

Software Engineering using Formal Methods

Reasoning about Programs with Dynamic Logic

Wolfgang Ahrendt

3 October 2013

Part I

Where are we?

Where Are We?

before specification of JAVA programs with JML

now dynamic logic (DL) for reasoning about JAVA programs

after that generating DL from JML+JAVA

+ verifying the resulting proof obligations

Motivation

Consider the method

```
public void doubleContent(int [] a) {  
    int i = 0;  
    while (i < a.length) {  
        a[i] = a[i] * 2;  
        i++;  
    }  
}
```

We want a **logic/calculus** allowing to **express/prove** properties like, e.g.:

If $a \neq \text{null}$

then `doubleContent` terminates normally

and afterwards all elements of `a` are twice the old value

Motivation (contd.)

One such logic is **dynamic logic** (DL).

The above statemet in DL would be:

$$\begin{aligned} & a \neq \text{null} \\ & \wedge a \neq b \\ & \wedge \forall \text{int } i; ((0 \leq i \wedge i < a.\text{length}) \rightarrow a[i] = b[i]) \\ \rightarrow & \langle \text{doubleContent}(a); \rangle \\ & \forall \text{int } i; ((0 \leq i \wedge i < a.\text{length}) \rightarrow a[i] = 2 * b[i]) \end{aligned}$$

- ▶ DL combines first-order logic (FOL) with programs
- ▶ Theory of DL extends theory of FOL
- ▶ Necessary to look closer at FOL at first
- ▶ Then extend towards DL

introducing **dynamic logic** for JAVA

- ▶ recap first-order logic (FOL)
- ▶ semantics of FOL
- ▶ dynamic logic = extending FOL with
 - ▶ **dynamic interpretations**
 - ▶ **programs** to describe state change

Part II

First-Order Semantics

First-Order Semantics

From propositional to first-order semantics

- ▶ In prop. logic, an interpretation of variables with $\{T, F\}$ sufficed
- ▶ In first-order logic we must assign meaning to:
 - ▶ function symbols (incl. constants)
 - ▶ predicate symbols
- ▶ Respect typing: `int i`, `List l` **must** denote different elements

What we need (to interpret a first-order formula)

1. A collection of **typed universes** of elements
2. A mapping from **variables** to elements
3. For each **function symbol**, a mapping from arguments to results
4. For each **predicate symbol**, a set of argument tuples where that predicate holds

First-Order Domains/Universes

1. A collection of **typed universes** of elements

Definition (Universe/Domain)

A non-empty set \mathcal{D} of elements is a **universe** or **domain**.

Each element of \mathcal{D} has a fixed type given by $\delta : \mathcal{D} \rightarrow T_\Sigma$

- ▶ Notation for the domain elements of type $\tau \in T_\Sigma$:

$$\mathcal{D}^\tau = \{d \in \mathcal{D} \mid \delta(d) = \tau\}$$

- ▶ Each type $\tau \in T_\Sigma$ must 'contain' at least one domain element:

$$\mathcal{D}^\tau \neq \emptyset$$

First-Order States

3. For each **function symbol**, a mapping from arguments to results
4. For each **predicate symbol**, a set of argument tuples where that predicate holds

Definition (First-Order State)

Let \mathcal{D} be a domain with typing function δ .

For each f be declared as $\tau f(\tau_1, \dots, \tau_r)$;

and each p be declared as $p(\tau_1, \dots, \tau_r)$;

$\mathcal{I}(f)$ is a mapping $\mathcal{I}(f) : \mathcal{D}^{\tau_1} \times \dots \times \mathcal{D}^{\tau_r} \rightarrow \mathcal{D}^{\tau}$

$\mathcal{I}(p)$ is a set $\mathcal{I}(p) \subseteq \mathcal{D}^{\tau_1} \times \dots \times \mathcal{D}^{\tau_r}$

Then $\mathcal{S} = (\mathcal{D}, \delta, \mathcal{I})$ is a **first-order state**

