Modular Verification of Temporal Safety Properties of Procedural Programs

Dilian Gurov

KTH Royal Institute of Technology, Stockholm

SEFM 2011 Tutorial, Montevideo, 15 November 2011

Two out of three

Two out of three

- Modular verification of temporal properties:
  Grumberg & Long 1994: finite–state systems, ACTL
  Kupferman & Vardi 2000: finite–state systems, ACTL*
  based on maximal model construction

Two out of three

- Modular verification of temporal properties:
  Grumberg & Long 1994: finite–state systems, ACTL
  Kupferman & Vardi 2000: finite–state systems, ACTL*
  based on maximal model construction

- Modular verification of procedural programs:
  "built–in" for Hoare–logic based approaches

Two out of three

- Modular verification of temporal properties:
  Grumberg & Long 1994: finite–state systems, ACTL
  Kupferman & Vardi 2000: finite–state systems, ACTL*
  based on maximal model construction

- Modular verification of procedural programs:
  "built–in" for Hoare–logic based approaches

- Model checking procedural programs:
  Das, Lerner & Seigl 2002: property simulation (ESP)
  Esparza et al 2002: model checking pushdown systems

This work

- started in 2001
- original goal: verify Javacard programs in the presence of post–issuance loading of applets on smart cards
- joint work with Marieke Huisman, Christoph Sprenger, Irem Aktug, Siavash Soleimanifard, Afshin Amighi, Pedro Gomez

Compositionality and Modularity

Compositionality and Modularity

Compositionality as a *mathematical principle*:

- express the meaning of the whole through the meaning of the parts
- example: denotational semantics
- example: definitions and proofs by structural induction

Compositionality and Modularity

Compositionality as a *mathematical principle*:

- express the meaning of the whole through the meaning of the parts
- example: denotational semantics
- example: definitions and proofs by structural induction

Modularity as a *systems design principle*:

- control the complexity of the system
  by braking it down into manageable pieces that are
  designed, implemented, tested and maintained *independently*

Verification

Verification

Verification as a *systems design task*:

- match a model of the system against a specification

Verification

Verification as a *systems design task*:

- match a model of the system against a specification

Modular Verification:

- specify and verify every module independently
- infer system correctness from module correctness

Verification

Verification as a *systems design task*:

- match a model of the system against a specification

Modular Verification:

- specify and verify every module independently
- infer system correctness from module correctness
  i.e., *relativize* global properties on local ones

Verification

Verification as a *systems design task*:

- match a model of the system against a specification

Modular Verification:

- specify and verify every module independently
- infer system correctness from module correctness
  i.e., *relativize* global properties on local ones

This relativization allows verification in the presence of *variability*

Variability

Variability

Temporal variability:

- static code evolution
- dynamic code replacement
- dynamic code loading: code not available at verification time

Variability

Temporal variability:

- static code evolution
- dynamic code replacement
- dynamic code loading: code not available at verification time

Spacial variablility:

- multiple variants, as arising from software product lines

Verification in the presence of variability

Consider a system with four modules (components):

Verification in the presence of variability

Consider a system with four modules (components):

- A implemented, stable

Verification in the presence of variability

Consider a system with four modules (components):

- A implemented, stable
- B implemented, expected to evolve

Verification in the presence of variability

Consider a system with four modules (components):

- A implemented, stable
- B implemented, expected to evolve
- C implemented, multiple variants

Verification in the presence of variability

Consider a system with four modules (components):

- A implemented, stable
- B implemented, expected to evolve
- C implemented, multiple variants
- D not yet implemented/available

Verification in the presence of variability

Consider a system with four modules (components):

- A implemented, stable
- B implemented, expected to evolve
- C implemented, multiple variants
- D not yet implemented/available

How shall one plan for the verification of a global property ψ?

Verification in the presence of variability

Consider a system with four modules (components):

- A implemented, stable
- B implemented, expected to evolve
- C implemented, multiple variants
- D not yet implemented/available

How shall one plan for the verification of a global property ψ?

- as early as possible

Verification in the presence of variability

Consider a system with four modules (components):

- A implemented, stable
- B implemented, expected to evolve
- C implemented, multiple variants
- D not yet implemented/available

How shall one plan for the verification of a global property ψ?

- as early as possible
- with minimal effort: reuse results

Relativization

Relativization

Relativize global property on local specifications. Three tasks:

Relativization

Relativize global property on local specifications. Three tasks:

1. specify modules B, C, D

Relativization

Relativize global property on local specifications. Three tasks:

1. specify modules B, C, D
2. verify

$$impl(B) \models spec(B)$$

$$impl(C) \models spec(C)$$

$$impl(D) \models spec(D)$$

Relativization

Relativize global property on local specifications. Three tasks:

1. specify modules B, C, D
2. verify

$$\mathit{impl}(B) \models \mathit{spec}(B)$$

$$\mathit{impl}(C) \models \mathit{spec}(C)$$

$$\mathit{impl}(D) \models \mathit{spec}(D)$$

3. verify

$$\mathit{impl}(A) + \mathit{spec}(B) + \mathit{spec}(C) + \mathit{spec}(D) \models \psi$$

Relativization

Relativize global property on local specifications. Three tasks:

1. specify modules B, C, D
2. verify
$$impl(B) \models spec(B)$$
$$impl(C) \models spec(C)$$
$$impl(D) \models spec(D)$$
3. verify
$$impl(A) + spec(B) + spec(C) + spec(D) \models \psi$$

But... how, and is there an algorithmic solution?

Program Models

One approach is to use a unifying formal model to represent modules and whole programs.

Program Models

One approach is to use a unifying formal model to represent modules and whole programs. Then, for the second task:

$$impl(B) \models spec(B)$$

$$impl(C) \models spec(C)$$

$$impl(D) \models spec(D)$$

perform the following steps:

Program Models

One approach is to use a unifying formal model to represent modules and whole programs. Then, for the second task:

$$impl(B) \models spec(B)$$

$$impl(C) \models spec(C)$$

$$impl(D) \models spec(D)$$

perform the following steps:

1. from module implementations: extract models

Program Models

One approach is to use a unifying formal model to represent modules and whole programs. Then, for the second task:

$$impl(B) \models spec(B)$$

$$impl(C) \models spec(C)$$

$$impl(D) \models spec(D)$$

perform the following steps:

1. from module implementations: extract models
2. model check models against local specifications:

$$mod(impl(B)) \models spec(B)$$

$$mod(impl(C)) \models spec(C)$$

$$mod(impl(D)) \models spec(D)$$

Program Models

For the third task:

$$\mathit{impl}(A) + \mathit{spec}(B) + \mathit{spec}(C) + \mathit{spec}(D) \models \psi$$

perform the following steps:

Program Models

For the third task:

$$impl(A) + spec(B) + spec(C) + spec(D) \models \psi$$

perform the following steps:

1. from module implementations: extract models

Program Models

For the third task:

$$impl(A) + spec(B) + spec(C) + spec(D) \models \psi$$

perform the following steps:

1. from module implementations: extract models
2. from module specifications: construct (so-called maximal) models

Program Models

For the third task:

$$impl(A) + spec(B) + spec(C) + spec(D) \models \psi$$

perform the following steps:

1. from module implementations: extract models
2. from module specifications: construct (so-called maximal) models
3. compose extracted with constructed models

Program Models

For the third task:

$$impl(A) + spec(B) + spec(C) + spec(D) \models \psi$$

perform the following steps:

1. from module implementations: extract models
2. from module specifications: construct (so-called maximal) models
3. compose extracted with constructed models
4. model check composed model against global property ψ:
   $mod(impl(A)) + max(spec(B)) + max(spec(C)) + max(spec(D)) \models \psi$

Simulation: A refinement pre–order on models

Simulation: A refinement pre–order on models

We require the following conditions:

Simulation: A refinement pre–order on models

We require the following conditions:

1. extracted models simulate module implementations

Simulation: A refinement pre–order on models

We require the following conditions:

1. extracted models simulate module implementations
2. maximal models simulate models satisfying module specifications

Simulation: A refinement pre–order on models

We require the following conditions:

1. extracted models simulate module implementations
2. maximal models simulate models satisfying module specifications
3. simulation is monotone w.r.t. composition

Simulation: A refinement pre–order on models

We require the following conditions:

1. extracted models simulate module implementations
2. maximal models simulate models satisfying module specifications
3. simulation is monotone w.r.t. composition
4. simulation preserves properties (backwards)

Simulation: A refinement pre–order on models

We require the following conditions:

1. extracted models simulate module implementations
2. maximal models simulate models satisfying module specifications
3. simulation is monotone w.r.t. composition
4. simulation preserves properties (backwards)

The third task:

$$mod(impl(A)) + max(spec(B)) + max(spec(C)) + max(spec(D)) \models \psi$$

thus entails:

$$impl(A) + impl(B) + impl(C) + impl(D) \models \psi$$

Our Setup

Our Setup

Program model: Flow graphs capturing purely control flow

- behaviour as induced pushdown automaton

Our Setup

Program model: Flow graphs capturing purely control flow

- behaviour as induced pushdown automaton

Properties: legal sequences of method invocations

- temporal safety properties

Our Setup

Program model: Flow graphs capturing purely control flow

- behaviour as induced pushdown automaton

Properties: legal sequences of method invocations

- temporal safety properties

Verification: pushdown automata model checking

- essentially a language inclusion problem

Our Setup

Program model: Flow graphs capturing purely control flow

- behaviour as induced pushdown automaton

Properties: legal sequences of method invocations

- temporal safety properties

Verification: pushdown automata model checking

- essentially a language inclusion problem

Most details in:
Compositional Verification of Sequential Programs with Procedures
Dilian Gurov, Marieke Huisman and Christoph Sprenger
Journal of Information and Computation
vol. 206, no. 7, pp. 840–868, 2008

Tutorial Outline

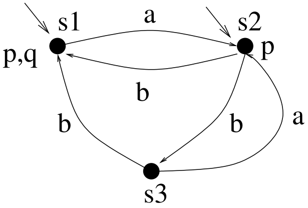
Tutorial Outline

1. Preliminaries: Models, Simulation, Logic
2. Flow Graphs, Behaviour and Extraction
3. Property Specification and Verification
4. Maximal Flow Graphs
5. Tool Support
6. Application: Software Product Lines
7. Conclusion

1. Models, Simulation, Logic

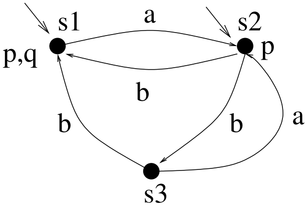
1. Models, Simulation, Logic

A model $\mathcal{M}$



1. Models, Simulation, Logic

A model $\mathcal{M}$



Definition (Model)

A structure $\mathcal{M} = (S, L, \rightarrow, A, \lambda)$ where:

(i) S a set of states

(ii) L a set of transition labels

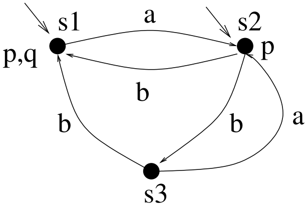
(iii) $\rightarrow \subseteq S \times L \times S$ a transition relation

(iv) A a set of atomic propositions

(v) $\lambda : S \rightarrow \mathcal{P}(A)$ a valuation

1. Models, Simulation, Logic

A model $\mathcal{M}$



Definition (Model)

A structure $\mathcal{M} = (S, L, \rightarrow, A, \lambda)$ where:

(i) S a set of states

(ii) L a set of transition labels

(iii) $\rightarrow \subseteq S \times L \times S$ a transition relation

(iv) A a set of atomic propositions

(v) $\lambda : S \rightarrow \mathcal{P}(A)$ a valuation

An intialised model $(\mathcal{M}, E)$ is a model $\mathcal{M}$ with a designated set of entry states $E \subseteq S$

Simulation

Let $\mathcal{M}_1 = (S_1, L, \rightarrow_1, A, \lambda_1)$ and $\mathcal{M}_2 = (S_2, L, \rightarrow_2, A, \lambda_2)$ be models over the same sets of labels and atomic propositions.

Simulation

Let $\mathcal{M}_1 = (S_1, L, \rightarrow_1, A, \lambda_1)$ and $\mathcal{M}_2 = (S_2, L, \rightarrow_2, A, \lambda_2)$ be models over the same sets of labels and atomic propositions.

Definition (Simulation)

- A binary relation $R \subseteq S_1 \times S_2$ is a simulation if whenever $(s_1, s_2) \in R$
  - (i) $\lambda_1(s_1) = \lambda_2(s_2)$
  - (ii) for any $a \in L$ and $s_1' \in S_1$
    $s_1 \xrightarrow{a}_1 s_1'$ entails $s_2 \xrightarrow{a}_2 s_2'$ for some $s_2' \in S_2$ such that $(s_1', s_2') \in R$

Simulation

Let $\mathcal{M}_1 = (S_1, L, \rightarrow_1, A, \lambda_1)$ and $\mathcal{M}_2 = (S_2, L, \rightarrow_2, A, \lambda_2)$ be models over the same sets of labels and atomic propositions.

Definition (Simulation)

- A binary relation $R \subseteq S_1 \times S_2$ is a simulation if whenever $(s_1, s_2) \in R$
  - (i) $\lambda_1(s_1) = \lambda_2(s_2)$
  - (ii) for any $a \in L$ and $s_1' \in S_1$
    $s_1 \xrightarrow{a}_1 s_1'$ entails $s_2 \xrightarrow{a}_2 s_2'$ for some $s_2' \in S_2$ such that $(s_1', s_2') \in R$
- $s_2 \in S_2$ simulates $s_1 \in S_1$ if
  there is a simulation relation R so that $(s_1, s_2) \in R$

Simulation

Let $\mathcal{M}_1 = (S_1, L, \rightarrow_1, A, \lambda_1)$ and $\mathcal{M}_2 = (S_2, L, \rightarrow_2, A, \lambda_2)$ be models over the same sets of labels and atomic propositions.

