Software Engineering using Formal Methods

Proof Obligations

Wolfgang Ahrendt & Richard Bubel & Wojciech Mostowski

11 October 2011

This Part

making the connection between

JML

and

Dynamic Logic / KeY

This Part

making the connection between

JML

and

Dynamic Logic / KeY

- generating,

This Part

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])

paper + sources:
see 'KeY Quicktour' on course page, under 'Links, Papers, and Software'

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;
```

Nested Specification Cases

```
@ 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;
```

Recall: pure vs. assignable \nothing

method `charge()` has exceptional behavior case:

```
@ public exceptional_behavior
@   requires amount <= 0;
@   assignable \nothing;
```

Recall: `pure` vs. `assignable \nothing`

method `charge()` has exceptional behavior case:

```
@ public exceptional_behavior
@   requires amount <= 0;
@   assignable \nothing;
```

`assignable \nothing` prohibits side effects

Recall: `pure` vs. `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`** is method-global, also prohibits non-termination & exceptions
- **`assignable`** clause is local to specification case
  (here, there is only one)

Recall: `pure` vs. `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`** is method-global, also prohibits non-termination & exceptions
- **`assignable`** clause is local to specification case (here, there is only one)
- **`pure`** not usable in this particular context

Generating Proof Obligations (POs)

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 **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`

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 (or **Ctrl-B**)

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

Generating Proof Obligations

for loading more proof obligations:
re-open **Proof Obligation Browser** under **Tools** menu (or **Ctrl-B**)

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 (or **Ctrl-B**)

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 (or **Ctrl-B**)

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:

principles of translating JML to proof obligations in DL

- 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 **`\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`"

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

```
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`

Identifying the Method's Implementation

JAVA's dynamic dispatch selects a method's implementation *at runtime*

Identifying the Method's Implementation

JAVA's dynamic dispatch selects a method's implementation *at runtime*

for a method call `m(`*args*`)`,
KeY models selection of implementation from `package.Class` by
`m(`*args*`)@package.Class`

Identifying the Method's Implementation

JAVA's dynamic dispatch selects a method's implementation *at runtime*

for a method call `m(`*args*`)`,
KeY models selection of implementation from `package.Class` by
`m(`*args*`)@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`

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

$\mathcal{F}$(`v`)	=	`v = TRUE`
$\mathcal{F}$(`f`)	=	$\mathcal{T}$(`f`) `= TRUE`
$\mathcal{F}$(`m()`)	=	$\mathcal{T}$(`m`)`() = 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` JAVA expressions

$\mathcal{T}$ may add '`self.`' or '`@ClassName`' (see pp. 16, 17)

$\mathcal{E}$ may add casts, transform operators (see p. 15)

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

$$\mathcal{F}(\texttt{(\textbackslash forall T x; e\_0)}) = \texttt{\textbackslash forall T x; !x = null -> } \mathcal{F}(\texttt{e\_0})$$

$$\mathcal{F}(\texttt{(\textbackslash exists T x; e\_0)}) = \texttt{\textbackslash exists T x; !x = null \& } \mathcal{F}(\texttt{e\_0})$$

$$\mathcal{F}(\texttt{(\textbackslash forall T x; e\_0; e\_1)}) = \texttt{\textbackslash forall T x; !x = null \& } \mathcal{F}(\texttt{e\_0}) \texttt{ -> } \mathcal{F}(\texttt{e\_1})$$

$$\mathcal{F}(\texttt{(\textbackslash exists T x; e\_0; e\_1)}) = \texttt{\textbackslash exists T x; !x = null \& } \mathcal{F}(\texttt{e\_0}) \texttt{ \& } \mathcal{F}(\texttt{e\_1})$$

Translating Preconditions

if selected contract *Contr* has preconditions

```
@ requires b_1;
@ ...
@ requires b_n;
```

they are translated to

Translating Preconditions

if selected contract *Contr* has preconditions

```
@ requires b_1;
@ ...
@ requires b_n;
```

they are translated to

$$\mathcal{PRE}(Contr) = \mathcal{F}(\texttt{b\_1})\ \texttt{\&}\ \ldots\ \texttt{\&}\ \mathcal{F}(\texttt{b\_n})$$

Translating Class Invariants

the invariant

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

is translated to

Translating Class Invariants

the invariant

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

is translated to

$$\mathcal{INV}(\texttt{inv\_i})$$

$$=$$

```
\forall C o; ((o.<created> = TRUE & !o = null) ->
                                      {self:=o}F(inv_i))
```

Translating Postconditions

if selected contract *Contr* has postconditions

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

they are translated to

Translating Postconditions

if selected contract *Contr* has postconditions

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

they are translated to

$$\mathcal{POST}(\mathit{Contr}) = \mathcal{F}(\texttt{b\_1})\ \texttt{\&}\ \ldots\ \texttt{\&}\ \mathcal{F}(\texttt{b\_n})$$

Translating Postconditions

if selected contract *Contr* has postconditions

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

they are translated to

$$\mathcal{POST}(\mathit{Contr}) = \mathcal{F}(\texttt{b\_1}) \ \texttt{\&} \ \ldots \ \texttt{\&} \ \mathcal{F}(\texttt{b\_n})$$

