

Formal Methods for Software Development

Proof Obligations

Wolfgang Ahrendt

16 October 2020

Master Theses in Formal Methods

- ▶ Presentation of Master thesis topics by Formal Methods group
- ▶ Thursday, 22nd Oct 10:30-11:30
- ▶ online meeting (via Zoom)
- ▶ link will be announced also via our course Canvas

making the connection between

JML

and

Dynamic Logic / KeY

- ▶ generating,
- ▶ understanding,
- ▶ and proving

DL proof obligations from JML specifications

From JML Contracts via Intermediate Format to Proof Obligations (PO)

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

Intermediate Format

(*pre*, *post*, *div*, *var*, *mod*)

Translation

PO Generation

Proof obligation as DL formula

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

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

The following introduces principles of this process

Normalisation:

Making Implicit Information Explicit

Implicit Information

- ▶ Meaning of `normal_` and `exceptional_behavior`
- ▶ `non_null` by default
- ▶ `\invariant_for(this)` in `requires`, `ensures`, `signals` clauses

Turn into **general** behavior spec. case

1. Add to
 - ▶ `normal_behavior` the clause `signals (Throwable t) false;`
 - ▶ `exceptional_behavior` the clause `ensures false;`
2. Replace `normal_behavior/exceptional_behavior` by `behavior`

Normalisation:

Making Implicit Information Explicit

Implicit Information

- ▶ Meaning of `normal_` and `exceptional_behavior`
- ▶ `non_null` by default
- ▶ `\invariant_for(this)` in `requires`, `ensures`, `signals` clauses

Making `non_null` explicit in method specifications

1. Where nullable is absent, add `o != null` to preconditions (for parameters^a) and postconditions (for return values^a).
E.g., for method `void m(Object o)` add `requires o != null;`
2. Thereafter add `nullable`, where absent, to *all* parameter^a and return type^a declarations

^aof reference type

Normalisation:

Making Implicit Information Explicit

Implicit Information

- ▶ Meaning of `normal_` and `exceptional_behavior`
- ▶ `non_null` by default
- ▶ `\invariant_for(this)` in `requires`, `ensures`, `signals` clauses

Making `\invariant_for(this)` explicit in method specifications

1. Add explicit `\invariant_for(this)` to non-helper method specs:
 - ▶ `requires \invariant_for(this);`
 - ▶ `ensures \invariant_for(this);`
 - ▶ `signals (Throwable t) \invariant_for(this);`
2. Thereafter add **helper**, where absent, to *all* methods

Normalisation: Example

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

becomes

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

Normalisation: Example

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

becomes

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

Normalisation: Example

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

becomes

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

Next Normalisation Steps (Not detailed)

- ▶ Expanding pure modifier:
 - ▶ add to each specification case
 - ▶ `assignable \nothing;`
 - ▶ `diverges false;`
 - ▶ remove pure
- ▶ Where clauses with defaults (e.g., `diverges`, `assignable`) are absent, add explicit clauses

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) S1;
  @ signals (T2 exc) S2;
  @*/
```

```
/*@ public behavior
  @ requires R1 && R2;
  @ ensures E1 && E2;
  @ signals (Throwable exc)
  @   (exc instanceof T1 ==> S1)
  @   &&
  @   (exc instanceof T2 ==> S2);
  @*/
```

Translating JML into Intermediate Format

Intermediate format for contract of method m

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

with

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

Translating JML Expressions to DL-Terms:

Arithmetic Expressions

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 {  
    int f;  
}
```

JML expressions `f` and `this.f`
translated to

DL term `select(heap, self, f)`, pretty-printed as `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
- ▶ **true**, **false** are formulas,
boolean constants **TRUE**, **FALSE** are terms
- ▶ Atomic formulas take terms as arguments; e.g.:
 - ▶ $x - y < 5$
 - ▶ $b = \text{TRUE}$

