Procedure-Modular Verification of Temporal Safety Properties

Siavash Soleimanifard

School of Computer Science and Communication
KTH Royal Institute of Technology
Stockholm

Licentiate Thesis Presentation
23 May 2012

Outline

- Scope and goal
- Modular verification
- CVPP framework & toolset
- Contributions
- ProMoVer
- Verification of product families
- Boolean flow graphs
- Conclusion & future work

Scope and Goals

- Verification of software systems in the presence of variability

Scope and Goals

- Verification of software systems in the presence of variability
  - open systems

Scope and Goals

- Verification of software systems in the presence of variability
  - open systems
  - mobile code

Scope and Goals

- Verification of software systems in the presence of variability
  - open systems
  - mobile code
  - code evolution

Scope and Goals

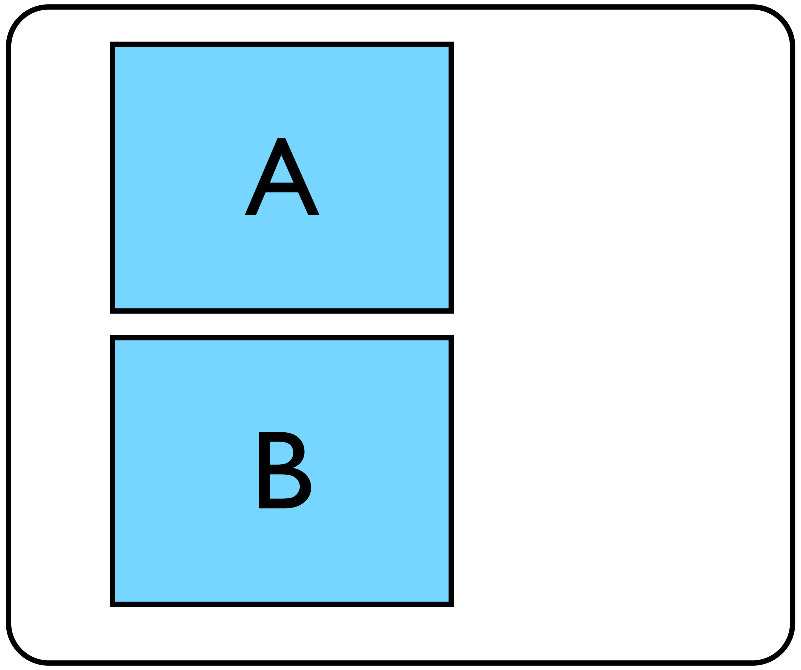
- Verification of software systems in the presence of variability
  - open systems
  - mobile code
  - code evolution
  - multiple implementation

Scope and Goals

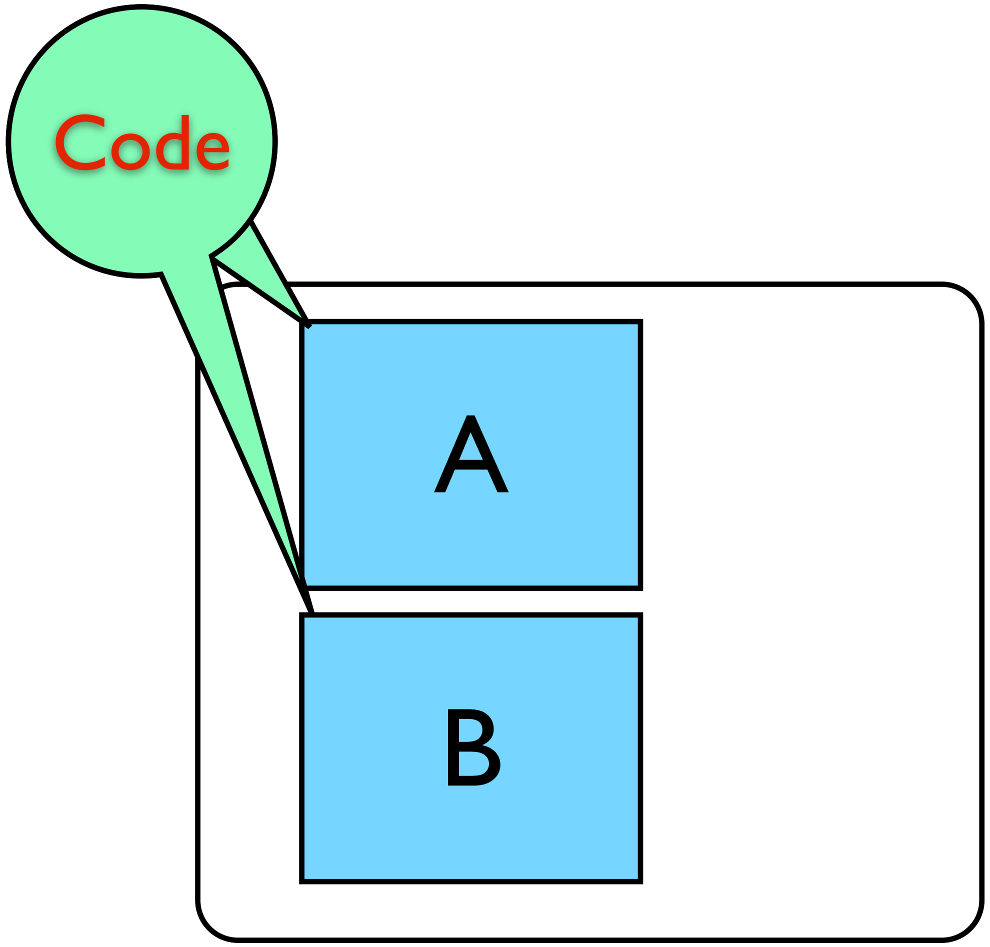
- Verification of software systems in the presence of variability
  - open systems
  - mobile code
  - code evolution
  - multiple implementation
- Any solution should be modular

Modular Verification

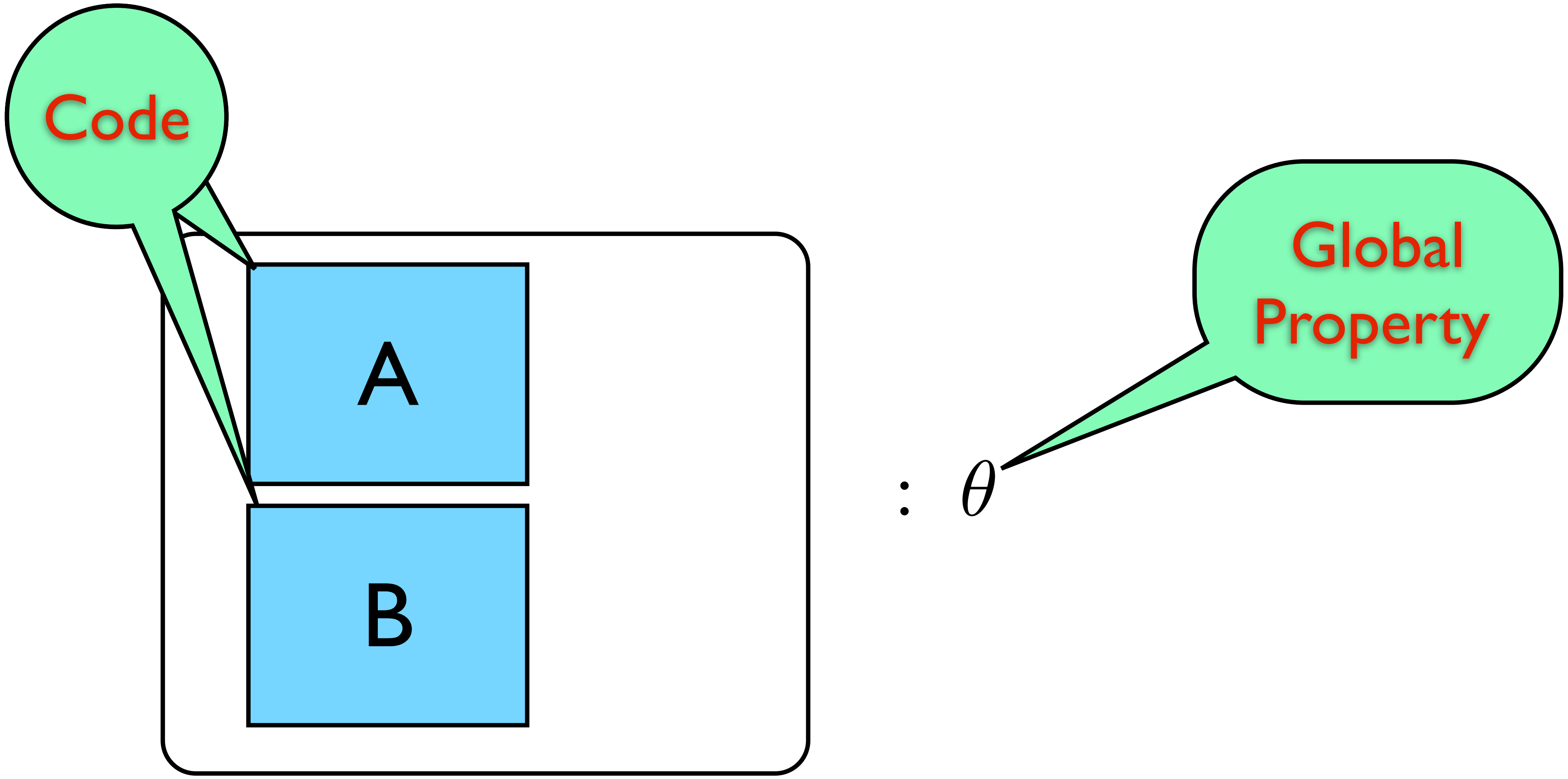
Modular Verification



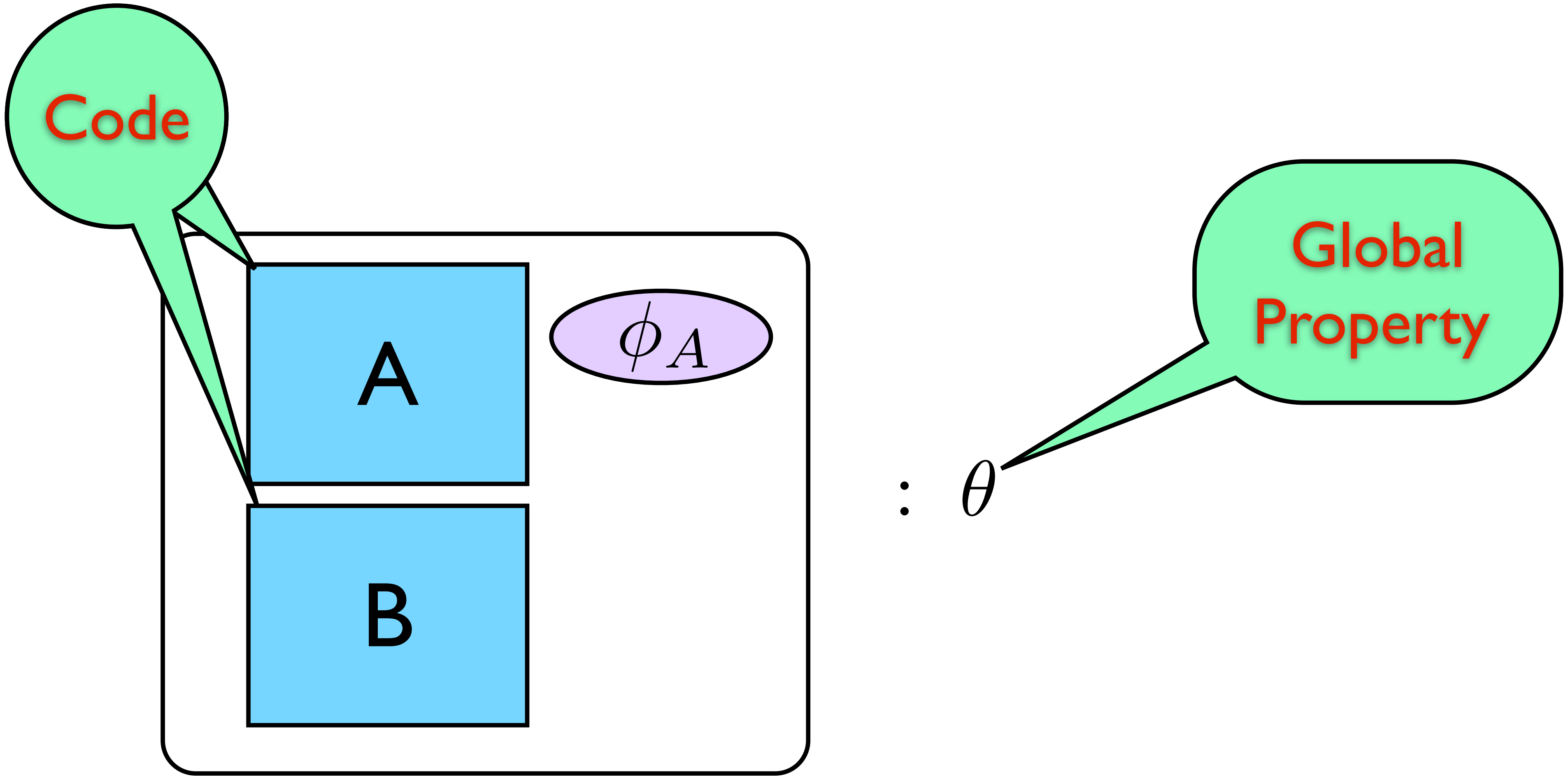
Modular Verification



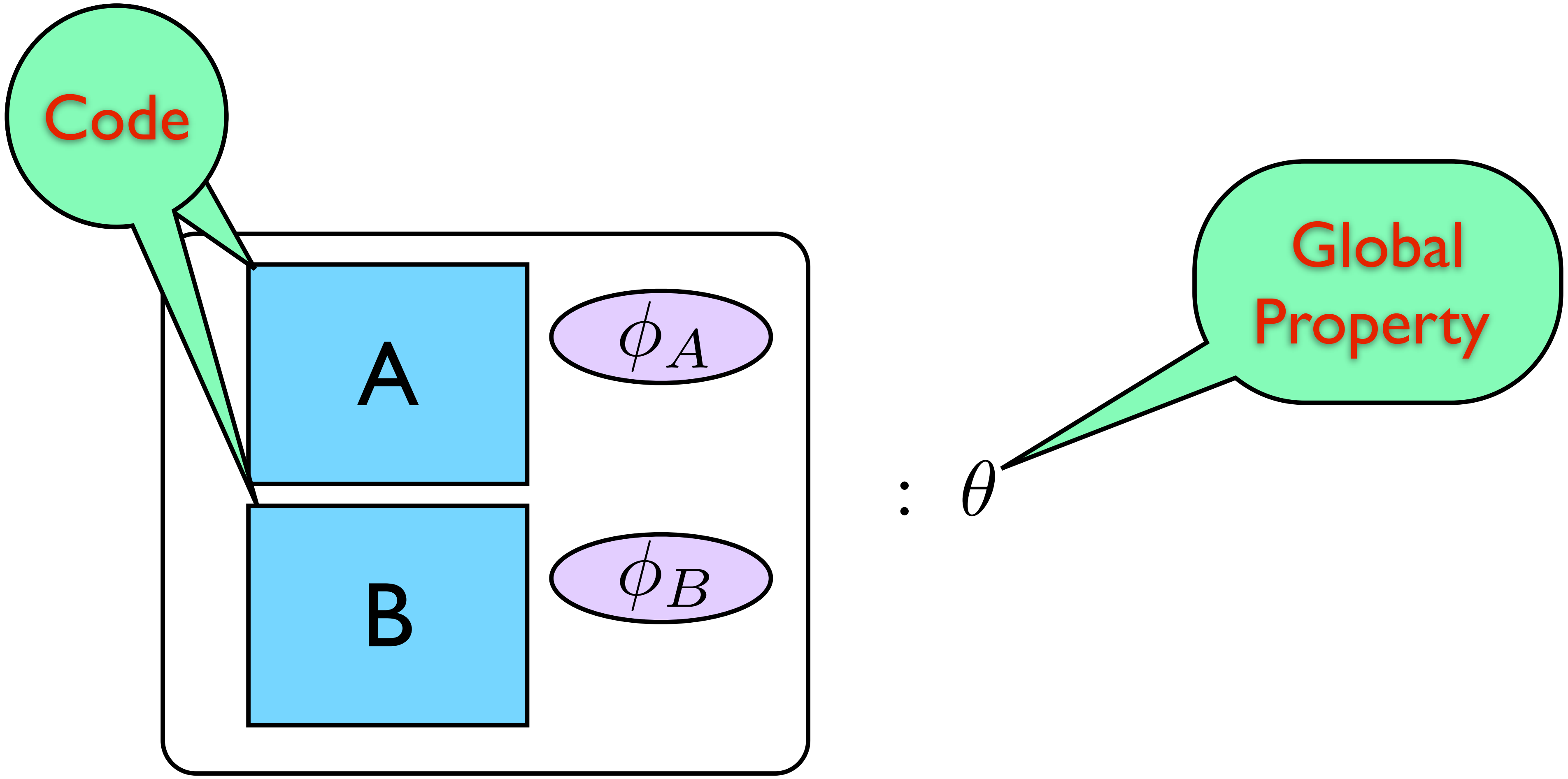
Modular Verification



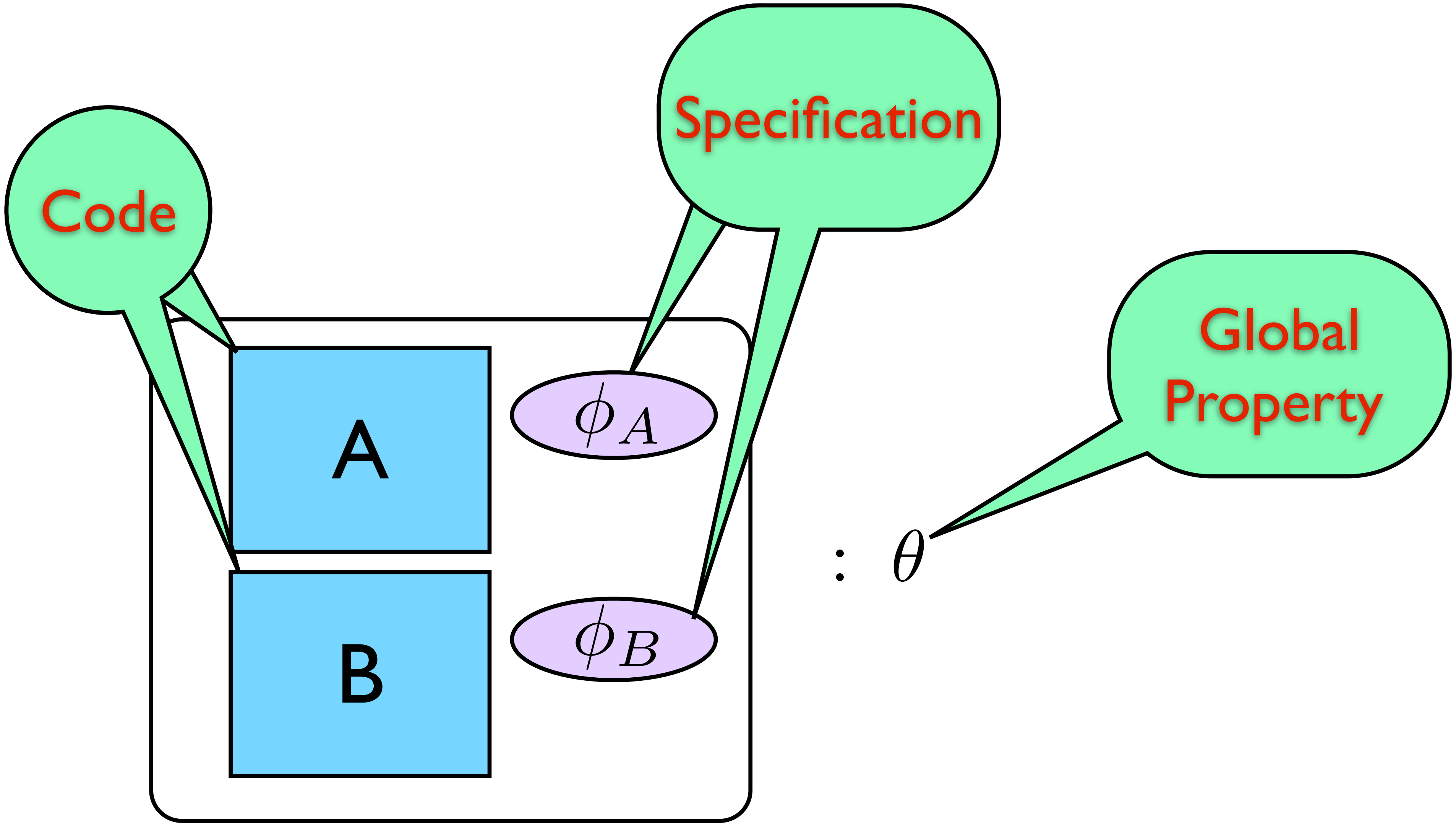
Modular Verification



Modular Verification

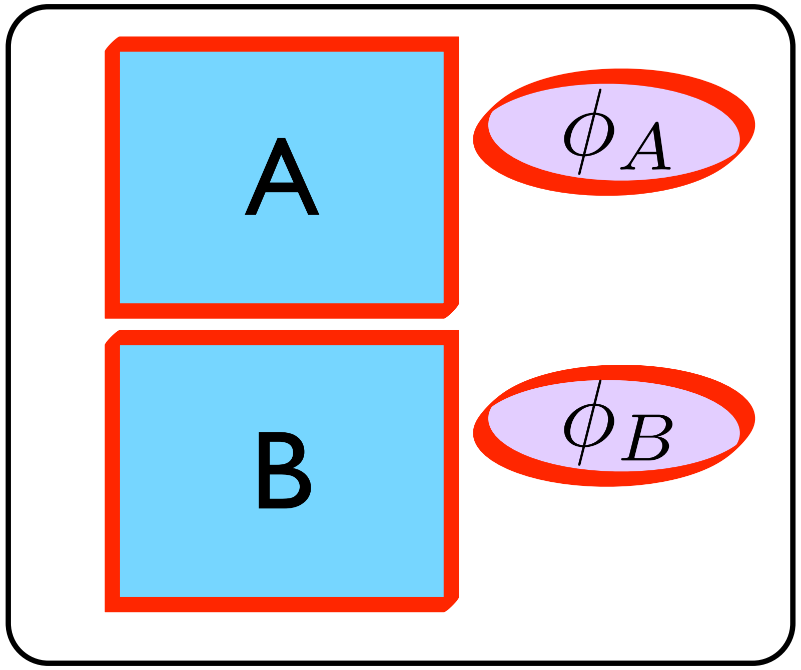


Modular Verification



Modular Verification

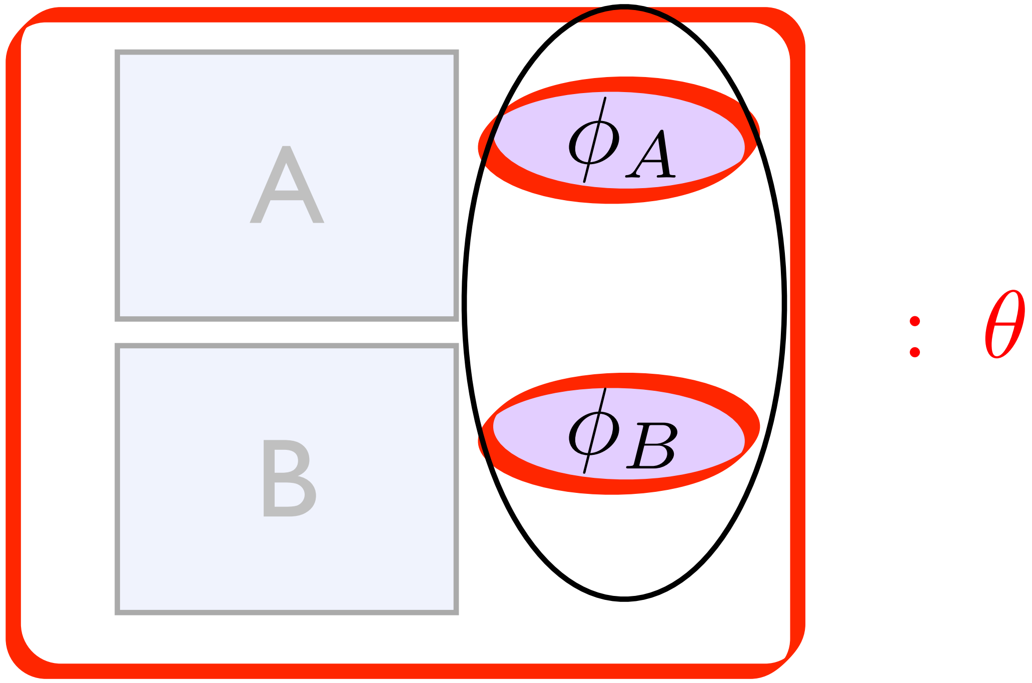
Task I: Local Check



$: \theta$

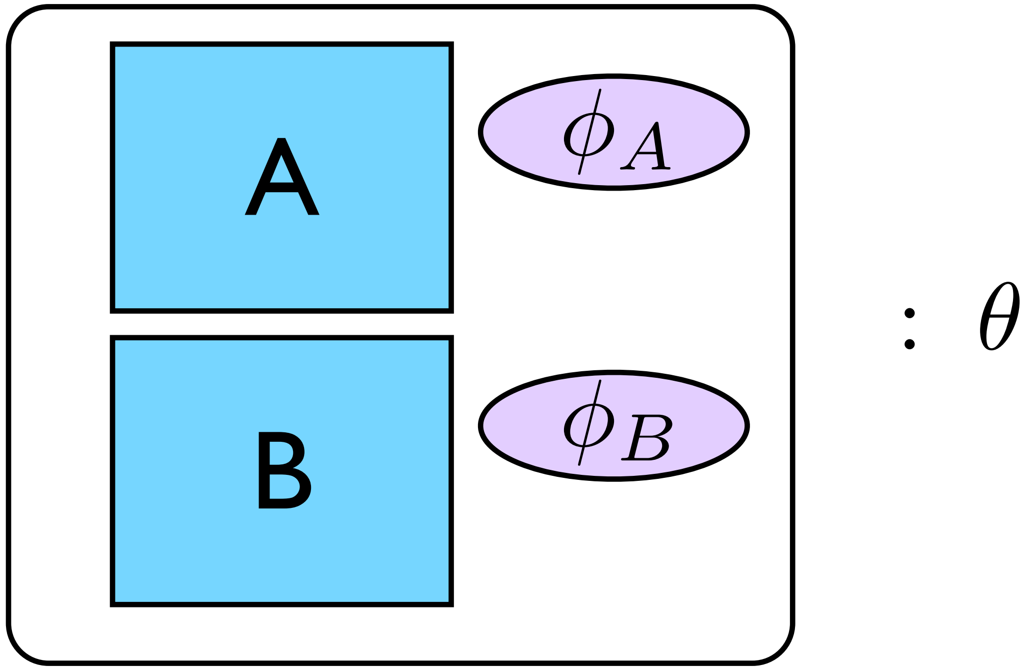
Modular Verification

Task II: Global Check



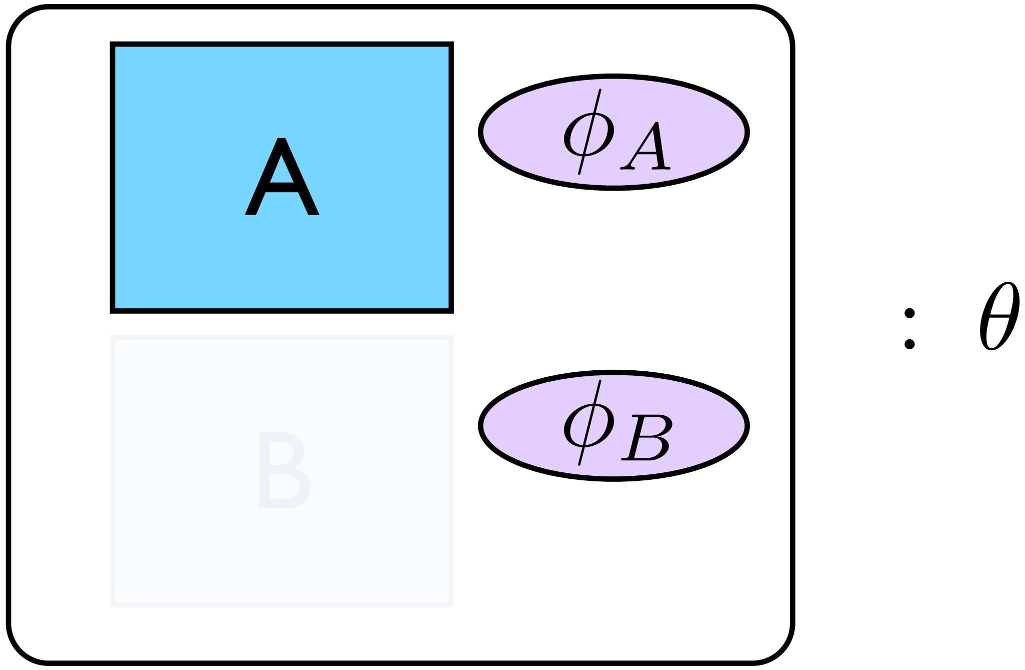
Modularity & Variability

- Open sys. & Mobile code
- Code evolution
- Multiple implementations



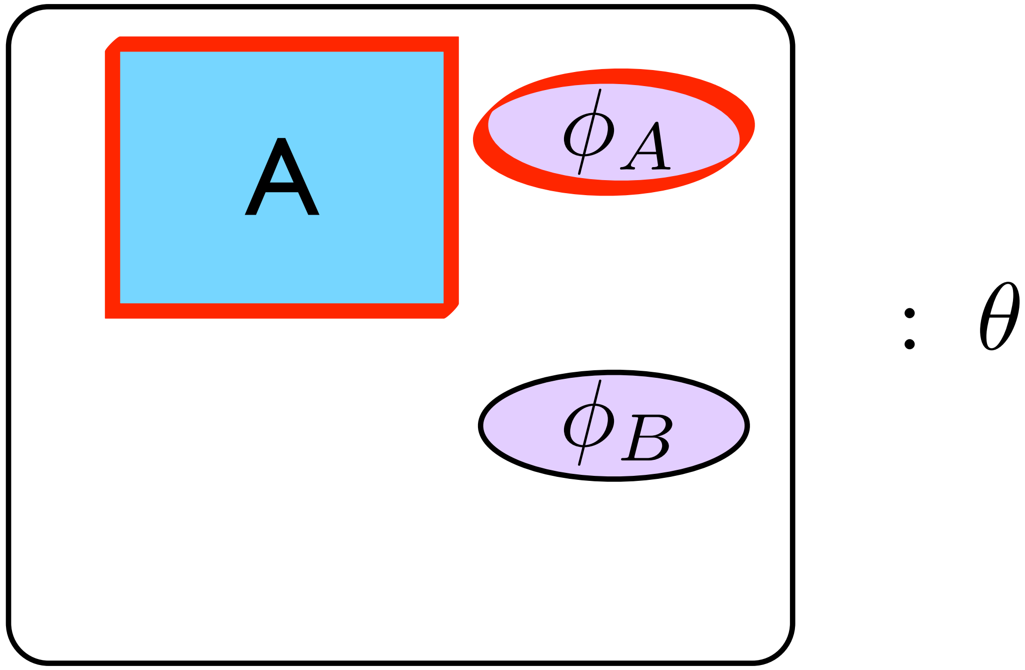
Modularity & Variability

- Open sys. & Mobile code
- Code evolution
- Multiple implementations



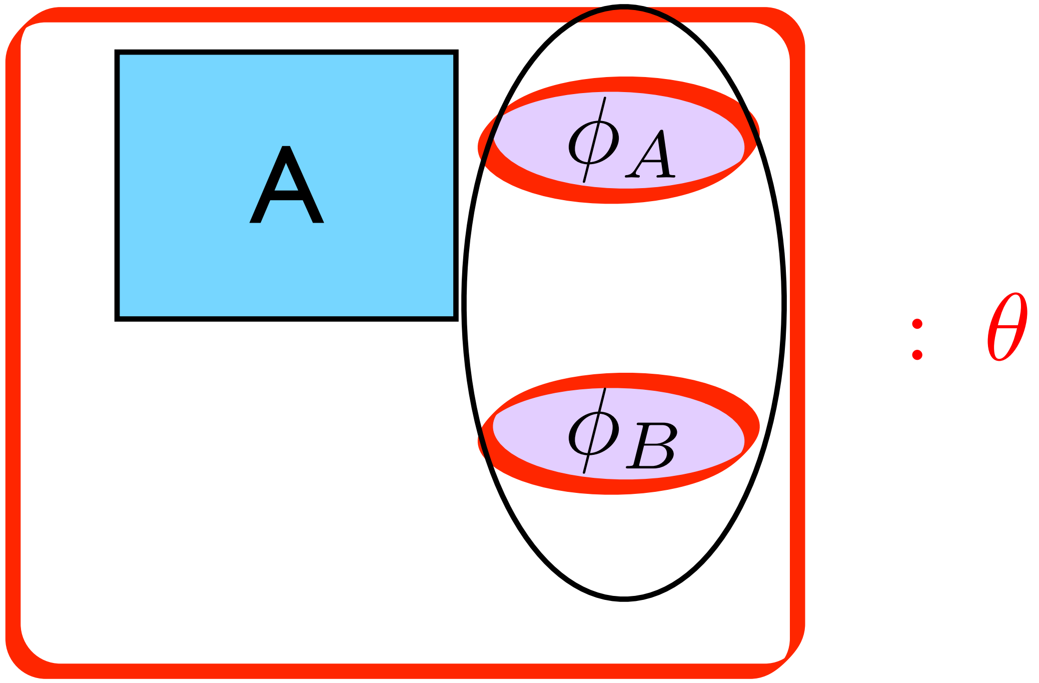
Modularity & Variability

- Open sys. & Mobile code
- Code evolution
- Multiple implementations



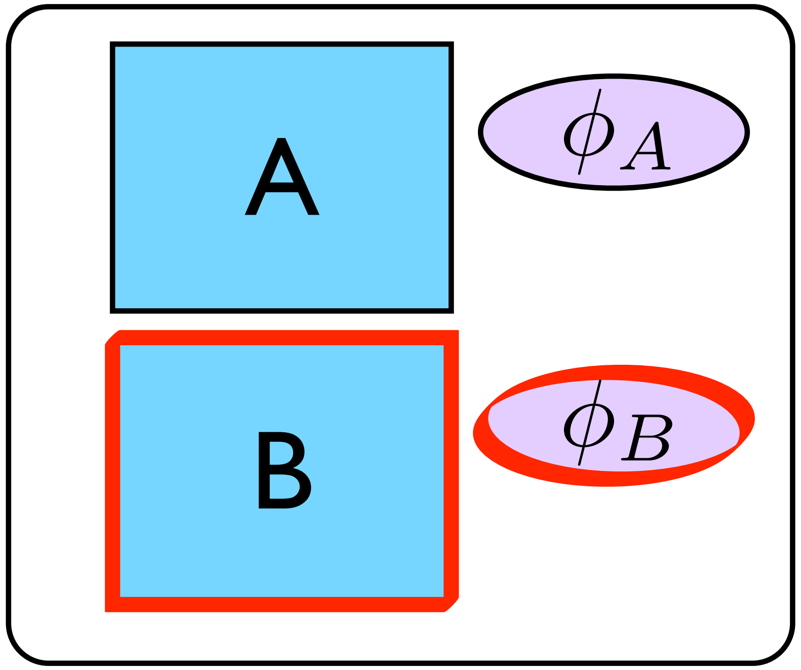
Modularity & Variability

- Open sys. & Mobile code
- Code evolution
- Multiple implementations



Modularity & Variability

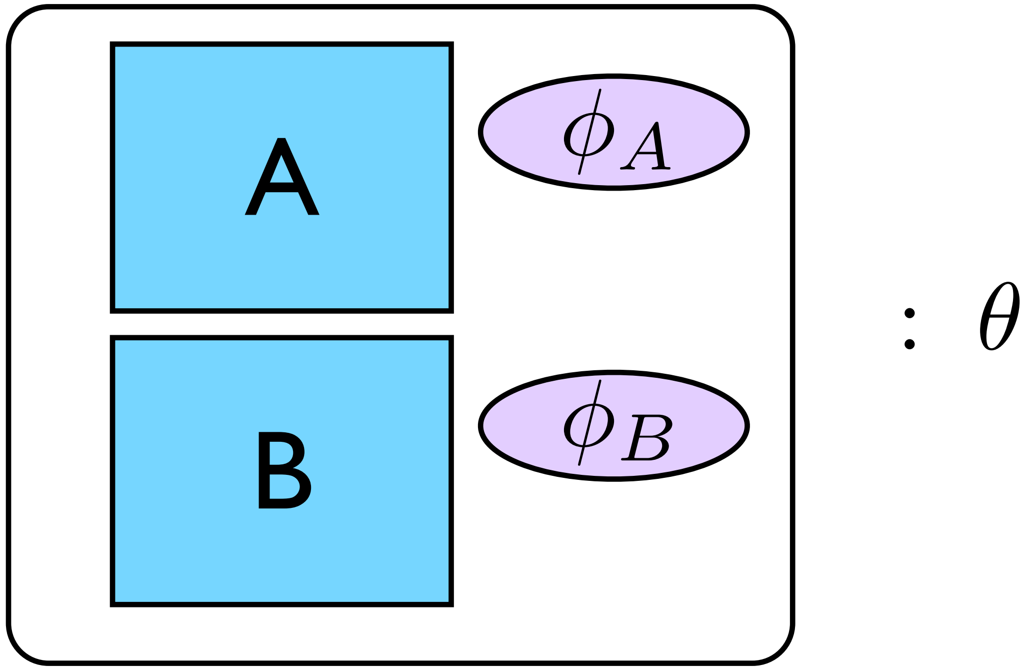
- Open sys. & Mobile code
- Code evolution
- Multiple implementations



