Specifying the Java Collections Framework in JavaDL

cand. inform. Denis Lohner

Institut für Theoretische Informatik - Universität Karlsruhe

6th KeY Symposium, 2007

Betreuer: Dipl.Inform. R. Bubel
verantw. Betreuer: Prof. Dr. P. H. Schmitt
Outline

1. Motivation
2. Specification by example
3. Interface specification
4. Using specifications
5. A "new" method contract rule
6. Demo
7. Conclusion
Outline

1 Motivation

2 Specification by example

3 Interface specification

4 Using specifications

5 A "new" method contract rule

6 Demo

7 Conclusion
Outline

1. Motivation
2. Specification by example
3. Interface specification
4. Using specifications
5. A "new" method contract rule
6. Demo
7. Conclusion
Outline

1 Motivation
2 Specification by example
3 Interface specification
4 Using specifications
5 A ”new” method contract rule
6 Demo
7 Conclusion
Outline

1 Motivation
2 Specification by example
3 Interface specification
4 Using specifications
5 A "new" method contract rule
6 Demo
7 Conclusion
Outline

1. Motivation
2. Specification by example
3. Interface specification
4. Using specifications
5. A “new” method contract rule
6. Demo
7. Conclusion
Motivation

Problem

1. *No sources of the JDK library available in KeY*  
   ⇒ *symbolical execution of library calls fail*

2. *For native methods sources not even exist*

Why specifying the Java Collections Framework?

- JCF used in many projects
- Case study
Motivation

Problem

1. *No sources of the JDK library available in KeY*
   ⇒ *symbolical execution of library calls fail*

2. *For native methods sources not even exist*

Why specifying the Java Collections Framework?

- JCF used in many projects
- Case study
Motivation

Problem
1. No sources of the JDK library available in KeY
   ⇒ symbolical execution of library calls fail
2. For native methods sources not even exist

Why specifying the Java Collections Framework?
- JCF used in many projects
- Case study
Motivation

Problem

1. No sources of the JDK library available in KeY
   ⇒ symbolical execution of library calls fail
2. For native methods sources not even exist

Why specifying the Java Collections Framework?

- JCF used in many projects
- Case study
Motivation

Problem

1. No sources of the JDK library available in KeY
   ⇒ symbolical execution of library calls fail
2. For native methods sources not even exist

Why specifying the Java Collections Framework?

- JCF used in many projects
- Case study
Specification by example

Normal case

Example Method

SomeLibrary.copy(java.lang.Object[] src, java.lang.Object[] dest)

Precondition

src != null & src.<created> = TRUE &
dest != null & dest.<created> = TRUE &
src.length = dest.length &
\forall int i; ( (0 <= i & i < src.length) ->
arrayStoreValid(dest, src[i]) )

Postcondition

\forall int i; ( (0 <= i & i < src.length) -> dest[i] = src[i] )

Modifies

dest[0 .. src.length]
Specification by example

Example Method

SomeLibrary.copy(java.lang.Object[] src, java.lang.Object[] dest)

Precondition

src != null & src.<created> = TRUE &
dest != null & dest.<created> = TRUE &
src.length = dest.length &
\forall int i; ( (0 <= i & i < src.length) ->
arrayStoreValid(dest, src[i]) )

Postcondition

\forall int i; ( (0 <= i & i < src.length) -> dest[i] = src[i] )

Modifies

dest[0 .. src.length]
Specification by example

Normal case

Example Method

SomeLibrary.copy(java.lang.Object[] src, java.lang.Object[] dest)

Precondition

src != null & src.<created> = TRUE &
dest != null & dest.<created> = TRUE &
src.length = dest.length &
\forall int i; ( (0 <= i & i < src.length) ->
arrayStoreValid(dest, src[i]) )

Postcondition

\forall int i; ( (0 <= i & i < src.length) -> dest[i] = src[i] )

Modifies

dest[0 .. src.length]
**Specification by example**

**Normal case**

**Example Method**

```java
SomeLibrary.copy(java.lang.Object[] src, java.lang.Object[] dest)
```

**Precondition**

```plaintext
src !null & src.<created> = TRUE &
dest !null & dest.<created> = TRUE &
src.length = dest.length &
\forall int i; ( (0 <= i & i < src.length) -> arrayStoreValid(dest, src[i]) )
```

**Postcondition**

```plaintext
\forall int i; ( (0 <= i & i < src.length) -> dest[i] = src[i] )
```

**Modifies**

```plaintext
dest[0 .. src.length]
```
Specification by example

Normal case

Example Method
SomeLibrary.copy(java.lang.Object[] src, java.lang.Object[] dest)

Precondition
src != null & src.<created> = TRUE &
dest != null & dest.<created> = TRUE &
src.length = dest.length &
\forall int i; ( (0 <= i & i < src.length) ->
arrayStoreValid(dest, src[i]) )