Translating Boolean JML Expressions

$\mathcal{F}(v)$	$=$	$v = \text{TRUE}$
$\mathcal{F}(o.f)$	$=$	$\mathcal{E}(o.f) = \text{TRUE}$
$\mathcal{F}(m())$	$=$	$\mathcal{E}(m)() = \text{TRUE}$
$\mathcal{F}(!b_0)$	$=$	$!\mathcal{F}(b_0)$
$\mathcal{F}(b_0 \ \&\& \ b_1)$	$=$	$\mathcal{F}(b_0) \ \& \ \mathcal{F}(b_1)$
$\mathcal{F}(b_0 \ \ b_1)$	$=$	$\mathcal{F}(b_0) \ \ \mathcal{F}(b_1)$
$\mathcal{F}(b_0 \ ==> \ b_1)$	$=$	$\mathcal{F}(b_0) \ -> \ \mathcal{F}(b_1)$
$\mathcal{F}(b_0 \ <==> \ b_1)$	$=$	$\mathcal{F}(b_0) \ <-> \ \mathcal{F}(b_1)$
$\mathcal{F}(e_0 == e_1)$	$=$	$\mathcal{E}(e_0) = \mathcal{E}(e_1)$
$\mathcal{F}(e_0 != e_1)$	$=$	$!(\mathcal{E}(e_0) = \mathcal{E}(e_1))$
$\mathcal{F}(e_0 >= e_1)$	$=$	$\mathcal{E}(e_0) >= \mathcal{E}(e_1)$

$v/f/m()$ **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:

$$\begin{aligned} \mathcal{F}((\backslash\text{forall } T \ x; e_0; e_1)) = \\ \backslash\text{forall } T \ x; (\\ \quad (!x=\text{null} \ \& \ x.<\text{created}> = \text{TRUE} \ \& \ \mathcal{F}(e_0)) \\ \quad \rightarrow \mathcal{F}(e_1)) \end{aligned}$$
$$\begin{aligned} \mathcal{F}((\backslash\text{exists } T \ x; e_0; e_1)) = \\ \backslash\text{exists } T \ x; (\\ \quad (!x=\text{null} \ \& \ x.<\text{created}> = \text{TRUE} \ \& \ \mathcal{F}(e_0)) \\ \quad \& \ \mathcal{F}(e_1)) \end{aligned}$$

\mathcal{F} Translates boolean JML Expressions to Formulas

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

$$\mathcal{F}((\backslash\text{forall } \text{int } x; e_0; e_1)) = \\ \backslash\text{forall } \text{int } x; ((\text{inInt}(x) \ \& \ \mathcal{F}(e_0)) \rightarrow \mathcal{F}(e_1))$$

$$\mathcal{F}((\backslash\text{exists } \text{int } x; e_0; e_1)) = \\ \backslash\text{exists } \text{int } x; (\text{inInt}(x) \ \& \ \mathcal{F}(e_0) \ \& \ \mathcal{F}(e_1))$$

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

Predefined predicate symbol with fixed interpretation

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

Translating Class Invariants

$\mathcal{F}(\backslash\text{invariant_for}(e)) = \text{Object} :: \langle \text{inv} \rangle(\text{heap}, \mathcal{E}(e))$

- ▶ $\backslash\text{invariant_for}(e)$ translated to built-in predicate $\text{Object} :: \langle \text{inv} \rangle$, applied to heap and the translation of e
- ▶ $\text{Object} :: \langle \text{inv} \rangle$ is considered a specification-only field $\langle \text{inv} \rangle$ of class Object (inherited by all sub-types of Object)
- ▶ Given that o is of type T , KeY can expand (during proof construction) ' $\text{Object} :: \langle \text{inv} \rangle(\text{heap}, o)$ ' to the invariant of T
- ▶ $\text{Object} :: \langle \text{inv} \rangle(\text{heap}, o)$ pretty printed as $o.\langle \text{inv} \rangle$
- ▶ Read '**invariant of o** '

Translating JML into Intermediate Format

Intermediate format for contract of method m

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

with

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

Translation of Ensures Clauses

What is missing for ensures clauses?

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

Translating `\result`

For `\result` used in ensures clause of method $T\ m(\dots)$:

$$\mathcal{E}(\backslash\text{result}) = \text{result}$$

where `result` $\in PVar$ of type T does not occur in the program.

