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 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 breaking it down into manageable pieces that are designed, implemented, tested and maintained independently
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 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.
Verification

Verification as a *systems design task*:

- match a model of the system against a specification
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 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

Temporal variability: static code evolution, dynamic code replacement, dynamic code loading: code not available at verification time

Spatial variability: multiple variants, as arising from software product lines
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 variability:
- 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
Consider a system with four modules (components):

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

Verification in the presence of variability
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

How shall one plan for the verification of a global property $\psi$ as early as possible with minimal effort: reuse results
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 $\psi$?
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 $\psi$?
- 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 \( \psi \)?
- as early as possible
- with minimal effort: reuse results
Relativization

1. Specify modules B, C, D
2. Verify $\text{impl}(B) \cup \text{spec}(B) = \text{impl}(C) \cup \text{spec}(C) = \text{impl}(D) \cup \text{spec}(D)$
3. Verify $\text{impl}(A) + \text{spec}(B) + \text{spec}(C) + \text{spec}(D) \cup \psi$

But... how, and is there an algorithmic solution?
Relativize global property on local specifications. Three tasks:

1. Specify modules $B$, $C$, $D$
2. Verify $\text{impl}(B) \mid = \text{spec}(B)$
3. $\text{impl}(C) \mid = \text{spec}(C)$
4. $\text{impl}(D) \mid = \text{spec}(D)$

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

Relativize global property on local specifications. Three tasks:

1. specify modules B, C, D

2. verify \( \text{impl}(B) = \text{spec}(B) \) \( \text{impl}(C) = \text{spec}(C) \) \( \text{impl}(D) = \text{spec}(D) \)

3. verify \( \text{impl}(A) + \text{spec}(B) + \text{spec}(C) + \text{spec}(D) = \psi \)

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

Relativize global property on local specifications. Three tasks:

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

\[
\begin{align*}
\operatorname{impl}(B) & \models \operatorname{spec}(B) \\
\operatorname{impl}(C) & \models \operatorname{spec}(C) \\
\operatorname{impl}(D) & \models \operatorname{spec}(D)
\end{align*}
\]
Relativization

Relativize global property on local specifications. Three tasks:

1. Specify modules B, C, D
2. Verify

\[ \text{impl}(B) \models \text{spec}(B) \]
\[ \text{impl}(C) \models \text{spec}(C) \]
\[ \text{impl}(D) \models \text{spec}(D) \]

3. Verify

\[ \text{impl}(A) + \text{spec}(B) + \text{spec}(C) + \text{spec}(D) \models \psi \]
Relativization

Relativize global property on local specifications. Three tasks:

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

\[
\text{impl}(B) \models \text{spec}(B) \\
\text{impl}(C) \models \text{spec}(C) \\
\text{impl}(D) \models \text{spec}(D)
\]

3. verify

\[
\text{impl}(A) + \text{spec}(B) + \text{spec}(C) + \text{spec}(D) \models \psi
\]

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:

\[
\begin{align*}
\text{impl}(B) & \models \text{spec}(B) \\
\text{impl}(C) & \models \text{spec}(C) \\
\text{impl}(D) & \models \text{spec}(D)
\end{align*}
\]

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:

\[
\begin{align*}
impl(B) & \models spec(B) \\
impl(C) & \models spec(C) \\
impl(D) & \models spec(D)
\end{align*}
\]

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:

\[
\text{impl}(B) \models \text{spec}(B)
\]

\[
\text{impl}(C) \models \text{spec}(C)
\]

\[
\text{impl}(D) \models \text{spec}(D)
\]

perform the following steps:

1. from module implementations: extract models

2. model check models against local specifications:

\[
\text{mod(impl}(B)) \models \text{spec}(B)
\]

\[
\text{mod(impl}(C)) \models \text{spec}(C)
\]

\[
\text{mod(impl}(D)) \models \text{spec}(D)
\]
Program Models

For the third task:

\[ \text{impl}(A) + \text{spec}(B) + \text{spec}(C) + \text{spec}(D) = \psi \]

perform the following steps:
Program Models

For the third task:

\[ \text{impl}(A) + \text{spec}(B) + \text{spec}(C) + \text{spec}(D) \models \psi \]

perform the following steps:

1. from module implementations: extract models
Program Models

For the third task:

\[ \text{impl}(A) + \text{spec}(B) + \text{spec}(C) + \text{spec}(D) \models \psi \]