Postcondition
\forall int i; ( (0 <= i & i < src.length) -> dest[i] = src[i] )

Modifies
dest[0 .. src.length]
Specification by example

Normal case

Example Method

SomeLibrary.copy(java.lang.Object[] src, java.lang.Object[] dest)

Precondition

src != null & src.<created> = TRUE &
dest != null & dest.<created> = TRUE &
src.length = dest.length &
\forall int i; ( (0 <= i & i < src.length) ->
arrayStoreValid(dest, src[i]) )

Postcondition

\forall int i; ( (0 <= i & i < src.length) -> dest[i] = src[i] )

Modifies

dest[0 .. src.length]
Specification by example

Normal case

Example Method

```java
SomeLibrary.copy(java.lang.Object[] src, java.lang.Object[] dest)
```

Precondition

```plaintext
src != null & src.<created> = TRUE &
dest != null & dest.<created> = TRUE &
src.length = dest.length &
\forall int i; ( (0 <= i & i < src.length) ->
arrayStoreValid(dest, src[i]) )
```

Postcondition

```plaintext
\forall int i; ( (0 <= i & i < src.length) -> dest[i] = src[i] )
```

Modifies

```plaintext
dest[0 .. src.length]
```
Precondition

\[
\begin{align*}
\text{src.length} &= \text{dest.length} & \\
\forall \text{i} &\in \text{Int}; (0 \leq \text{i} \land \text{i} < \text{src.length}) \rightarrow \\
\text{arrayStoreValid(dest, src[i])}
\end{align*}
\]

Postcondition

\[
\begin{align*}
\text{exc} = \text{null} &\rightarrow \\
\forall \text{i} &\in \text{Int}; (0 \leq \text{i} \land \text{i} < \text{src.length}) \rightarrow \text{dest[i]} = \text{src[i]} \\
\land \\
\text{exc} \neq \text{null} &\rightarrow \\
( \\
\text{NullPointerException}::\text{instance(exc)} = \text{TRUE} & \\
( \text{src} \neq \text{null} \land \text{dest} \neq \text{null} ) \rightarrow \\
( \text{NullPointerException}::\text{instance(exc)} = \text{FALSE} ) & \\
\text{NullPointerException}::\text{instance(exc)} = \text{TRUE} \rightarrow \\
( \text{dest} = \text{null} | \forall \text{i}; \text{dest[i]} = \text{dest[i]@pre} )
\end{align*}
\]
Specification by example

**Exceptional case**

**Precondition**

\[
\text{src.length} = \text{dest.length} \land \\
\forall i \in \mathbb{N} \mid 0 \leq i < \text{src.length} \Rightarrow \text{arrayStoreValid}(\text{dest}, \text{src}[i])
\]

**Postcondition**

\[
\text{exc} = \text{null} \Rightarrow \\
\forall i \in \mathbb{N} \mid 0 \leq i < \text{src.length} \Rightarrow \text{dest}[i] = \text{src}[i]
\]

&

\[
\text{exc} \neq \text{null} \Rightarrow \\
( \text{NullPointerException::instance}(\text{exc}) = \text{TRUE} \land \\
( \text{src} \neq \text{null} \land \text{dest} \neq \text{null}) \Rightarrow \\
( \text{NullPointerException::instance}(\text{exc}) = \text{FALSE}) \land \\
\text{NullPointerException::instance}(\text{exc}) = \text{TRUE} \Rightarrow \\
( \text{dest} = \text{null} \lor \\
\forall i \in \mathbb{N} \mid 0 \leq i < \text{src.length} \Rightarrow \text{dest}[i] = \text{dest}[i]@\text{pre})
\]
**Specification by example**

**Exceptional case**

### Precondition

```latex
text{src.length} = \text{dest.length} \land \\
\forall \text{i} \in \mathbb{N}; (0 \leq \text{i} \land \text{i} < \text{src.length}) \rightarrow \\
\text{arrayStoreValid(dest, src[i]) }
```