Translating `\old` Expressions

`\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 heap in method's pre-state)

2. Define:

$$\mathcal{E}(\backslash\text{old}(e)) = \mathcal{E}_{\text{heap}}^{\text{heapAtPre}}(e)$$

($\mathcal{E}_x^y(e)$ replaces all occurrences of x in $\mathcal{E}(e)$ by y)

Example

$$\mathcal{F}(\text{o.f} == \backslash\text{old}(\text{o.f}) + 1) =$$

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

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

$$\mathcal{E}(\text{o.f}) = \mathcal{E}_{\text{heap}}^{\text{heapAtPre}}(\text{o.f}) + 1 =$$

$$\text{select}(\text{heap}, \text{o}, \text{f}) = \text{select}(\text{heapAtPre}, \text{o}, \text{f}) + 1 =$$

$$\text{o.f} = \text{o.f@heapAtPre} + 1 \quad (\text{by pretty printing})$$

Translation of Ensures and Signals Clauses

Given the **normalised** JML contract

```
/*@ public behavior
   @ ...
   @ ensures E;
   @ signals (Throwable exc) S;
   @ ...
  @*/
```

Define

$$\mathcal{F}_{\text{ensures}} = \mathcal{F}(E)$$

$$\mathcal{F}_{\text{signals}} = \mathcal{F}(S)$$

Recall (pp.8,12) that S is either false, or it has the form

(exc **instanceof** ExcType1 ==> ExcPost1) && ...;

In the following, assume exc is fresh program variable of type Throwable

Combining Ensures and Signals to *post*

The DL formula *post* is then defined as

$$(\text{exc} = \text{null} \rightarrow \mathcal{F}_{\text{ensures}}) \wedge (\text{exc} \neq \text{null} \rightarrow \mathcal{F}_{\text{signals}})$$

Important special case:

Normalisation of `normal_behavior` contract gives

`signals (Throwable exc) false;`

In that case, *post* is:

$$\begin{aligned} & (\text{exc} = \text{null} \rightarrow \mathcal{F}_{\text{ensures}}) \wedge (\text{exc} \neq \text{null} \rightarrow \mathcal{F}_{\text{signals}}) \\ \Leftrightarrow & (\text{exc} = \text{null} \rightarrow \mathcal{F}_{\text{ensures}}) \wedge (\text{exc} \neq \text{null} \rightarrow \mathcal{F}(\text{false})) \\ \Leftrightarrow & (\text{exc} = \text{null} \rightarrow \mathcal{F}_{\text{ensures}}) \wedge (\text{exc} \neq \text{null} \rightarrow \text{false}) \\ \Leftrightarrow & (\text{exc} = \text{null} \rightarrow \mathcal{F}_{\text{ensures}}) \wedge \text{exc} = \text{null} \\ \Leftrightarrow & \text{exc} = \text{null} \wedge \mathcal{F}_{\text{ensures}} \end{aligned}$$

Translating JML into Intermediate Format

Intermediate format for contract of method m

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

with

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

The Divergence Indicator

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

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 $o \in D^{\text{Object}}$, $f \in D^{\text{Field}}$

The DL Type LocSet

Predefined type with $D(\text{LocSet}) = 2^{\text{Location}}$
and the functions (all with result type LocSet):

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

Translating Assignable Clauses—Example

Example

```
assignable \everything;
```

is translated into the DL term

```
allLocs
```

Example

```
assignable this.next, this.content[5..9];
```

is translated into the DL term

```
union(singleton(self,next),  
      arrayRange(self.content,5,9))
```

Translating JML into Intermediate Format

Intermediate format for contract of method m

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

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),
- ▶ a modifies set mod , either of type `LocSet` or `\strictly_nothing` ✓

From JML Contracts via Intermediate Format to Proof Obligations (PO)

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

Intermediate Format

(*pre*, *post*, *div*, *var*, *mod*)

Translation

PO Generation

Proof obligation as DL formula

$$\begin{aligned} pre \rightarrow \\ \langle \text{this.m(params);} \rangle \\ (post \wedge frame) \end{aligned}$$

Generating a PO from the Intermediate Format: Idea

Given intermediate format of contract of m implemented in class C :

$(pre, post, TOTAL, var, mod)$



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

In case of $div = \text{PARTIAL}$, box modality is used

Generating a PO from Intermediate Format: Method Identification

$$pre \rightarrow \langle \text{self.m}(\text{args}) \rangle (post \wedge \text{frame})$$

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

`m(args)@C`

Meaning: implementation of `m` in class `C`

Generating a PO from Intermediate Format: Exceptions

$$pre \rightarrow \langle \text{self.m}(\text{args})@C \rangle (post \wedge frame)$$

Postcondition *post* states either

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

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

How to refer to an exception in post-state?

$$pre \rightarrow \left\langle \begin{array}{l} \text{exc} = \text{null}; \\ \text{try } \{ \\ \quad \text{self.m}(\text{args})@C \\ \} \text{ catch (Throwable e) \{exc = e;\} \end{array} \right\rangle (post \wedge frame)$$

Recall: generation of *post* (pp.23,24) uses program variable *exc*

The Generic Precondition *genPre*

$pre \rightarrow \langle \text{exc}=\text{null}; \text{ try } \{\text{self.m}(\text{args})@C\} \text{ catch } \dots \rangle (\text{post} \wedge \text{frame})$

is still not complete.

