Formal Specification and Verification of Object-Oriented Programs

Proof Obligations

TECHNISCHE UNIVERSITÄT DARMSTADT

This Part

making the connection between

JML

and

Dynamic Logic / KeY

- generating,
- understanding,
- and proving

DL proof obligations from JML specifications

From JML Contracts to DL Contracts to Proof Obligations (PO)


```
public class A {
 /*@ public normal_behavior
   @ requires <Precondition>;
   @ ensures <Postcondition>;
   @ assignable <locations>;
   @*/
 public int m(params) {..}
}
```

Translation

Functional JavaDL method contract

$F = (\mathit{pre}, \mathit{post}, \mathit{div}, \mathit{var}, \mathit{mod})$

PO Generation

Proof obligation as DL formula

$$\mathit{pre} \rightarrow \langle \mathit{this.m}(\mathit{params}); \rangle (\mathit{post}\ \&\ \mathit{frame})$$

JML Translation: Normalizing JML Contracts

Normalization of JML Contracts

1. Flattening of nested specifications
2. Making implicit specifications explicit
3. Processing of modifiers
4. Adding of default clauses if not present
5. Contraction of several clauses

We look only at some aspects of this process

Normalisation: Making Implicit Specifications Explicit

TECHNISCHE UNIVERSITÄT DARMSTADT

Implicit Specifications

- **non_null** by default
- Implicit **\invariant_for(this)** as **requires**, **ensures** & **signals** clause
- Kind of behavior

Making **non_null** explicit for method specifications

1. Deactivate implicit **non_null** by adding **nullable** to parameter and return type declarations, if of reference type and **nullable** not already present
2. Add explicit non null specifications to preconditions (for parameters) and postcondition (for return value)
   E.g., for a parameter T p add **requires** `p != null`**;** if T reference type, but not an array over reference types (more complicated in those cases)

Normalisation: Making Implicit Specifications Explicit

TECHNISCHE UNIVERSITÄT DARMSTADT

Implicit Specifications

- **non_null** by default
- Implicit **\invariant_for(this)** as **requires**, **ensures** & **signals** clause
- Kind of behavior

Making \invariant_for(this) explicit for method specifications

1. Deactivate implicit **\invariant_for(this)** by adding `helper` modifier to method (if not already present)
2. Add explicit **\invariant_for(this)** as
   - **requires \invariant_for(this);** (same for **ensures**) and
   - **signals** `(Throwable t)`**\invariant_for(this);**

Normalisation: Example

```
/*@ public normal_behavior
  @ requires c.id >= 0;
  @ ensures \result == ( ... );
  @*/
  public boolean addCategory(Category c) { ... }
```

becomes

```
/*@ public normal_behavior
  @ requires c.id >= 0;
  @ requires c != null;
  @ requires \invariant_for(this);
  @ ensures \result == (...);
  @ ensures \invariant_for(this);
  @ signals (Throwable exc) \invariant_for(this);
  @*/
  public boolean addCategory(/*@ nullable @*/Category c) { ... }
```

Normalisation: Making Implicit Specifications Explicit

TECHNISCHE UNIVERSITÄT DARMSTADT

Implicit Specifications

- **non_null** by default
- Implicit **\invariant_for(this)** as **requires**, **ensures** & **signals** clause
- Kind of behavior

Making 'kind of behavior' explicit

1. Deactivate implicit behavior specification by replacing **normal_behavior** (**exceptional_behavior**) by **behavior**
2. Add in case of replaced
   - **normal_behavior** the clause **signals** `(Throwable t)`**false**;
   - **exceptional_behavior** the clause **ensures false**;

Normalisation: Example

TECHNISCHE UNIVERSITÄT DARMSTADT

```
/*@ public behavior
  @ requires c.id >= 0;
  @ requires c != null;
  @ requires \invariant_for(this);
  @ ensures \result == (...);
  @ ensures \invariant_for(this);
  @ signals (Throwable exc) \invariant_for(this);
  @ signals (Throwable exc) false;
  @*/
  public boolean addCategory(/*@ nullable @*/Category c) { ... }
```

