Software Engineering using Formal Methods

Reasoning about Programs with Dynamic Logic

Wolfgang Ahrendt, Laura Kovács

10 October 2013

Dynamic Logic

(Java) Dynamic Logic

Typed FOL

- + (Java) programs p
- + modalities $\langle \mathrm{p} \rangle \phi$, $[\mathrm{p}]\phi$ (p program, ϕ DL formula)
- + ... (later)

Remark on Hoare Logic and DL

In Hoare logic $\{\mathrm{Pre}\}\ \mathrm{p}\ \{\mathrm{Post}\}$ (Pre, Post must be FOL)

In DL $\mathrm{Pre} \rightarrow [\mathrm{p}]\mathrm{Post}$ (Pre, Post any DL formula)

Proving DL Formulas

An Example

$$
\begin{array}{l}
\forall\ \mathtt{int}\ x;\\
\quad (x \doteq \mathtt{n} \land x >= 0 \rightarrow\\
\qquad [\ \mathtt{i} = 0; \mathtt{r} = 0;\\
\qquad\ \ \mathtt{while}(\mathtt{i} < \mathtt{n})\{\mathtt{i} = \mathtt{i} + 1; \mathtt{r} = \mathtt{r} + \mathtt{i};\}\\
\qquad\ \ \mathtt{r} = \mathtt{r} + \mathtt{r} - \mathtt{n};\\
\qquad]\mathtt{r} \doteq x * x)
\end{array}
$$

How can we prove that the above formula is valid
(i.e. satisfied in all states)?

Semantics of Sequents

$\Gamma = \{\phi_1, \dots, \phi_n\}$ and $\Delta = \{\psi_1, \dots, \psi_m\}$ sets of program formulas where all logical variables occur bound

Recall: $s \models (\Gamma \Longrightarrow \Delta)$ iff $s \models (\phi_1 \wedge \cdots \wedge \phi_n) \rightarrow (\psi_1 \vee \cdots \vee \psi_m)$

Define semantics of DL sequents identical to semantics of FOL sequents

Definition (Validity of Sequents over Program Formulas)

A sequent $\Gamma \Longrightarrow \Delta$ over program formulas is valid iff

$$s \models (\Gamma \Longrightarrow \Delta) \text{ in all states } s$$

Consequence for program variables

Initial value of program variables implicitly "universally quantified"

Symbolic Execution of Programs

Sequent calculus decomposes top-level operator in formula
What is “top-level” in a sequential program `p; q; r;` ?

Symbolic Execution (King, late 60s)

- Follow the natural control flow when analysing a program
- Values of some variables unknown: symbolic state representation

Example

Compute the final state after termination of

```
x=x+y; y=x-y; x=x-y;
```

Symbolic Execution of Programs Cont'd

General form of rule conclusions in symbolic execution calculus

$$\langle \texttt{stmt; rest} \rangle \phi, \qquad [\texttt{stmt; rest}]\phi$$

- Rules symbolically execute *first* statement ('active statement')
- Repeated application of such rules corresponds to symbolic program execution

Example (`updates/swap2.key`, Demo, active statement)

```
\programVariables {
  int x; int y; }

\problem {
    x > y -> \<{x=x+y; y=x-y; x=x-y;}\> y > x
}
```

Symbolic Execution of Programs Cont'd

Symbolic execution of conditional

$$\text{if}\ \frac{\Gamma, \mathtt{b} \doteq \mathbf{true} \Longrightarrow \langle \mathtt{p;\ rest} \rangle \phi, \Delta \qquad \Gamma, \mathtt{b} \doteq \mathbf{false} \Longrightarrow \langle \mathtt{q;\ rest} \rangle \phi, \Delta}{\Gamma \Longrightarrow \langle \mathbf{if}\ \mathtt{(b)\ \{\ p\ \}}\ \mathbf{else}\ \mathtt{\{\ q\ \}\ ;\ rest} \rangle \phi, \Delta}$$

Symbolic execution must consider all possible execution branches

Symbolic execution of loops: unwind

$$\text{unwindLoop}\ \frac{\Gamma \Longrightarrow \langle \mathbf{if}\ \mathtt{(b)\ \{\ p;}\ \mathbf{while}\ \mathtt{(b)\ p\ \};\ rest} \rangle \phi, \Delta}{\Gamma \Longrightarrow \langle \mathbf{while}\ \mathtt{(b)\ \{p\};\ rest} \rangle \phi, \Delta}$$

