Specifying Contracts for Methods

Module III Lecture 4

Assume-quarantee specifications as program annotations

Recall some definitions

- *Formal Specification* - **using mathematical notation to give a precise description of what a program should do**

- *Formal Verification* - **using precise rules to mathematically prove that a program satisfies a formal specification**

- *Formal Development (Refinement)* - **developing programs in a way that ensures mathematically they meet their formal specifications**

Introduction to Hoare style specifications

- Verification of programs is based on formal specifications and on related verification method.

  We will use Floyd-Hoare logic (FHL)

- Proof systems of the FHL style depend on particular programming language with its syntax and semantics

- In this lecture

  - we study the specification of deterministic sequential *while*-programs
  - and extend this to JML specifications of OO programs, namely, to method contracts.

Programs as state transition systems

- Programs are structured specifications of state transition systems.
- Programming language defines constructs for specifying single transitions and transition compositions.
- State components specified using datatypes are referred in conditions of command constructs like *if*-, *while-, for-, case*-command etc.

Some notations

- Imperative programs are built out of *commands* like assignment, *if*-, *while*-, *for*-, *case*, etc
- Formally, the terms '*program*' and '*command*' are synonymous.
- '*Program*' will only be used for commands representing complete algorithm.
- The '*assertion*' is used for conditions on program variables that occur in correctness specifications.

Imperative programs - state

- Executing an imperative program has the effect of changing the *state* i.e. the values of program variables,

  but

  they may have states consisting of other things than the values of variables (e.g. I/O ports).

Imperative programs - execution

- To use an imperative program (or method)
  - first establish a state,
    i.e. set some variables to have values of interest
  - then execute the program,
    (to transform the initial state into a final one)
  - inspect the values of variables in the final state to get the result.

Simple *while*-language

	% Expressions
• E ::= N|V|$E1$+$E2$|$E1$-$E2$|$E1$×$E2$| …	% Arithmetic
• B ::= T|F|$E1$=$E2$|$E1$≤$E2$| …	% Logic
• C ::=	%Commands:
SKIP	% empty command (place holder)
| V := E	% assignment
| V(E1) := E2	% array assignment
| $C1$; $C2$	% sequential execution
| IF B THEN $C1$ ELSE $C2$	% conditional execution
| BEGIN VAR $V1$;…;VAR Vn; C END	% block command (var. scoping)
| WHILE B DO C	% *while* - loop
| FOR V := $E1$ UNTIL $E2$ DO C	% *for* - loop

Terminology and notations

- *Variable*
  - `V1, V2, ..., Vn`
- *Program state* - valuation of program (and control) variables
- *Command* - gives a rule how the program state changes
  - `C1, C2, ... , Cn`
- *Program* - command that includes all the commands in the algorithm
  - `C`
- *Expression*
  - Arithmetic expression gives a value: `E1, E2, ... , En`
  - Boolean expression gives a *truth*-value: `B1, B2, ... , Bn`
- *Assertion* – logical expression on program variables in the pre- and postconditions of the specification, also in invariants
  - `S1, S2, ... , Sn`

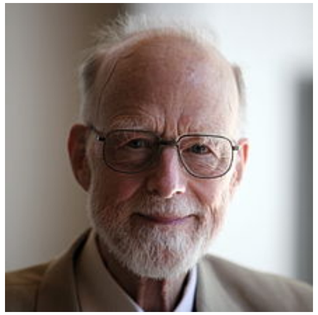
Formal specification

- Describes the intended behaviour of the program
- Specifies *what* the program must do
- Has well-defined *synax* and *semantics* that helps avoiding *ambiguous* and *controversial* specifications
- Can be used to prove the *correctness of the program*
- Can be used to generate *tests* and *counterexamples*

We will use formalism that is based on FHL and predicate calculus

Hoare's notation

Sir Tony Hoare



- **C.A.R. Hoare introduced the following notation called a** *partial correctness specification* **for specifying what a program does:**

  $$\{P\}\ C\ \{Q\}$$

  where:

  - C **is a program from the programming language whose programs are being specified**
  - P **and** Q **are conditions on the program variables used in** C

Partial Correctness

- **An expression** $\{P\}\ C\ \{Q\}$ **is called a** *partial correctness specification*
  - P **is called its** *precondition*
  - Q **its** *postcondition*
- $\{P\}\ C\ \{Q\}$ **is true if**
  - **whenever** C **is executed in a state satisfying** P
  - **and** *if* **the execution of** C **terminates**
  - **then the state in which** C**'s execution terminates satisfies** Q