Definition (Simulation)

- A binary relation $R \subseteq S_1 \times S_2$ is a simulation if whenever $(s_1, s_2) \in R$
  - (i) $\lambda_1(s_1) = \lambda_2(s_2)$
  - (ii) for any $a \in L$ and $s_1' \in S_1$
    $s_1 \xrightarrow{a}_1 s_1'$ entails $s_2 \xrightarrow{a}_2 s_2'$ for some $s_2' \in S_2$ such that $(s_1', s_2') \in R$
- $s_2 \in S_2$ simulates $s_1 \in S_1$ if
  there is a simulation relation R so that $(s_1, s_2) \in R$
- $(\mathcal{M}_2, E_2)$ simulates $(\mathcal{M}_1, E_1)$ if
  every $s_1 \in E_1$ is simulated by some $s_2 \in E_2$

Logic

Logic

Definition (Simulation Logic)

The formulas of the logic are inductively defined through the BNF:

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

where $p \in A$ and $a \in L$

Logic

Definition (Simulation Logic)

The formulas of the logic are inductively defined through the BNF:

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

where $p \in A$ and $a \in L$

Example

Some example formulas and their meaning:

Logic

Definition (Simulation Logic)

The formulas of the logic are inductively defined through the BNF:

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

where $p \in A$ and $a \in L$

Example

Some example formulas and their meaning:

- $[a]\,\mathrm{ff}$

Logic

Definition (Simulation Logic)

The formulas of the logic are inductively defined through the BNF:

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

where $p \in A$ and $a \in L$

Example

Some example formulas and their meaning:

- $[a]\,\mathrm{ff}$
- $[a]\,\mathrm{ff} \wedge [b]\,\mathrm{ff}$

Logic

Definition (Simulation Logic)

The formulas of the logic are inductively defined through the BNF:

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

where $p \in A$ and $a \in L$

Example

Some example formulas and their meaning:

- $[a]\,\mathrm{ff}$
- $[a]\,\mathrm{ff} \wedge [b]\,\mathrm{ff}$
- $[a]\,\mathrm{ff} \vee [b]\,\mathrm{ff}$

Logic

Definition (Simulation Logic)

The formulas of the logic are inductively defined through the BNF:

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

where $p \in A$ and $a \in L$

Example

Some example formulas and their meaning:

- $[a]\,\mathrm{ff}$
- $[a]\,\mathrm{ff} \wedge [b]\,\mathrm{ff}$
- $[a]\,\mathrm{ff} \vee [b]\,\mathrm{ff}$
- $\nu X.\ p \wedge [a]\,\mathrm{ff} \wedge [b]\,X$

Maximal Models

Maximal Models

Definition (Maximal Model)

A maximal model for a formula ϕ is an initialized model S such that:

(i) S satisfies ϕ

(si) S simulates all initialized models satisfying ϕ

Maximal Models

Definition (Maximal Model)

A maximal model for a formula ϕ is an initialized model S such that:

(i) S satisfies ϕ

(si) S simulates all initialized models satisfying ϕ

Theorem

Every simulation logic formula ϕ has a maximal model S_ϕ

Maximal Models

Definition (Maximal Model)

A maximal model for a formula ϕ is an initialized model S such that:

(i) S satisfies ϕ

(si) S simulates all initialized models satisfying ϕ

Theorem

Every simulation logic formula ϕ has a maximal model S_ϕ

Corollary

Maximal models are unique up to simulation equivalence

Constructing Maximal Models

Labels $\{a, b\}$, atoms $\{p\}$, formula $[b]\,\mathrm{ff} \wedge p$

Constructing Maximal Models

Labels $\{a, b\}$, atoms $\{p\}$, formula $[b]\,\mathrm{ff} \wedge p$

The formula as an *equation system*:

$$X = [b]\,\mathrm{ff} \wedge p$$

Constructing Maximal Models

Labels $\{a, b\}$, atoms $\{p\}$, formula $[b]\,\mathrm{ff} \wedge p$

The formula as an *equation system*:

$$X = [b]\,\mathrm{ff} \wedge p$$

convert into *simulation normal form*:

$$\begin{array}{rcl} X & = & [a]\,(Y_1 \vee Y_2) \wedge [b]\,\mathrm{ff} \wedge p \\ Y_1 & = & [a]\,(Y_1 \vee Y_2) \wedge [b]\,(Y_1 \vee Y_2) \wedge p \\ Y_2 & = & [a]\,(Y_1 \vee Y_2) \wedge [b]\,(Y_1 \vee Y_2) \wedge \neg p \end{array}$$

Constructing Maximal Models

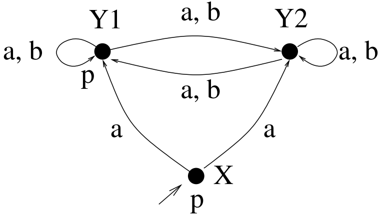
Labels $\{a, b\}$, atoms $\{p\}$, formula $[b]\,\mathrm{ff} \wedge p$

The formula as an *equation system*:

$$X = [b]\,\mathrm{ff} \wedge p$$

convert into *simulation normal form*:

$$\begin{array}{lcl} X & = & [a]\,(Y_1 \vee Y_2) \wedge [b]\,\mathrm{ff} \wedge p \\ Y_1 & = & [a]\,(Y_1 \vee Y_2) \wedge [b]\,(Y_1 \vee Y_2) \wedge p \\ Y_2 & = & [a]\,(Y_1 \vee Y_2) \wedge [b]\,(Y_1 \vee Y_2) \wedge \neg p \end{array}$$



$(\mathcal{M}, E)$

2. Flow Graphs, Interfaces and Behaviour

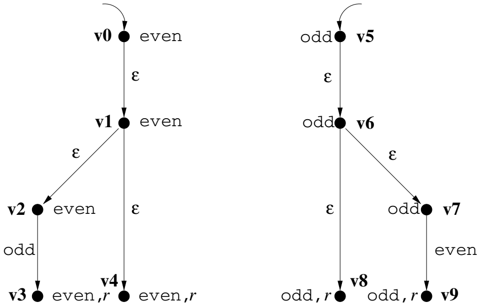
Flow Graphs: The structure of program control flow (as a model)

```
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);

   }
}
```



2. Flow Graphs, Interfaces and Behaviour

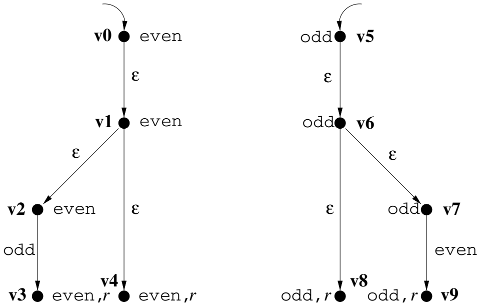
Flow Graphs: The structure of program control flow (as a model)

```
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);

   }
}
```



Interfaces: provided and required methods

Flow Graph Behaviour

A flow graph induces a *pushdown automaton* (PDA):

- configurations (v, σ) are pairs of control point and call stack
- productions induced by:
  - non–call edges: stack unchanged, rewrite control point
  - call edges: push target node on stack, new control point is entry node of called method
  - return nodes: pop stack, new control point is old top of stack

Flow Graph Behaviour

A flow graph induces a *pushdown automaton* (PDA):

- configurations (v, σ) are pairs of control point and call stack
- productions induced by:
  - non–call edges: stack unchanged, rewrite control point
  - call edges: push target node on stack, new control point is entry node of called method
  - return nodes: pop stack, new control point is old top of stack

The behaviour of a flow graph is the behaviour of the induced PDA (again a model)

Flow Graph Behaviour

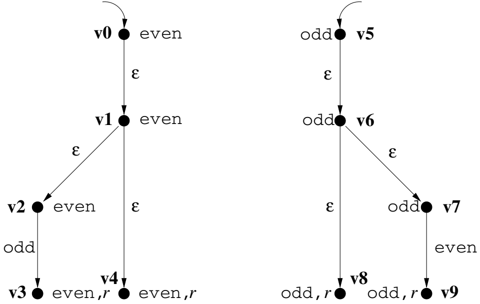
Flow Graph:

```
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);

   }
}
```



