Formal Specification and Verification

Proof Obligations

Bernhard Beckert

Based on a lecture by Wolfgang Ahrendt and Reiner Hähnle at Chalmers University, Göteborg
making the connection between

JML

and

Dynamic Logic / KeY
This Part

making the connection between

JML

and

Dynamic Logic / KeY

▶ generating,
making the connection between

JML

and

Dynamic Logic / KeY

generating,

understanding,
This Part

making the connection between

JML

and

Dynamic Logic / KeY

generating,
understanding,
and proving

DL proof obligations from JML specifications
Tutorial Example

we follow ‘KeY Quicktour for JML’ (cited below as [KQJ])

scenario: simple PayCard
Inspecting JML Specification

inspect quicktour/jml/paycard/PayCard.java

follow [KQJ, 2.2]
New JML Feature I: Nested Specification Cases

method charge() has nested specification case:

@ public normal_behavior
@ requires amount > 0;
@ {
@  requires amount + balance < limit && isValid() == true;
@  ensures \result == true;
@  ensures balance == amount + \old(balance);
@  assignable balance;
@
@
@  also
@
@
@  requires amount + balance >= limit;
@  ensures \result == false;
@  ensures unsuccessfulOperations
@    == \old(unsuccessfulOperations) + 1;
@  assignable unsuccessfulOperations;
@  |
}
Nested Specification Cases

nested specification cases allow to factor out common preconditions

@ public normal_behavior
@ requires R;
@ {|
@   requires R1;
@   ensures E1;
@   assignable A1;
@
@   also
@
@   requires R2;
@   ensures E2;
@   assignable A2;
@ |}

expands to ... (next page)
Nested Specification Cases

(previous page) ... expands to

@ public normal_behavior
@ requires R;
@ requires R1;
@ ensures E1;
@ assignable A1;
@
@
@ also
@
@
@ public normal_behavior
@ requires R;
@ requires R2;
@ ensures E2;
@ assignable A2;
@ public normal_behavior
@ requires amount>0;
@ {
@ requires amount+balance<limit && isValid()==true;
@ ensures \result == true;
@ ensures balance == amount + \old(balance);
@ assignable balance;
@
@ also
@
@ requires amount + balance >= limit;
@ ensures \result == false;
@ ensures unsuccessfulOperations
@ == \old(unsuccessfulOperations) + 1;
@ assignable unsuccessfulOperations;
@ }

expands to ... (next page)
Nested Specification Cases

(Previous page) ... expands to

@ public normal_behavior
@ requires amount > 0;
@ requires amount + balance < limit && isValid() == true;
@ ensures \result == true;
@ ensures balance == amount + \old(balance);
@ assignable balance;
@

@ also
@
@ public normal_behavior
@ requires amount > 0;
@ requires amount + balance >= limit;
@ ensures \result == false;
@ ensures unsuccessfulOperations
@ == \old(unsuccessfulOperations) + 1;
@ assignable unsuccessfulOperations;
method charge() has exceptional behavior case:
@ public exceptional_behavior
@ requires amount <= 0;
@ assignable \nothing;
New JML Feature II: assignable \nothing

method charge() has exceptional behavior case:
@ public exceptional_behavior
@ requires amount <= 0;
@ assignable \nothing;

assignable \nothing prohibits side effects

difference to pure:
  ▶ pure also prohibits non-termination
  ▶ assignable clause is local to specification case
    (here: local to exceptional_behavior)
generate **EnsuresPost** PO for normal behavior of `charge()`
Generating Proof Obligations (POs)

generate **EnsuresPost** PO for normal behavior of `charge()`

follow [KQJ, 3.1+3.2]

summary:

- start KeY prover
- in quicktour/jml, open paycard
- select charge and **EnsuresPost**
- inspect **Assumed Invariants**
Generating Proof Obligations (POs)

generate **EnsuresPost** PO for normal behavior of `charge()`

follow [KQJ, 3.1+3.2]

summary:
- start KeY prover
- in quicktour/jml, open paycard
- select charge and **EnsuresPost**
- inspect **Assumed Invariants**
  assuming less invariants:
  - is fully sound
  - can compromise provability
Generating Proof Obligations (POs)

generate **EnsuresPost** PO for normal behavior of `charge()`

follow [KQJ, 3.1+3.2]

**summary:**
- start KeY prover
- in `quicktour/jml`, open `paycard`
- select `charge` and **EnsuresPost**
- inspect **Assumed Invariants**
  assuming less invariants:
  - is fully sound
  - can compromise provability

**sometimes invariants of other classes also needed** (select class+inv.)
Generating Proof Obligations (POs)

**generate EnsurePost PO for normal behavior of charge()**

follow [KQJ, 3.1+3.2]

**summary:**

- start KeY prover
- in quicktour/jml, open paycard
- select charge and EnsurePost
- inspect Assumed Invariants
  assuming less invariants:
  - is fully sound
  - can compromise provability
  
  **sometimes invariants of other classes also needed** (select class+inv.)
- select contract which modifies balance
Generating Proof Obligations (POs)

generate **EnsuresPost** PO for normal behavior of `charge()`

follow [KQJ, 3.1+3.2]

summary:
- start KeY prover
- in quicktour/jml, open paycard
- select charge and **EnsuresPost**
- inspect **Assumed Invariants**
  assuming less invariants:
  - is fully sound
  - can compromise provability
  sometimes invariants of *other* classes also needed (select class+inv.)
- select contract which **modifies** balance
  (in JML: **modifies** synonymous for **assignable**)
Generating Proof Obligations (POs)

generate **EnsuresPost** PO for normal behavior of `charge()`

follow [KQJ, 3.1+3.2]

summary:
- start KeY prover
- in quicktour/jml, open paycard
- select charge and **EnsuresPost**
- inspect **Assumed Invariants**
  assuming less invariants:
  - is fully sound
  - can compromise provability
- sometimes invariants of other classes also needed (select class+inv.)
- select contract which **modifies** balance
  (in JML: **modifies** synonymous for **assignable**)
- **Current Goal** pane displays **proof obligation** as DL sequent
Generating Proof Obligations

for loading more proof obligations:
re-open Proof Obligation Browser under Tools menu

generate EnsuresPost PO for normal behavior of isValid()
Generating Proof Obligations

for loading more proof obligations:
re-open **Proof Obligation Browser** under **Tools** menu

- generate **EnsuresPost** PO for normal behavior of isValid()
- generate **EnsuresPost** PO for exceptional behavior of charge()
Generating Proof Obligations

for loading more proof obligations:
re-open **Proof Obligation Browser** under **Tools** menu

generate **EnsuresPost** PO for normal behavior of `isValid()`

generate **EnsuresPost** PO for exceptional behavior of `charge()`

generate **PreservesOwnInv** PO for `charge()`

expressing that `charge()` preserves all invariants (of its own class)
Generating Proof Obligations

for loading more proof obligations:
re-open **Proof Obligation Browser** under **Tools** menu

- generate **EnsuresPost** PO for normal behavior of isValid()
- generate **EnsuresPost** PO for exceptional behavior of charge()
- generate **PreservesOwnInv** PO for charge()

expressing that charge() preserves all invariants (of its own class)

follow [KQJ, 4.3.1+4.3.2]
Translating JML to POs in DL

in the following:

- issues in translating arithmetic expressions
- translating this
- identifying the method’s implementation
- translating boolean JML expressions to first-order logic formulas
- translating preconditions
- translating class invariants
- translating postconditions
- storing \texttt{old} fields prior to method invocation
- storing actual parameters prior to method invocation
- expressing that 'exceptions are (not) thrown'
- putting everything together
Translating JML to POs in DL

WARNING:

following presentation is
- incomplete
- not fully precise
- simplifying
- omitting details/complications
- deviating from exact implementation in KeY
Translating JML to POs in DL

