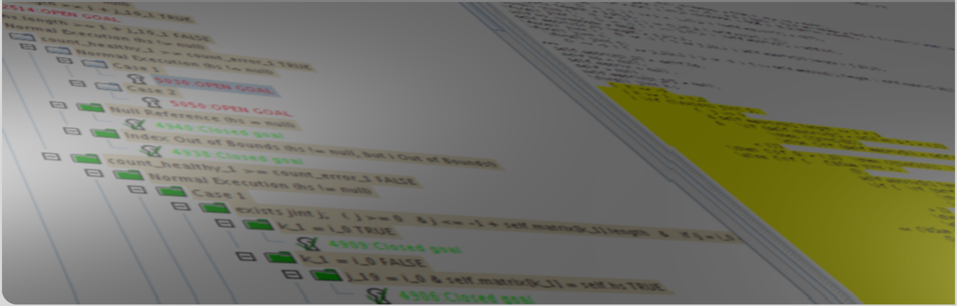
Karlsruhe Institute of Technology

Deductive Verification of Information Flow Properties of Java Programs

Christoph Scheben | July 13, 2011

INSTITUTE FOR THEORETICAL COMPUTER SCIENCE



KIT – University of the State of Baden-Wuerttemberg and
National Laboratory of the Helmholtz Association

www.kit.edu

Aim

Static verification of explicit and implicit flows in Java programs:

1. Program-level specification of information flow properties
    - considered programming language: Java
    - considered specification language: JML
2. Deductive verification of such properties without approximation of information flow dependencies
    - verification system: KeY
    - low level specification: JavaDL (Java Dynamic Logic)

Non-Interference

Prominent information flow property: non-interference

Simple case:

- program *P*
- partion of the program variables of *P* in
  - low security variables *low* and
  - high security variables *high*

Definition (Non-interference – Version 1)

For program *P* the high variables *high* do not interfere with the low variables *low*

$$\Leftrightarrow$$

when starting *P* with arbitrary values for *low*, then the values of *low* after executing *P*, are independent of the choices of *high*.

Non-Interference

Prominent information flow property: non-interference

Simple case:

- program *P*
- partion of the program variables of *P* in
  - low security variables *low* and
  - high security variables *high*

Definition (Non-interference – Version 1)

For program *P* the high variables *high* do not interfere with the low variables *low*

$$\Leftrightarrow$$

when starting *P* with arbitrary values for *low*, then the values of *low* after executing *P*, are independent of the choices of *high*.

Non-Interference

Prominent information flow property: non-interference

Simple case:

- program *P*
- partion of the program variables of *P* in
  - low security variables *low* and
  - high security variables *high*

Definition (Non-interference – Version 1)

For program *P* the high variables *high* do not interfere with the low variables *low*

$$\Leftrightarrow$$

when starting *P* with arbitrary values for *low*, then the values of *low* after executing *P*, are independent of the choices of *high*.

Non-Interference

Prominent information flow property: non-interference

Simple case:

- program *P*
- partion of the program variables of *P* in
  - low security variables *low* and
  - high security variables *high*

Definition (Non-interference – Version 1)

For program *P* the high variables *high* do not interfere with the low variables *low*

$$\Leftrightarrow$$

when starting *P* with arbitrary values for *low*, then the values of *low* after executing *P*, are independent of the choices of *high*.

Non-Interference

Prominent information flow property: non-interference

Simple case:

- program *P*
- partion of the program variables of *P* in
  - low security variables *low* and
  - high security variables *high*

Definition (Non-interference – Version 1)

For program *P* the high variables *high* do not interfere with the low variables *low*

$$\Leftrightarrow$$

when starting *P* with arbitrary values for *low*, then the values of *low* after executing *P*, are independent of the choices of *high*.