Normalisation

TECHNISCHE UNIVERSITÄT DARMSTADT

Implicit Specifications

- **non_null** by default
- Implicit **\invariant_for(this)** as **requires**, **ensures** & **signals** clause
- Kind of behavior

Next Normalisation Steps (Not detailed)

- Expanding **pure** modifier
- Adding default clauses (e.g., for **diverges,assignable**) if clause not present

Normalisation: Clause Contraction

Merge multiple clauses of the same kind into a single one of that kind.

For instance,

```
/*@ public behavior
  @ requires R1;
  @ requires R2;
  @ ensures E1;
  @ ensures E2;
  @ signals (T1 exc) ExcPost1
 ;
  @ signals (T2 exc) ExcPost2
 ;
  @*/
```

```
/*@ public behavior
  @ requires R1 && R2;
  @ ensures E1 && E2;
  @ signals (Throwable exc)
  @  (exc instanceof T1 ==> ExcPost1)
  @  &&
  @  (exc instanceof T2 ==> ExcPost2);
  @*/
```

Normalisation: Finishing Up

Other not considered steps (here):

- Separating functional from dependency contracts (later lecture)
- Separating diverging contracts: We consider here only contracts with either
  - **diverges false** (default for exceptional and normal behavior) or
  - **diverges true**

Translating JML into Functional DL Contracts

TECHNISCHE UNIVERSITÄT DARMSTADT

Functional DL contract F for a method m

$$F = (pre, post, div, var, mod)$$

with

- a precondition DL formula pre,
- a postcondition DL formula $post$,
- a divergence indicator $div \in \{TOTAL, PARTIAL\}$,
- a variant var a term of type `any`,
- and a modifies set mod is either a term of type `LocSet` or `\strictly_nothing`

Translating JML Expressions to DL-Terms: Arithmetic Expressions

TECHNISCHE UNIVERSITÄT DARMSTADT

Translation replaces arithmetic JAVA operators by generalized operators

Generic towards various integer semantics (JAVA, Math).

Example:

"+" becomes "`javaAddInt`" or "`javaAddLong`"
"-" becomes "`javaSubInt`" or "`javaSubLong`"
…

Translating JML Expressions to DL-Terms: The `this` Reference

The **this** reference

- explicit or
- implicit

has only a meaning within a program (refers to currently executing instance).

On logic level (outside the modalities) no such context exists.

`this` reference translated to a program variable (named by convention) `self`

e.g., given class

```
public class MyClass {
  private int f;
}
```

In JML expressions

- `f` or `this.f` translated to `select(heap, self, f)`

Translating Boolean JML Expressions

First-order logic treated fundamentally different in JML and KeY logic

JML

- Formulas no separate syntactic category
- Instead: JAVA's `boolean` expressions extended with first-order concepts (i.p. quantifiers)

Dynamic Logic

- Formulas and expressions completely separate
- Truth constants `true`, `false` are formulas, `boolean` constants `TRUE`, `FALSE` are terms
- Atomic formulas take terms as arguments; e.g.:
  - `x - y < 5`
  - `b = TRUE`