### Postcondition

```latex
\text{exc} = \text{null} \rightarrow \\
\forall \text{i} \in \mathbb{N}; (0 \leq \text{i} \land \text{i} < \text{src.length}) \rightarrow \text{dest[i]} = \text{src[i]}
\land \\
\text{exc} \neq \text{null} \rightarrow \\
( \\
\text{NullPointerException::instance(exc)} = \text{TRUE} \land \\
( \text{src} \neq \text{null} \land \text{dest} \neq \text{null} ) \rightarrow \\
( \text{NullPointerException::instance(exc)} = \text{FALSE} ) \land \\
\text{NullPointerException::instance(exc)} = \text{TRUE} \rightarrow \\
( \text{dest} = \text{null} \lor \forall \text{i} \in \mathbb{N}; \text{dest[i]} = \text{dest[i]}@pre )
)```

Denis Lohner

Specifying the JCF
### Exceptional case

#### Precondition

\[
\text{src.length} = \text{dest.length} & \\
\forall i \in \mathbb{N} \mid (0 \leq i < \text{src.length}) \rightarrow \\
\text{arrayStoreValid(dest, src[i])}
\]

#### Postcondition

\[
\text{exc} = \text{null} \rightarrow \\
\forall i \in \mathbb{N} \mid (0 \leq i < \text{src.length}) \rightarrow \text{dest}[i] = \text{src}[i] \\
\]

\[
\text{exc} \neq \text{null} \rightarrow \\
( \\
\text{NullPointerException}::\text{instance(exc)} = \text{TRUE} & \\
( \text{src} \neq \text{null} \& \text{dest} \neq \text{null} ) \rightarrow \\
( \text{NullPointerException}::\text{instance(exc)} = \text{FALSE} ) & \\
\text{NullPointerException}::\text{instance(exc)} = \text{TRUE} \rightarrow \\
( \text{dest} = \text{null} | \forall i \in \mathbb{N} \mid \text{dest}[i] = \text{dest}[i]@\text{pre} )
\)
Specification by example

Exceptional case

Precondition

\[
\begin{align*}
\text{src.length} &= \text{dest.length} \\ 
\forall i \in \text{src.length} : (0 \leq i \land i < \text{src.length}) &\implies \\
\text{arrayStoreValid(dest, src[i])}
\end{align*}
\]

Postcondition

\[
\begin{align*}
exc = \text{null} &\implies \\
\forall i \in \text{src.length} : (0 \leq i \land i < \text{src.length}) &\implies \text{dest[i]} = \text{src[i]}
\end{align*}
\]

\[
\begin{align*}
exc \neq \text{null} &\implies \\
( \\
\text{NullPointerException::instance(exc) = TRUE} &\land \\
( \text{src} \neq \text{null} \land \text{dest} \neq \text{null} ) &\implies \\
( \text{NullPointerException::instance(exc) = FALSE} &\land \\
\text{NullPointerException::instance(exc) = TRUE} &\implies \\
( \text{dest} = \text{null} | \forall i \in \text{src.length} : \text{dest[i]} = \text{dest[i}@pre} 
)\end{align*}
\]
Specification by example
Exceptional case

**Precondition**

\[
\text{src.length = dest.length} \land \\
\forall \text{int } i; (0 \leq i \land i < \text{src.length}) \rightarrow \\
\text{arrayStoreValid(dest, src[i])}
\]

**Postcondition**

\[
\text{exc = null} \rightarrow \\
\forall \text{int } i; (0 \leq i \land i < \text{src.length}) \rightarrow \text{dest[i] = src[i]}
\]

\[
\text{exc \neq null} \rightarrow \\
( \\
\text{NullPointerException::instance(exc) = TRUE} \land \\
( \text{src \neq null} \land \text{dest \neq null} ) \rightarrow \\
( \text{NullPointerException::instance(exc) = FALSE} ) \land \\
\text{NullPointerException::instance(exc) = TRUE} \rightarrow \\
( \text{dest = null} \lor \forall \text{int } i; \text{dest[i] = dest[i]@pre} )
\]
General Concept for specifying methods

**Precondition**

Nearly all the time "true"

**Postcondition**

Let $\phi_N$ be the postcondition for normal behaviour
Let $\psi_{Exc_i} (1 \leq i \leq n, n \in \mathbb{N})$ be the condition where the exception $Exc_i$ is thrown
Let $\phi_{Exc_i}$ be the postcondition that holds after $Exc_i$ has been thrown
Then the postcondition should look like this:

\[
( \text{exc} = \text{null} \rightarrow \phi_N ) \& \\
\text{exc} \neq \text{null} \rightarrow \\
( \bigvee_i Exc_i::\text{instance}(\text{exc}) = \text{TRUE} ) \& \\
\bigwedge_i ( !\psi_{Exc_i} \rightarrow Exc_i::\text{instance}(\text{exc}) = \text{FALSE} ) \& \\
\bigwedge_i ( Exc_i::\text{instance}(\text{exc}) = \text{TRUE} \rightarrow \phi_{Exc_i} )
\]
General Concept for specifying methods

**Precondition**

Nearly all the time "true"

**Postcondition**

Let \( \phi_N \) be the postcondition for normal behaviour

Let \( \psi_{Exc_i} \) \((1 \leq i \leq n, n \in N)\) be the condition where the exception \( Exc_i \) is thrown

Let \( \phi_{Exc_i} \) be the postcondition that holds after \( Exc_i \) has been thrown

