Procedure–Modular Verification of Control Flow Safety Properties

Siavash Soleimanifard
KTH, Stockholm

joint work with
Dilian Gurov
KTH, Stockholm
Marieke Huisman
University of Twente, The Netherlands

June 22, 2010
Modular Software Design

Modularity is helpful

- Complex and large systems
- Facilitating the reuse of components
## Modular Software Design

### Modularity is helpful
- Complex and large systems
- Facilitating the reuse of components

### Modularity in Software Verification
- Specifying components of a system, independently (locally)
- Specifying (global) property of the system
- Verifying the correctness of the system in independent two subtasks
  - (I) verifying local specifications, independently
  - (II) the composition of local specifications entails the global property
Modular Software Design

**Modularity is helpful**
- Complex and large systems
- Facilitating the reuse of components

**Modularity in Software Verification**
- Specifying components of a system, independently (locally)
- Specifying (global) property of the system
- Verifying the correctness of the system in independent two subtasks
  1. verifying local specifications, independently
  2. the composition of local specifications entails the global property

**Granularity**
- Different levels of granularity
  - Procedure–Modular
    - Modules are methods, e.g., Hoare logic
Our approach is algorithmic

- Accepts an annotated Java program as input
- Push-button tool support to verify the program
  - returns positive answer or negative answer with a counter example
Our Approach

Algorithmic Verification
- Our approach is algorithmic
  - Accepts an annotated Java program as input
  - Push-button tool support to verify the program
    - returns positive answer or negative answer with a counter example

Abstraction
- The price of algorithmic approach is abstraction
  - We abstract away from all data
  - Flow graphs
### Algorithmic Verification
- Our approach is algorithmic
  - Accepts an annotated Java program as input
  - Push-button tool support to verify the program
    - returns positive answer or negative answer with a counter example

### Abstraction
- The price of algorithmic approach is abstraction
  - We abstract away from all data
  - Flow graphs

### Properties
- We consider temporal safety properties of the control flow
  - Legal sequences of method invocations
Some Interesting Properties

Control Flow Safety Properties

A given method that changes certain sensitive data is only called from within another dedicated authentication method, i.e., unauthorized access is not possible.
Some Interesting Properties

Control Flow Safety Properties

- A given method that changes certain sensitive data is only called from within another dedicated authentication method, i.e., unauthorized access is not possible.
- In a voting system, candidate selection has to be finished, before the vote can be confirmed.
Some Interesting Properties

Control Flow Safety Properties

- A given method that changes certain sensitive data is only called from within another dedicated authentication method, i.e., unauthorized access is not possible.
- In a voting system, candidate selection has to be finished, before the vote can be confirmed.
- In a door access control system, the password has to be checked before the door is unlocked, and the password can only be changed if the door is unlocked.
Example of Tool Usage, Local Property

```java
/* @global_LTL_prop: */
/* even \rightarrow X ((even & entry) W odd) */
public class Number {
    /* @local_interface: requires {odd} */
    /* @local_prop: */
    nu X1. (([even call even] ff) \ (tau] X1) \ [even caret odd]
    nu X2. (([even call even] ff) \ (even caret odd] ff) \ ([tau] X2))
    */
    public boolean even(int n) {
        if (n == 0) return true;
        else return odd(n - 1);
    }
    /* @local_interface: requires {even} */
    /* @local_prop: */
    nu X1. (([odd call odd] ff) \ ([tau] X1) \ [odd caret even]
    nu X2. (([odd call odd] ff) \ ([odd caret even] ff) \ ([tau] X2))
    */
    public boolean odd(int n) {
        if (n == 0) return false;
        else return even(n - 1);
    }
}
Example of Tool Usage, Local Property

```java
/** @global_LTL_prop:
 *   even \implies X ((even && !entry) \until odd)
 */

public class Number {
    /** @local_interface: requires {odd}
 * @method even can only call method odd, and after returning from the call, no other
 * method can be called
 */

    public boolean even(int n) {
        if (n == 0) return true;
        else return odd(n-1);
    }

    /** @local_interface: requires {even}
 */

    public boolean odd(int n) {
        if (n == 0) return false;
        else return even(n-1);
    }
}
```
Example of Tool Usage, Local Property