First-Order States Cont'd

Example

Signature: `int i; int j; int f(int); Object obj; <(int,int);`

$\mathcal{D} = \{17, 2, o\}$

The following \mathcal{I} is a possible interpretation:

$$\mathcal{I}(i) = 17$$

$$\mathcal{I}(j) = 17$$

$$\mathcal{I}(obj) = o$$

| \mathcal{D}^{int} | $\mathcal{I}(f)$ |
|----------------------------|------------------|
| 2 | 2 |
| 17 | 2 |

| $\mathcal{D}^{\text{int}} \times \mathcal{D}^{\text{int}}$ | in $\mathcal{I}(<)$? |
|--|-----------------------|
| (2, 2) | <i>no</i> |
| (2, 17) | <i>yes</i> |
| (17, 2) | <i>no</i> |
| (17, 17) | <i>no</i> |

One of uncountably many possible first-order states!

Semantics of Reserved Signature Symbols

Definition

Reserved predicate symbol for **equality**: =

Interpretation is fixed as $\mathcal{I}(=) = \{(d, d) \mid d \in \mathcal{D}\}$

Exercise: write down all elements of the set $\mathcal{I}(=)$ for example domain

Signature Symbols vs. Domain Elements

- ▶ Domain elements different from the terms representing them
- ▶ First-order formulas and terms have **no access** to domain

Example

Signature: Object obj1, obj2;

Domain: $\mathcal{D} = \{o\}$

In this state, necessarily $\mathcal{I}(\text{obj1}) = \mathcal{I}(\text{obj2}) = o$

Variable Assignments

2. A mapping from variables to domain elements

Definition (Variable Assignment)

A **variable assignment** β maps variables to domain elements

It respects the variable type, i.e., if x has type τ then $\beta(x) \in \mathcal{D}^\tau$

Semantic Evaluation of Terms

Given a first-order state \mathcal{S} and a variable assignment β it is possible to evaluate first-order terms under \mathcal{S} and β

Definition (Valuation of Terms)

$val_{\mathcal{S},\beta} : \text{Term} \rightarrow \mathcal{D}$ such that $val_{\mathcal{S},\beta}(t) \in \mathcal{D}^\tau$ for $t \in \text{Term}_\tau$:

- ▶ $val_{\mathcal{S},\beta}(x) = \beta(x)$
- ▶ $val_{\mathcal{S},\beta}(f(t_1, \dots, t_r)) = \mathcal{I}(f)(val_{\mathcal{S},\beta}(t_1), \dots, val_{\mathcal{S},\beta}(t_r))$

Semantic Evaluation of Terms Cont'd

Example

Signature: `int i; int j; int f(int);`

$\mathcal{D} = \{17, 2, o\}$ Variables: `Object obj; int x;`

$$\mathcal{I}(i) = 17$$

$$\mathcal{I}(j) = 17$$

| \mathcal{D}^{int} | $\mathcal{I}(f)$ |
|----------------------------|------------------|
| 2 | 17 |
| 17 | 2 |

| Var | β |
|-----|---------|
| obj | o |
| x | 17 |

- ▶ $val_{S,\beta}(f(f(i)))$?
- ▶ $val_{S,\beta}(f(f(x)))$?
- ▶ $val_{S,\beta}(\text{obj})$?

