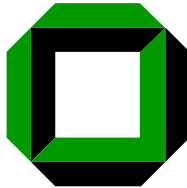


# Software Verification using JML

Prof. P.H. Schmitt

Institut für Theoretische Informatik  
Fakultät für Informatik  
Universität Karlsruhe (TH)



Dagstuhl, May 2006



# The Flight Manager Case Study

This case study has been conducted with

**THALES** Avionics, Toulouse.



# The Flight Manager Case Study

This case study has been conducted with

**THALES** Avionics, Toulouse.

The flight management system assists the pilot in navigating and managing an aircraft.

Among other things, it provides services to construct and maintain a trajectory that is used to define a path between given departure and arrival positions.

It is of great relevance for a safe flight and classified as level B software according to RTCA/DO-178B.



# Motivation

from the OOTiA Handbook issued by  
Federal Aviation Administration (FAA) and  
National Aeronautics and Space Administration (NASA)

## Guideline 3.3.7.3

### **Explicit pre/postcondition/invariant rule:**

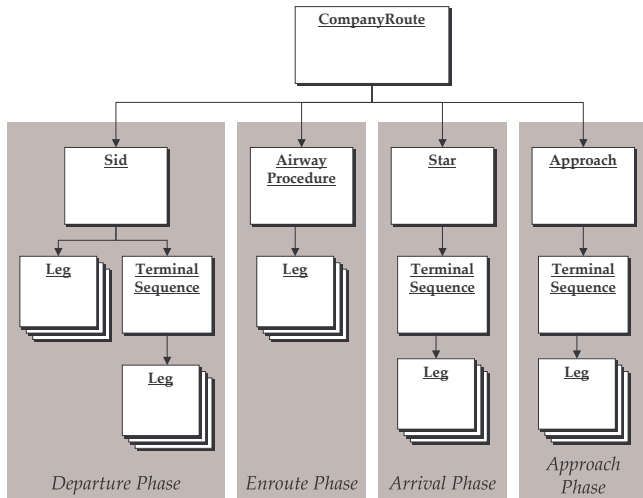
To ensure that all classes define their interfaces as contracts, all pre/postconditions and invariants for operations and methods must be explicitly stated and all errors returned by them must be specified.

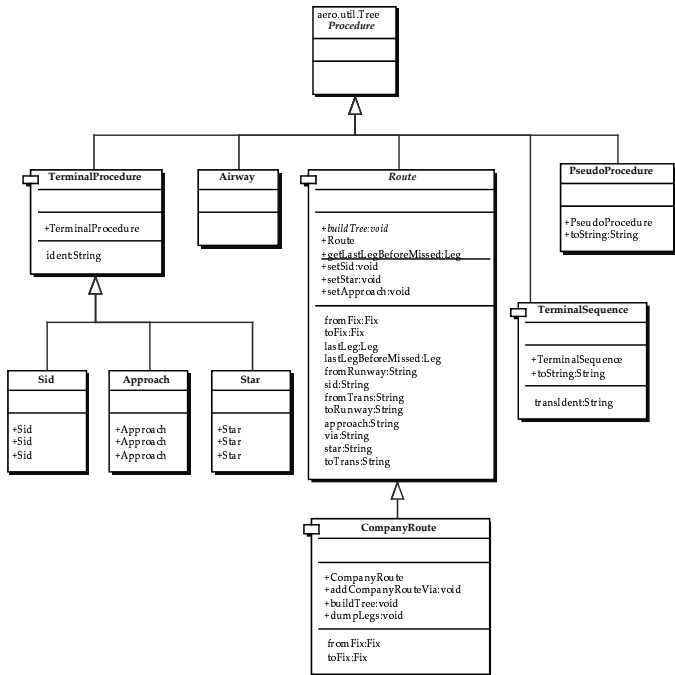
### **Frame condition rule:**

..., ideally each postcondition should also include a *frame condition* which indicates which variables are guaranteed not to change as a result of executing the operation/method.

