Modular Software Verification

Dilian Gurov

KTH Royal Institute of Technology, Stockholm, Sweden

RTA-CSIT 2014 Invited Talk
Tirana, 13 December 2014

Functional Verification of Procedural Programs: Hoare Logic

```
public class EvenOdd {

   //@ requires n >= 0;
   //@  ensures \result == (\exists int k; n == 2 * k);
   public boolean even(int n) {
      if (n == 0) return true;
      else return odd(n-1);
   }

   //@ requires n >= 0;
   //@  ensures \result == (\exists int k; n == 2 * k + 1);
   public boolean odd(int n) {
      if (n == 0) return false;
      else return even(n-1);
   }
}
```

Verification of Temporal Properties

- Temporal properties:
  "First call of `even` is not to itself"

- Temporal logics:
  - Linear-time Temporal Logic (LTL):
    $\texttt{even} \Rightarrow \mathsf{X}\ ((\texttt{even} \wedge \neg\texttt{entry})\ \mathsf{W}\ \texttt{odd})$
  - μ-calculus:
    $\texttt{even} \Rightarrow \nu X.\ [\mathsf{even\ call\ even}]\,\mathsf{ff} \wedge [\tau]\,X$

- Algorithmic verification: Model Checking
  Decidable for finite-state and push-down systems

Model Checking of Procedural Programs

Various techniques:

- Ball et al 2001: Predicate Abstraction
- Das et al 2002: Property Simulation
- Esparza et al 2002: Pushdown Systems

Not modular!

Modular Model Checking

- Can one infer a global property from the local specifications?
- Idea: use **maximal models**!
    - Grumberg & Long 1994: ACTL
    - Kupferman & Vardi 2000: ACTL*

  Developed for finite-state systems

Our work: Procedures + Temporal + Modular

- 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, Ina Schaefer, Afshin Amighi, Pedro Gomes

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 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

Spacial variability:

- multiple variants, as arising from software product lines

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

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

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

Variability is then dealt with naturally.
But... how, and is there an algorithmic solution?

Program Model

Our approach is to use a unifying **program 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:

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

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

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

Program Model

For the third task:

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

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

Our Setup

A. Program model: Flow graphs capturing control flow

- behaviour as induced pushdown automaton

B. Properties: legal sequences of method invocations

- temporal safety properties

C. Verification: pushdown automata model checking

- essentially a language inclusion problem

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

A. Program Model

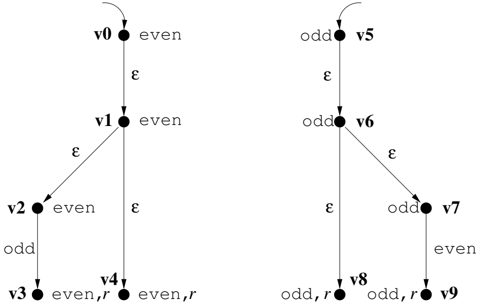
Flow Graph:

```
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:

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

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:

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

Flow Graph Extraction from Java Bytecode

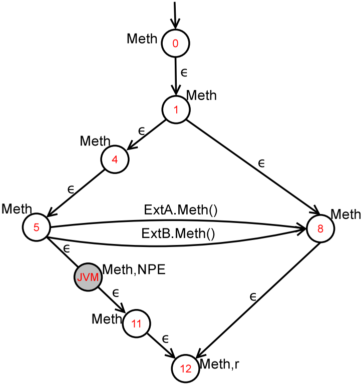
Java program:

```
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
```



Sound Control–Flow Graph Extraction for Java Programs with Exceptions
Afshin Amighi, Pedro Gomes, Dilian Gurov and Marieke Huisman
In Proceedings of SEFM 2012, LNCS 7504, pp. 33–47

B. Properties

Example **structural** property:

- "The program is tail recursive":

$$\nu X.\ [\texttt{even}]\, r \wedge [\texttt{odd}]\, r \wedge [\varepsilon]\, X$$

- can be checked with standard finite–state model checking

Example **behavioural** property:

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

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

- can be checked with PDA model checking

More behavioural properties

- “No send after read”
- “A vote is only submitted after validation”
- “Votes are only counted after voting has finished”
- “No non–atomic operations within transactions”

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:

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

Reducing Behavioural to Structural Properties
Dilian Gurov and Marieke Huisman
Theoretical Computer Science
vol. 480, pp. 69–103, 2013

Constructing Maximal Models

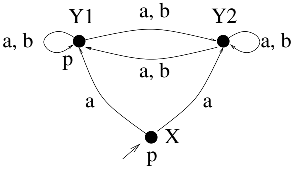
Atoms $\{p\}$, labels $\{a, b\}$, formula $[b]\,\mathrm{ff} \wedge p$

The formula as an **equation system**:

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

Converted into **simulation normal form**:

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



$(\mathcal{M}, E)$

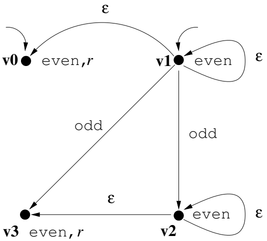
C. Verification

Structural specification for `even`:

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

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

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

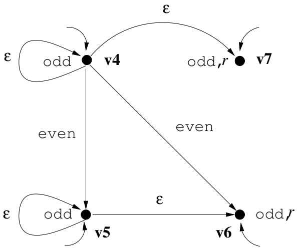


Structural specification for `odd`:

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

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

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

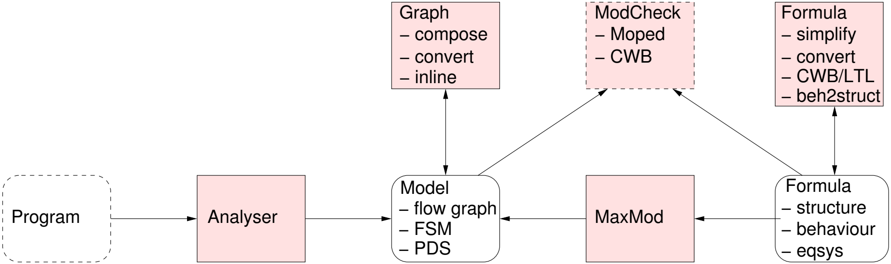


Verify the global behavioural specification:

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

Tool Support

The CVPP Tool Set



Graph
– compose
– convert
– inline

ModCheck
– Moped
– CWB

Formula
– simplify
– convert
– CWB/LTL
– beh2struct

Program

Analyser

Model
– flow graph
– FSM
– PDS

MaxMod

Formula
– structure
– behaviour
– eqsys

Automation

Full automation would require:

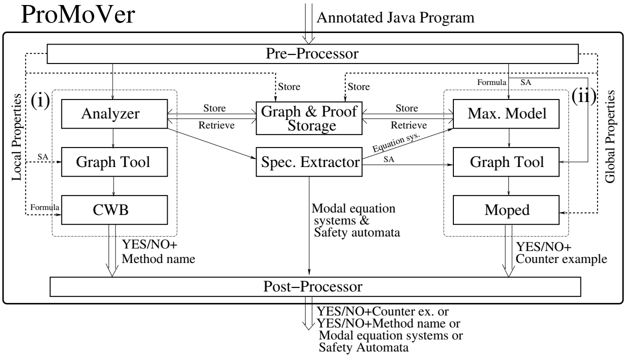
- single input to the checker
- local and global specs as annotations/comments
- inspired from JML based verification tools like ESC/Java
- pre– and post–processing

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

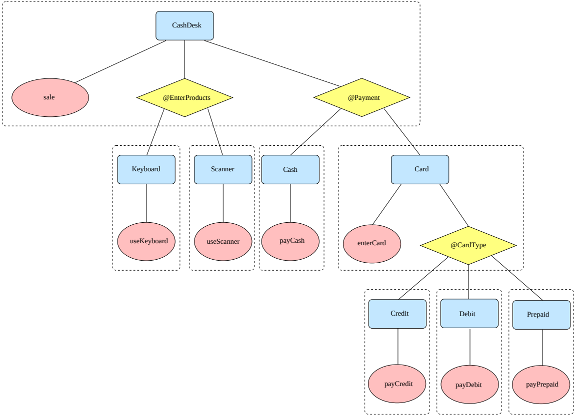
ProMoVer: A wrapper around CVPP



Procedure-Modular Verification of Temporal Safety Properties
Siavash Soleimanifard, Dilian Gurov and Marieke Huisman
Journal of Software and Systems Modeling, 2013

Application Area: 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!

Solution: relativize on properties of variation points!

Results in one verification task per region!

Compositional Algorithmic Verification of Software Product Lines
Ina Schaefer, Dilian Gurov and Siavash Soleimanifard
In Post–proceedings of FMCO 2010, LNCS 6957, pp. 184–203

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 if 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:
  - 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
  - object references