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

Outline

Outline

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

Outline

Outline

Outline

Motivation

Problem

1. *No sources of the JDK library available in KeY*
   $\Rightarrow$ *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*
   $\Rightarrow$ *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*
   $\Rightarrow$ *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*
   $\Rightarrow$ *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*
   $\Rightarrow$ *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

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

```
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

```
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

Exceptional case

Precondition

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

Postcondition

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

    NullPointerException::instance(exc) = TRUE &
    ( src != null & dest != null ) ->
      ( NullPointerException::instance(exc) = FALSE ) &
    NullPointerException::instance(exc) = TRUE ->
      ( dest = null | \forall int i; dest[i] = dest[i]@pre )

)
```

Specification by example
Exceptional case

Precondition

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

Postcondition

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

    NullPointerException::instance(exc) = TRUE &
    ( src != null & dest != null ) ->
      ( NullPointerException::instance(exc) = FALSE ) &
    NullPointerException::instance(exc) = TRUE ->
      ( dest = null | \forall int i; dest[i] = dest[i]@pre )

)
```

Specification by example
Exceptional case

Precondition

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

Postcondition

```
exc = null ->
\forall int i; ( (0 <= i & i < src.length) -> dest[i] = src[i] )
&
exc != null ->
(
    NullPointerException::instance(exc) = TRUE &
    ( src != null & dest != null ) ->
      ( NullPointerException::instance(exc) = FALSE ) &
    NullPointerException::instance(exc) = TRUE ->
      ( dest = null | \forall int i; dest[i] = dest[i]@pre )

)
```

Specification by example

Exceptional case

Precondition

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

Postcondition

```
exc = null ->
\forall int i; ( (0 <= i & i < src.length) -> dest[i] = src[i] )
&
exc != null ->
(
    NullPointerException::instance(exc) = TRUE &
    ( src != null & dest != null ) ->
      ( NullPointerException::instance(exc) = FALSE ) &
    NullPointerException::instance(exc) = TRUE ->
      ( dest = null | \forall int i; dest[i] = dest[i]@pre )

)
```

Specification by example

Exceptional case

Precondition

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

Postcondition

```
exc = null ->
\forall int i; ( (0 <= i & i < src.length) -> dest[i] = src[i] )
&
exc != null ->
(
    NullPointerException::instance(exc) = TRUE &
    ( src != null & dest != null ) ->
      ( NullPointerException::instance(exc) = FALSE ) &
    NullPointerException::instance(exc) = TRUE ->
      ( dest = null | \forall int i; dest[i] = dest[i]@pre )

)
```

Specification by example

Exceptional case

Precondition

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

Postcondition

```
exc = null ->
\forall int i; ( (0 <= i & i < src.length) -> dest[i] = src[i] )
&
exc != null ->
(
    NullPointerException::instance(exc) = TRUE &
    ( src != null & dest != null ) ->
      ( NullPointerException::instance(exc) = FALSE ) &
    NullPointerException::instance(exc) = TRUE ->
      ( dest = null | \forall int i; dest[i] = dest[i]@pre )

)
```

General Concept for specifying methods

Precondition

Nearly all the time "`true`"

Postcondition

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

$$
\begin{array}{l}
\texttt{(exc = null -> } \phi_N \texttt{) \&} \\
\texttt{exc != null ->} \\
\texttt{(} \\
\quad \texttt{(} \bigvee_i Exc_i\texttt{::instance(exc) = TRUE) \&} \\
\quad \bigwedge_i\texttt{(!}\psi_{Exc_i} \texttt{ -> } Exc_i\texttt{::instance(exc) = FALSE) \&} \\
\quad \bigwedge_i\texttt{(} Exc_i\texttt{::instance(exc) = TRUE -> } \phi_{Exc_i} \texttt{)} \\
\texttt{)}
\end{array}
$$

General Concept for specifying methods

Precondition

Nearly all the time "`true`"

Postcondition

Let ϕ_N be the postcondition for normal behaviour

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

Let ϕ_{Exc_i} be the postcondition that holds after Exc_i has been thrown

Then the postcondition should look like this:

(exc = null -> ϕ_N) &
exc != null ->
(
    ($\bigvee_i$ Exc_i::instance(exc) = TRUE) &
    $\bigwedge_i$(!ψ_{Exc_i} -> Exc_i::instance(exc) = FALSE) &
    $\bigwedge_i$(Exc_i::instance(exc) = TRUE -> ϕ_{Exc_i})
)

General Concept for specifying methods

Precondition

Nearly all the time "`true`"

Postcondition

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

$$
\begin{array}{l}
\texttt{(exc = null ->}\ \phi_N\ \texttt{) \&} \\
\texttt{exc != null ->} \\
\texttt{(} \\
\quad \texttt{(}\ \bigvee_i Exc_i\texttt{::instance(exc) = TRUE) \&} \\
\quad \bigwedge_i \texttt{(!}\psi_{Exc_i}\ \texttt{->}\ Exc_i\texttt{::instance(exc) = FALSE) \&} \\
\quad \bigwedge_i \texttt{(}\ Exc_i\texttt{::instance(exc) = TRUE ->}\ \phi_{Exc_i}\ \texttt{)} \\
\texttt{)}
\end{array}
$$

General Concept for specifying methods

Precondition

Nearly all the time "`true`"

Postcondition

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

$$
\begin{array}{l}
\texttt{(exc = null -> } \phi_N \texttt{) \&} \\
\texttt{exc != null ->} \\
\texttt{(} \\
\quad \texttt{(} \bigvee_i Exc_i\texttt{::instance(exc) = TRUE) \&} \\
\quad \bigwedge_i \texttt{(!}\psi_{Exc_i} \texttt{ -> } Exc_i\texttt{::instance(exc) = FALSE) \&} \\
\quad \bigwedge_i \texttt{(} Exc_i\texttt{::instance(exc) = TRUE -> } \phi_{Exc_i} \texttt{)} \\
\texttt{)}
\end{array}
$$

General Concept for specifying methods

Precondition

Nearly all the time "`true`"

Postcondition

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

(exc = null -> ϕ_N) &
exc != null ->
(
    ($\bigvee_i$ Exc_i::instance(exc) = TRUE) &
    $\bigwedge_i$(!ψ_{Exc_i} -> Exc_i::instance(exc) = FALSE) &
    $\bigwedge_i$(Exc_i::instance(exc) = TRUE -> ϕ_{Exc_i})
)

General Concept for specifying methods

Precondition

Nearly all the time "`true`"

Postcondition

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

$$
\begin{array}{l}
\texttt{(exc = null ->}\ \phi_N\ \texttt{) \&} \\
\texttt{exc != null ->} \\
\texttt{(} \\
\quad \texttt{(}\ \bigvee_i Exc_i\texttt{::instance(exc) = TRUE) \&} \\
\quad \bigwedge_i \texttt{(!}\psi_{Exc_i}\ \texttt{->}\ Exc_i\texttt{::instance(exc) = FALSE) \&} \\
\quad \bigwedge_i \texttt{(}\ Exc_i\texttt{::instance(exc) = TRUE ->}\ \phi_{Exc_i}\ \texttt{)} \\
\texttt{)}
\end{array}
$$

General Concept for specifying methods

Precondition

Nearly all the time "`true`"

Postcondition

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

$(\texttt{ exc = null -> } \phi_N \texttt{) \&}$
$\texttt{exc != null ->}$
$($
$\quad (\bigvee_i Exc_i\texttt{::instance(exc) = TRUE) \&}$
$\quad \bigwedge_i \texttt{(!}\psi_{Exc_i} \texttt{ -> } Exc_i\texttt{::instance(exc) = FALSE) \&}$
$\quad \bigwedge_i (Exc_i\texttt{::instance(exc) = TRUE -> } \phi_{Exc_i})$
$)$

General Concept for specifying methods

Precondition

Nearly all the time "`true`"