$: \theta$

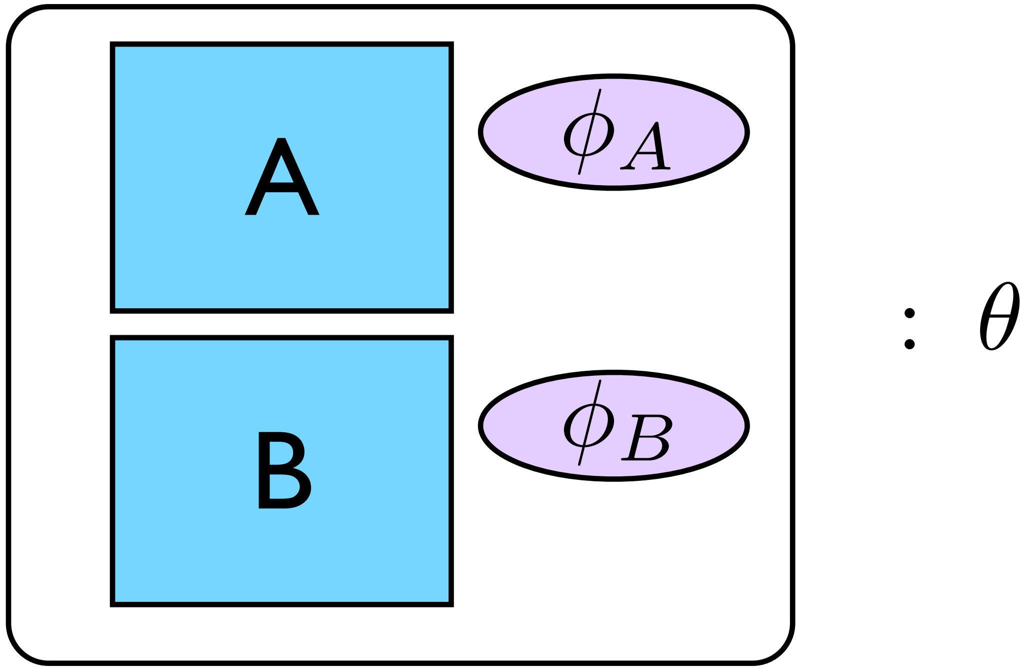
Modularity & Variability

- Open sys. & Mobile code
- Code evolution
- Multiple implementations



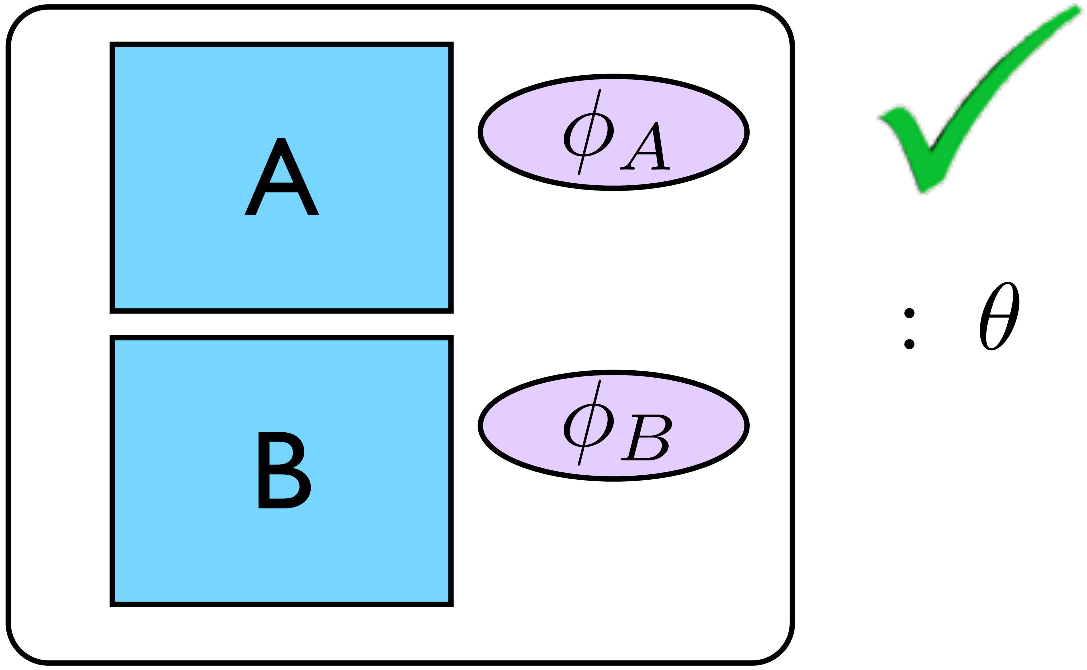
Modularity & Variability

- Open sys. & Mobile code
- Code evolution
- Multiple implementations



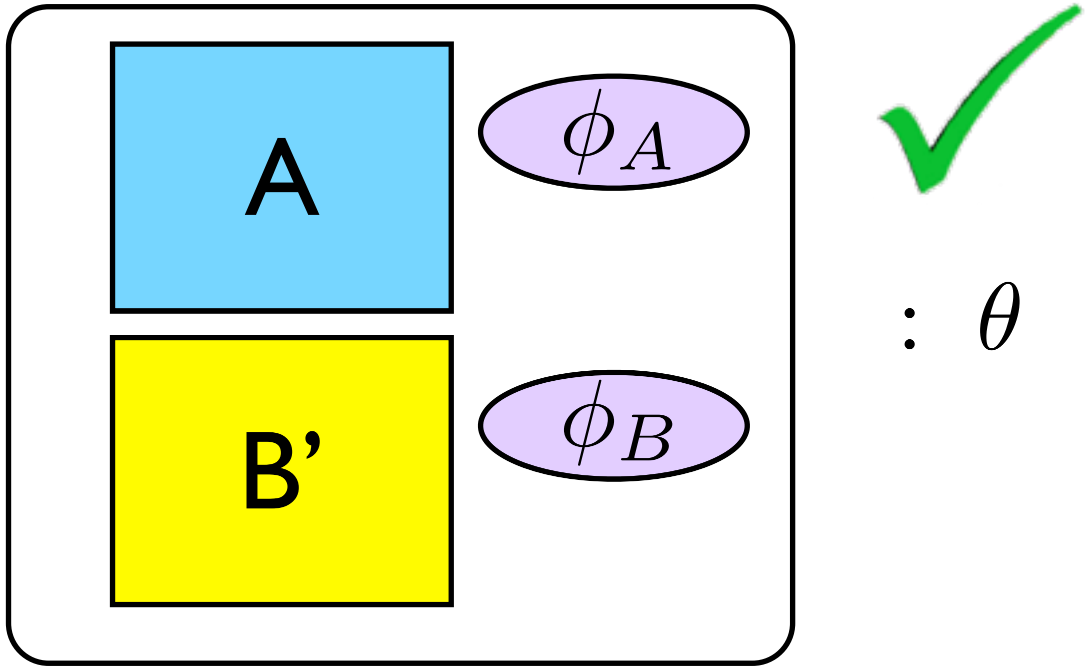
Modularity & Variability

- Open sys. & Mobile code
- Code evolution
- Multiple implementations



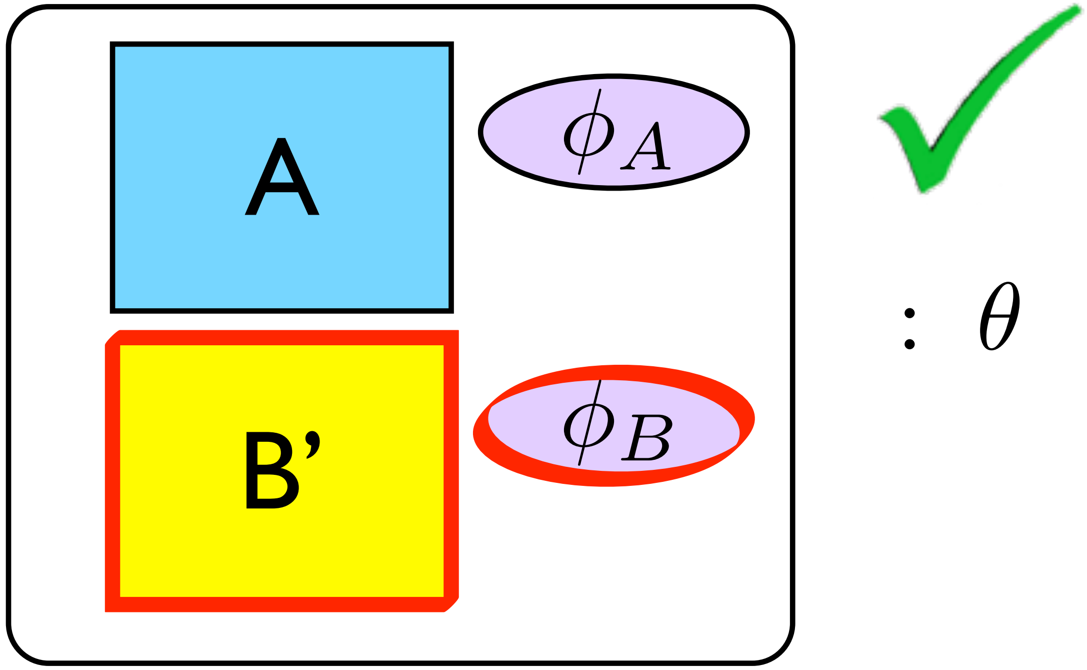
Modularity & Variability

- Open sys. & Mobile code
- Code evolution
- Multiple implementations



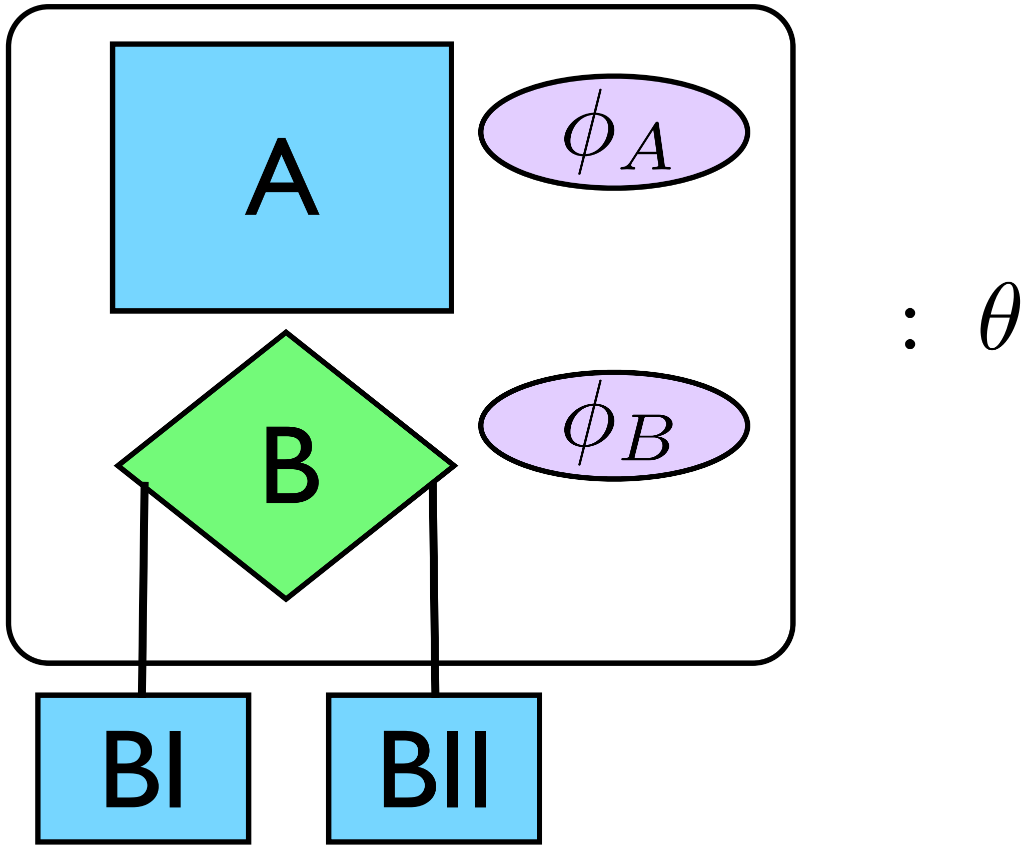
Modularity & Variability

- Open sys. & Mobile code
- Code evolution
- Multiple implementations



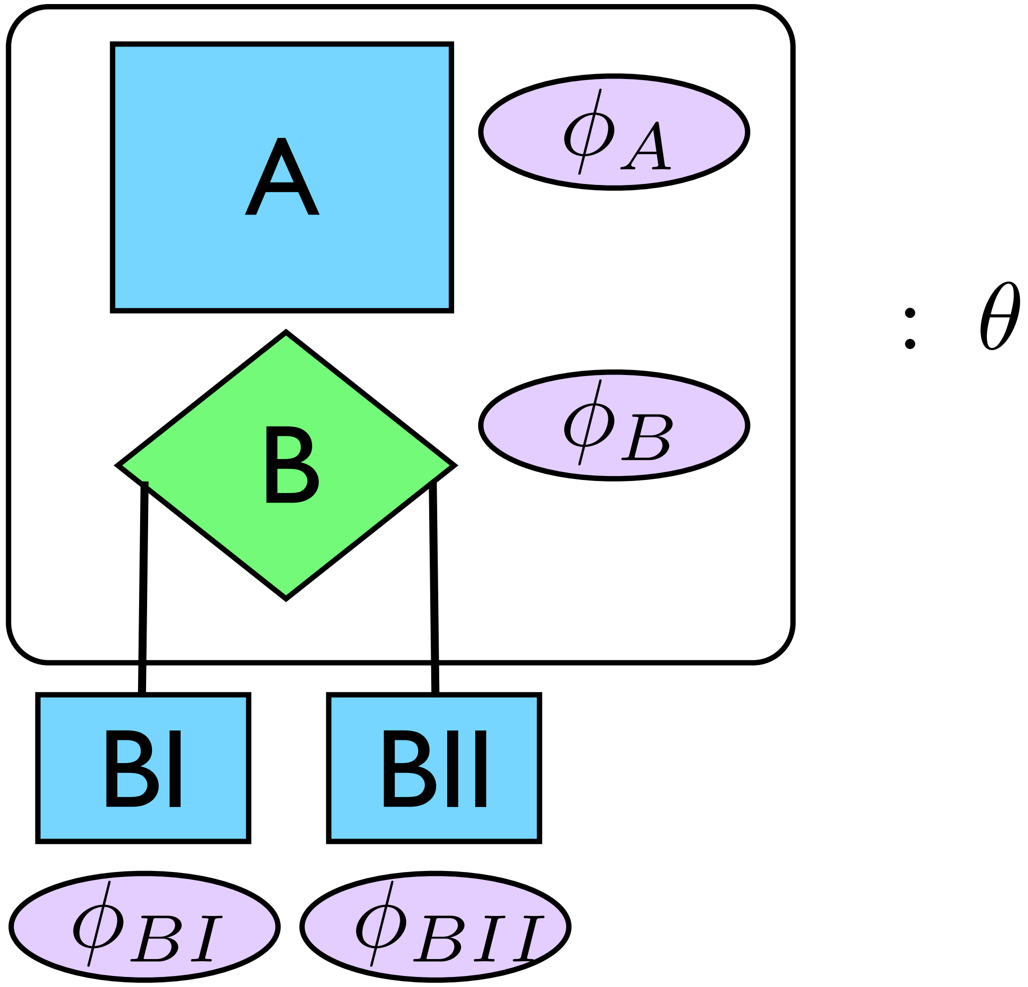
Modularity & Variability

- Open sys. & Mobile code
- Code evolution
- Multiple implementations
  - Product Families



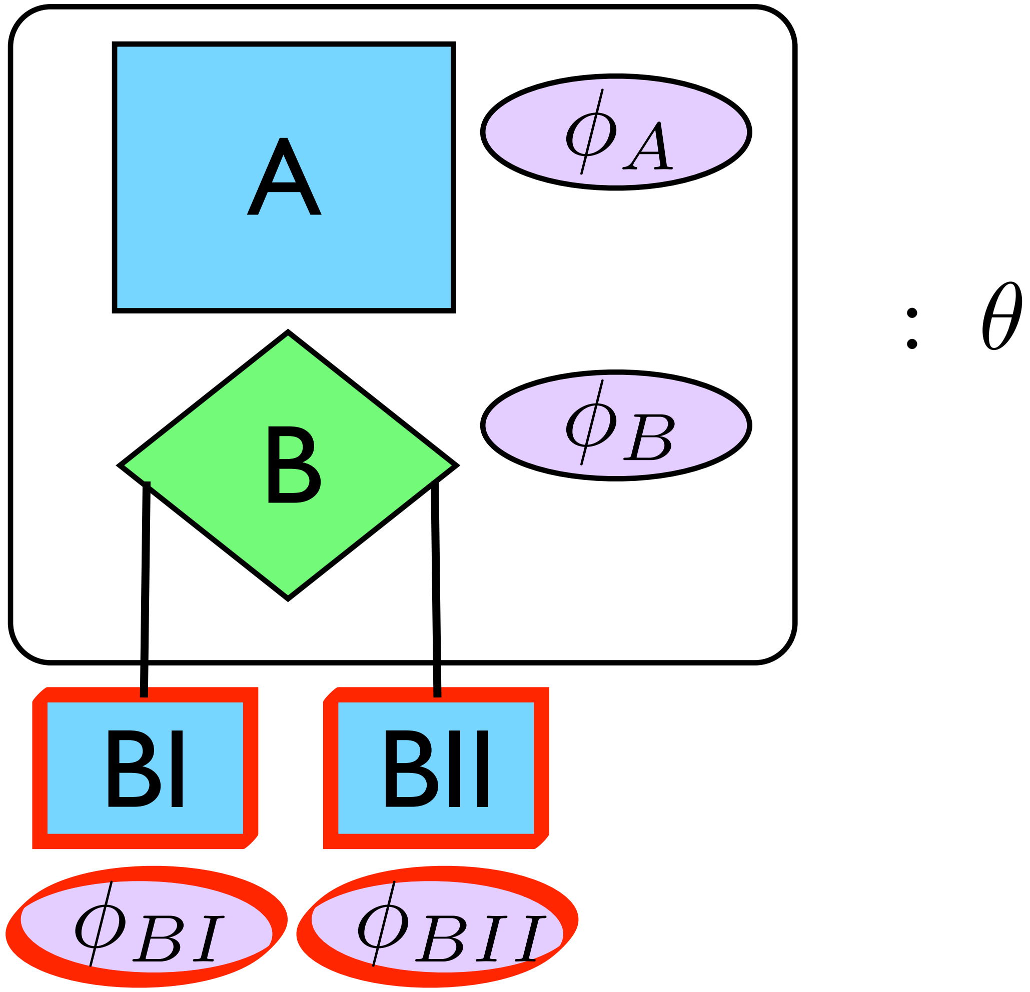
Modularity & Variability

- Open sys. & Mobile code
- Code evolution
- Multiple implementations
  - Product Families



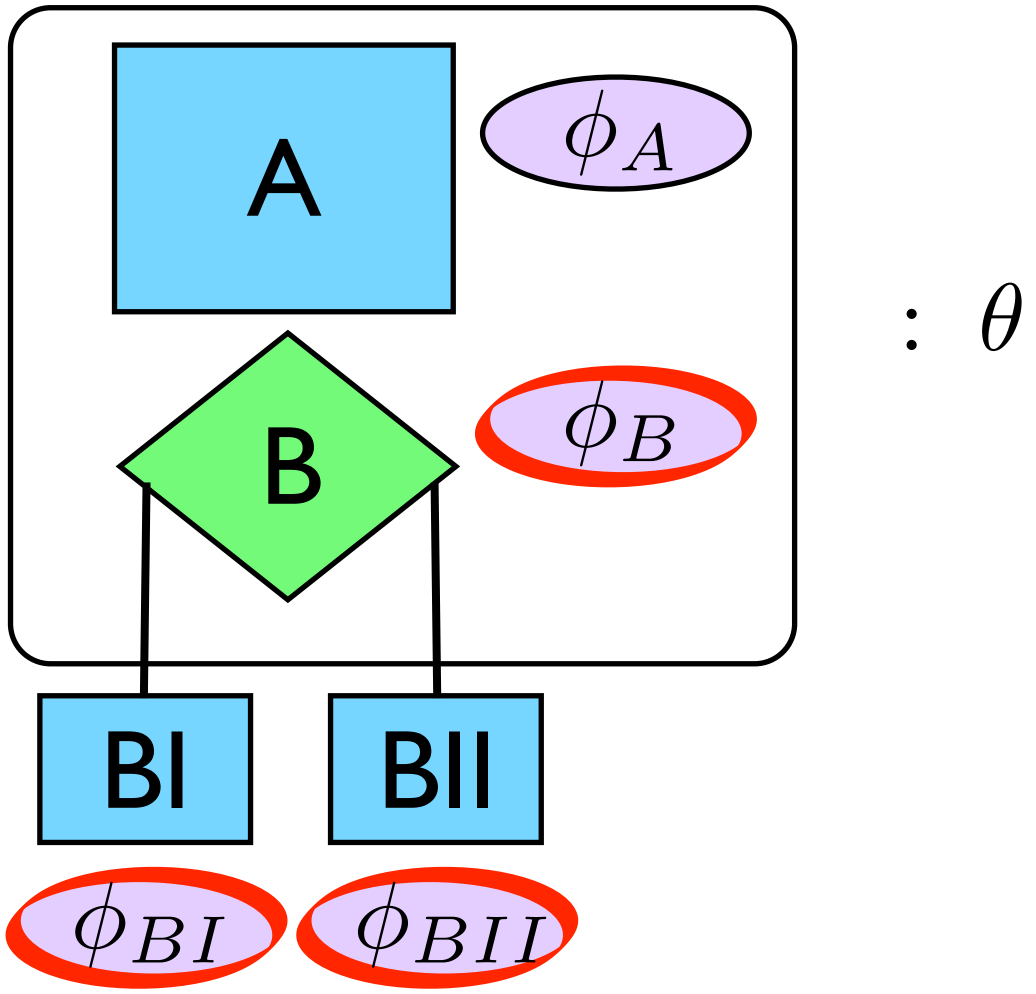
Modularity & Variability

- Open sys. & Mobile code
- Code evolution
- Multiple implementations
  - Product Families



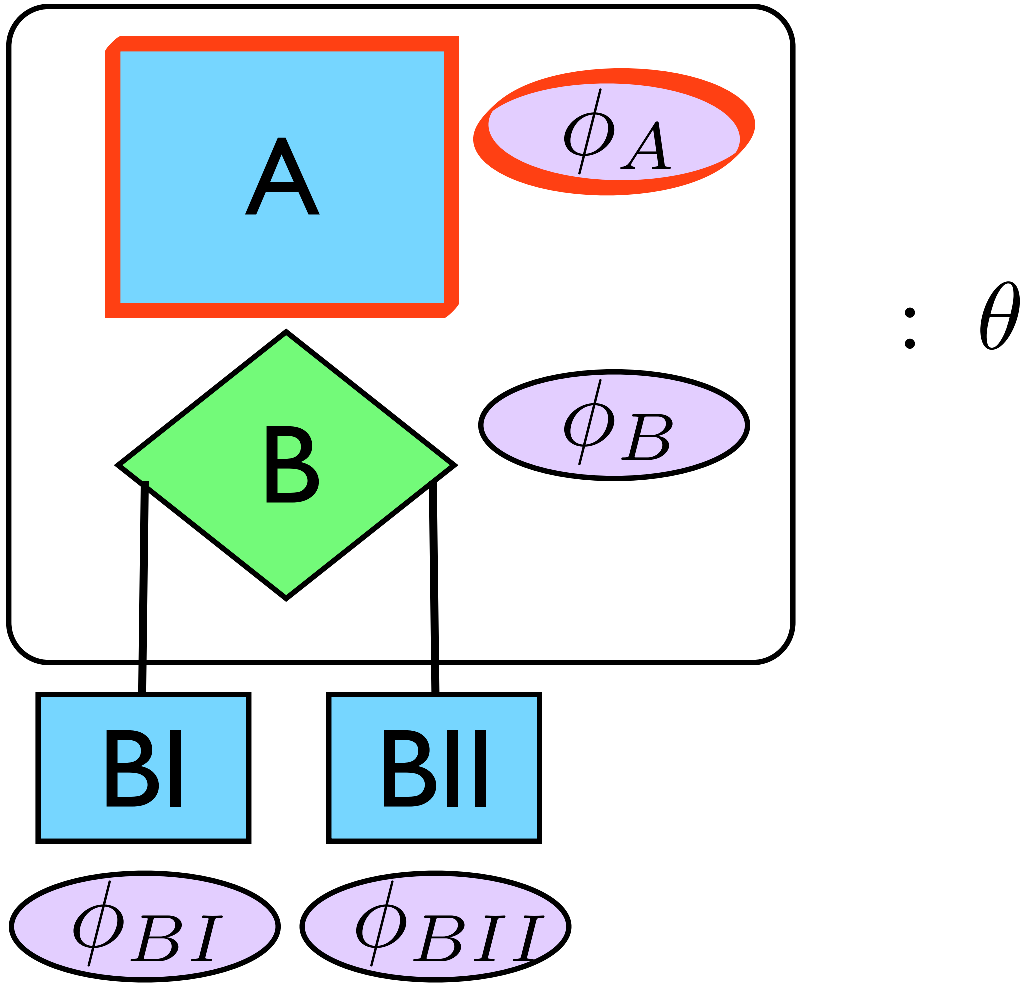
Modularity & Variability

- Open sys. & Mobile code
- Code evolution
- Multiple implementations
  - Product Families



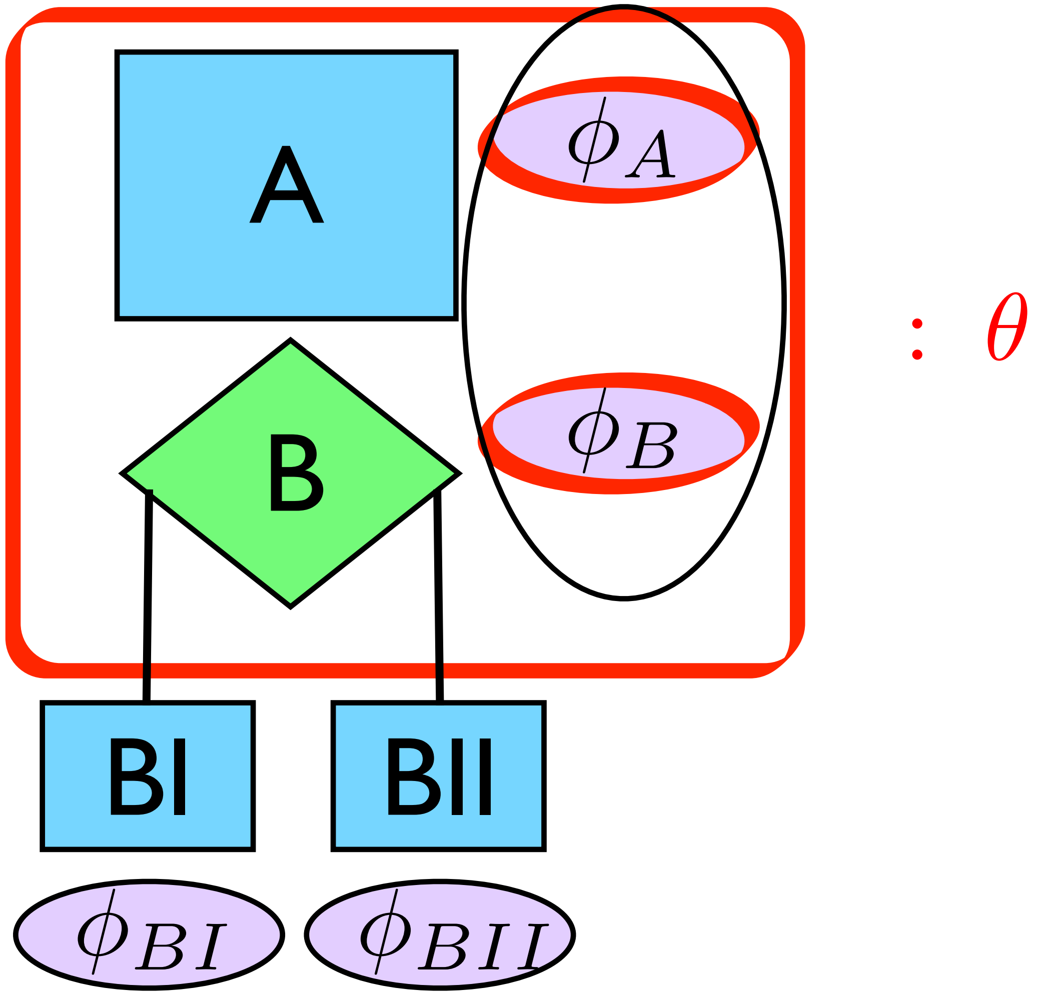
Modularity & Variability

- Open sys. & Mobile code
- Code evolution
- Multiple implementations
  - Product Families



Modularity & Variability

- Open sys. & Mobile code
- Code evolution
- Multiple implementations
  - Product Families



Existing Techniques

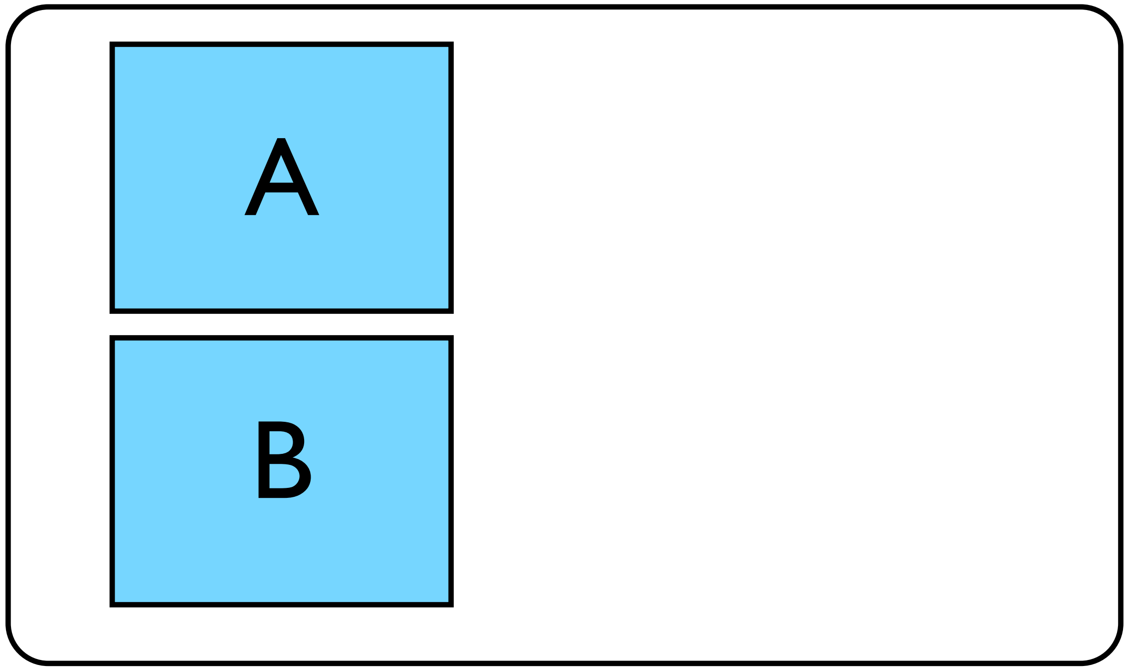
- Hoare logic
  - procedure-modular verification
  - predicate logic
  - theorem proving
- Modular verification and model checking
  - flexible level of granularity
  - temporal logic
  - model checking

Modular Verification

- O. Grumberg and D. Long 1994
  - **finite-state** models
  - **maximal models**
- D. Gurov, M. Huisman and C. Sprenger 2004
  - **infinite-state** models (pushdown systems)
  - **maximal models**
  - CVPP framework

CVPP

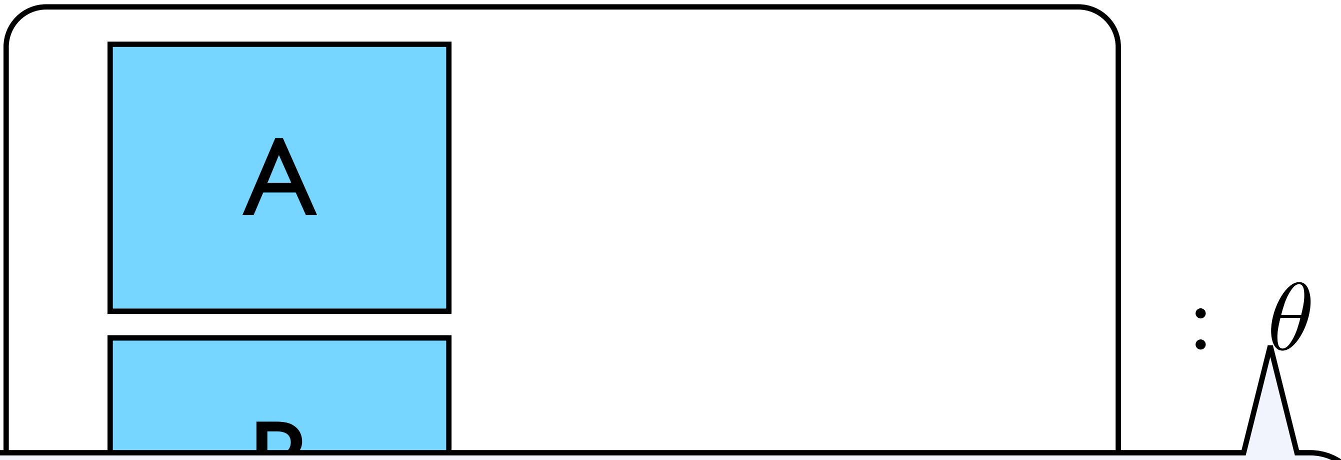
- Verify specs locally
- Construct maximal models from local specs
- Compose Max model
- Model check global property



$: \theta$

CVPP

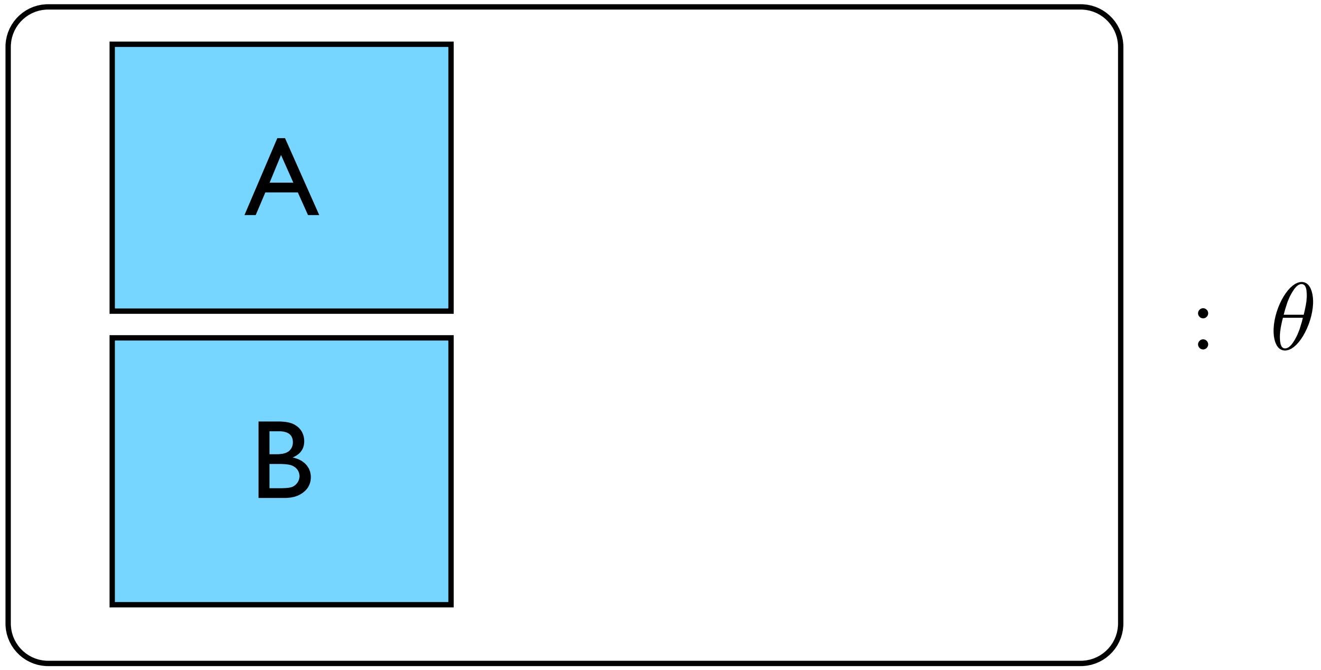
- Verify specs locally
- Construct maximal models from local specs
- Compose M
- Model chec property



- Temporal control flow
- Legal sequences of method invocation
    - a method to change sensitive data is only called within authentication method

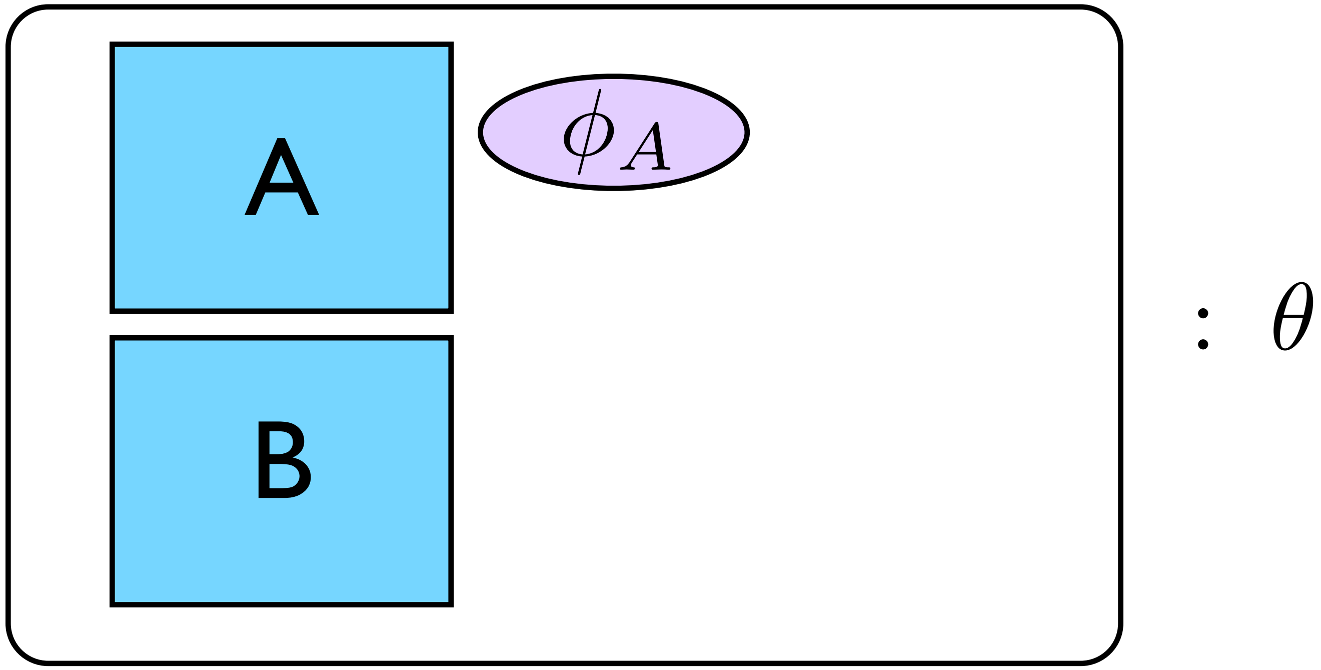
CVPP

- Verify specs locally
- Construct maximal models from local specs
- Compose Max model
- Model check global property



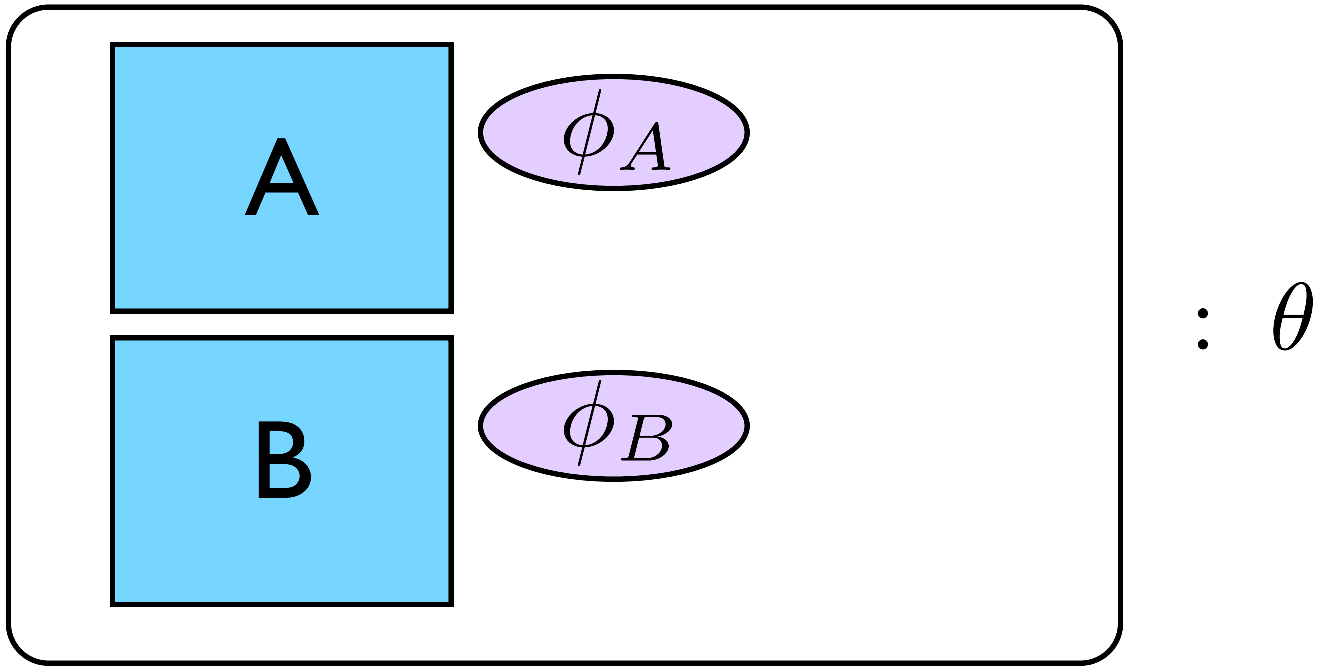
CVPP

- Specify modules
- Verify specs locally
- Construct maximal models from local specs
- Compose Max model
- Model check global property



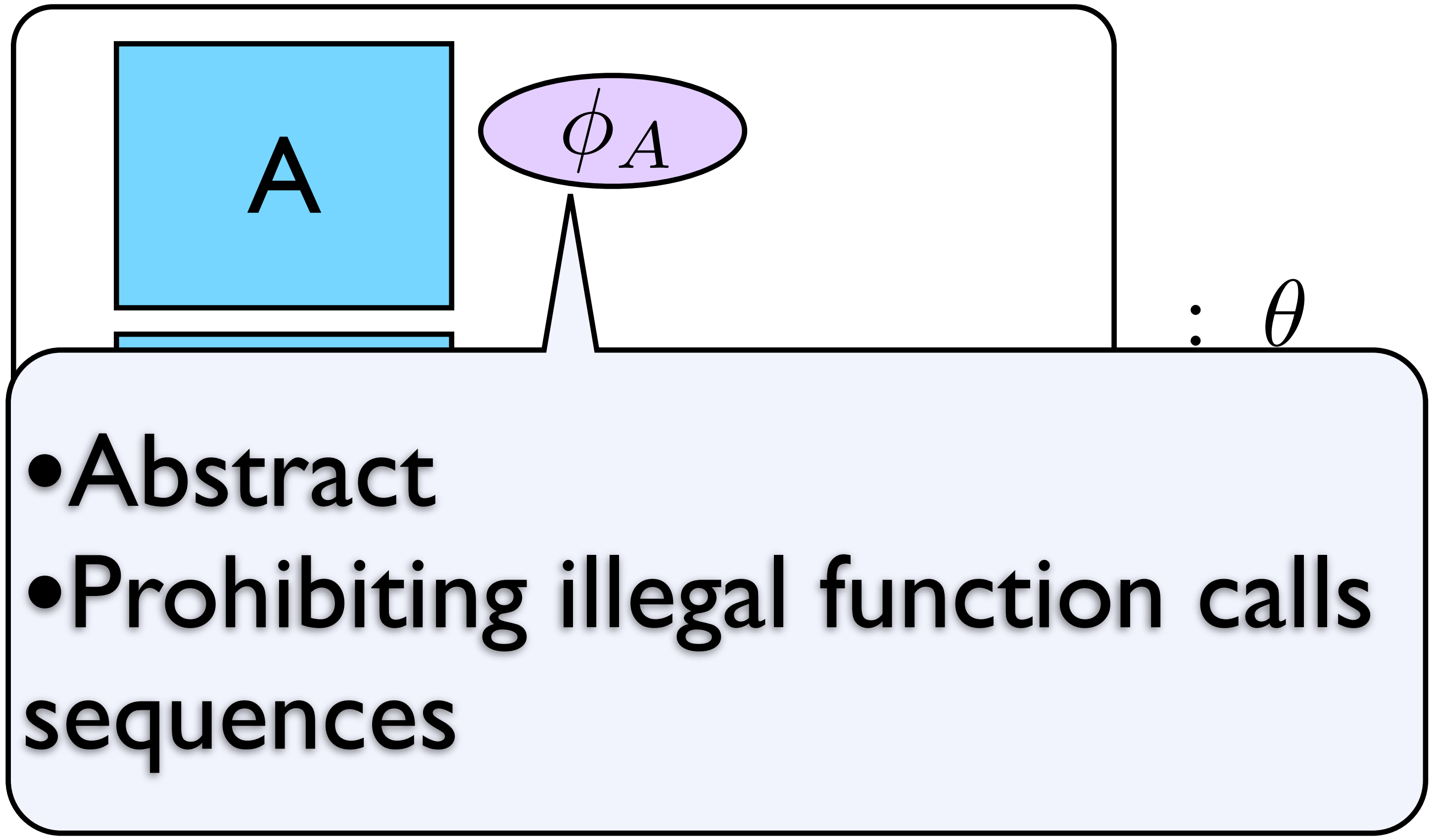
CVPP

- Specify modules
- Verify specs locally
- Construct maximal models from local specs
- Compose Max model
- Model check global property



CVPP

- Specify modules
- Verify specs locally
- Construct maximal models from local specs
- Compose Max model
- Model check global property

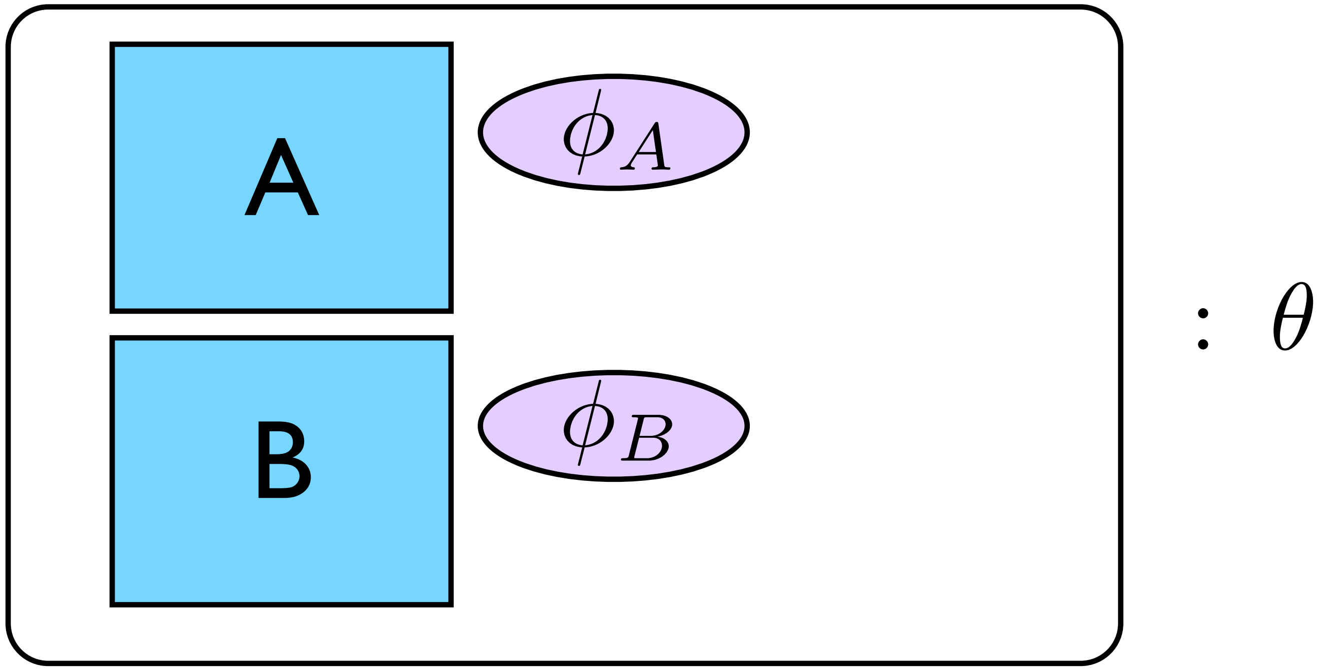


•Abstract

•Prohibiting illegal function calls sequences

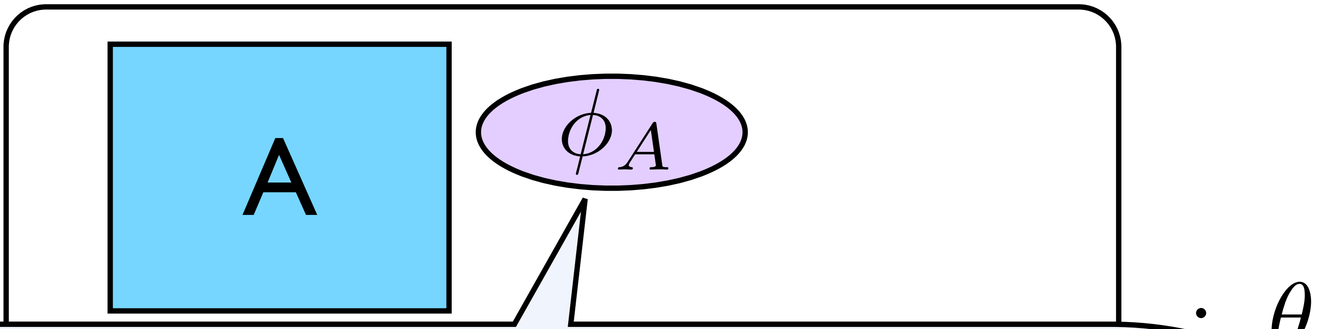
CVPP

- Specify modules
- Verify specs locally
- Construct maximal models from local specs
- Compose Max model
- Model check global property