Then the postcondition should look like this:

\[
\begin{align*}
( \text{exc} = \text{null} \rightarrow \phi_N ) & \land \\
\text{exc} \neq \text{null} \rightarrow \\
( & ( \lor_i Exc_i::\text{instance} ( \text{exc} ) = \text{TRUE} ) & \land \\
& \land_i ( \neg \psi_{Exc_i} \rightarrow Exc_i::\text{instance} ( \text{exc} ) = \text{FALSE} ) & \land \\
& \land_i ( Exc_i::\text{instance} ( \text{exc} ) = \text{TRUE} \rightarrow \phi_{Exc_i} )
\end{align*}
\]
General Concept for specifying methods

Precondition

Nearly all the time "true"

Postcondition

Let $\phi_N$ be the postcondition for normal behaviour
Let $\psi_{Exc_i} \ (1 \leq i \leq n, n \in N)$ be the condition where the exception $Exc_i$ is thrown
Let $\phi_{Exc_i}$ be the postcondition that holds after $Exc_i$ has been thrown
Then the postcondition should look like this:

( exc = null -> $\phi_N$ ) &
exc != null ->
( 
  ( $\forall_i Exc_i ::\text{instance(exc)} = \text{TRUE}$ ) &
  $\land_i ( \neg \psi_{Exc_i} \rightarrow Exc_i ::\text{instance(exc)} = \text{FALSE} )$ &
  $\land_i ( Exc_i ::\text{instance(exc)} = \text{TRUE} \rightarrow \phi_{Exc_i} )$
)
General Concept for specifying methods

Precondition

Nearly all the time "true"

Postcondition

Let $\phi_N$ be the postcondition for normal behaviour
Let $\psi_{Exc_i} (1 \leq i \leq n, n \in N)$ be the condition where the exception $Exc_i$ is thrown
Let $\phi_{Exc_i}$ be the postcondition that holds after $Exc_i$ has been thrown

Then the postcondition should look like this:

\[
( \text{exc} = \text{null} \rightarrow \phi_N ) \& \\
\text{exc} \neq \text{null} \rightarrow \\
( \\
( \lor_i Exc_i::\text{instance(exc)} = \text{TRUE} ) \& \\
\land_i( !\psi_{Exc_i} \rightarrow Exc_i::\text{instance(exc)} = \text{FALSE} ) \& \\
\land_i( Exc_i::\text{instance(exc)} = \text{TRUE} \rightarrow \phi_{Exc_i} )
)\]
General Concept for specifying methods

**Precondition**

Nearly all the time "true"

**Postcondition**

Let $\phi_N$ be the postcondition for normal behaviour
Let $\psi_{Exc_i} \ (1 \leq i \leq n, n \in N)$ be the condition where the exception $Exc_i$ is thrown
Let $\phi_{Exc_i}$ be the postcondition that holds after $Exc_i$ has been thrown

Then the postcondition should look like this:

\[
( \text{exc} = \text{null} \rightarrow \phi_N ) \& \\
\text{exc} \neq \text{null} \rightarrow \\
( \bigvee_i \text{Exc}_i::instance(\text{exc}) = \text{TRUE} ) \& \\
\bigwedge_i ( \neg \psi_{Exc_i} \rightarrow \text{Exc}_i::instance(\text{exc}) = \text{FALSE} ) \& \\
\bigwedge_i ( \text{Exc}_i::instance(\text{exc}) = \text{TRUE} \rightarrow \phi_{Exc_i} )
\]
General Concept for specifying methods

Precondition

Nearly all the time "true"

Postcondition

Let $\phi_N$ be the postcondition for normal behaviour
Let $\psi_{Exc_i} \ (1 \leq i \leq n, n \in N)$ be the condition where the exception $Exc_i$ is thrown
Let $\phi_{Exc_i}$ be the postcondition that holds after $Exc_i$ has been thrown
Then the postcondition should look like this:

\[
( \text{exc} = \text{null} \rightarrow \phi_N ) \& \\
\text{exc} \neq \text{null} \rightarrow \\
( ( \bigvee_i Exc_i::\text{instance(exc)} = \text{TRUE} ) \& \\
\bigwedge_i ( \neg \psi_{Exc_i} \rightarrow Exc_i::\text{instance(exc)} = \text{FALSE} ) \& \\
\bigwedge_i ( Exc_i::\text{instance(exc)} = \text{TRUE} \rightarrow \phi_{Exc_i} ) 
)\]
General Concept for specifying methods

Precondition

Nearly all the time "true"

Postcondition

Let $\phi_N$ be the postcondition for normal behaviour
Let $\psi_{Exc_i} (1 \leq i \leq n, n \in N)$ be the condition where the exception $Exc_i$ is thrown
Let $\phi_{Exc_i}$ be the postcondition that holds after $Exc_i$ has been thrown

Then the postcondition should look like this:

\[
\begin{align*}
( \text{exc} = \text{null} & \rightarrow \phi_N ) \& \\
\text{exc} \neq \text{null} & \rightarrow \\
( & ( \bigvee_i Exc_i::instance(\text{exc}) = \text{TRUE} ) \& \\
& \bigwedge_i ( \neg \psi_{Exc_i} \rightarrow Exc_i::instance(\text{exc}) = \text{FALSE} ) \& \\
& \bigwedge_i ( Exc_i::instance(\text{exc}) = \text{TRUE} \rightarrow \phi_{Exc_i} )
\end{align*}
\]
General Concept for specifying methods

**Precondition**
Nearly all the time "true"

**Postcondition**
Let $\phi_N$ be the postcondition for normal behaviour