Translating Boolean JML Expressions

$$
\begin{array}{lcl}
\mathcal{F}(\texttt{v}) & = & \texttt{v = TRUE} \\
\mathcal{F}(\texttt{o.f}) & = & \mathcal{E}(\texttt{o.f}) \texttt{ = TRUE} \\
\mathcal{F}(\texttt{m()}) & = & \mathcal{E}(\texttt{m})\texttt{()} \texttt{ = TRUE} \\
\mathcal{F}(\texttt{!b\_0}) & = & \texttt{!}\mathcal{F}(\texttt{b\_0}) \\
\mathcal{F}(\texttt{b\_0 \&\& b\_1}) & = & \mathcal{F}(\texttt{b\_0})\ \&\ \mathcal{F}(\texttt{b\_1}) \\
\mathcal{F}(\texttt{b\_0 || b\_1}) & = & \mathcal{F}(\texttt{b\_0}) \mid \mathcal{F}(\texttt{b\_1}) \\
\mathcal{F}(\texttt{b\_0 ==> b\_1}) & = & \mathcal{F}(\texttt{b\_0}) \texttt{->} \mathcal{F}(\texttt{b\_1}) \\
\mathcal{F}(\texttt{b\_0 <==> b\_1}) & = & \mathcal{F}(\texttt{b\_0}) \texttt{<->} \mathcal{F}(\texttt{b\_1}) \\
\mathcal{F}(\texttt{e\_0 == e\_1}) & = & \mathcal{E}(\texttt{e\_0}) \doteq \mathcal{E}(\texttt{e\_1}) \\
\mathcal{F}(\texttt{e\_0 != e\_1}) & = & \texttt{!}\mathcal{E}(\texttt{e\_0}) \doteq \mathcal{E}(\texttt{e\_1}) \\
\mathcal{F}(\texttt{e\_0 >= e\_1}) & = & \mathcal{E}(\texttt{e\_0}) \texttt{>=} \mathcal{E}(\texttt{e\_1})
\end{array}
$$

`v`/`f`/`m()` **`boolean`** variables/fields/pure methods

`b_0`, `b_1` **`boolean`** JML expressions, `e_0`, `e_1` JML expressions

$\mathcal{E}$ translates JML expressions to DL terms

$\mathcal{F}$ Translates `boolean` JML Expressions to Formulas

Quantified formulas over reference types:

$\mathcal{F}$((**\forall** T x; e_0; e_1)) =
\forall T x; (
(!x = **null** & **boolean**:select(heap, x, <created>) & $\mathcal{F}$(e_0)) $->$ $\mathcal{F}$(e_1))

$\mathcal{F}$((**\exists** T x; e_0; e_1)) =
\exists T x; (
!x = **null** & **boolean**:select(heap, x, <created>) & $\mathcal{F}$(e_0) & $\mathcal{F}$(e_1))

$\mathcal{F}$ Translates `boolean` JML Expressions to Formulas

Quantified formulas over primitive types, e.g., **int**

$\mathcal{F}$((**\forall** int x; e_0; e_1)) =
    \forall T x; (inInt(x) & $\mathcal{F}$(e_0) -> $\mathcal{F}$(e_1))

$\mathcal{F}$((**\exists** T x; e_0; e_1)) =
    \exists T x; (inInt(x) & $\mathcal{F}$(e_0) & $\mathcal{F}$(e_1))

`inInt` (similar `inLong`, `inByte`):

Predefined predicate symbol with fixed interpretation

Meaning: Argument is within the range of the Java **int** datatype.

Translating Class Invariants

$\mathcal{F}(\textbf{\textbackslash invariant\_for}(\mathsf{e})) = \texttt{java.lang.Object}::\texttt{<inv>}(\texttt{heap}, \mathcal{E}(e))$

Later in detail, here only:

- **\invariant_for** JML expressions are translated to formulas using placeholder predicates that are abbreviations for the translated invariants using $\mathcal{F}, \mathcal{E}$
- Pretty printed as:

$$e.<inv>()$$

Translating JML into Functional DL Contracts

TECHNISCHE UNIVERSITÄT DARMSTADT

Functional DL contract F for a method m

$$F = (pre, post, div, var, mod)$$

with

- a precondition DL formula *pre* $\checkmark$,
- a postcondition DL formula *post* $\checkmark$?almost ,
- a divergence indicator $div \in \{TOTAL, PARTIAL\}$,
- a variant *var* a term of type `any`,
- and a modifies set *mod* is either a term of type `LocSet` or `\strictly_nothing`

Translation of Ensures Clauses

TECHNISCHE UNIVERSITÄT DARMSTADT

What is missing for **ensures** clauses?