CVPP

- Specify modules
- Verify specs locally
- Construct ma
  models fr
  specs
- Compo
- Model che
  property

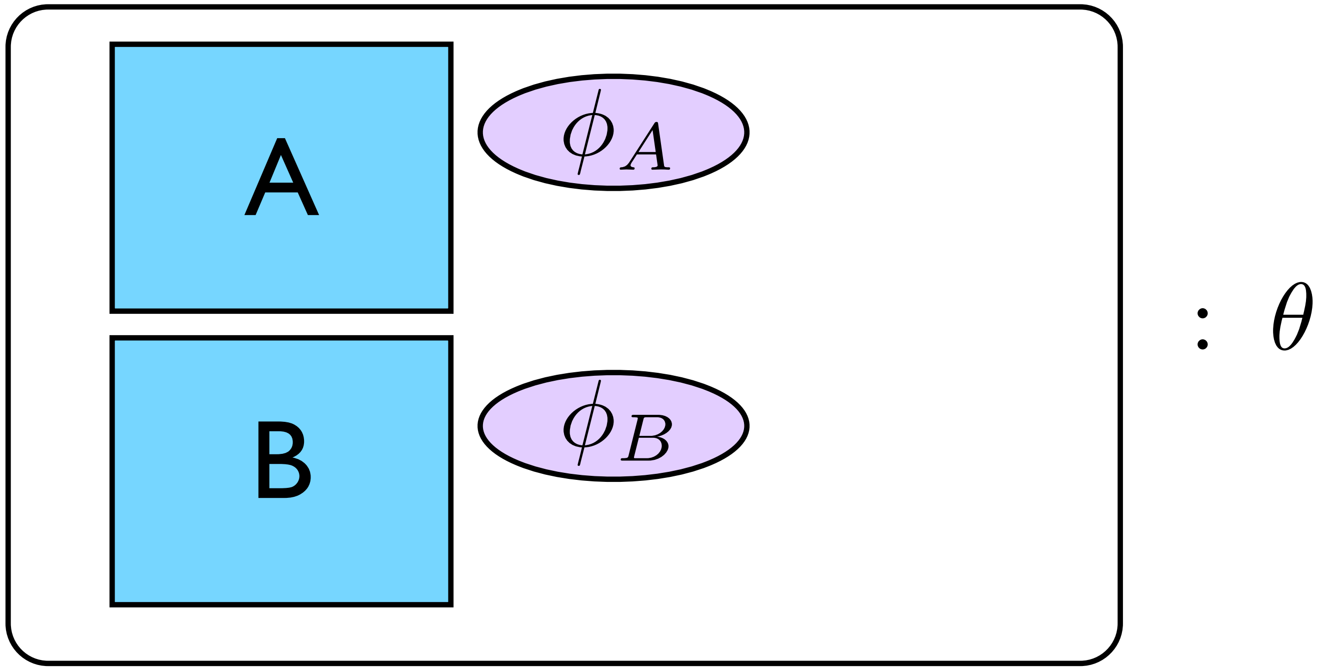


Simulation Logic

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

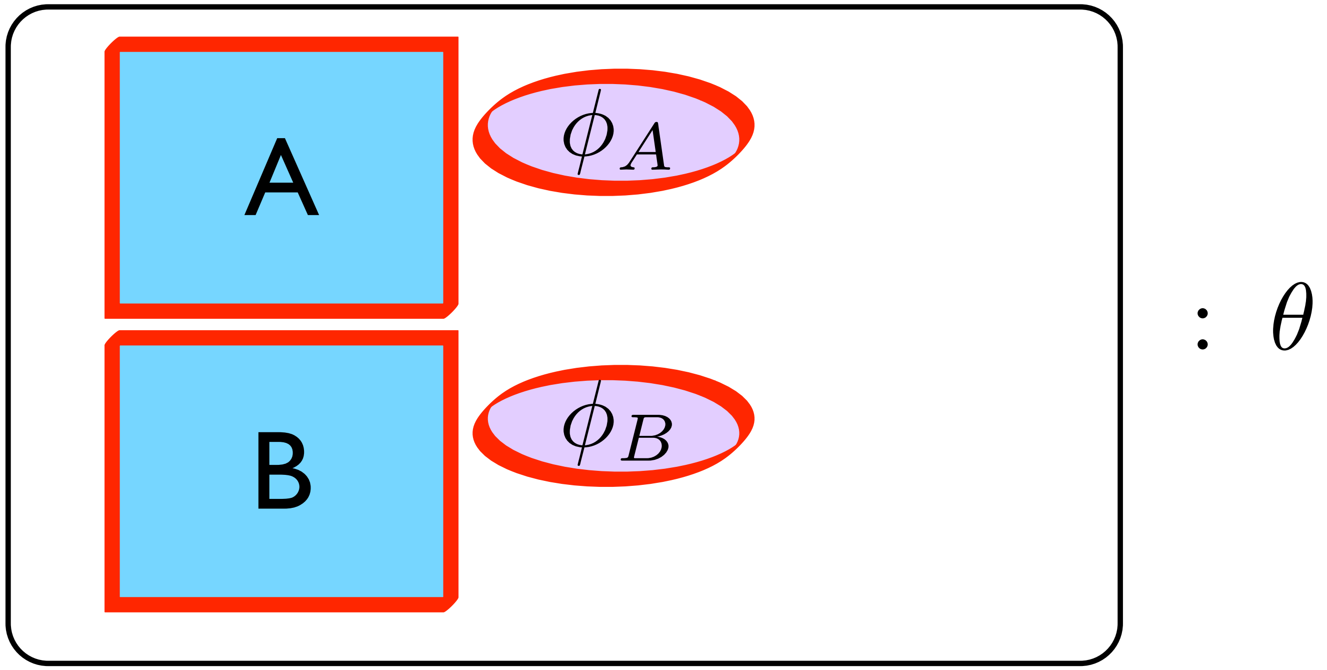
CVPP

- Specify modules
- Verify specs locally
- Construct maximal models from local specs
- Compose Max model
- Model check global property



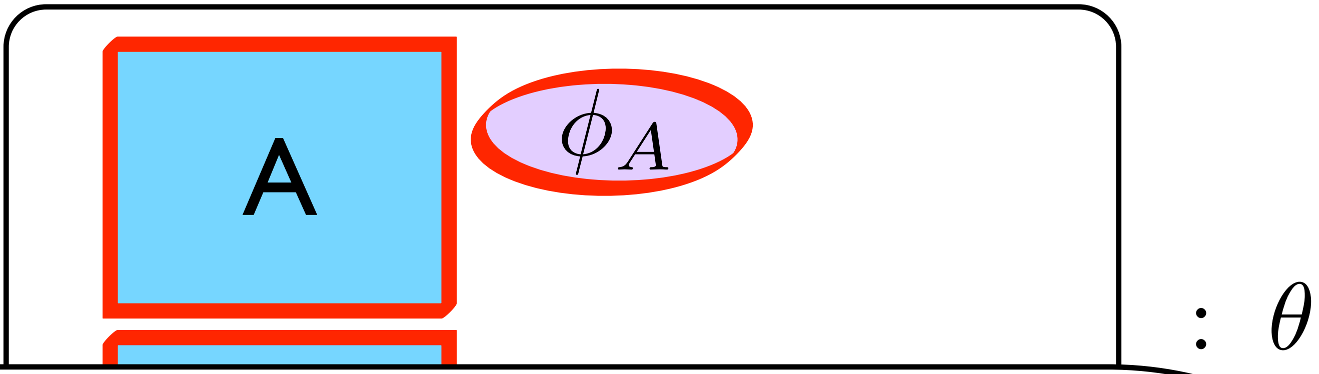
CVPP

- Specify modules
- Verify specs locally
- Construct maximal models from local specs
- Compose Max model
- Model check global property



CVPP

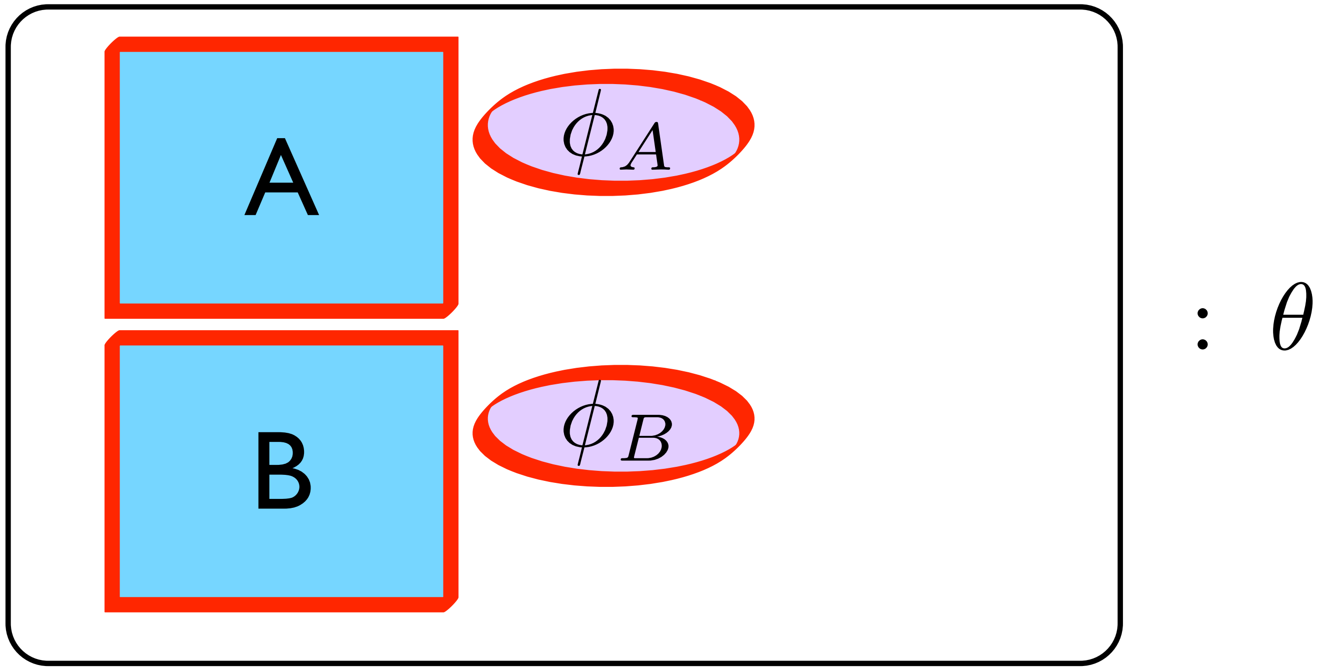
- Specify modules
- Verify specs locally
- Construct maximal models fro specs
- Compo
- Model ch property



- Extract **Flow Graphs** from module code
  - -Finite-State transition system
  - -Abstract away all program data
  - -Program structure
- Employ standard model checking for verification

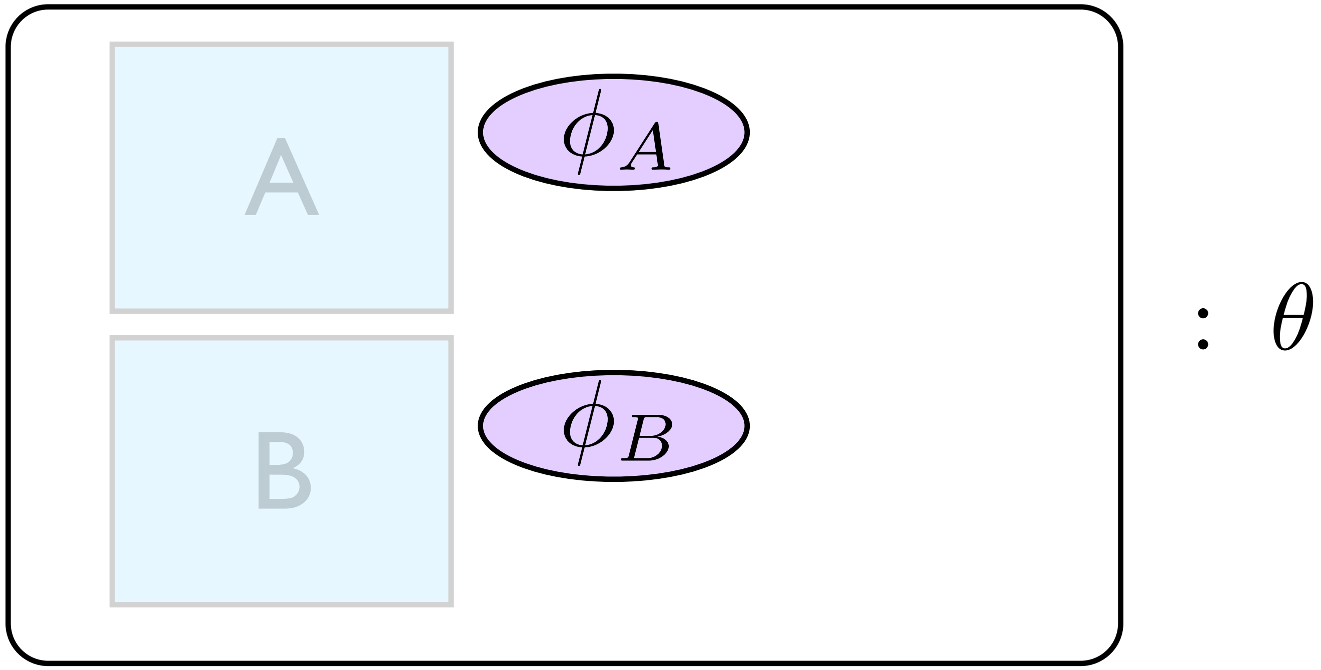
CVPP

- Specify modules
- Verify specs locally
- Construct maximal models from local specs
- Compose Max model
- Model check global property



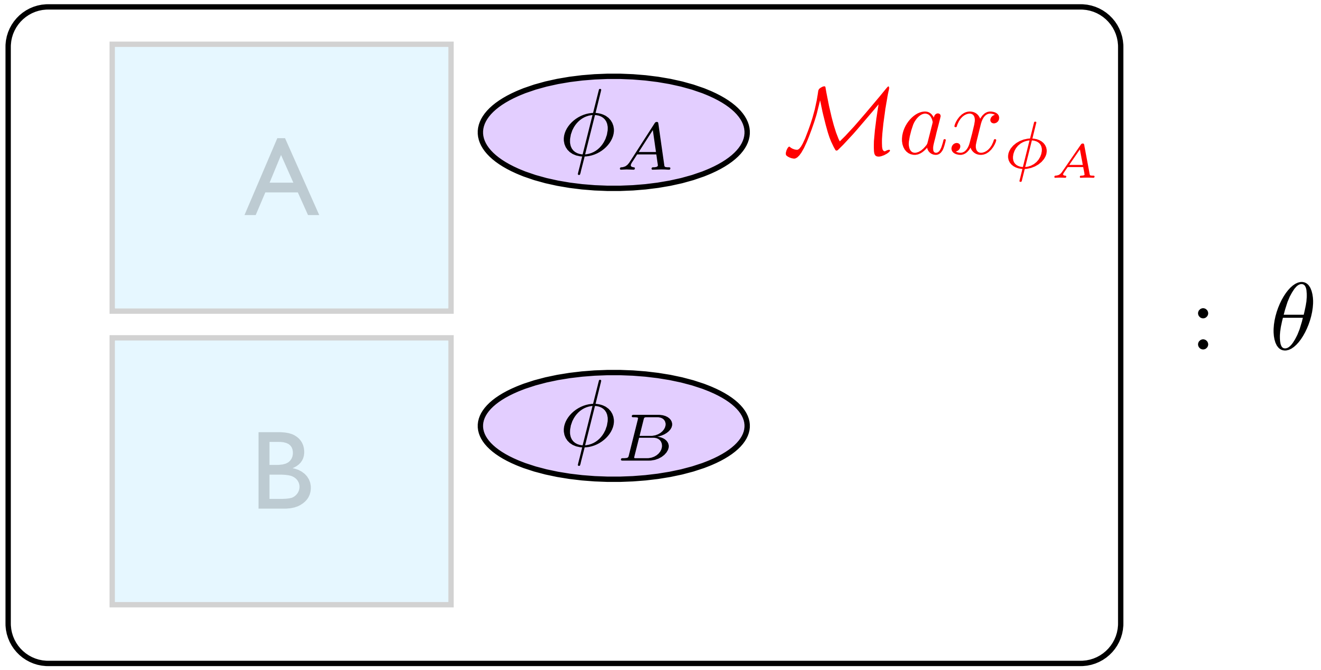
CVPP

- Specify modules
- Verify specs locally
- Construct maximal models from local specs
- Compose Max model
- Model check global property



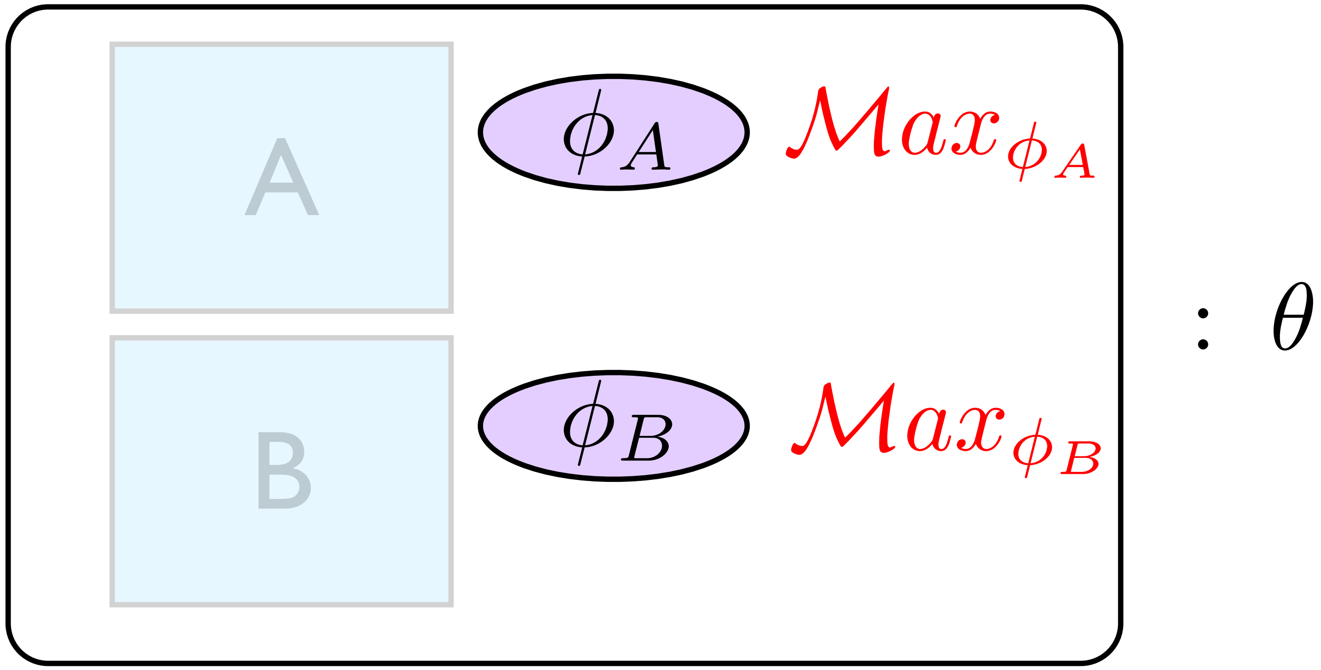
CVPP

- Specify modules
- Verify specs locally
- Construct maximal models from local specs
- Compose Max model
- Model check global property



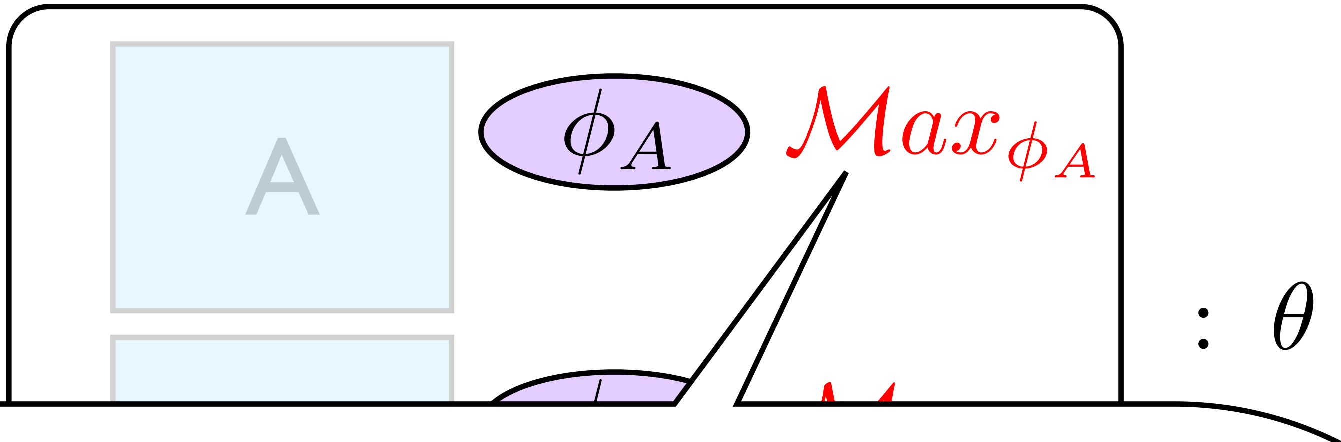
CVPP

- Specify modules
- Verify specs locally
- Construct maximal models from local specs
- Compose Max model
- Model check global property



CVPP

- Specify modules
- Verify specs locally
- Construct maximal models from lo specs
- Compose
- Model che property



•**Flow Graph** of property ϕ_A

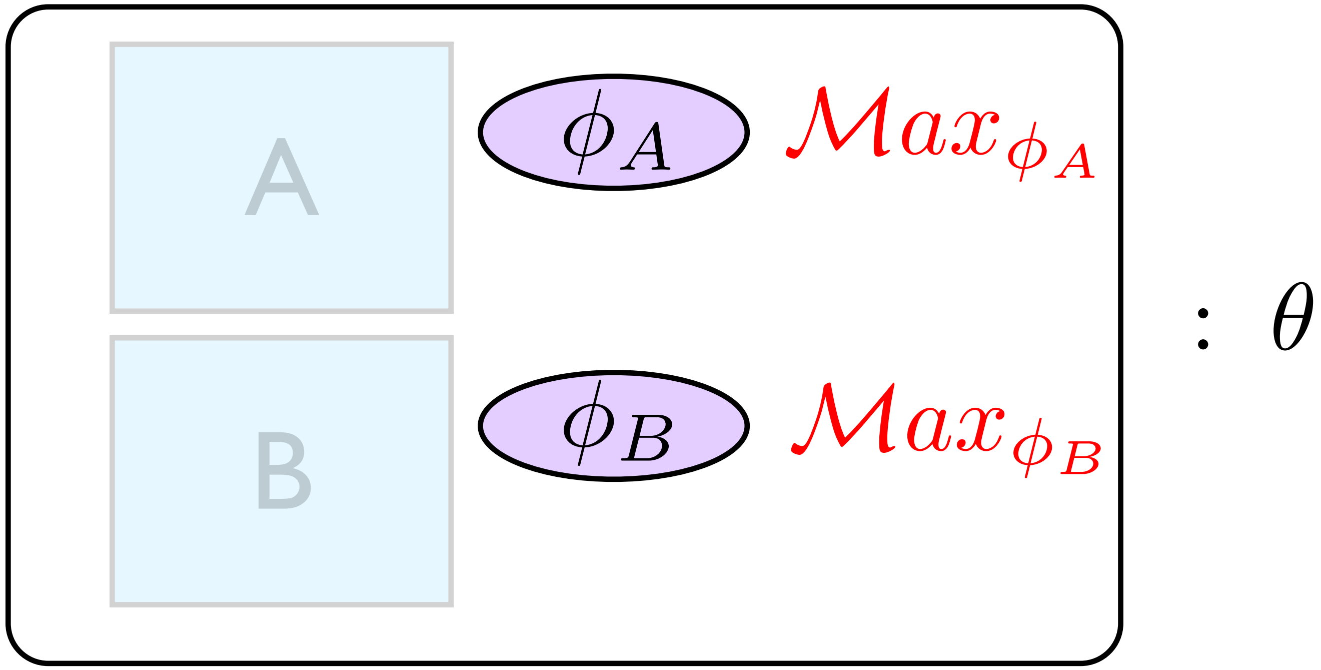
-Simulates all flow graphs satisfying ϕ_A

-Program structure

-Finite-State transition system

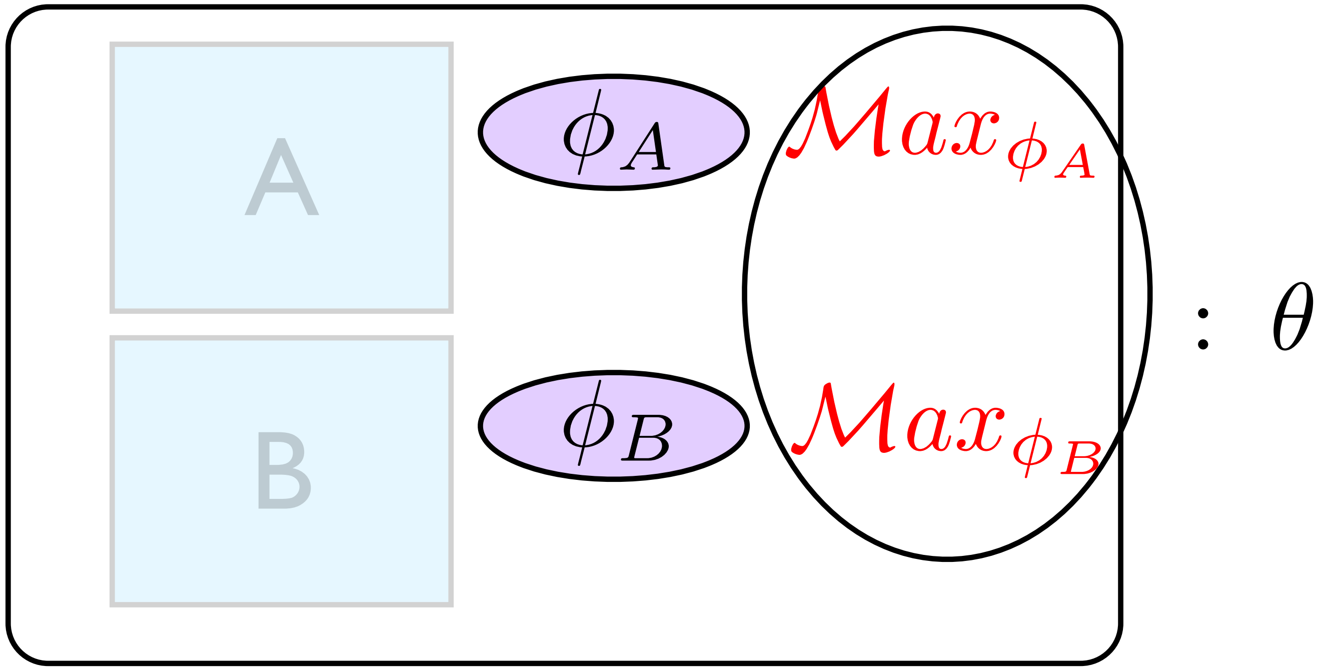
CVPP

- Specify modules
- Verify specs locally
- Construct maximal models from local specs
- Compose Max model
- Model check global property



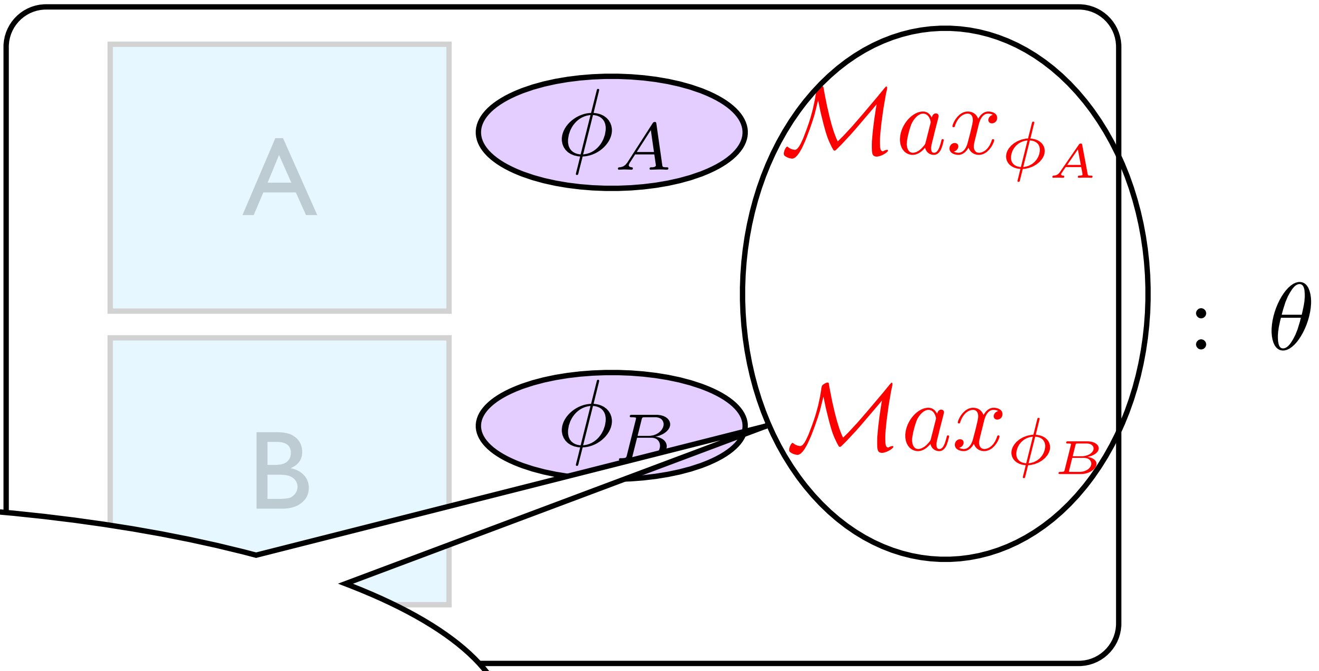
CVPP

- Specify modules
- Verify specs locally
- Construct maximal models from local specs
- Compose Max model
- Model check global property



CVPP

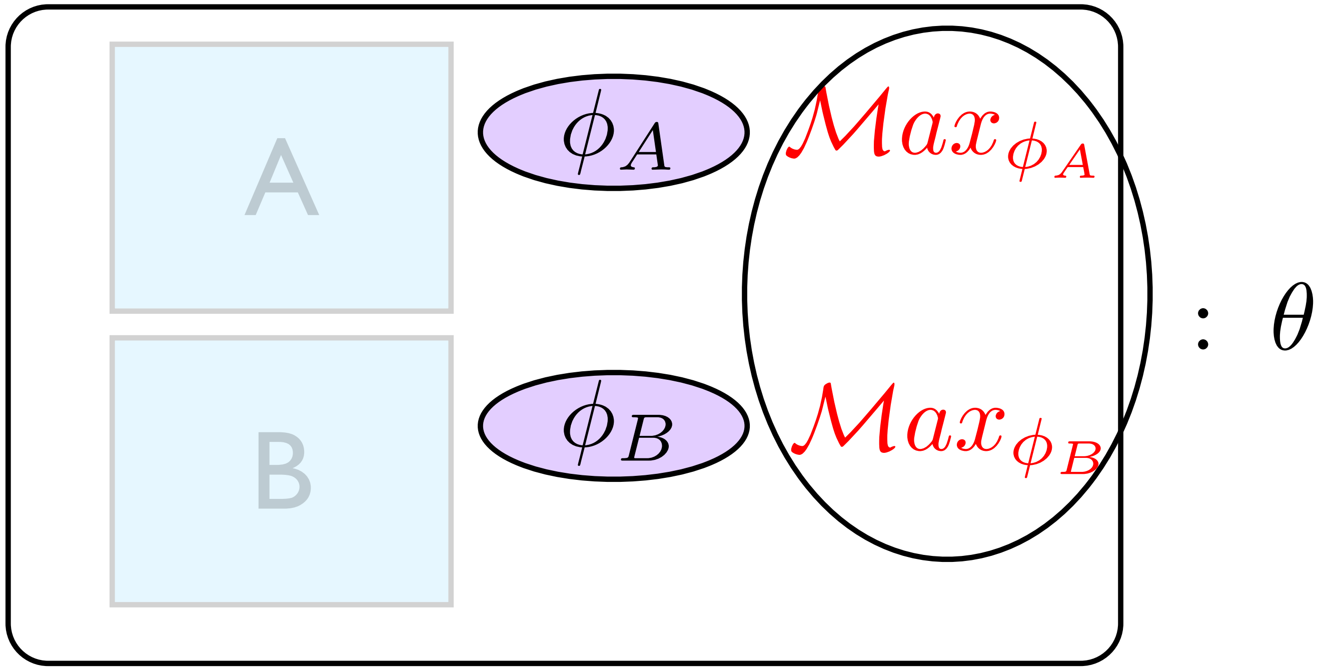
- Specify modules
- Verify specs locally
- Construct maximal models from local specs
- C
- Model check global property



Push Down Automata

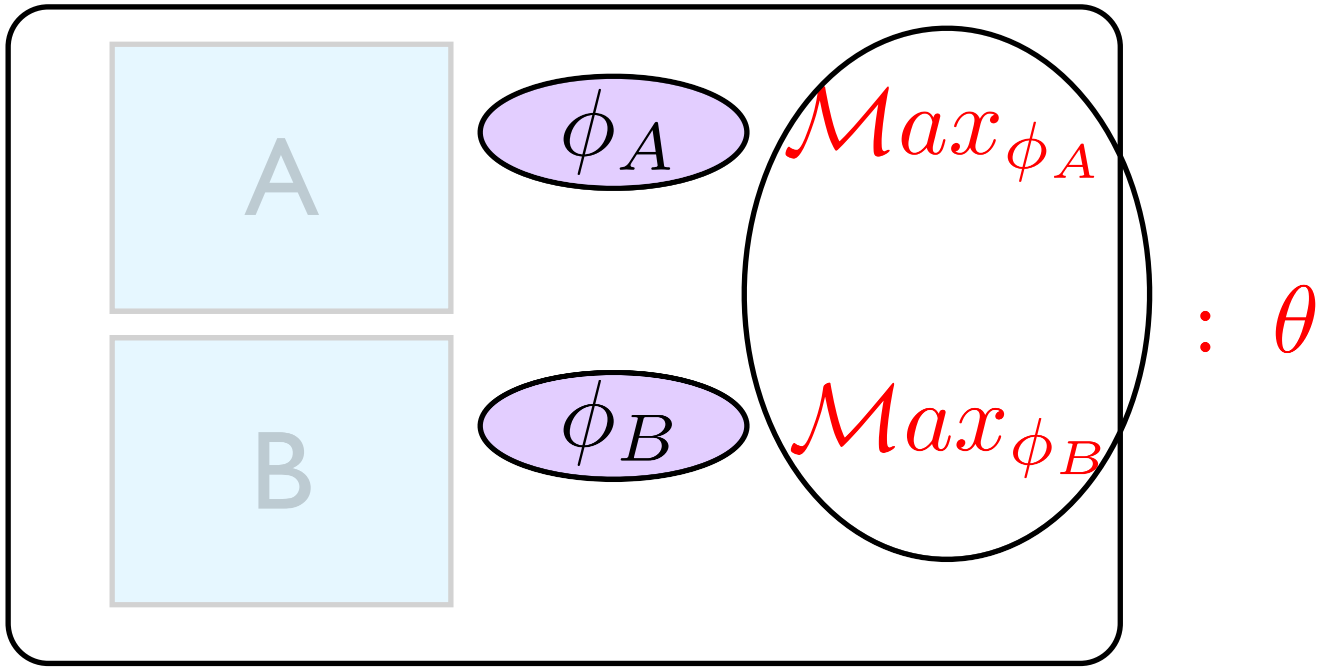
CVPP

- Specify modules
- Verify specs locally
- Construct maximal models from local specs
- Compose Max model
- Model check global property



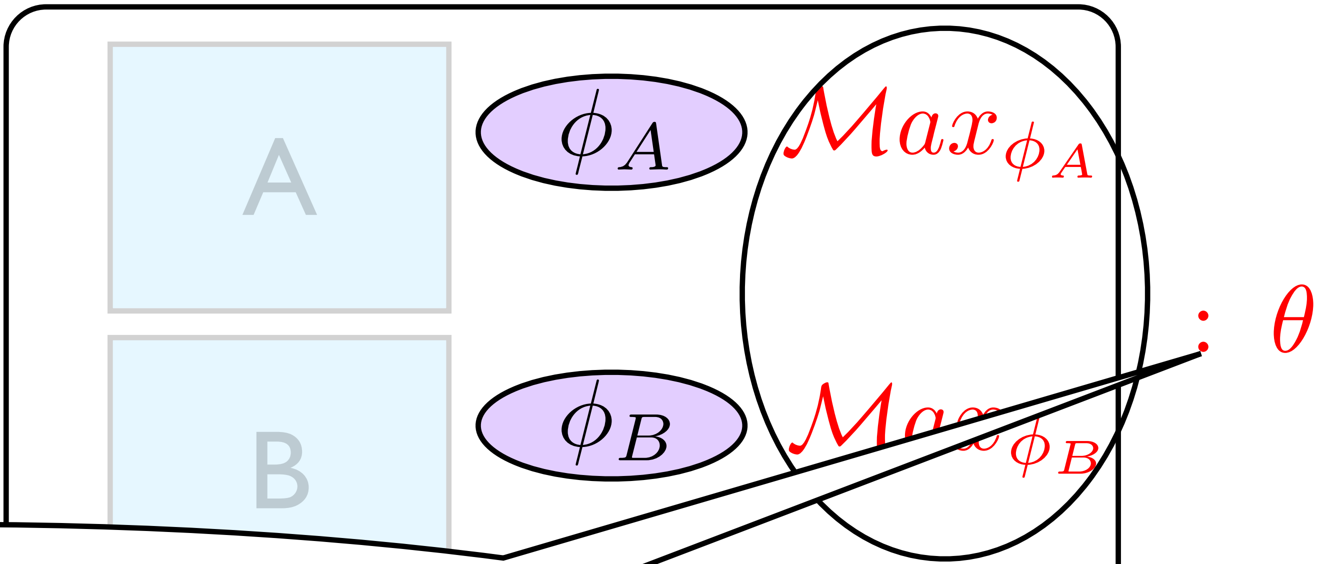
CVPP

- Specify modules
- Verify specs locally
- Construct maximal models from local specs
- Compose Max model
- Model check global property



CVPP

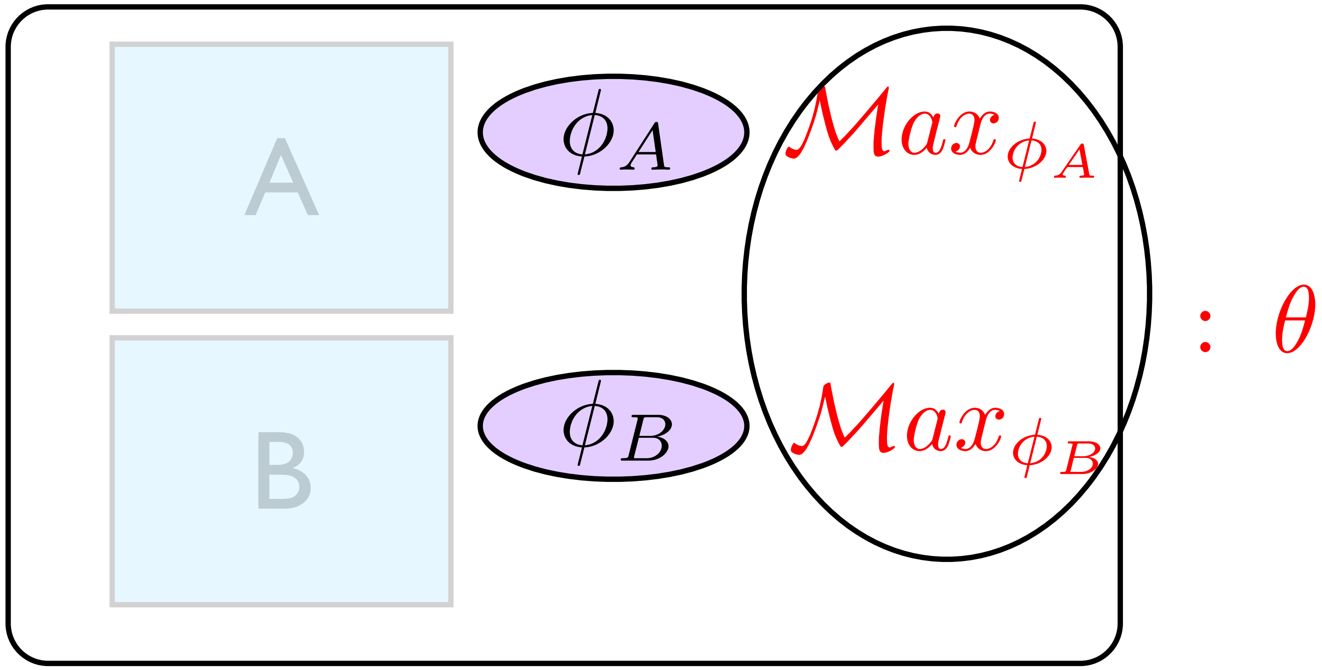
- Specify modules
- Verify specs locally
- Construct maximal models from local specs
- Model c
  property



