

Software Engineering using Formal Methods

Reasoning about Programs with Dynamic Logic

Wolfgang Ahrendt

08 October 2015

Dynamic Logic

(JAVA) Dynamic Logic

Typed FOL

- ▶ + (JAVA) programs p
- ▶ + modalities $\langle p \rangle \phi$, $[p] \phi$ (p program, ϕ DL formula)
- ▶ + ... (later)

Remark on Hoare Logic and DL

In Hoare logic $\{Pre\} p \{Post\}$

(Pre, Post must be FOL)

In DL $Pre \rightarrow [p]Post$

(Pre, Post any DL formula)

Proving DL Formulas

An Example

```
 $\forall$  int x;  
(x = n  $\wedge$  x  $\geq$  0  $\rightarrow$   
  [ i = 0; r = 0;  
    while(i < n){i = i + 1; r = r + i;}  
    r = r + r - n;  
  ]r = x * x)
```

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

Semantics of DL Sequents

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

Recall: $\mathcal{S} \models (\Gamma \Rightarrow \Delta)$ iff $\mathcal{S} \models (\phi_1 \wedge \dots \wedge \phi_n) \rightarrow (\psi_1 \vee \dots \vee \psi_m)$

Define semantics of DL sequents identical to semantics of FOL sequents

Definition (Validity of Sequents over DL Formulas)

A sequent $\Gamma \Rightarrow \Delta$ over DL formulas is **valid** iff

$$\mathcal{S} \models (\Gamma \Rightarrow \Delta) \text{ in all states } \mathcal{S}$$

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

- ▶ 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

$$\langle \text{stmt}; \text{rest} \rangle \phi, \quad [\text{stmt}; \text{rest}] \phi$$

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

Example (`symbolicExecution/simpleIf.key`,
Demo, active statement only)

```
\programVariables {  
  int x; int y; boolean b;  
}  
\problem {  
  \<{ if (b) { x = 1; } else { x = 2; } y = 3; }\> y > x  
}
```

Symbolic Execution of Programs Cont'd

Symbolic execution of conditional

$$\text{if} \frac{\Gamma, b = \text{true} \Rightarrow \langle p; \text{rest} \rangle \phi, \Delta \quad \Gamma, b = \text{false} \Rightarrow \langle q; \text{rest} \rangle \phi, \Delta}{\Gamma \Rightarrow \langle \text{if } (b) \{ p \} \text{ else } \{ q \} ; \text{rest} \rangle \phi, \Delta}$$

Symbolic execution must consider all possible execution branches

Symbolic execution of loops: unwind

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

Updates for KeY-Style Symbolic Execution

Needed: a Notation for Symbolic State Changes

- ▶ Symbolic execution should “walk” through program in natural **forward** direction
- ▶ Need **succinct representation** of state changes effected by each symbolic execution step
- ▶ Want to **simplify** effects of program execution **early**
- ▶ Want to **apply** state changes **late**
(to branching conditions and post condition)

We use dedicated notation for state changes: **updates**

Explicit State Updates

Definition (Syntax of Updates, Updated Terms/Formulas)

If v is program variable, t FOL term type-conformant to v , t' any FOL term, and ϕ any DL formula, then

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

Definition (Semantics of Updates)

State \mathcal{S} interprets program variables v with $\mathcal{I}_{\mathcal{S}}(v)$

β variable assignment for logical variables in t , define semantics ρ as:

$$\rho_{\beta}(\{v := t\})(\mathcal{S}) = \mathcal{S}' \text{ where } \mathcal{S}' \text{ identical to } \mathcal{S} \text{ except } \mathcal{I}_{\mathcal{S}'}(v) = \text{val}_{\mathcal{S},\beta}(t)$$

Explicit State Updates Cont'd

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

- ▶ Update semantics similar to that of assignment
- ▶ Value of update also depends on \mathcal{S} and **logical** variables in t , i.e., β
- ▶ Updates are **not assignments**: right-hand side is FOL term
 - $\{x := n\}\phi$ cannot be turned into assignment (n logical variable)
 - $\{x=i++;\}\phi$ cannot (immediately) be turned into update
- ▶ Updates are **not equations**: they **change** value of v

Computing Effect of Updates (Automated)

Rewrite rules for update followed by ...

program variable $\left\{ \begin{array}{l} \{x := t\}x \rightsquigarrow t \\ \{x := t\}y \rightsquigarrow y \end{array} \right.$

logical variable $\{x := t\}w \rightsquigarrow w$

complex term $\{x := t\}f(t_1, \dots, t_n) \rightsquigarrow f(\{x := t\}t_1, \dots, \{x := t\}t_n)$
(because f is rigid)