WARNING:

following presentation is

▸ incomplete
▸ not fully precise
▸ simplifying
▸ omitting details/complications
▸ deviating from exact implementation in KeY

aim of the following:

enable you to read/understand proof obligations
Translating JML to POs in DL

WARNING:

following presentation is

▶ incomplete
▶ not fully precise
▶ simplifying
▶ omitting details/complications
▶ deviating from exact implementation in KeY

aim of the following:

enable you to read/understand proof obligations

(notational remark: stick to ASCII syntax of KeY logic in this lecture)
Issues on Translating Arithmetic Expressions

often:

- KeY replaces arithmetic Java operators by generalized operators, generic towards various integer semantics (Java, Math), example: “+” becomes “javaAddInt”
- KeY inserts casts like (jint), needed for type hierarchy among primitive types, example: “0” becomes “(jint)(0)”
Issues on Translating Arithmetic Expressions

often:

- KeY replaces arithmetic JAVA operators by generalized operators, generic towards various integer semantics (JAVA, Math), example: “+” becomes “javaAddInt”
- KeY inserts casts like (jint), needed for type hierarchy among primitive types, example: “0” becomes “(jint)(0)”

(no need to memorize this)
Translating this

both

- explicit
- implicit

*this* reference translated to *self*
Translating this

both
  ▶ explicit
  ▶ implicit

this reference translated to self

e.g., given class

```java
public class MyClass {
    ...
    private int f;
    ...
}
```
Translating this

both
  ▶ explicit
  ▶ implicit
this reference translated to self

e.g., given class
public class MyClass {
  ...
  private int f;
  ...
}

▶ f translated to self.f
▶ this.f translated to self.f
Java’s dynamic dispatch selects a method’s implementation \textit{at runtime}
JAVA’s dynamic dispatch selects a method’s implementation \emph{at runtime}

for a method call \texttt{m(args)},
KeY models selection of implementation from \texttt{package.Class} by
\texttt{m(args)@package.Class}
Java’s dynamic dispatch selects a method’s implementation at runtime for a method call $m(\text{args})$, KeY models selection of implementation from `package.Class` by $m(\text{args})@\text{package.Class}$

Example:

```
charge(x)@paycard.PayCard
```

executes class `paycard.PayCard`’s implementation of method call `charge(x)`
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)

KeY logic

► formulas and expressions completely separate
► truth constants true, false are formulas,
  boolean constants TRUE, FALSE are expressions
► atomic formulas take expressions as arguments; e.g.:
  ► x - y < 5
  ► b = TRUE
F Translates boolean JML Expressions to Formulas

\[
\begin{align*}
F(v) & = v = \text{TRUE} \\
F(f) & = T(f) = \text{TRUE} \\
F(m()) & = T(m)() = \text{TRUE} \\
F(!b_0) & = \neg F(b_0) \\
F(b_0 \&\& b_1) & = F(b_0) \& F(b_1) \\
F(b_0 \mid\mid b_1) & = F(b_0) \mid F(b_1) \\
F(b_0 \Rightarrow b_1) & = F(b_0) \Rightarrow F(b_1) \\
F(b_0 \Leftrightarrow b_1) & = F(b_0) \Leftrightarrow F(b_1) \\
F(e_0 == e_1) & = \mathcal{E}(e_0) = \mathcal{E}(e_1) \\
F(e_0 != e_1) & = \neg \mathcal{E}(e_0) = \mathcal{E}(e_1) \\
F(e_0 >= e_1) & = \mathcal{E}(e_0) >= \mathcal{E}(e_1)
\end{align*}
\]

v,f,m() boolean variables/fields/pure methods  
b_0, b_1 boolean JML expressions  
e_0, e_1 JAVA expressions

\(T\) may add ‘self.’ or ‘@ClassName’ (see pp.16,17)  
\(E\) may add casts, transform operators (see p.15)
\( \mathcal{F}( (\forall T x; \ e_0) ) = \forall T x; \quad !x=\text{null} \rightarrow \mathcal{F}(e_0) \)