Flow Graph Behaviour

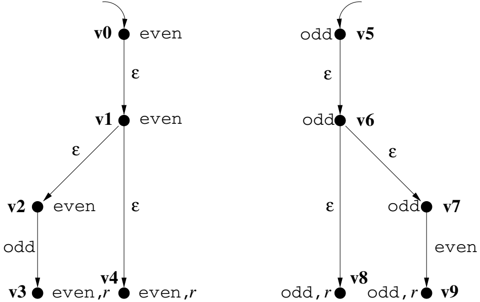
Flow Graph:

```
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);

   }
}
```



Example run through the behaviour, from an initial configuration:

(v_0, ε)

Flow Graph Behaviour

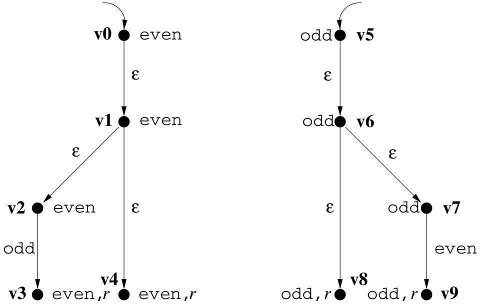
Flow Graph:

```
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);

   }
}
```



Example run through the behaviour, from an initial configuration:

$(v_0, \varepsilon) \xrightarrow{\tau} (v_1, \varepsilon)$

Flow Graph Behaviour

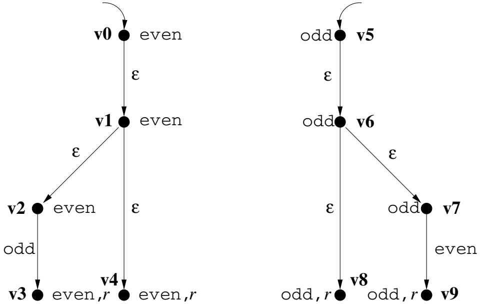
Flow Graph:

```
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);

   }
}
```



Example run through the behaviour, from an initial configuration:
$(v_0, \varepsilon) \xrightarrow{\tau} (v_1, \varepsilon) \xrightarrow{\tau} (v_2, \varepsilon)$

Flow Graph Behaviour

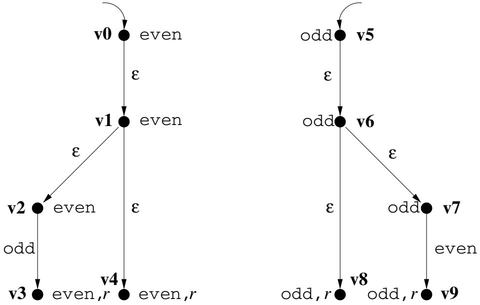
Flow Graph:

```
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);

   }
}
```



Example run through the behaviour, from an initial configuration:

$(v_0, \varepsilon) \xrightarrow{\tau} (v_1, \varepsilon) \xrightarrow{\tau} (v_2, \varepsilon) \xrightarrow{\text{even call odd}}$

(v_5, v_3)

Flow Graph Behaviour

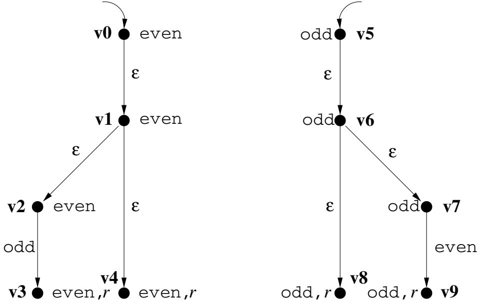
Flow Graph:

```
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);

   }
}
```



Example run through the behaviour, from an initial configuration:

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

Flow Graph Behaviour

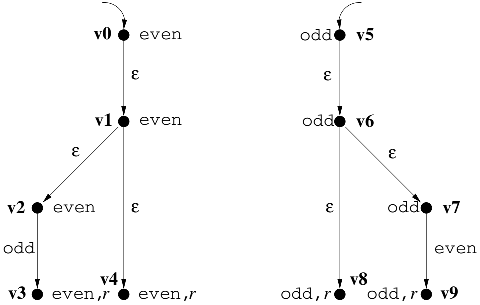
Flow Graph:

```
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);

   }
}
```



Example run through the behaviour, from an initial configuration:

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

Flow Graph Behaviour

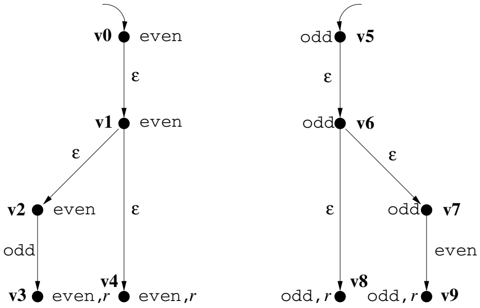
Flow Graph:

```
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);

   }
}
```



Example run through the behaviour, from an initial configuration:

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

Open Flow Graph Behaviour

How to treat external methods in open flow graphs?

Open Flow Graph Behaviour

How to treat external methods in open flow graphs?

One possibility is to treat calls to external methods as *atomic*

- ignores callback behaviour
- not relevant in a context–free setting (no data)

Open Flow Graph Behaviour

How to treat external methods in open flow graphs?

One possibility is to treat calls to external methods as *atomic*

- ignores callback behaviour
- not relevant in a context–free setting (no data)

Example run of method `even` as an open flow graph:

$$(v_0, \varepsilon) \xrightarrow{\tau} (v_1, \varepsilon) \xrightarrow{\tau} (v_2, \varepsilon) \xrightarrow{\text{even caret odd}} (v_3, \varepsilon)$$

Flow Graph Extraction from Java Bytecode

Flow Graph Extraction from Java Bytecode

Conceptually simple:

- labels become control points
- instructions define outgoing edges

Flow Graph Extraction from Java Bytecode

Conceptually simple:

- labels become control points
- instructions define outgoing edges

Complications: sound, precise, modular

- virtual method call resolution
- exceptional flow

Flow Graph Extraction from Java Bytecode

Java program:

```
public static void Meth(boolean flag, ExtA myobj) {
   try {
      if (flag) myobj.Meth();
   } catch (NullPointerException e) {}
}
```

Flow Graph Extraction from Java Bytecode

Java program:

```
public static void Meth(boolean flag, ExtA myobj) {
   try {
      if (flag) myobj.Meth();
   } catch (NullPointerException e) {}
}
```

Corresponding bytecode:

```
public static void Meth(boolean, ExtA);
Code:
 0:  iload_1
 1:  ifeq    8
 4:  aload_0
 5:  invokevirtual
 8:  goto    12
 11: astore_2
 12: return

Exception table:
 from   to  target type
   0     8    11   NullPointerException
```

Flow Graph Extraction from Java Bytecode

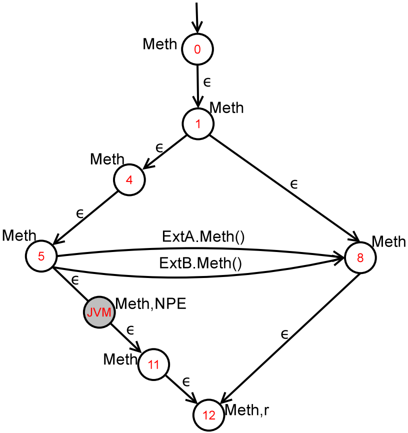
Java program:

```
public static void Meth(boolean flag, ExtA myobj) {
   try {
      if (flag) myobj.Meth();
   } catch (NullPointerException e) {}
}
```