FOL formula $\left\{ \begin{array}{l} \{x := t\}(\phi \ \& \ \psi) \rightsquigarrow \{x := t\}\phi \ \& \ \{x := t\}\psi \\ \dots \\ \{x := t\}(\forall \tau y; \phi) \rightsquigarrow \forall \tau y; (\{x := t\}\phi) \end{array} \right.$

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

Update rewriting delayed until p symbolically executed

Assignment Rule Using Updates

Symbolic execution of assignment using updates

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

- ▶ Simple! No variable renaming, etc.
- ▶ Works as long as t has no side effects

Demo

updates/assignmentToUpdate.key

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 an expression of the form $\{v_1 := r_1 \parallel \dots \parallel v_n := r_n\}$ where each $\{v_i := r_i\}$ is simple update

- ▶ All r_i computed in **old state** before update is applied
- ▶ Updates of all program variables v_i executed **simultaneously**
- ▶ Upon **conflict** $v_i = v_j, r_i \neq r_j$ later update ($\max\{i, j\}$) wins

Definition (Composition Sequential Updates/Conflict Resolution)

$$\{v_1 := r_1\}\{v_2 := r_2\} = \{v_1 := r_1 \parallel v_2 := \{v_1 := r_1\}r_2\}$$

$$\{v_1 := r_1 \parallel \dots \parallel v_n := r_n\}x = \begin{cases} x & \text{if } x \notin \{v_1, \dots, v_n\} \\ r_k & \text{if } x = v_k, x \notin \{v_{k+1}, \dots, v_n\} \end{cases}$$

Symbolic Execution with Updates (by Example)

$$\begin{aligned} & x < y \implies x < y \\ & \quad \vdots \\ & x < y \implies \{x:=y \parallel y:=x\} \langle \rangle y < x \\ & \quad \vdots \\ & x < y \implies \{t:=x \parallel x:=y \parallel y:=x\} \langle \rangle y < x \\ & \quad \vdots \\ & x < y \implies \{t:=x \parallel x:=y\} \{y:=t\} \langle \rangle y < x \\ & \quad \vdots \\ & x < y \implies \{t:=x\} \{x:=y\} \langle y=t; \rangle y < x \\ & \quad \vdots \\ & x < y \implies \{t:=x\} \langle x=y; y=t; \rangle y < x \\ & \quad \vdots \\ \implies & x < y \rightarrow \langle \text{int } t=x; x=y; y=t; \rangle y < x \end{aligned}$$

Parallel Updates Cont'd

Example

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

$$\begin{aligned} & (\{x := x+y\}\{y := x-y\})\{x := x-y\} \\ & \{x := x+y \parallel y := (x+y)-y\}\{x := x-y\} \\ & \{x := x+y \parallel y := (x+y)-y \parallel x := (x+y)-((x+y)-y)\} \\ & \{x := x+y \parallel y := x \parallel x := y\} \\ & \{y := x \parallel x := y\} \end{aligned}$$

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: $\forall \tau i; \langle \dots i \dots \rangle \phi$
(program variables \cap logical variables = \emptyset)

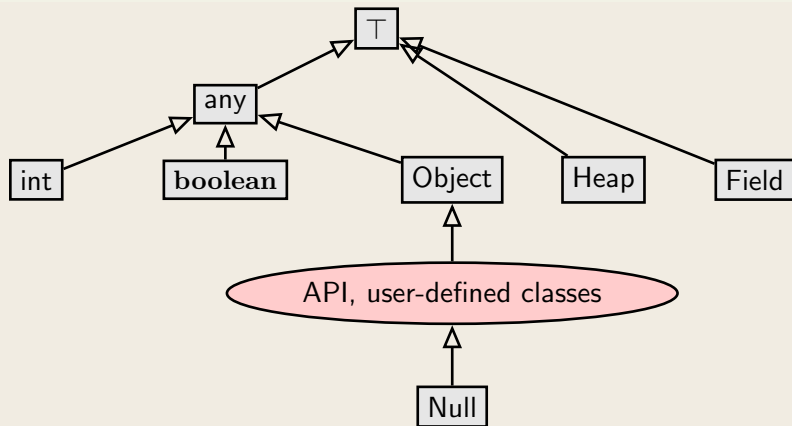
Instead

Quantify over **value**, and **assign** it to program variable:

$\forall \tau x; \{i := x\} \langle \dots i \dots \rangle \phi$

Modelling Java in FOL: Fixing a Type Hierarchy

Signature based on Java's type hierarchy



Each interface and class in API and in target program becomes type with appropriate subtype relation

Modelling the Heap in FOL

The Java Heap

Objects are stored on (i.e., in) the **heap**.

- ▶ Status of heap changes during execution
- ▶ Each heap associates values to object/field pairs

The Heap Model of KeY-DL

Each element of data type Heap represents a certain heap status.