Updates for KeY-Style Symbolic Execution

Needed: a Notation for Symbolic State Changes

- symbolic execution should ‘walk’ through program
  in natural direction
- need a succint representation of state changes
  effected by a program in one symbolic execution branch
- want to simplify effects of program execution early
- want to apply effects late
  (to branching conditions and post condition)

We use dedicated notation for simple state changes: updates

Explicit State Updates

Definition (Syntax of Updates, Updated Terms/Formulas)

If $\mathtt{v}$ is program variable, t FOL term type-compatible with $\mathtt{v}$, t' any FOL term, and ϕ any DL formula, then

- $\mathtt{v} := t$ is an update
- $\{\mathtt{v} := t\}t'$ is DL term
- $\{\mathtt{v} := t\}\phi$ is DL formula

Definition (Semantics of Updates)

State s interprets flexible symbols f with $\mathcal{I}_s(f)$
β variable assignment for logical variables in t, ρ transition relation:

$\rho(\{\mathtt{v} := t\})(s, \beta) = s'$ where s' identical to s except $\mathcal{I}_{s'}(v) = val_{s,\beta}(t)$

Explicit State Updates Cont'd

Facts about updates $\{\mathtt{v} := t\}$

- Update semantics almost identical to that of assignment
- Value of update also depends on logical variables in t, i.e., β
- Updates are not assignments: right-hand side is FOL term

  $\{\mathtt{x} := n\}\phi$ cannot be turned into assignment (n logical variable)

  $\langle\mathtt{x=i++;}\rangle\phi$ cannot directly be turned into update
- Updates are not equations: change value of flexible terms

Computing Effect of Updates (Automated)

Rewrite rules for update followed by ...

program variable $\begin{cases} \{\mathrm{x} := t\}\mathrm{y} & \leadsto & \mathrm{y} \\ \{\mathrm{x} := t\}\mathrm{x} & \leadsto & t \end{cases}$

logical variable $\{\mathrm{x} := t\}w \leadsto w$

complex term $\{\mathrm{x} := t\}f(t_1, \ldots, t_n) \leadsto f(\{\mathrm{x} := t\}t_1, \ldots, \{\mathrm{x} := t\}t_n)$

(f rigid)

FOL formula $\begin{cases} \{\mathrm{x} := t\}(\phi \ \& \ \psi) \leadsto \{\mathrm{x} := t\}\phi \ \& \ \{\mathrm{x} := t\}\psi \\ \ldots \\ \{\mathrm{x} := t\}(\forall \tau \ y; \phi) \leadsto \forall \tau \ y; (\{\mathrm{x} := t\}\phi) \end{cases}$

program formula No rewrite rule for $\{\mathrm{x} := t\}(\langle \mathrm{p} \rangle \phi)$ unchanged!

Update rewriting delayed until p symbolically executed

Assignment Rule Using Updates

Symbolic execution of assignment using updates

$$\text{assign}\ \frac{\Gamma \Longrightarrow \{\mathtt{x} := t\}\langle\mathtt{rest}\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle\mathtt{x\ =\ t;\ rest}\rangle\phi, \Delta}$$

- Simple! No variable renaming, etc.
- Works as long as t has no side effects (ok in simple DL)
- Special cases needed for $\mathtt{x} = t_1 + t_2$, etc.

Demo

`updates/assignmentToUpdate.key`

Parallel Updates

How to apply updates on updates?

Example

Symbolic execution of

```
    t=x; x=y; y=t;
```

yields:

```
    {t := x}{x := y}{y := t}
```

Need to compose three sequential state changes into a single one:
parallel updates

Parallel Updates Cont'd

Definition (Parallel Update)

A parallel update is expression of the form $\{l_1 := v_1 || \cdots || l_n := v_n\}$ where each $\{l_i := v_i\}$ is simple update

- All v_i computed in old state before update is applied
- Updates of all locations l_i executed simultaneously
- Upon conflict $\quad l_i = l_j,\ v_i \neq v_j \quad$ later update ($\max\{i, j\}$) wins