Postcondition

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

$$
\begin{array}{l}
\texttt{(exc = null ->}\ \phi_N\ \texttt{) \&} \\
\texttt{exc != null ->} \\
\texttt{(} \\
\quad \texttt{(}\ \bigvee_i Exc_i\texttt{::instance(exc) = TRUE) \&} \\
\quad \bigwedge_i \texttt{(!}\psi_{Exc_i}\ \texttt{->}\ Exc_i\texttt{::instance(exc) = FALSE) \&} \\
\quad \bigwedge_i \texttt{(}\ Exc_i\texttt{::instance(exc) = TRUE ->}\ \phi_{Exc_i}\ \texttt{)} \\
\texttt{)}
\end{array}
$$

Interface specification

Model functions

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

Interface specification
Model functions

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

Interface specification

Example

Method to be specified

`s = myList.size()@java.util.List;`

with `s` $\sqsubseteq$ `jint` and `myList` $\sqsubseteq$ `java.util.List`

Precondition

`true`

Postcondition

```
\if (_size(myList) <= java.lang.Integer.MAX_VALUE)
\then (s = _size(myList))
\else (s = java.lang.Integer.MAX_VALUE)
```

Modifies

`s`

Interface specification

Example

Method to be specified

`s = myList.size()@java.util.List;`

with `s` $\sqsubseteq$ `jint` and `myList` $\sqsubseteq$ `java.util.List`

Precondition

`true`

Postcondition

```
\if (_size(myList) <= java.lang.Integer.MAX_VALUE)
\then (s = _size(myList))
\else (s = java.lang.Integer.MAX_VALUE)
```

Modifies

`s`

Interface specification
Example

Method to be specified

`s = myList.size()@java.util.List;`

with `s` $\sqsubseteq$ `jint` and `myList` $\sqsubseteq$ `java.util.List`

Precondition

`true`

Postcondition

```
\if (_size(myList) <= java.lang.Integer.MAX_VALUE)
\then (s = _size(myList))
\else (s = java.lang.Integer.MAX_VALUE)
```

Modifies

s

Interface specification

Example

Method to be specified

`s = myList.size()@java.util.List;`

with `s` $\sqsubseteq$ `jint` and `myList` $\sqsubseteq$ `java.util.List`

Precondition

```
true
```

Postcondition

```
\if (_size(myList) <= java.lang.Integer.MAX_VALUE)
\then (s = _size(myList))
\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 <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

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 `<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

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 <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

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 <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

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 <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

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 `<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

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 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 $\texttt{S} \sqsubseteq \texttt{T}$
Let $\texttt{obj} \in \texttt{S}$

Method call vs. method body statement

- Method call

  $$\texttt{obj.m(}\mathit{params}\texttt{)}$$

  will be expanded to
- Method body statement

  $$\texttt{obj.m(}\mathit{params}\texttt{)@T}$$

  where `T` specifies where to find the implementation of `m(`*params*`)`

Using specifications

Let `S`, `T` be types with `S` $\sqsubseteq$ `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*`)`

Using specifications

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

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*`)`

Using specifications

Behavioral subtyping

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

Let `obj.m(`*params*`)@T` be a method body statement with `obj` $\in$ `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 `obj.m(`*params*`)@T` be a method body statement with `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 `obj.m(`*params*`)@T` be a method body statement with `obj` $\underline{\sqsubseteq}$ `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 $\texttt{S} \sqsubseteq \texttt{T}$
Let `obj.m(`*params*`)@T` be a method body statement with $\texttt{obj} \sqsubseteq \texttt{S}$

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
  $\Rightarrow$ 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
  $\Rightarrow$ 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
  $\Rightarrow$ 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 ϕ_N be the postcondition for normal behaviour

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

Let ϕ_{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 ϕ_N be the postcondition for normal behaviour
Let ψ_{Exc_i} $(1 \leq i \leq n, n \in N)$ be the condition where the exception Exc_i is thrown
Let ϕ_{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 ϕ_N be the postcondition for normal behaviour
Let ψ_{Exc_i} $(1 \leq i \leq n, n \in N)$ be the condition where the exception Exc_i is thrown
Let ϕ_{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 ϕ_N be the postcondition for normal behaviour
Let ψ_{Exc_i} $(1 \leq i \leq n, n \in N)$ be the condition where the exception Exc_i is thrown
Let ϕ_{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

$\bigwedge_{k<i\leq n}$ `!`ψ_{Exc_i}

Postcondition

(`exc = null ->` ϕ_N) `&`
`exc != null ->`
`(`
    ($\bigvee_{1\leq i\leq k}$ Exc_i`::instance(exc) = TRUE`) `&`
    $\bigwedge_{1\leq i\leq k}$(`!`$\psi_{Exc_i}$ `->` Exc_i`::instance(exc) = FALSE`) `&`
    $\bigwedge_{1\leq i\leq k}$(Exc_i`::instance(exc) = TRUE ->` ϕ_{Exc_i})
`)`

A "new" method contract rule

Then the contract that should be applied is

Precondition

$\bigwedge_{k<i\leq n}$ $!\psi_{Exc_i}$

Postcondition

(exc = null -> ϕ_N) &
exc != null ->
(
    ($\bigvee_{1\leq i\leq k}$ Exc_i::instance(exc) = TRUE) &
    $\bigwedge_{1\leq i\leq k}$($!\psi_{Exc_i}$ -> Exc_i::instance(exc) = FALSE) &
    $\bigwedge_{1\leq i\leq k}$(Exc_i::instance(exc) = TRUE -> ϕ_{Exc_i})
)

A "new" method contract rule

Then the contract that should be applied is

Precondition

$\bigwedge_{k<i\leq n} \; !\psi_{Exc_i}$

Postcondition