Corresponding bytecode:

```
public static void Meth(boolean, ExtA);
Code:
 0:  iload_1
 1:  ifeq    8
 4:  aload_0
 5:  invokevirtual
 8:  goto    12
 11: astore_2
 12: return

Exception table:
 from    to  target type
   0      8    11   NullPointerException
```



Flow Graph Extraction from Java Bytecode

Correctness:

- stated in terms of simulation

Flow Graph Extraction from Java Bytecode

Correctness:

- stated in terms of simulation

Tool support:

- SAWJA: a framework for static analysis of Java bytecode
- developed at Inria Rennes, France
- uses a stackless intermediate representation of Java bytecode

Flow Graph Extraction from Java Bytecode

Correctness:

- stated in terms of simulation

Tool support:

- SAWJA: a framework for static analysis of Java bytecode
- developed at Inria Rennes, France
- uses a stackless intermediate representation of Java bytecode

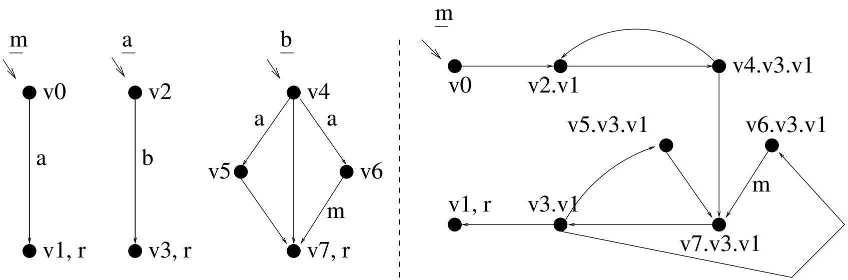
Details in:
Provably Correct Flow Graphs from Java Programs with Exceptions
Afshin Amighi, Pedro de Carvalho Gomez and Marieke Huisman
In Proceedings of FoVeOOS'11, pp. 31–48

Public Interface Abstraction

We can abstract from private methods through inlining:

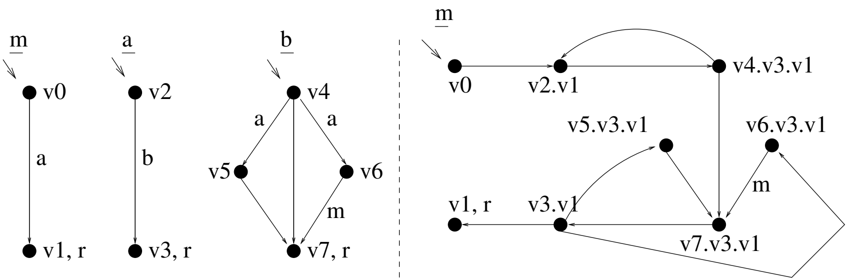
Public Interface Abstraction

We can abstract from private methods through inlining:



Public Interface Abstraction

We can abstract from private methods through inlining:



Details in:
Interface Abstraction for Compositional Verification
Dilian Gurov and Marieke Huisman
In Proceedings of SEFM'05, pp. 414–423

3. Property Specification and Verification

We instantiate both simulation and simulation logic to flow graphs and flow graph behaviour

3. Property Specification and Verification

We instantiate both simulation and simulation logic to flow graphs and flow graph behaviour

Example structural property:

- program is tail recursive:

$$\nu X.\ [\mathtt{even}]\, r \wedge [\mathtt{odd}]\, r \wedge [\epsilon]\, X$$

- can be checked with standard finite–state model checking

3. Property Specification and Verification

We instantiate both simulation and simulation logic to flow graphs and flow graph behaviour

Example structural property:

- program is tail recursive:

$$\nu X.\ [\mathtt{even}]\, r \wedge [\mathtt{odd}]\, r \wedge [\epsilon]\, X$$

- can be checked with standard finite–state model checking

Example behavioural property:

- first call of `even` is not to itself:

$$\mathtt{even} \Rightarrow \nu X.\ [\mathtt{even}\,\mathsf{call}\,\mathtt{even}]\, \mathsf{ff} \wedge [\tau]\, X$$

- can be checked with PDA model checking

More behavioural properties

More behavioural properties

- A security policy: "no send after read"

More behavioural properties

- A security policy: "no send after read"
  Interface: provided `a`, required `read`, `send`

More behavioural properties

- A security policy: "no send after read"
  Interface: provided `a`, required `read`, `send`
  Behavioural specification:
  $\phi = \nu X.\ [\tau]\, X \wedge [\text{a caret } \texttt{send}]\, X \wedge [\text{a call a}]\, X \wedge [\text{a ret a}]\, X \wedge [\text{a caret } \texttt{read}]\, \phi'$
  $\phi' = \nu Y.\ [\tau]\, Y \wedge [\text{a caret } \texttt{read}]\, Y \wedge [\text{a call a}]\, Y \wedge [\text{a ret a}]\, Y \wedge [\text{a caret } \texttt{send}]\, \text{ff}$

More behavioural properties

- A security policy: "no send after read"
  Interface: provided `a`, required `read`, `send`
  Behavioural specification:
  $\phi = \nu X.\ [\tau]\, X \wedge [\mathtt{a}\ \text{caret}\ \mathtt{send}]\, X \wedge [\mathtt{a}\ \text{call}\ \mathtt{a}]\, X \wedge [\mathtt{a}\ \text{ret}\ \mathtt{a}]\, X \wedge [\mathtt{a}\ \text{caret}\ \mathtt{read}]\, \phi'$
  $\phi' = \nu Y.\ [\tau]\, Y \wedge [\mathtt{a}\ \text{caret}\ \mathtt{read}]\, Y \wedge [\mathtt{a}\ \text{call}\ \mathtt{a}]\, Y \wedge [\mathtt{a}\ \text{ret}\ \mathtt{a}]\, Y \wedge [\mathtt{a}\ \text{caret}\ \mathtt{send}]\, \mathsf{ff}$

- "a vote only is submitted after validation"

More behavioural properties

- A security policy: "no send after read"
  Interface: provided $\mathtt{a}$, required $\mathtt{read}$, $\mathtt{send}$
  Behavioural specification:
  $\phi = \nu X.\ [\tau]\, X \wedge [\mathsf{a\ caret}\ \mathtt{send}]\, X \wedge [\mathsf{a\ call\ a}]\, X \wedge [\mathsf{a\ ret\ a}]\, X \wedge [\mathsf{a\ caret}\ \mathtt{read}]\, \phi'$
  $\phi' = \nu Y.\ [\tau]\, Y \wedge [\mathsf{a\ caret}\ \mathtt{read}]\, Y \wedge [\mathsf{a\ call\ a}]\, Y \wedge [\mathsf{a\ ret\ a}]\, Y \wedge [\mathsf{a\ caret}\ \mathtt{send}]\, \mathsf{ff}$

- "a vote only is submitted after validation"
- "votes are only counted after voting has finished"

More behavioural properties

- A security policy: "no send after read"
  Interface: provided `a`, required `read`, `send`
  Behavioural specification:
  $\phi = \nu X.\ [\tau]\, X \wedge [\text{a caret } \texttt{send}]\, X \wedge [\text{a call a}]\, X \wedge [\text{a ret a}]\, X \wedge [\text{a caret } \texttt{read}]\, \phi'$
  $\phi' = \nu Y.\ [\tau]\, Y \wedge [\text{a caret } \texttt{read}]\, Y \wedge [\text{a call a}]\, Y \wedge [\text{a ret a}]\, Y \wedge [\text{a caret } \texttt{send}]\, \mathrm{ff}$
- "a vote only is submitted after validation"
- "votes are only counted after voting has finished"
- "no non–atomic operations within transactions"

4. Maximal Flow Graphs

Given a structural property ϕ, is there a *maximal flow graph* for ϕ?