Two functions involving heaps:

- ▶ in F_{Σ} : Heap $\text{store}(\text{Heap}, \text{Object}, \text{Field}, \text{any})$;
 $\text{store}(h, o, f, v)$ returns heap like h , but with v associated to (o, f)
- ▶ in F_{Σ} : any $\text{select}(\text{Heap}, \text{Object}, \text{Field})$;
 $\text{select}(h, o, f)$ returns value associated to (o, f) in h

Modelling the Heap in FOL

Modelling instance fields

Person
<code>int age</code> <code>int id</code>
<code>int setAge(int newAge)</code> <code>int getId()</code>

- ▶ for each JAVA reference type C there is a type $C \in T_{\Sigma}$, for example, `Person`
- ▶ for each field f there is a **unique** constant f of type `Field`, for example, `id`
- ▶ domain of all `Person` objects: D^{Person}
- ▶ a heap relates objects and fields to values

Reading Field `id` of `Person p`

FOL notation `select(h, p, id)`

Key notation `p.id@h` (abbreviating `select(h, p, id)`)
`p.id` (abbreviating `select(heap, p, id)`)^a

^aheap is special program variable for “current” heap; mostly implicit in *o.f*

Modelling the Heap in FOL

Modelling instance fields

Person
int age int id
int setAge(int newAge) int getId()

- ▶ for each JAVA reference type C there is a type $C \in T_{\Sigma}$, for example, Person
- ▶ for each field f there is a **unique** constant f of type Field, for example, id
- ▶ domain of all Person objects: D^{Person}
- ▶ a heap relates objects and fields to values

Writing to Field id of Person p

FOL notation $\text{store}(h, p, \text{id}, 6238)$

KeY notation $h[p.\text{id} := 6238]$ (notation for store, not update)

The Algebra of Heaps

We do *not* formalise the *structure* (implementation) of heaps.
We formalise the *behaviour*, with an algebra of heap operations:

$$\text{select}(\text{store}(h, o, f, v), o, f) = v$$

$$(o \neq o' \vee f \neq f') \rightarrow \text{select}(\text{store}(h, o, f, x), o', f') = \text{select}(h, o', f')$$

Example

$$\text{select}(\text{store}(h, o, f, 15), o, f) \rightsquigarrow 15$$

$$\text{select}(\text{store}(h, o, f, 15), o, g) \rightsquigarrow \text{select}(h, o, g)$$

$$\text{select}(\text{store}(h, o, f, 15), u, f) \rightsquigarrow$$

$$\text{if } (o = u) \text{ then } (15) \text{ else } (\text{select}(h, u, f))$$

Pretty Printing

Shorthand Notations for Heap Operations

$o.f@h$ is `select(h, o, f)`
 $h[o.f := v]$ is `store(h, o, f, v)`

therefore:

$u.f@h[o.f := v]$ is `select(store(h, o, f, v), u, f)`
 $h[o.f := v][o'.f' := v']$ is `store(store(h, o, f, v), o', f', v')`

Very-Shorthand Notations for **Current** Heap

Current heap always in special variable `heap`.

$o.f$ is `select(heap, o, f)`
 $\{o.f := v\}$ is `update {heap := heap[o.f := x]}`

Modelling the Heap in FOL—The Full Story

Is formula `select(h, p, id) >= 0` **type-safe?**

1. Return type is any—need to 'cast' to `int`
2. There can be many fields with name `id`

Real Field Access

`int::select(h, p, Person::$id) >= 0` is type-safe

- ▶ `int::select` is a function name, not a cast
- ▶ can be understood *intuitively* as `(int)select`

General

For each `T` typed field `f` of class `C`, F_Σ contains

- ▶ a constant declared as `Field C::$f`
- ▶ a function declared as `T T::select(Heap, C, Field)`

Everything **blue** is a function name

Writing to Fields

We stick to the above:

Declaration: `Heap store(Heap, Object, Field, any);`

Usage: `store(h, p, Person::$id, 42)`

Field Update Assignment Rule

Changing the value of fields

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

$$\text{assign} \frac{\Gamma \Rightarrow \{o.f := t\} \langle \text{rest} \rangle \phi, \Delta}{\Gamma \Rightarrow \langle o.f = t; \text{rest} \rangle \phi, \Delta}$$

Admit on left-hand side of update **JAVA location expressions**

Field Update Assignment Rule

Changing the value of fields

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

$$\text{assign} \frac{\Gamma \Rightarrow \{\text{heap} := \text{store}(\text{heap}, p, \text{age}, 17)\} \langle \text{rest} \rangle \phi, \Delta}{\Gamma \Rightarrow \langle p.\text{age} = 17; \text{rest} \rangle \phi, \Delta}$$

