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
  - (I) verifying local specifications, independently
  - (II) 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

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

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

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

Properties

- We consider temporal safety properties of the control flow
  - Legal sequences of method invocations

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

```
/* @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);
   }
}
```

Example of Tool Usage, Local Property

```
/* @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}
    *
    *  @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

```
/* @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

in every program execution starting in method even, the first call is not to method even itself

```
/**
 *
 */
public class Number {
   /** @local_interface: requires {odd}
    *
    *  @
    *
    *
    */
   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);
   }
}
```

method even can only call method odd, and after returning from the call, no other method can be called

method odd can only call method even, and after returning from the call, no other method can be called

Example of Tool Usage, Verification Result

```
/** @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) /\ ([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: ‘‘NO’’

$$(\text{even}, \varepsilon) \xrightarrow{\texttt{even call odd}} (\text{odd}, \text{even}) \xrightarrow{\texttt{odd ret even}} (\text{even}, \varepsilon)$$

Overview

- Model and Logic
- Compositional Verification
- ProMoVer
- Case Study
- Conclusion

Flow Graph

Flow Graph Definition

Flow Graphs: *represents the control flow structure*

Flow Graph Interface: *required and provided methods*

Flow Graph

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);

   }}
```

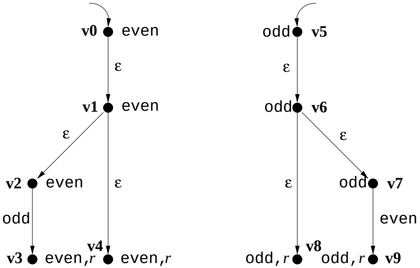


Figure: Flow graph of Number

Flow Graph

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);

   }}
```

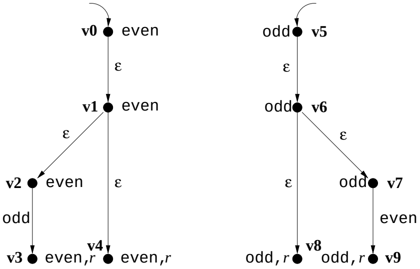


Figure: Flow graph of Number

Flow Graph Operator

Flow Graph Composition ($\uplus$): *disjoint union of flow graphs*

Flow Graph

Flow Graph Behavior

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

Behavior of Closed Flow Graph

```
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);