```java
/* @global_LTL_prop:
 *  even -> X ((even && !entry) W odd)
 */

public class Number {
    /** @local_interface: requires {odd} */
    /**
     * method even can only call method odd, and after returning from the call, no other
     * method can be called
     */
    public boolean even(int n) {
        if (n == 0) return true;
        else return odd(n - 1);
    }
    /** @local_interface: requires {even} */
    /**
     * method odd can only call method even, and after returning from the call, no other
     * method can be called
     */
    public boolean odd(int n) {
        if (n == 0) return false;
        else return even(n - 1);
    }
}
```
Example of Tool Usage, Global Property

```java
/**
 * in every program execution starting in method even, the first call is not to method even itself
 */

public class Number {
    /** @local_interface: requires {odd} */
    /**
     * @local prop: 
     * nu X1 . ( ( [even call even] ff ) /
     *       ( [tau] X1 ) /
     *       [even caret odd]
     *   nu X2 . ( [even call even] ff ) /
     *       ( [even caret odd] ff ) /
     *       ( [tau] X2 )
     */
    public boolean even(int n) {
        if (n == 0) return true;
        else return odd(n-1);
    }
    /** @local_interface: requires {even} */
    /**
     * @local prop: 
     * nu X1 . ( [odd call odd] ff ) /
     *       ( [tau] X1 ) /
     *       [odd caret even]
     *   nu X2 . ( [odd call odd] ff ) /
     *       ( [odd caret even] ff ) /
     *       ( [tau] X2 )
     */
    public boolean odd(int n) {
        if (n == 0) return false;
        else return even(n-1);
    }
}
```
Example of Tool Usage, Verification Result

```java
/** @global_LTL_prop:
 * even -> X ((even && !entry) W odd)
 */
public class Number {
  /** @local_interface: requires {odd}
   * @local_prop:
   * nu X1. ([even call even] ff) \ ([tau] X1) \ [even caret odd]
   * nu X2. ([even call even] ff) \ ([even caret odd] ff) \ ([tau] X2))
   */
  public boolean even(int n) {
    if (n == 0) return true;
    else return odd(n-1);
  }
  /** @local_interface: requires {even}
   * @local_prop:
   * nu X1. ([odd call odd] ff) \ ([tau] X1) \ [odd caret even]
   * nu X2. ([odd call odd] ff) \ ([odd caret even] ff) \ ([tau] X2))
   */
  public boolean odd(int n) {
    if (n == 0) return false;
    else return even(n-1);
  }
}
```

Verification result:

```
' 'YES'
```
Example of Tool Usage, Verification Result

/**
 * in every program execution starting in method even, the first call IS to method even itself
 */

public class Number {
    /** @local_interface: requires {odd}
     *
     * @local_prop:
     *  nu X1. (([even call even] ff) \ (evencaretodd)
     *      nu X2. (([even call even] ff) \ (evencaretodd) ff) \ ([tau] X2)
     */
    public boolean even(int n) {
        if (n == 0) return true;
        else return odd(n - 1);
    }
    /** @local_interface: requires {even}
     *
     * @local_prop:
     *  nu X1. (([odd call odd] ff) \ (oddcaretEven)
     *      nu X2. (([odd call odd] ff) \ (oddcaretEven) ff) \ ([tau] X2)
     */
    public boolean odd(int n) {
        if (n == 0) return false;
        else return even(n - 1);
    }
}

Verification result: ‘‘NO’’

(even, ε) \[\textit{even} \text{call} \textit{odd}\] \rightarrow (odd, even) \[\textit{odd} \text{ret} \textit{even}\] \rightarrow (even, ε)
Overview

- Model and Logic
- Compositional Verification
- ProMoVer
- Case Study
- Conclusion
### Flow Graph Definition

**Flow Graphs:** represents the control flow structure  
**Flow Graph Interface:** required and provided methods
Flow Graph Definition

**Flow Graphs:** represents the control flow structure

**Flow Graph Interface:** required and provided methods