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:

\[ \text{impl}(A) + \text{spec}(B) + \text{spec}(C) + \text{spec}(D) \models \psi \]

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:

\[
\text{impl}(A) + \text{spec}(B) + \text{spec}(C) + \text{spec}(D) \models \psi
\]

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 \( \psi \):
\[
\text{mod(impl}(A)) + \text{max(spec}(B)) + \text{max(spec}(C)) + \text{max(spec}(D)) \models \psi
\]
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:

\[ \text{mod} (\text{impl}(A)) + \max (\text{spec}(B)) + \max (\text{spec}(C)) + \max (\text{spec}(D)) \mid = \psi \]
Thus entails:

\[ \text{impl}(A) + \text{impl}(B) + \text{impl}(C) + \text{impl}(D) \mid = \psi \]
Simulation: A refinement pre–order on models

We require the following conditions:

\[ \text{impl}(A) + \max(\text{spec}(B)) + \max(\text{spec}(C)) + \max(\text{spec}(D)) \leq \psi \]

thus entails:

\[ \text{impl}(A) + \text{impl}(B) + \text{impl}(C) + \text{impl}(D) \leq \psi \]
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:

\[ \text{mod(impl(A)) + max(spec(B)) + max(spec(C)) + max(spec(D)) = } \psi \]

thus entails:

\[ \text{impl(A) + impl(B) + impl(C) + impl(D) = } \psi \]
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

Dilian Gurov (KTH) Modular Verification of Temporal Safety Prop 15 November 2011 12 / 41
Our Setup

Program model: Flow graphs capturing purely control flow
  - behaviour as induced pushdown automaton

Most details in:
  - Compositional Verification of Sequential Programs with Procedures
    Dilian Gurov, Marieke Huisman and Christoph Sprenger
    Journal of Information and Computation
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

Most details in:
Dilian Gurov, Marieke Huisman and Christoph Sprenger
Journal of Information and Computation
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
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
Definition (Model)

A structure $M = (S, L, \rightarrow, A, \lambda)$ 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) $\lambda : S \rightarrow P(A)$ a valuation

An initialised model $(M, E)$ is a model $M$ with a designated set of entry states $E \subseteq S$. 

Dilian Gurov (KTH)  
Modular Verification of Temporal Safety Prop  
15 November 2011  
14 / 41
1. Models, Simulation, Logic

A model $\mathcal{M}$

- States: $s_1, s_2, s_3$
- Transition labels: $p, q, a, b$
- Transition relation: $\rightarrow \subseteq S \times L \times S$
- Atomic propositions: $A$
- Valuation: $\lambda : S \rightarrow P(A)$

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

**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 $E \subseteq S$
Let $M_1 = (S_1, L, \rightarrow_1, A, \lambda_1)$ and $M_2 = (S_2, L, \rightarrow_2, A, \lambda_2)$ 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 $(s_1, s_2) \in R$
  1. $\lambda_1(s_1) = \lambda_2(s_2)$
  2. for any $a \in L$ and $s'_1 \in S_1$
  $$s_1 \overset{a}{\rightarrow}_1 s'_1$$ entails
  $$s_2 \overset{a}{\rightarrow}_2 s'_2$$ 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 $(s_1, s_2) \in R$
  1. $\lambda_1(s_1) = \lambda_2(s_2)$
  2. for any $a \in L$ and $s'_1 \in S_1$
     - $s_1 \xrightarrow{a} s'_1$ entails $s_2 \xrightarrow{a} s'_2$ 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

Definition (Simulation Logic)
The formulas of the logic are inductively defined through the BNF:

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

where \( p \in A \) and \( a \in L \)

Example

Some example formulas and their meaning:

\[ [a] \text{ff} \]
\[ [a] \text{ff} \land [b] \text{ff} \]
\[ [a] \text{ff} \lor [b] \text{ff} \]
\[ \nu X. p \land [a] \text{ff} \land [b] \text{ff} \]
Logic

Definition (Simulation Logic)

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

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

where \( p \in A \) and \( a \in L \)
Logic

Definition (Simulation Logic)

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

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

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:

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

where \( p \in A \) and \( a \in L \)

Example

Some example formulas and their meaning:

- \([a] \text{ff}\)
Logic

Definition (Simulation Logic)

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

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

where \( p \in A \) and \( a \in L \)

Example

Some example formulas and their meaning:

- \([a] \text{ff}\)
- \([a] \text{ff} \land [b] \text{ff}\)
Logic

Definition (Simulation Logic)

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

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

where \( p \in A \) and \( a \in L \)

Example

Some example formulas and their meaning:

- \([a] \text{ ff}\)
- \([a] \text{ ff} \land [b] \text{ ff}\)
- \([a] \text{ ff} \lor [b] \text{ ff}\)
Logic

Definition (Simulation Logic)

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

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

where \( p \in A \) and \( a \in L \)

Example

Some example formulas and their meaning:

- \([a] \sf{ff}\)
- \([a] \sf{ff} \land [b] \sf{ff}\)
- \([a] \sf{ff} \lor [b] \sf{ff}\)
- \(\nu X . \ p \land [a] \sf{ff} \land [b] X\)
Maximal Models

**Definition (Maximal Model)**
A maximal model for a formula $\phi$ is an initialized model $S$ such that:

(i) $S$ satisfies $\phi$

(ii) $S$ simulates all initialized models satisfying $\phi$

**Theorem**
Every simulation logic formula $\phi$ has a maximal model $S_{\phi}$

**Corollary**
Maximal models are unique up to simulation equivalence
Maximal Models

**Definition (Maximal Model)**

A maximal model for a formula $\phi$ is an initialized model $S$ such that:

(i) $S$ satisfies $\phi$

(ii) $S$ simulates all initialized models satisfying $\phi$
Maximal Models

Definition (Maximal Model)
A maximal model for a formula $\phi$ is an initialized model $S$ such that:

(i) $S$ satisfies $\phi$

(ii) $S$ simulates all initialized models satisfying $\phi$

Theorem
Every simulation logic formula $\phi$ has a maximal model $S_\phi$
Definition (Maximal Model)
A maximal model for a formula $\phi$ is an initialized model $S$ such that:

(i) $S$ satisfies $\phi$

(ii) $S$ simulates all initialized models satisfying $\phi$

Theorem
Every simulation logic formula $\phi$ has a maximal model $S_\phi$

Corollary
Maximal models are unique up to simulation equivalence
Constructing Maximal Models

Labels \( \{a, b\} \), atoms \( \{p\} \), formula \([b] \text{ff} \land p\)
Constructing Maximal Models

Labels \{a, b\}, atoms \{p\}, formula \([b] \text{ff} \land p\)

The formula as an *equation system*:

\[ X = [b] \text{ff} \land p \]
Constructing Maximal Models

Labels \{a, b\}, atoms \{p\}, formula \([b] \text{ff} \land p\)

The formula as an equation system:

\[ X = [b] \text{ff} \land p \]

convert into simulation normal form:

\[ X = [a](Y_1 \lor Y_2) \land [b] \text{ff} \land p \]
\[ Y_1 = [a](Y_1 \lor Y_2) \land [b](Y_1 \lor Y_2) \land p \]
\[ Y_2 = [a](Y_1 \lor Y_2) \land [b](Y_1 \lor Y_2) \land \neg p \]
Constructing Maximal Models

Labels \{a, b\}, atoms \{p\}, formula \([b] \text{ff} \land p\)

The formula as an equation system:

\[
X = [b] \text{ff} \land p
\]

convert into simulation normal form:

\[
X = [a] (Y_1 \lor Y_2) \land [b] \text{ff} \land p
\]

\[
Y_1 = [a] (Y_1 \lor Y_2) \land [b] (Y_1 \lor Y_2) \land p
\]

\[
Y_2 = [a] (Y_1 \lor Y_2) \land [b] (Y_1 \lor Y_2) \land \neg p
\]

\[
(M, E)
\]
2. Flow Graphs, Interfaces and Behaviour

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

```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);
    }
}
```

Interfaces: provided and required methods
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 \((v, \sigma)\) 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
A flow graph induces a *pushdown automaton* (PDA):

- configurations \((v, \sigma)\) 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);
    }
}


Example run through the behaviour, from an initial configuration: 

\((v_0, \varepsilon) \rightarrow \tau \rightarrow (v_1, \varepsilon) \rightarrow \text{even call} \rightarrow (v_5, v_3) \rightarrow \tau \rightarrow (v_6, v_3) \rightarrow \text{odd ret} \rightarrow (v_3, \varepsilon)\)
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: $(v_0, \varepsilon)$
Flow Graph Behaviour

Flow Graph:

```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);
    }
}
```

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:

```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);
    }
}
```

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:

```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);
    }
}
```

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?

