  - formulation and proving of associated theories
  - computations on the mathematical objects
    - ie it includes a functional program language
- Coq characteristics
  + Very expressive
  + Small TCB: xompleted proofs can be represented as proof objects that can be checked by small proof checker
  - Little automation
    - esp. compared to fast SAT solvers and SMT prover, or PVS

Formal language semantics

Basis for everything: a formal language sematics of Java

- operational semantics for Java bytecode,

  which formalises



in theorem prover Coq

Bicolano Java semantics: the JVM state

- JVM state can be formalised as

  `(h, (m,pc,os,l), cs)`

  - heap `h`
  - current stack frame `(m,pc,os,l)` consisting of
    - method name `m`
    - program counter `pc`
    - operand stack `os`
    - local variables `l`
  - call stack `cs`
    - list of stack frames
- special JVM states needed for exceptional states

  `((h, (m,pc,exp,l),cs)`

where `exp` is location of exception object (on the heap)

Bicolano: small-step semantics for bytecode

```
Inductive step (p:Program): State → State → Prop :=
 ...
| getfield_step_ok : ∀ h m pc pc' s l sf loc f v cn
   instructionAt m pc = Some (Getfield f) →
    next m pc = Some pc' →
    Heap.typeof h loc = Some (Heap.LocationObject cn) →
    defined_field p cn f →
    Heap.get h (Heap.DynamicField loc f) = Some v →
        step p (h (m pc  (Ref loc::s) l) sf)
               (h (m pc' (v::s)       l) sf)
```

[Whole semantics online at
http://mobius.inria.fr/twiki/pub/Bicolano/WebHome/SmallStepType.html]

Defensive vs "trusting" VM

Operational semantics can be defined in two styles:

1. defensive
   - VM state includes all type information, and execution performs all type checks
1. "trusting"
   - VM trusts the code to be well-typed

   Even offensive VM will do some runtime checks:
     - for non-nullness, arraybounds and downcasts

Having both allows a proof of soundness of bytecode verification

prove that all programs that pass the bcv execute the same on both VMs

Certified Analyses

The Bicolano semantics has been used for developing
certified checkers

- ie checker proven sound wrt operational semantics

incl.

- certified information flow verifier
  - using non-interference to characterise information flow

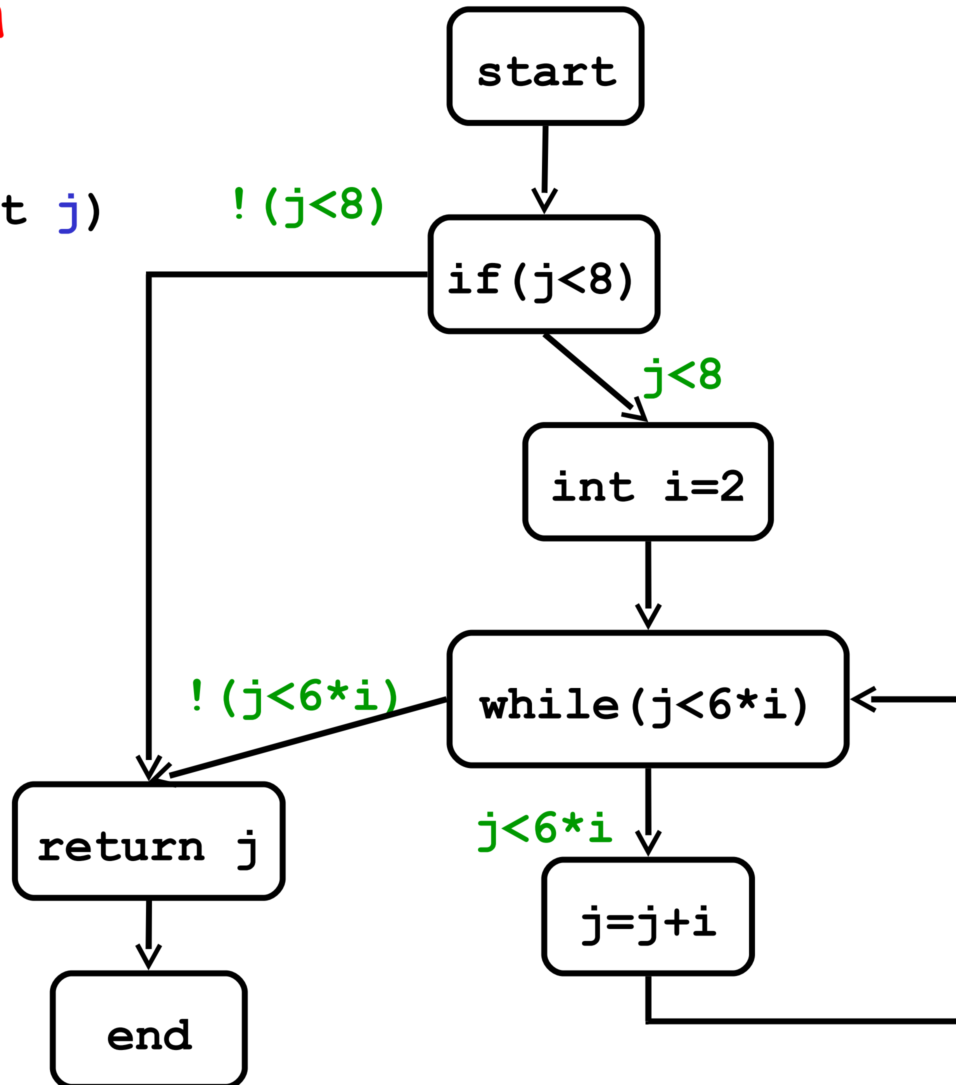
  [Gilles Barthe, David Pichardie and Tamara Rezk , A Certified Lightweight Non-interference Java Bytecode Verifier, ESOP 2007]

  These checkers exists then exists as
  - function that can evaluated inside Coq
  - extracted O'Caml program
- certified verification condition generator

  [Benjamine Gregoir and Jorge Luis Sacchini, Combining a verification condition generator for a bytecode language with static analyses]

Verification using VCGen
(i) program as graph

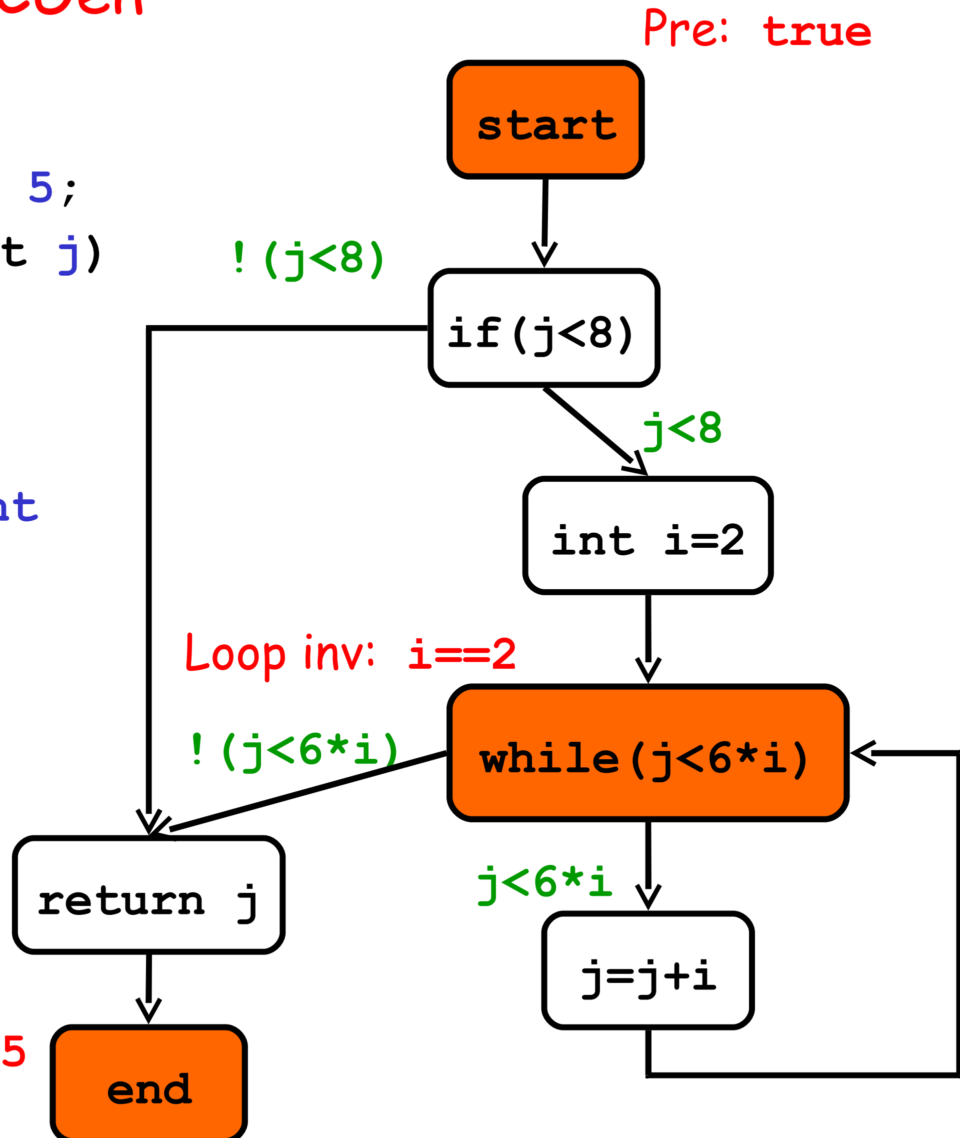
```
public int example(int j)
{
  if (j < 8) {
    int i = 2;
    while (j < 6*i){
      j = j + i;
    }
  }
  return j;
}
```