```java
class Number {
    public static boolean even(int n) {
        if (n == 0)
            return true;
        else
            return odd(n - 1);
    }

    public static boolean odd(int n) {
        if (n == 0)
            return false;
        else
            return even(n - 1);
    }
}
```

**Figure:** Flow graph of Number
Flow Graph Definition

**Flow Graphs:** represents the control flow structure

**Flow Graph Interface:** required and provided methods

class Number {
    public static boolean even(int n){
        if (n == 0)
            return true;
        else
            return odd(n-1);
    }
    public static boolean odd(int n){
        if (n == 0)
            return false;
        else
            return even(n-1);
    }
}

Figure: Flow graph of Number

Flow Graph Operator

**Flow Graph Composition (⊔):** disjoint union of flow graphs
Flow Graph

Flow Graph Behavior

- Flow graph induces push down automaton (PDA)
  - configurations \((v, \sigma)\): pairs of control point \(v\) and call stack \(\sigma\)
  - production induced by
    - non-call edges
    - call edges
    - return nodes

- Flow graph behavior is the behavior of induced PDA
class Number {
    public static boolean even(int n) {
        if (n == 0)
            return true;
        else
            return odd(n - 1);
    }

    public static boolean odd(int n) {
        if (n == 0)
            return false;
        else
            return even(n - 1);
    }
}

Figure: Flow graph of Number

\[(v_0, \varepsilon) \xrightarrow{\tau} (v_1, \varepsilon) \xrightarrow{\tau} (v_2, \varepsilon) \xrightarrow{\text{even call odd}} (v_5, v_3) \xrightarrow{\tau} (v_6, v_3) \xrightarrow{\tau} (v_8, v_3) \xrightarrow{\text{odd ret even}} (v_3, \varepsilon)\]
Simulation Logic

\[ \phi ::= p \mid \neg p \mid X \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2 \mid [a]\phi \mid \nu X. \phi \]
Simulation Logic

\[ \phi ::= p \mid \neg p \mid X \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2 \mid [a] \phi \mid \nu X. \phi \]

Example property in simulation logic

\[
\begin{align*}
\nu X_1. (([even \ call \ even] ff) \land ([\tau] X_1) \land [even \ call \ odd]) \\
\nu X_2. (([even \ call \ even] ff) \land ([even \ call \ odd] ff) \land ([\tau] X_2))
\end{align*}
\]
Simulation Logic

$$\phi ::= p \mid \neg p \mid X \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2 \mid [a] \phi \mid \nu X. \phi$$

- Example property in simulation logic
  
  $$nu \ X_1. \ ([\text{even call even}]ff) \land ([\tau]X_1) \land [\text{even call odd}]$$
  
  $$nu \ X_2. \ ([\text{even call even}]ff) \land ([\text{even call odd}]ff) \land ([\tau]X_2))$$

Weak LTL

$$\phi ::= p \mid \neg p \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2 \mid X \phi \mid G \phi \mid \phi_1 \ W \phi_2$$
Logics

Simulation Logic

\[ \phi ::= p \mid \neg p \mid X \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2 \mid [a]\phi \mid \nu X. \phi \]

- Example property in simulation logic
  
  \[ \nu X1. (([even \ call \ even]\text{ff}) \land ([\tau]X1) \land [even \ call \ odd]) \]
  
  \[ \nu X2. (([even \ call \ even]\text{ff}) \land ([even \ call \ odd]\text{ff}) \land ([\tau]X2)) \]

Weak LTL

\[ \phi ::= p \mid \neg p \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2 \mid X \phi \mid G \phi \mid \phi_1 W \phi_2 \]

- Example property in weak LTL
  
  \[ even \rightarrow X ((even \land \neg \text{entry}) W odd) \]
Compositional Verification

\[
\begin{array}{cccc}
C_1 & \cdots & C_i & \cdots & C_k \\
\end{array}
\]

\[
: \quad \phi
\]
Compositional Verification

\[ \psi_1 \rightarrow C_1 \rightarrow (\psi_1) \]

\[ \psi_i \rightarrow C_i \rightarrow (\psi_i) \]

\[ \psi_k \rightarrow C_k \rightarrow (\psi_k) \]

\[ \phi \]
Compositional Verification

\[ C_1 \quad \cdots \quad C_i \quad \cdots \quad C_k : \phi \]

\[ (\psi_1) \quad \cdots \quad (\psi_i) \quad \cdots \quad (\psi_k) \]
Compositional Verification

\[
\begin{align*}
C_1 & \quad \cdots \quad C_i & \quad \cdots \quad C_k \\