Admit on left-hand side of update **JAVA location expressions**

Field Update Assignment Rule

Changing the value of fields

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

$$\text{assign} \frac{\Gamma \Rightarrow \{\mathbf{p.age} := 17\} \langle \text{rest} \rangle \phi, \Delta}{\Gamma \Rightarrow \langle \mathbf{p.age} = 17; \text{rest} \rangle \phi, \Delta}$$

Admit on left-hand side of update **JAVA location expressions**

Dynamic Logic: KeY input file

```
\javaSource "path to source code referenced in problem";  
  
\programVariables { Person p; }  
  
\problem {  
    \<{    p.age = 18;    }\> p.age = 18  
}
```

KeY reads in all source files and creates automatically the necessary signature (types, program variables, field constants)

Demo

updates/firstAttributeExample.key

Refined Semantics of Program Modalities

Does abrupt termination count as normal termination?

No! Need to distinguish **normal** and **exceptional** termination

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

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

Example Reconsidered: Exception Handling

```
\javaSource "path to source code";  
  
\programVariables {  
  ...  
}  
  
\problem {  
  p != null -> \<{ p.age = 18; }\> p.age = 18  
}
```

Only provable when no top-level exception thrown

Demo

updates/secondAttributeExample.key

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: **self != 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**()

$\models \forall T x; \phi \rightarrow [p](\forall T x; \phi)$ is valid for rigid ϕ

Object Creation

Realizing Constant Domain Assumption

- ▶ Implicitly declared field `boolean <created>` in class `Object`
- ▶ Equal to `true` iff argument object has been created
- ▶ Object creation modeled as $\{\text{heap} := \text{create}(\text{heap}, o)\}$ for not (yet) created `o` (essentially sets `<created>` field of `o` to `true`)
- ▶ Normal heap function *store* “cannot” set value of field `<created>`

ObjectCreation(simplified)

$$\frac{\Gamma, \{u\}(\text{select}(\text{heap}, \text{ob}, \text{Object}::\langle\text{created}\rangle) = \text{FALSE}) \Rightarrow \{u\}(\{\text{heap} := \text{create}(\text{heap}, \text{ob})\}\{o := \text{ob}\}\langle\pi o.\langle\text{init}\rangle(\text{param}); \omega\rangle\phi), \Delta}{\Gamma \Rightarrow \{u\}(\langle\pi o = \text{new } T(\text{param}); \omega\rangle\phi), \Delta}$$

`ob` is a fresh program variable

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

API

Summary

Literature

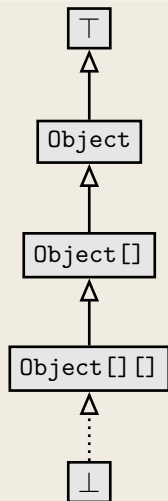
Dynamic Logic to (almost) full Java

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
- ▶ Value of entry in array $T[]$ a ; defined in class C depends on reference a to array in C and index
- ▶ Function $\text{arr} : \text{int} \rightarrow \text{Field}$ injective mapping from indices to fields
- ▶ Store array elements on heap, e.g., the value of $a[i]$ on the heap $\text{store}(\text{heap}, a, \text{arr}(i), 17)$ is 17
- ▶ Arrays a and b can refer to same object (aliases)
- ▶ KeY implements simplification and evaluation 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

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

Temporary variables store result of evaluating subexpression

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

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**

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

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

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

$$\Rightarrow \langle \pi \text{ if}(e \text{ instanceof } T) \{ \text{try}\{x=e; q\} \text{ finally } \{r\}\} \text{ else}\{r; \text{throw } e;\} \omega \rangle \phi$$

$$\Rightarrow \langle \pi \text{ try } \{ \text{throw } e; p \} \text{ catch}(T x) \{q\} \text{ finally } \{r\} \omega \rangle \phi$$

Demo

exceptions/try-catch.key

Java Features in Dynamic Logic: Aliasing

Demo

aliasing/attributeAlias1.key

Reference Aliasing

Naive alias resolution causes **proof split** (on $o = u$) at each access

$$\Rightarrow o.age = 1 \rightarrow \langle u.age = 2; \rangle o.age = u.age$$

Java Features in Dynamic Logic: Method Calls

Method Call

First evaluate arguments, leading to:

$$\{arg_0 := t_0 \parallel \dots \parallel arg_n := t_n \parallel c := t_c\} \langle c.m(arg_0, \dots, arg_n); \rangle \phi$$

Actions of rule **methodCall**

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

Method Body Expand

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

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

Demo

methods/instanceMethodInlineSimple.key

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

Null pointer exceptions

There are no “exceptions” in FOL: \mathcal{I} total on FSym

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

- ▶ KeY branches over $o \neq \mathbf{null}$ upon each field access

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