Non-Interference

Prominent information flow property: non-interference

Simple case:

- program *P*
- partion of the program variables of *P* in
  - low security variables *low* and
  - high security variables *high*

Definition (Non-interference – Version 1)

For program *P* the high variables *high* do not interfere with the low variables *low*

$$\Leftrightarrow$$

when starting *P* with arbitrary values for *low*, then the values of *low* after executing *P*, are independent of the choices of *high*.

Non-Interference

Prominent information flow property: non-interference

Simple case:

- program *P*
- partion of the program variables of *P* in
  - low security variables *low* and
  - high security variables *high*

Definition (Non-interference – Version 2)

For program *P* the high variables *high* do not interfere with the low variables *low*

$$\Leftrightarrow$$

running two instances of *P*, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.

Non-Interference

Prominent information flow property: non-interference

Simple case:

- program *P*
- partion of the program variables of *P* in
  - low security variables *low* and
  - high security variables *high*

Definition (Non-interference – Version 2)

For program *P* the high variables *high* do not interfere with the low variables *low*

$$\Leftrightarrow$$

running two instances of *P*, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.

Non-Interference

Prominent information flow property: non-interference

Simple case:

- program *P*
- partion of the program variables of *P* in
  - low security variables *low* and
  - high security variables *high*

Definition (Non-interference – Version 2)

For program *P* the high variables *high* do not interfere with the low variables *low*

$$\Leftrightarrow$$

running two instances of *P*, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.

Non-Interference

Prominent information flow property: non-interference

Simple case:

- program *P*
- partion of the program variables of *P* in
  - low security variables *low* and
  - high security variables *high*

Definition (Non-interference – Version 2)

For program *P* the high variables *high* do not interfere with the low variables *low*

$$\Leftrightarrow$$

running two instances of *P*, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.

Non-Interference

Prominent information flow property: non-interference

Simple case:

- program *P*
- partion of the program variables of *P* in
  - low security variables *low* and
  - high security variables *high*

Definition (Non-interference – Version 2)

For program *P* the high variables *high* do not interfere with the low variables *low*

$$\Leftrightarrow$$

running two instances of *P*, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.

Non-Interference

Prominent information flow property: non-interference

Simple case:

- program *P*
- partion of the program variables of *P* in
  - low security variables *low* and
  - high security variables *high*

Definition (Non-interference – Version 2)

For program *P* the high variables *high* do not interfere with the low variables *low*

$$\Leftrightarrow$$

running two instances of *P*, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.

Non-Interference

Prominent information flow property: non-interference

Simple case:

- program *P*
- partion of the program variables of *P* in
  - low security variables *low* and
  - high security variables *high*

Definition (Non-interference – Version 2)

For program *P* the high variables *high* do not interfere with the low variables *low*

$$\Leftrightarrow$$

running two instances of *P*, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.

Examples

Which methods are save?

```
class MiniExamples {
  public int l;
  private int h;

  void m_1() {
    l = h;
  }

  void m_2() {
    if (l>0) {h=1;}
    else {h=2;};
  }




  void m_3() {
    if (h>0) {l=1;}
    else {l=2;};
  }

  void m_4() {
    h=0; l=h;
  }
```

Examples

Which methods are save?

```
class MiniExamples {
  public int l;
  private int h;

  void m_1() {
    l = h;
  }

  void m_2() {
    if (l>0) {h=1;}
    else {h=2;};
  }




  void m_3() {
    if (h>0) {l=1;}
    else {l=2;};
  }

  void m_4() {
    h=0; l=h;
  }
```

Examples

Which methods are save?

```
class MiniExamples {
  public int l;
  private int h;

  void m_1() {
    l = h;
  }

  void m_2() {
    if (l>0) {h=1;}
    else {h=2;};
  }




  void m_3() {
    if (h>0) {l=1;}
    else {l=2;};
  }

  void m_4() {
    h=0; l=h;
  }
```

Examples

Which methods are save?

```
class MiniExamples {
  public int l;
  private int h;

  void m_1() {
    l = h;
  }

  void m_2() {
    if (l>0) {h=1;}
    else {h=2;};
  }



  void m_3() {
    if (h>0) {l=1;}
    else {l=2;};
  }

  void m_4() {
    h=0; l=h;
  }
```