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

- ▶ **this** is not **null**
- ▶ created objects can only point to created objects (no dangling references)
- ▶ integer parameters have correct range
- ▶ ...

Need to make these assumption on initial state explicit in DL.

Idea: Formalise assumption as additional precondition *genPre*

$(\text{genPre} \wedge pre) \rightarrow$
 $\langle \text{exc}=\text{null}; \text{ try } \{\text{self.m}(\text{args})@C\} \text{ catch } \dots \rangle (\text{post} \wedge \text{frame})$

The Generic Precondition *genPre* (background info)

$$\begin{aligned} \textit{genPre} := & \text{ wellFormed(heap) } \\ & \wedge \text{ self } \neq \text{ null } \\ & \wedge \text{ self. } \langle \text{created} \rangle = \text{ TRUE } \\ & \wedge \text{ C :: exactInstance(self) } \\ & \wedge \textit{paramsInRange} \end{aligned}$$

- ▶ *wellFormed(h)*: predefined predicate;
true iff *h* is regular Java heap
- ▶ *C :: exactInstance(o)*: predefined predicate;
true iff *o* has exact type *C* (not just subtype of *C*)
- ▶ *paramsInRange* formula stating that method arguments are in range

The Generic Precondition *genPre*

$$(genPre \wedge pre) \rightarrow \\ \langle exc=null; \text{ try } \{self.m(args)@C\} \text{ catch } \dots \rangle (post \wedge frame)$$

is still not complete.

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

$$(genPre \wedge pre) \rightarrow \{heapAtPre := heap\} \\ \langle exc=null; \text{ try } \{self.m(args)@C\} \text{ catch } \dots \rangle (post \wedge frame)$$

Recall: *heapAtPre* was used in translation of `\old`, p.22

Generating a PO from Intermediate Format: The *frame* DL Formula

$$\begin{aligned} & (genPre \wedge pre) \rightarrow \{heapAtPre := heap\} \\ & \quad \langle exc=null; \text{ try } \{self.m(args)\} \text{ catch } \dots \rangle \\ & \quad \quad \quad (post \wedge \textit{frame}) \end{aligned}$$

If $mod = \text{\textcolor{red}{strictly_nothing}}$ then \textit{frame} is defined as:

$$\forall o; \forall f; (o.f = o.f@heapAtPre)$$

Generating a PO from Intermediate Format: The *frame* DL Formula

$$(genPre \wedge pre) \rightarrow \{heapAtPre := heap\} \\ \langle exc=null; \text{ try } \{self.m(args)\} \text{ catch } \dots \rangle \\ (post \wedge \textcolor{blue}{frame})$$

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

$$\forall o; \forall f; \left(\begin{array}{l} (o, f) \in \{heap := heapAtPre\} mod \\ \vee o.<created>@heapAtPre = FALSE \\ \vee o.f = o.f@heapAtPre \end{array} \right)$$

Says that every location (o, f) either

- ▶ belongs to the modifies set (evaluated in the pre-state), or
- ▶ was not (yet) created before the method invocation, or
- ▶ holds the same value before and after the method execution

Generating a PO from Intermediate Format: Result Value

$$(genPre \wedge pre) \rightarrow \{heapAtPre := heap\} \\ \langle exc=null; \text{try } \{self.m(args)\} \text{ catch } \dots \rangle \\ (post \wedge frame)$$

is still not complete.

► For non-void methods, need to refer to result in *post*

$$(genPre \wedge pre) \rightarrow \{heapAtPre := heap\} \\ \langle exc=null; \text{try } \{result = self.m(args)\} \text{ catch } \dots \rangle \\ (post \wedge frame)$$

Recall: `\result` was translated to program variable `result`, see p.21

Examples

Demo