```
( exc = null -> φ_N ) &
exc != null ->
(
    ( ⋁_{1≤i≤k} Exc_i::instance(exc) = TRUE ) &
    ⋀_{1≤i≤k}( !ψ_{Exc_i} -> Exc_i::instance(exc) = FALSE ) &
    ⋀_{1≤i≤k}( Exc_i::instance(exc) = TRUE -> φ_{Exc_i} )
)
```

A "new" method contract rule

Then the contract that should be applied is

Precondition

$\bigwedge_{k<i\leq n}$ $!\psi_{Exc_i}$

Postcondition

(exc = null -> ϕ_N) &
exc != null ->
(
    ($\bigvee_{1\leq i\leq k}$ Exc_i::instance(exc) = TRUE) &
    $\bigwedge_{1\leq i\leq k}$($!\psi_{Exc_i}$ -> Exc_i::instance(exc) = FALSE) &
    $\bigwedge_{1\leq i\leq k}$(Exc_i::instance(exc) = TRUE -> ϕ_{Exc_i})
)

A "new" method contract rule

Then the contract that should be applied is

Precondition

$\bigwedge_{k<i\leq n} \ !\psi_{Exc_i}$

Postcondition

(exc = null -> ϕ_N) &
exc != null ->
(
    ($\bigvee_{1\leq i\leq k}$ Exc_i::instance(exc) = TRUE) &
    $\bigwedge_{1\leq i\leq k}$($!\psi_{Exc_i}$ -> Exc_i::instance(exc) = FALSE) &
    $\bigwedge_{1\leq i\leq k}$(Exc_i::instance(exc) = TRUE -> ϕ_{Exc_i})