Definition (Modified Variable Assignment)

Let y be variable of type τ , β variable assignment, $d \in \mathcal{D}^\tau$:

$$\beta_y^d(x) := \begin{cases} \beta(x) & x \neq y \\ d & x = y \end{cases}$$

Definition (Valuation of Formulas)

$val_{S,\beta}(\phi)$ for $\phi \in For$

- ▶ $val_{S,\beta}(p(t_1, \dots, t_r)) = T$ iff $(val_{S,\beta}(t_1), \dots, val_{S,\beta}(t_r)) \in \mathcal{I}(p)$
- ▶ $val_{S,\beta}(\phi \wedge \psi) = T$ iff $val_{S,\beta}(\phi) = T$ and $val_{S,\beta}(\psi) = T$
- ▶ ... as in propositional logic
- ▶ $val_{S,\beta}(\forall \tau x; \phi) = T$ iff $val_{S,\beta_x^d}(\phi) = T$ for all $d \in \mathcal{D}^\tau$
- ▶ $val_{S,\beta}(\exists \tau x; \phi) = T$ iff $val_{S,\beta_x^d}(\phi) = T$ for at least one $d \in \mathcal{D}^\tau$

Semantic Evaluation of Formulas Cont'd

Example

Signature: `int j; int f(int); Object obj; <(int,int);`

$\mathcal{D} = \{17, 2, o\}$

$I(j) = 17$
 $I(obj) = o$

| \mathcal{D}^{int} | $I(f)$ |
|----------------------------|--------|
| 2 | 2 |
| 17 | 2 |

| $\mathcal{D}^{\text{int}} \times \mathcal{D}^{\text{int}}$ | in $I(<)$? |
|--|-------------|
| (2, 2) | F |
| (2, 17) | T |
| (17, 2) | F |
| (17, 17) | F |

- ▶ $val_{S,\beta}(f(j) < j)$?
- ▶ $val_{S,\beta}(\exists \text{int } x; f(x) = x)$?
- ▶ $val_{S,\beta}(\forall \text{Object } o1; \forall \text{Object } o2; o1 = o2)$?

Definition (Satisfiability, Truth, Validity)

$$\begin{array}{ll} \text{val}_{\mathcal{S},\beta}(\phi) = T & (\mathcal{S}, \beta \text{ satisfies } \phi) \\ \mathcal{S} \models \phi & \text{iff for all } \beta : \text{val}_{\mathcal{S},\beta}(\phi) = T \quad (\phi \text{ is true in } \mathcal{S}) \\ \models \phi & \text{iff for all } \mathcal{S} : \mathcal{S} \models \phi \quad (\phi \text{ is valid}) \end{array}$$

Example

- ▶ $f(j) < j$ is true in \mathcal{S}
- ▶ $\exists \text{int } x; i = x$ is valid
- ▶ $\exists \text{int } x; \neg(x = x)$ is not satisfiable

Part III

Towards Dynamic Logic

Type Hierarchy

First, we *refine the type system* of FOL:

Definition (Type Hierarchy)

- ▶ T_Σ is set of **types**
- ▶ Given **subtype** relation ' \sqsubseteq ', with top element '*any*'
- ▶ $\tau \sqsubseteq \text{any}$ for all $\tau \in T_\Sigma$

Example (A Minimal Type Hierarchy)

$$\mathcal{T} = \{\text{any}\}$$

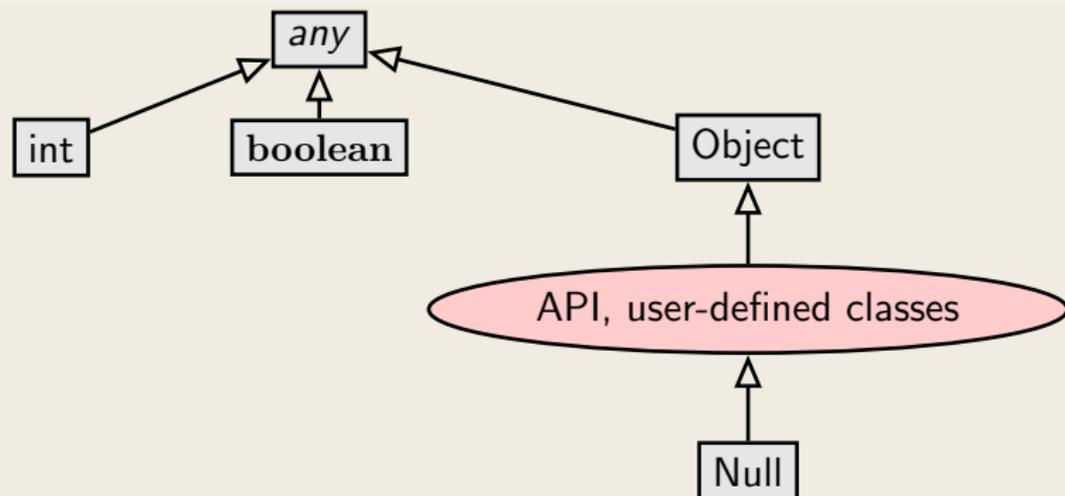
All signature symbols have same type *any*.