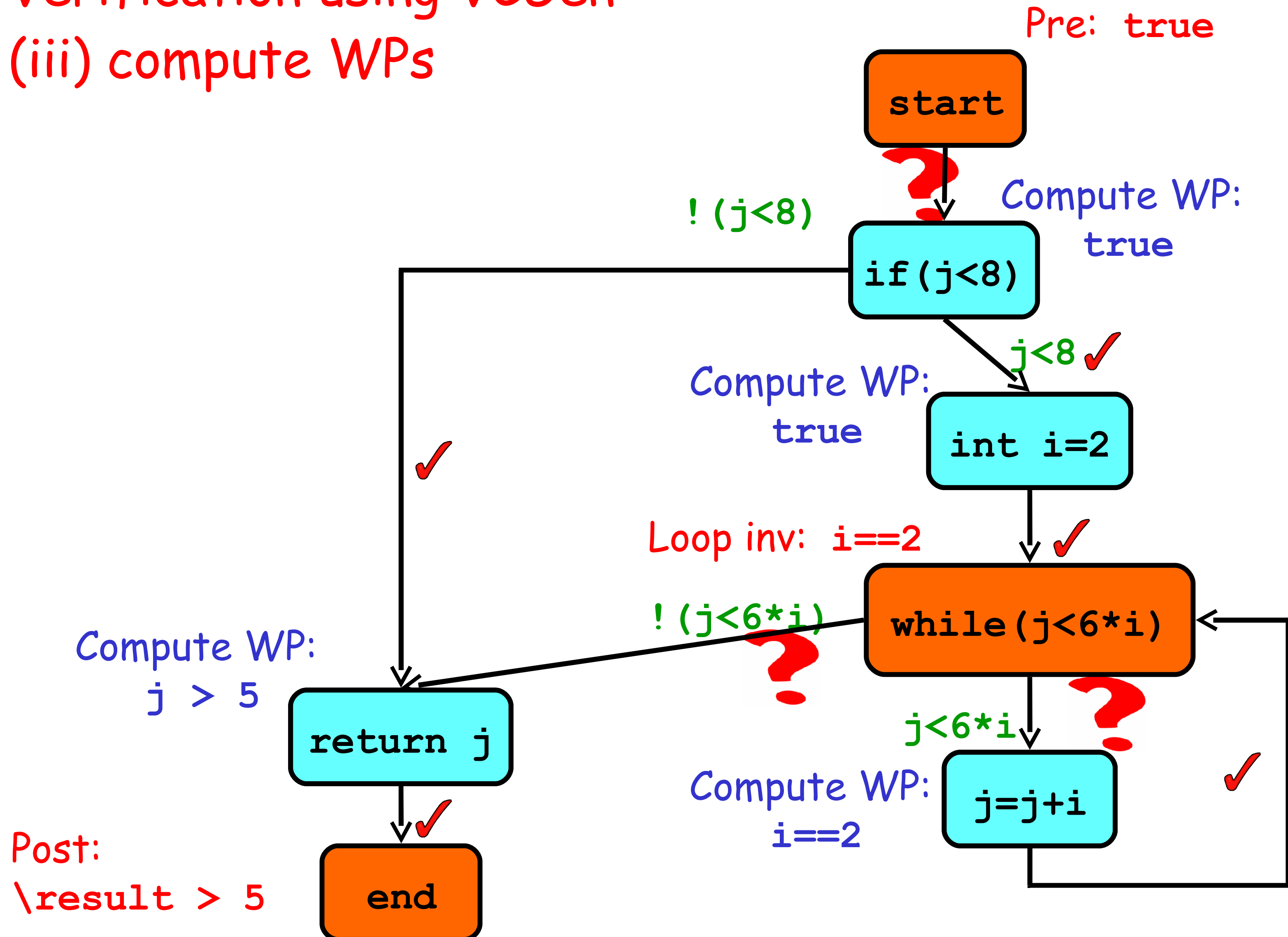
Verification using VCGen
(ii) add assertions

```
//@ ensures \result > 5;
public int example(int j)
{
  if (j < 8) {
    int i = 2;
    /*@ loop_invariant
          i==2;
    @*/
    while (j < 6*i){
      j = j + i;
    }
  }
  return j;
}
```

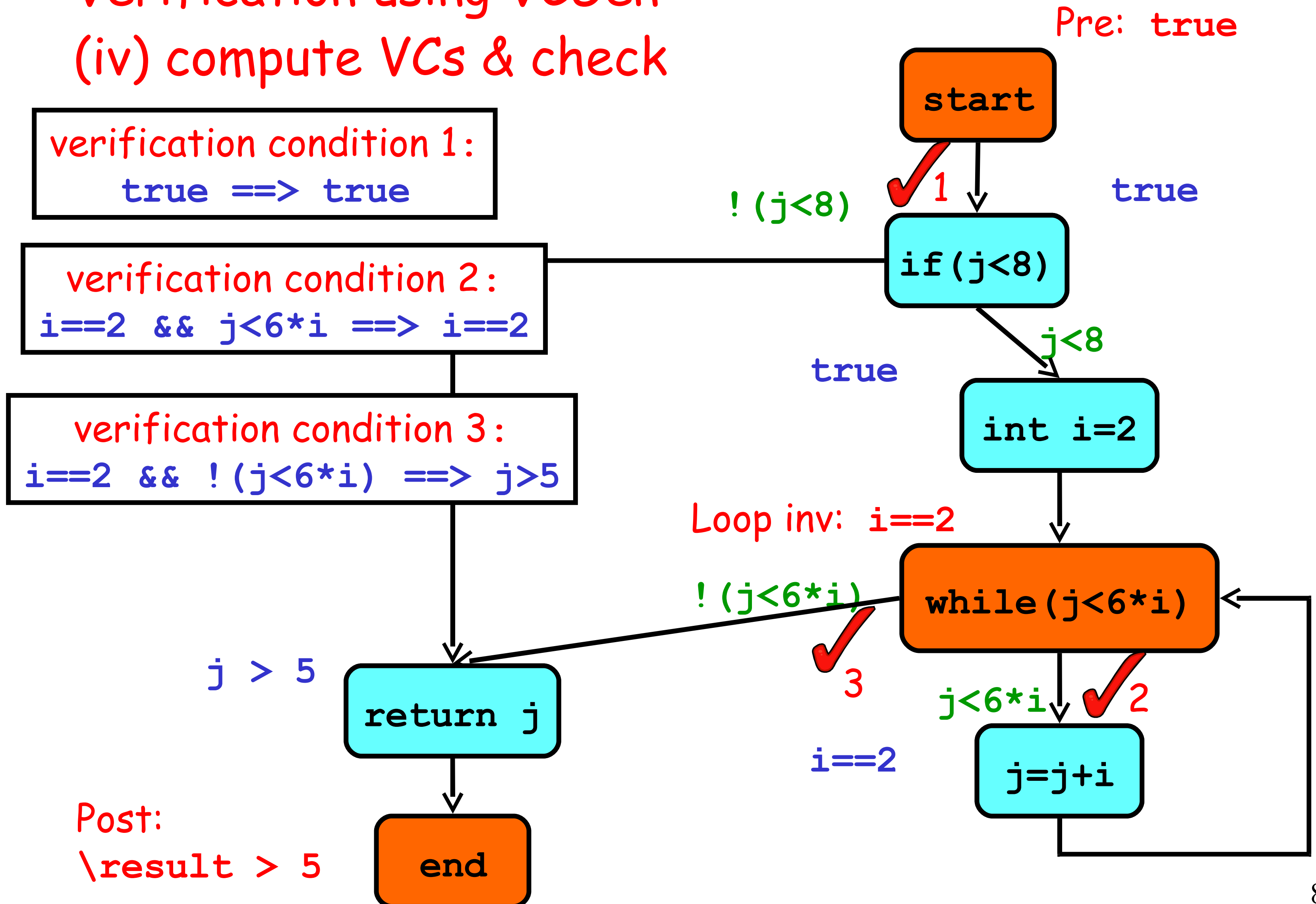
Post: \result > 5



Verification using VCGen
(iii) compute WPs



Verification using VCGen
(iv) compute VCs & check



byte vs source code VC generation

- For byte and source code this works essentially the same

- For bytecode it's a bit messier
  - smaller steps
    - eg `j = j + i;` becomes

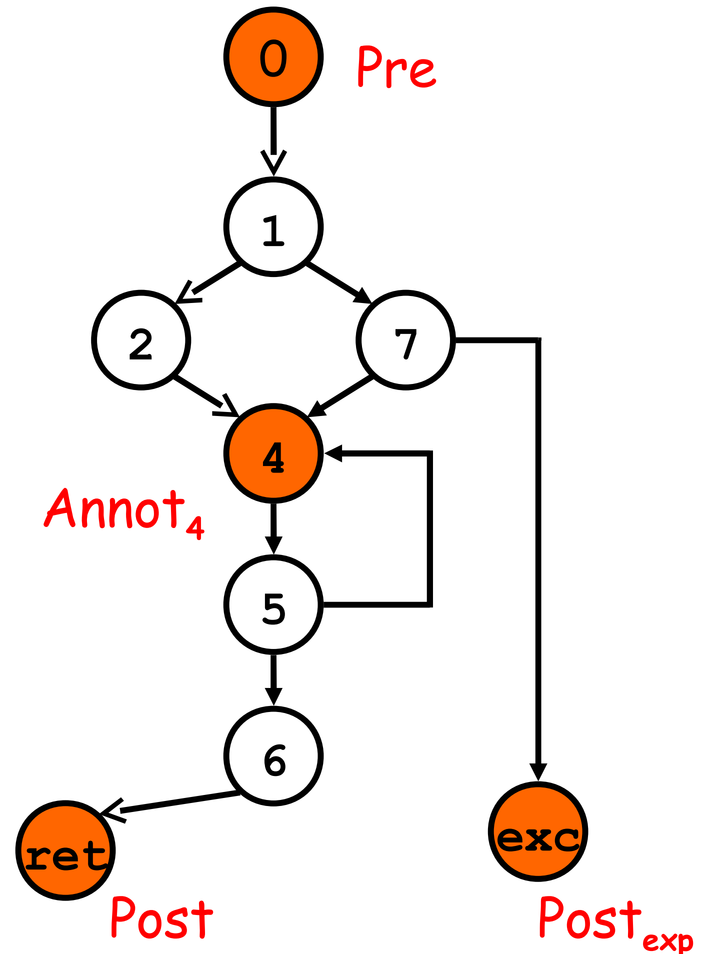
```
push i
push j
add
store j
```

  pushing & popping values on the operand stack,...

