Program Verification

Erik Poll
Digital Security
Radboud University Nijmegen
Overview

- Program Verification using Verification Condition Generators
- JML – a formal specification language for Java
  - used for the 2nd & last individual exercise
- Verifying the verifiers
- PCC – Proof Carrying Code
  - (later lecture)
Program verification

• program verification = **mathematically proving that a program satisfies some property**
  – for all possible executions: ie all possible inputs, all possible scheduling of parallel threads,....

• in industry, **testing** is often referred to as **verification**, but testing provides only **weaker** guarantees
  – Why?
    • because testing will only try some executions
  – except in rare case where you can do **exhaustive testing**
Program Verification
using
Verification Condition Generation
Program Verification using VCGen

One of the standard approaches for program verification:

using **Verification Condition Generator (VCGen)**:

1. Program is *annotated with properties* (the *specification*)
2. Verification Condition Generator produces a set of *logical properties*, the so-called *verification conditions*
3. If these verification conditions are true, the annotations are correct – ie the program satisfies the specification
Example verification using VCGen

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

These annotations give a pre- and postcondition that form the specification:

on any input, this method will return a result greater than 5

• is this specification always met?
• how do you know this?
• could an automated tool reproduce your reasoning?
Verification using VCGen
(i) program as graph

//@ ensures \result > 5;
public int example(int j) !(j<8) {
    if (j < 8) {
        int i = 2;
        while (j < 6*i){
            j = j + i;
        }
    }
    return j;
}
Verification using VCGen
(ii) add assertions

```java
//@ ensures \result > 5;
public int example(int j) { }(j<8)
{
    if (j < 8) {
        int i = 2;
        //@ loop_invariant
        i==2;
        @*/
        while (j < 6*i){
            j = j + i;
        }
    }
    return j;
    Post: \result > 5
}
End
```
Verification using VCGen
(iii) compute VCs & check

```
int i = 2

while (j < 6 * i)
    if (j < 8)
        verification condition:
        i == 2 && ! (j < 6 * i) ===> j > 5
        verification condition:
        i == 2 && j < 6 * i ===> i == 2
        verification condition:
        true ===> true
    endif
    j = j + i
endwhile

return j
end
```

Pre: true

Compute WP:
true

Post:
\( \text{result} > 5 \)
Verification condition generation

Given a postcondition and loop invariants

• compute a assertion $P_s$ for every state $s$
  based on assertions $P_{s'}$ of the states $s'$ reachable from $s$
  – key idea: $P_s$ is the weakest predicate such that if it hold in state $s$, and the program goes to state $s'$ then $P_{s'}$ will hold in that state $s'$

• all that remains to be verified
  – $\text{Pre} \Rightarrow P_0$
    the precondition specified in the program implies the assertion computed for the initial state
  – $\text{Loop}_s \Rightarrow P_s$
    each loop assertion specified in the program implies the assertion computed for that state
“Opposite” approach: forward instead of backwards

Instead of working backwards from the postcondition of the final state, you can work forward from the precondition in the initial state: you then compute strongest postconditions instead of weakest preconditions

This is very similar to symbolic execution of a program.
Tricky issues in program verification

Whatever the approach, the bottlenecks in program verification remain…

1. pointers / references & the heap
   Reasoning about data on the heap is difficult. Even in a language with automatic memory management, such as Java or C#, we still have the complication of aliasing

2. concurrency aka multi-threading
State-of-art in program verification for security

• Verification of Microsoft Hyper-V Hypervisor using the VCC program verifier for C [2009]
  the motivation to verify this is… security!
• Info on VCC
• Video presentation on VCC
  http://channel9.msdn.com/posts/Peli/Michal-Moskal-and-The-Verified-C-Compiler/

• Verification of seL4 microkernel in L4.verified project at NICTA
Need for specifications

A prerequisite for any program verification:
we need

1. meaningful specifications to verify

2. some convenient notation to write down such specifications, ie. a specification language
Overview

• Program Verification using Verification Condition Generators
• JML – a specification language for Java
• Verifying the verifiers
• PCC – Proof Carrying Code
JML
Formal specification for Java
JML

- formal specification language for Java
  - Properties can be specified in Design-By-Contract style, using pre/postconditions and invariants

- Various tools to check JML specification by eg
  - runtime checking
  - program verification
to make JML easy to use

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

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

- Properties specified using Java syntax, extended with some operators
  \( \text{old}( ), \text{result}, \text{forall}, \text{exists}, \Rightarrow, .. \)
  and some keywords
  \text{requires, ensures, invariant, ....}
Example JML