Definition (Composition Sequential Updates/Conflict Resolution)

$$\{l_1 := r_1\}\{l_2 := r_2\} \;=\; \{l_1 := r_1 || l_2 := \{l_1 := r_1\}r_2\}$$

$$\{l_1 := v_1 || \cdots || l_n := v_n\}\mathtt{x} \;=\; \begin{cases} \mathtt{x} & \text{if } \mathtt{x} \notin \{l_1, \ldots, l_n\} \\ v_k & \text{if } \mathtt{x} = l_k,\ \mathtt{x} \notin \{l_{k+1}, \ldots, l_n\} \end{cases}$$

Symbolic Execution with Updates (by Example)

$$\mathtt{x} < \mathtt{y} \implies \mathtt{x} < \mathtt{y}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{x:=y} \parallel \mathtt{y:=x}\}\langle\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x} \parallel \mathtt{x:=y} \parallel \mathtt{y:=x}\}\langle\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x} \parallel \mathtt{x:=y}\}\{\mathtt{y:=t}\}\langle\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x}\}\{\mathtt{x:=y}\}\langle\mathtt{y=t;}\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x}\}\langle\mathtt{x=y;\ y=t;}\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\implies \mathtt{x} < \mathtt{y} \mathrel{-\!\!>} \langle\mathtt{int\ t=x;\ x=y;\ y=t;}\rangle\, \mathtt{y} < \mathtt{x}$$

Parallel Updates Cont'd

Example

symbolic execution of `x=x+y; y=x-y; x=x-y;` gives

```
({x := x+y}{y := x-y}){x := x-y} =
{x := x+y || y := (x+y)-y}{x := x-y} =
{x := x+y || y := (x+y)-y || x := (x+y)-((x+y)-y)} =
{x := x+y || y := x || x := y} =
{y := x || x := y}
```

KeY automatically deletes overwritten (unnecessary) updates

Demo

`updates/swap2.key`

Parallel updates to store intermediate state of symbolic computation

Another use of Updates

If you would like to quantify over a program variable ...

Not allowed: $\forall \tau\ \mathtt{i}; \langle \mathtt{...i...} \rangle \phi$ (program $\neq$ logical variable)

Instead

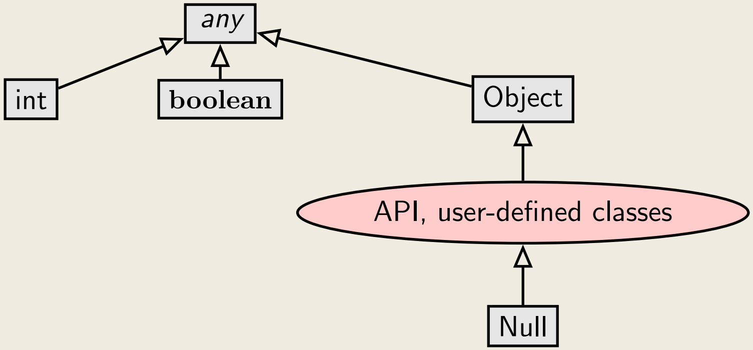
Quantify over value, and assign it to program variable:

$\forall \tau\ i_0; \{\mathtt{i} := i_0\} \langle \mathtt{...i...} \rangle \phi$

- Titlepage
- Symbolic Execution
- Updates
- Parallel Updates
- **Modeling OO Programs**
- Self
- Object Creation
- Round Tour
  - Java Programs
  - Arrays
  - Side Effects
  - Abrupt Termination
  - Aliasing
  - Method Calls
  - Null Pointers
  - Initialization
  - API
- Summary
- Literature

Java Type Hierarchy

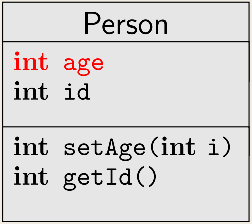
Signature based on Java's type hierarchy



Each class referenced in API and target program is in signature with appropriate partial order

Modelling Fields

Modeling instance fields



Person

int age
int id

int setAge(int i)
int getId()

- Each $o \in \mathrm{D}^{\text{Person}}$ has associated age value
- $\mathcal{I}(\text{age})$ is mapping from Person to int
- Field values can be changed
- For each class C with field a of type τ: FSym_f declares flexible function τ a(C);

Field Access

Signature FSym_f: int age(Person); Person p;