Let $\psi_{Exc_i} \ (1 \leq i \leq n, n \in N)$ be the condition where the exception $Exc_i$ is thrown
Let $\phi_{Exc_i}$ be the postcondition that holds after $Exc_i$ has been thrown
Then the postcondition should look like this:

$$
( \text{exc} = \text{null} \rightarrow \phi_N ) \& \\
\text{exc} \neq \text{null} \rightarrow \\
( \\
\quad ( \bigvee_i \text{Exc}_i::\text{instance}(\text{exc}) = \text{TRUE} ) \& \\
\quad \bigwedge_i ( !\psi_{Exc_i} \rightarrow \text{Exc}_i::\text{instance}(\text{exc}) = \text{FALSE} ) \& \\
\quad \bigwedge_i ( \text{Exc}_i::\text{instance}(\text{exc}) = \text{TRUE} \rightarrow \phi_{Exc_i} )
$$
**Problem**

*Method behaviour is described by attribute changes*

*But:*

*Interfaces don't contain any attributes*

**Solution**

Introduce some function symbols for storing necessary information ("model functions")

E.g. \nonRigid[Location] int .size(java.util.List) for remembering a Lists actual size
Problem

*Method behaviour is described by attribute changes*

*But:*

*Interfaces don’t contain any attributes*

Solution

Introduce some function symbols for storing necessary information ("model functions")

E.g. `\nonRigid[Location] int _size(java.util.List)` for remembering a List's actual size
**Interface specification**

**Example**

### Method to be specified

\[ s = \text{myList}.\text{size}() @ \text{java.util.List}; \]

\[ \text{with } s \subseteq \text{jint and myList } \subseteq \text{java.util.List} \]

### Precondition

\[ \text{true} \]

### Postcondition

\[ \text{if } (\text{size}(\text{myList}) \leq \text{java.lang.Integer.MAX\_VALUE}) \]
\[ \text{then } (s = \text{size}(\text{myList})) \]
\[ \text{else } (s = \text{java.lang.Integer.MAX\_VALUE}) \]

### Modifies

\[ s \]
Interface specification

Example

Method to be specified

s = myList.size() @ java.util.List;
with \( s \in \text{jint} \) and \( \text{myList} \in \text{java.util.List} \)

Precondition

true

Postcondition

\( \text{if (size(myList) \leq \text{java.lang.Integer.MAX VALUE})} \)
\( \text{then (s = size(myList))} \)
\( \text{else (s = java.lang.Integer.MAX VALUE)} \)

Modifies

s
Interface specification

Example

Method to be specified

\[ s = \text{myList}.\text{size}() @ \text{java.util.List}; \]
\[ \text{with } s \in \text{jint and myList} \in \text{java.util.List} \]

Precondition

true

Postcondition

\[
\text{if (size(myList) } \leq \text{ java.lang.Integer.MAX\_VALUE) }
\text{then (s = size(myList))}
\text{else (s = java.lang.Integer.MAX\_VALUE)}
\]

Modifies

s
**Interface specification**

**Example**

**Method to be specified**

\[ s = \text{myList.size()} @ \text{java.util.List}; \]

with \[ s \subseteq \text{jint} \text{ and myList} \subseteq \text{java.util.List} \]

**Precondition**

\[ \text{true} \]

**Postcondition**

\[ \text{if (size(myList) <= java.lang.Integer.MAX\_VALUE)} \]
\[ \text{then (s = size(myList))} \]
\[ \text{else (s = java.lang.Integer.MAX\_VALUE)} \]

**Modifies**

\[ s \]
Interface specification
Problems with model functions

Introducing model methods yields to two additional problems.

1. How to initialize a model function?

Answer
Write a method contract for the \texttt{<init>} function of the appropriate class.

Symbolical execution $\leftrightarrow$ use of method contracts

Solution
Never use both for the same object in one proof and assure correctness by

- Proof obligation inserts new non rigid predicate
- Check in preconditions of contracts for it
Introducing model methods yields to two additional problems.

1. How to initialize a model function?

**Answer**

Write a method contract for the `<init>` function of the appropriate class

2. Symbolical execution <-> use of method contracts

**Solution**

Never use both for the same object in one proof and assure correctness by

- Proof obligation inserts new non rigid predicate
- Check in preconditions of contracts for it
Introducing model methods yields to two additional problems.

1. How to initialize a model function?

**Answer**

Write a method contract for the `<init>` function of the appropriate class.

2. Symbolical execution <-> use of method contracts

**Solution**

Never use both for the same object in one proof and assure correctness by

- Proof obligation inserts new non rigid predicate