Example (Type Hierarchy for Java)

(see next slide)

Modelling Java in FOL: Fixing a Type Hierarchy

Signature based on Java's type hierarchy (simplified)



Each class in API and target program is a type, with appropriate subtyping.

Modelling Classes and Fields in FOL

Modeling instance fields

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

- ▶ domain of all Person objects: D^{Person}
- ▶ each $o \in D^{\text{Person}}$ has associated age value
- ▶ $\mathcal{I}(\text{age})$ is **mapping** from **Person** to **int**
- ▶ for each class C with field τ a:
FSym declares function τ a(C);

Field Access

Signature FSym: `int age(Person);` `Person p;`

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

Typed FOL `age(p) >= 0`

KeY postfix notation for FOL `p.age >= 0`

Navigation expressions in KeY look exactly as in **JAVA/JML**

Dynamic View

Only static properties expressible in typed FOL, e.g.,

- ▶ Values of fields in a certain range
- ▶ Property (invariant) of a subclass implies property of a superclass
- ▶ ...

Considers only one state at a time.

Goal: Express functional properties of a program, e.g.

If method `setAge` is called on an object `o` of type `Person`
and the method argument `newAge` is positive
then afterwards field `age` has same value as `newAge`.

Observation

Need a logic that allows us to

- ▶ relate different program states, i.e., **before** and **after** execution, within one formula
- ▶ program variables/fields represented by constant/function symbols that depend on program state

Dynamic Logic meets the above requirements.

Dynamic Logic

(JAVA) Dynamic Logic

Typed FOL

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

An Example

$$i > 5 \rightarrow [i = i + 10;] i > 15$$

Meaning?

If **program variable** i is greater than 5, then **after** executing $i = i + 10;$, i is greater than 15.

Type Hierarchy

Dynamic Logic = Typed FOL + ...

Type hierarchy

$T_{\Sigma} = \{\text{int}, \text{boolean}, \text{any}\}$ with **int**, **boolean** incomparable, both are subtypes of *any*

int and **boolean** are the only types for today.
Classes, interfaces etc. in next lecture.

Program Variables

Dynamic Logic = Typed FOL + ...

$$i > 5 \rightarrow [i = i + 10;]i > 15$$

Program variable i refers to different values **before** and **after** execution of a program.

- ▶ Program variables like i are state-dependent constant symbols.
- ▶ Value of state dependent symbols changeable by program.

Three words **one** meaning: flexible, state-dependent, non-rigid

Rigid versus Flexible Symbols

Signature of dynamic logic defined as in FOL, **but**:
In addition there are flexible symbols

Rigid versus Flexible

- ▶ **Rigid** symbols, same interpretation in **all** program states
 - ▶ First-order variables (aka **logical variables**)
 - ▶ Built-in functions and predicates such as $0, 1, \dots, +, *, \dots, <, \dots$
- ▶ **Flexible** (or **non-rigid**) symbols, interpretation depends on state

Capture side effects on state during program execution

- ▶ Functions modeling **program variables** and **fields** are flexible

Any term containing at least one flexible symbol is also flexible

Signature of Dynamic Logic

Definition (Dynamic Logic Signature)

$$\Sigma = (\text{PSym}_r, \text{FSym}_r, \text{FSym}_f, \alpha), \quad \text{FSym}_r \cap \text{FSym}_f = \emptyset$$

Rigid Predicate Symbols $\text{PSym}_r = \{>, >=, \dots\}$

Rigid Function Symbols $\text{FSym}_r = \{+, -, *, 0, 1, \dots\}$

Flexible Function Symbols $\text{FSym}_f = \{i, j, k, \dots\}$

Standard typing: `boolean TRUE`; `<(int,int)`; etc.