\( \mathcal{F}( (\exists T x; \ e_0) ) = \exists T x; \quad !x=\text{null} \land \mathcal{F}(e_0) \)

\( \mathcal{F}( (\forall T x; \ e_0; e_1) ) = \forall T x; \quad !x=\text{null} \land \mathcal{F}(e_0) \rightarrow \mathcal{F}(e_1) \)

\( \mathcal{F}( (\exists T x; \ e_0; e_1) ) = \exists T x; \quad !x=\text{null} \land \mathcal{F}(e_0) \land \mathcal{F}(e_1) \)
if selected contract \( Contr \) has preconditions

\@ requires \( b_1 \);
\@ ... \\
\@ requires \( b_n \);

they are translated to
Translating Preconditions

if selected contract \( \text{Contr} \) has \text{preconditions}

\@ \text{requires } b_1;
\@ \ldots 
\@ \text{requires } b_n;

they are translated to

\[
\mathcal{PRE}(\text{Contr}) = \mathcal{F}(b_1) \& \ldots \& \mathcal{F}(b_n)
\]
the invariant

class C {
    ...
    //@ invariant inv_i;
    ...
}

is translated to
Translating Class Invariants

The invariant

```java
class C {
    ...
    //@ invariant inv_i;
    ...
}
```

is translated to

\[
\forall C \ o; ((o.<\text{created}\text{>} = \text{TRUE} \ & \ !o = \text{null}) \rightarrow \{\text{self:=o}\}F(\text{inv_i}))
\]
if selected contract $Contr$ has \textit{postconditions}

\begin{verbatim}
@ ensures b_1;
@ ...
@ ensures b_n;
\end{verbatim}

they are translated to
Translating Postconditions

if selected contract *Contr* has postconditions

@ ensures b_1;
@ ...
@ ensures b_n;

they are translated to

\[ POST(Contr) = \mathcal{F}(b_1) \land \ldots \land \mathcal{F}(b_n) \]
Translating Postconditions

if selected contract \( Contr \) has postconditions
\[ @ \text{ ensures } b_1; \]
\[ @ \ldots \]
\[ @ \text{ ensures } b_n; \]

they are translated to

\[
\text{POST}(Contr) = \mathcal{F}(b_1) \& \ldots \& \mathcal{F}(b_n)
\]

special treatment of expressions in post-condition: see next slide
Translating Expressions in Postconditions

below, we assume the following assignable clause

@ assignable <assignable_fields> ;
Translating Expressions in Postconditions

below, we assume the following assignable clause

@ assignable <assignable_fields>;

translating expressions in postconditions (interesting cases only):

\[ E(\text{result}) = \text{result} \]

\[ E(\text{old}(e)) = E_{old}(e) \]

\[ E_{old} \text{ defined like } E, \text{ with the exception of:} \]

\[ E_{old}(e.f) = f\text{AtPre}(E_{old}(e)) \]

\[ E_{old}(f) = f\text{AtPre}(\text{self}) \]

for \( f \in <\text{assignable_fields}> \)
below, we assume the following assignable clause

@ assignable <assignable_fields>;

translating expressions in postconditions (interesting cases only):

\[ E(\text{result}) = \text{result} \]

\[ E(\text{old}(e)) = E_{old}(e) \]

\( E_{old} \) defined like \( E \), with the exception of:

\[ E_{old}(e.f) = fAtPre(E_{old}(e)) \]

\[ E_{old}(f) = fAtPre(self) \]

for \( f \in <assignable\_fields> \)