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

    //@ 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;
    }
}
```
JML basics

• preconditions requires
• postconditions ensures
• exceptional postconditions signals
• (object) invariants invariant
  - must be established by constructors
  - must be preserved by methods
    • ie. assuming invariant holds in pre-state, it must hold in the post-state
Exceptional postconditions: signals

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

But you can ignore this for the practical exercise! There we will always prove that no exceptions can be thrown.

JML convention: a method may only throw exceptions that are explicitly listed in the throws clause. (Java allows implicit Runtime- exceptions, eg Nullpointer- and ArrayIndexOutOfBound; JML does not!)
non_null

- Lots of invariants and preconditions are about reference not being null, eg
  
  ```java
  int[] a; //@ invariant a != null;
  ```

- Therefore there is a shorthand

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

- But, as most references are non-null, some JML tools adopt this as default, so that only nullable fields, arguments and return types need to be annotated, eg

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

- JML may move to adopting JSR308 Java tags for this

  ```java
  @Nullable int[] b;
  ```
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;
  ```
  but the former may allow tools to give more precise feedback, namely on whether P or Q is not satisfied
What can you do with this?

• documentation/specification
  - explicitly record detailed design decisions & document assumptions (and hence obligations!)
  - precise, unambiguous documentation
    • parsed & type checked
• use tools for
  - runtime assertion checking
    • eg when testing code
  - compile time program analysis
    • up to full formal program verification
assert and loop_invariant

*Inside* method bodies, JML allows

- **assertions**

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

- **loop invariants**

  ```java
  //@ loop_invariant 0<= n && n < a.length &
  //          (\forall int i; 0<= i & i < n;
  //    a[i] != null );
  //@
  ```
- **Program verification tools**, such as ESC/Java2, KeY, Krakatoa, ... can do program verification of JML-annotated Java code
  - there is a limit to what fully automated tools, such as ESC/Java2, can verify
    - eg. they won't be able to prove Fermat's Last theorem

- So far, only really feasible for small programs
  - incl. realistic Java Card smart card applications

- In addition to doing the verification, which is a lot of work, a bottleneck is expressing the security property you want to verify
JML for security

JML can be used to specify for instance

1. **which – if any - exceptions can be thrown**
   incorrectly/not handling errors common source of security problems

2. **security-critical invariants to be preserved**
   even when exceptions occur

3. **assumptions on input the application relies on**

4. **any property expressible by security automaton**

Simply trying to verify that a program throws no exceptions – or just no Nullpointer-exceptions - will expose many (implicit) invariants and assumptions on input
Related work

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

- **SparkAda for Ada**
  by Praxis High Integrity System
  Commercially used, esp. for safety-critical software (eg in avionics) rather than security-critical software
Overview

• Program Verification using Verification Condition Generators
• JML – a specification language for Java
  – used for the 2nd & last individual exercise
• Verifying the verifiers
• PCC – Proof Carrying Code
Verifying the verifiers
Verifying the verifiers?

• (How) can we know that a verification condition generator – or any other verification tool – is correct?

• More generally, how can we know that any program analysis tool is correct?

Eg
  – a type checker
  – program optimisations performed by a compiler
  – a whole compiler
Basis for such verification?

- How do we define what it means for a program verification tool to be correct?
- With respect to what can we define correctness?

- Ultimately, the basis for this is a formal semantics of the programming language,
  - i.e. some model that describes the meaning of programs, by giving the meaning of all programming language constructs

- The broader research field of formal methods studies techniques to produce such models and verify properties of them – using tools (e.g., theorem provers)
Example approach in Mobius project

- certified program verification (and PCC) for sequential Java
- basis for everything
  - formal operational semantics in Coq: Bicolano
- certified executable checkers
  - for specific safety properties
    - eg. an information flow analysis
- certified Verification Condition Generator (VCGen)
  - for more general properties expressible in JML
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: completed 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 semantics 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)
Verifying a type checker

Operational semantics can be defined in two styles:

1. defensive
   - VM state includes all type information, and execution performs all type checks, at run time

2. normal
   - VM trusts the code to be well-typed, and only does the minimum runtime checks requires for non-nullness, array bounds 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
Verifying a compiler

By defining two formal operational semantics
1. for source code
2. for byte code (or machine code)

one can formally verify the correctness of a compiler \( C \), by proving the following diagram commutes:

\[
\begin{array}{ccc}
S_0 & \text{source code program } & M & \Rightarrow & S_n \\
\downarrow & & & & \downarrow \\
S_0 & \text{compiled program } & C(M) & \Rightarrow & S_n \\
\end{array}
\]

where \( \sim \) relates source to matching byte states.
State of the art in compiler verification

- The ongoing CompCert project has produced a fully verified compiler for a large subset of C – down to machine code.
Overview

- Program Verification using Verification Condition Generators
- JML – a specification language for Java
  - used for the 2\textsuperscript{nd} & last individual exercise
- Verifying the verifiers
- PCC – Proof Carrying Code
  - a specific way of using the machinery of formal program verification for security; if time allows there’ll be a lecture on this