Flexible constant/function symbols FSym_f used to model

- ▶ program variables (constants) and
- ▶ fields (unary flexible functions)

Dynamic Logic Signature - KeY input file

```
\sorts {  
    // only additional sorts (predefined:  int/boolean/any)  
}  
\functions {  
    // only additional rigid functions  
    // (arithmetic functions like +,- etc.  predefined)  
}  
\predicates { /* same as for functions */ }  
  
\programVariables { // flexible functions  
    int i, j;  
    boolean b;  
}
```

Empty sections can be left out.

Variables

Logical Variables

Typed **logical variables** (**rigid**), declared locally in **quantifiers** as $\top x;$

Program Variables

Flexible constants `int i; boolean p;` used as **program variables**

Dynamic Logic Programs

Dynamic Logic = Typed FOL + programs ...

Programs here: any legal sequence of JAVA statements.

Example

Signature for FSym_f : `int r; int i; int n;`

Signature for FSym_r : `int 0; int +(int,int); int -(int,int);`

Signature for PSym_r : `<(int,int);`

```
i=0;
r=0;
while (i<n) {
    i=i+1;
    r=r+i;
}
r=r+r-n;
```

Which value does the program compute in r ?

Relating Program States: Modalities

DL extends FOL with two additional (mix-fix) operators:

- ▶ $\langle p \rangle \phi$ (diamond)
- ▶ $[p] \phi$ (box)

with p a program, ϕ another DL formula

Intuitive Meaning

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

Attention: JAVA programs are deterministic, i.e., **if** a JAVA program terminates then exactly **one** state is reached from a given initial state.

Dynamic Logic - Examples

Let i , j , old_i , old_j denote program variables.

Give the meaning in natural language:

1. $i = old_i \rightarrow \langle i = i + 1; \rangle i > old_i$

If $i = i + 1;$ is executed in a state where i and old_i have the same value, then the program terminates and in its final state the value of i is greater than the value of old_i .

2. $i = old_i \rightarrow [\text{while}(\text{true})\{i = old_i - 1;\}] i > old_i$

If the program is executed in a state where i and old_i have the same value and if the program terminates then in its final state the value of i is greater than the value of old_i .

3. $\forall x. (\langle p \rangle i = x \leftrightarrow \langle q \rangle i = x)$

p and q are equivalent concerning termination and the final value of i .

Dynamic Logic - KeY input file

— KeY —

```
\programVariables { // Declares global program variables
    int i, j;
    int old_i, old_j;
}
```

```
\problem { // The problem to verify is stated here.
    i = old_i -> \<{ i = i + 1; }\> i > old_i
}
```

— KeY —

Visibility: Program variables declared

- ▶ global can be accessed anywhere in the formula.
- ▶ inside modality like $pre \rightarrow \langle \text{int } j; p \rangle post$ only visible in p and $post$ and only if declaration on top level.

Dynamic Logic Formulas

Definition (Dynamic Logic Formulas (DL Formulas))

- ▶ Each FOL formula is a DL formula
 - ▶ If p is a program and ϕ a DL formula then $\left\{ \begin{array}{l} \langle p \rangle \phi \\ [p] \phi \end{array} \right\}$ is a DL formula
 - ▶ DL formulas closed under FOL quantifiers and connectives
-
- ▶ Program variables are **flexible constants**: never bound in quantifiers
 - ▶ Program variables need not be declared or initialized in program
 - ▶ Programs contain no logical variables
 - ▶ Modalities can be arbitrarily nested

Example (Well-formed? If yes, under which signature?)