4. Maximal Flow Graphs

Given a structural property ϕ, is there a *maximal flow graph* for ϕ?

The maximal model of a structural property may not be a legal flow graph!

4. Maximal Flow Graphs

Given a structural property ϕ, is there a *maximal flow graph* for ϕ?

The maximal model of a structural property may not be a legal flow graph!

However, given an interface *I* we can *characterize* flow graphs with that interface — in structural simulation logic!

4. Maximal Flow Graphs

Given a structural property ϕ, is there a *maximal flow graph* for ϕ?

The maximal model of a structural property may not be a legal flow graph!

However, given an interface I we can *characterize* flow graphs with that interface — in structural simulation logic!

For example, for closed interface $I = \{a, b\}$ we have:

$$\theta_I = (\nu X.\ a \wedge [a, b, \epsilon]\, X) \vee (\nu Y.\ b \wedge [a, b, \epsilon]\, Y)$$

4. Maximal Flow Graphs

Given a structural property ϕ, is there a *maximal flow graph* for ϕ?

The maximal model of a structural property may not be a legal flow graph!

However, given an interface I we can *characterize* flow graphs with that interface — in structural simulation logic!

For example, for closed interface $I = \{a, b\}$ we have:

$$\theta_I = (\nu X.\ a \wedge [a, b, \epsilon]\, X) \vee (\nu Y.\ b \wedge [a, b, \epsilon]\, Y)$$

Then, the maximal flow graph for a structural formula ϕ and interface I is simply the maximal model for $\phi \wedge \theta_I$

Modular Verification for Structural Properties

Since structural simulation is monotone w.r.t. flow graph composition, we can thus support modular verification for structural properties!

Modular Verification for Structural Properties

Since structural simulation is monotone w.r.t. flow graph composition, we can thus support modular verification for structural properties!

Theorem

Structural simulation entails behavioural simulation

Modular Verification for Structural Properties

Since structural simulation is monotone w.r.t. flow graph composition, we can thus support modular verification for structural properties!

Theorem

Structural simulation entails behavioural simulation

Hence, we can even verify global behavioural properties with local structural specifications!

Modular Verification for Structural Properties

Since structural simulation is monotone w.r.t. flow graph composition, we can thus support modular verification for structural properties!

Theorem

Structural simulation entails behavioural simulation

Hence, we can even verify global behavioural properties with local structural specifications!

For instance, specify `even` and `odd` structurally, and verify the global behavioural specification:

$$\mathtt{even} \Rightarrow \nu X.\ [\mathtt{even}\ \mathsf{call}\ \mathtt{even}]\,\mathsf{ff} \wedge [\tau]\,X$$

Modular Verification: Example

Modular Verification: Example

Structural specification for even:

Interface: prov. even, req. odd

$\phi_{\texttt{even}} = \nu X.\ [\mathsf{even}]\,\mathsf{ff} \wedge [\mathsf{odd}]\,\phi'_{\texttt{even}} \wedge [\epsilon]\,X$
$\phi'_{\texttt{even}} = \nu Y.\ [\mathsf{even}]\,\mathsf{ff} \wedge [\mathsf{odd}]\,\mathsf{ff} \wedge [\epsilon]\,Y$

Modular Verification: Example

Structural specification for `even`:

Interface: prov. `even`, req. `odd`

$$\phi_{\texttt{even}} = \nu X.\ [\texttt{even}]\,\mathsf{ff} \wedge [\texttt{odd}]\,\phi'_{\texttt{even}} \wedge [\epsilon]\,X$$
$$\phi'_{\texttt{even}} = \nu Y.\ [\texttt{even}]\,\mathsf{ff} \wedge [\texttt{odd}]\,\mathsf{ff} \wedge [\epsilon]\,Y$$

Structural specification for `odd`:

Interface: prov. `odd`, req. `even`

$$\phi_{\texttt{odd}} = \nu X.\ [\texttt{odd}]\,\mathsf{ff} \wedge [\texttt{even}]\,\phi'_{\texttt{odd}} \wedge [\epsilon]\,X$$
$$\phi'_{\texttt{odd}} = \nu Y.\ [\texttt{odd}]\,\mathsf{ff} \wedge [\texttt{even}]\,\mathsf{ff} \wedge [\epsilon]\,Y$$

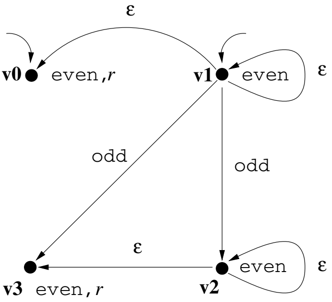
Modular Verification: Example

Structural specification for even:

Interface: prov. even, req. odd

$\phi_{\texttt{even}} = \nu X.\ [\texttt{even}]\,\mathrm{ff} \wedge [\texttt{odd}]\,\phi'_{\texttt{even}} \wedge [\epsilon]\,X$

$\phi'_{\texttt{even}} = \nu Y.\ [\texttt{even}]\,\mathrm{ff} \wedge [\texttt{odd}]\,\mathrm{ff} \wedge [\epsilon]\,Y$

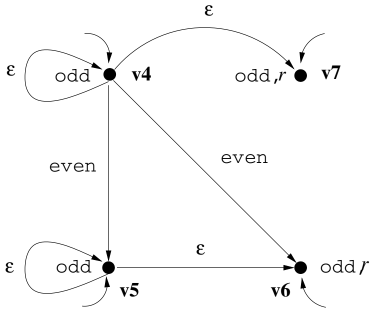


Structural specification for odd:

Interface: prov. odd, req. even

$\phi_{\texttt{odd}} = \nu X.\ [\texttt{odd}]\,\mathrm{ff} \wedge [\texttt{even}]\,\phi'_{\texttt{odd}} \wedge [\epsilon]\,X$

$\phi'_{\texttt{odd}} = \nu Y.\ [\texttt{odd}]\,\mathrm{ff} \wedge [\texttt{even}]\,\mathrm{ff} \wedge [\epsilon]\,Y$



Behavioural Properties

Given a behavioural property ϕ, is there a *maximal flow graph* for ϕ?

Behavioural Properties

Given a behavioural property ϕ, is there a *maximal flow graph* for ϕ?

The maximal model of a behavioural property is not a legal flow graph!

Behavioural Properties

Given a behavioural property ϕ, is there a *maximal flow graph* for ϕ?

The maximal model of a behavioural property is not a legal flow graph!

Several possibilities:

Behavioural Properties

Given a behavioural property ϕ, is there a *maximal flow graph* for ϕ?

The maximal model of a behavioural property is not a legal flow graph!

Several possibilities:

- use maximal models at the expense of completeness: false negatives

Behavioural Properties

Given a behavioural property ϕ, is there a *maximal flow graph* for ϕ?

The maximal model of a behavioural property is not a legal flow graph!

Several possibilities:

- use maximal models at the expense of completeness: false negatives
- translate behavioural properties to structural ones: expensive

Behavioural Properties

Given a behavioural property ϕ, is there a *maximal flow graph* for ϕ?

The maximal model of a behavioural property is not a legal flow graph!