Examples

Which methods are save?

```
class MiniExamples {
  public int l;
  private int h;

  void m_1() {
    l = h;
  }

  void m_2() {
    if (l>0) {h=1;}
    else {h=2;};
  }




  void m_3() {
    if (h>0) {l=1;}
    else {l=2;};
  }

  void m_4() {
    h=0; l=h;
  }
```

Examples

Which methods are save?

```
    void m_5() {
        l=h; l=l-h;
    }

```

```
    void m_6() {
        if (false) l=h;
    }

}
```

Examples

Which methods are save?

```
  void m_5() {
    l=h; l=l-h;
  }

```

```
  void m_6() {
    if (false) l=h;
  }

}
```

Examples

Which methods are save?

```
void m_5() {
    l=h; l=l-h;
}

void m_6() {
    if (false) l=h;
}

}
```

Formalising Non-Interference in JavaDL

Definition (Non-interference)

For program *P* the high variables *high* do not interfere with the low variables *low*

$$\Leftrightarrow$$

running two instances of *P*, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.

$$\begin{aligned}\forall l_{in} \forall h_{in}^1 \forall h_{in}^2 \forall l_{out}^1 \forall l_{out}^2 (\quad & \{low := l_{in} \mid\mid high := h_{in}^1\}[P]low = l_{out}^1 \\ \wedge \; & \{low := l_{in} \mid\mid high := h_{in}^2\}[P]low = l_{out}^2 \\ \rightarrow \; & l_{out}^1 = l_{out}^2 \quad)\end{aligned}$$

Formalising Non-Interference in JavaDL

Karlsruhe Institute of Technology

Definition (Non-interference)

For program *P* the high variables *high* do not interfere with the low variables *low*

$$\Leftrightarrow$$

running two instances of *P*, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.

$$\begin{aligned}\forall l_{in}\forall h^1_{in}\forall h^2_{in}\forall l^1_{out}\forall l^2_{out}(\quad & \{low := l_{in} \parallel high := h^1_{in}\}[P]low = l^1_{out} \\ \wedge\ & \{low := l_{in} \parallel high := h^2_{in}\}[P]low = l^2_{out} \\ \rightarrow\ & l^1_{out} = l^2_{out} \quad)\end{aligned}$$

Formalising Non-Interference in JavaDL

Definition (Non-interference)

For program *P* the high variables *high* do not interfere with the low variables *low*

$$\Leftrightarrow$$

running two instances of *P*, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.

$$\begin{aligned}\forall l_{in} \forall h^1_{in} \forall h^2_{in} \forall l^1_{out} \forall l^2_{out} (\quad & \{low := l_{in} \,||\, high := h^1_{in}\}[P]low = l^1_{out} \\ \wedge\ & \{low := l_{in} \,||\, high := h^2_{in}\}[P]low = l^2_{out} \\ \rightarrow\ & l^1_{out} = l^2_{out} \quad)\end{aligned}$$

Formalising Non-Interference in JavaDL

Definition (Non-interference)

For program *P* the high variables *high* do not interfere with the low variables *low*

$$\Leftrightarrow$$

running two instances of *P*, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.

$$\begin{aligned}\forall l_{in}\forall h^1_{in}\forall h^2_{in}\forall l^1_{out}\forall l^2_{out}(\quad & \{low := l_{in} \parallel high := h^1_{in}\}[P]low = l^1_{out} \\ \wedge & \{low := l_{in} \parallel high := h^2_{in}\}[P]low = l^2_{out} \\ \rightarrow & \; l^1_{out} = l^2_{out} \quad)\end{aligned}$$

Formalising Non-Interference in JavaDL

Definition (Non-interference)

For program *P* the high variables *high* do not interfere with the low variables *low*

$$\Leftrightarrow$$

running two instances of *P*, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.