‘fAtPre’ meant to refer to field ‘\( f \) in the pre-state’
given an assignable field \( f \) of class \( C \)

```java
class C {
    ...
    private T f;
    ...
}
```

translation of postcondition replaced \( f \) in \( \text{old}(..) \) by \( fAtPre \) (p.24)

left to do: store pre-state values of \( f \) in \( fAtPre \)
Storing Pre-State of a Field

given an assignable field $f$ of class $C$

```java
class C {
    ...
    private $T f;
    ...
}
```

translation of postcondition replaced $f$ in $\text{old}(..)$ by $f\text{AtPre}$ (p.24)

left to do: store pre-state values of $f$ in $f\text{AtPre}$

\[
STORE(f) = \]

formal specification and verification: proof obligations

B. Beckert 39 / 55
given an **assignable** field `f` of class `C`

class `C` {
    ...
    private `T` `f`;
    ...
}

translation of postcondition replaced `f` in `\texttt{old}(..)` by `fAtPre` (p.24)
left to do: store pre-state values of `f` in `fAtPre`

\[
\text{STORE}(f) = \\texttt{for C o; fAtPre(o) := o.f}
\]
Storing Pre-State of a Field

given an assignable field $f$ of class $C$

```java
class C {
    ...
    private $T f$;
    ...
}
```

translation of postcondition replaced $f$ in `old(..)` by $fAtPre$ (p.24)

left to do: store pre-state values of $f$ in $fAtPre$

$$STORE(f) = \text{\LaTeX code}$$

note: not a formula, but
Storing Pre-State of a Field

given an assignable field \( f \) of class \( C \)

```java
class C {
    ...
    private T f;
    ...
}
```

translation of postcondition replaced \( f \) in `\( \text{old}(..) \)` by \( fAtPre \) (p.24)

left to do: store pre-state values of \( f \) in \( fAtPre \)

\[
STORE(f) = \forall C \ o; \ fAtPre(o) := o.f
\]

note: not a formula, but a quantified update
if selected contract $Contr$ has preconditions

@ assignable $f_1, \ldots, f_n$;

then pre-state of all assignable fields can be stored by
if selected contract $Contr$ has preconditions

@ assignable $f_1, \ldots, f_n$;

then pre-state of all assignable fields can be stored by one parallel update:
if selected contract $Contr$ has preconditions

@ assignable $f_1, \ldots, f_n$;

then pre-state of all assignable fields can be stored by one parallel update:

$$STORE(Contr) = \{ \, STORE(f_1) \parallel \ldots \parallel STORE(f_n) \, \}$$
Expressing Normal Termination

how can you express in DL:
method call \texttt{m()} will \textbf{not} throw an exception
Expressing Normal Termination

how can you express in DL:
method call m() will not throw an exception
(if method body from class C in package p is invoked)
Expressing Normal Termination

how can you express in DL:
method call \texttt{m()} will \textbf{not} throw an exception
(if method body from class \texttt{C} in package \texttt{p} is invoked)

\{ exc = \texttt{null};
  try {
    \texttt{m()}@p.C;
  } catch (java.lang.Throwable e) {
    exc = e;
  }
} \Rightarrow exc = \texttt{null}
how can you express in DL:
method call m() will not throw an exception
(if method body from class C in package p is invoked)

\(<\{\text{exc} = \text{null};\text{try}\{\text{m()}@p.C;\text{catch (java.lang.Throwable e)}\{\text{exc} = e;\}\}\}\rangle\text{exc} = \text{null}

note difference:

- JAVA assignments
- equation, i.e., formula (in KeY output format)
Expressing Exceptional Termination

how can you express in DL:
method call m() will throw an exception
Expressing Exceptional Termination

how can you express in DL:
method call \texttt{m()} \textbf{will} throw an exception
(if method body from class \texttt{C} in package \texttt{p} is invoked)
Expressing Exceptional Termination

how can you express in DL:
method call \texttt{m()} \textbf{will} throw an exception
(if method body from class \texttt{C} in package \texttt{p} is invoked)

\[<\{ \texttt{exc} = \texttt{null}; \]
\[ \texttt{try} \{ \]
\[ \quad \texttt{m()}@\texttt{p.C}; \]
\[ \} \texttt{catch} (\texttt{java.lang.Throwable e}) \{ \]
\[ \quad \texttt{exc} = \texttt{e}; \]
\[ \} \] \[\textbf{!exc} = \texttt{null} \]
Expressing Exceptional Termination

how can you express in DL:
method call \( m() \) will throw an exception
(if method body from class \( C \) in package \( p \) is invoked)