- Translation of **\result**
- Translation of **\old**`(.)` expressions

Translating **\result**

The JML expression **\result** used in a ensures clause of a method *T m*(*params*) is translated to:

$$\mathcal{E}(\backslash\mathbf{result}) = \mathit{res}$$

where $\mathit{res} \in \mathit{PVar}$ of type T not occurring in the program.

Translating \old Expressions

$\backslash\mathbf{old}(e)$ evaluates e in the prestate of the method

Accesses to `heap` must be evaluated w.r.t. to the 'old' heap

1. Introduce a global program variables `heapAtPre` of type `Heap`
   (Intention: `heapAtPre` refers to the heap in the method's pre-state)
2. Define: $\mathcal{E}(\backslash\mathbf{old}(e)) = \mathcal{E}_{\mathtt{heap}}^{\mathtt{heapAtPre}}(e)$
   ($\mathcal{E}_{\mathtt{heap}}^{\mathtt{heapAtPre}}$ uses `heapAtPre` instead of `heap` for heap-sensitive expressions)

Example

$\mathcal{E}(\mathtt{o.f\ ==\ }\backslash\mathbf{old}\mathtt{(o.f)+\ 1}) =$

```
int::select(heap, o, f) = int::select(heapAtPre, o, f) + 1
```

Translation of Signals Clauses

TECHNISCHE UNIVERSITÄT DARMSTADT

```
signals (Throwable exc)
  (exc instanceof ExcType1 ==> ExcPost1) && ...;
```

Translation using $\mathcal{F}, \mathcal{E}$ as normal

A global fresh program variable *exc* of type `Throwable` is used from PVar

Combining Signals and Ensures to *post*

TECHNISCHE UNIVERSITÄT DARMSTADT

The DL formula *post* is then defined as

$$(exc \doteq \mathbf{null} \rightarrow \mathcal{F}(\mathbf{ensures}))\ \&\ (exc\, != \mathbf{null} \rightarrow \mathcal{F}(\mathbf{signals}))$$

Note: A normal behavior contract has normalized $\mathbf{signals}$ `(Throwable exc)`$\mathbf{false}$;
As expected we get:

$$\begin{array}{rl} & (exc \doteq \mathbf{null} \rightarrow \mathcal{F}(\mathbf{ensures}))\ \&\ (exc\, != \mathbf{null} \rightarrow \mathcal{F}(\mathbf{false})) \\ \Leftrightarrow & (exc \doteq \mathbf{null} \rightarrow \mathcal{F}(\mathbf{ensures}))\ \&\ (exc\, != \mathbf{null} \rightarrow \mathtt{false}) \\ \Leftrightarrow & (exc \doteq \mathbf{null} \rightarrow \mathcal{F}(\mathbf{ensures}))\ \&\ exc = \mathbf{null} \end{array}$$

Translating JML into Functional DL Contracts

Functional DL contract F for a method m

$$F = (pre, post, div, var, mod)$$

with

- a precondition DL formula pre ✓ ,
- a postcondition DL formula $post$ ✓ ,
- a divergence indicator $div \in \{TOTAL, PARTIAL\}$,✓
- a variant var a term of type `any` (postponed to later lecture),
- and a modifies set mod is either a term of type `LocSet` or `\strictly_nothing`

The Divergence Indicator

$$div = \begin{cases} TOTAL & \text{if normalised JML contract contains clause } \mathbf{diverges\ \ false;} \\ PARTIAL & \text{if normalised JML contract contains clause } \mathbf{diverges\ \ true;} \end{cases}$$

Translating Assignable Clauses: The DL Type `LocSet`

Assignable clauses are translated to

a term of type `LocSet` or the special value `\strictly_nothing`

Intention: A term of type `LocSet` represents a set of locations

Definition (Locations)

A location is a tuple (o, f) with $o \in D^{\mathtt{Object}}$, $f \in D^{\mathtt{Field}}$

Note: Location is a semantic and not a syntactic entity.