$$\begin{aligned}\forall l_{in} \forall h^1_{in} \forall h^2_{in} \forall l^1_{out} \forall l^2_{out} (\quad & \{low := l_{in} \mid\mid high := h^1_{in}\}[P]low = l^1_{out} \\ \wedge\ & \{low := l_{in} \mid\mid high := h^2_{in}\}[P]low = l^2_{out} \\ \rightarrow\ & l^1_{out} = l^2_{out} \quad)\end{aligned}$$

Formalising Non-Interference in JavaDL

Definition (Non-interference)

For program *P* the high variables *high* do not interfere with the low variables *low*

$$\Leftrightarrow$$

running two instances of *P*, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.

$$\begin{aligned}\forall l_{in} \forall h_{in}^{1} \forall h_{in}^{2} \forall l_{out}^{1} \forall l_{out}^{2} (\quad & \{low := l_{in} \mid\mid high := h_{in}^{1}\}[P]low = l_{out}^{1} \\ & \wedge \{low := l_{in} \mid\mid high := h_{in}^{2}\}[P]low = l_{out}^{2} \\ & \rightarrow l_{out}^{1} = l_{out}^{2} \quad)\end{aligned}$$

Formalising Non-Interference in JavaDL

Definition (Non-interference)

For program *P* the high variables *high* do not interfere with the low variables *low*

$$\Leftrightarrow$$

running two instances of *P*, with equal values of the low variables, and arbitrary values for the high variables result in the low variables having equal values.

$$\begin{aligned}\forall l_{in}\forall h_{in}^{1}\forall h_{in}^{2}\forall l_{out}^{1}\forall l_{out}^{2}(\quad & \{low := l_{in} \parallel high := h_{in}^{1}\}[P]low = l_{out}^{1} \\ \land\ & \{low := l_{in} \parallel high := h_{in}^{2}\}[P]low = l_{out}^{2} \\ \to\ & l_{out}^{1} = l_{out}^{2} \quad)\end{aligned}$$

Simple Example

```
class SecurePasswordFile {
  private int[] names, passwords;
  //@ invariant names.length == passwords.length;
  public boolean check(int user, int password) {
    //@ loop_invariant ...
    for (int i = 0; i < names.length; i++) {
      if (names[i] == user &&
        passwords[i] == password) {
        return true;
      }
    }
    return false;
  }
}
```

Simple Example

```
class SecurePasswordFile {
   private int[] names, passwords;
   //@ invariant names.length == passwords.length;
   public boolean check(int user, int password) {
      //@ loop_invariant ...
      for (int i = 0; i < names.length; i++) {
         if (names[i] == user &&
            passwords[i] == password) {
            return true;
         }
      }
     return false;
   }
}
```

high variables

Simple Example

```
class SecurePasswordFile {
    private int[] names, passwords;
    //@ invariant names.length == passwords.length;
    public boolean check(int user, int password) {
      //@ loop_invariant ...
      for (int i = 0; i < names.length; i++) {
        if (names[i] == user &&
          passwords[i] == password) {
          return true;
        }
      }
     return false;
    }
}
```

high variables

low variables

Simple Example

```
     // General assumtions + class invariants //
     wellFormed(heap1) ∧ ...

     // Symbolic execution //
   ∧ { heap := heap1 }
     \[{ r = pwf.check(user, password); }\]
     r = outR1
   ∧ { heap := heap2 }
     \[{ r = pwf.check(user, password); }\]
     r = outR2

     // Comparision of the low variables //
  → outR1 = outR2
```

Simple Example

```
class SecurePasswordFile {
  private int[] names, passwords;
  //@ invariant names.length == passwords.length;
  public boolean check(int user, int password) {
    //@ loop_invariant ...
    for (int i = 0; i < names.length; i++) {
      if (names[i] == user &&
          passwords[i] == password) {
          return true;
      }
    }
    return false;
  }
}
```

JML – Security levels

How to define *low* and *high* variables in JML?