- check in preconditions of contracts for it
Introducing model methods yields to two additional problems.

1. **How to initialize a model function?**

   **Answer**
   Write a method contract for the `<init>` function of the appropriate class.

2. **Symbolical execution <-> use of method contracts**

   **Solution**
   Never use both for the same object in one proof and assure correctness by
   - Proof obligation inserts new non rigid predicate
   - check in preconditions of contracts for it
Introducing model methods yields to two additional problems.

1. How to initialize a model function?

**Answer**

Write a method contract for the `<init>` function of the appropriate class.

2. Symbolical execution `<->` use of method contracts

**Solution**

Never use both for the same object in one proof and assure correctness by

- Proof obligation inserts new non rigid predicate
- Check in preconditions of contracts for it
Introducing model methods yields to two additional problems.

1. How to initialize a model function?

**Answer**

Write a method contract for the `<init>` function of the appropriate class

2. Symbolical execution <-> use of method contracts

**Solution**

Never use both for the same object in one proof and assure correctness by

- Proof obligation inserts new non rigid predicate
- check in preconditions of contracts for it
Loading of contracts

The Library mechanism of KeY is used to load the contracts, i.e. the specifications are stored in KeY-files.

Application of Contracts

Applying contracts within a proof is done via the MethodContractRule.
Using specification

Libraries

Loading of contracts
The Library mechanism of KeY is used to load the contracts, i.e. the specifications are stored in KeY-files

Application of Contracts
Applying contracts within a proof is done via the MethodContractRule
Using specifications

Let $S$, $T$ be types with $S \sqsubseteq T$

Let $\text{obj} \sqsubseteq S$

Method call vs. method body statement

- Method call

  \[
  \text{obj.m}(\text{params})
  \]

  will be expanded to

- Method body statement

  \[
  \text{obj.m}(\text{params})@T
  \]

  where $T$ specifies where to find the implementation of $\text{m}(\text{params})$
Using specifications

Let $S$, $T$ be types with $S \sqsubseteq T$
Let $\text{obj} \sqsubseteq S$

Method call vs. method body statement

- Method call
  
  $\text{obj.m}(params)$

  will be expanded to

- Method body statement
  
  $\text{obj.m}(params)@T$

  where $T$ specifies where to find the implementation of $m(params)$
Let $S, T$ be types with $S \subseteq T$
Let $obj \in S$

Method call vs. method body statement

- **Method call**
  
  $obj.m(params)$

  will be expanded to

- **Method body statement**
  
  $obj.m(params)@T$

  where $T$ specifies where to find the implementation of $m(params)$
Let S, T be types with $S \sqsubseteq T$
Let $\text{obj.m}(params)@T$ be a method body statement with $\text{obj} \sqsubseteq S$

### Which contracts are available?

Contracts written for Method $m(params)$ in type $T$ or a supertype

### Which contracts should be available?

Contracts written for Method $m(params)$ in type $S$ or a supertype
Using specifications
Behavioral subtyping

Let $S$, $T$ be types with $S \sqsubseteq T$.
Let $\text{obj}.m(params)@T$ be a method body statement with $\text{obj} \sqsubseteq S$.

**Which contracts are available?**
Contracts written for Method $m(params)$ in type $T$ or a supertype.

**Which contracts should be available?**
Contracts written for Method $m(params)$ in type $S$ or a supertype.
Let $S, T$ be types with $S \subseteq T$

Let $\text{obj}.m(params)@T$ be a method body statement with $\text{obj} \sqsubseteq S$

**Which contracts are available?**
Contracts written for Method $m(params)$ in type $T$ or a supertype

**Which contracts should be available?**
Contracts written for Method $m(params)$ in type $S$ or a supertype
Let $S$, $T$ be types with $S \sqsubseteq T$

Let $\text{obj}.m(params)@T$ be a method body statement with \text{obj} $\sqsubseteq S$

### Which contracts are available?

Contracts written for Method $m(params)$ in type $T$ or a supertype

### Which contracts should be available?

Contracts written for Method $m(params)$ in type $S$ or a supertype
Using specifications