The DL Type `LocSet`

The Data Type `LocSet`

Predefined type with $D(\texttt{LocSet}) = 2^{Location}$ and the functions (incomplete):

`unique LocSet empty`	empty set of locations: $I(\texttt{empty}) = \emptyset$
`unique LocSet allLocs`	set of all locations, i.e., $I(\texttt{allLocs}) = \{(d, f) \mid f.a.\ d \in D^{\texttt{Object}}, f \in D^{\texttt{Field}}\}$
`LocSet singleton(Object, Field)`	singleton set
`LocSet union(LocSet, LocSet)`	
`LocSet intersect(LocSet, LocSet)`	
`LocSet allFields(Object)`	set of all locations for the given object
`LocSet allObjects(Field)`	set of all locations for the given field; e.g., $\{(d, f) \mid \text{f.a. } d \in D^{\texttt{Object}}\}$
`LocSet arrayRange(Object, int, int)`	set representing all array locations in the specified range (both inclusive)

Translating Assignable Clauses—Example

TECHNISCHE UNIVERSITÄT DARMSTADT

Example

```
assignable \everything;
```

is translated into the DL term

$$\mathtt{allLocs}$$

Example

```
assignable this.next, this.content[0..this.content.length-1];
```

is translated into the DL term

$$\begin{array}{l}\mathtt{union(singleton(self,next)}\\ \qquad \mathtt{arrayRange(0, javaSubInt(length(self.content), 1))}\end{array}$$

Translating JML into Functional DL Contracts

TECHNISCHE UNIVERSITÄT DARMSTADT

Functional DL contract F for a method m

$$F = (pre, post, div, var, mod)$$

with

- a precondition DL formula pre $\checkmark$,
- a postcondition DL formula $post$ $\checkmark$,
- a divergence indicator $div \in \{TOTAL, PARTIAL\}$ $\checkmark$,
- a variant var a term of type `any` (postponed),
- a modifies set mod is either a term of type `LocSet` or `\strictly_nothing` $\checkmark$

From JML Contracts to DL Contracts to Proof Obligations (PO)


```
public class A {
 /*@ public normal_behavior
   @ requires <Precondition>;
   @ ensures <Postcondition>;
   @ assignable <locations>;
   @*/
 public int m(params) {..}
}
```

Translation

Functional JavaDL method contract

$F = (\mathit{pre}, \mathit{post}, \mathit{div}, \mathit{var}, \mathit{mod})$

PO Generation

Proof obligation as DL formula

$$\mathit{pre} \rightarrow \langle \mathit{this.m}(\mathit{params});\rangle (\mathit{post} \;\&\; \mathit{frame})$$

Generating a PO from a Functional DL Contract: Idea

Given functional contract F_m for a method m implemented in class C:

$$F_m = (pre, post, \mathtt{TOTAL}, var, mod)$$

$$\Big\Downarrow \ \textbf{PO Generation}$$

$$pre \rightarrow \langle self.m(args)\rangle(post \ \ \& \ \ \underbrace{frame}_{\text{correctness of assignable}} \ )$$

(in case of *div* = `PARTIAL` box modality is used)

Generating a PO from a Functional DL Contract: Method Identification

TECHNISCHE UNIVERSITÄT DARMSTADT

$$pre \rightarrow \langle self.m(args) \rangle (post \ \ \& \ \ frame)$$

- Dynamic dispatch: `self.m(...)` causes split into all possible implementations
- Special statement Method Body Statement:

  `m(`*args*`)@package.Class`

  Meaning: Placeholder for the method body of class `package.Class`

Generating a PO from a Functional DL Contract: Exceptions

$$pre \rightarrow \langle self.m(args)@package.Class\rangle(post \;\;\&\;\; frame)$$

Postcondition *post* states either

- that no exception is thrown or
- that in case of an exception the exceptional postcondition holds

but: $\langle \mathbf{throw}\ \texttt{exc;}\rangle\varphi$ is trivially false

How to refer to an exception in post-state?