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

\[(v_0, \varepsilon) \tau^- \rightarrow (v_1, \varepsilon) \tau^- \rightarrow (v_2, \varepsilon) \tau^- \rightarrow (v_3, \varepsilon)\]
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

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

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
Java program:

```java
public static void Meth(boolean flag, ExtA myobj) {
    try {
        if (flag) myobj.Meth();
    } catch (NullPointerException e) {}  // Add a docstring for this catch block
}
```

Corresponding bytecode:
```
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:

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

Corresponding bytecode:

```java
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:

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
<th>target type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>NullPointerException</td>
</tr>
</tbody>
</table>
Java program:

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

Corresponding bytecode:

```java
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:

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

Dilian Gurov (KTH)
Modular Verification of Temporal Safety Prop
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:

  \[ \nu X. \ [\text{even}] \ r \land [\text{odd}] \ r \land [\epsilon] \ X \]

- 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:

$$\nu X. \lbrack \text{even} \rbrack \, r \land \lbrack \text{odd} \rbrack \, r \land \lbrack \epsilon \rbrack \, X$$

- can be checked with standard finite–state model checking

Example behavioural property:

- first call of even is not to itself:

$$\text{even} \Rightarrow \nu X. \lbrack \text{even call even}\rbrack \, ff \land [\tau] \, X$$

- can be checked with PDA model checking
More behavioural properties

A security policy: "no send after read"

Interface: provided

\( \phi = \nu X [\tau] X \land [a \text{ caret} \text{ send}] X \land [a \text{ call} a] X \land [a \text{ ret} a] X \land [a \text{ caret} \text{ read}] \phi' \)

\( \phi' = \nu Y [\tau] Y \land [a \text{ caret} \text{ read}] Y \land [a \text{ call} a] Y \land [a \text{ ret} a] Y \land [a \text{ caret} \text{ send}] \ff \)

"a vote only is submitted after validation"

"votes are only counted after voting has finished"

"no non–atomic operations within transactions"
More behavioural properties

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

φ = ν X [τ] X ∧ a caret send X ∧ a call X ∧ a ret a
φ′ = ν Y [τ] Y ∧ a caret read Y ∧ a call Y ∧ a ret Y ∧ a caret send Y
More behavioural properties

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

- "a vote only is submitted after validation"
- "votes are only counted after voting has finished"
- "no non–atomic operations within transactions"
More behavioural properties

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

- "a vote only is submitted after validation"
More behavioural properties

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

- "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:
  \[
  \phi = \nu X. \ [\tau] X \land [\text{caret send}] X \land [\text{call a}] X \land [\text{ret a}] X \land [\text{caret read}] \phi' \\
  \phi' = \nu Y. \ [\tau] Y \land [\text{caret read}] Y \land [\text{call a}] Y \land [\text{ret a}] Y \land [\text{caret send}] \text{ff}
  \]

- "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 $\phi$, is there a maximal flow graph for $\phi$?
4. Maximal Flow Graphs

Given a structural property $\phi$, is there a \textit{maximal flow graph} for $\phi$?

The maximal model of a structural property may not be a legal flow graph!
4. Maximal Flow Graphs

Given a structural property $\phi$, is there a maximal flow graph for $\phi$?

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 $\phi$, is there a maximal flow graph for $\phi$?

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:

$$\theta_I = (\nu X. \ a \land [a, b, \epsilon] X) \lor (\nu Y. \ b \land [a, b, \epsilon] Y)$$
Given a structural property $\phi$, is there a maximal flow graph for $\phi$?

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:

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

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

Theorem

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

For instance, specify

$$\text{even} \implies \nu X . [ \text{even} \text{call} \text{even} ] \text{ff} \land [ \tau ] X D$$
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:

\[
\text{even} \Rightarrow \nu X. \ [\text{even call even}] \text{ff} \land [\tau] X
\]
Modular Verification: Example

Structural specification for even:
Interface: prov. even, req. odd
\[\text{even}\phi = \nu X \land \text{even}ff \land \text{odd}phi' \land \epsilon X \phi' = \nu Y \ land \text{even}ff \land \text{odd}ff \land \epsilon Y \phi' = \nu Y \]

Structural specification for odd:
Interface: prov. odd, req. even
\[\text{odd}\phi = \nu X \land \text{odd}ff \land \text{even}phi' \land \epsilon X \phi' = \nu Y \ land \text{odd}ff \land \text{even}ff \land \epsilon Y \phi' = \nu Y \]
Modular Verification: Example

Structural specification for even:

Interface: prov. even, req. odd

\[ \phi_{even} = \nu X. \ [even] \text{ff} \land [odd] \phi'_{even} \land [\epsilon] X \]
\[ \phi'_{even} = \nu Y. \ [even] \text{ff} \land [odd] \text{ff} \land [\epsilon] Y \]
Modular Verification: Example

Structural specification for even:

Interface: prov. even, req. odd

$$\phi_{even} = \nu X. \ [even] \text{ff} \land \ [odd] \phi'_{even} \land \ [\epsilon] X$$

$$\phi'_{even} = \nu Y. \ [even] \text{ff} \land \ [odd] \text{ff} \land \ [\epsilon] Y$$

Structural specification for odd:

Interface: prov. odd, req. even

$$\phi_{odd} = \nu X. \ [odd] \text{ff} \land \ [even] \phi'_{odd} \land \ [\epsilon] X$$

$$\phi'_{odd} = \nu Y. \ [odd] \text{ff} \land \ [even] \text{ff} \land \ [\epsilon] Y$$
Modular Verification: Example

Structural specification for even:

Interface: prov. even, req. odd

$\phi_{\text{even}} = \nu X. \ [\text{even}] \text{ff} \land [\text{odd}] \phi'_{\text{even}} \land [\epsilon] X$

$\phi'_{\text{even}} = \nu Y. \ [\text{even}] \text{ff} \land [\text{odd}] \text{ff} \land [\epsilon] Y$

Structural specification for odd:

Interface: prov. odd, req. even

$\phi_{\text{odd}} = \nu X. \ [\text{odd}] \text{ff} \land [\text{even}] \phi'_{\text{odd}} \land [\epsilon] X$

$\phi'_{\text{odd}} = \nu Y. \ [\text{odd}] \text{ff} \land [\text{even}] \text{ff} \land [\epsilon] Y$
Given a behavioural property $\phi$, is there a *maximal flow graph* for $\phi$?
Given a behavioural property \( \phi \), is there a \textit{maximal flow graph} for \( \phi \)?

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

Given a behavioural property $\phi$, is there a *maximal flow graph* for $\phi$?

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

Several possibilities:
Given a behavioural property $\phi$, is there a *maximal flow graph* for $\phi$?

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 $\phi$, is there a *maximal flow graph* for $\phi$?

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
Given a behavioural property $\phi$, is there a maximal flow graph for $\phi$?

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

\[ \phi = \nu X \left[ \tau \right] X \land \left[ \text{a caret} \text{ send} \right] X \land \left[ \text{a call} \right] X \land \left[ \text{a ret} \right] X \land \left[ \text{a caret} \text{ read} \right] \]

\[ \phi' = \nu Y \left[ \tau \right] Y \land \left[ \text{a caret} \text{ read} \right] Y \land \left[ \text{a call} \right] Y \land \left[ \text{a ret} \right] Y \land \left[ \text{a caret} \text{ send} \right] \]

Gives rise to several structural properties, most notably:

\[ \psi = \nu X \left[ \epsilon \right] X \land \left[ \text{send} \right] X \land \left[ \text{a } \psi' \right] \land \left[ \text{read} \psi' \right] \]

\[ \psi' = \nu Y \left[ \epsilon \right] Y \land \left[ \text{read} \right] Y \land \left[ \text{a } \right] Y \land \left[ \text{send} \right] \]

Details in: 

Reducing Behavioural to Structural Properties

Dilian Gurov and Marieke Huisman

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

Behavioural property "no send after read":