Problems

- MethodContractRule available only on method body statement
  ⇒ Possible huge proof split up (e.g. java.util.List has many subtypes), hence same proof has to be done n times

Solution

Adapt MethodContractRule to use method call

- Used specifications must be proven

Solution

Need possibility to give feedback which contracts can not be proven
(native methods)
Using specifications

Problems

- MethodContractRule available only on method body statement
  ⇒ Possible huge proof split up (e.g. java.util.List has many subtypes), hence same proof has to be done n times

Solution

Adapt MethodContractRule to use method call

- Used specifications must be proven

Solution

Need possibility to give feedback which contracts can not be proven (native methods)
Using specifications

Problems

- MethodContractRule available only on method body statement
  ⇒ Possible huge proof split up (e.g. java.util.List has many subtypes), hence same proof has to be done n times

Solution

Adapt MethodContractRule to use method call

- Used specifications must be proven

Solution

Need possibility to give feedback which contracts can not be proven (native methods)
A "new" method contract rule

Remember from creating specifications

Let $\phi_N$ be the postcondition for normal behaviour
Let $\psi_{\text{Exc}_i} \ (1 \leq i \leq n, n \in \mathbb{N})$ be the condition where the exception $\text{Exc}_i$ is thrown
Let $\phi_{\text{Exc}_i}$ be the postcondition that holds after $\text{Exc}_i$ has been thrown

Let $\text{Exc}_1$ to $\text{Exc}_k (1 \leq k \leq n)$ be caught by a program
A "new" method contract rule

Remember from creating specifications

Let $\phi_N$ be the postcondition for normal behaviour
Let $\psi_{\text{Exc}_i} (1 \leq i \leq n, n \in \mathbb{N})$ be the condition where the exception $\text{Exc}_i$ is thrown
Let $\phi_{\text{Exc}_i}$ be the postcondition that holds after $\text{Exc}_i$ has been thrown
Let $\text{Exc}_1$ to $\text{Exc}_k (1 \leq k \leq n)$ be caught by a program
A "new" method contract rule

Remember from creating specifications

Let $\phi_N$ be the postcondition for normal behaviour
Let $\psi_{Exc_i} (1 \leq i \leq n, n \in \mathbb{N})$ be the condition where the exception $Exc_i$ is thrown
Let $\phi_{Exc_i}$ be the postcondition that holds after $Exc_i$ has been thrown

Let $Exc_1$ to $Exc_k (1 \leq k \leq n)$ be caught by a program
A "new" method contract rule

Remember from creating specifications

Let $\phi_N$ be the postcondition for normal behaviour
Let $\psi_{Exc_i} \ (1 \leq i \leq n, n \in N)$ be the condition where the exception $Exc_i$ is thrown
Let $\phi_{Exc_i}$ be the postcondition that holds after $Exc_i$ has been thrown

Let $Exc_1$ to $Exc_k (1 \leq k \leq n)$ be caught by a program
A "new" method contract rule

Then the contract that should be applied is

**Precondition**

\[ \bigwedge_{k<i\leq n} \neg \psi_{\text{Exc}_i} \]

**Postcondition**

( \text{exc} = \text{null} \rightarrow \phi_N ) \&

\text{exc} \neq \text{null} \rightarrow

( ( \bigvee_{1\leq i\leq k} \text{Exc}_i::\text{instance}(\text{exc}) = \text{TRUE} ) \&

\bigwedge_{1\leq i\leq k} ( \neg \psi_{\text{Exc}_i} \rightarrow \text{Exc}_i::\text{instance}(\text{exc}) = \text{FALSE} ) \&

\bigwedge_{1\leq i\leq k} ( \text{Exc}_i::\text{instance}(\text{exc}) = \text{TRUE} \rightarrow \phi_{\text{Exc}_i} ) )
A "new" method contract rule

Then the contract that should be applied is

**Precondition**

\[ \bigwedge_{k < i \leq n} !\psi_{\text{Exc}_i} \]

**Postcondition**

\[
( \text{exc} = \text{null} \rightarrow \phi_N ) \&
\text{exc} \neq \text{null} \rightarrow \\
( \bigvee_{1 \leq i \leq k} \text{Exc}_i::\text{instance}(\text{exc}) = \text{TRUE} ) \&
\bigwedge_{1 \leq i \leq k} ( !\psi_{\text{Exc}_i} \rightarrow \text{Exc}_i::\text{instance}(\text{exc}) = \text{FALSE} ) \&
\bigwedge_{1 \leq i \leq k} ( \text{Exc}_i::\text{instance}(\text{exc}) = \text{TRUE} \rightarrow \phi_{\text{Exc}_i} )
\]
A "new" method contract rule

Then the contract that should be applied is

**Precondition**

\[ \bigwedge_{k < i \leq n} \neg \psi_{Exc_i} \]

**Postcondition**

\[
( \text{exc} = \text{null} \rightarrow \phi_N ) \land \\
\text{exc} \neq \text{null} \rightarrow \\
( \\
\quad ( \bigvee_{1 \leq i \leq k} \text{Exc}_i::instance(\text{exc}) = \text{TRUE} ) \land \\