  - intermediate assertions will also talk about the operand stack

Verification Condition generation

- code of method induces a control-flow graph
- partially annotated by method spec (Pre, $Post$, $Post_{excp}$) and $Annot_{pc}$
- if we have assertion for at least one node on every cycle, we can compute assertion WP_{pc} for every node



Computing assertions for VC

Assertion WP_{pc} : InitialState x State → Prop computed from assertions of reachable nodes

$$WP_{pc}(s_0,s) = \bigwedge_{(pc,pc') \in Graph} \left(Cond_{pc,pc'}(s) \Rightarrow Transform_{pc,pc'}(P_{pc'}(s_0,s)) \right)$$

where

- $P_{pc'}$ is $Annot_{pc}$ if given or $WP_{pc'}$ otherwise
- $Cond_{pc,pc'}(s)$ is the condition to go from pc to pc'
- $Transform_{pc,pc'}$ is predicate transformer to update assertion according to side effect of bytecode executed

Verification conditions

The verification conditions are then

- $Pre \Rightarrow WP_0$

  the precondition implies the WP computed for the initial state

- $Annot_{pc} \Rightarrow WP_{pc}$

  each intermediate assertion implies the WP computed for that state

Soundness of VCGen in Coq

Define **WP**: **Program × Method × PC → Assertion**
and **VCGen**: **Program × Method → Set(Prop)**

Prove

Suppose all **vc ∈ VCGen(Program,method)** are true.

For all executions of **method** in some initial state s_o with **Pre**(s_o)

- if method terminates normally in state s then **Post**(s_o,s)
- if it terminates exceptionally in state s then $\mathbf{Post_{excp}}$(s_o,s)

using the operational semantics

[Benjamin Grégoire and Jorge Luis Sacchini, Combining a verification condition generator for a bytecode language with static analyses, TGC'2008]

Simplifying VCs

The possibility of exceptions greatly increases the complexity of VCs.

Eg $\quad pc_1 \quad$ `istore x`

$\quad pc_2 \quad$ `getfield f`

$\quad pc_3 \quad$ ...

Then $\quad WP(pc_2) \; = \; lv(x) \neq null \Rightarrow WP(pc_3)$

$\qquad\qquad\qquad\quad \wedge \; lv(x) = null \Rightarrow Post_{exp}$

But if we know x is not null

$$WP(pc_2) \; = \; lv(x) \neq null \Rightarrow WP(pc_3)$$

Safety annotations to reduce VCs

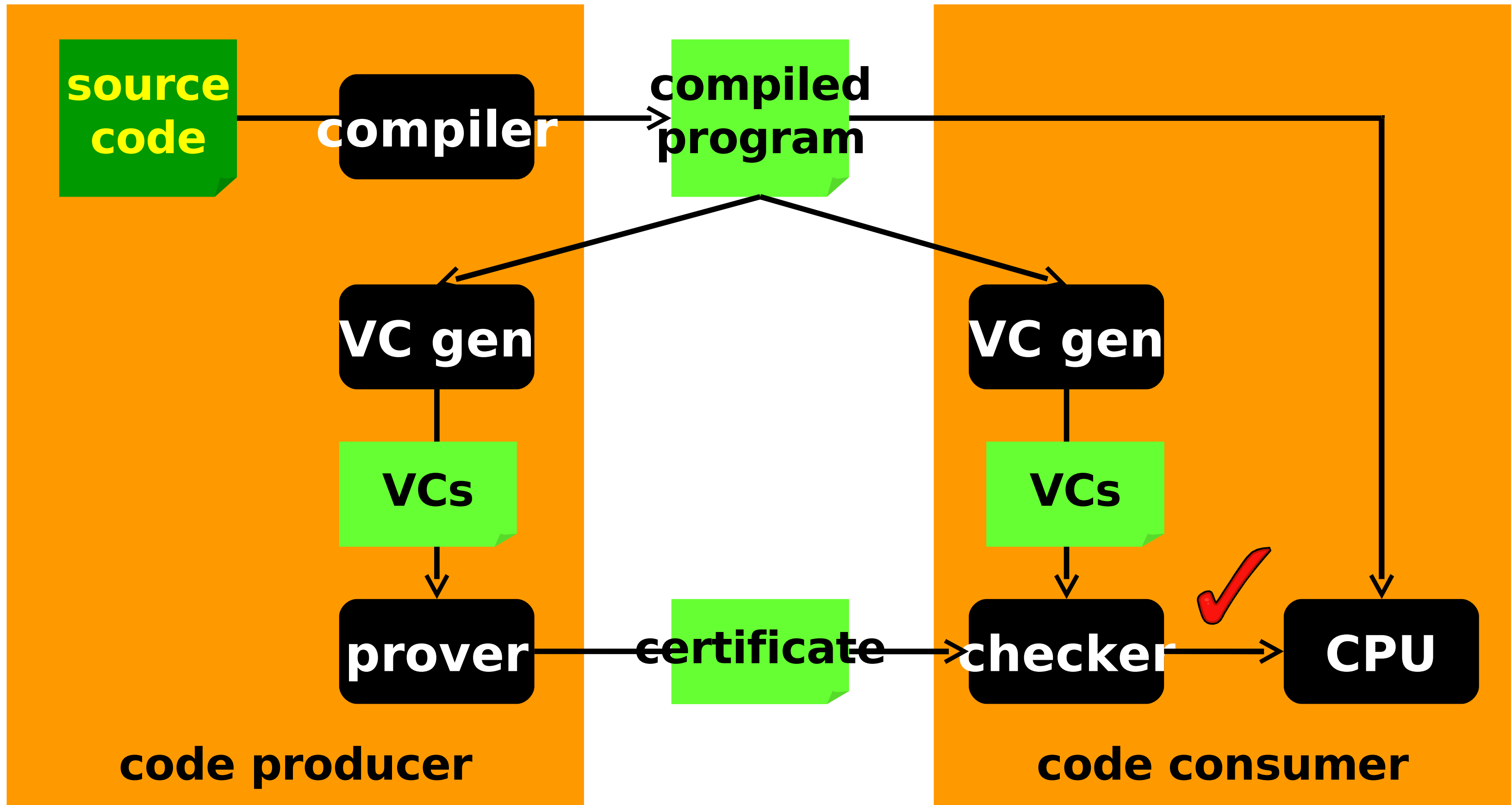
By attaching safety annotations to exclude exceptional executions we can reduce complexity of VCs

eg about non-nullness of references

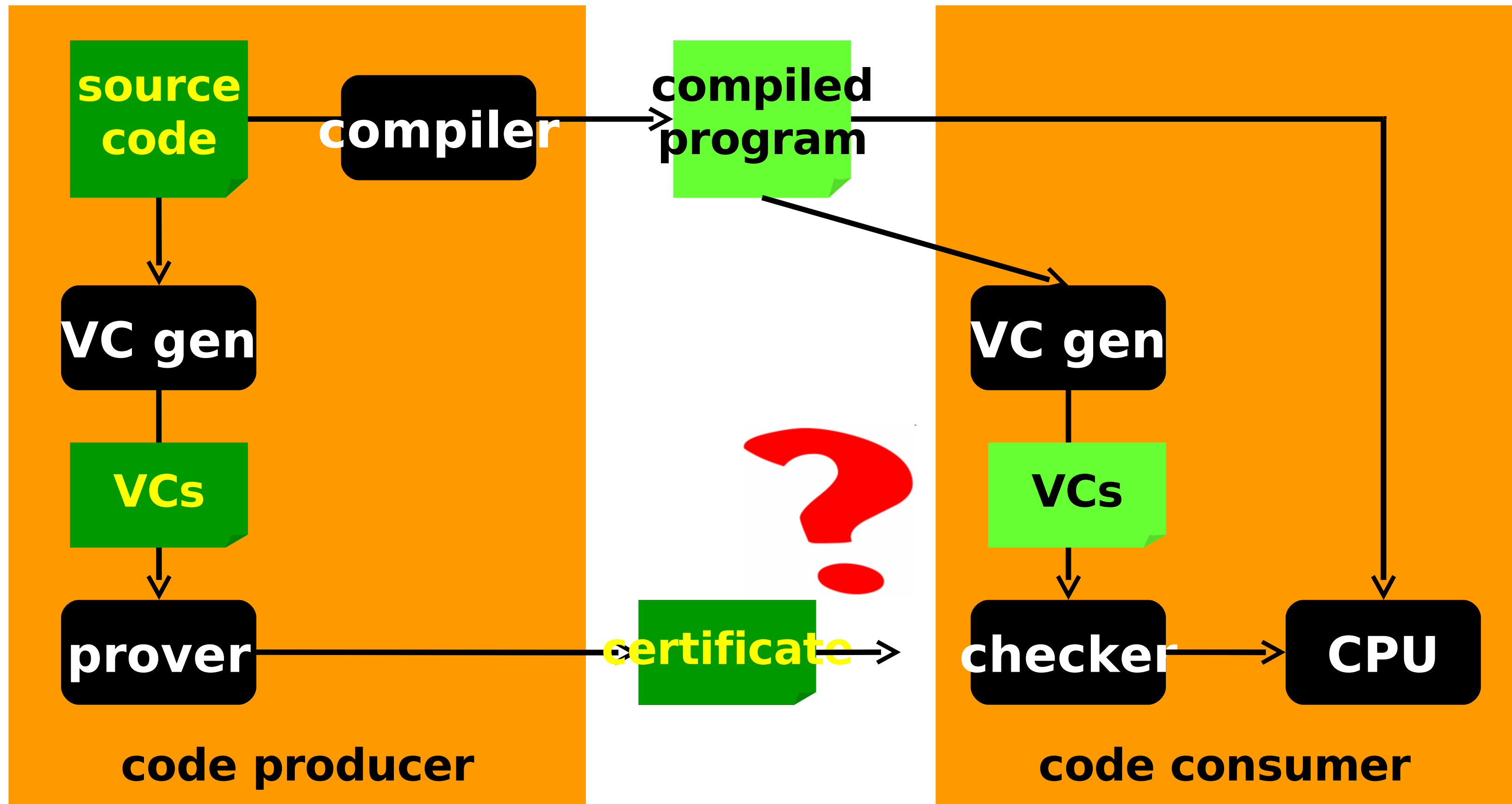
The correctness of these safety annotations can be checked using PCC