$$pre \rightarrow \left\langle \begin{array}{l} \texttt{exc = }\mathbf{null}\texttt{;} \\ \mathbf{try}\ \texttt{\{} \\ self.m(args)@package.Class \\ \texttt{\} }\mathbf{catch}\texttt{ (Throwable t)\{exc = t;\}} \end{array} \right\rangle (post \;\;\&\;\; frame)$$

(reminder: Normalisation uses program variable *exc* when translating **signals**)

The Free-Precondition *freePre*

$$pre \rightarrow \langle \texttt{exc=}\mathbf{null}\texttt{;}\ \mathbf{try}\ \texttt{\{ self.m(args)\}}\ \mathbf{catch}\ \ldots\rangle(post\ \ \&\ \ frame)$$

Additional properties (known to hold in Java, but not formalized), e.g.,

- **this** is not **null**
- created objects cannot reference not yet created objects (dangling references)
- integer parameters have correct range
- …

& need to refer to prestate in post, e.g. for old-expressions

Need to make these assumption on initial state explicit in DL! (Why?)

Idea: Formalise assumption as additional precondition *freePre* (free precondition)
Extend general shape:

$$(freePre \land pre) \rightarrow \{\texttt{heapAtPre} := \texttt{heap}\}$$
$$\langle \texttt{exc=}\mathbf{null}\texttt{;}\ \mathbf{try}\ \texttt{\{ self.m(args)\}}\ \mathbf{catch}\ \ldots\rangle(post\ \ \&\ \ frame)$$

The Free-Precondition *freePre*

$$
\begin{aligned}
\textit{freePre} := \ & \texttt{wellFormed}(\texttt{heap}) \\
\land\ & \texttt{paramsInRange} \\
\land\ & \texttt{self}\,! = \mathbf{null} \\
\land\ & \texttt{boolean} :: \texttt{select}(\texttt{heap}, \texttt{self}, <\texttt{created}>) = \texttt{TRUE} \\
\land\ & \texttt{package.Class} :: \texttt{exactInstance}(\texttt{self}) \\
\land\ & \texttt{exc} = \mathbf{null}
\end{aligned}
$$

- `wellFormed`: predefined predicate; true iff. given heap is regular Java heap
- `paramsInRange` formula stating that the method arguments are in range
- `C :: exactInstance`: predefined predicate; true iff. given argument has *C* as exact type (i.e., is not of a subtype)

Generating a PO from a Functional DL Contract: The *frame* DL Formula

$$(\mathit{freePre} \land \mathit{pre}) \rightarrow$$
$$\{\mathtt{heapAtPre} := \mathtt{heap}\}\langle \mathtt{exc=}\mathbf{null}\mathtt{;}\ \mathbf{try}\ \mathtt{\{\ self.m(args)\}}\ \mathbf{catch}\ \ldots\rangle$$
$$(\mathit{post}\ \ \&\ \ \mathit{frame})$$

If *mod* = `\strictly_nothing` then *frame* is defined

$$\forall o; \forall f; (\mathtt{any} :: \mathtt{select}(\mathtt{heapAtPre}, o, f) = \mathtt{any} :: \mathtt{select}(\mathtt{heap}, o, f))$$

If *mod* is a location set, then *frame* is defined as:

$$\begin{aligned}\forall o; \forall f; \big(\quad & \mathtt{boolean} :: \mathtt{select}(\mathtt{heaptAtPre}, o, <\mathtt{created}>) = \mathtt{FALSE} \\ & \lor\ \mathtt{any} :: \mathtt{select}(\mathtt{heapAtPre}, o, f) = \mathtt{any} :: \mathtt{select}(\mathtt{heap}, o, f) \\ & \lor\ (o, f) \in \{\mathtt{heap} := \mathtt{heapAtPre}\}\mathit{mod}\big)\end{aligned}$$

States that any location (o, f)

- belongs to an object that is not (yet) created before the method invocation or
- holds the same value after the invocation as before the invocation, or
- belongs to the modifies set (evaluated in the pre-state).

Examples

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Demo