Employ PDA/PDS model checking, Moped

CVPP

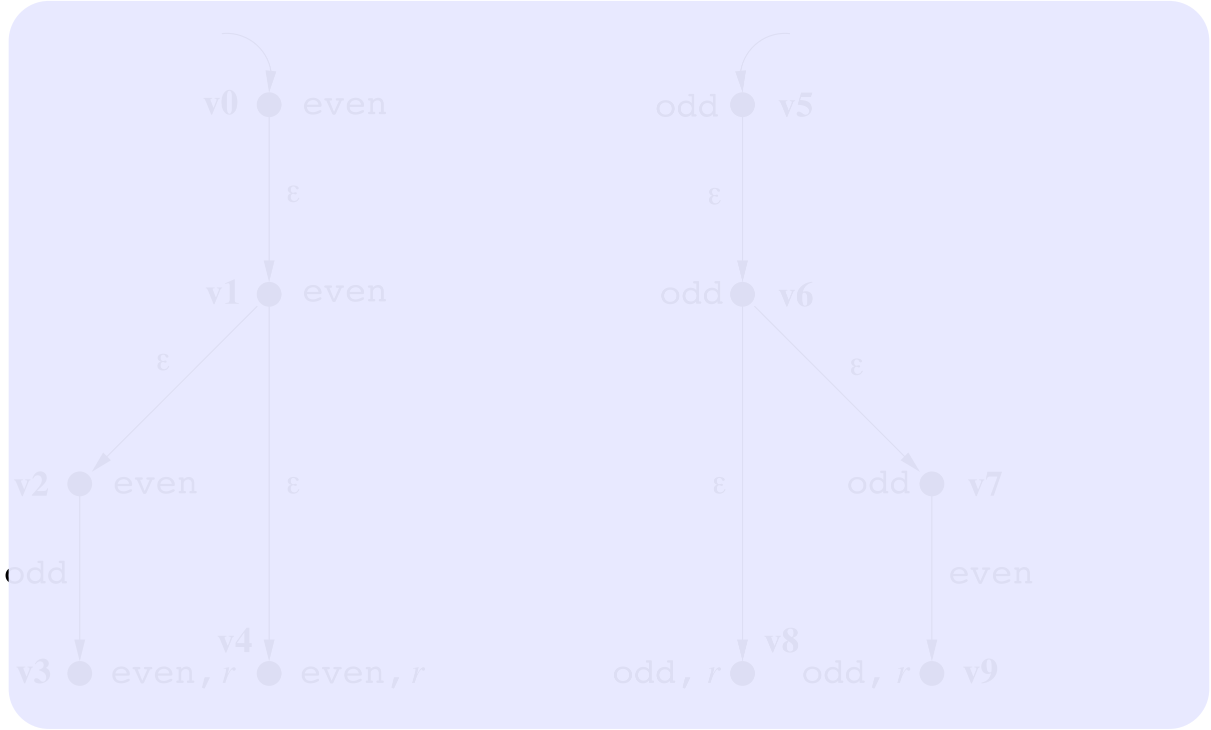
- Specify modules
- Verify specs locally
- Construct maximal models from local specs
- Compose Max model
- Model check global property



CVPP -- Program Model

Flow Graph:

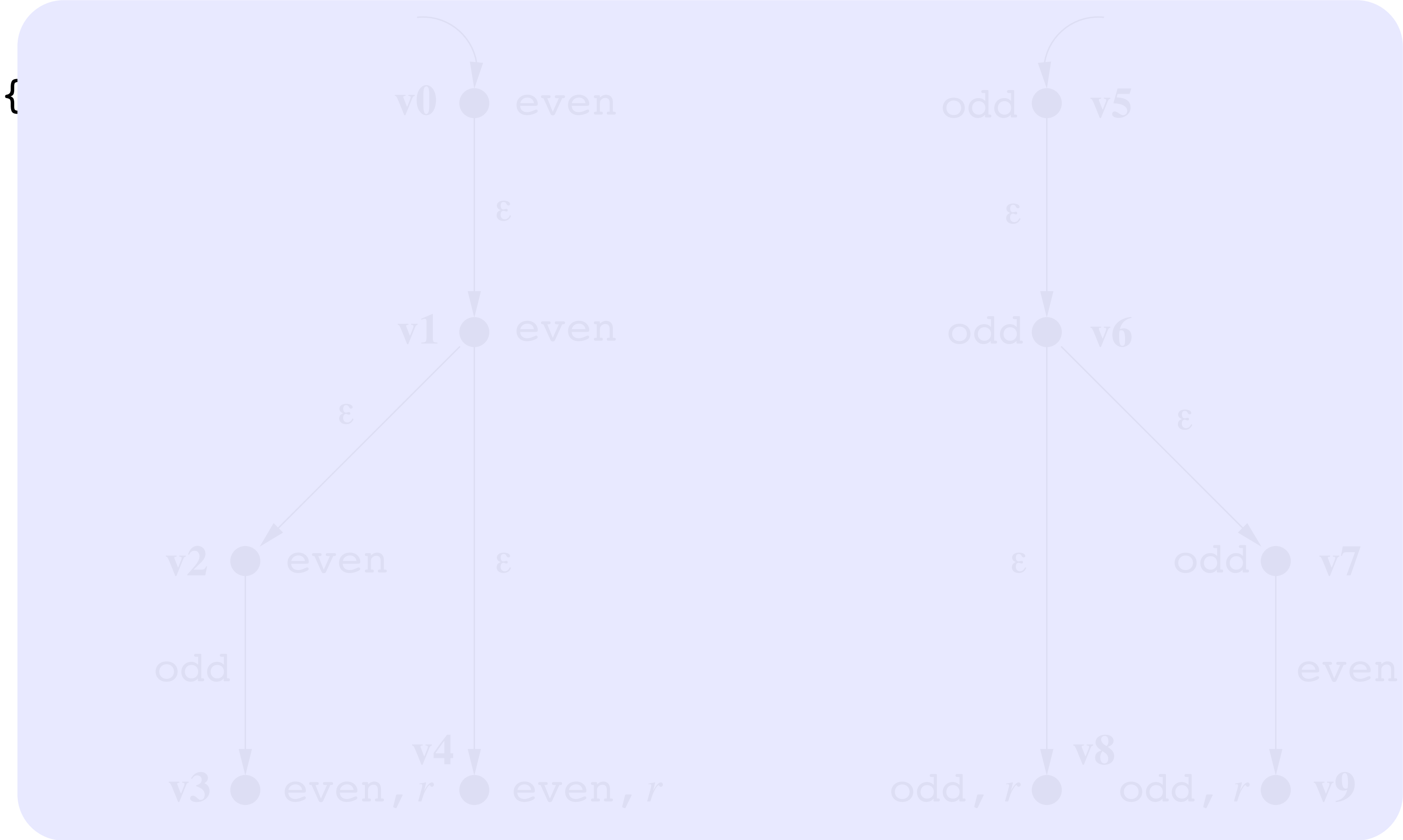
```
class Number {

   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);

   }
}
```



CVPP -- Program Model

Flow Graph:

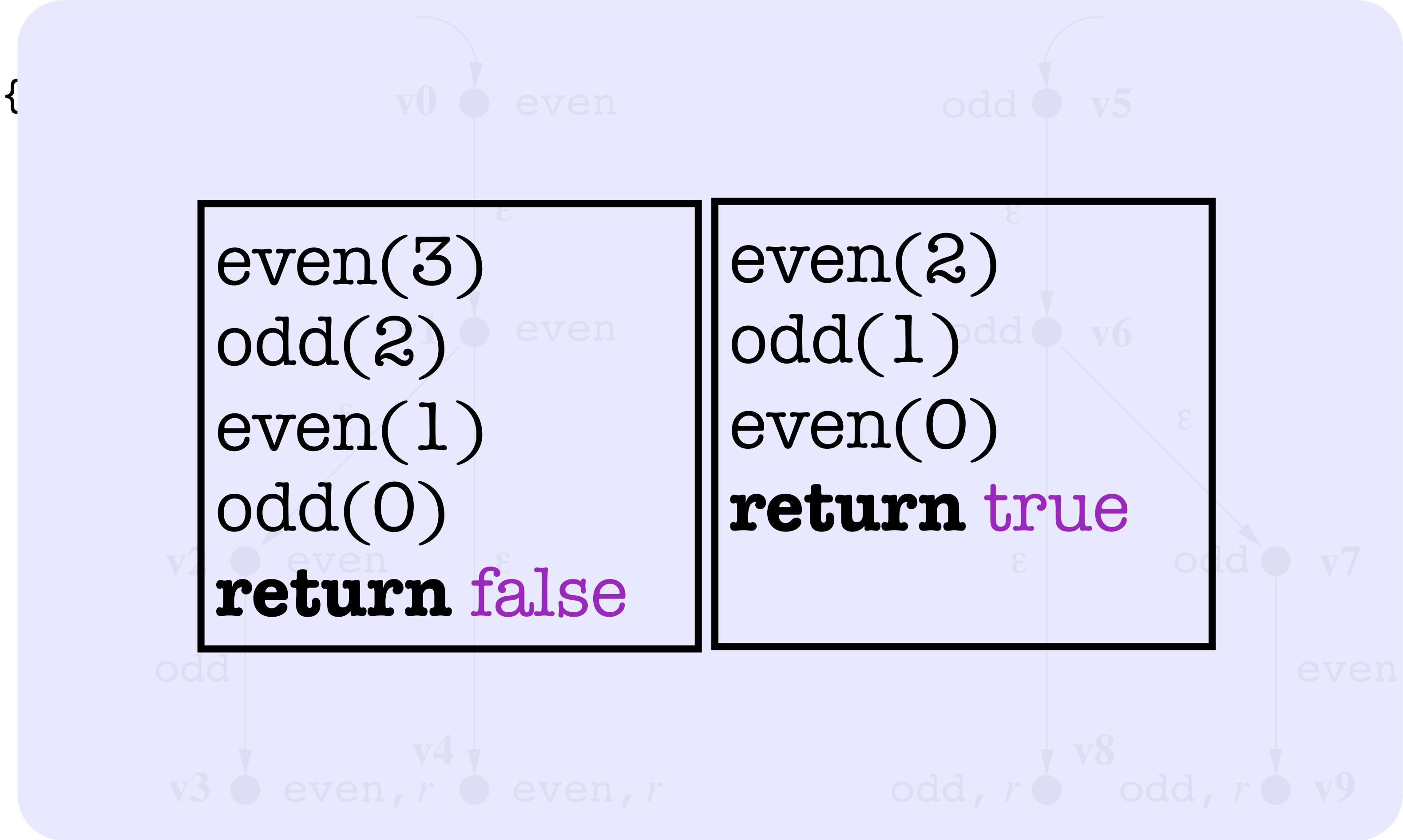
```
class Number {

   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);

   }
}
```



CVPP -- 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);

   }
}
```



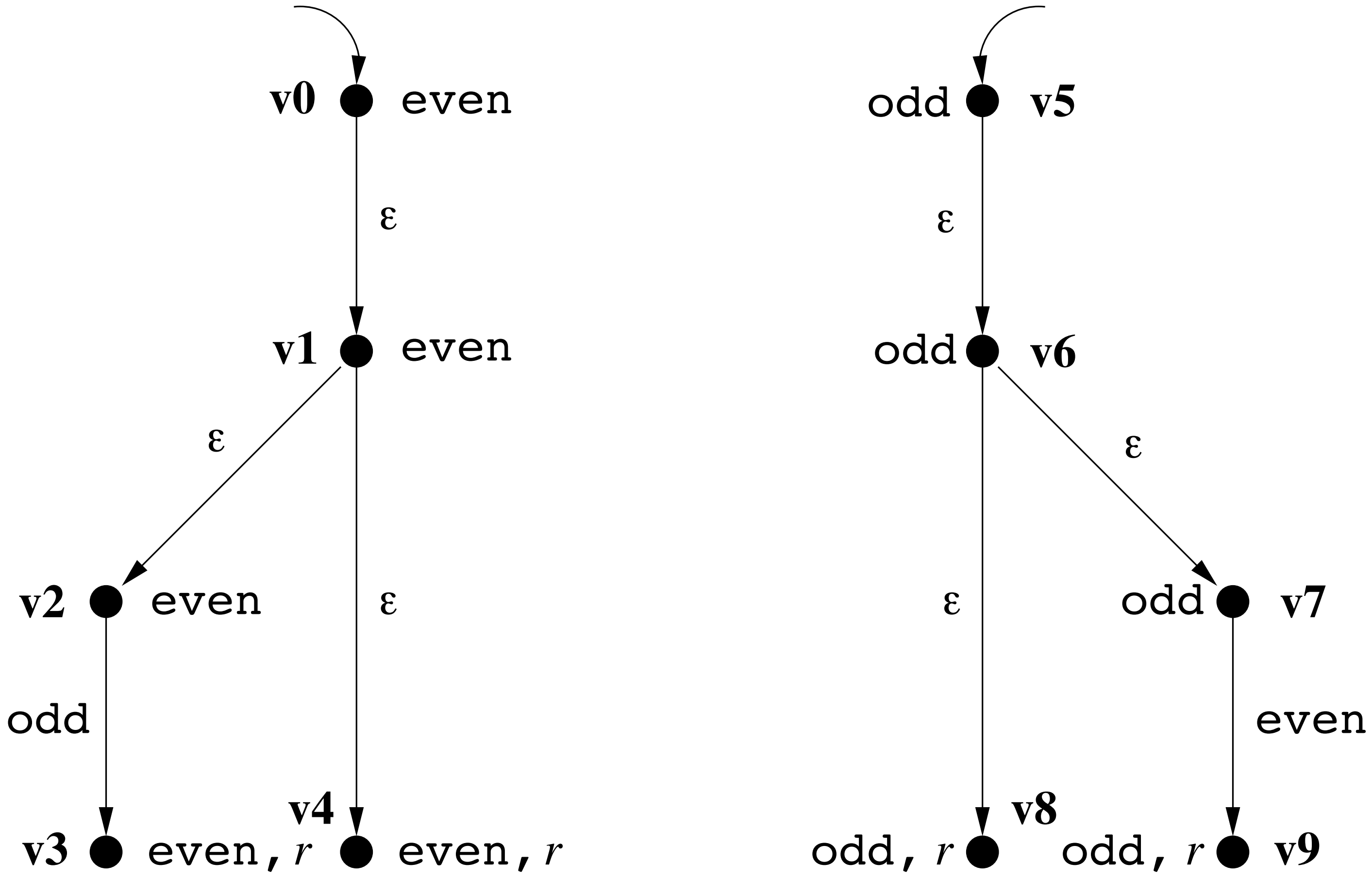
even(3)
odd(2)
even(1)
odd(0)
return false

even(2)
odd(1)
even(0)
return true

CVPP -- Program Model

Flow Graph:

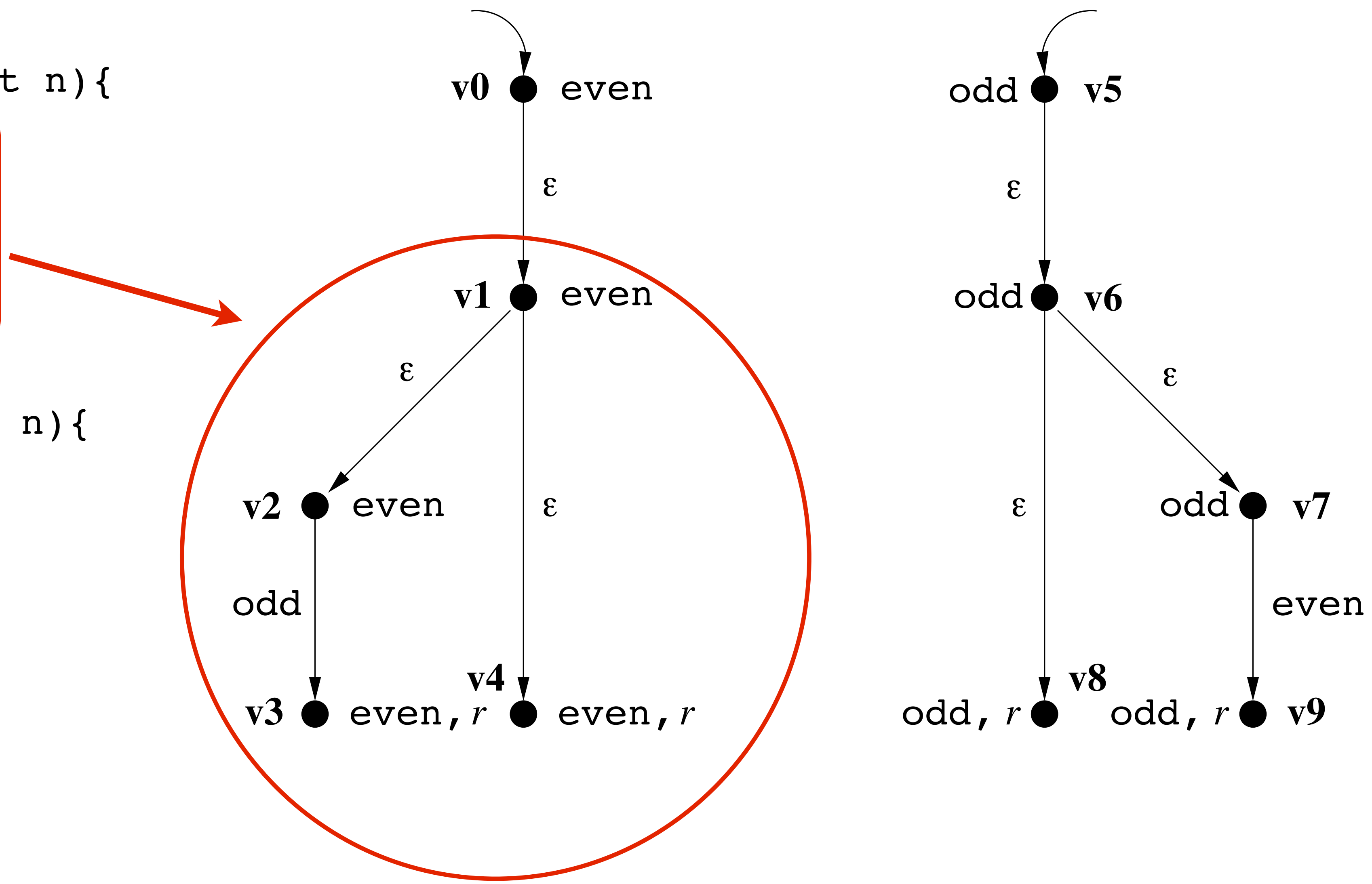
```
class Number {

   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);

   }
}
```



CVPP -- Program Model

Flow Graph:

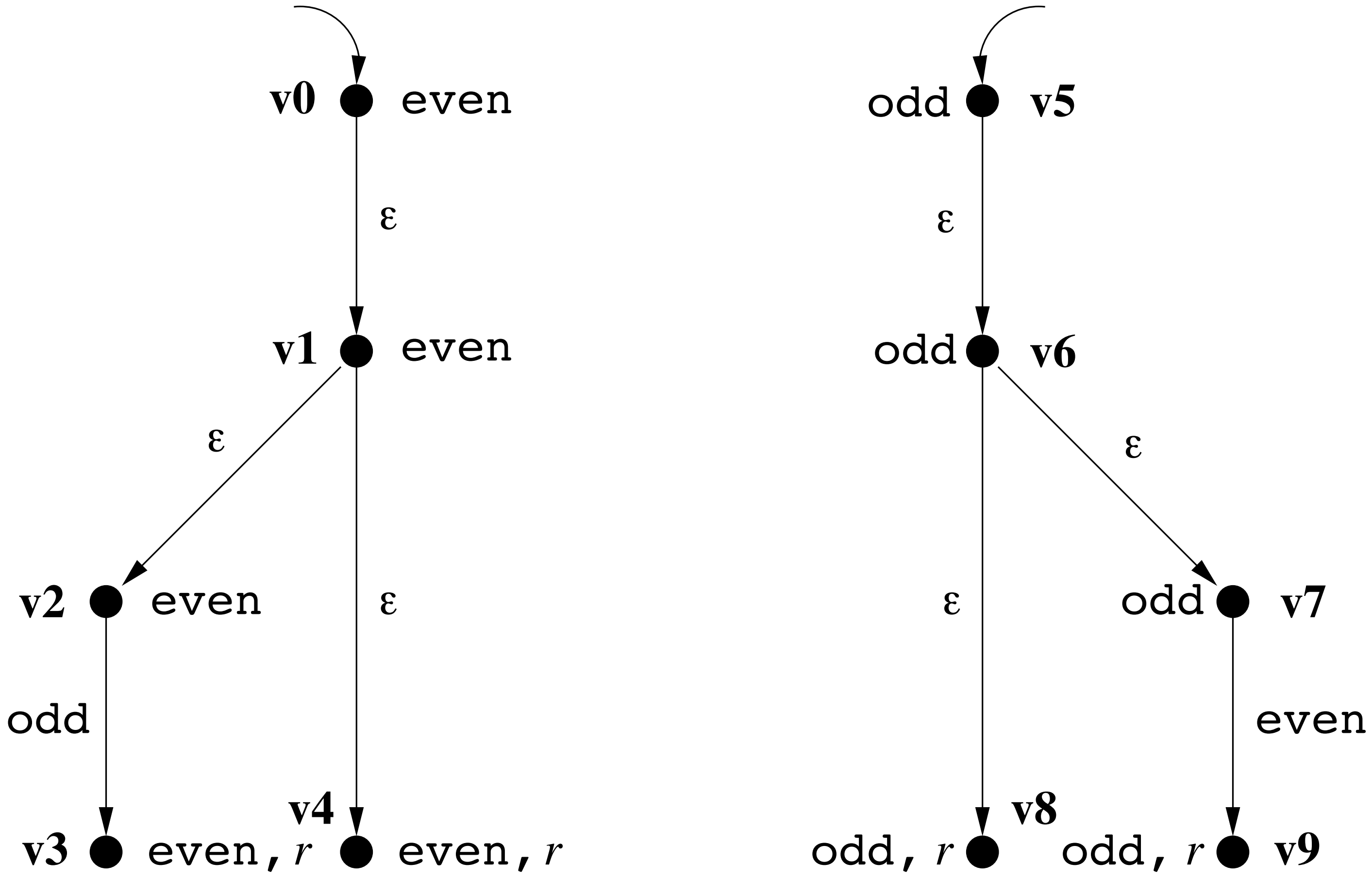
```
class Number {

   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);

   }
}
```



CVPP -- Program Model

Flow Graph:

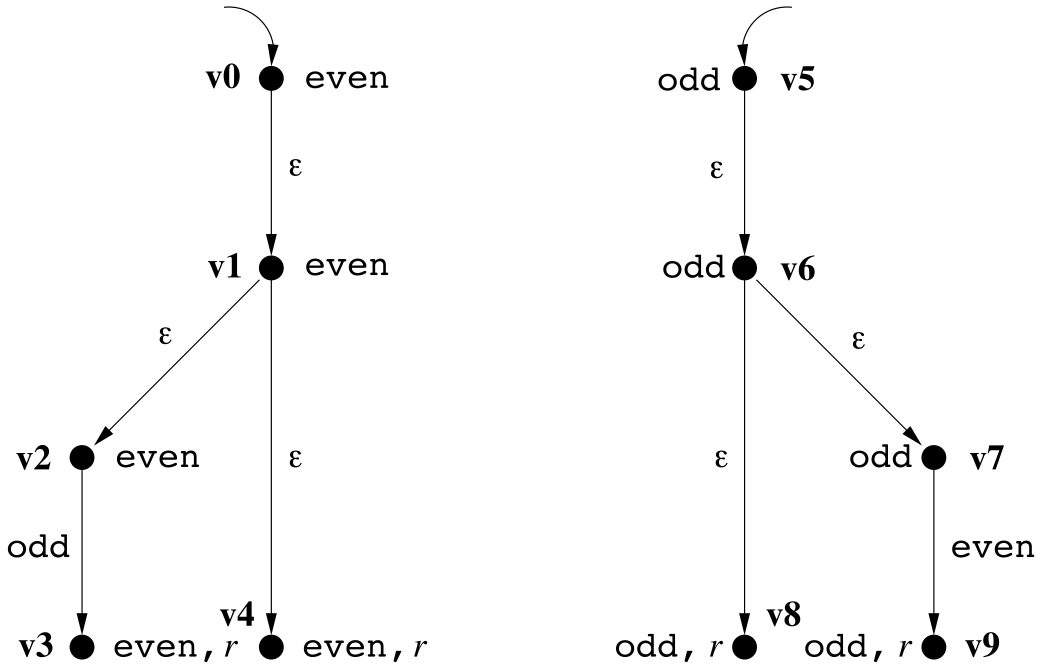
```
class Number {

   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);

   }
}
```



CVPP -- Program Model

Flow Graph:

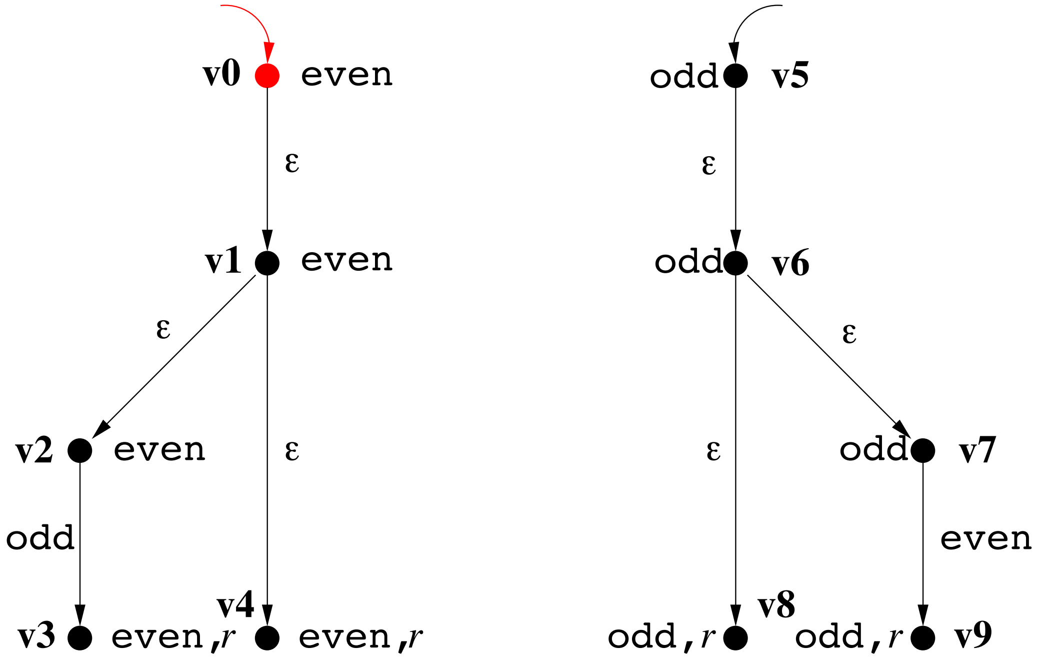
```
class Number {

   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);

   }
}
```



CVPP -- 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);

   }
}
```



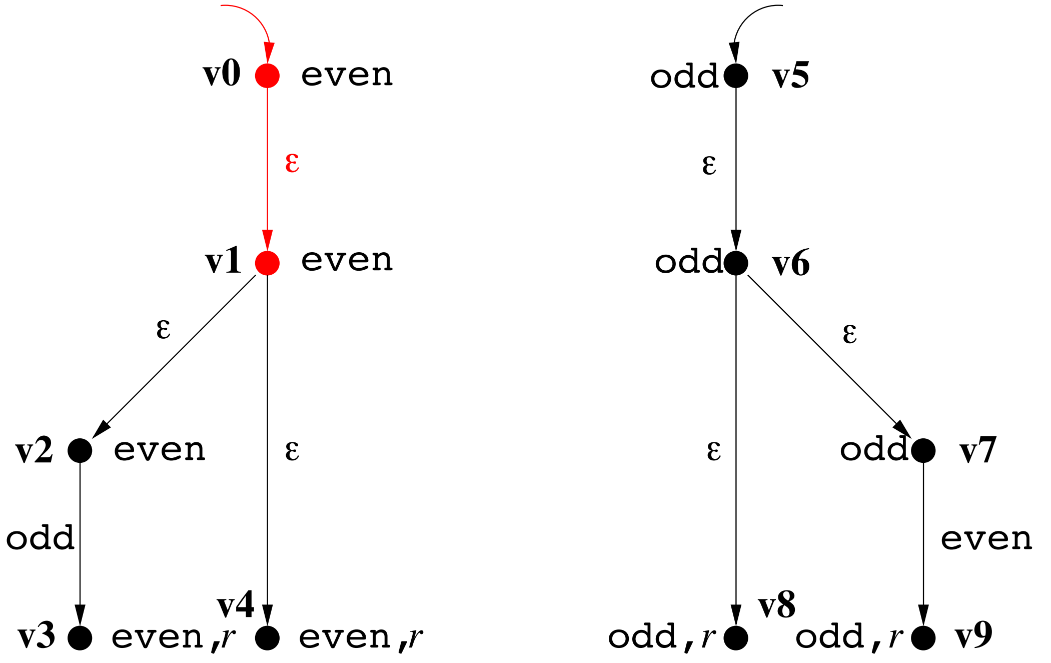
Example Run:

(v_0, ε)

CVPP -- 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);

   }
}
```



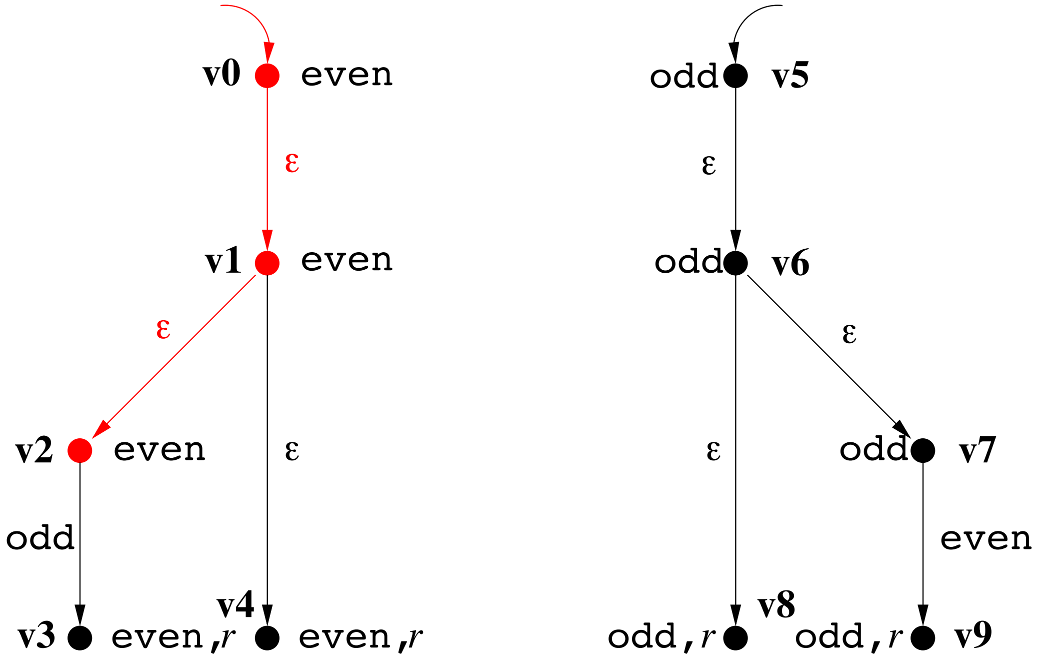
Example Run:

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

CVPP -- 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);

   }
}
```



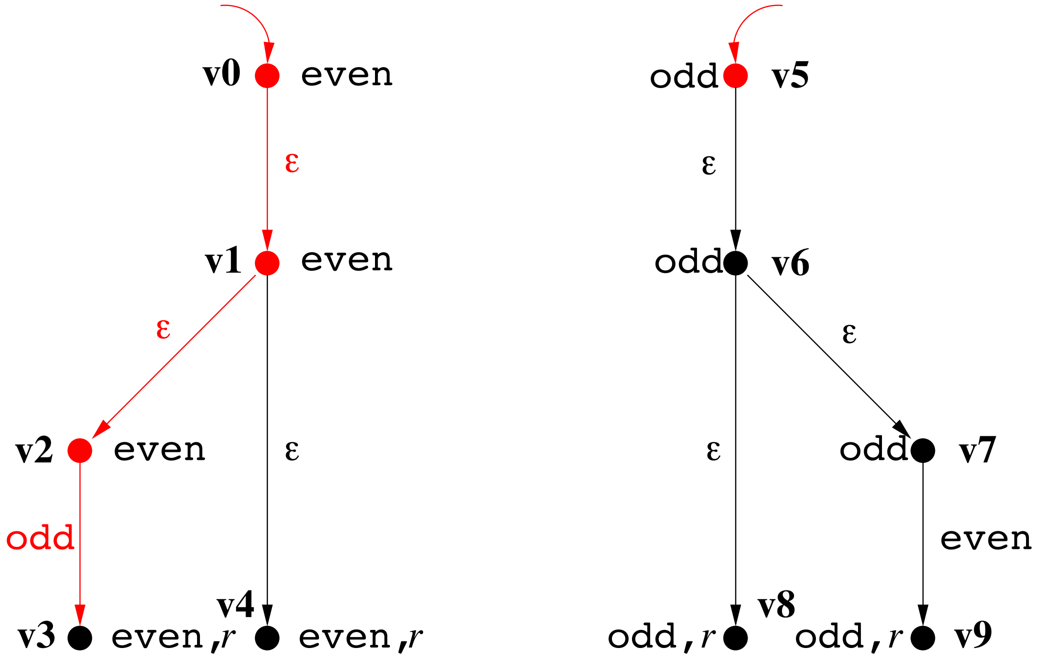
Example Run:

$$(v_0,\varepsilon)\xrightarrow{\tau}(v_1,\varepsilon)\xrightarrow{\tau}(v_2,\varepsilon)$$

CVPP -- 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);

   }
}
```



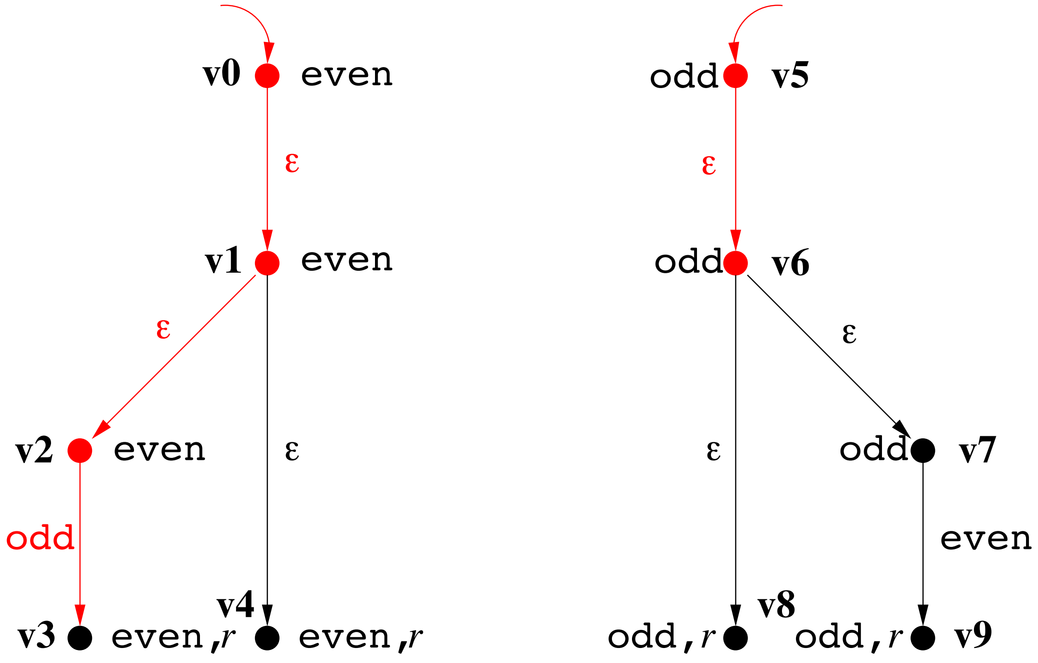
Example Run:

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

CVPP -- 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);

   }
}
```



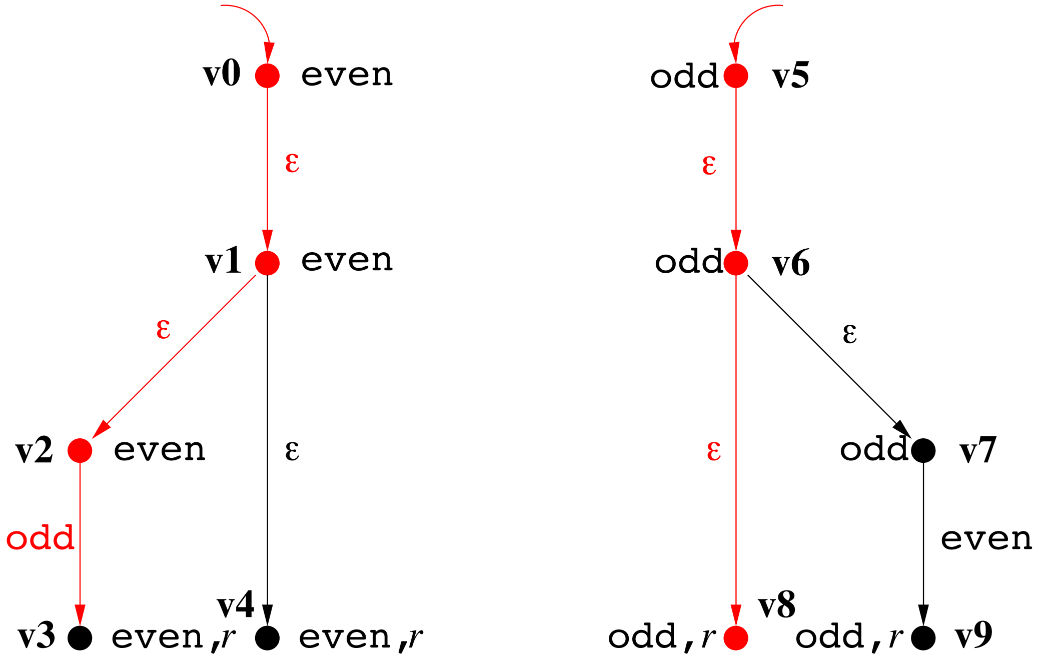
Example Run:

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

CVPP -- 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);