eg using a certified non-nullness analysis

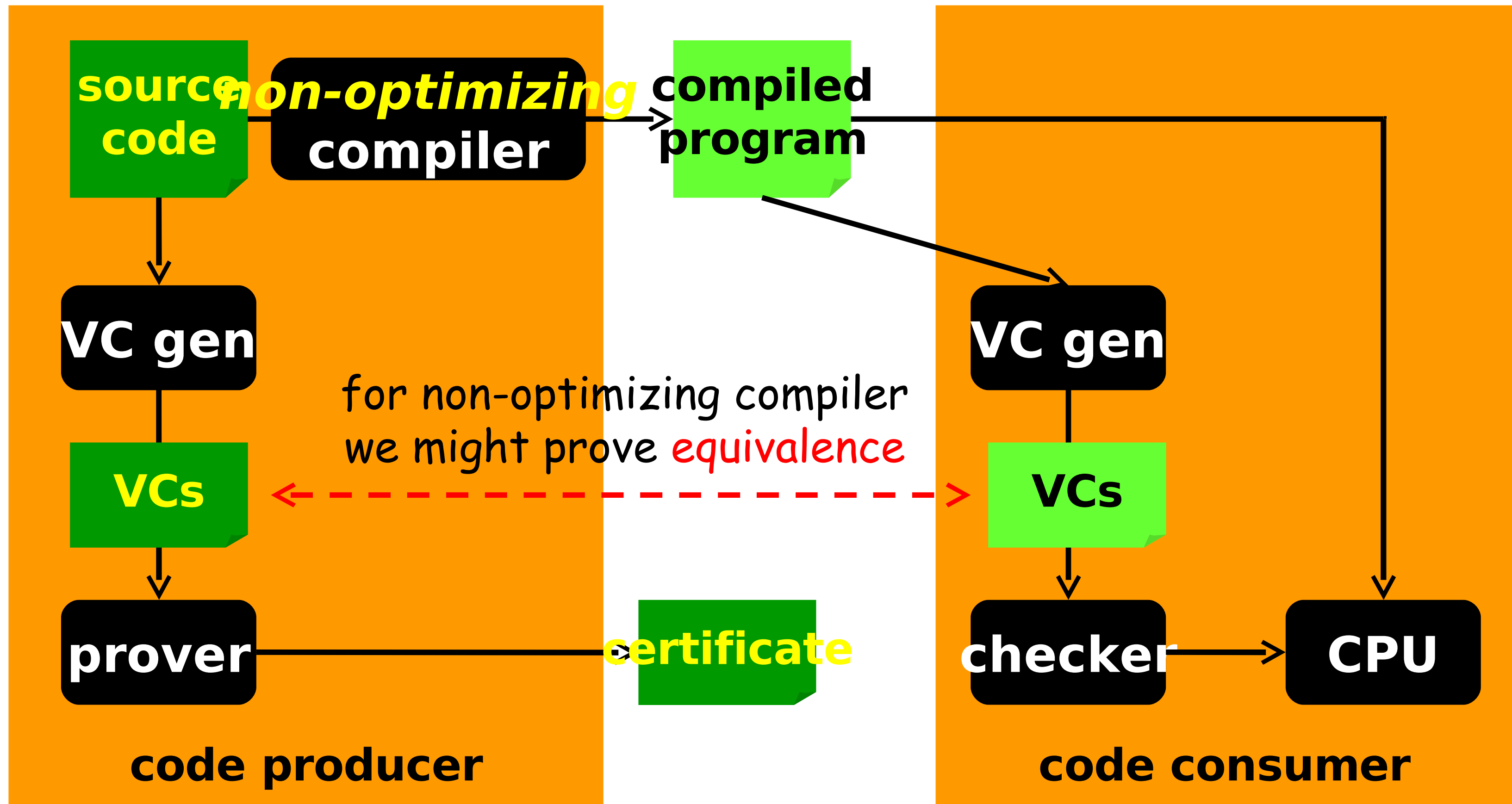
Traditional PCC



Source code verification



(i) prove preservation of proof obligations...