\(<\{ \) exc = \texttt{null};
    try {
        m()@p.C;
    } \ catch (java.lang.Throwable e) {
        exc = e;
    }
\}> !exc = \texttt{null} & <typing of exc>
PO for Normal Behavior Contract

PO for a normal behavior contract $Contr$ for void method $m()$, with chosen assumed invariants $inv_1, ..., inv_n$

$$\Rightarrow$$

$$INV(inv_1)$$

$$\& ...$$

$$\& INV(inv_n)$$

$$\& PRE(Contr)$$

$$\Rightarrow STORE(Contr)$$

\begin{verbatim}
\(<\{ exc = null;
     try {
       m()@p.C;
     } catch (java.lang.Throwable e) {
       exc = e;
     }
   }\> exc = null \& POST(Contr)
\end{verbatim}
PO for a normal behavior contract $Contr$ for method $m()$, where $Contr$ has clause $\text{diverges true;}$

$\implies$

$INV(inv_1)$

$\& \ldots$

$\& INV(inv_n)$

$\& PRE(Contr)$

$\rightarrow STORE(Contr)$

\[
\{ \text{exc = null; try } \{
    \text{m()@p.C; }
    } \text{ catch (java.lang.Throwable e) } \{
    \text{exc = e; }
    \} \}
\}\] exc = null $\& POST(Contr)$
PO for Normal Behavior of Non-Void Method

PO for a normal behavior contract $Contr$ for non-void method $m()$,

$$
\Rightarrow
$$

\begin{align*}
\text{INV}(& \text{inv}_1) \\
& \text{...} \\
& \text{INV}(& \text{inv}_n) \\
& \text{PRE}(Contr) \\
\Rightarrow & \text{STORE}(Contr) \\
\langle& \text{exc} = \text{null;} \\
& \text{try} \{ \\
& \text{result} = m()@p.C; \\
& \text{catch} \text{ (java.lang.Throwable e)} \{ \\
& \text{exc} = e; \\
& \} \\
\rangle & \text{exc} = \text{null \& POST}(Contr)
\end{align*}
PO for Normal Behavior of Non-Void Method

PO for a normal behavior contract $Contr$ for non-void method $m()$, 

$$\Rightarrow \quad INV(inv_1)$$

$$& \ldots$$

$$& INV(inv_n)$$

$$& PRE(Contr)$$

$$\Rightarrow STORE(Contr)$$

$$\{\text{exc} = \text{null;}

\text{try} \{

\text{result} = m()@p.C;

\text{catch} (\text{java.lang.Throwable e}) \{

\text{exc} = e;

\}

\} \Rightarrow \text{exc} = \text{null} \& POST(Contr)$$

recall: $POST(Contr)$ translated $\backslash result$ to $\text{result}$ (p.24)
assume method \( m() \) has contracts \( Contr_1, \ldots, Contr_j \)

PO stating that:

Invariants \( inv_1, \ldots, inv_n \) are preserved in all cases covered by a contract.

\[
\Rightarrow \\
\text{INV}(inv_1) \land \ldots \land \text{INV}(inv_n) \\
\land (\text{PRE}(Contr_1) \lor \ldots \lor \text{PRE}(Contr_1))
\]

\[
\Rightarrow \left\{ \begin{array}{l}
\text{exc} = \text{null;}
\text{try} \{
\text{m()}@p.C;
\} \text{ catch (java.lang.Throwable e) } \{
\text{exc} = e;
\}
\}\right\
\text{INV}(inv_1) \land \ldots \land \text{INV}(inv_n)
\]
Examples

don’t fit on slide: execute quicktour with KeY instead
Literature for this Lecture

Essential

KeY Quicktour