   }
}
```



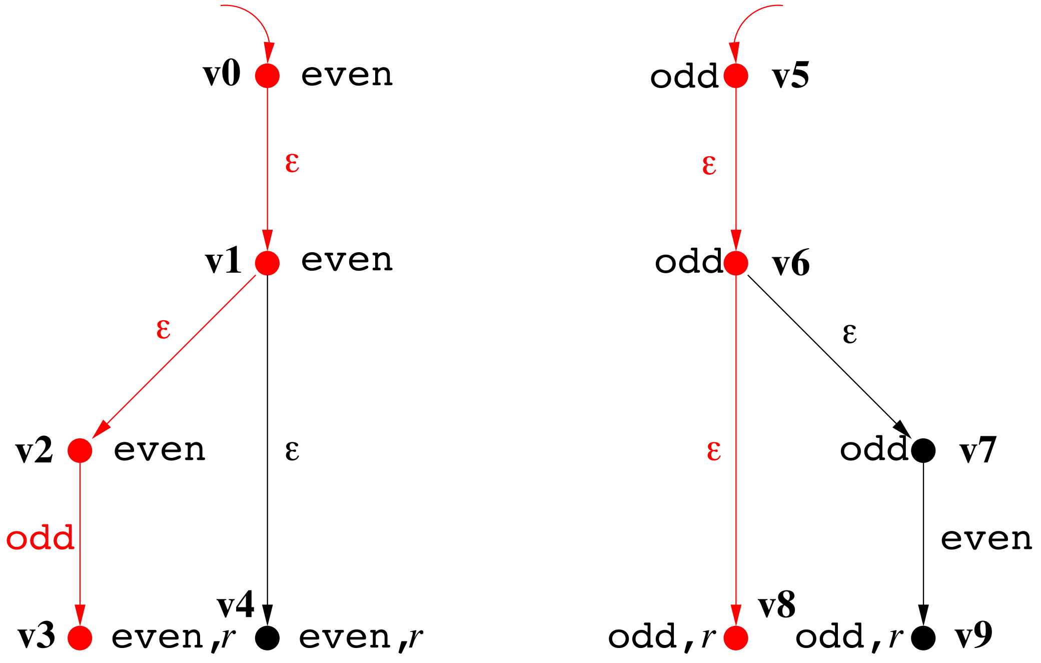
Example Run:

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

CVPP -- 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:

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

$$(v_8, v_3) \xrightarrow{\text{odd ret even}} (v_3, \varepsilon)$$

CVPP -- Maximal Models

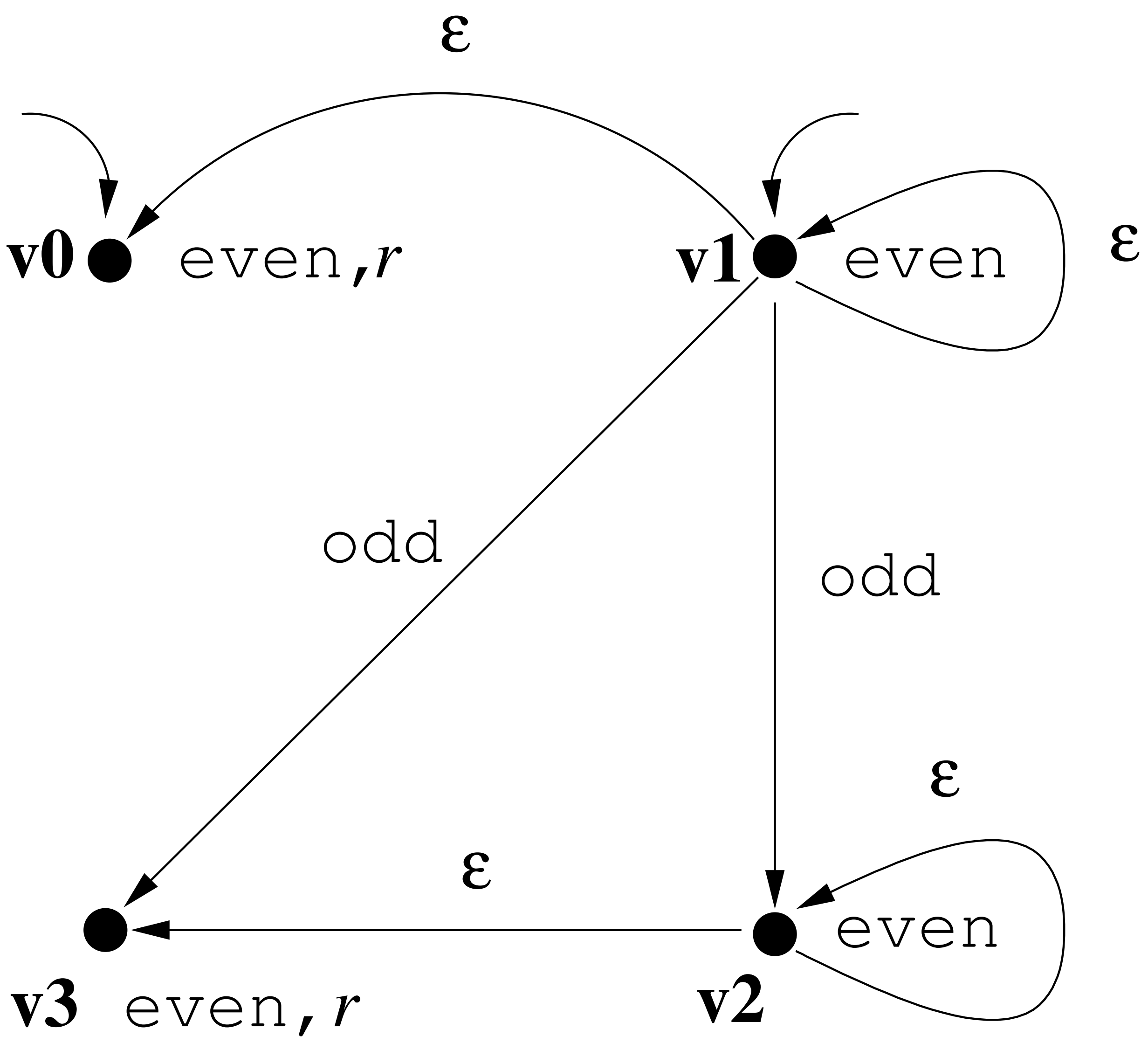
Local Specification for method even:

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

CVPP -- Maximal Models

Local Specification for method even:

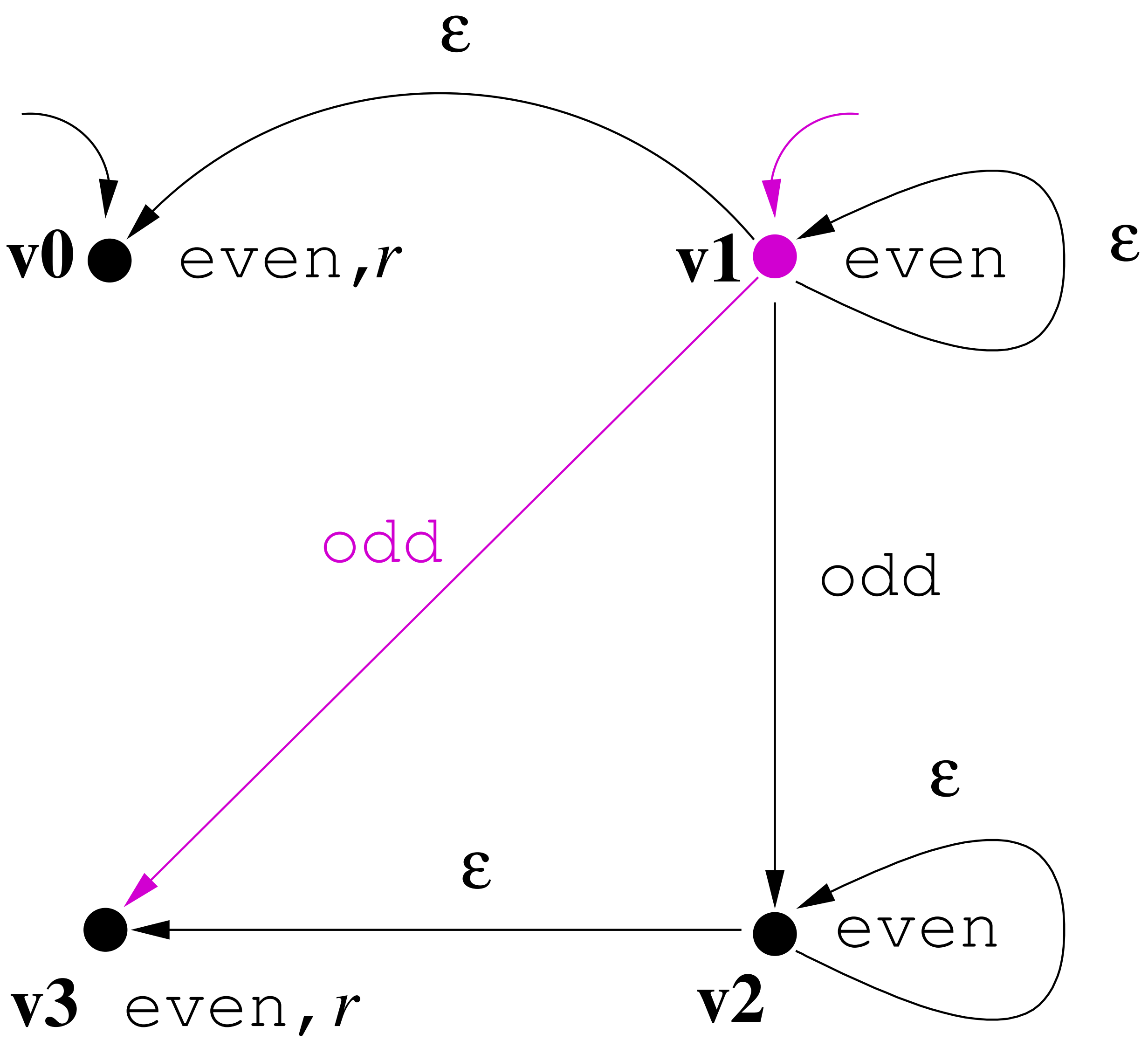
method even can only call method odd, and after returning from the call, no other method can be called



CVPP -- Maximal Models

Local Specification for method even:

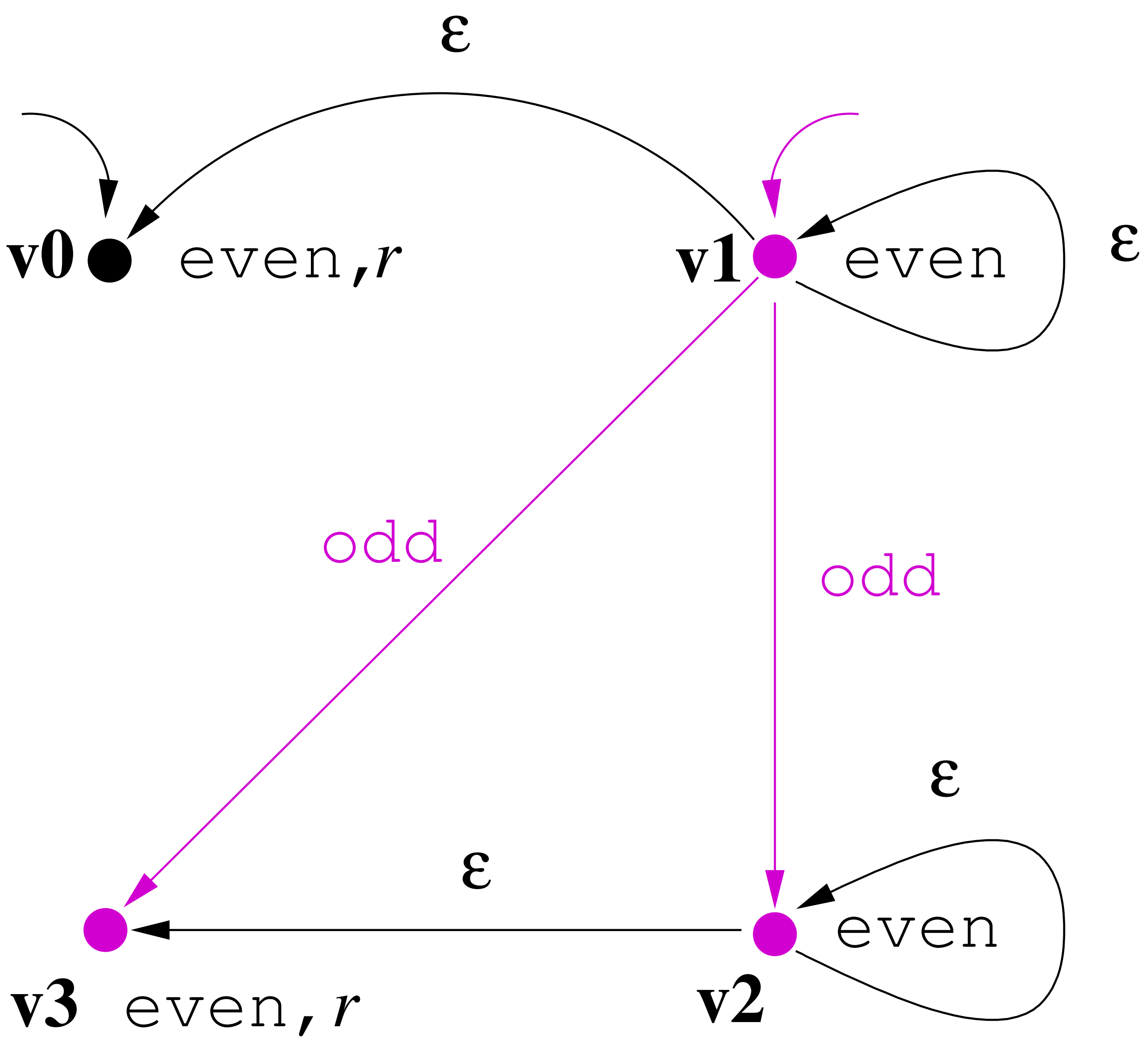
method even can only call method odd, and after returning from the call, no other method can be called



CVPP -- Maximal Models

Local Specification for method even:

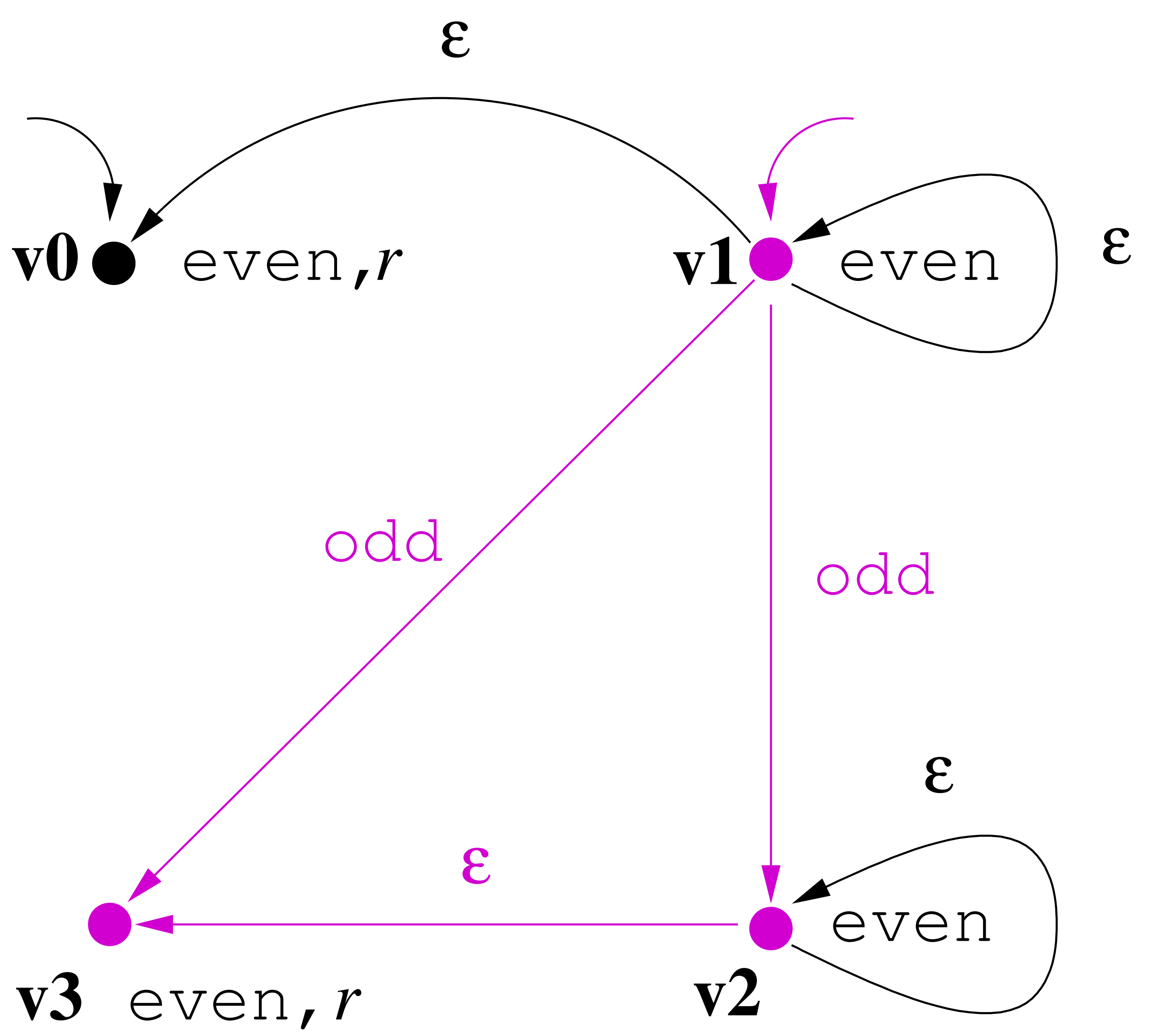
method even can only call method odd, and after returning from the call, no other method can be called



CVPP -- Maximal Models

Local Specification for method even:

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



Contributions

Contributions

- Full automation of the usage of CVPP
  - ProMoVer: procedure-modular verification
  - Annotation language
- Enhancing the usability
  - Different specification languages
  - Specification extraction
  - Proof storage and reuse

Contributions

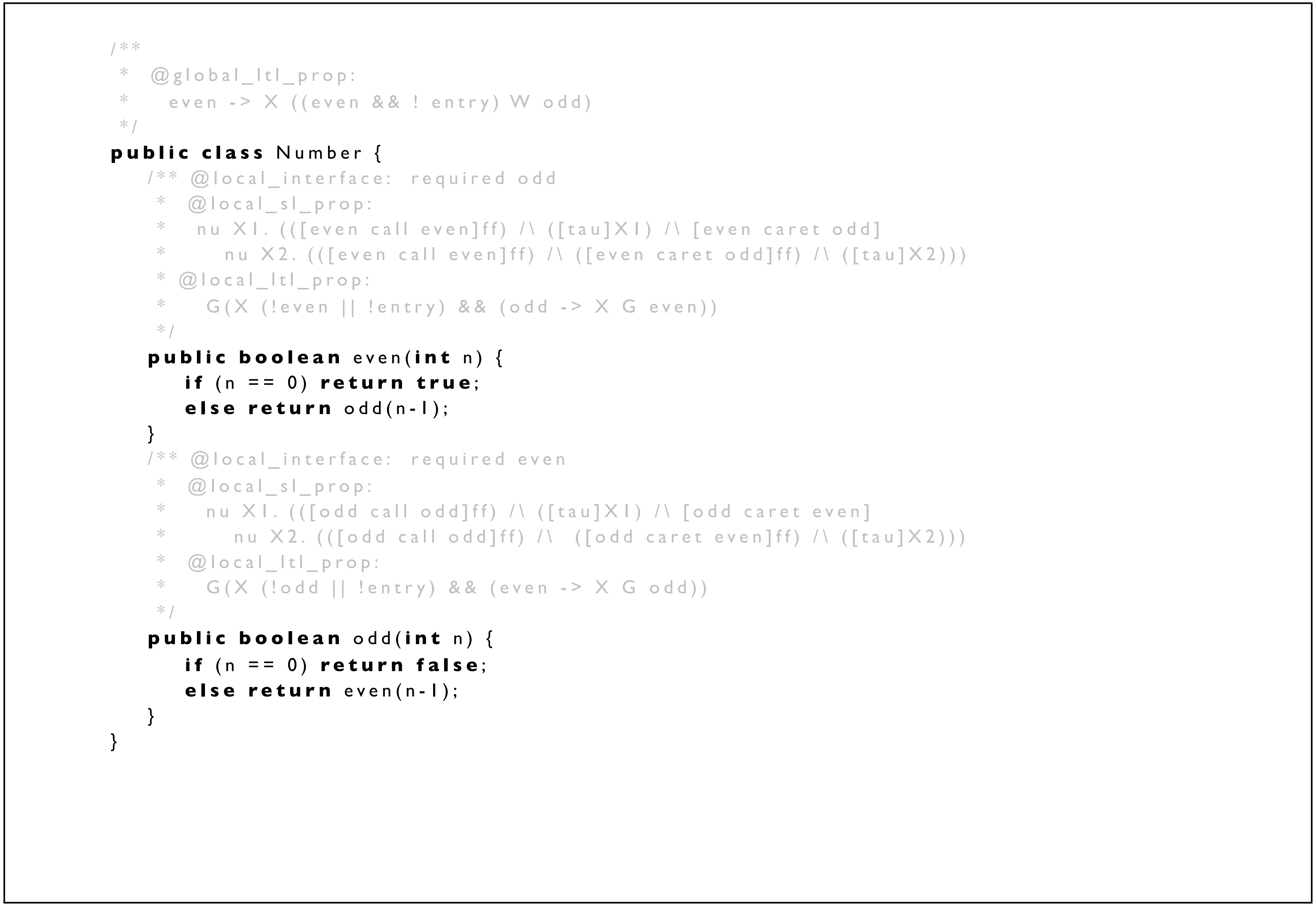
- Evaluating and identifying application areas
  - Experiments with product families
- Extending the class of properties
  - Encoding data from finite domains through control

Papers

- **ProMoVer**
  - Siavash Soleimanifard, Dilian Gurov, and Marieke Huisman. *Procedure-modular verification of control flow safety properties.* In FTfJP '10
  - Siavash Soleimanifard, Dilian Gurov, and Marieke Huisman. *ProMoVer: Modular verification of temporal safety properties.* In SEFM '11
  - Siavash Soleimanifard, Dilian Gurov, and Marieke Huisman. *Procedure-modular specification and verification of temporal safety properties.* Submitted to the SoSyM special issue on SEFM 2011.
- **Product Families**
  - Ina Schaefer, Dilian Gurov, and Siavash Soleimanifard. *Compositional algorithmic verification of software product lines.* In FMCO '10

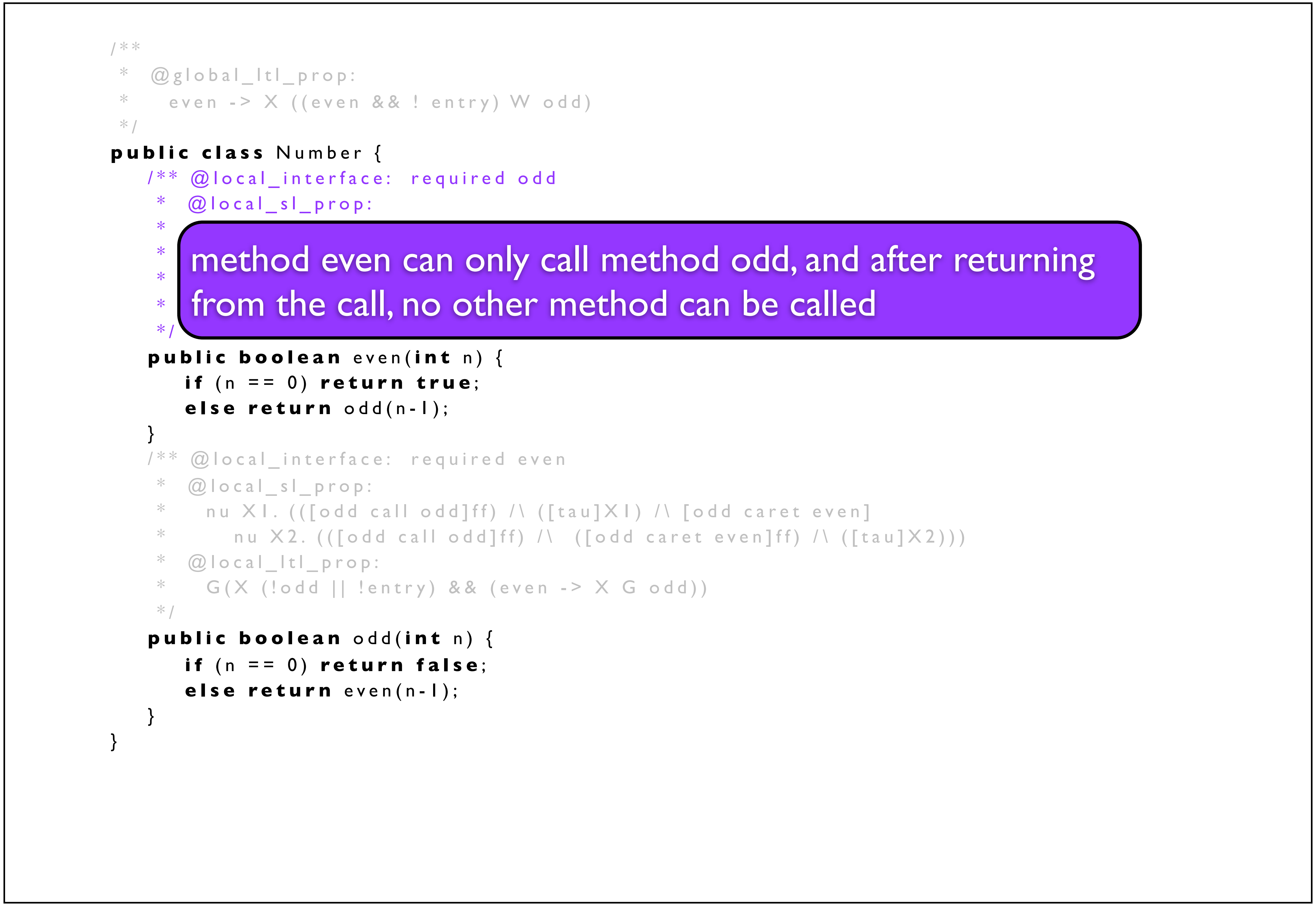
ProMoVer

ProMoVer -- Usage



```
/**
 *  @global_ltl_prop:
 *    even -> X ((even && ! entry) W odd)
 */
public class Number {
    /** @local_interface:  required 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)))
     * @local_ltl_prop:
     *    G(X (!even || !entry) && (odd -> X G even))
     */
    public boolean even(int n) {
        if (n == 0) return true;
        else return odd(n-1);
    }
    /** @local_interface:  required 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)))
     *  @local_ltl_prop:
     *    G(X (!odd || !entry) && (even -> X G odd))
     */
    public boolean odd(int n) {
        if (n == 0) return false;
        else return even(n-1);
    }
}
```

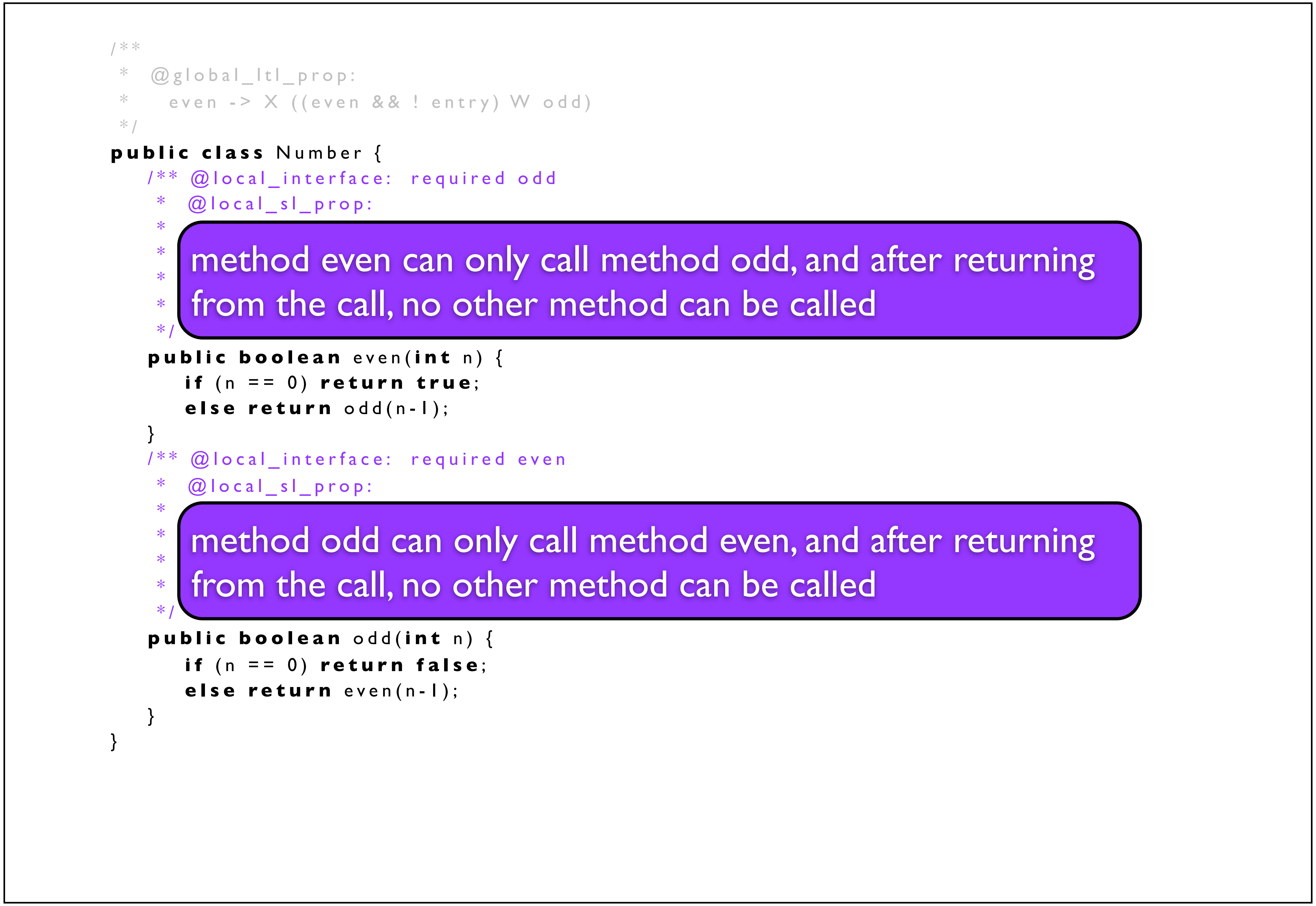
ProMoVer -- Usage



```
/**
 *  @global_ltl_prop:
 *    even -> X ((even && ! entry) W odd)
 */
public class Number {
    /** @local_interface:  required odd
     *  @local_sl_prop:
     *
     *
     *
     *
     */
    public boolean even(int n) {
        if (n == 0) return true;
        else return odd(n-1);
    }
    /** @local_interface:  required 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)))
     *  @local_ltl_prop:
     *    G(X (!odd || !entry) && (even -> X G odd))
     */
    public boolean odd(int n) {
        if (n == 0) return false;
        else return even(n-1);
    }
}
```

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

ProMoVer -- Usage

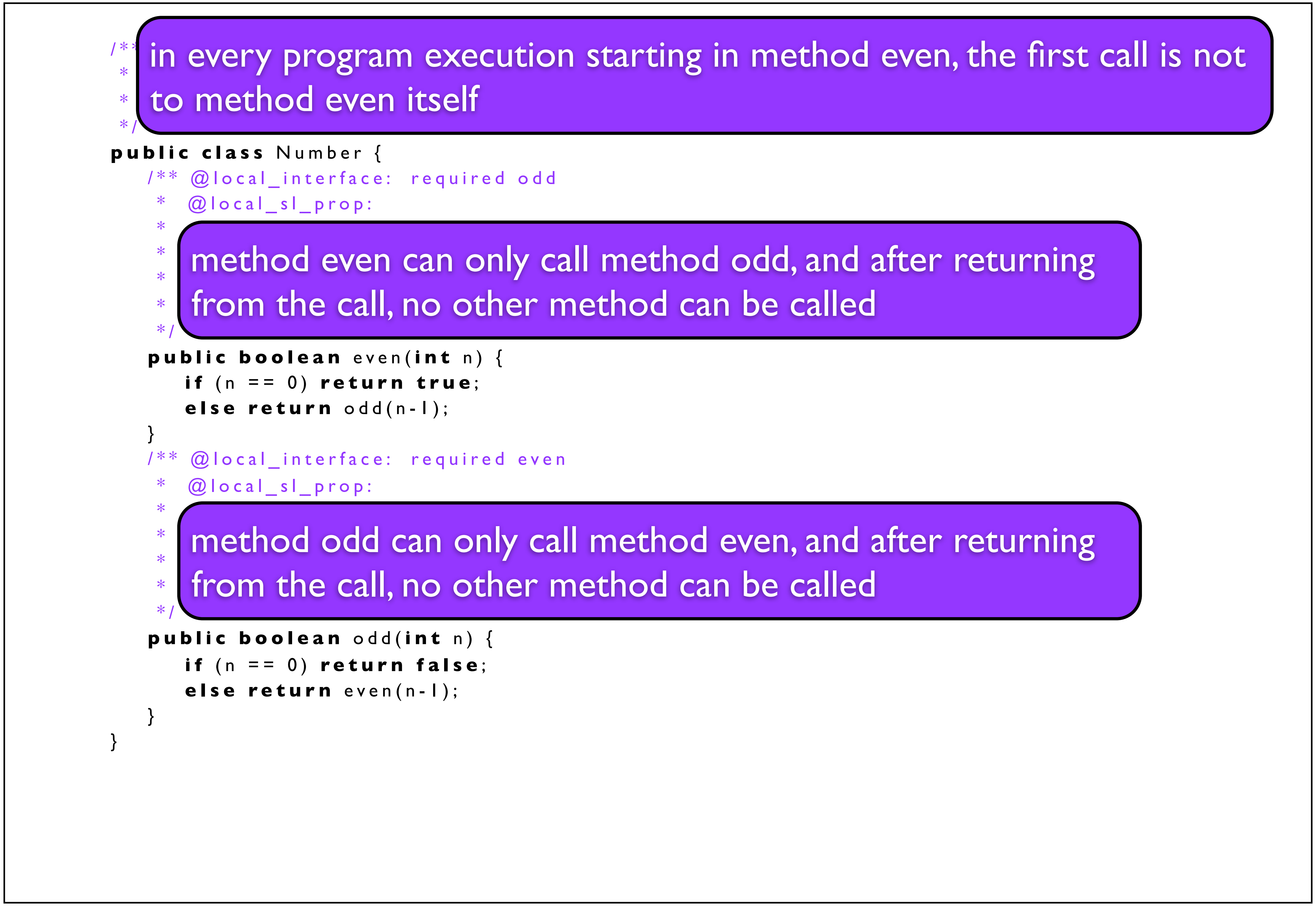


```
/**
 *  @global_ltl_prop:
 *    even -> X ((even && ! entry) W odd)
 */
public class Number {
    /** @local_interface: required odd
     *  @local_sl_prop:
     *
     *
     *
     *
     */
    public boolean even(int n) {
        if (n == 0) return true;
        else return odd(n-1);
    }
    /** @local_interface: required even
     *  @local_sl_prop:
     *
     *
     *
     *
     */
    public boolean odd(int n) {
        if (n == 0) return false;
        else return even(n-1);
    }
}
```

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

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

ProMoVer -- Usage



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

```
/**
 *
 *
 */
public class Number {
    /** @local_interface: required odd
     *  @local_sl_prop:
     *
     *
     *
     *
     */
```

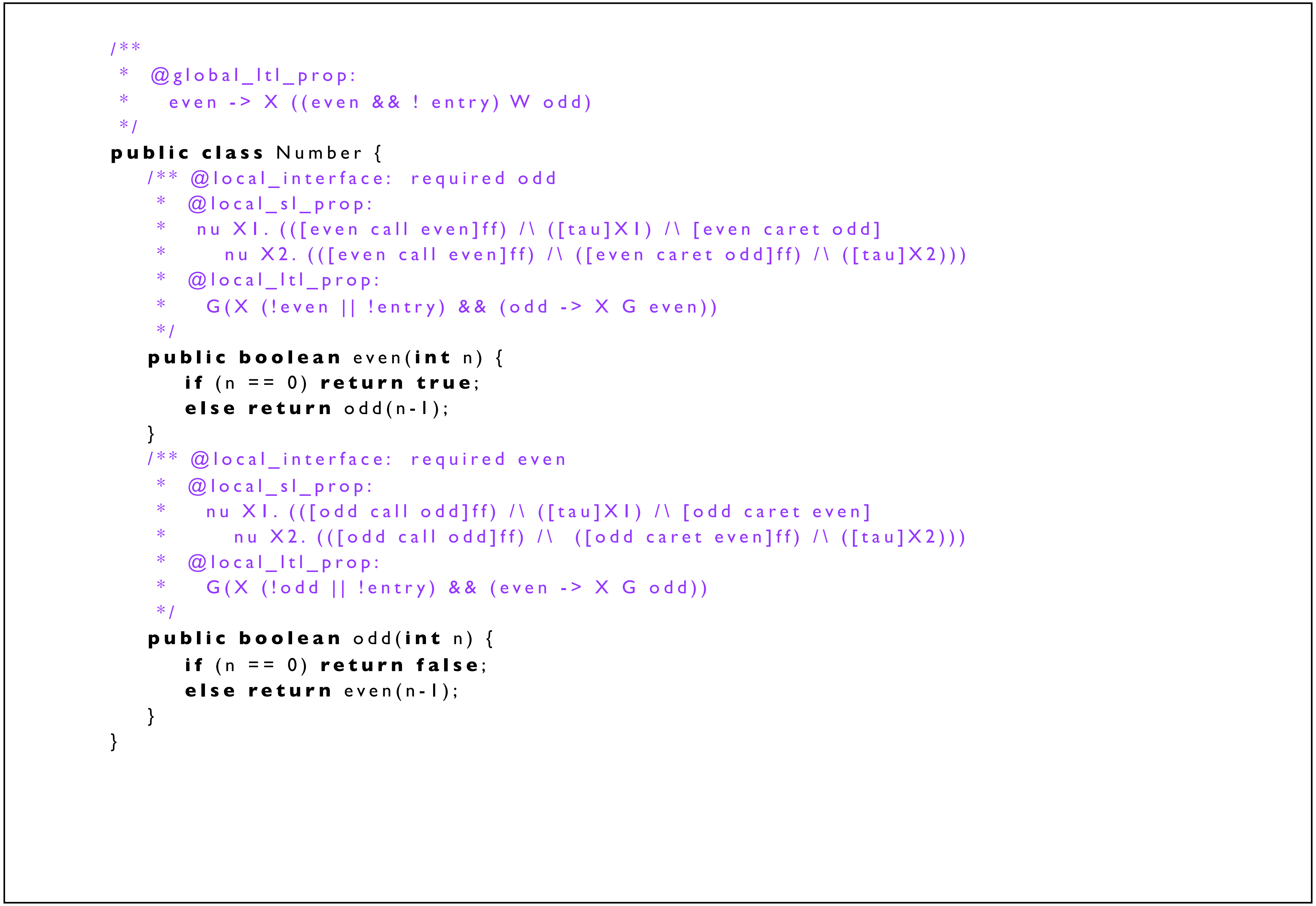
method even can only call method odd, and after returning from the call, no other method can be called

```
    public boolean even(int n) {
        if (n == 0) return true;
        else return odd(n-1);
    }
    /** @local_interface: required even
     *  @local_sl_prop:
     *
     *
     *
     *
     */
```

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

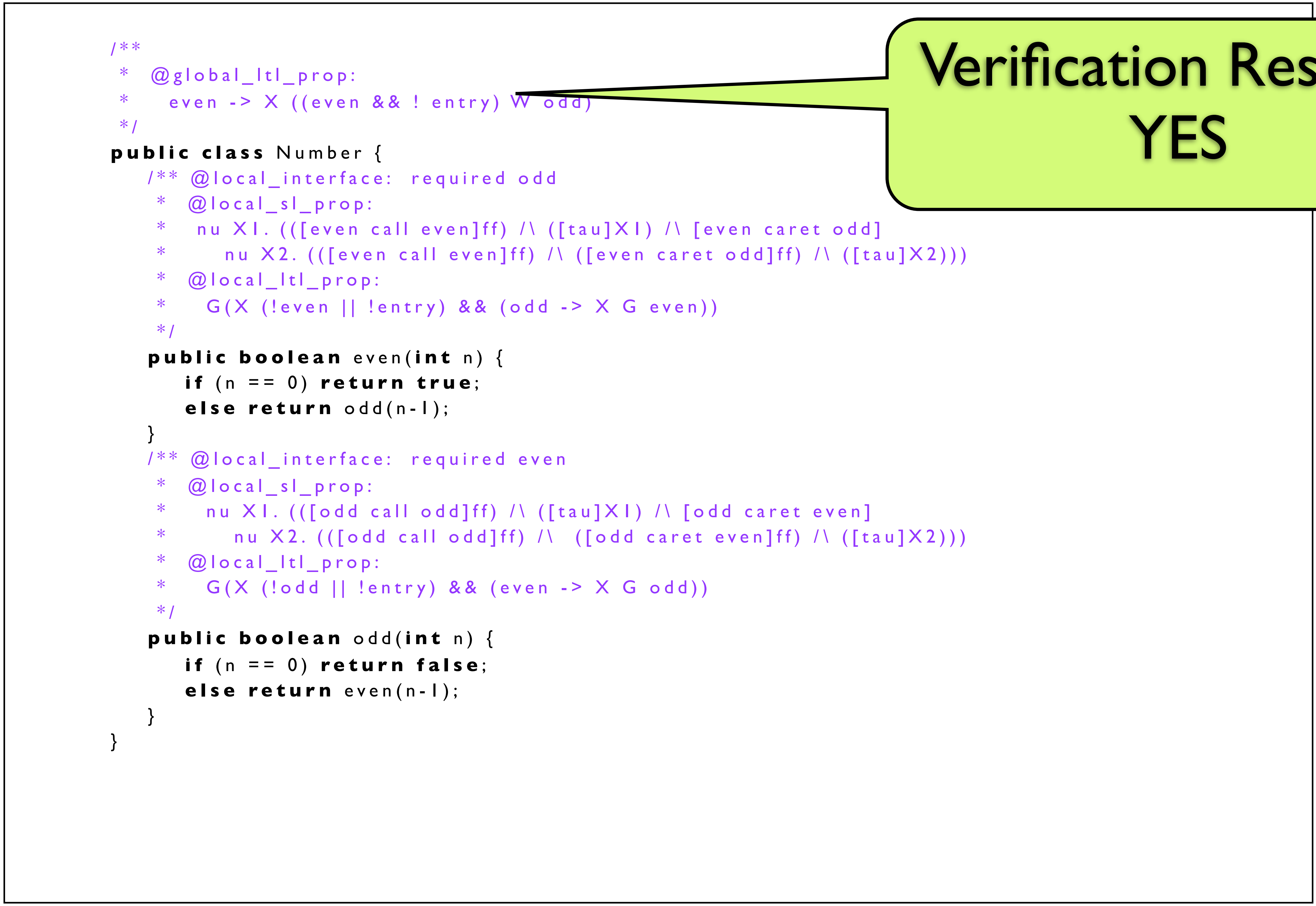
```
    public boolean odd(int n) {
        if (n == 0) return false;
        else return even(n-1);
    }
}
```

ProMoVer -- Usage



```
/**
 *  @global_ltl_prop:
 *    even -> X ((even && ! entry) W odd)
 */
public class Number {
    /** @local_interface:  required 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)))
     *  @local_ltl_prop:
     *    G(X (!even || !entry) && (odd -> X G even))
     */
    public boolean even(int n) {
        if (n == 0) return true;
        else return odd(n-1);
    }
    /** @local_interface:  required 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)))
     *  @local_ltl_prop:
     *    G(X (!odd || !entry) && (even -> X G odd))
     */
    public boolean odd(int n) {
        if (n == 0) return false;
        else return even(n-1);
    }
}
```

ProMoVer -- Usage



```
/**
 *  @global_ltl_prop:
 *    even -> X ((even && ! entry) W odd)
 */
public class Number {
    /** @local_interface:  required 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)))
     *  @local_ltl_prop:
     *    G(X (!even || !entry) && (odd -> X G even))
     */
    public boolean even(int n) {
        if (n == 0) return true;
        else return odd(n-1);
    }
    /** @local_interface:  required 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)))
     *  @local_ltl_prop:
     *    G(X (!odd || !entry) && (even -> X G odd))
     */
    public boolean odd(int n) {
        if (n == 0) return false;
        else return even(n-1);
    }
}
```

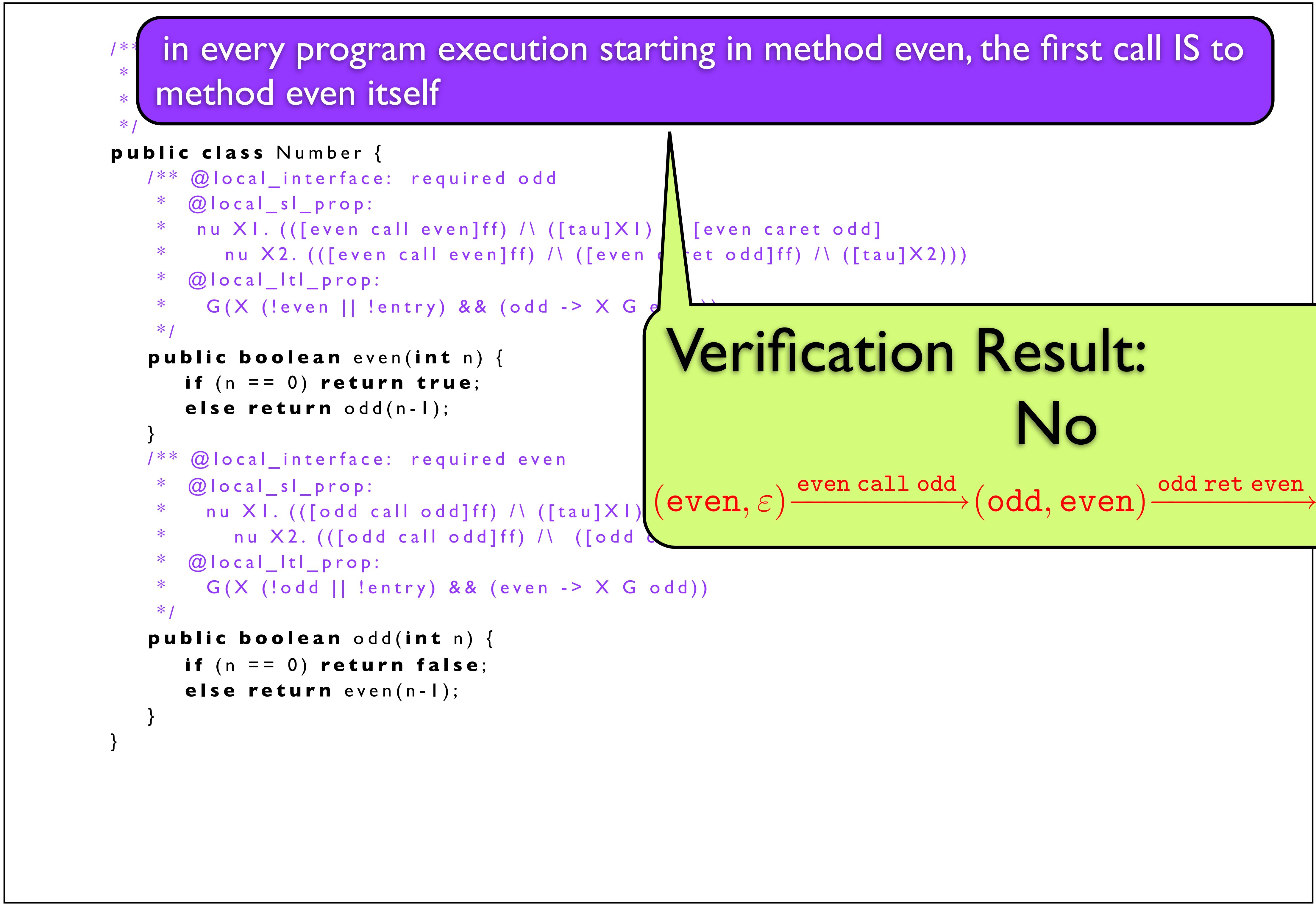
Verification Result:
YES

ProMoVer -- Usage

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

```
/**
 *
 *
 */
public class Number {
    /** @local_interface: required 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)))
     * @local_ltl_prop:
     *   G(X (!even || !entry) && (odd -> X G even))
     */
    public boolean even(int n) {
        if (n == 0) return true;
        else return odd(n-1);
    }
    /** @local_interface: required 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)))
     * @local_ltl_prop:
     *   G(X (!odd || !entry) && (even -> X G odd))
     */
    public boolean odd(int n) {
        if (n == 0) return false;
        else return even(n-1);
    }
}
```

ProMoVer -- Usage