   }}
```

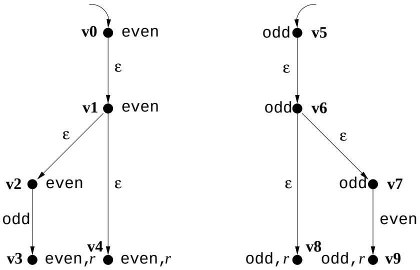


Figure: Flow graph of Number

$$(v_0, \varepsilon) \xrightarrow{\tau} (v_1, \varepsilon) \xrightarrow{\tau} (v_2, \varepsilon) \xrightarrow{\texttt{even call odd}} (v_5, v_3) \xrightarrow{\tau} (v_6, v_3) \xrightarrow{\tau}$$
$$(v_8, v_3) \xrightarrow{\texttt{odd ret even}} (v_3, \varepsilon)$$

Simulation Logic

$$\phi ::= p \mid \neg p \mid X \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid [a]\phi \mid \nu X.\, \phi$$

Logics

Simulation Logic

$$\phi ::= p \mid \neg p \mid X \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid [a]\phi \mid \nu X.\, \phi$$

- Example property in simulation logic

$$nu\ X1.\ (([even\ \texttt{call}\ even]ff) \wedge ([tau]X1) \wedge [even\ \texttt{call}\ odd]$$
$$nu\ X2.\ (([even\ \texttt{call}\ even]ff) \wedge ([even\ \texttt{call}\ odd]ff) \wedge ([tau]X2))$$

Logics

Simulation Logic

$$\phi ::= p \mid \neg p \mid X \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid [a]\phi \mid \nu X.\, \phi$$

- Example property in simulation logic

$$\begin{aligned}&\mathit{nu}\ X1.\ (([\mathit{even}\ \texttt{call}\ \mathit{even}]\mathit{ff}) \wedge ([\mathit{tau}]X1) \wedge [\mathit{even}\ \texttt{call}\ \mathit{odd}]\\&\quad \mathit{nu}\ X2.\ (([\mathit{even}\ \texttt{call}\ \mathit{even}]\mathit{ff}) \wedge ([\mathit{even}\ \texttt{call}\ \mathit{odd}]\mathit{ff}) \wedge ([\mathit{tau}]X2))\end{aligned}$$

Weak LTL

$$\phi ::= p \mid \neg p \mid \phi_1 \wedge \phi_2 \mid \phi_1\ \vee\ \phi_2 \mid \texttt{X}\ \phi \mid \texttt{G}\ \phi \mid \phi_1\ \texttt{W}\ \phi_2$$

Logics

Simulation Logic

$$\phi ::= p \mid \neg p \mid X \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid [a]\phi \mid \nu X.\, \phi$$

- Example property in simulation logic

$$nu\ X1.\ (([even\ \texttt{call}\ even]ff) \wedge ([tau]X1) \wedge [even\ \texttt{call}\ odd]$$
$$nu\ X2.\ (([even\ \texttt{call}\ even]ff) \wedge ([even\ \texttt{call}\ odd]ff) \wedge ([tau]X2))$$