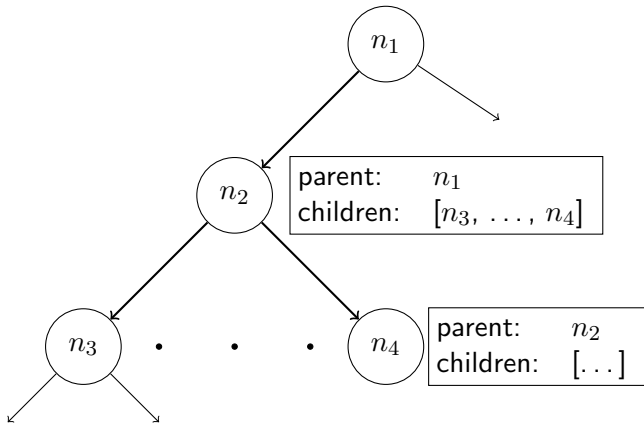


# Top Level Concept





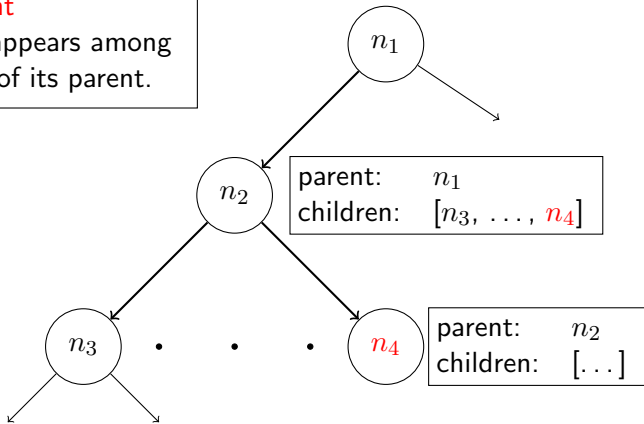
# Invariant for Class Tree



# Invariant for Class Tree

## First Invariant

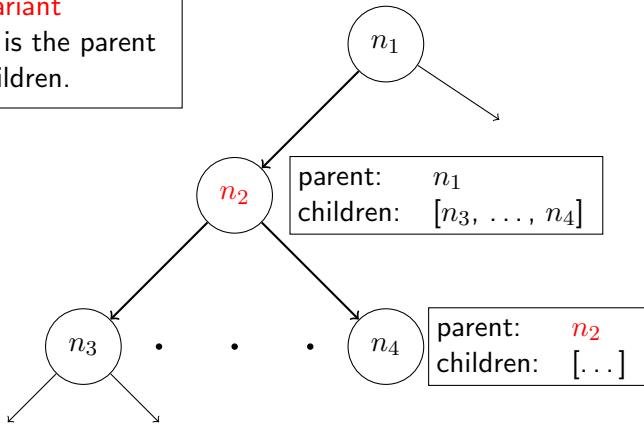
Every node appears among the children of its parent.



# Invariant for Class Tree

## Second Invariant

Every node is the parent of all its children.



## Invariant in JML

```
public class Tree
private /*@ spec_public @*/ List children;
private /*@ spec_public @*/ Tree parent;
/*@ public instance invariant
@   (\ forall int i;
@     0 <= i && i < children.specEnd;
@     ((Tree)(children.specList[i])).parent
@       = this );
@*/

/*@ public instance invariant
@   (parent != null) ==> (\ exists int i;
@     0 <= i && i < parent.children.specEnd;
@     parent.children.specList[i] == this);
@*/
```



# Demo Example

## Method add preserves invariant

```
/*@ public normal_behavior  
  @ requires t instanceof Tree;  
  @ assignable  
  @ children.specList[children.specList.length];  
  @ assignable children.specList;  
  @*/
```

```
public boolean add(Object t) {  
    ((Tree)t).setParent(this);  
    return children.add(t);  
}
```