)

A "new" method contract rule

Then the contract that should be applied is

Precondition

$\bigwedge_{k<i\leq n}$ $!\psi_{Exc_i}$

Postcondition

(exc = null -> ϕ_N) &
exc != null ->
(
    ($\bigvee_{1\leq i\leq k}$ Exc_i::instance(exc) = TRUE) &
    $\bigwedge_{1\leq i\leq k}$($!\psi_{Exc_i}$ -> Exc_i::instance(exc) = FALSE) &
    $\bigwedge_{1\leq i\leq k}$(Exc_i::instance(exc) = TRUE -> ϕ_{Exc_i})
)

Demo

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 ϕ_1 be the precondition of the first and ϕ_2 the precondition of the second
Let ψ_1 be the postcondition of the first and ψ_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

ϕ_1 `|` ϕ_2

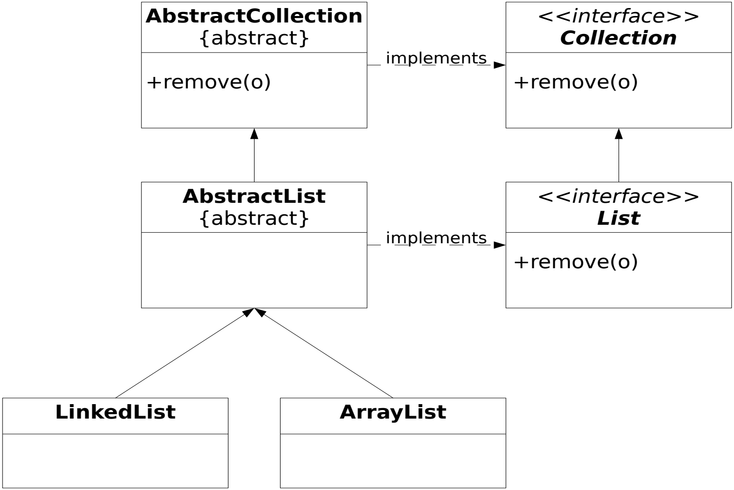
Postcondition

`(` ϕ_1`@pre ->` ψ_1 `) & (` ϕ_2`@pre ->` ψ_2 `)`

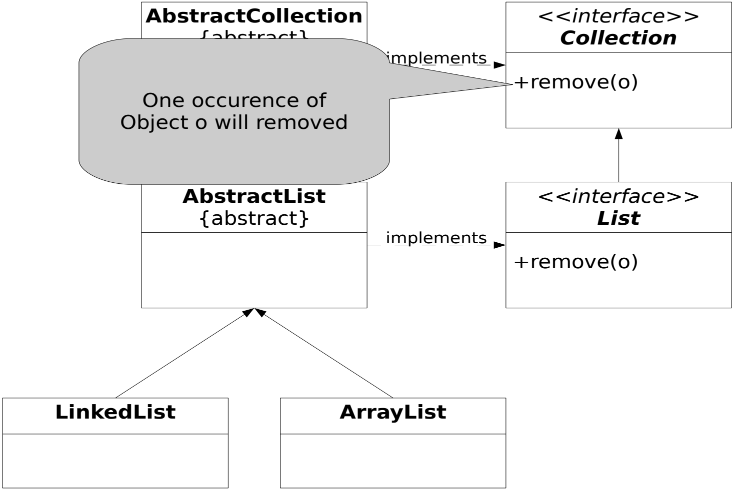
Modifies

$M_1 \cup M_2$

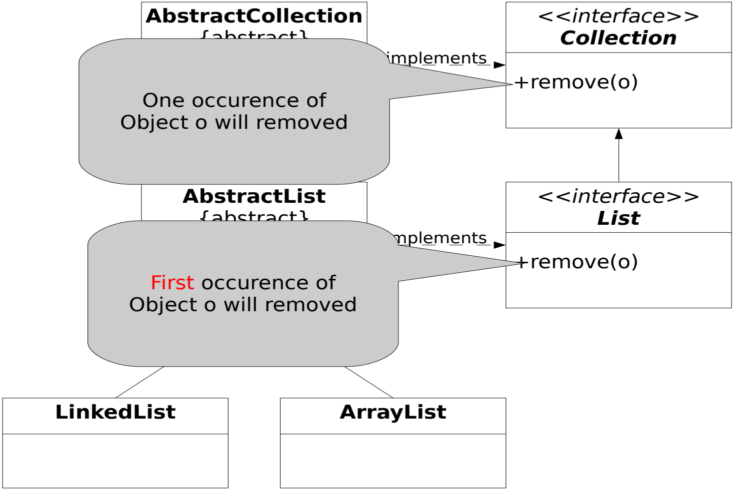
Behavioral subtyping



Behavioral subtyping



Behavioral subtyping



Behavioral subtyping