Java/JML expression `p.age >= 0`

Typed FOL $\text{age}(p){>=}0$

KeY postfix notation `p.age >= 0`

Navigation expressions in typed FOL look exactly as in JAVA/JML

Modeling Fields in FOL Cont'd

Resolving Overloading

Overloading resolved by qualifying with class name: `p.age@(Person)`

Changing the value of fields

How to translate assignment to field `p.age=17;` ?

$$\text{assign}\ \frac{\Gamma \Longrightarrow \{\mathtt{l} := t\}\{\mathtt{p.age} := 17\}\langle \mathtt{rest} \rangle \phi, \Delta}{\Gamma \Longrightarrow \langle \mathtt{l\ =\ tp.age\ =\ 17;\ rest} \rangle \phi, \Delta}$$

Admit on left-hand side of update program location expressions

Generalise Definition of Updates

Definition (Syntax of Updates, Updated Terms/Formulas)

If l is program location (e.g., $o.a$), t FOL term type-compatible with l, t' any FOL term, and ϕ any DL formula, then

- $l := t$ is an update
- $\{l := t\}t'$ is DL term
- $\{l := t\}\phi$ is DL formula

Definition (Semantics of Updates, Field Case)

State s interprets field a with $\mathcal{I}_s(a)$
β variable assignment for logical variables in t

$\rho(\{o.a := t\})(s, \beta) = s'$ where s' identical to s except
$\mathcal{I}_{s'}(a)(o) = val_{s,\beta}(t)$

Dynamic Logic - KeY input file

KeY

```
\javaSource "path to source code";

\programVariables { Person p; }

\problem {
      \<{   p.age = 18;  }\> p.age = 18
}
```

KeY

KeY reads in all source files and creates automatically the necessary signature (sorts, field functions)

Demo `updates/firstAttributeExample.key`

Refined Semantics of Program Modalities

Does abrupt termination count as 'normal' termination?
No! Need to distinguish 'normal' and exceptional termination

- $\langle \mathrm{p} \rangle \phi$: $\mathtt{p}$ terminates normally and formula ϕ holds in final state (total correctness)
- $[\mathrm{p}]\phi$: If $\mathtt{p}$ terminates normally then formula ϕ holds in final state (partial correctness)

Abrupt termination on top-level counts as non-termination!

Dynamic Logic - KeY input file

KeY

```
\javaSource "path to source code";

\programVariables {
  ...
}

\problem {
      p != null -> \<{   p.age = 18;  }\> p.age = 18
}
```

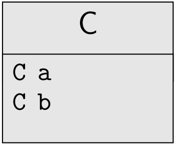
KeY

Only provable when no top-level exception thrown

A Warning on Updates

Computing the effect of updates with field locations is complex

Example



C

C a
C b

- Signature FSym_f: `C a(C);` `C b(C);` `C o;`
- Consider $\{\mathtt{o.a} := \mathtt{o}\}\{\mathtt{o.b} := \mathtt{o.a}\}$
- First update may affect left side of second update
- `o.a` and `o.b` might refer to same object (be aliases)

KeY applies rules automatically, you don't need to worry about details

The Self Reference

Modeling reference this to the receiving object

Special name for the object whose JAVA code is currently executed:

in JML: `Object` **this**;

in Java: `Object` **this**;

in KeY: `Object self;`

Default assumption in JML-KeY translation: $\texttt{self} \mathrel{!=} \textbf{null}$

Which Objects do Exist?

How to model object creation with **new** ?

Constant Domain Assumption

Assume that domain $\mathcal{D}$ is the same in all states of LTS $K = (S, \rho)$

Desirable consequence:
Validity of rigid FOL formulas unaffected by programs containing **new**()

$$\models \forall\, \tau\ x;\ \phi \;\texttt{->}\; [\mathrm{p}](\forall\, \tau\ x;\ \phi) \qquad \text{is valid for rigid } \phi$$

Realizing Constant Domain Assumption

- Flexible function **boolean** `<created>(Object);`
- Equal to **true** iff argument object has been created
- Initialized as $\mathcal{I}(\texttt{<created>})(o) = F$ for all $o \in \mathcal{D}$
- Object creation modeled as $\{\texttt{o.<created>} := \textbf{true}\}$ for next "free" `o`