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):

$$\mathcal{E}(\texttt{\textbackslash result}) = \texttt{result}$$

$$\mathcal{E}(\texttt{\textbackslash old(e)}) = \mathcal{E}_{old}(\texttt{e})$$

$\mathcal{E}_{old}$ defined like $\mathcal{E}$, with the exception of:

$$\mathcal{E}_{old}(\texttt{e.f}) = \texttt{fAtPre}(\mathcal{E}_{old}(\texttt{e}))$$
$$\mathcal{E}_{old}(\texttt{f}) = \texttt{fAtPre(self)}$$

for f $\in$ *<assignable_fields>*

Translating Expressions in Postconditions

below, we assume the following assignable clause

```
@ assignable <assignable_fields>;
```

translating expressions in postconditions (interesting cases only):

$$\mathcal{E}(\texttt{\textbackslash result}) = \texttt{result}$$

$$\mathcal{E}(\texttt{\textbackslash old(e)}) = \mathcal{E}_{old}(\texttt{e})$$

$\mathcal{E}_{old}$ defined like $\mathcal{E}$, with the exception of:

$$\mathcal{E}_{old}(\texttt{e.f}) = \texttt{fAtPre}(\mathcal{E}_{old}(\texttt{e}))$$
$$\mathcal{E}_{old}(\texttt{f}) = \texttt{fAtPre(self)}$$

for $\texttt{f} \in$ *<assignable_fields>*

'`fAtPre`' meant to refer to field '`f`' *in the pre-state*

Storing Pre-State of a Field

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

```
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

Storing Pre-State of a Field

given an **`assignable`** field `f` of class `C`

```
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`

$$\mathcal{STORE}(\texttt{f})$$
$$=$$

Storing Pre-State of a Field

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

```
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

$$\mathcal{STORE}(\texttt{f})$$
$$=$$
$$\texttt{\textbf{\char`\\for} C o; fAtPre(o) := o.f}$$

Storing Pre-State of a Field

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

```
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

$$\mathcal{STORE}(\texttt{f})$$

$$=$$

\for C o; fAtPre(o) := o.f

note: not a formula, but

Storing Pre-State of a Field

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

```
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

$$\mathcal{STORE}(\texttt{f})$$

$$=$$

\for C o; fAtPre(o) := o.f

note: not a formula, but a quantified update
(more proper explanation tomorrow)

Storing Pre-State of All Assignable Fields

if selected contract *Contr* has assignable clause:

```
@ assignable f_1, ..., f_n;
```

then pre-state of *all* assignable fields can be stored by

Storing Pre-State of All Assignable Fields

if selected contract *Contr* has assignable clause:

```
@ assignable f_1, ..., f_n;
```

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

Storing Pre-State of All Assignable Fields

if selected contract *Contr* has assignable clause:

```
@ assignable f_1, ..., f_n;
```

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

$$\mathcal{STORE}(Contr) \\ = \\ \{\ \mathcal{STORE}(\texttt{f\_1})\ \texttt{||}\ \ldots\ \texttt{||}\ \mathcal{STORE}(\texttt{f\_n})\ \}$$

Expressing Normal Termination

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

```
\<{ exc = null;
    try {
      m()@p.C;
    } catch (Throwable e) {
      exc = e;
    }
  }\> exc = null
```

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)

```
\<{ exc = null;
    try {
      m()@p.C;
    } catch (Throwable e) {
      exc = e;
    }
  }\> exc = null
```

note difference:

- JAVA assignments
- equation, i.e., formula

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 `m()` will throw an exception
(if method body from class `C` in package `p` is invoked)

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 = null;
    try {
      m()@p.C;
    } catch (Throwable e) {
      exc = e;
    }
  }\> !exc = 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 = null;
    try {
      m()@p.C;
    } catch (Throwable e) {
      exc = e;
    }
  }\> !exc = 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`

```
==>
      INV(inv_1)
    & ...
    & INV(inv_n)
    & PRE(Contr)
 -> STORE(Contr)
      \<{ exc = null;
          try {
            m()@p.C;
          } catch (Throwable e) {
            exc = e;
          }
        }\> exc = null & POST(Contr)
```

PO for Normal Behavior Allowing Non-Termination

PO for a normal behavior contract *Contr* for method `m()`,
where *Contr* has clause **`diverges true;`**

```
==>
      INV(inv_1)
    & ...
    & INV(inv_n)
    & PRE(Contr)
 -> STORE(Contr)
      \[{ exc = null;
          try {
            m()@p.C;
          } catch (Throwable e) {
            exc = e;
          }
        }\] exc = null & POST(Contr)
```

PO for Normal Behavior of Non-Void Method

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

```
==>
      INV(inv_1)
    & ...
    & INV(inv_n)
    & PRE(Contr)
 -> STORE(Contr)
      \<{ exc = null;
          try {
            result = m()@p.C;
          } catch (Throwable e) {
            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()`,

```
==>
      INV(inv_1)
    & ...
    & INV(inv_n)
    & PRE(Contr)
 -> STORE(Contr)
      \<{ exc = null;
          try {
            result = m()@p.C;
          } catch (Throwable e) {
            exc = e;
          }
        }\> exc = null & POST(Contr)
```

recall: $\mathcal{POST}$(*Contr*) translates **`\result`** to `result` (p. 24)

PO for Preserving Invariants

assume method `m()` has contracts $Contr_1$, ..., $Contr_j$

PO stating that:

Invariants `inv_1`, ..., `inv_n` are preserved
in all cases covered by a contracts.

```
==>
     INV(inv_1) & ... & INV(inv_n)
   & ( PRE(Contr_1) | ...  | PRE(Contr_1) )
 -> \[{ exc = null;
        try {
          m()@p.C;
        } catch (Throwable e) {
          exc = e;
        }
     }\] INV(inv_1) & ... & INV(inv_n)
```

Examples

don't fit on slide: execute quicktour with KeY instead

Literature for this Lecture

Essential

KeY Quicktour see course page, under 'Links, Papers, and Software'