\quad \bigwedge_{1 \leq i \leq k} (!\psi_{Exc_i} \rightarrow \text{Exc}_i::instance(\text{exc}) = \text{FALSE}) \land \\
\quad \bigwedge_{1 \leq i \leq k} (\text{Exc}_i::instance(\text{exc}) = \text{TRUE} \rightarrow \phi_{Exc_i}) \\
)\]
A "new" method contract rule

Then the contract that should be applied is

**Precondition**

\[
\bigwedge_{k<i\leq n} \neg \psi_{Exc_i}
\]

**Postcondition**

\[
( \text{exc} = \text{null} \rightarrow \phi_N ) \land \text{exc} \neq \text{null} \rightarrow \\
( \\
\bigvee_{1\leq i\leq k} \text{Exc}_i::\text{instance}(\text{exc}) = \text{TRUE} ) \land \\
\bigwedge_{1\leq i\leq k} ( \neg \psi_{Exc_i} \rightarrow \text{Exc}_i::\text{instance}(\text{exc}) = \text{FALSE} ) \land \\
\bigwedge_{1\leq i\leq k} ( \text{Exc}_i::\text{instance}(\text{exc}) = \text{TRUE} \rightarrow \phi_{Exc_i} )
\]
A "new" method contract rule

Then the contract that should be applied is

**Precondition**

\[ \bigwedge_{k < i \leq n} \neg \psi_{\text{Exc}_i} \]

**Postcondition**

\[
\begin{align*}
( \text{exc} = \text{null} \rightarrow \phi_N ) & \land \\
\text{exc} \neq \text{null} \rightarrow \\
( \bigvee_{1 \leq i \leq k} \text{Exc}_i::\text{instance}(	ext{exc}) = \text{TRUE} ) & \land \\
\bigwedge_{1 \leq i \leq k} ( \neg \psi_{\text{Exc}_i} \rightarrow \text{Exc}_i::\text{instance}(	ext{exc}) = \text{FALSE} ) & \land \\
\bigwedge_{1 \leq i \leq k} ( \text{Exc}_i::\text{instance}(	ext{exc}) = \text{TRUE} \rightarrow \phi_{\text{Exc}_i} )
\end{align*}
\]
Then the contract that should be applied is

**Precondition**

\[ \bigwedge_{k<i \leq n} !\psi_{\text{Exc}_i} \]

**Postcondition**

\[
( \text{exc} = \text{null} \rightarrow \phi_N ) & \\
\text{exc} \neq \text{null} \rightarrow \\
( \\
\bigvee_{1 \leq i \leq k} \text{Exc}_i::\text{instance}(\text{exc}) = \text{TRUE} ) & \\
\bigwedge_{1 \leq i \leq k} ( !\psi_{\text{Exc}_i} \rightarrow \text{Exc}_i::\text{instance}(\text{exc}) = \text{FALSE} ) & \\
\bigwedge_{1 \leq i \leq k} ( \text{Exc}_i::\text{instance}(\text{exc}) = \text{TRUE} \rightarrow \phi_{\text{Exc}_i} )
\]
Demo

Proving the contract of a simple method
containsNullElements(java.util.List)
Conclusion

- Method contracts are capable of specifying library behaviour
- For interfaces: use of model functions necessary
- Need for thinking about the method contract rule
Conclusion

- Method contracts are capable of specifying library behaviour
- For interfaces: use of model functions necessary
- Need for thinking about the method contract rule
Conclusion

- Method contracts are capable of specifying library behaviour
- For interfaces: use of model functions necessary
- Need for thinking about the method contract rule
Tanks for your attention
Combining contracts

Assume 2 contracts given for one method
Let $\phi_1$ be the precondition of the first and $\phi_2$ the precondition of the second
Let $\psi_1$ be the postcondition of the first and $\psi_2$ the postcondition of the second
Let $M_1$ be the modifier set of the first and $M_2$ the modifier set of the second

Then a valid contract for the method is

**Precondition**
$\phi_1 \mid \phi_2$

**Postcondition**
$(\phi_1@\text{pre} \rightarrow \psi_1) \& (\phi_2@\text{pre} \rightarrow \psi_2)$

**Modifies**
$M_1 \cup M_2$
Behavioral subtyping

```
AbstractCollection
{abstract}
+remove(o)
<<interface>>
Collection
+remove(o)

AbstractList
{abstract}
<<interface>>
List
+remove(o)

LinkedList
ArrayList
```
Behavioral subtyping

AbstractCollection
{abstract}

One occurrence of Object o will be removed

AbstractList
{abstract}

Collection

List

<<interface>>
+remove(o)

<<interface>>
+remove(o)

LinkedList

ArrayList

implements

implements
Behavioral subtyping

AbstractCollection
{abstract}

+remove(o)
<<interface>>

Collection

+remove(o)

List

AbstractList
{abstract}

First occurrence of Object o will removed

LinkedList

ArrayList

<<interface>>

Behavioral subtyping

AbstractCollection
{abstract}
+remove(o)

<<interface>>
Collection
+remove(o)

<<interface>>
List
+remove(o)

AbstractList
{abstract}

implements
AbstractCollection
{abstract}
+remove(o)

<<interface>>
Collection
+remove(o)

<<interface>>
List
+remove(o)

LinkedList
ArrayList

implements
List
+remove(o)

implements
Collection
+remove(o)