- ▶ $\forall \text{int } y; ((\langle x = 1; \rangle x = y) \leftrightarrow (\langle x = 1*1; \rangle x = y))$

Well-formed if FSym_f contains $\text{int } x$;

- ▶ $\exists \text{int } x; [x = 1;](x = 1)$

Not well-formed, because logical variable occurs in program

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

Well-formed if FSym_f contains $\text{int } x$;
program formulas can be nested

Dynamic Logic Semantics: States

First-order state can be considered as **program state**

- ▶ Interpretation of **flexible** symbols can vary from state to state (eg, program variables, field values)
- ▶ Interpretation of **rigid** symbols is the same in all states (eg, built-in functions and predicates)

Program states as first-order states

From now, consider program state s as **first-order state** $(\mathcal{D}, \delta, \mathcal{I})$

- ▶ Only interpretation \mathcal{I} of flexible symbols in FSym_f can change
 \Rightarrow only record values of $f \in \text{FSym}_f$
- ▶ *States* is set of all states s

Kripke Structure

Definition (Kripke Structure)

Kripke structure or **Labelled transition system** $K = (States, \rho)$

- ▶ **State** (=first-order model) $s = (\mathcal{D}, \delta, \mathcal{I}) \in States$
- ▶ **Transition relation** $\rho : Program \rightarrow (States \rightarrow States)$

$$\rho(p)(s1) = s2$$

iff.

program p executed in state $s1$ terminates **and** its final state is $s2$,
otherwise undefined.

- ▶ ρ is the **semantics** of programs $\in Program$
- ▶ $\rho(p)(s)$ can be undefined (\rightarrow):
 p may **not terminate** when started in s
- ▶ Our programs are **deterministic** (unlike PROMELA):
 $\rho(p)$ is a function (at most one value)

Semantic Evaluation of Program Formulas

Definition (Validity Relation for Program Formulas)

- ▶ $s \models \langle p \rangle \phi$ iff $\rho(p)(s)$ is defined and $\rho(p)(s) \models \phi$
(p terminates and ϕ is true in the final state after execution)
- ▶ $s \models [p] \phi$ iff $\rho(p)(s) \models \phi$ whenever $\rho(p)(s)$ is defined
(If p terminates then ϕ is true in the final state after execution)

- ▶ **Duality:** $\langle p \rangle \phi$ iff $\neg [p] \neg \phi$
Exercise: justify this with help of semantic definitions
- ▶ **Implication:** if $\langle p \rangle \phi$ then $[p] \phi$
Total correctness implies partial correctness
 - ▶ converse is false
 - ▶ holds only for deterministic programs

More Examples

valid?

meaning?

Example

$$\forall \tau y; ((\langle p \rangle x = y) \leftrightarrow (\langle q \rangle x = y))$$

Not valid in general

Programs p behave q equivalently on variable τx

Example

$$\exists \tau y; (x = y \rightarrow \langle p \rangle \text{true})$$

Not valid in general

Program p terminates if initial value of x is suitably chosen

Semantics of Programs

In labelled transition system $K = (\text{States}, \rho)$:
 $\rho : \text{Program} \rightarrow (\text{States} \rightarrow \text{States})$ is **semantics** of programs $p \in \text{Program}$

ρ defined recursively on programs

Example (Semantics of assignment)

States s interpret flexible symbols f with $\mathcal{I}_s(f)$

$\rho(x=t;)(s) = s'$ where s' identical to s except $\mathcal{I}_{s'}(x) = \text{val}_s(t)$

Very tedious task to define ρ for JAVA. \Rightarrow Not in this course.
Next lecture, we go directly to calculus for program formulas!

Literature for this Lecture

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, 3.6.3, 3.6.4)

Note: Not lecture Tuesday Oct. 8.