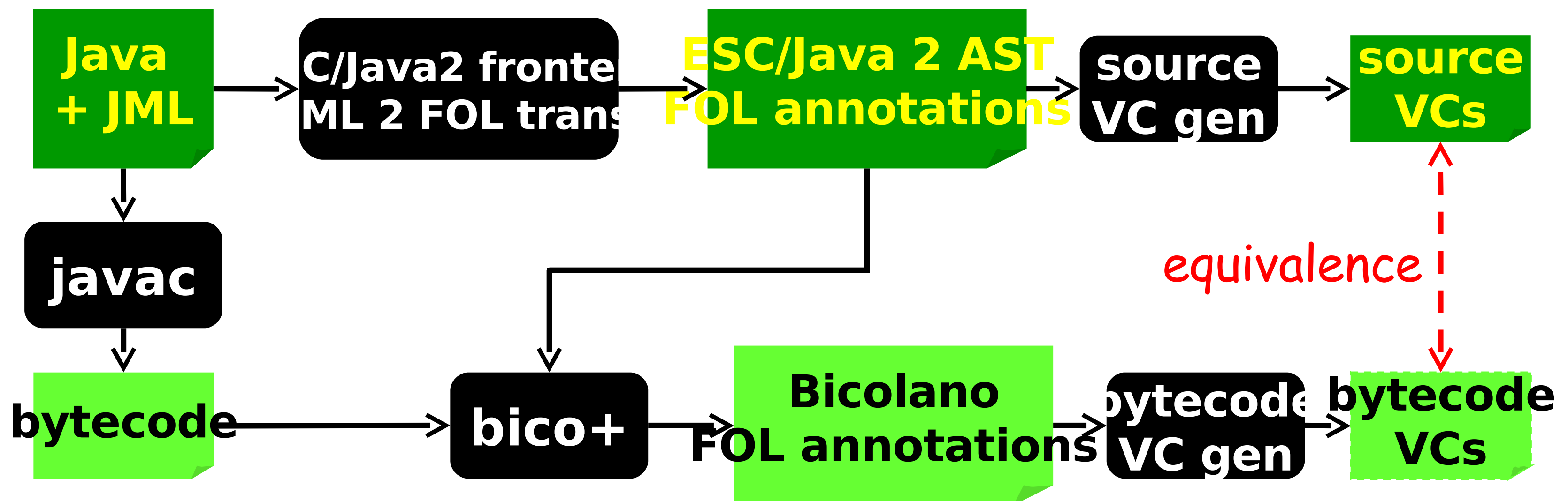


Source vs bytecode VCs

Proof of equivalence by Julien Charles and Hermann Lehner

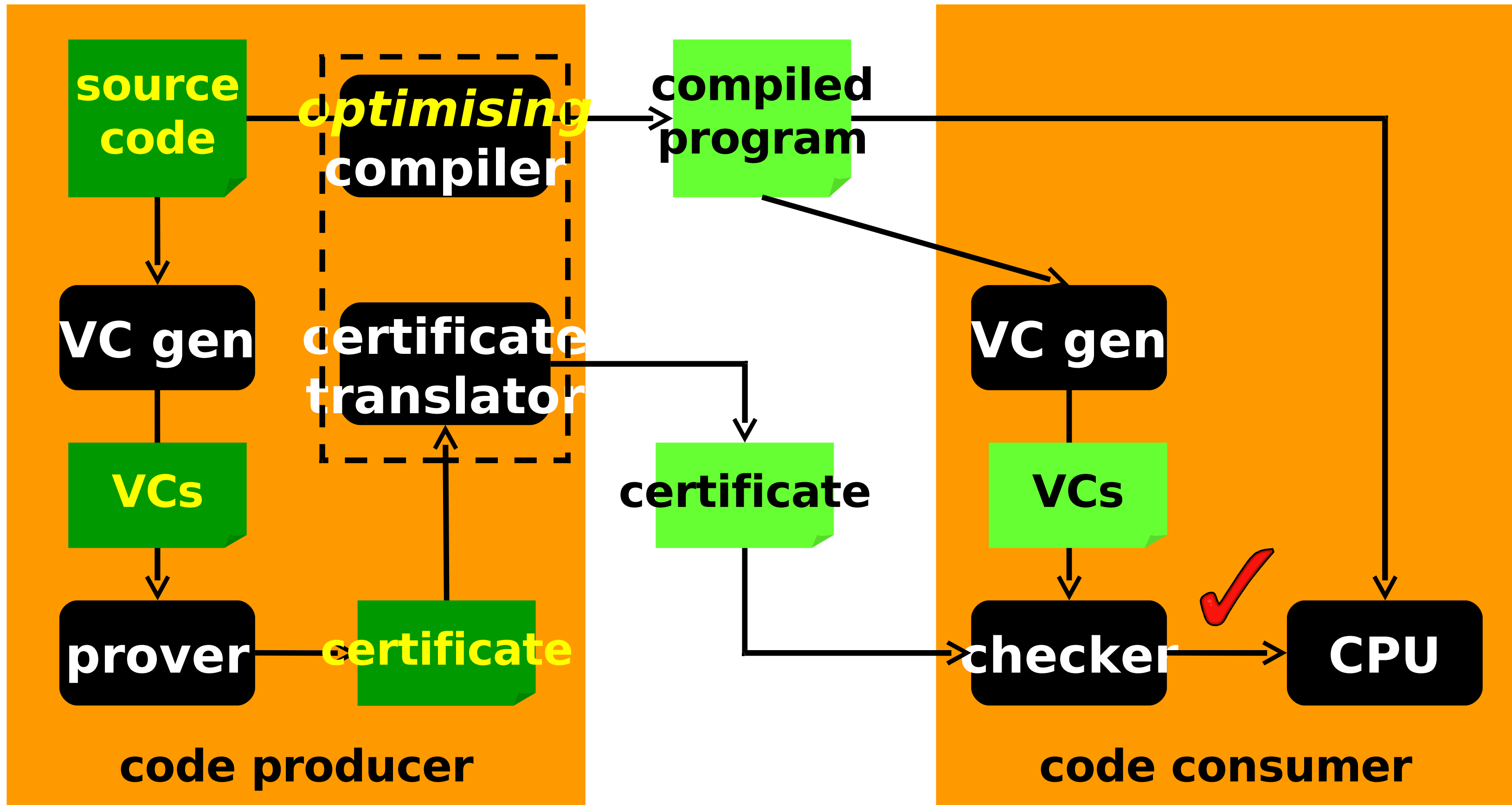
For a given JML-annotated source code program,
VCs generated for bytecode and sourcecode are equivalent

Note this also involves a formalisation of a *source code VCGen* in Coq



(ii) perform certificate translation

[Gilles Barthe and César Kunz, An Introduction to Certificate Translation , FOSAD'2009]



BML (Bytecode Modeling Language)

- Bytecode counterpart of JML

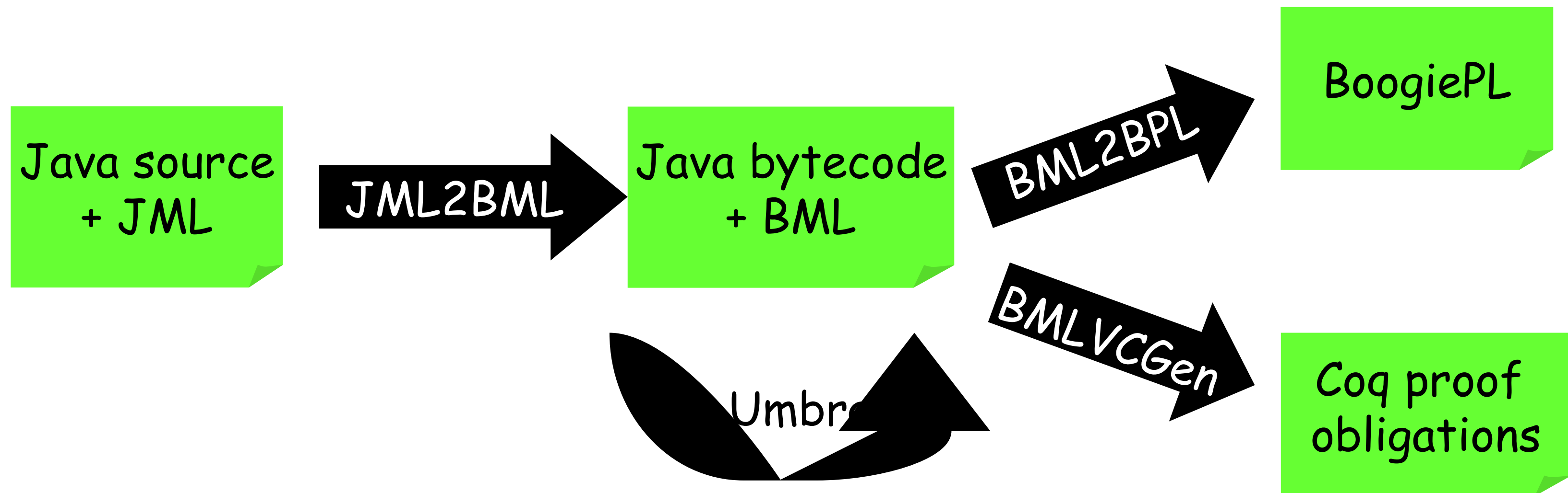
- Central idea:
  Java bytecode can be annotated with BML,
  just like Java sourcecode can be annotated with JML

- Why would we want this?
  - preserve information at bytecode level, for the benefit of bytecode analyses
  - adding computed assertions in .class file
  - in PCC setting, to enable certification of arbitrary properties expressable in BML
    - after all, code consumer only sees the byte code

BML

- Java annotations are *not* preserved in bytecode
  - hence neither are JML annotations
- Java tags (eg @NonNull) *are* preserved at bytecode level
  - but we cannot express JML annotations using Java tags
    - or only very clumsily
- BML defines a format to add annotations in .class files
  - encoding BML annotations using new class attributes

BML tools



- JML2BML compiler
- Umbra editor for Bytecode & BML
  - by Jacek Chrząaszcz, Tomasz Batkiewicz, and Aleksy Schubert (WU)
- BMLVCGen
  - by Benjamin Gregoire (INRIA) and Jorge Sacchini (Univ. Rosario)
- BML2BPL compiler
  - by Hermann Lehner, Ovidio Mallo and Peter Müller (ETH)

Overview

- Context & Proof Carrying Code (PCC)
- The JML specification language for Java
- The Mobius PCC infrastructure for Java
- Applications & case studies

Applications & case studies

Formal methods for real-world Java applications and security ?

Small security-critical applications seem best place to start, eg

- Java Card applications
  - small & simple, and highly security-critical
- Java mobile applications
  - aka J2ME MIDP CLDC
  - larger and more complicated, but commercial interest in checking for security problems (by telcos)

No PCC (yet), just coming up with answer to the question

What would we want to verify anyway ?

is hard enough!

Java Card

Java Card

- dialect of Java for programming smartcards
  - superset of a subset of normal Java
- subset of Java (due to hardware constraints)
  - no threads, doubles, strings, garbage collection
  - a very restricted API
- with some extras (due to hardware peculiarities)
  - communication via byte sequences (APDUs)
  - persistent & transient data in EEPROM & RAM
  - transaction mechanism
- .cap files: compressed .class file format

new JavaCard 3.0 standard adds many standard Java features

Java Card vs Java

Java Card applets are executed in a sandbox,
- just like applets in a web browser

But important differences:
- no bytecode verifier on most cards (due to space required)
- downloading applets controlled by digital signatures instead
  - plus bytecode verification, if card supports it
- sandbox more restrictive, and includes runtime firewall between applets
  - firewall disallows sharing of references between applets

Java Card API

- The Java Card API is very small
  - less than 60 classes, including **`Object`** & all **`Exceptions`**

  and has been fully specified in JML

- But... the API calls for transactions interact with the language semantics, so cannot be fully specified in JML..
  - in fact, buggy implementations of transactions have been shown to break Java type soundness
  - The KeY tool provides semantics of Java Card incl. the behaviour of transactions

Formal methods for Java Card

The good news: we can verify realisitic Java Card applications with modern program verifiers

Eg [Peter Schmitt and Isabel Tonin, Verifying the Mondex Case Study. SEFM'2007]

What we can verify

- for starters, that code never throws runtime exceptions
- interesting invariants
  - eg to rule out integer overflow on 16 bit hardware
- maximum heap size in RAM used
  - max. stack size would be interesting too
- functional properties
  - eg conformance to state-based model of security protocol or access control

Example: verifying RAM heap space usage

In JavaCard, the only objects allocated in RAM are arrays

```
//@ ensures _RAM_used == \old(_RAM_used)+2+size;
public static byte[]
   JCSystem.makeTransientByteArray(short size);

//@ ensures _RAM_used == \old(_RAM_used)+2+2*size;
public static short[]
   JCSystem.makeTransientShortArray(short size);
```

ghost field
_RAM_used
to track RAM
usage (in bytes)

Formal methods for Java Card

The bad news: program verification cannot address all security worries for smartcards, especially not

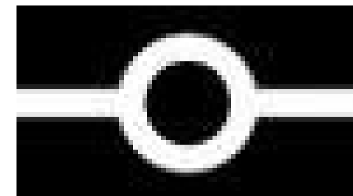
- leaking of sensitive data via power consumption
  - esp. by DPA (Differential Power Analysis) attacks
- the behaviour of code under induced faults
  - eg power glitches or shooting laser at chip surface

Java Card case study: the electronic passport

e-passports

- e-passport contains RFID chip / contactless smartcard
  - in Dutch passports, a Java Card
- chip stores digitally signed information:
  - initially just facial images (photos)
  - soon also fingerprints
    - (EU countries: 21 Sept 2009)
  - later maybe iris