Titlepage

Symbolic Execution

Updates

Parallel Updates

Modeling OO Programs

Self

Object Creation

Round Tour

- Java Programs
- Arrays
- Side Effects
- Abrupt Termination
- Aliasing
- Method Calls
- Null Pointers
- Initialization
- API

Summary

Literature

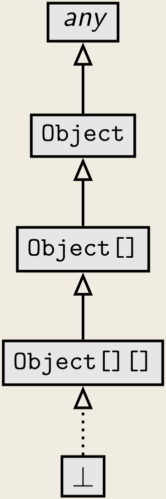
Dynamic Logic to (almost) full Java

KeY supports full sequential Java, with some limitations:

- Limited concurrency
- No generics
- No I/O
- No floats
- No dynamic class loading or reflexion
- API method calls: need either JML contract or implementation

Java Features in Dynamic Logic: Arrays

Arrays



- JAVA type hierarchy includes array types that occur in given program (for finiteness)
- Types ordered according to JAVA subtyping rules
- Model array with flexible function `T []`(`ARR`,**`int`**)
- Instead of `[](a,i)`, we write `a[i]`
- Arrays `a` and `b` can refer to same object (aliases)
- KeY implements update application and simplification rules for array locations

Java Features in Dynamic Logic: Complex Expressions

Complex expressions with side effects

- JAVA expressions may contain assignment operator with side effect
- JAVA expressions can be complex, nested, have method calls
- FOL terms have no side effect on the state

Example (Complex expression with side effects in Java)

`int i = 0; if ((i=2)>= 2) i++;` value of `i` ?

Complex Expressions Cont'd

Decomposition of complex terms by symbolic execution

Follow the rules laid down in JAVA Language Specification

Local code transformations

$$\text{evalOrderIteratedAssgnmt}\quad \frac{\Gamma \Longrightarrow \langle \texttt{y = t; x = y;}\ \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \texttt{x = y = t;}\ \omega\rangle\phi, \Delta}\quad \texttt{t}\ \text{simple}$$

Temporary variables store result of evaluating subexpression

$$\text{ifEval}\quad \frac{\Gamma \Longrightarrow \langle \textbf{boolean}\ \texttt{v0; v0 = b;}\ \textbf{if}\ \texttt{(v0) p;}\ \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \textbf{if}\ \texttt{(b) p;}\ \omega\rangle\phi, \Delta}\quad \texttt{b}\ \text{complex}$$

Guards of conditionals/loops always evaluated (hence: side effect-free) before conditional/unwind rules applied

Java Features in Dynamic Logic: Abrupt Termination

Abrupt Termination: Exceptions and Jumps

Redirection of control flow via **return**, **break**, **continue**, exceptions

$$\langle \pi\ \mathbf{try}\ \{\mathtt{p}\}\ \mathbf{catch}(\mathtt{e})\ \{\mathtt{q}\}\ \mathbf{finally}\ \{\mathtt{r}\}\ \omega\rangle\phi$$

Rules ignore inactive prefix, work on **active statement**, leave postfix

Rule tryThrow matches **try**–**catch** in pre-/postfix and active throw

$$\frac{\Longrightarrow \langle \pi\ \mathrm{if}\ (\mathtt{e}\ \mathrm{instanceof}\ \mathtt{T})\ \{\mathrm{try}\{\mathtt{x}\texttt{=}\mathtt{e};\mathtt{q}\}\ \mathrm{finally}\ \{\mathtt{r}\}\}\mathrm{else}\{\mathtt{r};\mathrm{throw}\ \mathtt{e};\}\ \omega\ \rangle\phi}{\Longrightarrow \langle \pi\ \mathbf{try}\ \{\mathbf{throw}\ \mathtt{e};\ \mathtt{p}\}\ \mathbf{catch}(\mathtt{T}\ \mathtt{x})\ \{\mathtt{q}\}\ \mathbf{finally}\ \{\mathtt{r}\}\ \omega\rangle\phi}$$

Demo: `exceptions/try-catch.key`, `try-catch-dispatch.key`, `try-catch-finally.key`