# A SHORT DEMO NOW



# KeY Project Partners



**Universität Karlsruhe**



**Chalmers University**



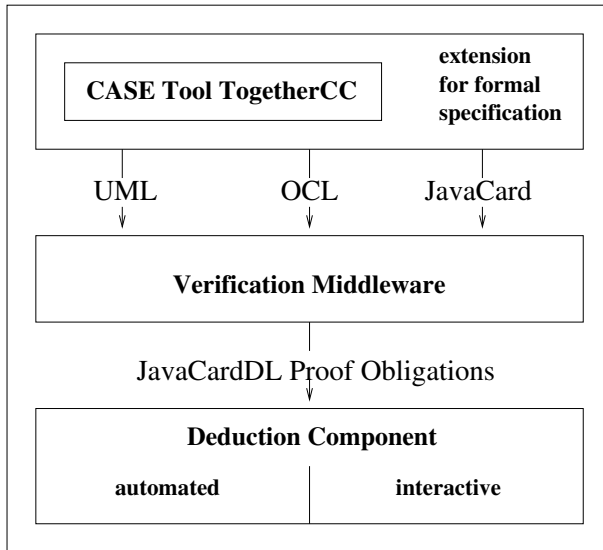
**Universität Koblenz**



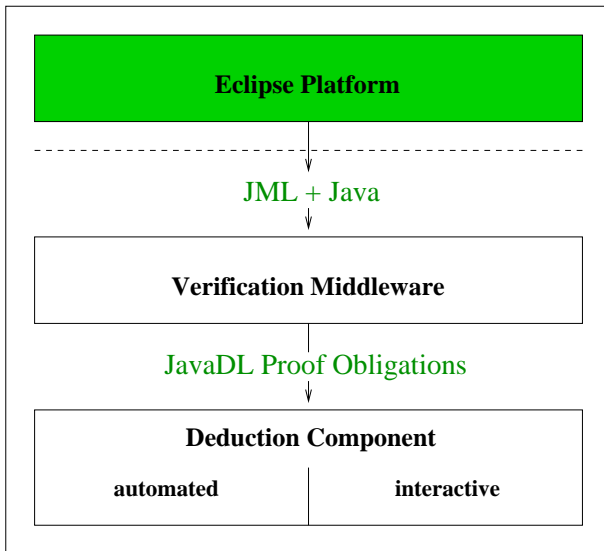
**École Polytechnique Fédérale de Lausanne**



# The KeY System with OCL Frontend



# The KeY System with JML Frontend



# Dynamic Logic

- ▶ A Logic for Reasoning about Programs



# Dynamic Logic

- ▶ A Logic for Reasoning about Programs
- ▶ DL is a multimodal logic



# Dynamic Logic

- ▶ A Logic for Reasoning about Programs
- ▶ DL is a multimodal logic
- ▶ the program states are the possible worlds



# Dynamic Logic

- ▶ A Logic for Reasoning about Programs
- ▶ DL is a multimodal logic
- ▶ the program states are the possible worlds
- ▶ two modalities  $[\alpha]$  and  $\langle\alpha\rangle$  for each program  $\alpha$



# Dynamic Logic

- ▶ A Logic for Reasoning about Programs
- ▶ DL is a multimodal logic
- ▶ the program states are the possible worlds
- ▶ two modalities  $[\alpha]$  and  $\langle\alpha\rangle$  for each program  $\alpha$



# Dynamic Logic

- ▶ A Logic for Reasoning about Programs
- ▶ DL is a multimodal logic
- ▶ the program states are the possible worlds
- ▶ two modalities  $[\alpha]$  and  $\langle\alpha\rangle$  for each program  $\alpha$

Semantics



# Dynamic Logic

- ▶ A Logic for Reasoning about Programs
- ▶ DL is a multimodal logic
- ▶ the program states are the possible worlds
- ▶ two modalities  $[\alpha]$  and  $\langle\alpha\rangle$  for each program  $\alpha$

## Semantics

- ▶  $[\alpha]F$  is true in state  $s$  iff  
 $F$  is true in all states  $\alpha$ -reachable from  $s$   
partial correctness



# Dynamic Logic

- ▶ A Logic for Reasoning about Programs
- ▶ DL is a multimodal logic
- ▶ the program states are the possible worlds
- ▶ two modalities  $[\alpha]$  and  $\langle\alpha\rangle$  for each program  $\alpha$