```
/**
 *
 *
 */
public class Number {
   /** @local_interface: required odd
    * @local_sl_prop:
    *  nu X1. (([even call even]ff) /\ ([tau]X1)    [even caret odd]
    *     nu X2. (([even call even]ff) /\ ([even c  et odd]ff) /\ ([tau]X2)))
    * @local_ltl_prop:
    *   G(X (!even || !entry) && (odd -> X G e
    */
   public boolean even(int n) {
      if (n == 0) return true;
      else return odd(n-1);
   }
   /** @local_interface: required even
    * @local_sl_prop:
    *   nu X1. (([odd call odd]ff) /\ ([tau]X1)
    *      nu X2. (([odd call odd]ff) /\  ([odd c
    * @local_ltl_prop:
    *   G(X (!odd || !entry) && (even -> X G odd))
    */
   public boolean odd(int n) {
      if (n == 0) return false;
      else return even(n-1);
   }
}
```

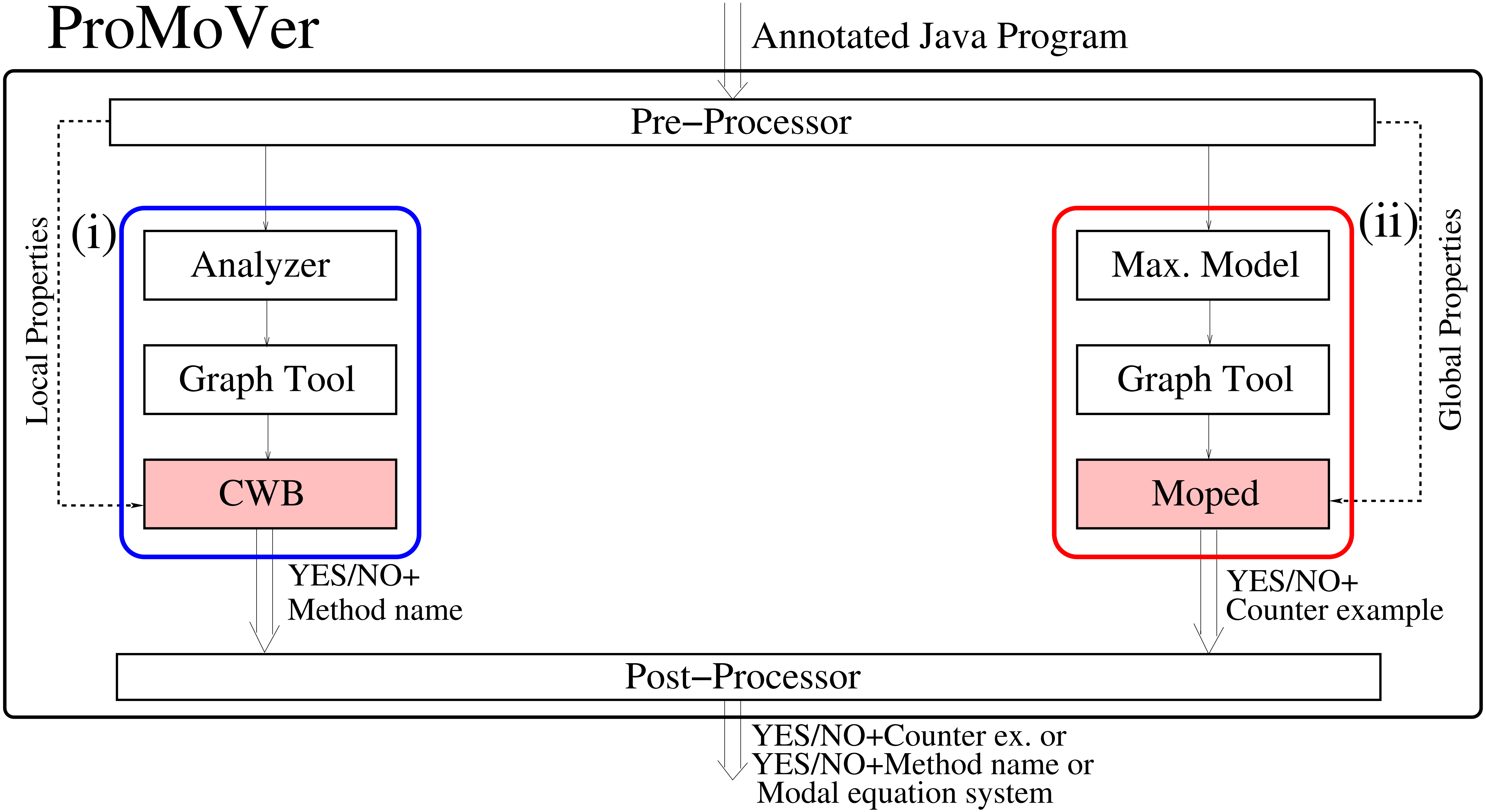
in every program execution starting in method even, the first call IS to method even itself

Verification Result:

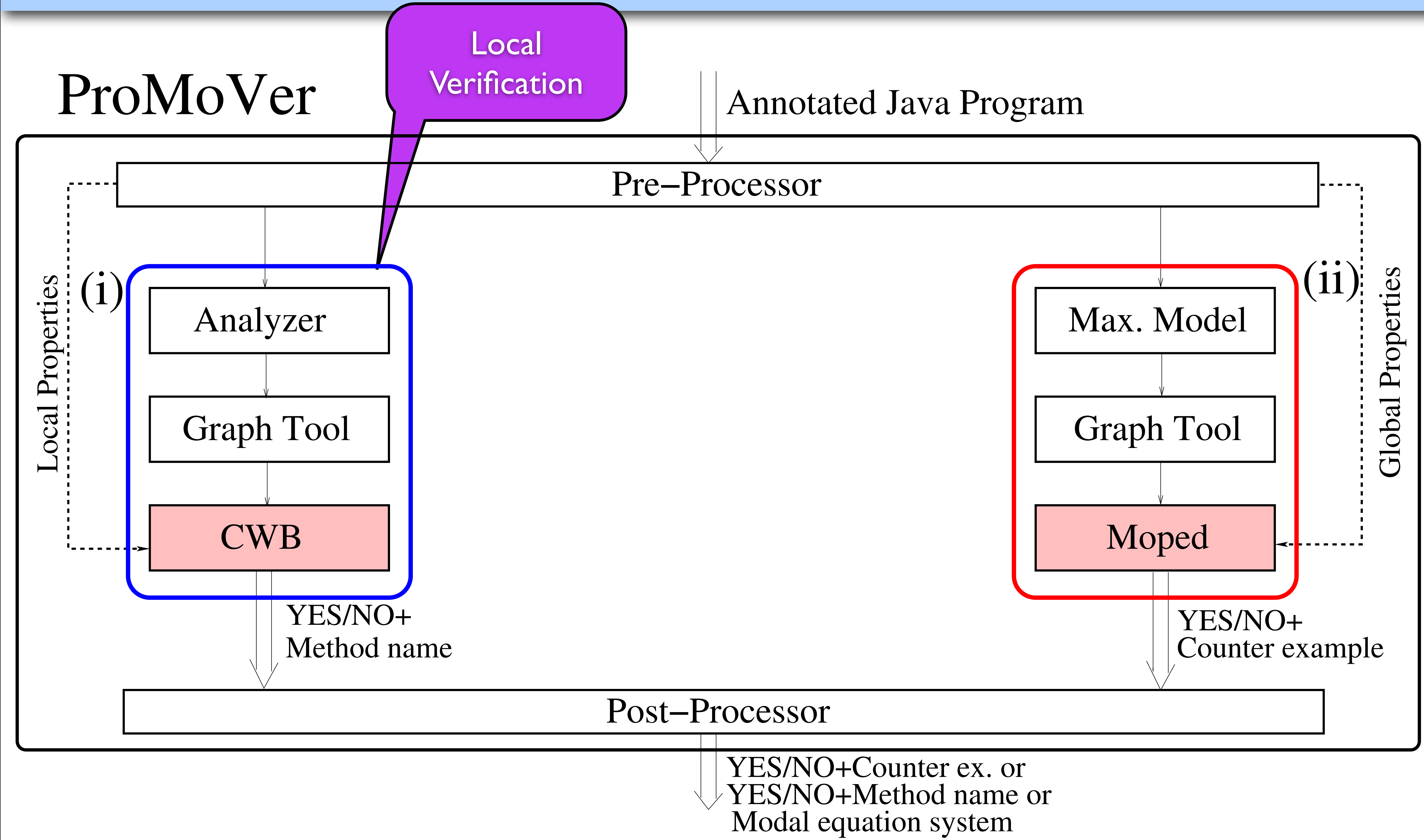
No

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

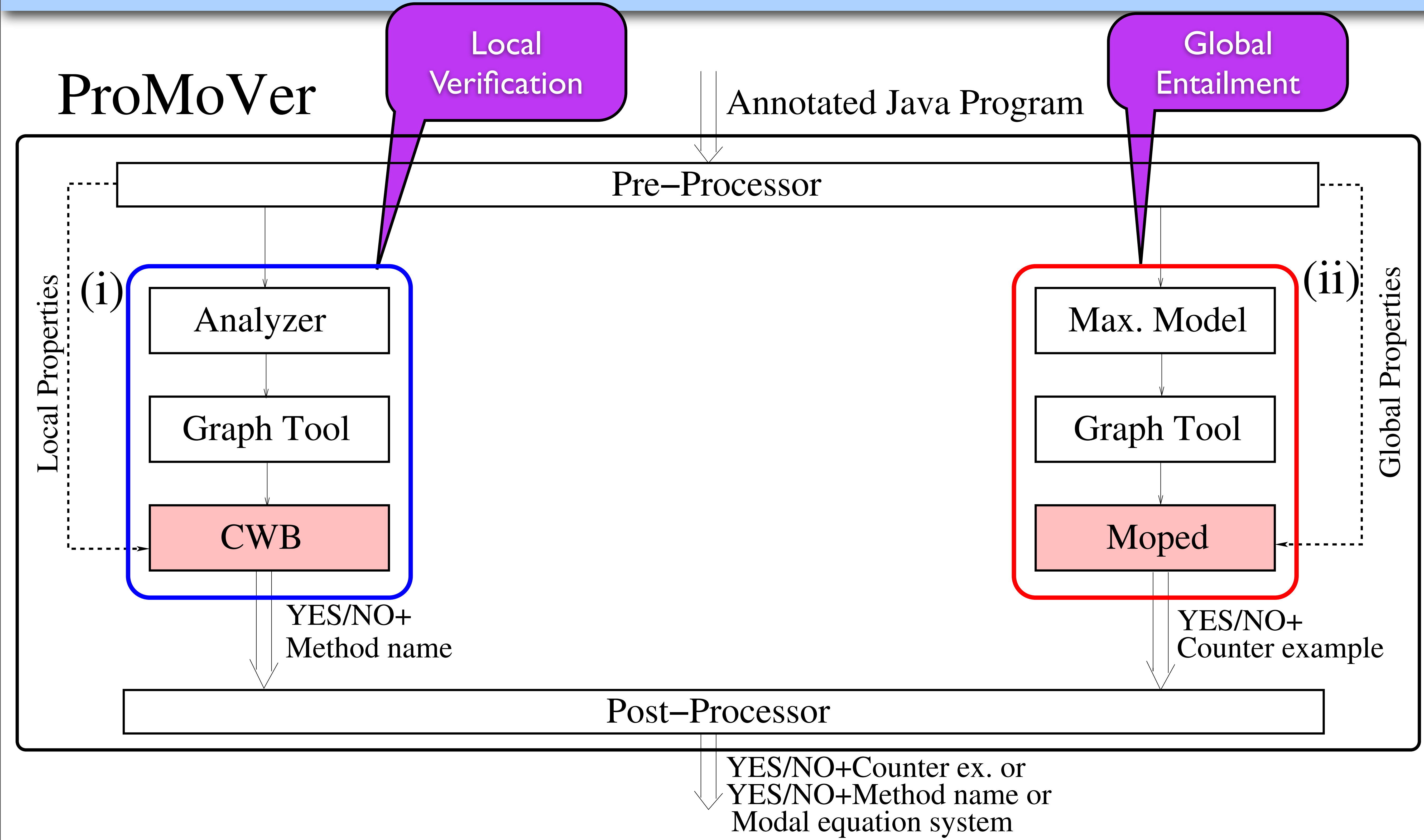
ProMoVer


ProMoVer
Annotated Java Program
Pre–Processor
Local Properties
(i)
Analyzer
Graph Tool
CWB
(ii)
Max. Model
Graph Tool
Moped
Global Properties
YES/NO+
Method name
YES/NO+
Counter example
Post–Processor
YES/NO+Counter ex. or
YES/NO+Method name or
Modal equation system

ProMoVer


Local Verification
ProMoVer
Annotated Java Program
Pre–Processor
Local Properties
(i)
Analyzer
Graph Tool
CWB
(ii)
Global Properties
Max. Model
Graph Tool
Moped
YES/NO+
Method name
YES/NO+
Counter example
Post–Processor
YES/NO+Counter ex. or
YES/NO+Method name or
Modal equation system

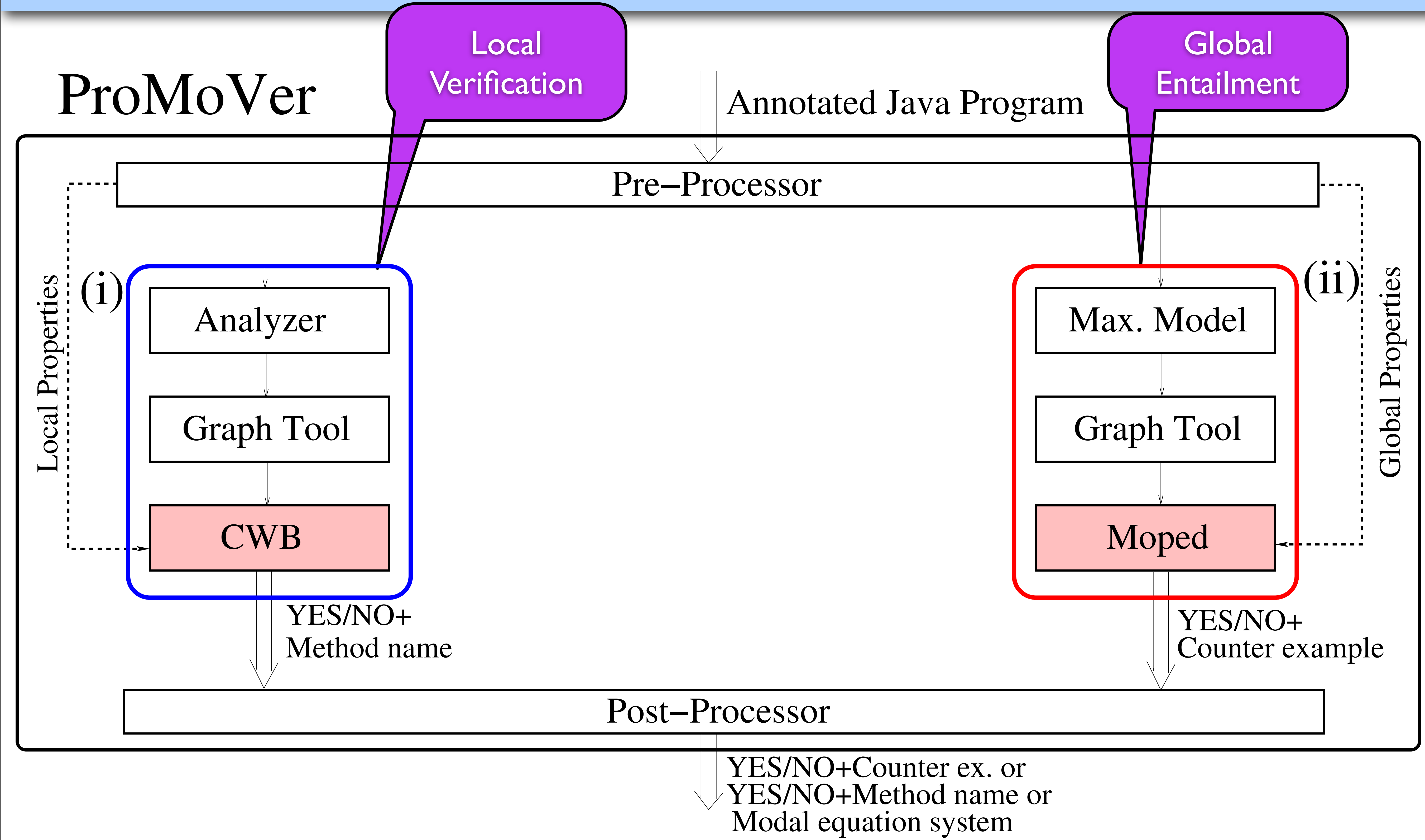
ProMoVer


ProMoVer
Local Verification
Global Entailment
Annotated Java Program
Pre–Processor
Local Properties
(i)
Analyzer
Graph Tool
CWB
(ii)
Max. Model
Graph Tool
Moped
Global Properties
YES/NO+
Method name
YES/NO+
Counter example
Post–Processor
YES/NO+Counter ex. or
YES/NO+Method name or
Modal equation system

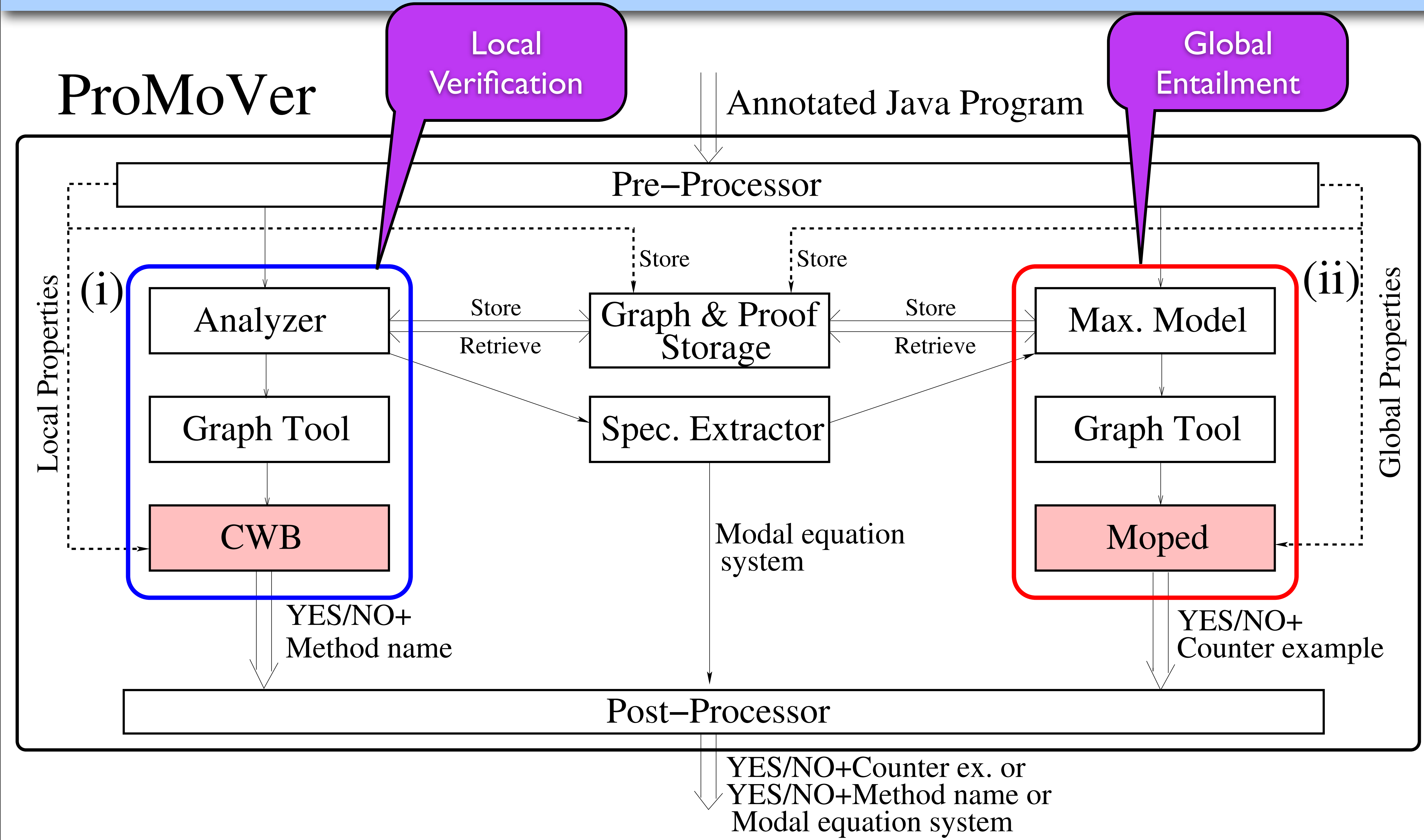
ProMoVer

- Different specification languages
  - Safety fragment of modal mu-calculus
  - Modal equation systems
  - Safety LTL
  - Safety automata
- Specification extractor
- Proof storage and reuse mechanism

ProMoVer


ProMoVer
Local Verification
Global Entailment
Annotated Java Program
Pre–Processor
Local Properties
(i)
Analyzer
Graph Tool
CWB
YES/NO+
Method name
(ii)
Max. Model
Graph Tool
Moped
Global Properties
YES/NO+
Counter example
Post–Processor
YES/NO+Counter ex. or
YES/NO+Method name or
Modal equation system

ProMoVer



ProMoVer
Local Verification
Global Entailment
Annotated Java Program
Pre–Processor
Local Properties
Global Properties
(i)
(ii)
Store
Store
Analyzer
Store
Retrieve
Graph & Proof Storage
Store
Retrieve
Max. Model
Graph Tool
Spec. Extractor
Graph Tool
CWB
Modal equation system
Moped
YES/NO+ Method name
YES/NO+ Counter example
Post–Processor
YES/NO+Counter ex. or
YES/NO+Method name or
Modal equation system

Case Studies

Evaluating ProMoVer with three Java-Card applications

Case Studies

Evaluating ProMoVer with three Java-Card applications

Global Property