Java Features in Dynamic Logic: Aliasing

Demo

`aliasing/attributeAlias1.key`

Reference Aliasing

Naive alias resolution causes proof split (on $\mathtt{o} \doteq \mathtt{u}$) at each access

$$\Longrightarrow \mathtt{o.age} \doteq 1 \ \texttt{->} \ \langle \texttt{u.age = 2;} \rangle \mathtt{o.age} \doteq \mathtt{u.age}$$

Java Features in Dynamic Logic: Method Calls

Method Call with actual parameters $arg_0, \ldots, arg_n$

$$\{arg_0 := t_0 \,||\, \cdots \,||\, arg_n := t_n \,||\, c := t_c\}\langle c.\mathtt{m}(arg_0, \ldots, arg_n);\rangle\phi$$

where $\mathtt{m}$ declared as $\textbf{void}\ \mathtt{m}(\tau_0\ \mathtt{p}_0, \ldots, \tau_\mathtt{n}\ \mathtt{p}_\mathtt{n})$

Actions of rule methodCall

- for each formal parameter $\mathtt{p_i}$ of $\mathtt{m}$:
  declare and initialize new local variable $\tau_\mathtt{i}\ \mathtt{p\#i} = arg_i;$
- look up implementation class C of $\mathtt{m}$ and split proof
  if implementation cannot be uniquely determined
- create concrete method invocation $c.\mathtt{m}(\mathtt{p\#0}, \ldots, \mathtt{p\#n})\mathtt{@}C$

Method Calls Cont'd

Method Body Expand

1. Execute code that binds actual to formal parameters $\tau_{\mathtt{i}}$ `p#i =`arg_i;
2. Call rule methodBodyExpand

$$\frac{\Gamma \Longrightarrow \langle \pi \ \mathtt{method\text{-}frame(source=C,\ this=c)\{\ body\ \}}\ \omega \rangle \phi, \Delta}{\Gamma \Longrightarrow \langle \pi \ \mathtt{c.m(p\#0,\ldots,p\#n)@C;}\ \omega \rangle \phi, \Delta}$$

Demo

`methods/ instanceMethodInlineSimple.key`

A Round Tour of Java Features in DL Cont'd

Localisation of Fields and Method Implementation

JAVA has complex rules for localisation of fields and method implementations

- Polymorphism
- Late binding
- Scoping (class vs. instance)
- Context (static vs. runtime)
- Visibility (private, protected, public)

Proof split into cases when implementation not statically determined

A Round Tour of Java Features in DL Cont'd

Null pointer exceptions

There are no "exceptions" in FOL: $\mathcal{I}$ total on FSym

Need to model possibility that o $\doteq \mathbf{null}$ in o.a

- KeY branches over o $!= \mathbf{null}$ upon each field access

A Round Tour of Java Features in DL Cont'd

Object initialization

JAVA has complex rules for object initialization

- Chain of constructor calls until Object
- Implicit calls to **super**()
- Visbility issues
- Initialization sequence

Coding of initialization rules in methods `<createObject>()`, `<init>()`,... which are then symbolically executed

A Round Tour of Java Features in DL Cont'd

Formal specification of Java API

How to perform symbolic execution when JAVA API method is called?

1. API method has reference implementation in JAVA
   Call method and execute symbolically

   Problem Reference implementation not always available

   Problem Breaks modularity
2. Use JML contract of API method:
   - 2.1 Show that requires clause is satisfied
   - 2.2 Obtain postcondition from ensures clause
   - 2.3 Delete updates with modifiable locations from symbolic state

Java Card API in JML or DL

DL version available in KeY, JML work in progress See W. Mostowski

`http://limerick.cost-ic0701.org/home/`
`verifying-java-card-programs-with-key`

Summary

- Most JAVA features covered in KeY
- Several of remaining features available in experimental version
  - Simplified multi-threaded JMM
  - Floats
- Degree of automation for loop-free programs is very high
- Proving loops requires user to provide invariant
  - Automatic invariant generation sometimes possible
- Symbolic execution paradigm lets you use KeY w/o understanding details of logic

Literature for this Lecture

Essential

KeY Book Verification of Object-Oriented Software (see course web page), Chapter 10: Using KeY

KeY Book Verification of Object-Oriented Software (see course web page), Chapter 3: Dynamic Logic, Sections 3.1, 3.2, 3.4, 3.5, 3.6.1, 3.6.2, 3.6.3, 3.6.4, 3.6.5, 3.6.7