## Semantics

- ▶  $[\alpha]F$  is true in state  $s$  iff  
 $F$  is true in all states  $\alpha$ -reachable from  $s$   
partial correctness
- ▶  $\langle\alpha\rangle F$  is true in state  $s$  iff  
 $F$  is true in some state  $\alpha$ -reachable from  $s$   
total correctness



# Simple Examples

▶  $\langle x = 1; \rangle x \doteq 1$

true in all states



# Simple Examples

▶  $\langle x = 1; \rangle x \doteq 1$

▶  $[\text{while}(\text{true})\{\alpha\}] \text{false}$

true in all states

true in all states



# Simple Examples

- ▶  $\langle x = 1; \rangle x \doteq 1$  true in all states
- ▶  $[\text{while}(\text{true})\{\alpha\}] \text{false}$  true in all states
- ▶  $\langle \alpha \rangle \text{true}$  true if  $\alpha$  terminates



# Simple Examples

- ▶  $\langle x = 1; \rangle x \doteq 1$
- ▶  $[\text{while}(\text{true})\{\alpha\}] \text{ false}$
- ▶  $\langle \alpha \rangle \text{ true}$
- ▶  $\langle \alpha_1 \rangle \text{ true} \rightarrow \langle \alpha_2 \rangle \text{ true}$

true in all states

true in all states

true if  $\alpha$  terminates

says: if  $\alpha_1$  terminates  
then  $\alpha_2$  terminates.



# Simple Examples

- ▶  $\langle x = 1; \rangle x \doteq 1$  true in all states
- ▶  $[\text{while}(\text{true})\{\alpha\}] \text{false}$  true in all states
- ▶  $\langle \alpha \rangle \text{true}$  true if  $\alpha$  terminates
- ▶  $\langle \alpha_1 \rangle \text{true} \rightarrow \langle \alpha_2 \rangle \text{true}$  says: if  $\alpha_1$  terminates then  $\alpha_2$  terminates.
- ▶ **Hoare Triples**  
 $\{F_1\}\alpha\{F_2\}$  same as  $F_1 \rightarrow [\alpha]F_2$



# Simple Examples

- ▶  $\langle x = 1; \rangle x \doteq 1$  true in all states
- ▶  $[\text{while}(\text{true})\{\alpha\}] \text{false}$  true in all states
- ▶  $\langle \alpha \rangle \text{true}$  true if  $\alpha$  terminates
- ▶  $\langle \alpha_1 \rangle \text{true} \rightarrow \langle \alpha_2 \rangle \text{true}$  says: if  $\alpha_1$  terminates then  $\alpha_2$  terminates.
- ▶ **Hoare Triples**  
 $\{F_1\}\alpha\{F_2\}$  same as  $F_1 \rightarrow [\alpha]F_2$
- ▶ **Duality of modal operators**  
 $[\alpha]F \leftrightarrow \neg\langle \alpha \rangle\neg F$



# Hoare's Assignment Rule

$$\frac{\Gamma(z/x), x \doteq t(z/x) \Rightarrow F, \Delta(z/x)}{\Gamma \Rightarrow \langle x = t \rangle F, \Delta}$$

where  $x$  and  $t$  are of type  $s$

and  $z$  is a **new** variable of type  $s$ .



# Hoare's Assignment Rule

$$\frac{\Gamma(z/x), x \doteq t(z/x) \Rightarrow F, \Delta(z/x)}{\Gamma \Rightarrow \langle x = t \rangle F, \Delta}$$

where  $x$  and  $t$  are of type  $s$

and  $z$  is a **new** variable of type  $s$ .

Works well for simple programming languages.



# Hoare's Assignment Rule

$$\frac{\Gamma(z/x), x \doteq t(z/x) \Rightarrow F, \Delta(z/x)}{\Gamma \Rightarrow \langle x = t \rangle F, \Delta}$$

where  $x$  and  $t$  are of type  $s$   
and  $z$  is a **new** variable of type  $s$ .

Works well for simple programming languages.

Not suitable for real-life programming languages with features like



# Hoare's Assignment Rule

$$\frac{\Gamma(z/x), x \doteq t(z/x) \Rightarrow F, \Delta(z/x)}{\Gamma \Rightarrow \langle x = t \rangle F, \Delta}$$

where  $x$  and  $t$  are of type  $s$   
and  $z$  is a **new** variable of type  $s$ .