No non-atomic operation **within** a transaction

Case Studies

Evaluating ProMoVer with three Java-Card applications

Global Property

No non-atomic operation **within** a transaction

Application	Lines of Code	Local Model Check	Maximal Model Cons.	Global Model Check	Total Time
AccountAccessor	190	0.5 sec	0.7 sec	0.9 sec	8.7 sec
TransitApplet	918	0.5 sec	0.9 sec	0.9 sec	13.2 sec
JavaPurse	884	0.5 sec	13.0 sec	1.1 sec	22.5 sec

Case Studies

Evaluating ProMoVer with three Java-Card applications

Global Property

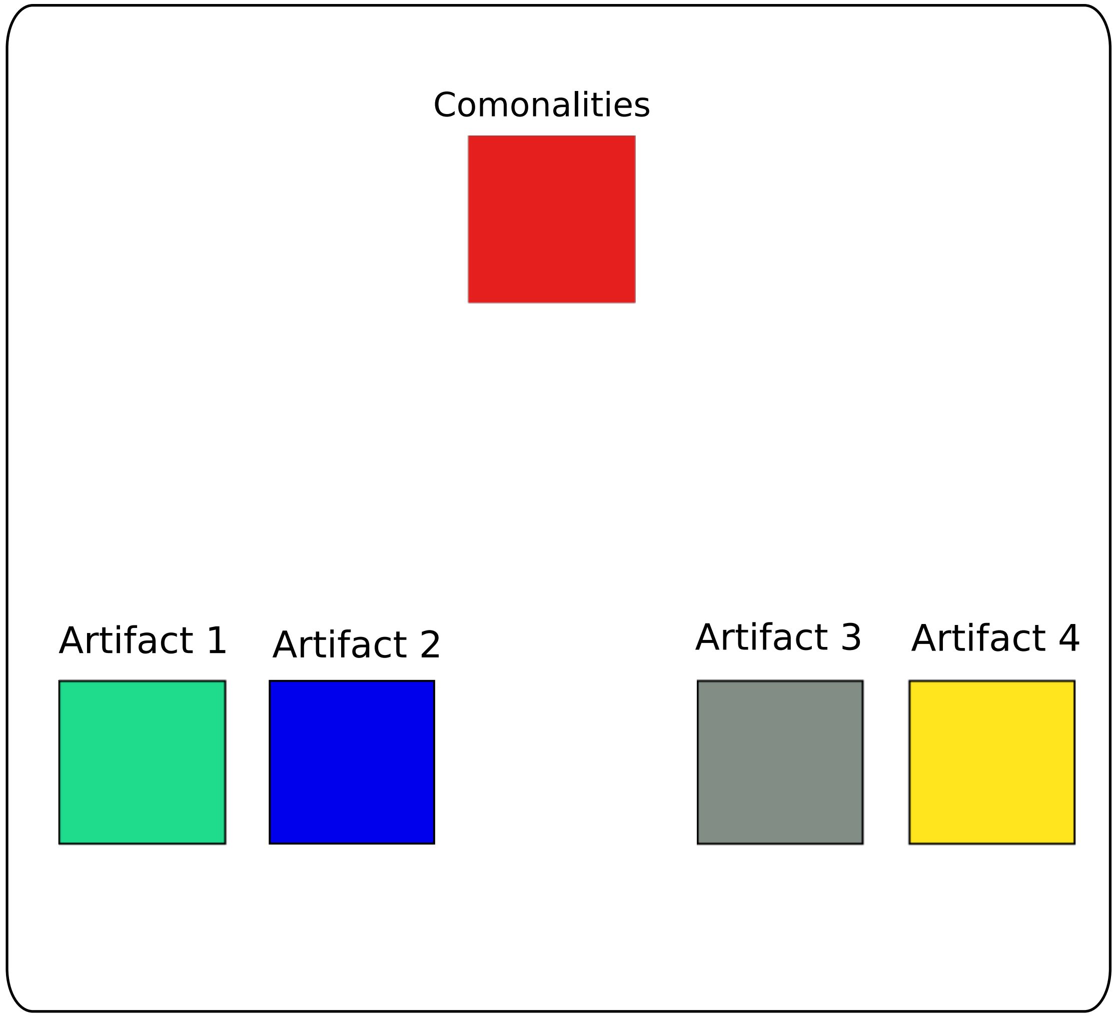
No non-atomic operation **within** a transaction

Application	Lines of Code	Local Model Check	Maximal Model Cons.	Global Model Check	Total Time	Code Change TT%	Spec. Change TT%
AccountAccessor	190	0.5 sec	0.7 sec	0.9 sec	8.7 sec	66	52
TransitApplet	918	0.5 sec	0.9 sec	0.9 sec	13.2 sec	44	37
JavaPurse	884	0.5 sec	13.0 sec	1.1 sec	22.5 sec	40	24

Product Families

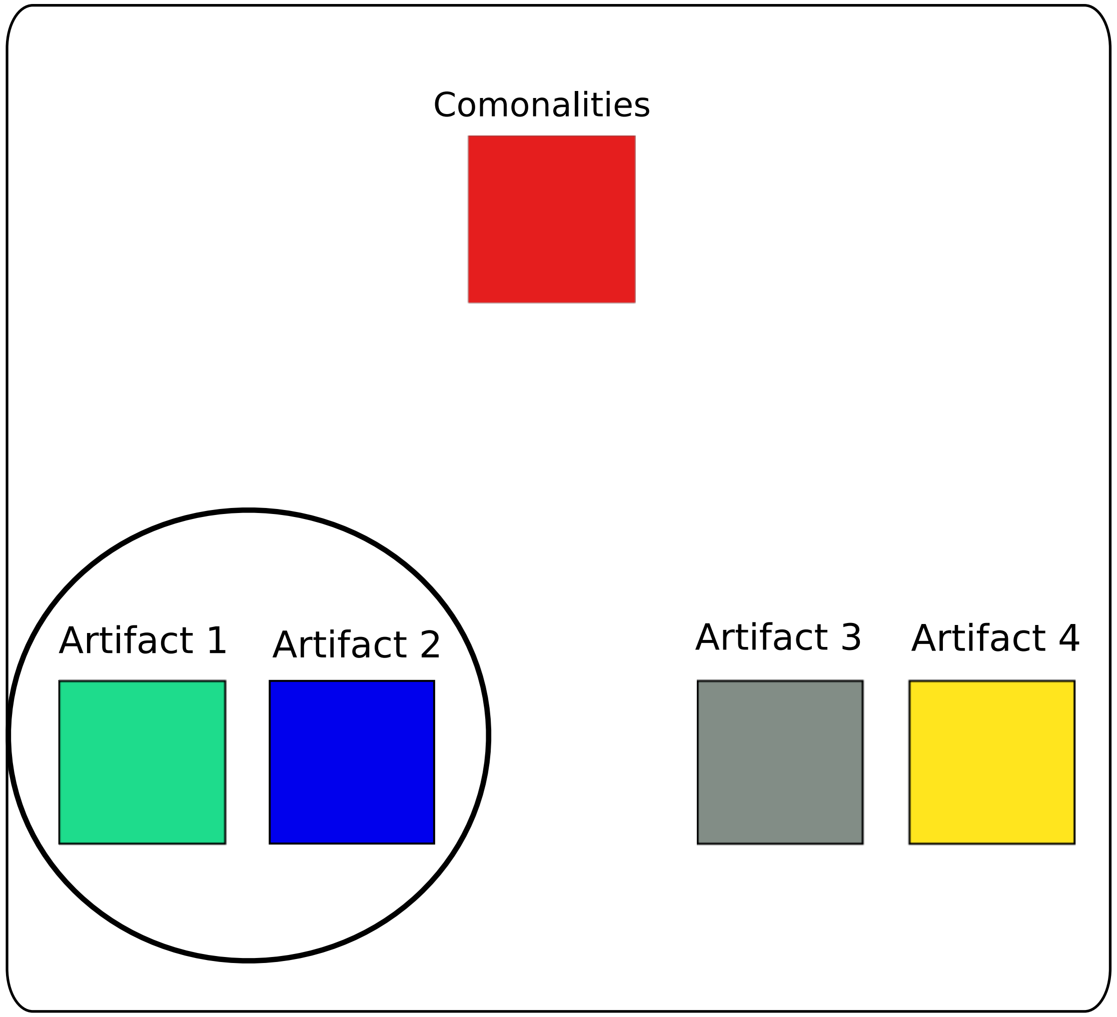
Product Families

- Set of products with well-defined commonalities and variabilities



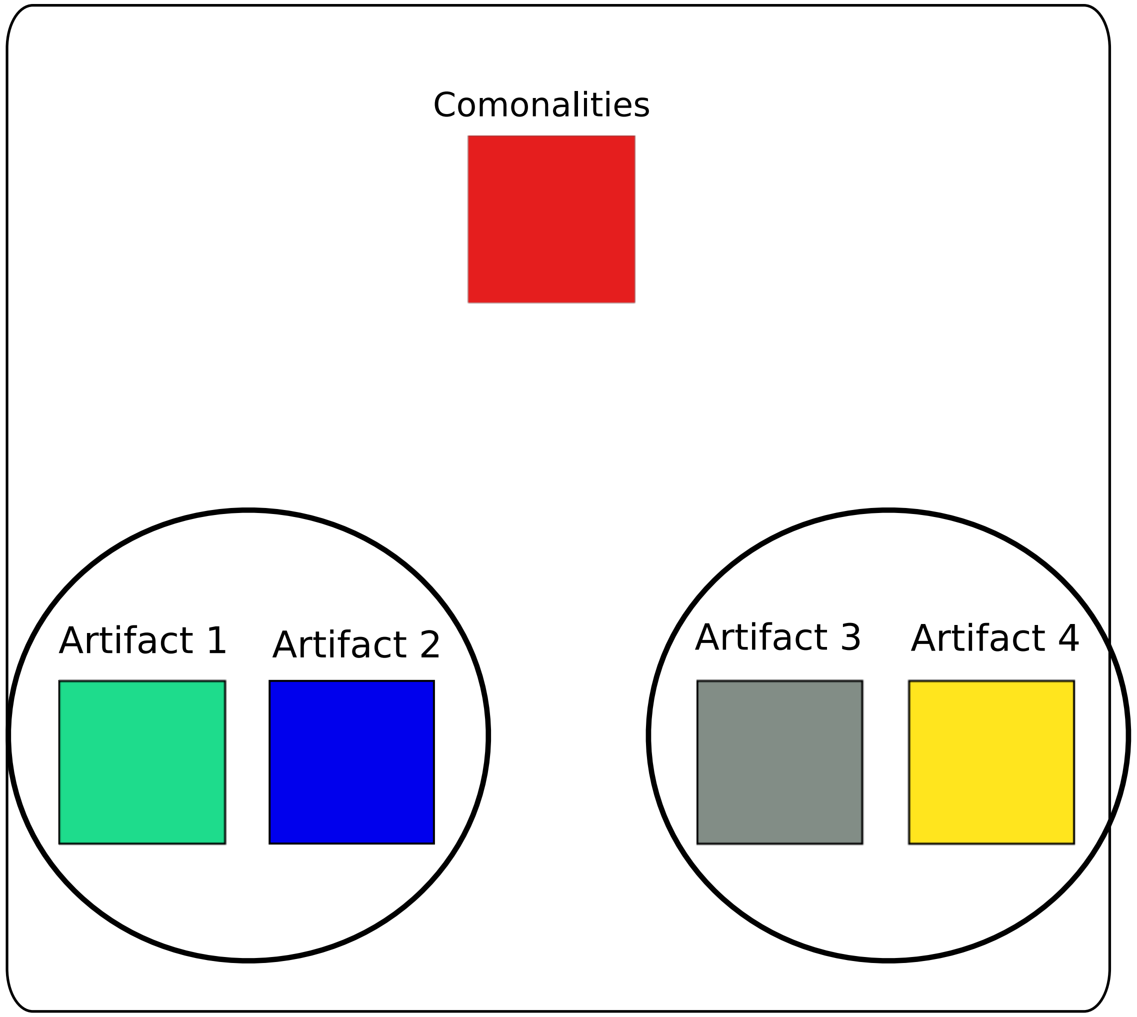
Product Families

- Set of products with well-defined commonalities and variabilities



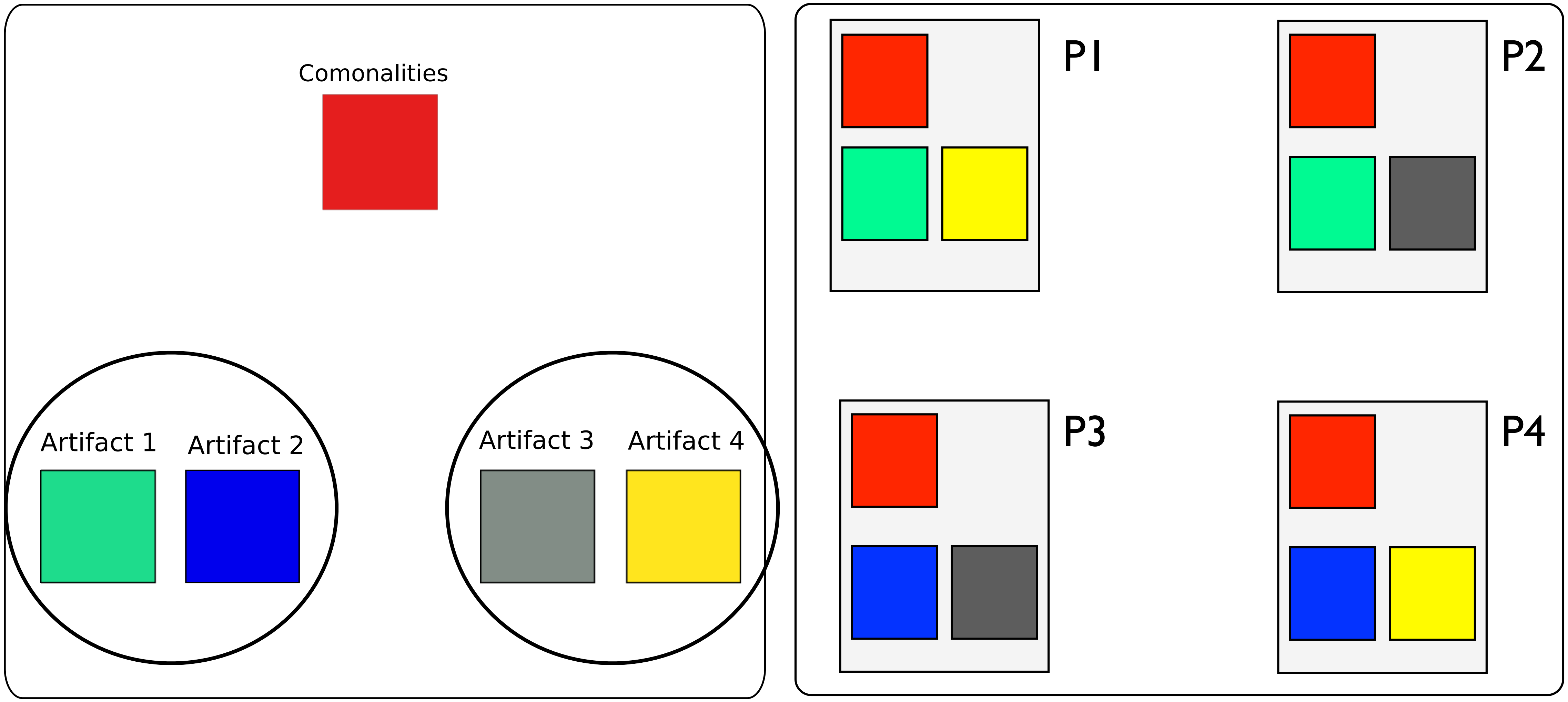
Product Families

- Set of products with well-defined commonalities and variabilities

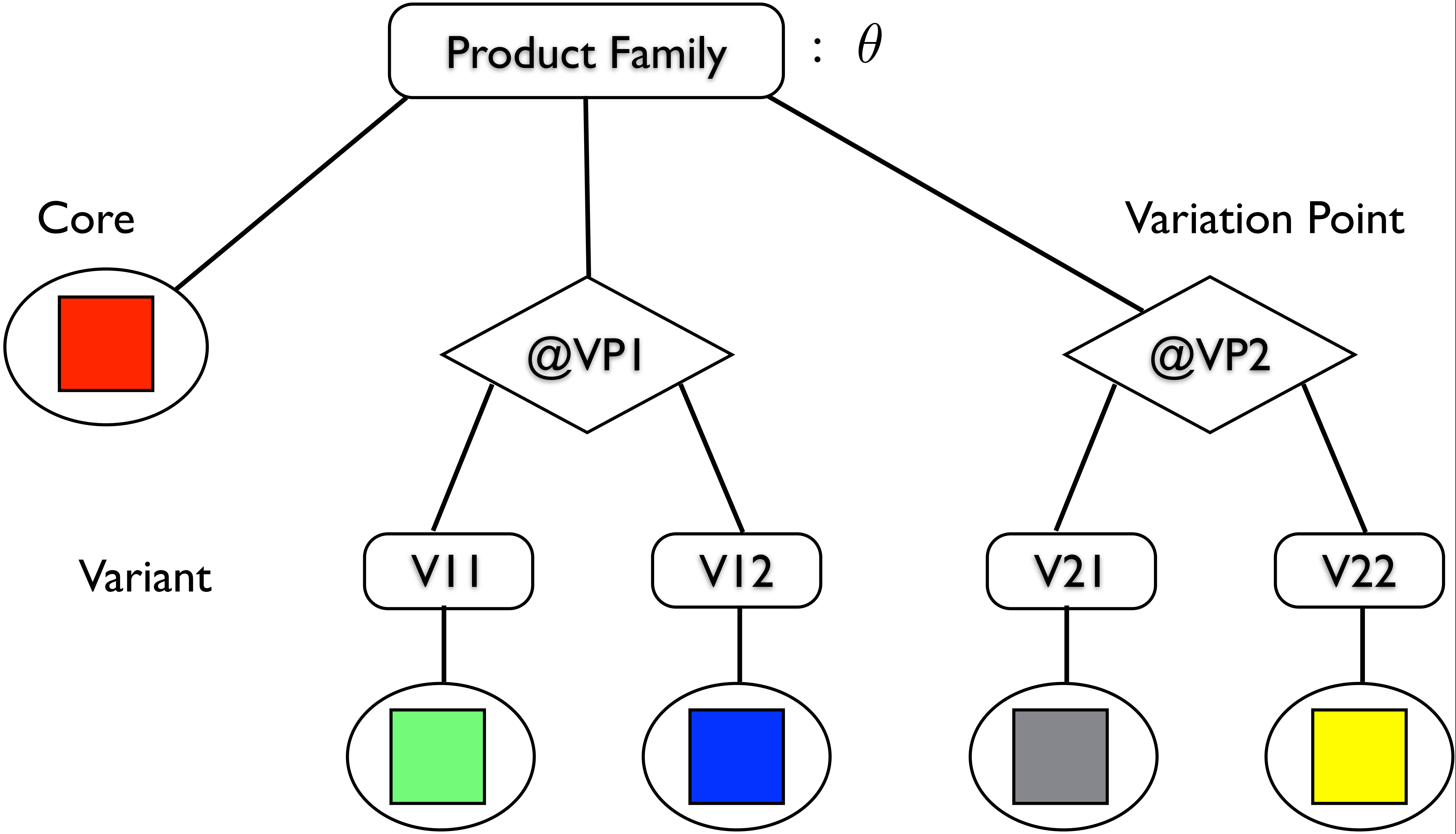


Product Families

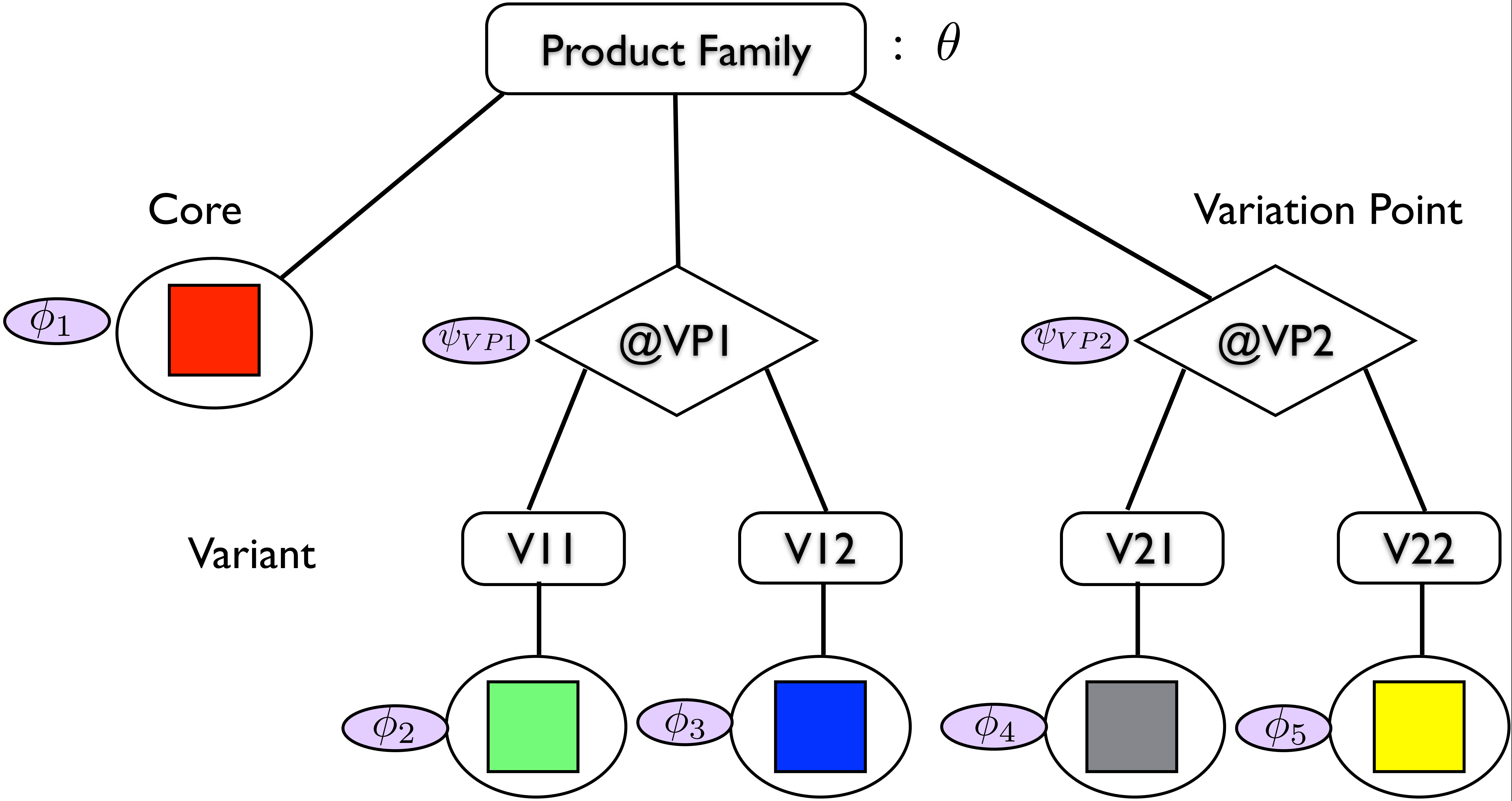
- Set of products with well-defined commonalities and variabilities



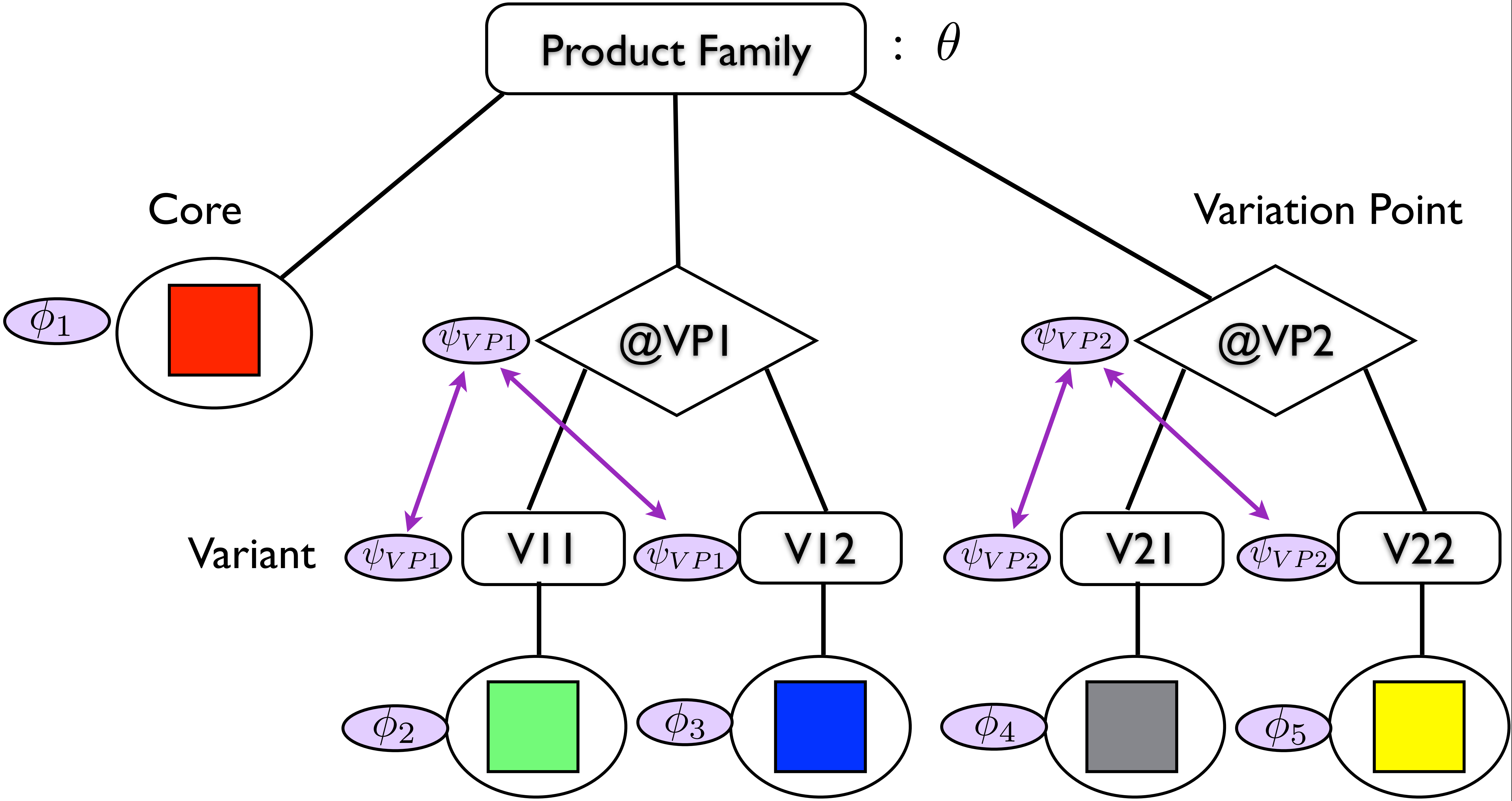
Hierarchical Variability



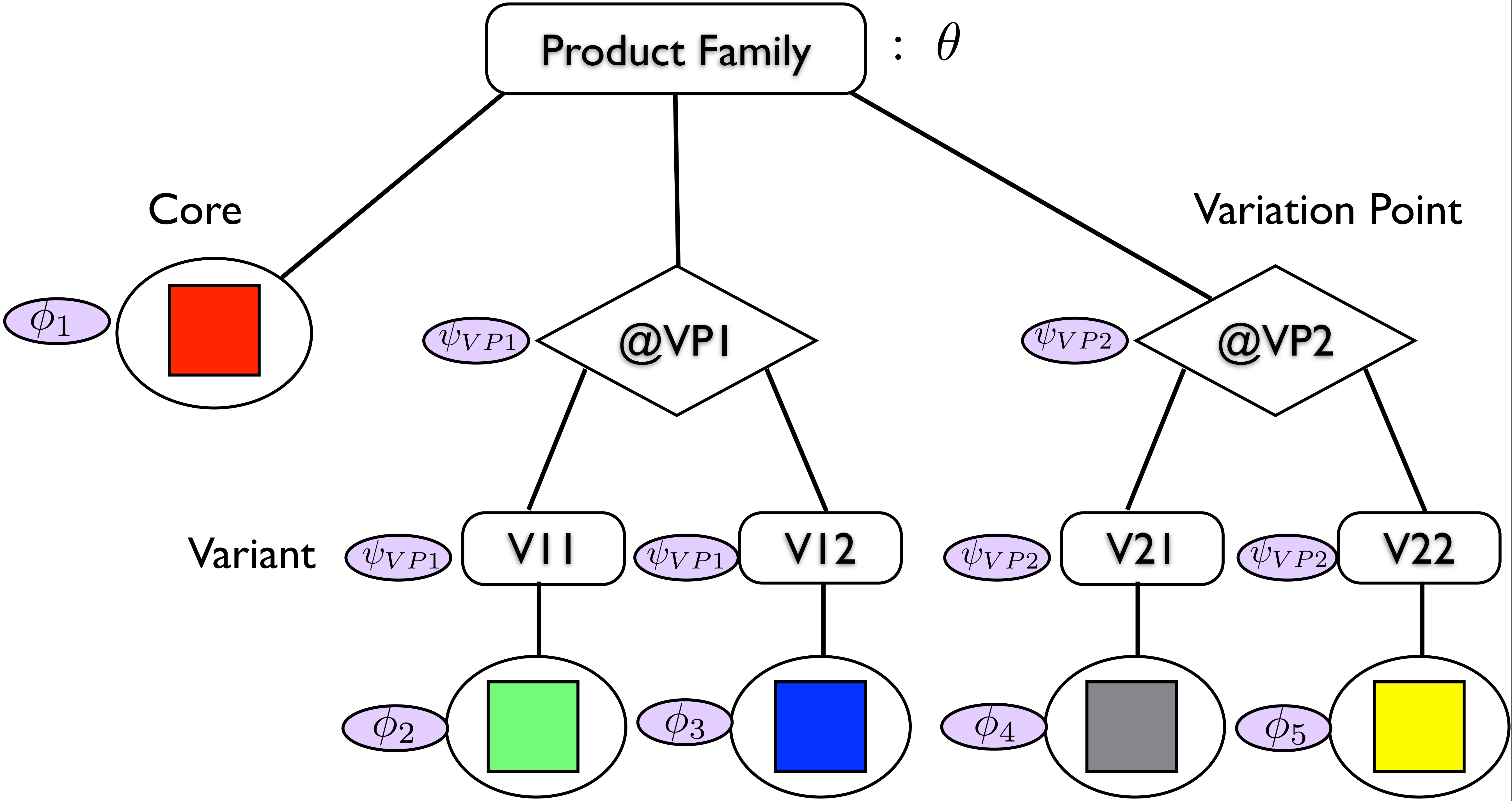
Hierarchical Variability



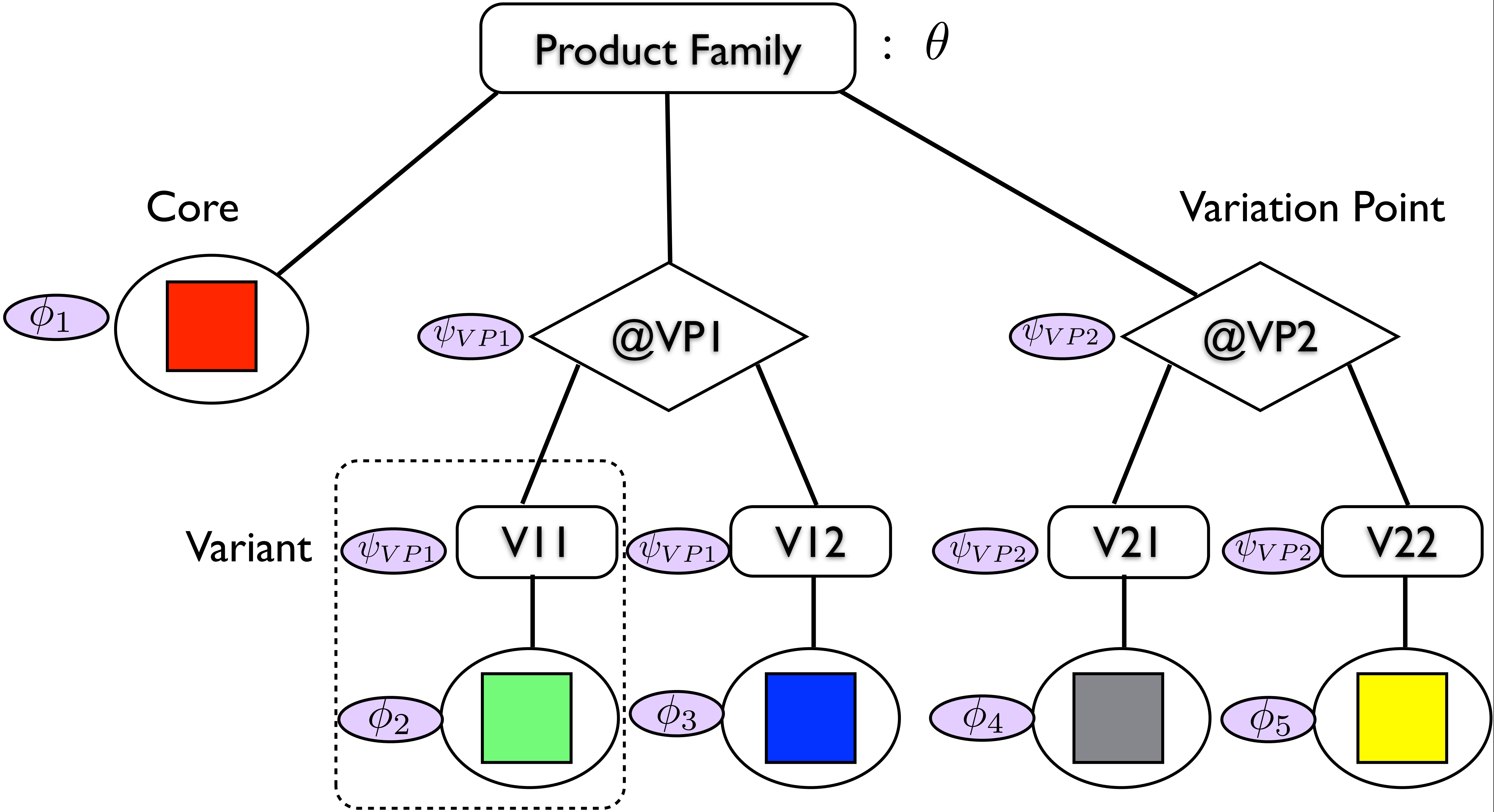
Hierarchical Variability



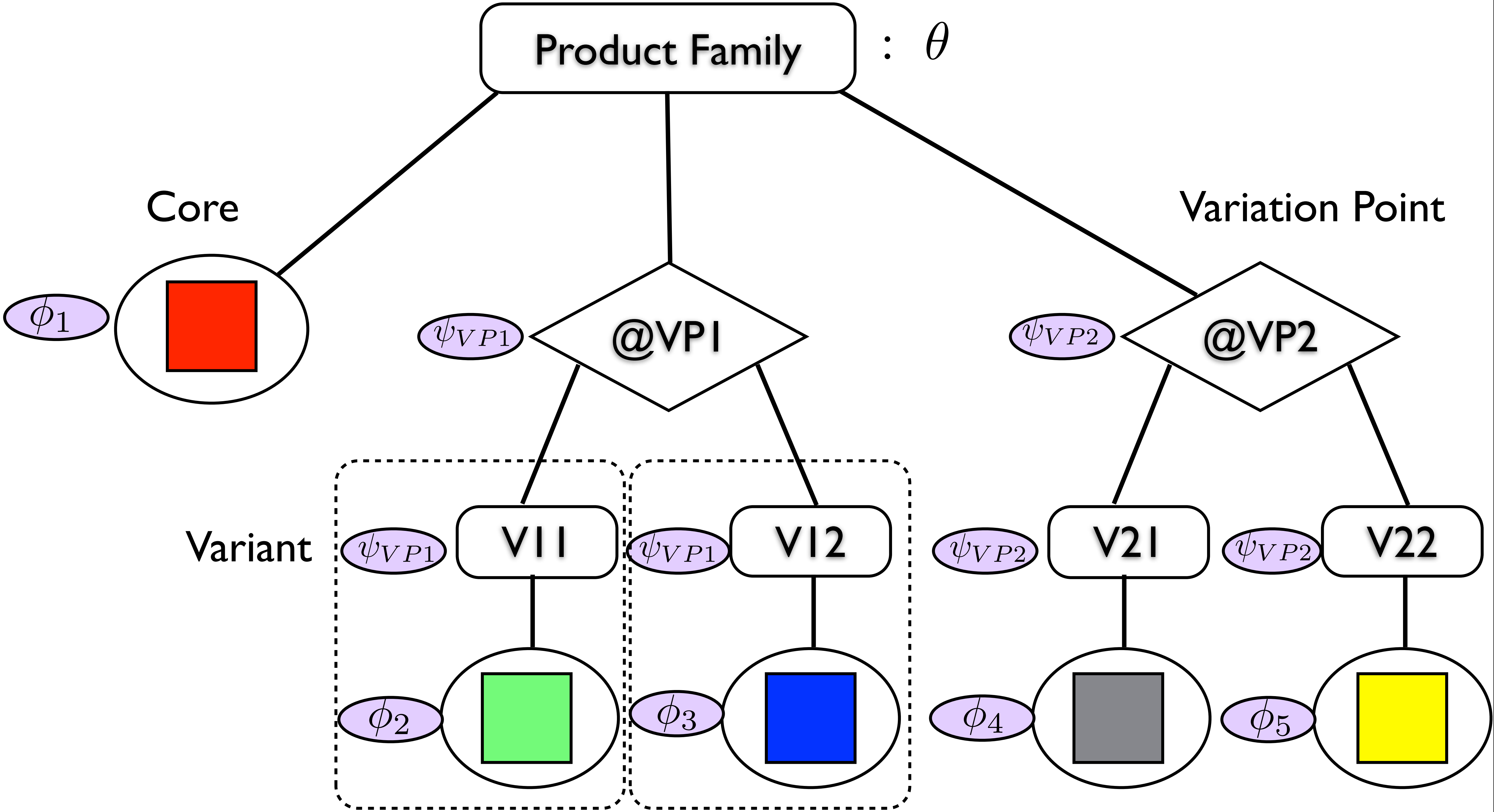
Hierarchical Variability



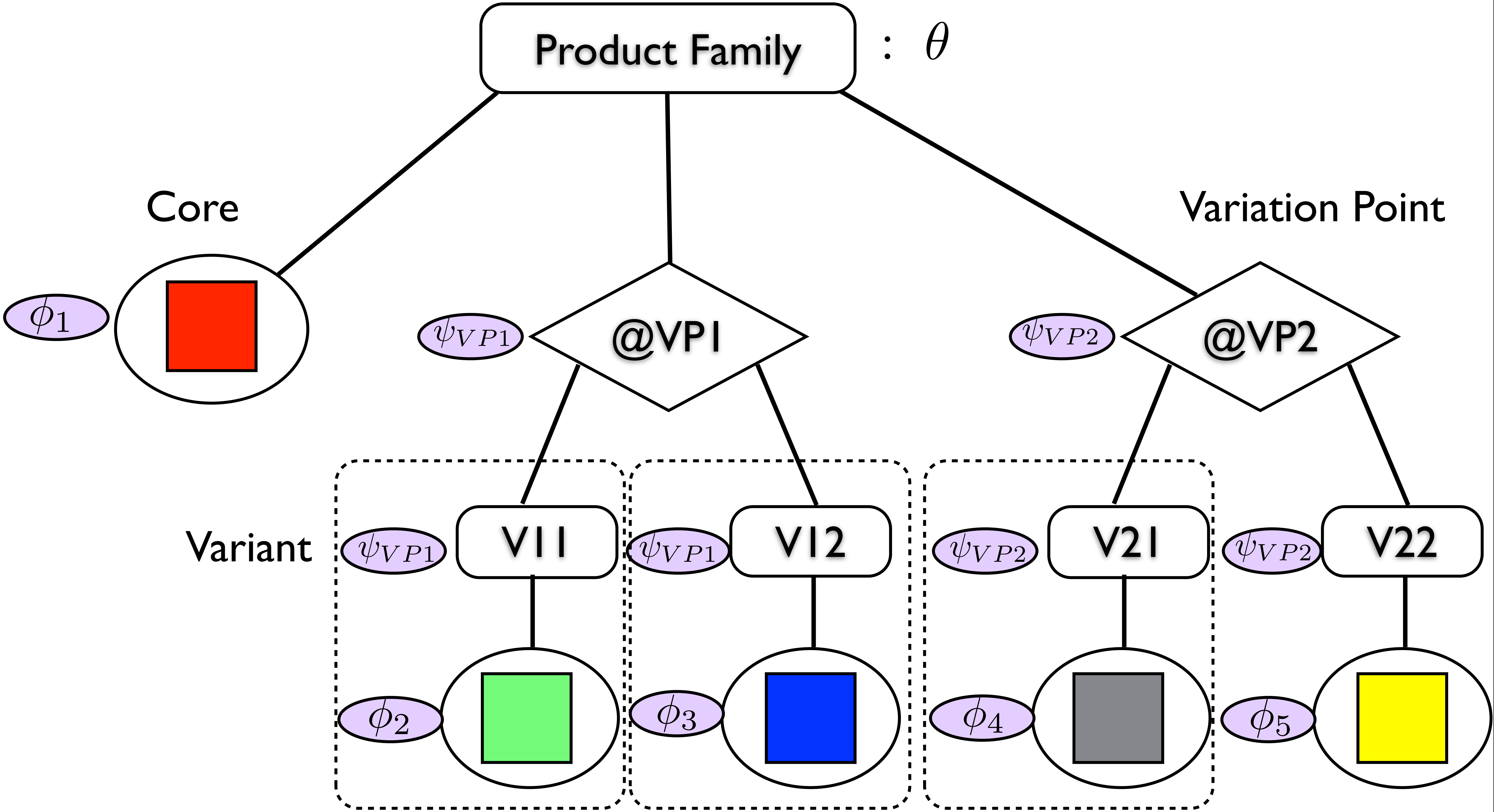
Hierarchical Variability



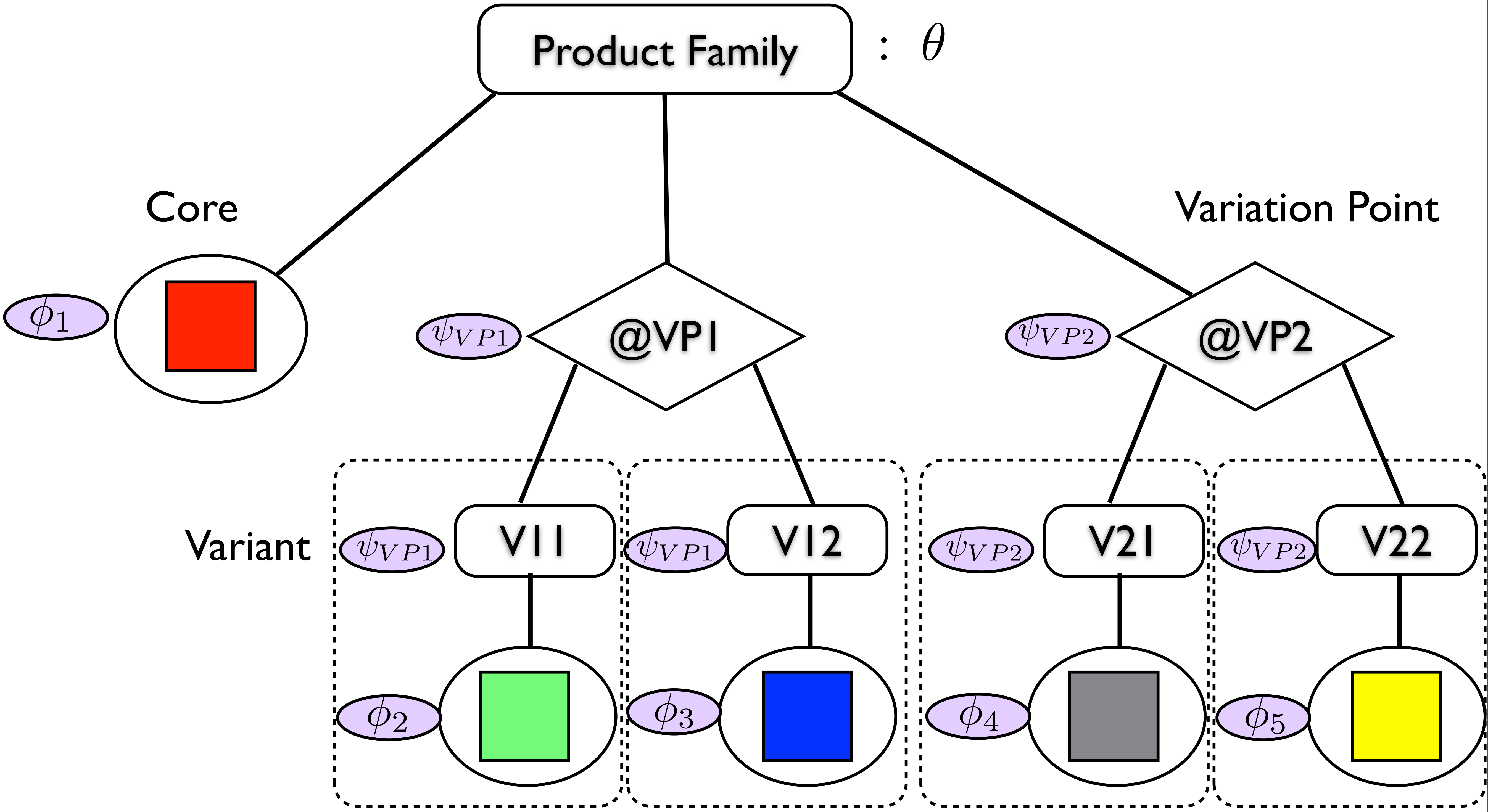
Hierarchical Variability



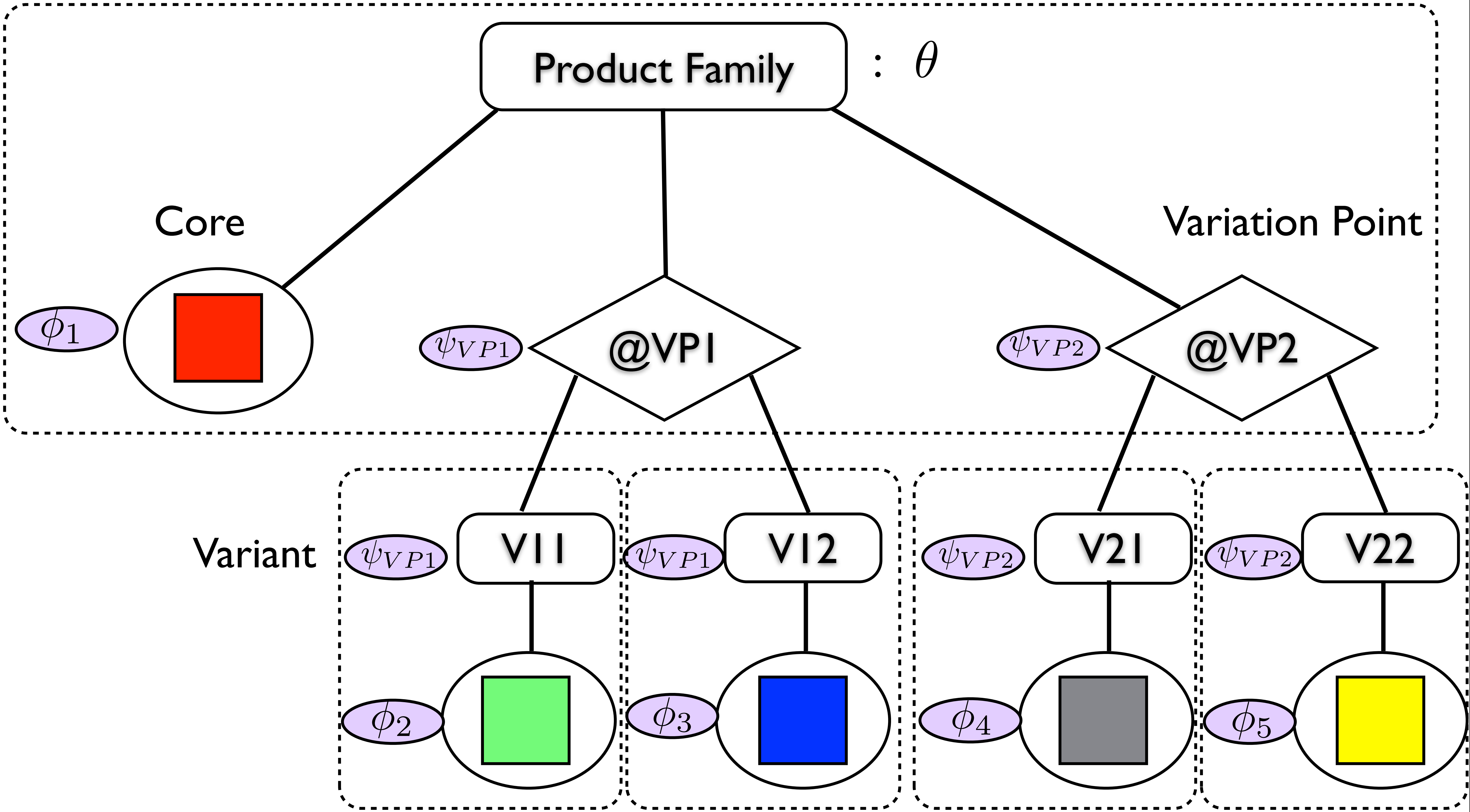
Hierarchical Variability



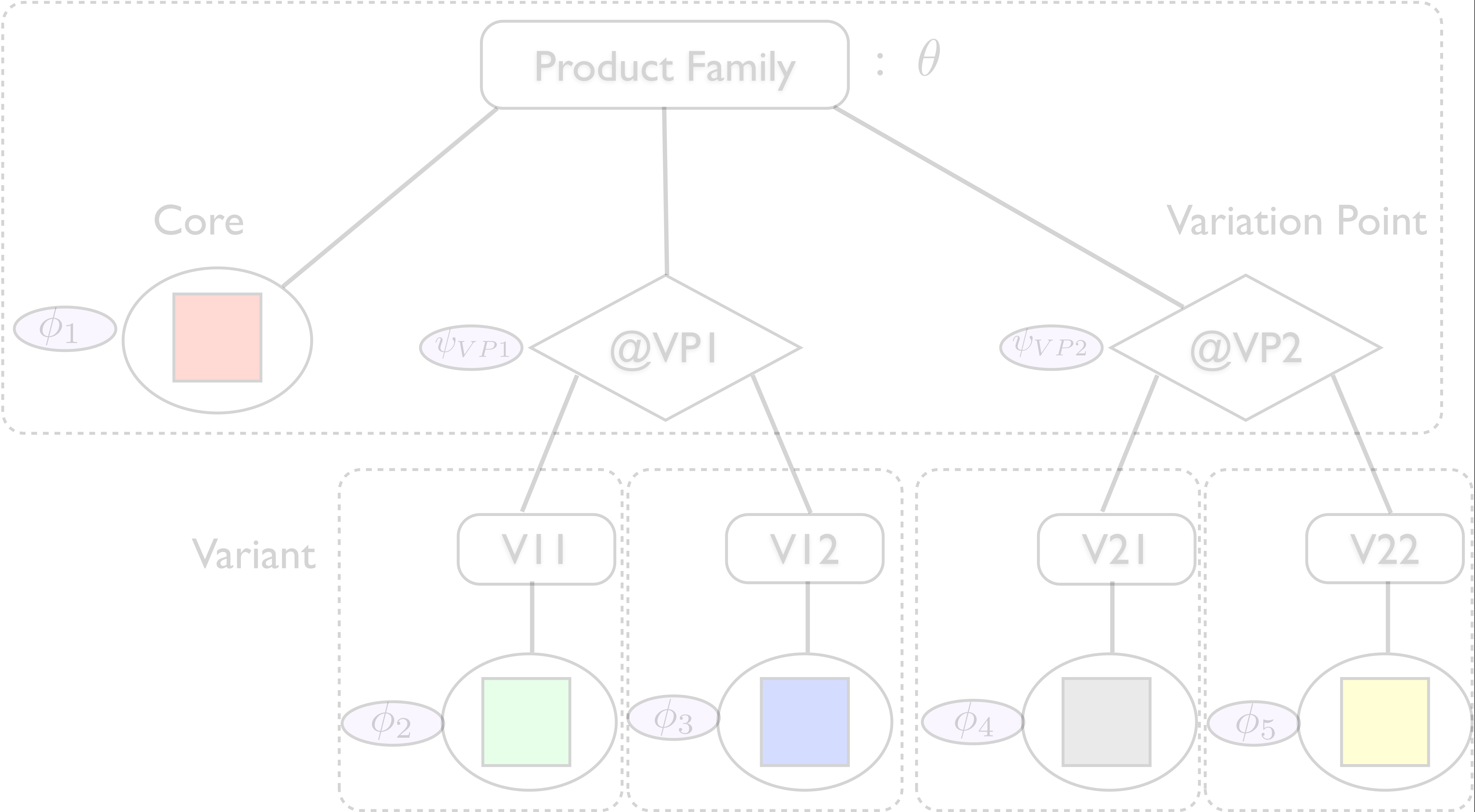
Hierarchical Variability



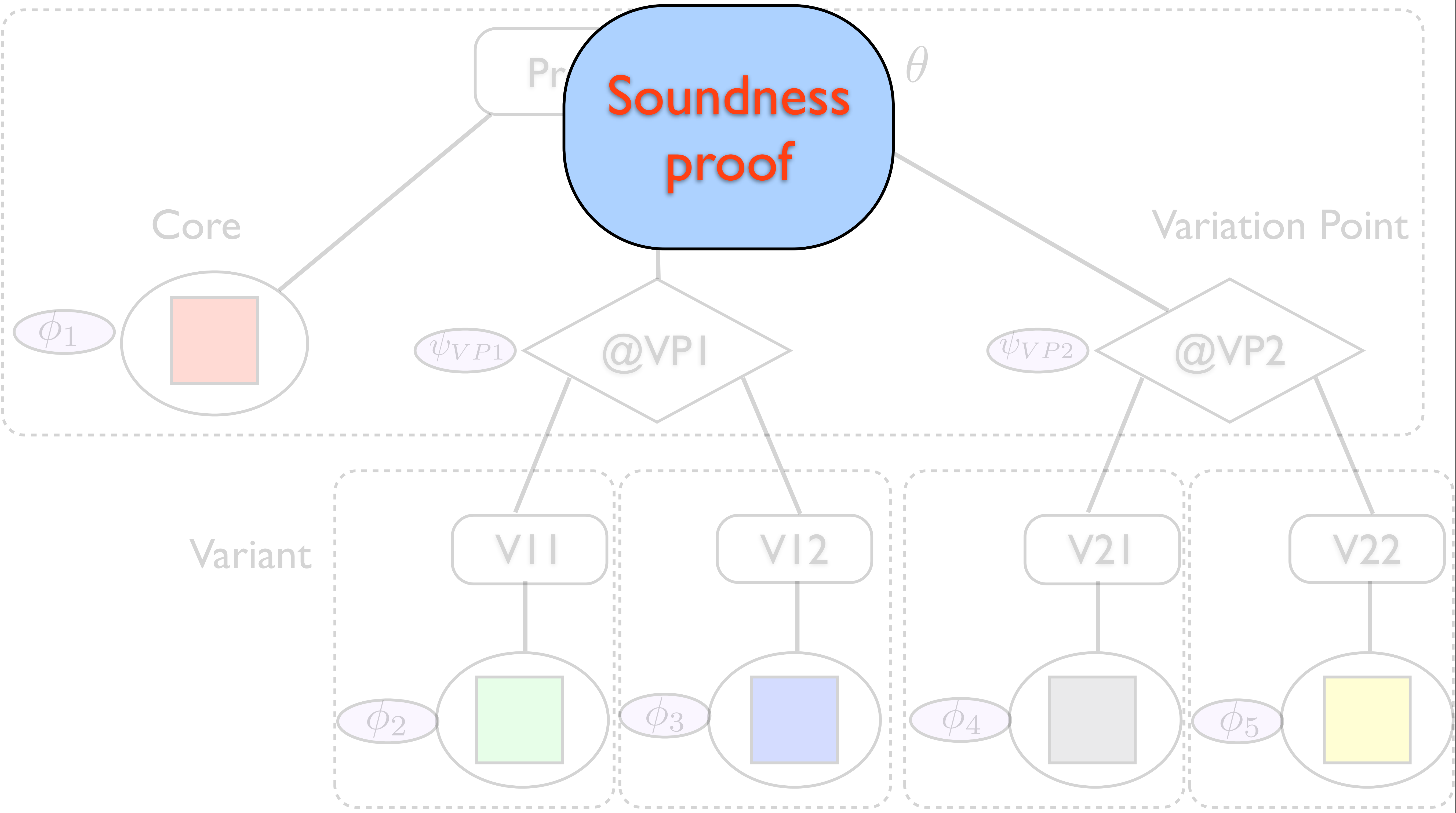
Hierarchical Variability



Hierarchical Variability

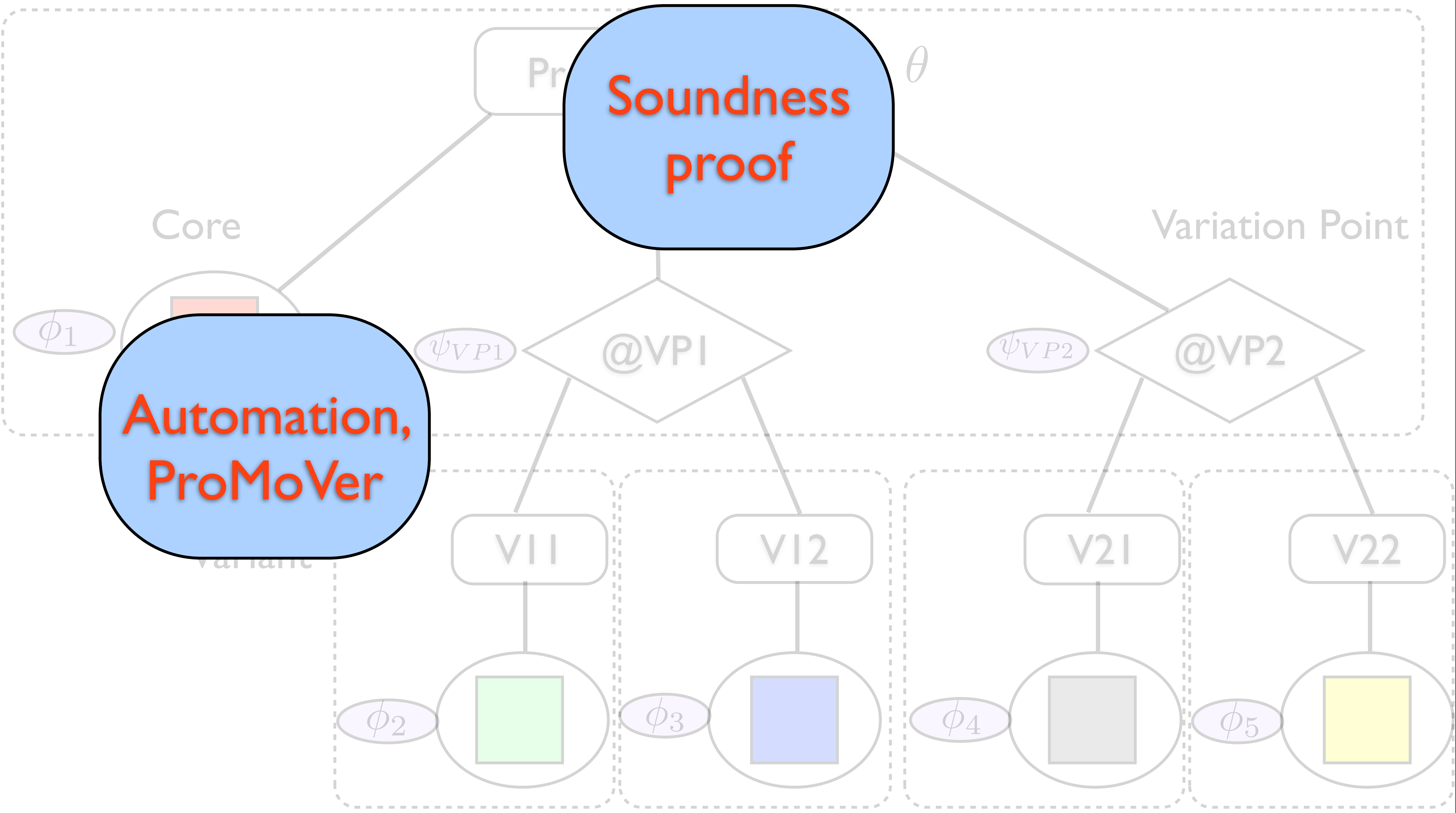


Hierarchical Variability



Soundness proof

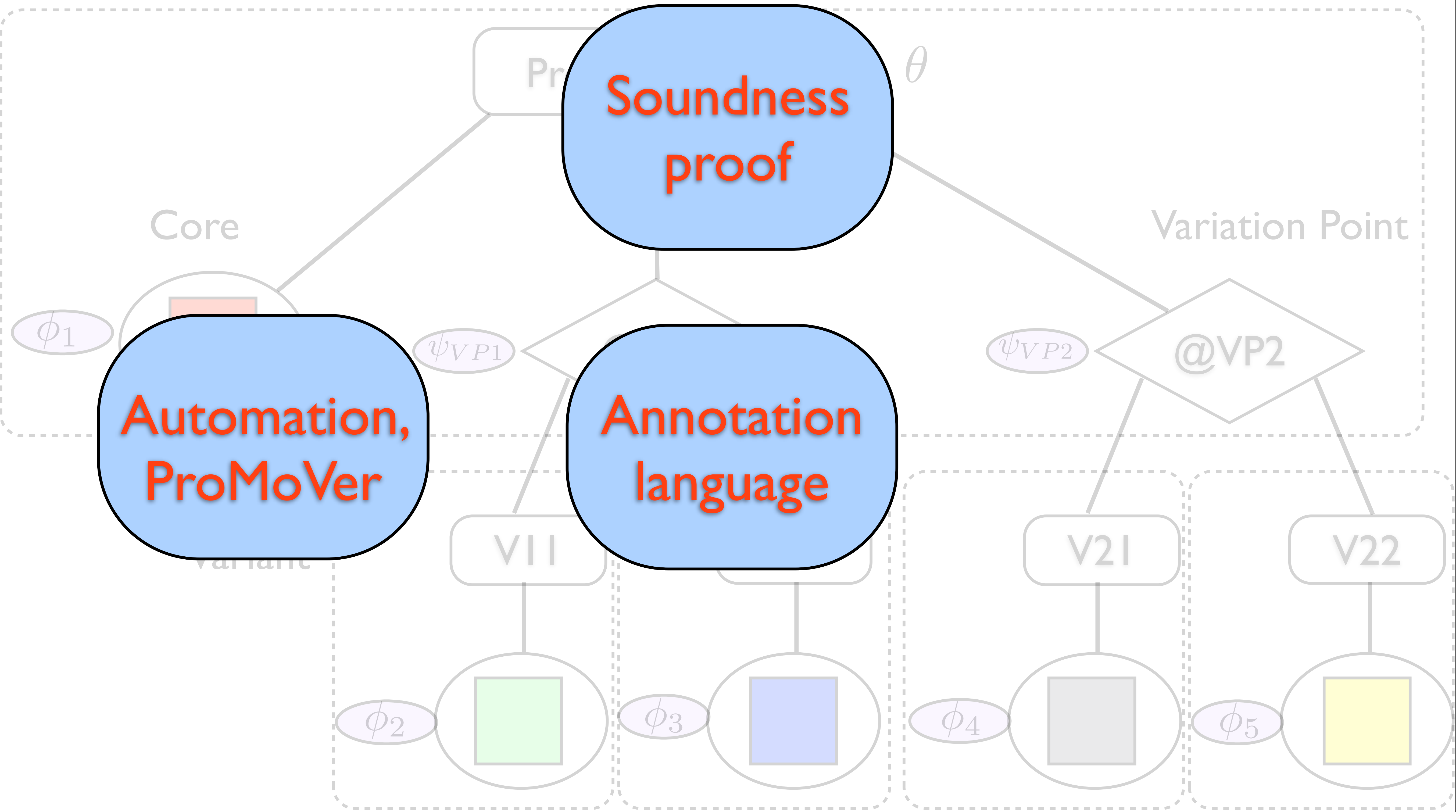
Hierarchical Variability



Soundness proof

Automation, ProMoVer

Hierarchical Variability

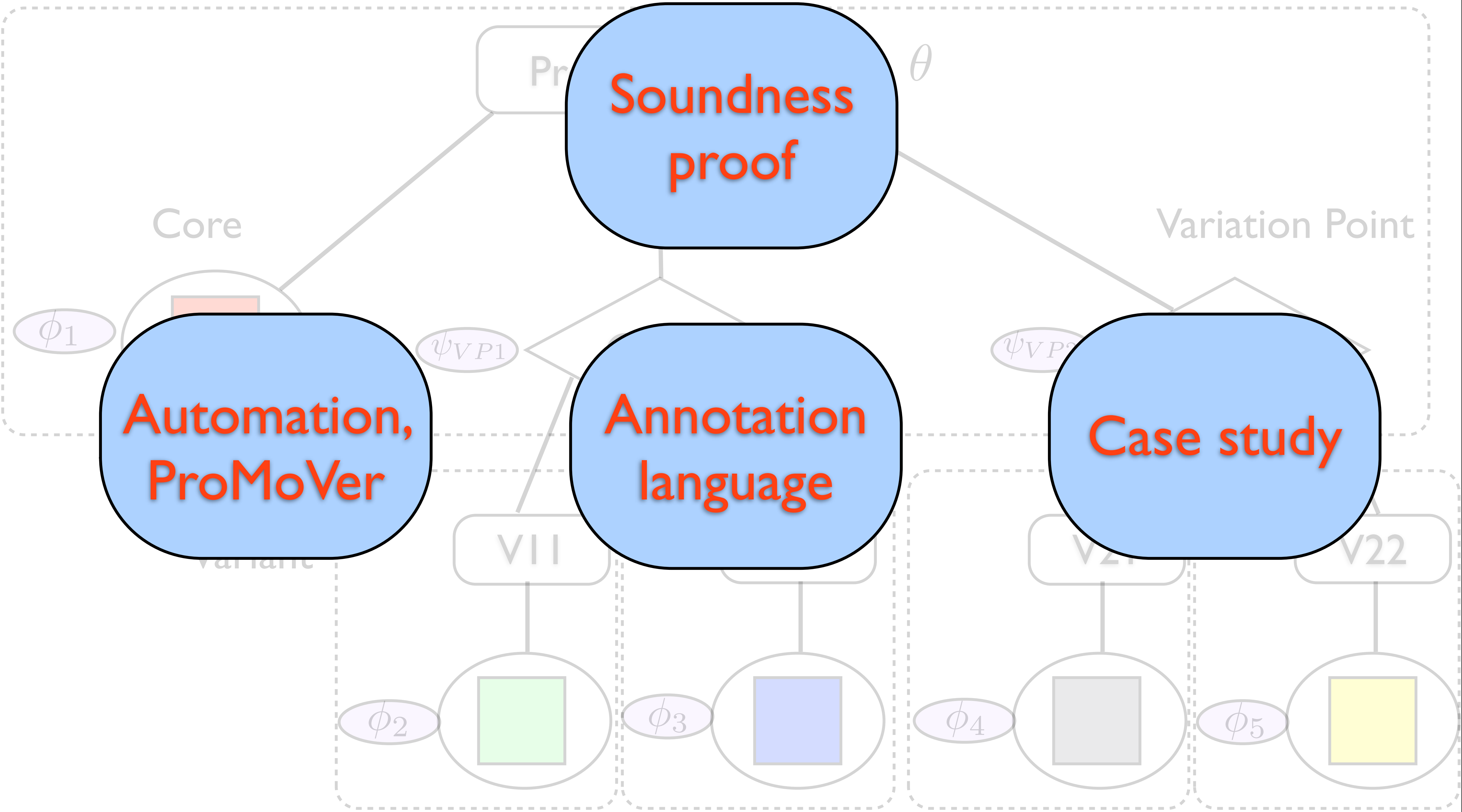


Soundness proof

Automation, ProMoVer

Annotation language

Hierarchical Variability



Case Studies

Case Studies

Application	Depth	Modules	Products	non-comp. Time	comp. Time
Cash Desk	1	7	9	79 sec	9 sec
Cash Desk with Coupons	1	9	18	117 sec	10 sec
Cash Desk with Cards	2	15	27	278 sec	11 sec
Cash Desk with Cards & Coupon	2	17	54	652 sec	12 sec

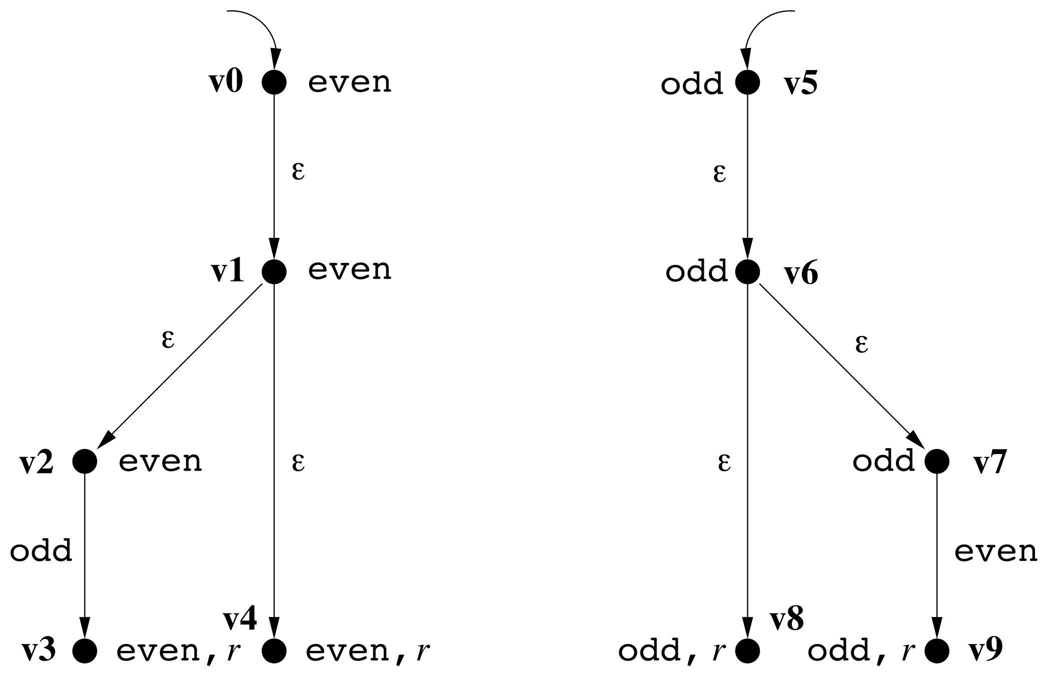
Boolean Flow Graph

Boolean Flow Graphs

- Flow Graphs
  - encoding data through control
    - reuse the CVPP machinery
    - no direct correspondence with the code
- Behaviour extended by passing and returning values
- Maximal model construction with data
- Evaluated by some examples

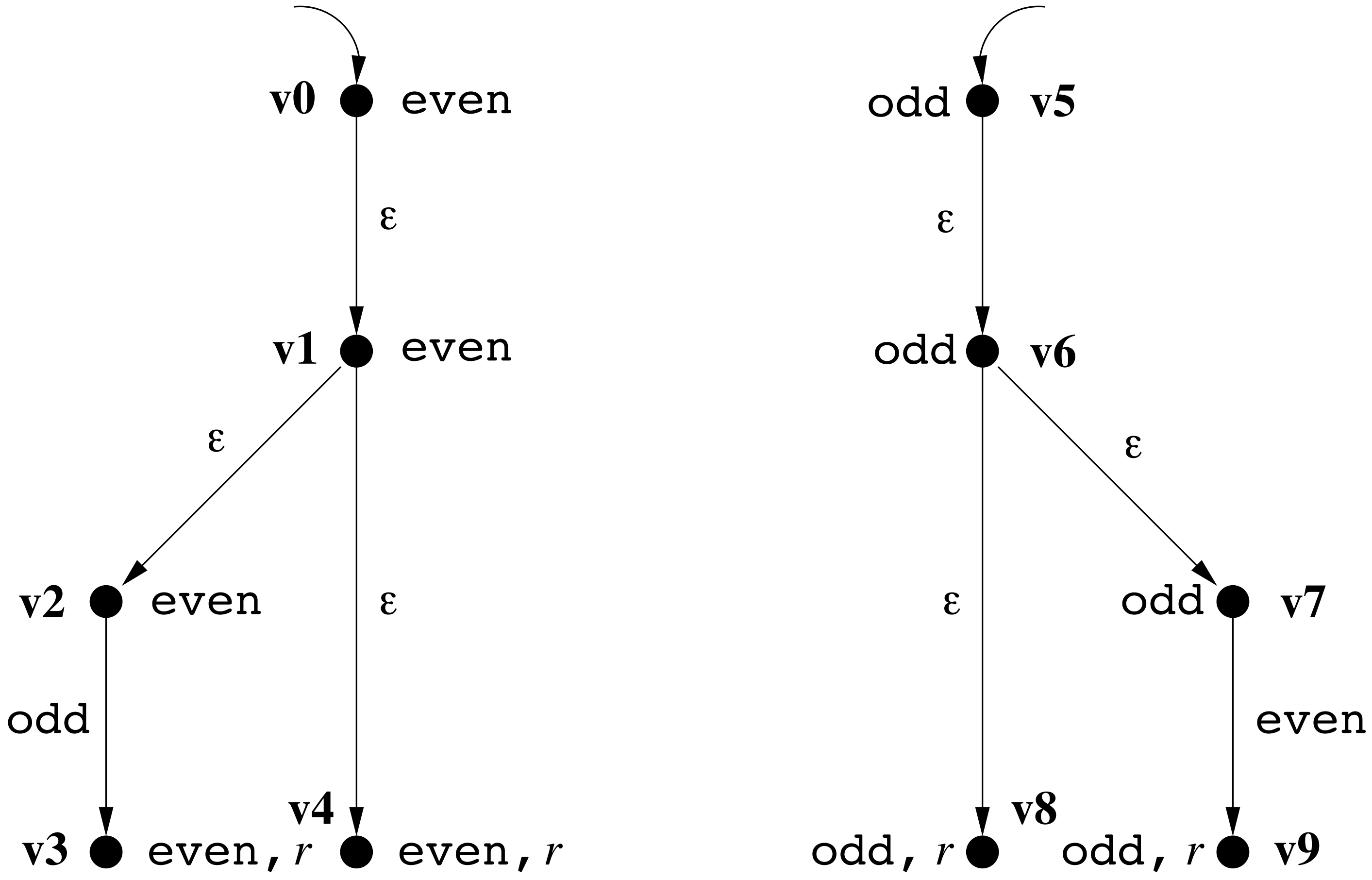
Boolean Flow Graphs