- introduction pushed by US in the wake of 9/11
  - to solve what problem??
- international standard by ICAO (International Civil Aviation Organization, branch of United Nations)



e-passport logo

Security mechanisms

- Passive Authentication (PA)
  - digital signature on passport data on chip
- Basic Authentication Control (BAC)
  - access control to chip, to prevent unauthorised access & eavesdropping
- Secure Messaging (SM)
  - encryption of communications after BAC
- Active Authentication (AA)
  - chip authentication
    - ie prevent cloning
- Extended Access Control (EAC)
  - chip and terminal authentication

ICAO mandatory (PA)

ICAO optional, EU mandatory (BAC, SM)

ICAO optional (AA)

EU only, mandatory for 'advanced' biometrics, ie fingerprint & iris (EAC)

Basic Access Control (BAC)

protects against unauthorised access and eavesdropping

Machine Readable Zone

P<AUSCITIZEN<<JOHN<<<<<<<<<<<<<<<<<<<<<<<<<<<

M0992136<2AUS7906123M1508189<06100178D<<<<02

optically read MRZ

send MRZ

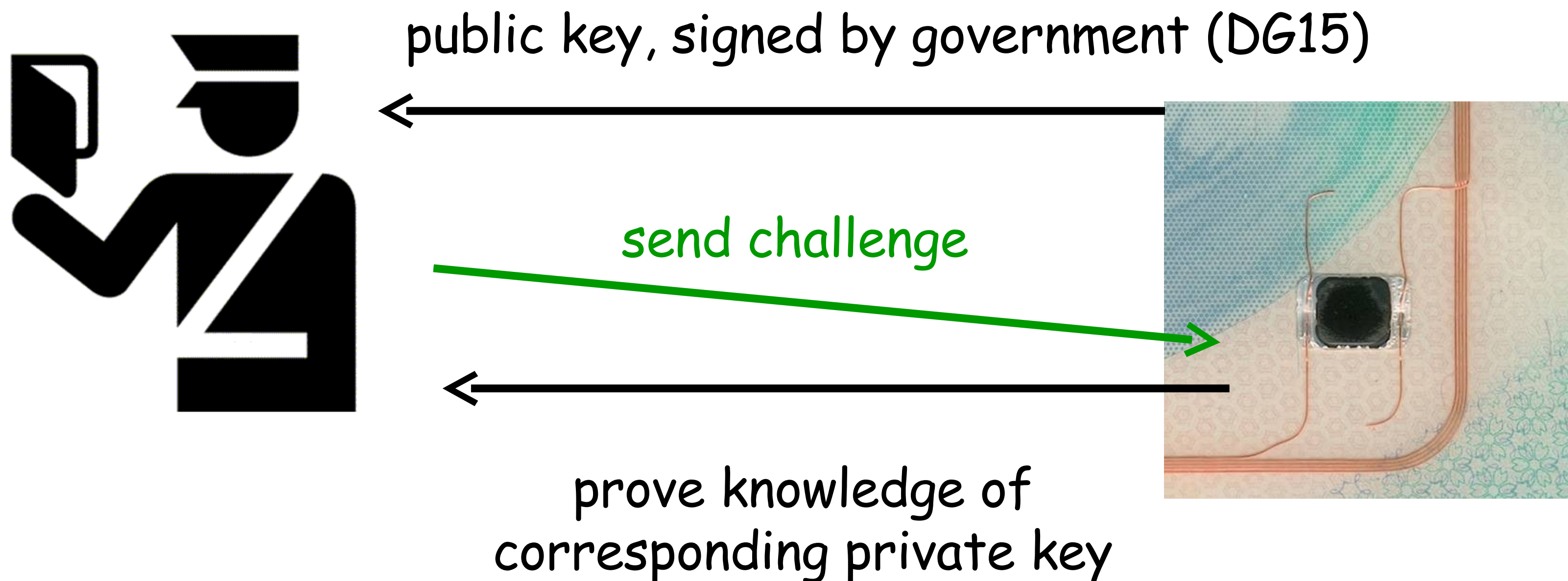
receive additional info

encrypted,
using Secure Messaging

Active Authentication (AA)

protects against passport cloning (which BAC doesn't)

ie authentication of the passport chip



Extended Access Control (EAC)

Two phases

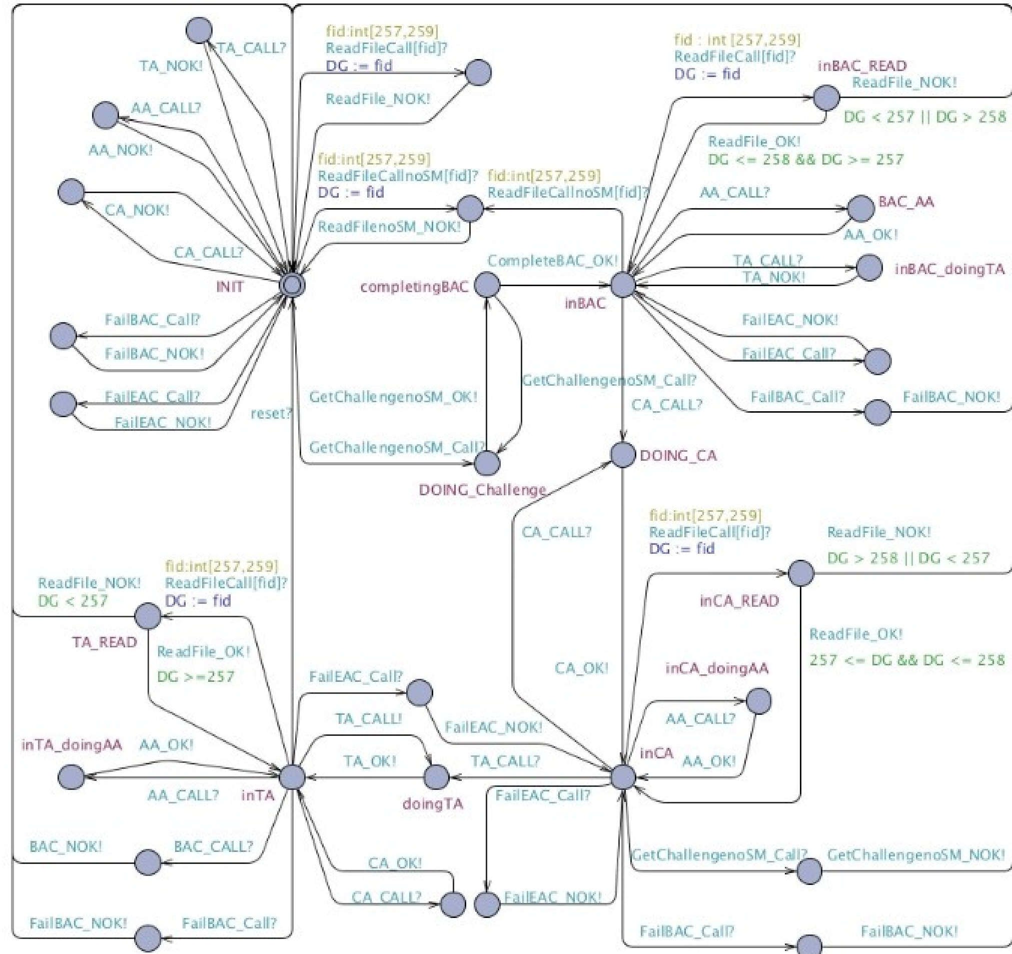
1. Chip Authentication (CA)
   - standard challenge-response
   - replaces AA
   - starts Secure Messaging (SM) with stronger keys

1. Terminal Authentication (TA)
   - uses traditional challenge-response & certificates
     - terminal sends its certificate and associated certificate chain to the chip
     - chip sends challenge
     - terminal replies with signed challenge

Specified by German BSI (*Federal Office for Information Security*)

Formal spec – as state diagram

- distinguishing states
- defining transitions between states
- defining (dis)allowed operations in each state



Using such specs?

- No country will gives us their Java Card passport code to verify...
- Our own open source Java Card implementation of the standard (http://jmrtd.org) contained some bugs...
  - just manual code inspection, not formal verification
- The state diagram spec used for model-based testing
  - automated
  - exhaustive – trying out all operations in all states
  - eg using TorXakis tool,
    - with Haskell representation of the state diagram

[joint work with Wojtek Mostowski, Julien Schmaltz, Jan Tretmans]

MIDP

Java-enabled mobile phones MIDP

- aka J2ME (Java 2 Micro Edition),
  MIDP (Mobile Information Device Profile), with
  CLDC (Connected Limited Device Configuration) API

- special API functionality
  - eg. support for sms:// as well as http://
- fine-grained sandboxing of applications,
  called midlets

J2ME MIDP security model

- sandbox offering fine-grained access control to "dangerous" functionality
  - dangerous = costs money, eg. using network to phone or sms
- code is trusted or not depending on digital signatures
- *trusted* code can use network,
- *untrusted* code is denied network access,
- *semi-trusted* code has to ask user permission
  - via pop-up message
  - permission may have to be asked only once, once per session, or once per sms, depending how trusted the code

mobile phone application security threats?

- malicious midlets making expensive calls, sending expensive sms messages, subscribing to sms services
- SMS spam by rogue midlets
- stealing confidential data: phone book or diary content, location data
  - unwanted information flow
- Denial-of-Service
- X-rated contents, eg via backdoor in game
- ...

Telecom providers want to avoid malicious or buggy midlets that cause problems

  - costs them money and loses them customers!