Works well for simple programming languages.

Not suitable for real-life programming languages with features like

- ▶ aliasing
- ▶ abrupt termination
- ▶ exceptions
- ▶ atomicity (transaction) concept



# Assignments via Updates

$$\frac{\Gamma \Rightarrow \mathcal{U}\{loc := val\} \langle \pi \ \omega \rangle \phi, \Delta}{\Gamma \Rightarrow \mathcal{U} \langle \pi \ loc = val; \ \omega \rangle \phi, \Delta}$$



# Assignments via Updates

$$\frac{\Gamma \Rightarrow \mathcal{U}\{loc := val\} \langle \pi \ \omega \rangle \phi, \Delta}{\Gamma \Rightarrow \mathcal{U} \langle \pi \ loc = val; \ \omega \rangle \phi, \Delta}$$

$\pi$  is inactive prefix



# Assignments via Updates

$$\frac{\Gamma \Rightarrow \mathcal{U}\{loc := val\} \langle \pi \ \omega \rangle \phi, \Delta}{\Gamma \Rightarrow \mathcal{U}\langle \pi \ \mathit{loc} = \mathit{val}; \ \omega \rangle \phi, \Delta}$$

$\pi$  is inactive prefix

$loc := val$  first active statement



# Assignments via Updates

$$\frac{\Gamma \Rightarrow \mathcal{U}\{loc := val\} \langle \pi \ \omega \rangle \phi, \Delta}{\Gamma \Rightarrow \mathcal{U} \langle \pi \ loc = val; \ \omega \rangle \phi, \Delta}$$

$\pi$  is inactive prefix

$loc := val$  first active statement

$\omega$  is the rest of the program



# Assignments via Updates

$$\frac{\Gamma \Rightarrow \mathcal{U}\{loc := val\} \langle \pi \ \omega \rangle \phi, \Delta}{\Gamma \Rightarrow \mathcal{U} \langle \pi \ loc = val; \ \omega \rangle \phi, \Delta}$$

$\pi$  is inactive prefix

$loc := val$  first active statement

$\omega$  is the rest of the program

$\mathcal{U}$  is an arbitrary sequence of updates



## Assignments via Updates

$$\frac{\Gamma \Rightarrow \mathcal{U}\{loc := val\} \langle \pi \ \omega \rangle \phi, \Delta}{\Gamma \Rightarrow \mathcal{U} \langle \pi \ loc = val; \ \omega \rangle \phi, \Delta}$$

$\pi$  is inactive prefix

$loc := val$  first active statement

$\omega$  is the rest of the program

$\mathcal{U}$  is an arbitrary sequence of updates

An update  $loc := val$  is an assignment  $\langle loc = val; \rangle$  where



# Assignments via Updates

$$\frac{\Gamma \Rightarrow \mathcal{U}\{loc := val\} \langle \pi \ \omega \rangle \phi, \Delta}{\Gamma \Rightarrow \mathcal{U} \langle \pi \ loc = val; \ \omega \rangle \phi, \Delta}$$

$\pi$  is inactive prefix

$loc := val$  first active statement

$\omega$  is the rest of the program

$\mathcal{U}$  is an arbitrary sequence of updates

An update  $loc := val$  is an assignment  $\langle loc = val; \rangle$  where

- ▶  $loc$  is free of side effects and is
  - ▶ a program variable  $var$ , or
  - ▶ a field access  $obj.attr$ , or
  - ▶ an array access  $arr[i]$ ;



# Assignments via Updates

$$\frac{\Gamma \Rightarrow \mathcal{U}\{loc := val\} \langle \pi \ \omega \rangle \phi, \Delta}{\Gamma \Rightarrow \mathcal{U} \langle \pi \ loc = val; \ \omega \rangle \phi, \Delta}$$

$\pi$  is inactive prefix

$loc := val$  first active statement

$\omega$  is the rest of the program

$\mathcal{U}$  is an arbitrary sequence of updates

An update  $loc := val$  is an assignment  $\langle loc = val; \rangle$  where

- ▶  $loc$  is free of side effects and is
  - ▶ a program variable  $var$ , or
  - ▶ a field access  $obj.attr$ , or
  - ▶ an array access  $arr[i]$ ;
- ▶ and  $val$  is a term free of side effects.