- Definition of *low* and *high* with respect to some security level.

Definition (Security level)

A security level is a set of heap locations.

- All heap locations of a security level are *low* with respect to that level, all other *high*.
- Definition of security levels in JML via model fields of type "location set".

JML – Security levels

Example

```
/*@ model \locset pwdFileManager;
  @ accessible pwdFileManager: footprint;
  @ represents pwdFileManager =
  @    names, names[*], passwords, passwords[*];
  @*/
```

Informal semantics:

- Set of locations defined by the evaluation of the model field in the current heap.
- Might evaluate to different security levels in different heaps.

JML – Preserving Security Levels

Example

```
/*@ normal_behavior
  @   ...
  @   respects    anyUser;
  @*/
boolean check(int user, int password) { ...
```

Informal semantics:

- Set of security levels for which a method fulfills the non-interference property.

JML – Parameter Dependencies

Example

```
/*@ normal_behavior
  @    ...
  @    secure_for checkUser, checkUser:checkUser;
  @*/
boolean check(int user, int password) { ...
```

Informal semantics:

- Parameter pre-condition: the value which is passed to the method depends at most on the specified locations.
- Return value post-condition: the return value depends at most on the specified locations.

JML – Declassification

Example

```
/*@ normal_behavior
  @   ...
  @   declassify ( \exists int i;
  @                   0 <= i && i < names.length;
  @                       names[i] == user
  @                    && passwords[i] == password
  @              )
  @              \from  pwdFileManager
  @              \to    checkUser
  @              \if    true;
  @*/
boolean check(int user, int password) { ...
```

JML – Declassification

Informal semantics:

- Information to be declassified in form of a term or formula.
- May depend at most on the locations specified in the “from“ part.
- May flow at most to the locations specified in then “to” part.
- Declassification only if the “if” part evaluates to true (in the pre-heap).

Semantic form of declassification:

- Declassification is part of the method contract.

Full Example – JML Specification

```
class SecurePasswordFile {

  /*@ model \locset checkUser;
    @ accessible checkUser: footprint;
    @ represents checkUser \such_that
    @         \subset(checkUser, footprint);
    @
    @ model \locset anyUser;
    @ accessible anyUser: footprint;
    @ represents anyUser \such_that
    @         \subset(anyUser, footprint);
    @
    @ invariant names.length == passwords.length;
    @*/
  private int[] names, passwords;
```

Full Example – JML Specification

```
  /*@ normal_behavior
    @   modifies      \nothing;
    @   secure_for    checkUser,checkUser:checkUser;
    @   respects      anyUser;
    @   declassify    ( \exists int i;
    @                     0 <= i && i < names.length;
    @                        names[i] == user
    @                     && passwords[i] == password
    @                 )
    @                 \to checkUser;
    @*/
  public boolean check(int user, int password) {
    ...
  }
}
```

Generalising the JavaDL Formalisation

Simple version for program variables partitioned into high and low variables:

$$\forall l_{in} \forall h^1_{in} \forall h^2_{in} \forall l^1_{out} \forall l^2_{out} (\quad \{low := l_{in} \parallel high := h^1_{in}\}[P] low = l^1_{out} \\ \land \{low := l_{in} \parallel high := h^2_{in}\}[P] low = l^2_{out} \\ \rightarrow l^1_{out} = l^2_{out} \quad)$$

Generalised version for arbitrary (definable) similarity relations $\sim_{in}$ and $\sim_{out}$ defined over program variables (heaps) h^1 and h^2:

$$\forall h^1_{in} \forall h^2_{in} \forall h^1_{out} \forall h^2_{out} (\quad \{heap := h^1_{in}\}[P] heap = h^1_{out} \\ \land \{heap := h^2_{in}\}[P] heap = h^2_{out} \\ \land h^1_{in} \sim_{in} h^2_{in} \\ \rightarrow h^1_{out} \sim_{out} h^2_{out} \quad)$$