\[ \phi = \nu X. \quad [\tau] X \land [\text{a caret send}] X \land [\text{a call a}] X \land [\text{a ret a}] X \land [\text{a caret read}] \phi' \]

\[ \phi' = \nu Y. \quad [\tau] Y \land [\text{a caret read}] Y \land [\text{a call a}] Y \land [\text{a ret a}] Y \land [\text{a caret send}] \text{ff} \]
Property Translation

Behavioural property ”no send after read”:

\[
\begin{align*}
\phi &= \nu X. [\tau] X \land [\text{a caret send}] X \land [\text{a call a}] X \land [\text{a ret a}] X \land [\text{a caret read}] \phi' \\
\phi' &= \nu Y. [\tau] Y \land [\text{a caret read}] Y \land [\text{a call a}] Y \land [\text{a ret a}] Y \land [\text{a caret send}] ff
\end{align*}
\]

gives rise to several structural properties, most notably:

\[
\begin{align*}
\psi &= \nu X. [\epsilon] X \land [\text{send}] X \land [\text{a}] \psi' \land [\text{read}] \psi' \\
\psi' &= \nu Y. [\epsilon] Y \land [\text{read}] Y \land [\text{a}] ff \land [\text{send}] ff
\end{align*}
\]
Property Translation

Behavioural property "no send after read":

\[ \phi = \nu X. [\tau] X \land [a \text{ caret send}] X \land [a \text{ call a}] X \land [a \text{ ret a}] X \land [a \text{ caret read}] \phi' \]
\[ \phi' = \nu Y. [\tau] Y \land [a \text{ caret read}] Y \land [a \text{ call a}] Y \land [a \text{ ret a}] Y \land [a \text{ caret send}] \text{ff} \]

gives rise to several structural properties, most notably:

\[ \psi = \nu X. [\epsilon] X \land [\text{send}] X \land [a] \psi' \land [\text{read}] \psi' \]
\[ \psi' = \nu Y. [\epsilon] Y \land [\text{read}] Y \land [a] \text{ff} \land [\text{send}] \text{ff} \]

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:

\[
\begin{align*}
\phi_{\text{even}} &= \nu X. \ [\text{even caret even}] \text{ff} \land [\text{even caret odd}] \phi'_{\text{even}} \land [\tau] X \\
\phi'_{\text{even}} &= \nu Y. \ [\text{even caret even}] \text{ff} \land [\text{even caret odd}] \text{ff} \land [\tau] Y
\end{align*}
\]
Restricted Behavioural Logic: Atomic Calls

Behavioural specification of even:

$$\phi_{\text{even}} = \nu X. \ [\text{even caret even}] \text{ff} \land [\text{even caret odd}] \phi'_{\text{even}} \land [\tau] X$$

$$\phi'_{\text{even}} = \nu Y. \ [\text{even caret even}] \text{ff} \land [\text{even caret odd}] \text{ff} \land [\tau] Y$$

gives rise to a single structural property:

$$\phi_{\text{even}} = \nu X. \ [\text{even}] \text{ff} \land [\text{odd}] \phi'_{\text{even}} \land [\epsilon] X$$

$$\phi'_{\text{even}} = \nu Y. \ [\text{even}] \text{ff} \land [\text{odd}] \text{ff} \land [\epsilon] Y$$
Restricted Behavioural Logic: Atomic Calls

Behavioural specification of even:

\[
\phi_{\text{even}} = \nu X. \ [\text{even caret even}] \text{ff} \land [\text{even caret odd}] \phi'_{\text{even}} \land [\tau] X \\
\phi'_{\text{even}} = \nu Y. \ [\text{even caret even}] \text{ff} \land [\text{even caret odd}] \text{ff} \land [\tau] Y
\]

gives rise to a single structural property:

\[
\phi_{\text{even}} = \nu X. \ [\text{even}] \text{ff} \land [\text{odd}] \phi'_{\text{even}} \land [\epsilon] X \\
\phi'_{\text{even}} = \nu Y. \ [\text{even}] \text{ff} \land [\text{odd}] \text{ff} \land [\epsilon] Y
\]

obtained through a direct translation!
5. Tool Support

The CVPP Tool Set

- Program
- Analyser
  - Graph
    - compose
    - convert
    - inline
  - Model
    - flow graph
    - FSM
    - PDS
- MaxMod
- ModCheck
  - Moped
  - CWB
  - Formula
    - simplify
    - convert
    - CWB/LTL
    - beh2struct
- Formula
  - structure
  - behaviour
  - eqsys
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

Dilian Gurov (KTH)
Modular Verification of Temporal Safety Prop
15 November 2011 36 / 41
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
**ProMoVer: A wrapper around CVPP**

![Diagram of ProMoVer system]

- **Local Properties**
  - Analyzer
  - Graph Tool
  - CWB

- **Global Properties**
  - Max. Model
  - Moped

- **Counterexample**
  - Spec. Extractor
  - Graph Tool

- **System Components**
  - Pre-Processor
  - Post-Processor

Details in:

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

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
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