# Definition of Updates

- ▶ (function update)  $f(t_1, \dots, t_n) := t$   
for any term  $f(t_1, \dots, t_n)$  with non-rigid leading function symbol  $f$  with result type  $A$  and  $t$  a term of type  $A'$  with  $A' \sqsubseteq A$ ,



# Definition of Updates

- ▶ (function update)  $f(t_1, \dots, t_n) := t$   
for any term  $f(t_1, \dots, t_n)$  with non-rigid leading function symbol  $f$  with result type  $A$  and  $t$  a term of type  $A'$  with  $A' \sqsubseteq A$ ,
- ▶ (sequential update)  $u_1 ; u_2$   
for updates  $u_1, u_2$ ,



# Definition of Updates

- ▶ (function update)  $f(t_1, \dots, t_n) := t$   
for any term  $f(t_1, \dots, t_n)$  with non-rigid leading function symbol  $f$  with result type  $A$  and  $t$  a term of type  $A'$  with  $A' \sqsubseteq A$ ,
- ▶ (sequential update)  $u_1 ; u_2$   
for updates  $u_1, u_2$ ,
- ▶ (parallel update)  $u_1 \parallel u_2$   
for updates  $u_1, u_2$ ,



# Definition of Updates

- ▶ (function update)  $f(t_1, \dots, t_n) := t$   
for any term  $f(t_1, \dots, t_n)$  with non-rigid leading function symbol  $f$  with result type  $A$  and  $t$  a term of type  $A'$  with  $A' \sqsubseteq A$ ,
- ▶ (sequential update)  $u_1 ; u_2$   
for updates  $u_1, u_2$ ,
- ▶ (parallel update)  $u_1 \parallel u_2$   
for updates  $u_1, u_2$ ,
- ▶ (quantified update)  $\text{for } x; \varphi; u$   
for any update  $u$ , variable  $x$  and DLFormula  $\varphi$ .



# Demo Example

## Method add preserves invariant

```
/*@ public normal_behavior  
  @ requires t instanceof Tree;  
  @ assignable  
  @ children.specList[children.specList.length];  
  @ assignable children.specList;  
  @*/
```

```
public boolean add(Object t) {  
    ((Tree)t).setParent(this);  
    return children.add(t);  
}
```



# Modifier Sets

## Definition

A set of terms

$$Mod = \{f_1(t_1), \dots, f_n(t_n)\}$$



# Modifier Sets

## Definition

A set of terms

$$Mod = \{f_1(t_1), \dots, f_n(t_n)\}$$

is a correct modifier set for a program  $\alpha$



# Modifier Sets

## Definition

A set of terms

$$Mod = \{f_1(t_1), \dots, f_n(t_n)\}$$

is a correct modifier set for a program  $\alpha$

if for all states  $s_1, s_2$  with  $s_2 \in \text{poststates}(s_1, \alpha)$



# Modifier Sets

## Definition

A set of terms

$$Mod = \{f_1(t_1), \dots, f_n(t_n)\}$$

is a correct modifier set for a program  $\alpha$

if for all states  $s_1, s_2$  with  $s_2 \in \text{poststates}(s_1, \alpha)$

$$f^{s_1}(o) \neq f^{s_2}(o)$$

only if there is  $1 \leq i \leq n$  with

- ▶  $f = f_i$
- ▶ and  $t^{s_1}(t_i) = o$



# Anonymous Updates

## Definition

Let  $Mod = \{f_1(t_1), \dots, f_n(t_n)\}$  be a modifier set for program  $\alpha$ .



# Anonymous Updates

## Definition

Let  $Mod = \{f_1(t_1), \dots, f_n(t_n)\}$  be a modifier set for program  $\alpha$ .

The anonymous update  $\mathcal{V}(Mod)$  consists of the updates

$$f_i(t_i) := f'_i(t_i)$$



# Anonymous Updates

## Definition

Let  $Mod = \{f_1(t_1), \dots, f_n(t_n)\}$  be a modifier set for program  $\alpha$ .