Several possibilities:

- use maximal models at the expense of completeness: false negatives
- translate behavioural properties to structural ones: expensive
- restrict behavioural logic: atomic calls only: caret

Property Translation

Property Translation

Behavioural property "no send after read":

$$\begin{array}{rcl}\phi & = & \nu X.\ [\tau]\, X \wedge [\mathsf{a\ caret}\ \mathtt{send}]\, X \wedge [\mathsf{a\ call\ a}]\, X \wedge [\mathsf{a\ ret\ a}]\, X \wedge [\mathsf{a\ caret}\ \mathtt{read}]\, \phi' \\ \phi' & = & \nu Y.\ [\tau]\, Y \wedge [\mathsf{a\ caret}\ \mathtt{read}]\, Y \wedge [\mathsf{a\ call\ a}]\, Y \wedge [\mathsf{a\ ret\ a}]\, Y \wedge [\mathsf{a\ caret}\ \mathtt{send}]\, \mathsf{ff}\end{array}$$

Property Translation

Behavioural property "no send after read":

$$
\begin{array}{rcl}
\phi & = & \nu X.\ [\tau]\, X \wedge [\mathsf{a\ caret}\ \mathtt{send}]\, X \wedge [\mathsf{a\ call\ a}]\, X \wedge [\mathsf{a\ ret\ a}]\, X \wedge [\mathsf{a\ caret}\ \mathtt{read}]\, \phi' \\
\phi' & = & \nu Y.\ [\tau]\, Y \wedge [\mathsf{a\ caret}\ \mathtt{read}]\, Y \wedge [\mathsf{a\ call\ a}]\, Y \wedge [\mathsf{a\ ret\ a}]\, Y \wedge [\mathsf{a\ caret}\ \mathtt{send}]\, \mathsf{ff}
\end{array}
$$

gives rise to several structural properties, most notably:

$$
\begin{array}{rcl}
\psi & = & \nu X.\ [\epsilon]\, X \wedge [\mathtt{send}]\, X \wedge [\mathsf{a}]\, \psi' \wedge [\mathtt{read}]\, \psi' \\
\psi' & = & \nu Y.\ [\epsilon]\, Y \wedge [\mathtt{read}]\, Y \wedge [\mathsf{a}]\, \mathsf{ff} \wedge [\mathtt{send}]\, \mathsf{ff}
\end{array}
$$

Property Translation

Behavioural property "no send after read":

$$
\begin{aligned}
\phi &= \nu X.\ [\tau]\, X \wedge [\text{a caret } \mathtt{send}]\, X \wedge [\text{a call a}]\, X \wedge [\text{a ret a}]\, X \wedge [\text{a caret } \mathtt{read}]\, \phi' \\
\phi' &= \nu Y.\ [\tau]\, Y \wedge [\text{a caret } \mathtt{read}]\, Y \wedge [\text{a call a}]\, Y \wedge [\text{a ret a}]\, Y \wedge [\text{a caret } \mathtt{send}]\, \text{ff}
\end{aligned}
$$

gives rise to several structural properties, most notably:

$$
\begin{aligned}
\psi &= \nu X.\ [\epsilon]\, X \wedge [\mathtt{send}]\, X \wedge [\text{a}]\, \psi' \wedge [\mathtt{read}]\, \psi' \\
\psi' &= \nu Y.\ [\epsilon]\, Y \wedge [\mathtt{read}]\, Y \wedge [\text{a}]\, \text{ff} \wedge [\mathtt{send}]\, \text{ff}
\end{aligned}
$$

Details in:

Reducing Behavioural to Structural Properties

Dilian Gurov and Marieke Huisman

In Proceedings of VMCAI'09, pp. 136–150

Restricted Behavioural Logic: Atomic Calls

Behavioural specification of `even`:

$$
\begin{array}{rcl}
\phi_{\texttt{even}} & = & \nu X.\ [\texttt{even}\ \mathsf{caret}\ \texttt{even}]\,\mathsf{ff} \wedge [\texttt{even}\ \mathsf{caret}\ \texttt{odd}]\,\phi'_{\texttt{even}} \wedge [\tau]\,X \\
\phi'_{\texttt{even}} & = & \nu Y.\ [\texttt{even}\ \mathsf{caret}\ \texttt{even}]\,\mathsf{ff} \wedge [\texttt{even}\ \mathsf{caret}\ \texttt{odd}]\,\mathsf{ff} \wedge [\tau]\,Y
\end{array}
$$

Restricted Behavioural Logic: Atomic Calls

Behavioural specification of even:

$$
\begin{array}{rcl}
\phi_{\texttt{even}} & = & \nu X.\ [\text{even caret even}]\,\text{ff} \wedge [\text{even caret odd}]\,\phi'_{\texttt{even}} \wedge [\tau]\,X \\
\phi'_{\texttt{even}} & = & \nu Y.\ [\text{even caret even}]\,\text{ff} \wedge [\text{even caret odd}]\,\text{ff} \wedge [\tau]\,Y
\end{array}
$$

gives rise to a *single* structural property:

$$
\begin{array}{rcl}
\phi_{\texttt{even}} & = & \nu X.\ [\text{even}]\,\text{ff} \wedge [\text{odd}]\,\phi'_{\texttt{even}} \wedge [\epsilon]\,X \\
\phi'_{\texttt{even}} & = & \nu Y.\ [\text{even}]\,\text{ff} \wedge [\text{odd}]\,\text{ff} \wedge [\epsilon]\,Y
\end{array}
$$

Restricted Behavioural Logic: Atomic Calls

Behavioural specification of even:

$$\begin{array}{rcl} \phi_{\texttt{even}} & = & \nu X.\ [\text{even caret even}]\,\text{ff} \wedge [\text{even caret odd}]\,\phi'_{\texttt{even}} \wedge [\tau]\,X \\ \phi'_{\texttt{even}} & = & \nu Y.\ [\text{even caret even}]\,\text{ff} \wedge [\text{even caret odd}]\,\text{ff} \wedge [\tau]\,Y \end{array}$$

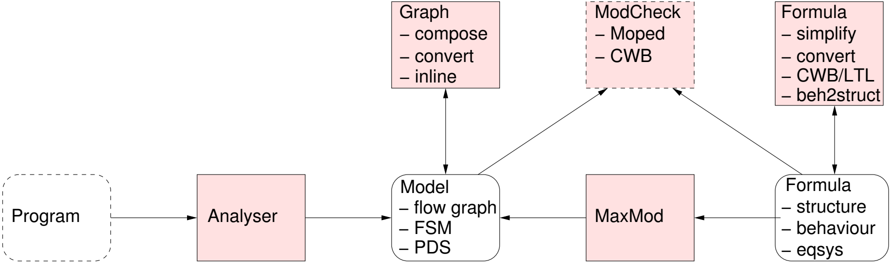
gives rise to a *single* structural property:

$$\begin{array}{rcl} \phi_{\texttt{even}} & = & \nu X.\ [\text{even}]\,\text{ff} \wedge [\text{odd}]\,\phi'_{\texttt{even}} \wedge [\epsilon]\,X \\ \phi'_{\texttt{even}} & = & \nu Y.\ [\text{even}]\,\text{ff} \wedge [\text{odd}]\,\text{ff} \wedge [\epsilon]\,Y \end{array}$$

obtained through a *direct* translation!