- current approach to preventing this: static analysis & testing

Some MIDP security bugs

- exploitable bug in BCV
  - found by Adam Gowdiak
- Phenoelit attack midlet on
  - creates race condition
    to let user unwittingly
    authorise SMS
    text message

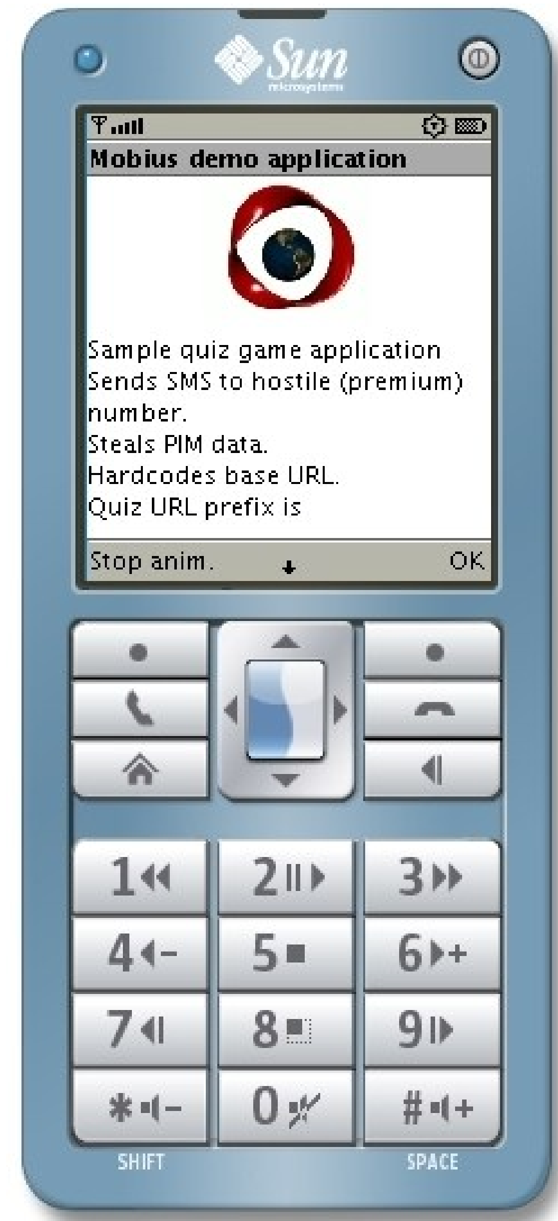
limits of MIDP security model

But even without such bugs in platform

User cannot make security decisions

- user gets confused
- will press ok anyway
- can be tricked ot tempted into making bad decisions
- can't recognize expensive numbers
- can't spot information leaking
- ...

as illustrated by the Mobius game



limits of MIDP security model

- Provider might want to certify compliance with richer security policies, eg
  - midlet will only dial to numbers beginning with 06 or +316
  - midlet will only dial number supplied by user or taken from phone book
  - midlet will not calculate phone number
    - eg `dial((5*x+y)/2);` is very suspicious code
  - midlet will send at most 3 SMS
  - midlet will only send SMS at certain "points"
  - ...

Example policies expressed in JML

- Midlet only opens https-connections

```
//@ requires url.startsWith("https");
Connector.open(String url);
```

- After accessing PIM (Personal Information Management) information, the midlet only uses https

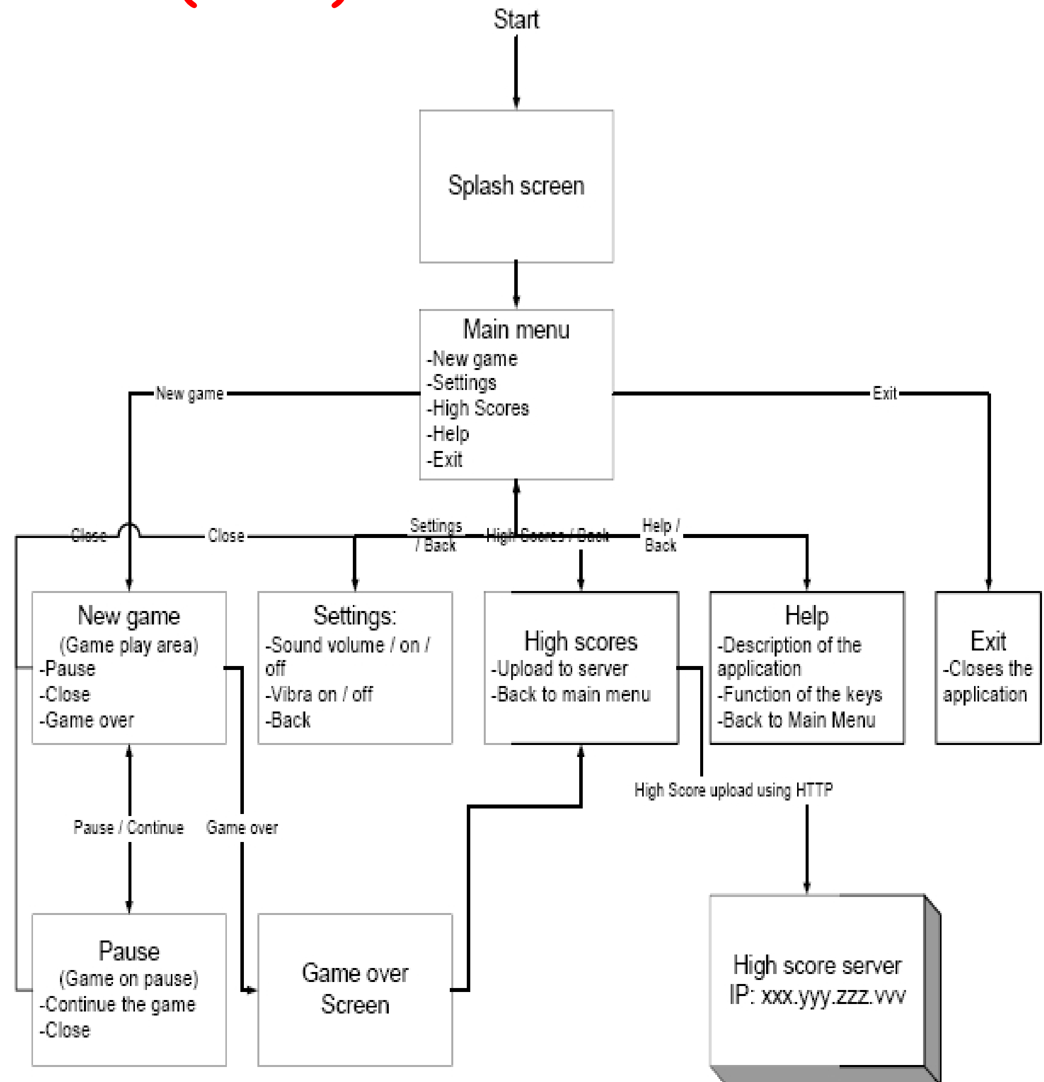
```
/*@ requires _PIM_accessed
              ==> url.startsWith("https"); @*/
Connector.open(String url);
```

boolean ghost field `_PIM_accessed` set by PIM API calls

Unified Testing Criteria (UTC) of JavaVerified.com

current practice for describing behaviour of midlets:
graph showing screens & transitions between them

conformance checked by testing



Verifying conformace to flow graphs

- Notion of flow graph formalised as midlet navigation graph [by Pierre Cregut]
  - describing application flow
  - marking where sensitive API calls are done
  - also prototype tool to extract navigation graph from code
- Midlet navigation graph expressable in JML, and conformance of midlet to graph can then be specified & verified
  - using ghost fields in the API to track state
- Still
  - a lot of work to annotate midlet
  - hard to separate (i) annotations expressing the policy from (ii) additional annotations needed for verification
    - this would be required for PCC

Case study
The MIDP-SSH midlet

joint work with Aleksy Schubert (Warsaw University)

Verification of MIDP-SSH

- MIDP-SSH is an open source SSH client for Java-enabled mobile phones
  - SSH is a protocol similar to SSL

  Provides a secure shell
  - ie. confidentiality & integrity of network traffic

- SSH (v2) is secure, but what about this implementation?

Our analysis proceeded in two stages
1. informal, ad-hoc code inspection
2. formal, systematic verification

[Erik Poll and Aleksy Schubert, Verifying an implementation of SSH, WITS'07]

Motivation

There is a lot of work on verifying security protocols, but to secure the weakest link we should maybe look

- *not* at the cryptographic primitives
- *not* at the security protocol
- but at the software implementing this

1. Flaws found in ad-hoc, manual code inspection

- Weak/no authentication
  no storage of public server keys
  - but fingerprint (hash value) is reported
- Poor use of Java access control (ie. visibility modifiers)

```
public static java.lang.Random rnd = ...;
final static int[] blowfish_sbox = ...;
```

  - Such bugs can be pointed out by automated tools, eg. Findbugs, or prevented by tools, eg. JAMIT tool for automated tightening of acces modifiers
  - Not a real threat (yet) on MIDP, due to current limits on running multiple applications.
- Lack of input validation
  missing checks for terminal control characters

2a. Proving exception freeness

Results:

- Improvements in code needed to avoid some runtime exceptions
  - esp ArrayIndexOutOfBoundsExceptions, that could occur when handling of malformed packets

  Note that

    - such cases are hard to catch using testing, because of huge search space of possible malformed packets
    - in a C(++) application these bugs would be buffer overflow vulnerabilities!
- Also spotted: a missing check of a MAC (Message Authentication Code)
  - process of annotating code *forces* a thorough code inspection

Beyond proving exception freeness: proving functional correctness

- Exception freeness looks at what application should *not* do
  - it should not crash with unexpected runtime exceptions
- How about looking at what it *should* do ?