(\psi_1) & \quad \cdots \quad (\psi_i) & \quad \cdots \quad (\psi_k)
\end{align*}
\]

\[
\vdash \phi
\]

\[
\vdash \phi
\]
Compositional Verification Based on Maximal Flow Graphs

\[ C_1 \quad \ldots \quad C_i \quad \ldots \quad C_k : \phi \]

\[ (\psi_1) \quad \ldots \quad (\psi_i) \quad \ldots \quad (\psi_k) : \phi \]

\[ \text{Const. Const. Const.} \]

\[ \text{Max}(\psi_1, l_1) \quad \ldots \quad \text{Max}(\psi_i, l_i) \quad \ldots \quad \text{Max}(\psi_k, l_k) \]
Compositional Verification Based on Maximal Flow Graphs

\[
\begin{align*}
C_1 & \quad \ldots \quad C_i \quad \ldots \quad C_k \\
\vdots & \quad \vdots \quad \vdots \quad \vdots \\
(\psi_1) & \quad \ldots \quad (\psi_i) \quad \ldots \quad (\psi_k) \\
\vdots & \quad \vdots \quad \vdots \quad \vdots \\
\text{Max}(\psi_1, l_1) & \quad \uplus \quad \text{Max}(\psi_i, l_i) \quad \uplus \quad \text{Max}(\psi_k, l_k)
\end{align*}
\]

\[
\models \phi
\]
## Procedure–Modular Verification

**I**  
- Extract flow graph for each method and model check it against its local property

**II**  
- Construct maximal model from local property and interface of each method
- Compose the maximal models and model check the composition result against global property
Figure: Overview of ProMoVer and its underlying tool set
Case Study

Program

- JavaPurse: a Java Card application for electronic purse
- Uses **Transaction mechanism** for atomic update operations
- 19 methods
- Around 1000 lines of Java code
- With 222 method invocations, 21 method calls to **NonAtomic** methods
Global Property

- non-atomic array operation should not be invoked within a transaction

\[ G(\text{beginTransaction} \rightarrow \neg \text{NonAtomicOp W commitTransaction}) \]
Case Study

Global Property
- non-atomic array operation should not be invoked within a transaction

\[ G(\text{beginTransaction} \rightarrow \neg \text{NonAtomicOpW\ commitTransaction}) \]

Local Specifications
- The implementation was available
- Specification: capture the method invocation ordering
- It is possible to write specification independent from the implementation
## Case Study

### Global Property
- non-atomic array operation should not be invoked within a transaction

\[ G(\text{beginTransaction} \rightarrow \neg \text{NonAtomicOp W commitTransaction}) \]

### Local Specifications
- The implementation was available
- Specification: capture the method invocation ordering
- It is possible to write specification independent from the implementation

### Verification Result
- Positive answer in 150 seconds
- Task(/) performed in 142 seconds
  - Analyzer(SOOT) needed 141 seconds
- Task(//) performed in 4 seconds
Conclusions

ProMoVer

An automated tool for procedure–modular verification

- Verifies temporal safety properties
- Gets annotated Java programs
- Fully automated

We evaluated ProMoVer by a small but realistic case study
  - The results seem promising
    - Handle a real case study
Conclusions

**ProMoVer**

An automated tool for procedure–modular verification

- Verifies temporal safety properties
- Gets annotated Java programs
- Fully automated

We evaluated ProMoVer by a small but realistic case study

- The results seem promising
  - Handle a real case study

**Improvements Needed**

- Replace Analyzer(SOOT)
- To support for alternative notations
Future Work

Prove Reuse

To provide support for prove reuse
Future Work

Prove Reuse
To provide support for prove reuse

Scalability
Investigate the scalability of the approach
- Evaluate our approach by a larger case study
- Interface abstraction by in-lining private methods
Future Work

Prove Reuse
To provide support for prove reuse

Scalability
Investigate the *scalability* of the approach
- Evaluate our approach by a larger case study
- Interface abstraction by in-lining private methods

Wider Range of Properties
To find more interesting properties
- by adding data
  - by using Boolean programs
Questions
Maximal Flow Graph for property $\psi$, is a flow graph that simulates all flow graphs holding $\psi$.

**Figure:** Maximal Flow graph of Number