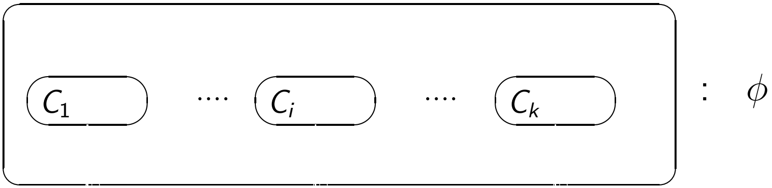
Weak LTL

$$\phi ::= p \mid \neg p \mid \phi_1 \wedge \phi_2 \mid \phi_1\ \vee\ \phi_2 \mid \mathtt{X}\ \phi \mid \mathtt{G}\ \phi \mid \phi_1\ \mathtt{W}\ \phi_2$$

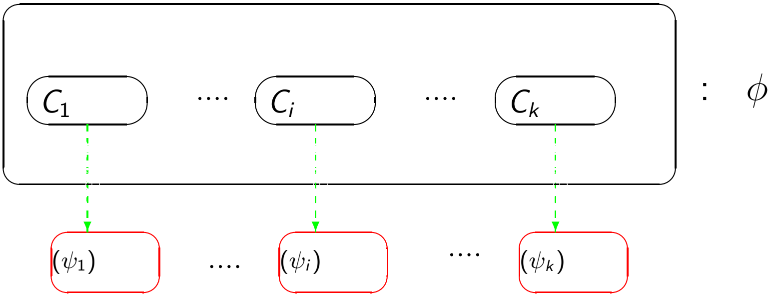
- Example property in weak LTL

$$even \rightarrow\ X\ ((even\ \wedge\ \neg entry)\ W\ odd)$$

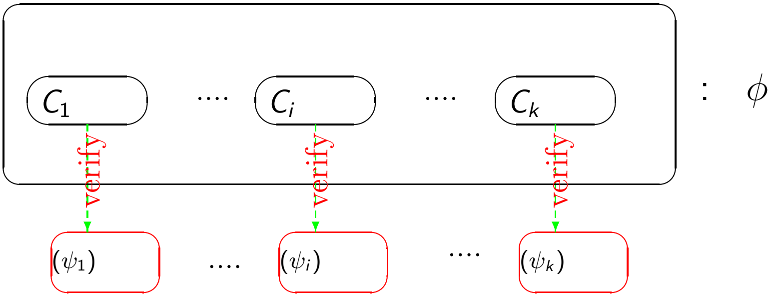
Compositional Verification



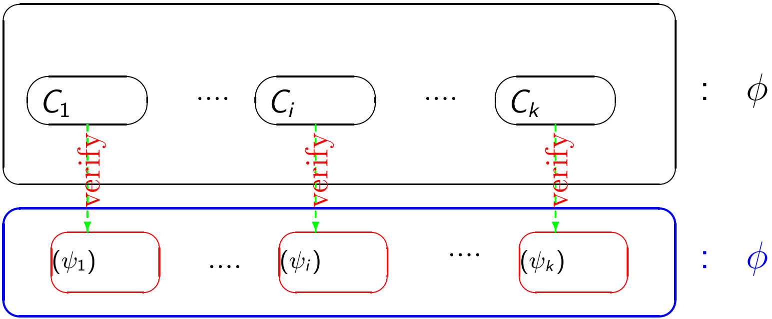
Compositional Verification



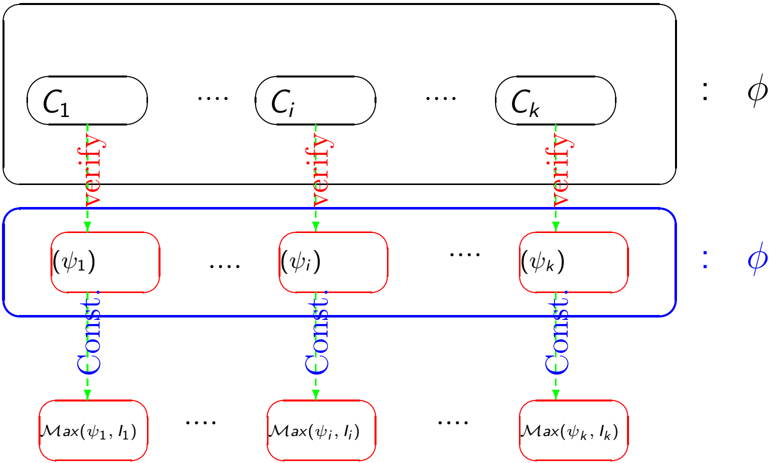
Compositional Verification



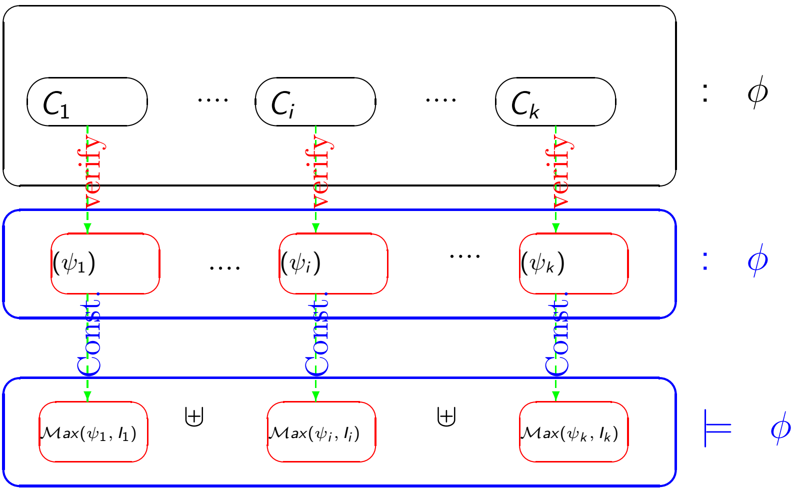
Compositional Verification



Compositional Verification Based on Maximal Flow Graphs



Compositional Verification Based on Maximal Flow Graphs



Procedure–Modular Verification

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

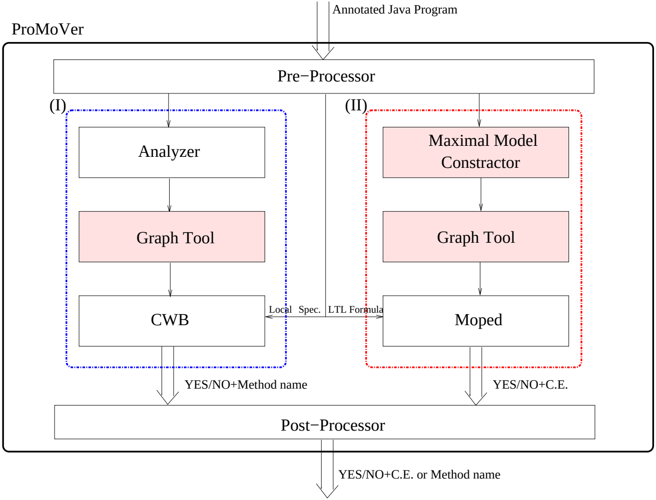


Figure: Overview of ProMoVer and its underlying tool set

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

Case Study

Global Property

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

$$\mathtt{G}\,(\mathtt{beginTransaction} \rightarrow \neg\mathtt{NonAtomicOp}\;\mathtt{W}\;\mathtt{commitTransaction})$$

Case Study

Global Property

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

$$\begin{aligned}&\mathtt{G}\,(\mathtt{beginTransaction} \rightarrow \\ &\qquad \neg\mathtt{NonAtomicOp}\,\mathtt{W}\,\mathtt{commitTransaction})\end{aligned}$$

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

$$\texttt{G}\,(\texttt{beginTransaction} \rightarrow \neg\texttt{NonAtomicOp}\,\texttt{W}\,\texttt{commitTransaction})$$

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(I) performed in 142 seconds
  - Analyzer(Soot) needed 141 seconds
- Task(II) 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 Graphs

Maximal Flow Graph for property ψ, is a flow graph that simulates all flow graphs holding ψ.

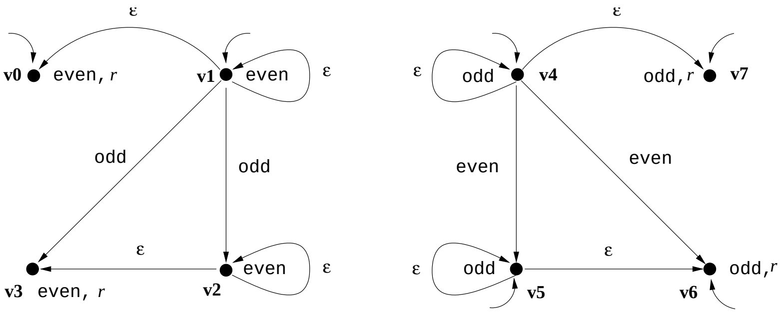


Figure: Maximal Flow graph of Number