Generalising the JavaDL Formalisation

Where $h^1_{in} \sim_{in} h^2_{in}$ has the form:

- All elements of the respects clause are low variables,

$$\forall Object\ o : \forall Field\ f : (o, f) \in \{heap := h^1_{in}\}respects$$
$$\rightarrow \{heap := h^1_{in}\}o.f = \{heap := h^2_{in}\}o.f$$

- all parameters with dependencies $\subseteq$ respects are low and

$$\wedge \bigwedge_{i \in \{1..n_{par}\}} (\{heap := h^1_{in}\}(secure\_for_i \subseteq respects) \rightarrow par^1_i = par^2_i)$$

- all declassifications with to-part $\subseteq$ respects are known.

$$\wedge \bigwedge_{i \in \{1..n_{decl}\}} (\{heap := h^1_{in}\}(to_i \subseteq respects)$$
$$\rightarrow (\{heap := h^1_{in}\}decl_i \leftrightarrow \{heap := h^2_{in}\}decl_i))$$

Generalising the JavaDL Formalisation

Where $h^1_{out} \sim_{out} h^2_{out}$ has the form:

- All elements of the respects clause are low variables,

$$\forall Object\ o : \forall Field\ f : (o, f) \in \{heap := h^1_{in}\} respects$$
$$\rightarrow \{heap := h^1_{out}\} o.f = \{heap := h^2_{out}\} o.f$$

- all parameters with dependencies $\subseteq$ respects are low and

$$\wedge \bigwedge_{i \in \{1..n_{par}\}} (\{heap := h^1_{in}\}(secure\_for_i \subseteq respects) \rightarrow par^1_i = par^2_i)$$

- if the return dependencies $\subseteq$ respects, then return is low.

$$\wedge (\{heap := h^1_{in}\}(secure\_for_{return} \subseteq respects) \rightarrow return^1 = return^2)$$

Full Example – JavaDL Formalisation

```
  // General Assumtions + Class Invariants
  wellFormed(heapAtPre1) ∧ ...

  // Symbolic Execution
∧ {heap:=heapAtPre1} \[{ ...

  // Input−Relation
∧ equalsAtLocs(heapAtPre1, heapAtPre2,
   {heap:=heapAtPre1}self.anyUser ∩ {})

∧ (  {heap:=heapAtPre1}self.passwordFileUser
   ⊆ {heap:=heapAtPre1}self.anyUser
   → user1 = user2 )

 ...
```

Full Example – JavaDL Formalisation

```
    // Input−Relation −− Declassification
  ∧ (   {heap:=heapAtPre1}self.passwordFileUser
      ⊆ {heap:=heapAtPre1}self.anyUser
      → (   {heap:=heapAtPre1}
                ∃ int i0;
                  (   0 ≤ i0 ∧ i0 < self.names.length
                   ∧ inInt(i0)
                   ∧ self.names[i0] = user1
                   ∧ self.passwords[i0] = password1)
          ↔ {heap:=heapAtPre2}
                ∃ int i1;
                  (   0 ≤ i1 ∧ i1 < self.names.length
                   ∧ inInt(i1)
                   ∧ self.names[i1] = user1
                   ∧ self.passwords[i1] = password1)))
```

Full Example – JavaDL Formalisation

```
    // Output-Relation
→  equalsAtLocs(heapAtPost1, heapAtPost2,
     {heap:=heapAtPre1}self.anyUser ∩ {})

 ∧ (  {heap:=heapAtPre1}self.passwordFileUser
      ⊆ {heap:=heapAtPre1}self.anyUser
      → result1 = result2 )

 ∧ (  {heap:=heapAtPre1}self.passwordFileUser
         ⊆ {heap:=heapAtPre1}self.anyUser
         → user1 = user2 )

    ...
```

Not tackled

- Comparison of objects.
- How to use information flow contracts.
- Quantitative analysis of specifications.