5. Tool Support

The CVPP Tool Set



Automation

Full automation would require:

Automation

Full automation would require:

- single input to the checker
- local and global specs as annotations/comments
- inspired from JML based verification tools like ESC/Java
- pre– and post–processing

Automation

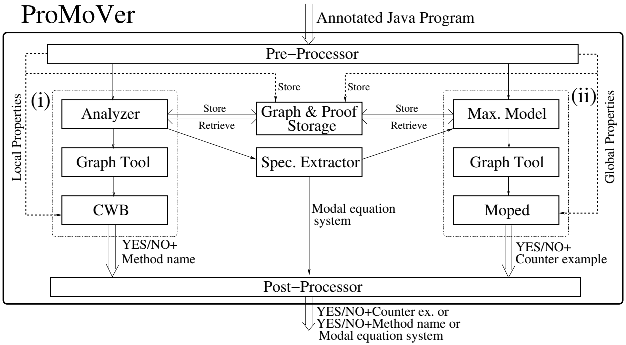
Full automation would require:

- single input to the checker
- local and global specs as annotations/comments
- inspired from JML based verification tools like ESC/Java
- pre– and post–processing

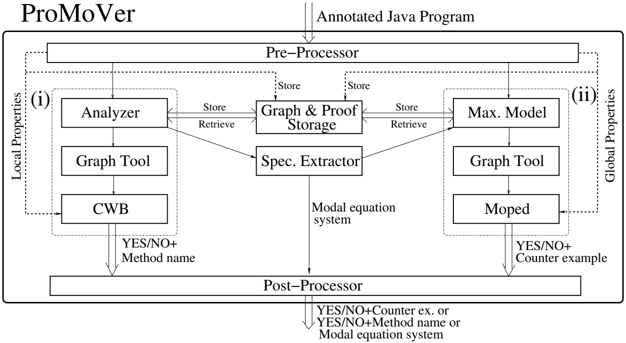
```
/** @global_LTL_prop:
 *    even -> X ((even && !entry) W odd)
 */
public class EvenOdd {
   /** @local_interface: requires {odd}
    *
    *  @local_SL_prop:
    *   nu X1. (([even call even]ff) /\ ([tau]X1) /\
    *     [even caret odd] nu X2.
    *       (([even call even]ff) /\
    *        ([even 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_SL_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);
   }
}
```

ProMoVer: A wrapper around CVPP



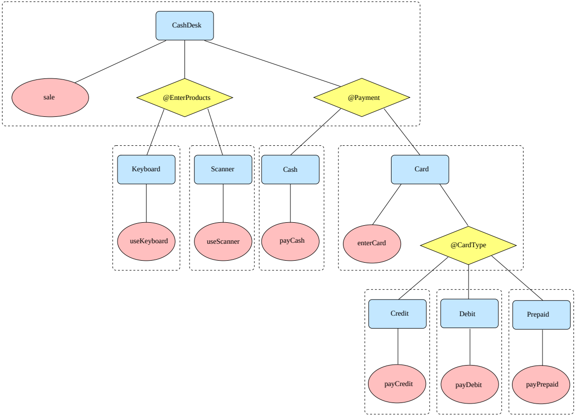
ProMoVer: A wrapper around CVPP



Details in:
ProMoVer: Modular Verification of Temporal Safety Properties
Siavash Soleimanifard, Dilian Gurov and Marieke Huisman
In Proceedings of SEFM'11, pp. 366–381

6. Application: Software Product Lines

A hierarchical variability model for software product lines:



Software Product Lines Verification

The number of products can be exponential in the size (number of regions) of the variability model! Needs compositional treatment!

Software Product Lines Verification

The number of products can be exponential in the size (number of regions) of the variability model! Needs compositional treatment!

Solution: relativize on properties of variation points!

Software Product Lines Verification

The number of products can be exponential in the size (number of regions) of the variability model! Needs compositional treatment!

Solution: relativize on properties of variation points!

Results in one verification task per region!

Software Product Lines Verification

The number of products can be exponential in the size (number of regions) of the variability model! Needs compositional treatment!

Solution: relativize on properties of variation points!

Results in one verification task per region!

Details in:
Compositional Algorithmic Verification of Software Product Lines
Ina Schaefer, Dilian Gurov and Siavash Soleimanifard
In Post–proceedings of: FMCO'10, pp. 184–203

7. Conclusion

7. Conclusion

Strengths:

- algorithmic verification of temporal safety properties
- modular: allows dealing with variability
- sound and complete at flow graph level
- tools and wrappers for various scenarios

7. Conclusion

Strengths:

- algorithmic verification of temporal safety properties
- modular: allows dealing with variability
- sound and complete at flow graph level
- tools and wrappers for various scenarios

Limitations:

- limited properties: no data
- computationally expensive:

7. Conclusion

Strengths:

- algorithmic verification of temporal safety properties
- modular: allows dealing with variability
- sound and complete at flow graph level
- tools and wrappers for various scenarios

Limitations:

- limited properties: no data
- computationally expensive:
  - flow graph extraction

7. Conclusion

Strengths:

- algorithmic verification of temporal safety properties
- modular: allows dealing with variability
- sound and complete at flow graph level
- tools and wrappers for various scenarios

Limitations:

- limited properties: no data
- computationally expensive:
  - flow graph extraction
  - maximal flow graph construction

7. Conclusion

Strengths:

- algorithmic verification of temporal safety properties
- modular: allows dealing with variability
- sound and complete at flow graph level
- tools and wrappers for various scenarios

Limitations:

- limited properties: no data
- computationally expensive:
  - flow graph extraction
  - maximal flow graph construction
  - PDA model checking

7. Conclusion

Strengths:

- algorithmic verification of temporal safety properties
- modular: allows dealing with variability
- sound and complete at flow graph level
- tools and wrappers for various scenarios

Limitations:

- limited properties: no data
- computationally expensive:
  - flow graph extraction
  - maximal flow graph construction
  - PDA model checking
  - property translation and simplification

Future Work

- take pragmatic approaches to deal with bottlenecks:

Future Work

- take pragmatic approaches to deal with bottlenecks:
    - flow graph extraction: sacrifice precision

Future Work

- take pragmatic approaches to deal with bottlenecks:
  - flow graph extraction: sacrifice precision
  - maximal flow graph construction: avoid when possible

Future Work

- take pragmatic approaches to deal with bottlenecks:
  - flow graph extraction: sacrifice precision
  - maximal flow graph construction: avoid when possible
  - PDA model checking: use FSM model checking instead

Future Work

- take pragmatic approaches to deal with bottlenecks:
  - flow graph extraction: sacrifice precision
  - maximal flow graph construction: avoid when possible
  - PDA model checking: use FSM model checking instead
  - property translation and simplification: restrict logics

Future Work

- take pragmatic approaches to deal with bottlenecks:
  - flow graph extraction: sacrifice precision
  - maximal flow graph construction: avoid when possible
  - PDA model checking: use FSM model checking instead
  - property translation and simplification: restrict logics
- add data in a controlled way:

Future Work

- take pragmatic approaches to deal with bottlenecks:
  - flow graph extraction: sacrifice precision
  - maximal flow graph construction: avoid when possible
  - PDA model checking: use FSM model checking instead
  - property translation and simplification: restrict logics
- add data in a controlled way:
  - Boolean data

Future Work

- take pragmatic approaches to deal with bottlenecks:
  - flow graph extraction: sacrifice precision
  - maximal flow graph construction: avoid when possible
  - PDA model checking: use FSM model checking instead
  - property translation and simplification: restrict logics
- add data in a controlled way:
  - Boolean data
  - references