- This requires some formal specification of the SSH protocol

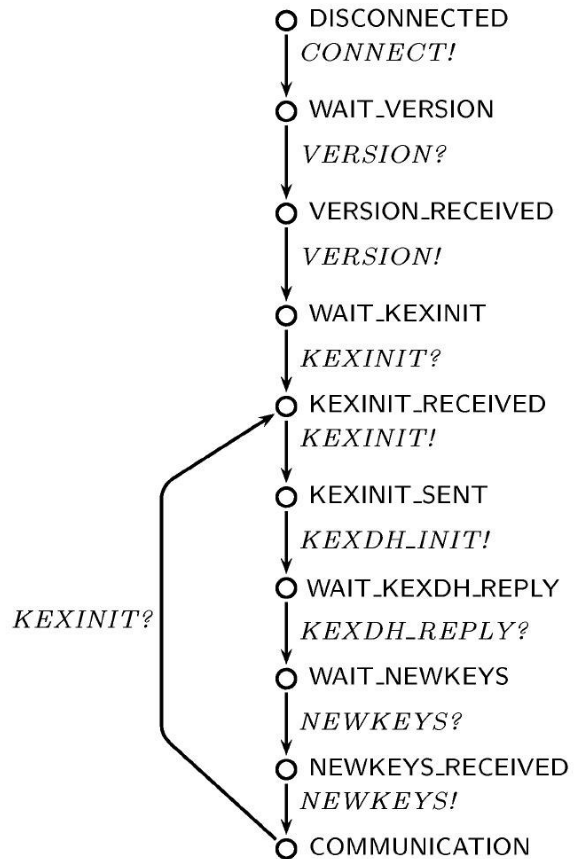
The SSH protocol

- Official specification given in RFCs 4250-4254
  - Over 100 pages of text
  - Many options & variants
    - effectively, SSH is a collection of protocols

- The official specification far removed from typical formal description of security protocols.

- We defined a partial formal specification of SSH as Finite State Machine (FSM) aka automaton
  - SSH client effectively implements a FSM, which has to respond to 20 kinds of messages in right way

The basic SSH protocol as FSM



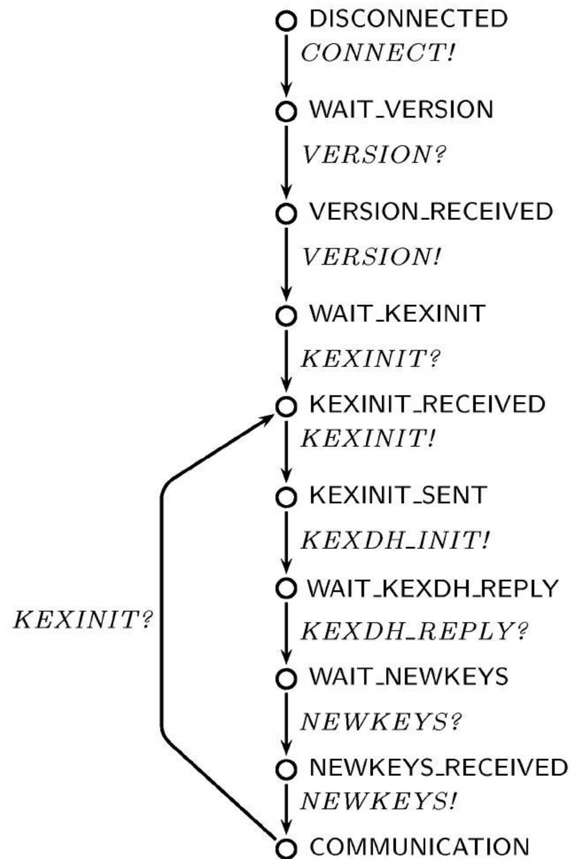
This FSM defines a typical, correct protocol run

SSH as abstract security protocol

- This FSM can also be written in the common notation used for security protocol verification

1. $C \rightarrow S$: CONNECT
2. $S \rightarrow C : V_C$ // VERSION of the server
3. $C \rightarrow S : V_S$ // VERSION of the client
4. $S \rightarrow C : I_S$ // KEXINIT
5. $C \rightarrow S : I_C$ // KEXINIT
6. $C \rightarrow S$: exp(g,X) // KEXDH INIT
7. $S \rightarrow C : K_S$.exp(g, Y).$\{H\}_{inv(K_S)}$ // KEXDH REPLY
8. ...

The basic SSH protocol as FSM



However, this FSM defines

- only one correct protocol run
- no *incorrect* protocol runs

How do we specify:

i. optional features in the RFCs, which allow various correct protocol runs?
ii. how *incorrect* protocol runs should be handled?

Specifying SSH protocol - choices

DISCONNECTED
CONNECT!
WAIT_VERSION
VERSION?
VERSION!
VERSION_RECEIVED
VERSION_SENT
VERSION!
VERSION?
WAIT_KEXINIT
KEXINIT?
KEXINIT!
KEXINIT_RECEIVED
KEXINIT_SENT_NOTR
KEXINIT?
KEXINIT!
KEXDH_INIT!
KEXDH_KEXINIT_SENT_NOTR
KEXINIT_SENT
KEXINIT?
KEXDH_INIT!
KEXINIT_KEXDH_INIT_SENT
WAIT_KEXDH_REPLY
KEXDH_REPLY?
KEXDH_REPLY?
WAIT_NEWKEYS
NEWKEYS?
NEWKEYS!
KEXINIT!
KEXINIT?
NEWKEYS_RECEIVED
NEWKEYS_SENT
NEWKEYS!
NEWKEYS?
COMMUNICATION

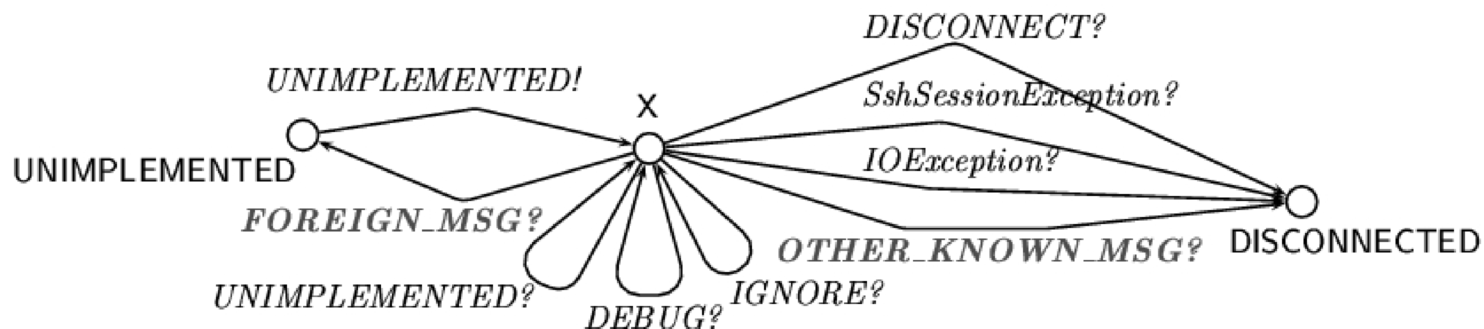
with possible choices spelled out

Specifying SSH protocol - errors

To handle incorrect runs, there are, in every state X, additional messages that

- should be ignored, *or*
- should be ignored after a reply "UNIMPLEMENTED", *or*
- should lead to disconnection.

In every state X, we have to add an 'aspect' of the form below



Specifying SSH protocol as FSM

- Obtaining these FSM from the informal specification of SSH given in the RFCs is hard:
  - notion of state is completely implicit in the RFCs
  - constraints of correct sequences of messages given in many places
    - Eg constraints such as "once a party sends a SSH_MSG_KEXINIT message [. . .], until it sends a SSH_MSG_NEWKEYS message, it MUST NOT send any messages other than [. . .]"
  - not clear if underspecification is always deliberate
    - eg order of VERSION messages from client to server and vv.

- Anyone implementing SSH will effectively have to extract the same information from the RFCs as is given by our FSM

2b. Verifying the code against FSM

- AutoJML tool used to produce JML annotations from FSM
  - tool extended to cope with multiple of diagrams

- Obvious security flaw:

  implementation doesn't record the state correctly (at all!)
  - Hence, an attacker can ask for username/password before session key has been established

- Improved code was successfully verified against the FSM

Effort

- Formal specification & verification of the protocol implementation (4.5 kloc) took around 6 weeks
  - ie. proving
    a) exception freeness, and
    b) adherence to our formal specification given by FSM

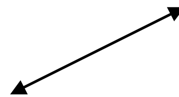
    a) catches errors in handling malformed messages
    b) catches errors in handling incorrect/unusual sequences of messages

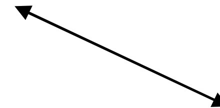
  - incl. 2 weeks understanding & formalising SSH specs

Central problem: how to relate

typical abstract security protols:
tens of lines

?





official spec of SSH:
>100 pages of RFCs

code:
4.5 kloc of Java

Conclusions

- The official specification of SSH can be improved.
  In particular, including an explicit notion of state would help (and make security flaws as found in MIDP-SSH much less likely)

- Our verification can catch errors in handling
  - incorrectly formatted messages, and
  - incorrect sequences of messages

- But, our verification is still *incomplete*
  - as we only use a a *partial* formal specification of SSH

Wrap-up

Some parting thoughts...

- We can completely formalise realistic sequential programming languages, like sequential Java, to provide a PCC infrastructure
- Getting workable, modular specification & verification techniques for Java is still a challenge
  - JML still under construction
  - still a gap between full-blown JML and Mobius infrastructure
  - not to mention concurrency ...
- Getting workable formal definitions of security properties of interest is also still a challenge:
  - What do we want to verify – and maybe certify – anyway?