```
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);

   }
}
```



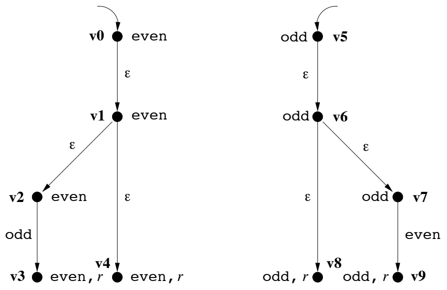
Boolean Flow Graphs

```
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);

   }
}
```



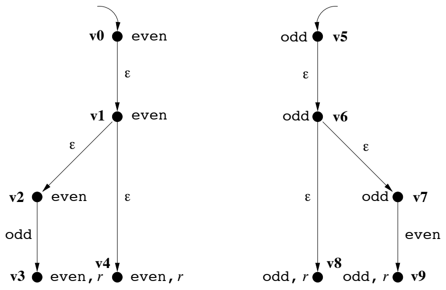
Boolean Flow Graphs

```
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);

   }
}
```



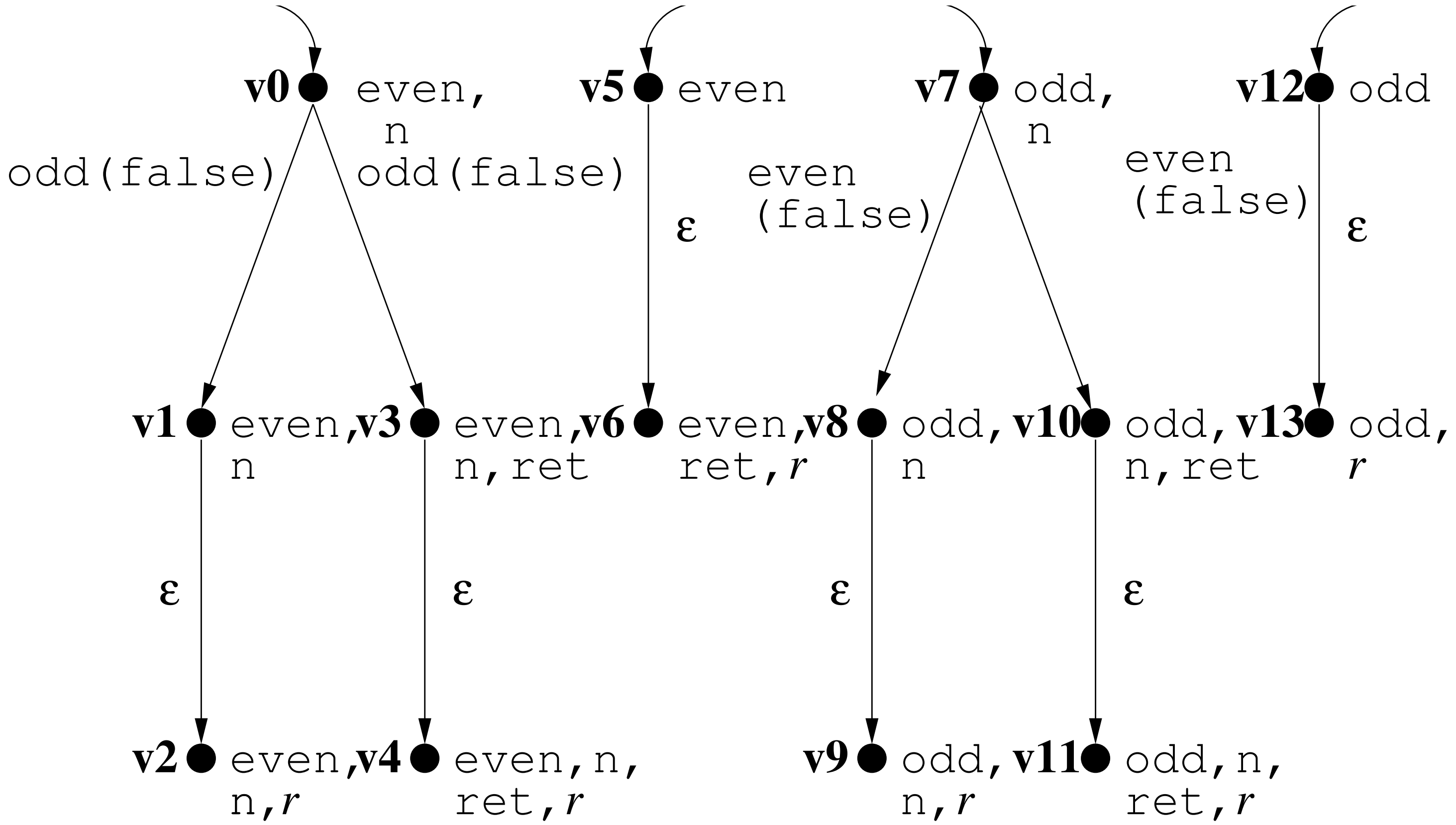
Boolean Flow Graphs

```
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);

   }
}
```



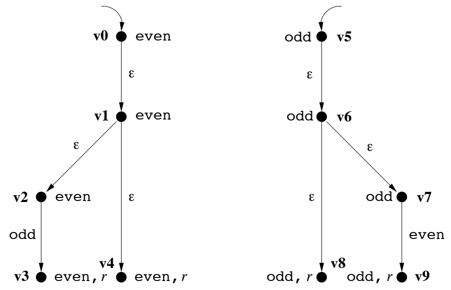
```
bool even(n)
begin
    if (!n) then
        return T;
    else
        return odd(!n);
    fi
end
```

```
bool odd(n)
begin
    if (!n) then
        return F;
    else
        return even(!n);
    fi
end
```

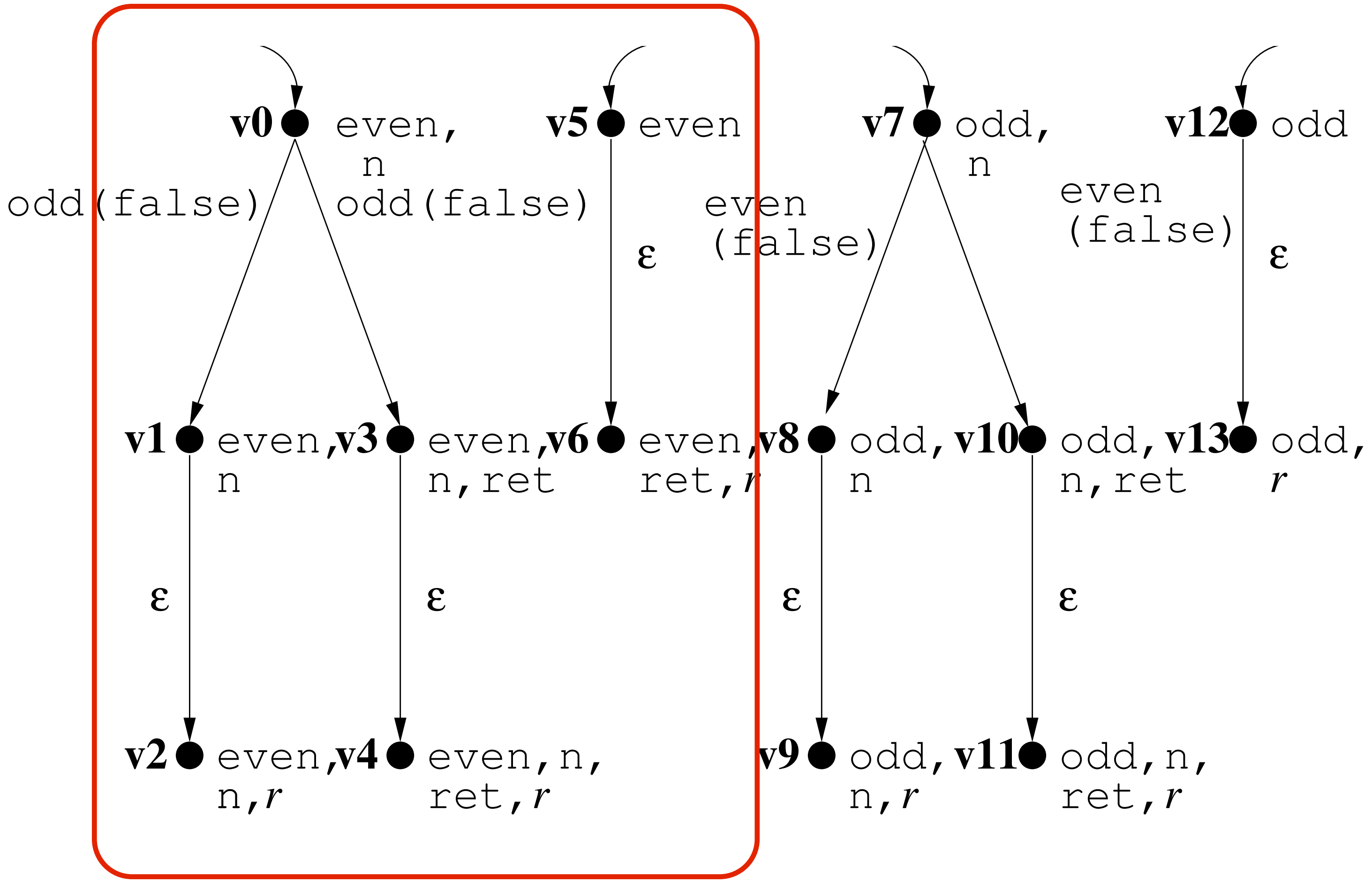


Boolean Flow Graphs

```
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);

   }
}
```

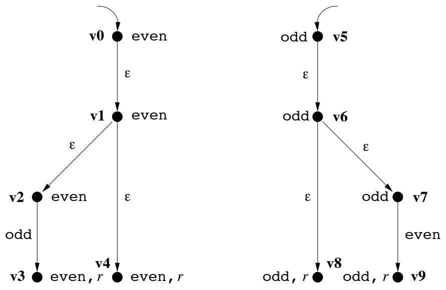


```
bool even(n)
begin
    if (!n) then
        return T;
    else
        return odd(!n);
    fi
end

bool odd(n)
begin
    if (!n) then
        return F;
    else
        return even(!n);
    fi
end
```

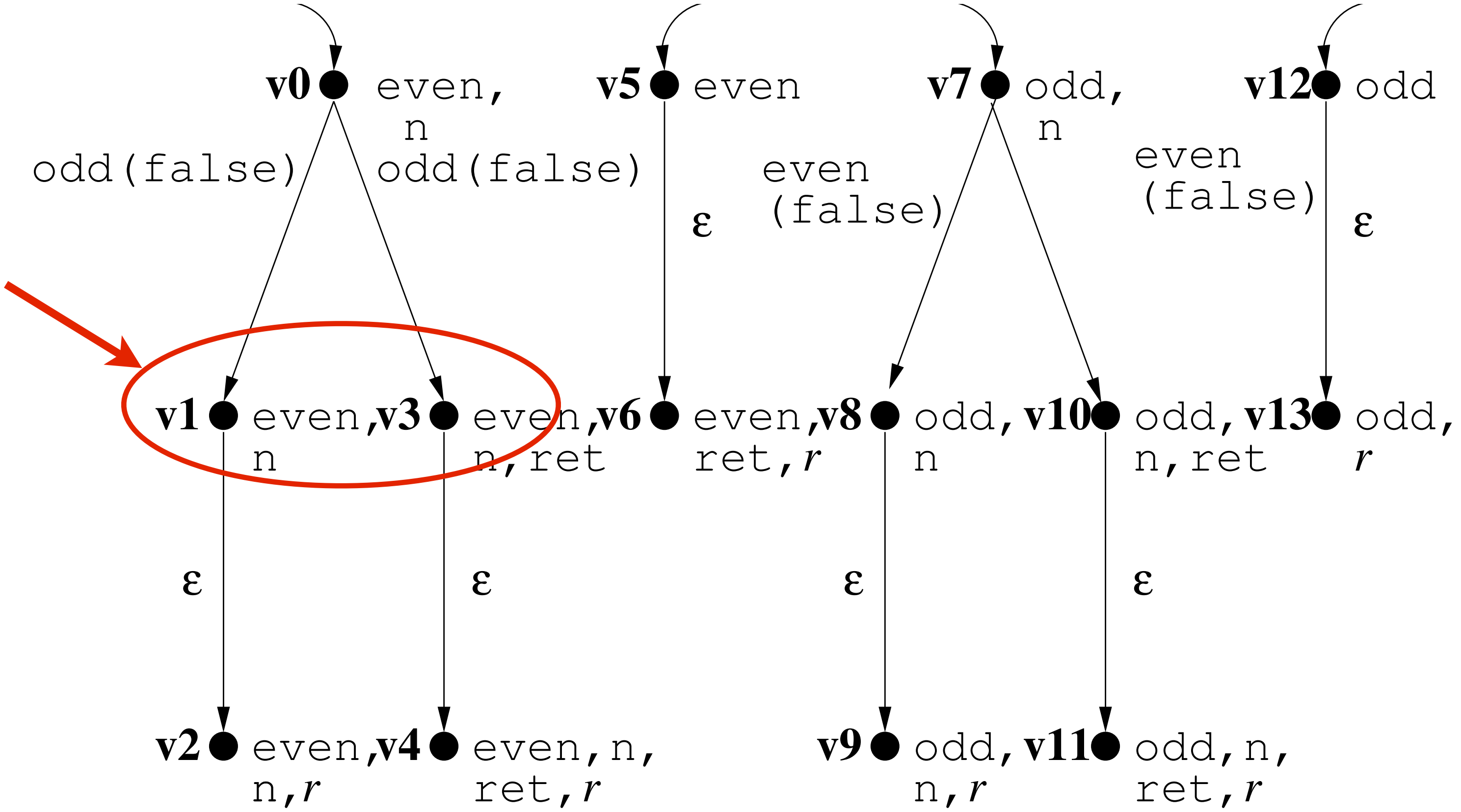


Boolean Flow Graphs

```
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);

   }
}
```



```
bool even(n)
begin
    if (!n) then
        return T;
    else
        return odd(!n);
    fi
end

bool odd(n)
begin
    if (!n) then
        return F;
    else
        return even(!n);
    fi
end
```



Conclusion

- **ProMoVer:** a completely automated tool for procedure-modular verification
  - algorithmic
  - light weight
    - Spec. extractor
    - proof storage & reuse
  - modular: support open systems, variability
  - temporal safety properties
    - meaningful abstraction at procedure level

Conclusion

- **modular verification of product families**
  - hierarchical model
  - compositional verification
- **Boolean flow graphs**
  - encoding finite data through control
  - state-space blow up

Future Work

- ProMoVer
  - support more specification languages
- Product families
  - richer model
  - case study: compare to other approaches
- CVPP framework
  - extend the class of properties by:
    - symbolic data, e.g., Boolean and object references

Future Work -- BOP

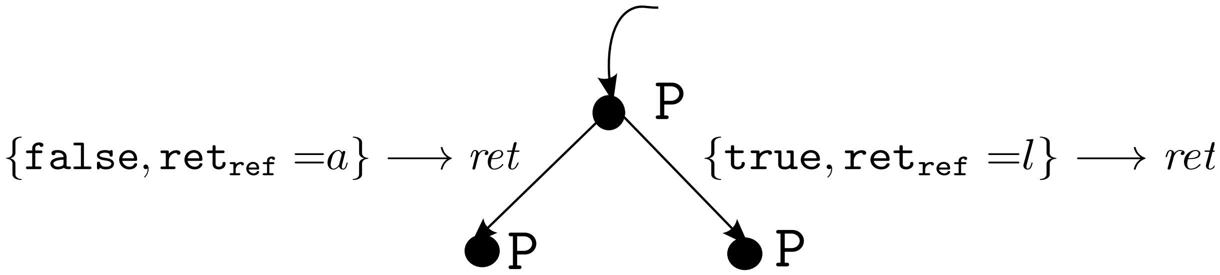
```
decl ref x,y;

void main()
begin
   x := new;
   y := new;
   if (x = y) then y := P(x);
              else x := P(y);
   fi
   del(x);
   del(y);
end

ref P(ref a)
begin
   decl ref l;
   l := a;
   if (l = a) then return l;
              else return a;
   fi
end
```

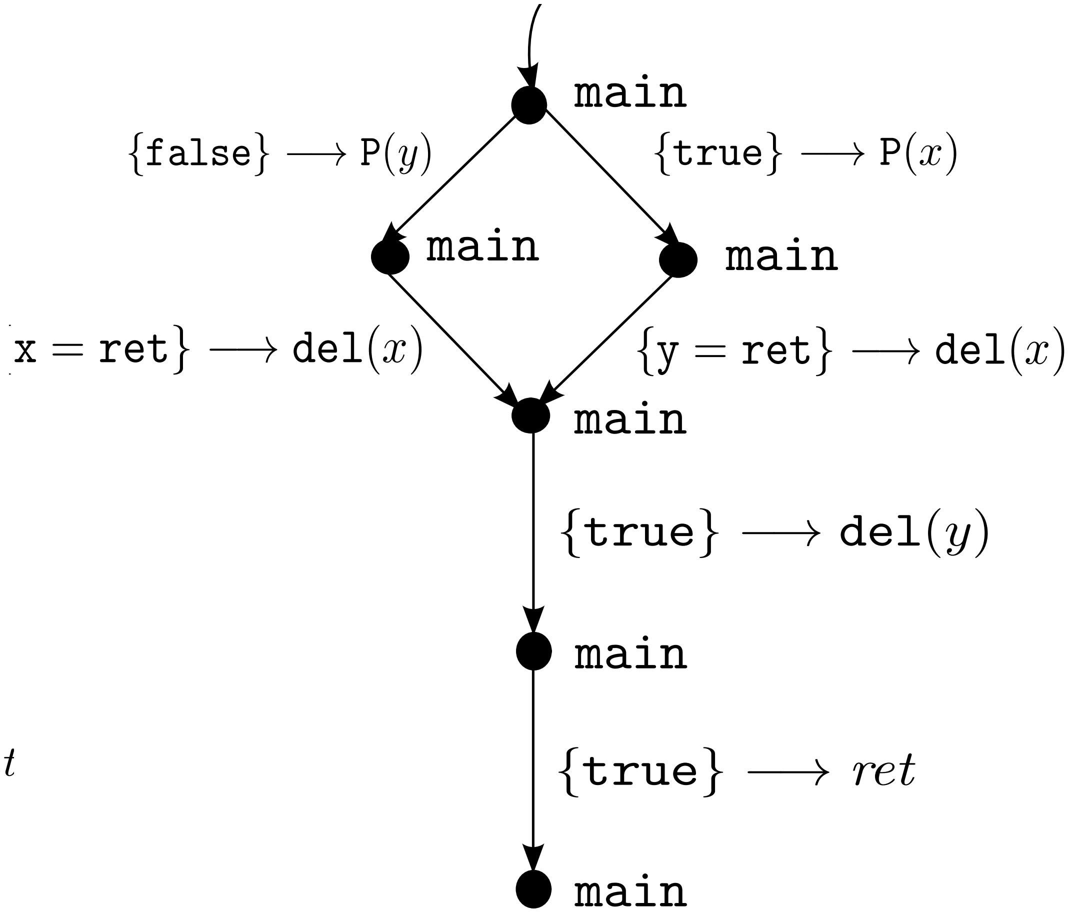
Future Work -- BOP

```
decl ref x,y;

void main()
begin
    x := new;
    y := new;
    if (x = y) then y := P(x);
               else x := P(y);
    fi
    del(x);
    del(y);
end

ref P(ref a)
begin
    decl ref l;
    l := a;
    if (l = a) then return l;
               else return a;
    fi
end
```

Future Work -- BOP

```
decl ref x,y;

void main()
begin
   x := new;
   y := new;
   if (x = y) then y := P(x);
               else x := P(y);
   fi
   del(x);
   del(y);
end

ref P(ref a)
begin
   decl ref l;
   l := a;
   if (l = a) then return l;
               else return a;
   fi
end
```

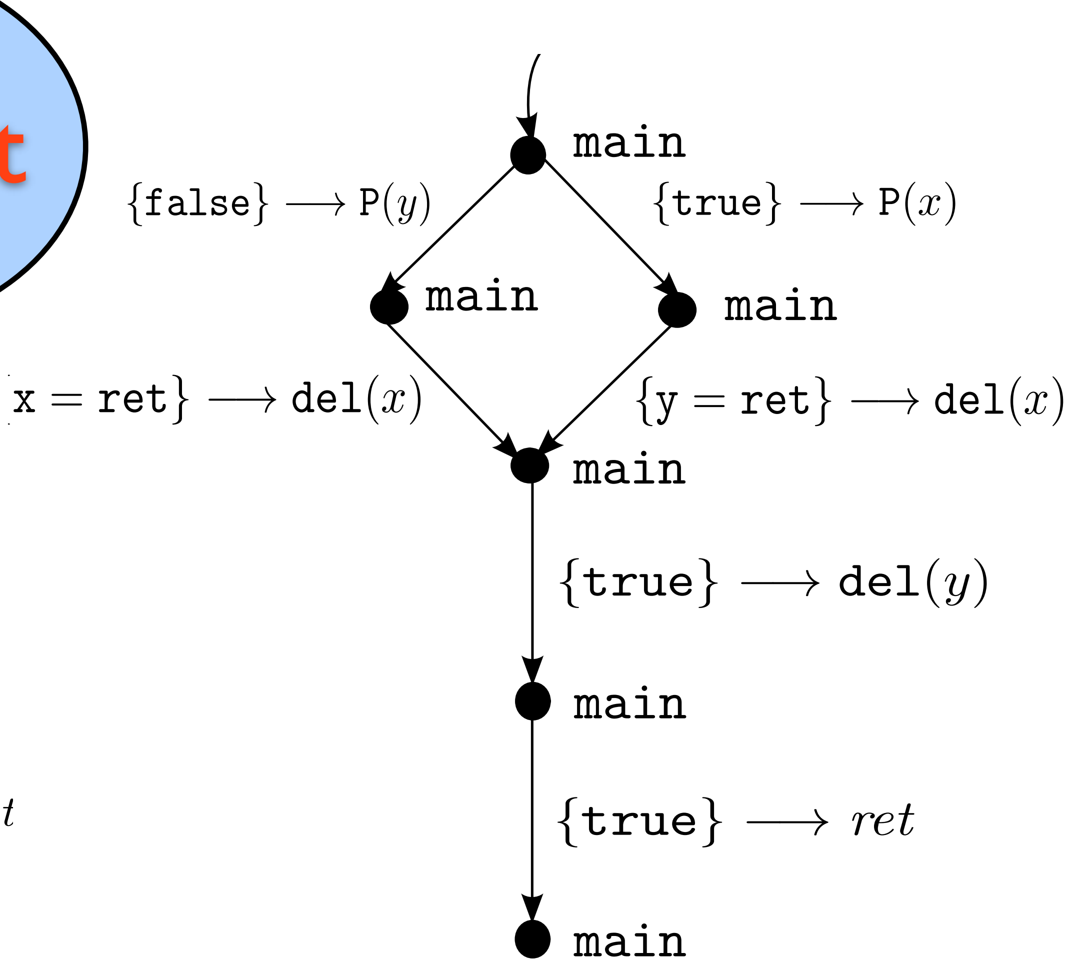

P
$\{\texttt{false}, \texttt{ret}_{\texttt{ref}} = a\} \longrightarrow ret$
$\{\texttt{true}, \texttt{ret}_{\texttt{ref}} = l\} \longrightarrow ret$
P
P

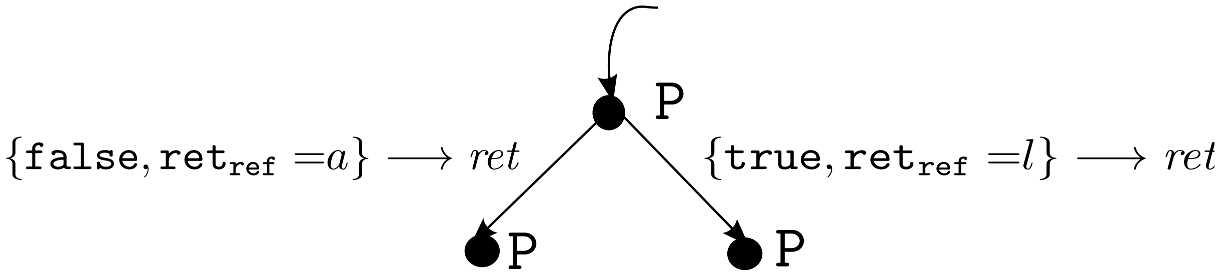

main
$\{\texttt{false}\} \longrightarrow \texttt{P}(y)$
$\{\texttt{true}\} \longrightarrow \texttt{P}(x)$
main
main
$\texttt{x} = \texttt{ret}\} \longrightarrow \texttt{del}(x)$
$\{\texttt{y} = \texttt{ret}\} \longrightarrow \texttt{del}(x)$
main
$\{\texttt{true}\} \longrightarrow \texttt{del}(y)$
main
$\{\texttt{true}\} \longrightarrow ret$
main

Future Work -- BOP

```
decl ref x,y;

void main()
begin
   x := new;
   y := new;
   if (x = y) then y := P(x);
              else x := P(y);
   fi
   del(x);
   del(y);
end

ref P(ref a)
begin
   decl ref l;
   l := a;
   if (l = a) then return l;
              else return a;
   fi
end
```

Abstract



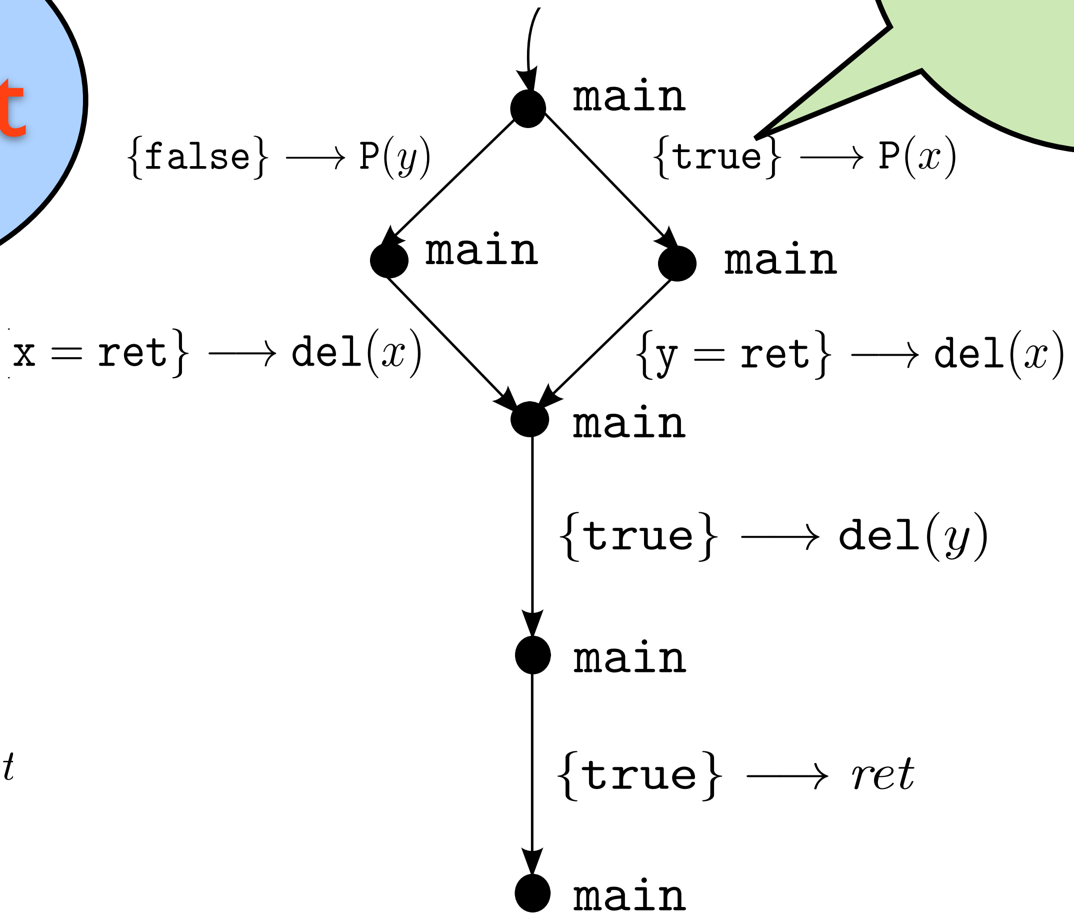


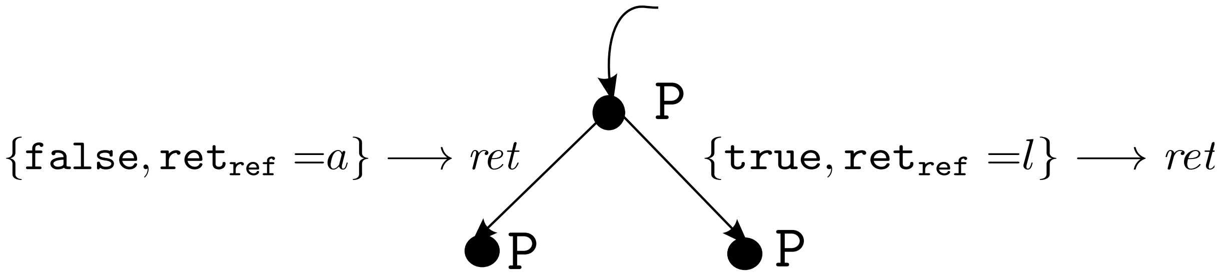
Future Work -- BOP

```
decl ref x,y;

void main()
begin
   x := new;
   y := new;
   if (x = y) then y := P(x);
              else x := P(y);
   fi
   del(x);
   del(y);
end

ref P(ref a)
begin
   decl ref l;
   l := a;
   if (l = a) then return l;
              else return a;
   fi
end
```

Abstract

Conditional calls



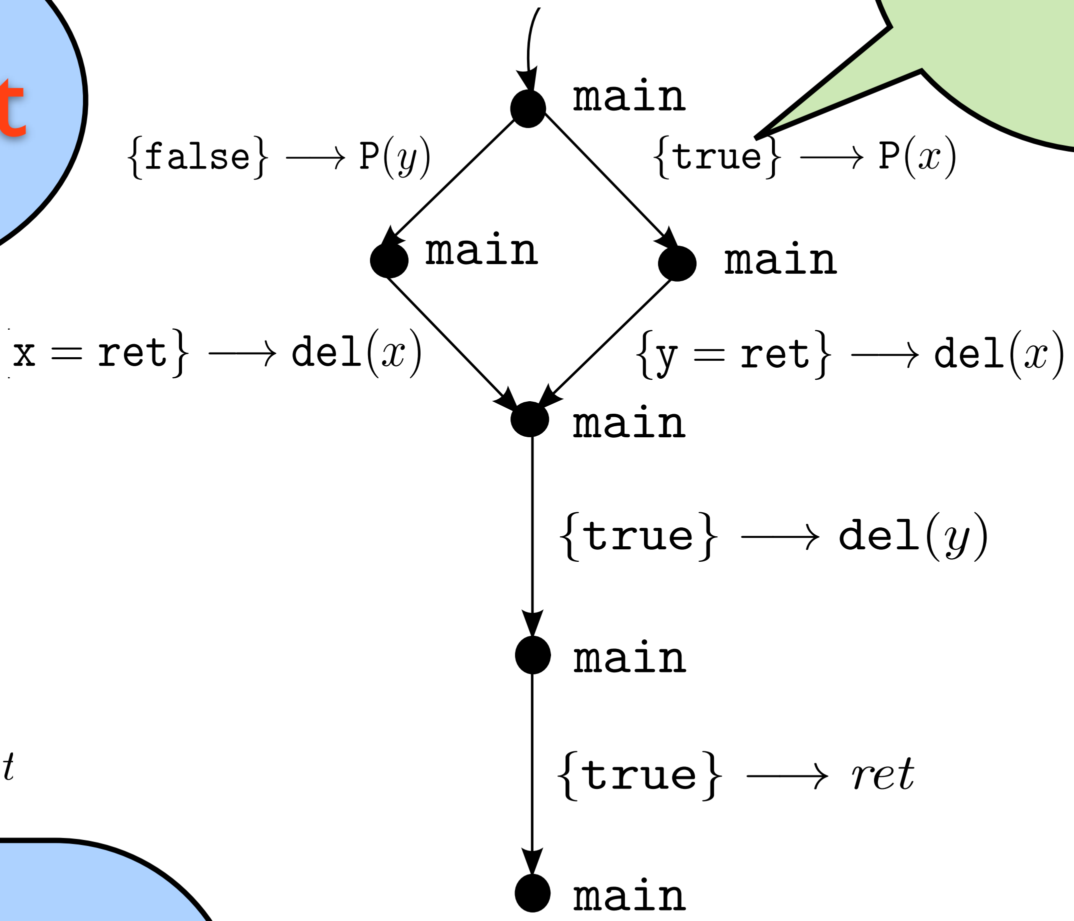


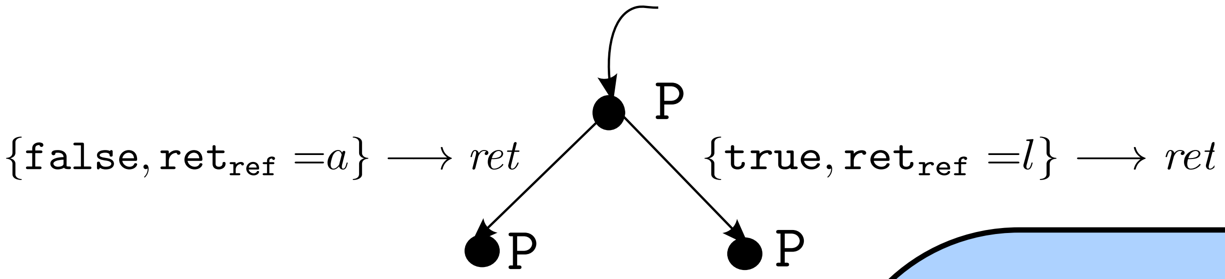
Future Work -- BOP

```
decl ref x,y;

void main()
begin
   x := new;
   y := new;
   if (x = y) then y := P(x);
               else x := P(y);
   fi
   del(x);
   del(y);
end

ref P(ref a)
begin
   decl ref l;
   l := a;
   if (l = a) then return l;
               else return a;
   fi
end
```

Abstract

Conditional calls




P
{false, $\mathtt{ret}_{\mathtt{ref}}$ = a} ⟶ ret
{true, $\mathtt{ret}_{\mathtt{ref}}$ = l} ⟶ ret
P
P

Problem with loops

Thanks for listening!