The anonymous update  $\mathcal{V}(Mod)$  consists of the updates

$$f_i(t_i) := f'_i(t_i)$$

the  $f'_i$  are new function symbols



# Anonymous Updates

## Definition

Let  $Mod = \{f_1(t_1), \dots, f_n(t_n)\}$  be a modifier set for program  $\alpha$ .

The anonymous update  $\mathcal{V}(Mod)$  consists of the updates

$$f_i(t_i) := f'_i(t_i)$$

the  $f'_i$  are new function symbols

for  $1 \leq i \leq n$



# Verifying Correctness of Modifier Sets

Let  $Mod$  be a modifier set for an operation  $op$  with body  $\alpha$ .

## Theorem

If

$$\begin{aligned} & \text{precond}_{op} \wedge \text{generalPre} \wedge \text{defPre} \wedge \\ & \mathcal{V}(Mod)\langle anon(); \rangle \text{true} \\ & \rightarrow \\ & [\alpha] \mathcal{V}'(Mod)\langle anon(); \rangle \text{true} \end{aligned}$$

is valid,

then  $Mod$  is a correct modifier set for  $op$ .



# Verifying Correctness of Modifier Sets

Let  $Mod$  be a modifier set for an operation  $op$  with body  $\alpha$ .

## Theorem

If

$$\begin{aligned} & \text{precond}_{op} \wedge \text{generalPre} \wedge \text{defPre} \wedge \\ & \mathcal{V}(Mod)\langle anon(); \rangle \text{true} \\ & \rightarrow \\ & [\alpha] \mathcal{V}'(Mod)\langle anon(); \rangle \text{true} \end{aligned}$$

is valid,

then  $Mod$  is a correct modifier set for  $op$ .

Theorem by Andreas Roth. For a proof see his dissertation.



# Correctness Theorem

## Simple Instance

Assume  $Mod = \{a\}$ .

### Theorem

If

$$\begin{aligned} & \text{precond}_{op} \wedge \text{generalPre} \wedge \\ & \{a := a'\} \langle \text{anon}(); \rangle \text{true} \\ & \rightarrow \\ & [\alpha] \{a := a'\} \langle \text{anon}(); \rangle \text{true} \end{aligned}$$

is valid,

then  $\{a\}$  is a correct modifier set for  $op$ .



# Correctness Theorem

## Next Simple Instance

Assume  $Mod = \{a, f(a)\}$ .

### Theorem

If

$$\text{precond}_{op} \wedge \text{generalPre} \wedge a\_old = a$$
$$\{f(a) := f'(a), a := a'\} \langle anon(); \rangle \text{true}$$

→

$$[\alpha] \{f(a\_old) := f'(a\_old), a := a'\} \langle anon(); \rangle \text{true}$$

is valid,

then  $\{a, f(a)\}$  is a correct modifier set for  $op$ .



# State of Affairs

Implementation of modifier verification within KeY is almost finished.



# State of Affairs

Implementation of modifier verification within KeY is almost finished.

ESC/Java2 project, led by Joe Kiniry and David Cok, develop a tool for static analysis of Java program code and its JML annotations. Uses but does not verify assignable clauses.



# State of Affairs

Implementation of modifier verification within KeY is almost finished.

ESC/Java2 project, led by Joe Kiniry and David Cok, develop a tool for static analysis of Java program code and its JML annotations. Uses but does not verify assignable clauses.

The ChAsE tool is an extension of ESC/Java, developed by Néstor Cataño at INRIA Sophia-Antipolis tests JML assignable clauses.



# State of Affairs

Implementation of modifier verification within KeY is almost finished.

ESC/Java2 project, led by Joe Kiniry and David Cok, develop a tool for static analysis of Java program code and its JML annotations. Uses but does not verify assignable clauses.

The ChAsE tool is an extension of ESC/Java, developed by Néstor Cataño at INRIA Sophia-Antipolis tests JML assignable clauses.

Boogie offers option to check modifies clauses.



**THE END**