Examples

- $\{\mathtt{X} = 1\}$ `Y:=X` $\{\mathtt{Y} = 1\}$
  - **This says that *if* the command `Y:=X` is executed in a state satisfying the condition $\mathtt{X} = 1$**
  - **i.e. a state in which the value of `X` is 1**
  - ***then*, if the execution terminates (which it does)**
  - **then the condition $\mathtt{Y} = 1$ will hold**
  - **Clearly this specification is true**

Examples

- $\{\mathtt{X} = 1\}$ `Y:=X` $\{\mathtt{Y} = 2\}$
  - **This says that if the execution of `Y:=X` terminates when started in a state satisfying $\mathtt{X} = 1$**
  - **then $\mathtt{Y} = 2$ will hold**
  - **This is clearly false**
- $\{\mathtt{X} = 1\}$ `WHILE T DO SKIP` $\{\mathtt{Y} = 2\}$
  - **This specification is true!**

Total correctness

- **A stronger kind of specification is a** *total correctness specification*
  - **There is no standard notation for such specifications**
  - **We shall use** $[P]\ C\ [Q]$
- **A total correctness specification** $[P]\ C\ [Q]$ **is true if and only if**
  - **Whenever** C **is executed in a state satisfying** P**, then the execution of** C **terminates**
  - **After** C **terminates** Q **holds**

Example

- $[\mathtt{X} = 1]$ `Y:=X; WHILE T DO SKIP` $[\mathtt{Y} = 1]$
  - **This says that the execution of** `Y:=X;WHILE T DO SKIP` **terminates when started in a state satisfying** $\mathtt{X} = 1$
  - **after which** $\mathtt{Y} = 1$ **will hold**
  - **This is clearly false**

Total correctness

- **Informally:**

  $$\textit{Total correctness} = \textit{Termination} + \textit{Partial correctness}$$

- **Total correctness is the ultimate goal**
  - **usually easier to show partial correctness and termination separately**

Total correctness

- **Termination is usually straightforward to show, but there are examples where it is not: no one knows whether the program below terminates for all values of** X

```
WHILE X>1 DO
  IF ODD(X) THEN X := (3×X)+1 ELSE X := X DIV 2
```

  - The expression X DIV 2 evaluates to the result of rounding down X/2 to a whole number

- **Exercise: Write a specification which is true if and only if the program above terminates**

Examples

- $\{\mathtt{T}\}\ C\ \{Q\}$
  - **This says that whenever C halts, Q holds**
- $\{P\}\ C\ \{\mathtt{T}\}$
  - **This specification is true for every condition P and every command C**
  - **Because $\mathtt{T}$ is always true**

Examples

- $[P]\ C\ [\mathtt{T}]$
  - **This says that C terminates if initially P holds**
  - **It says nothing about the final state**
- $[\mathtt{T}]\ C\ [P]$
  - **This says that C always terminates and ends in a state where P holds**

A more complicated example

```
{T}
 BEGIN
   R:=X;
   Q:=0;
   WHILE Y≤R DO
     BEGIN R:=R-Y; Q:=Q+1 END
 END
{R < Y ∧ X = R + (Y × Q)}
```

$$\left.\begin{array}{l}\texttt{BEGIN}\\ \quad\texttt{R:=X;}\\ \quad\texttt{Q:=0;}\\ \quad\texttt{WHILE Y}\leq\texttt{R DO}\\ \qquad\texttt{BEGIN R:=R-Y; Q:=Q+1 END}\\ \texttt{END}\end{array}\right\} C$$

- **This is $\{\mathtt{T}\}\ C\ \{\mathtt{R} < \mathtt{Y}\ \wedge\ \mathtt{X} = \mathtt{R} + (\mathtt{Y} \times \mathtt{Q})\}$**
  - **where C is the command indicated by the braces above**
  - **The specification is true if whenever the execution of C halts, then `Q` is quotient and `R` is the remainder resulting from dividing `Y` into `X`**
  - **It is true (even if `X` is initially negative!)**
  - **In this example a program variable `Q` is used. This should not be confused with the Q used in previous examples to range over postconditions**

Annotate First

- **It is helpful to think up these statements, before you start the proof and annotate the program with them**
  - **The information is then available when you need it in the proof**
  - **This can help avoid you being bogged down in details**
  - **The annotation should be true whenever control reaches that point** in program!

Annotation example

- **Example, the following program could be annotated at the points indicated.**

```
{T}
 BEGIN
   R:=X;
   Q:=0; {R=X ∧ Q=0} ⟵P₁
   WHILE Y≤R DO {X = R+Y×Q} ⟵P₂
     BEGIN R:=R-Y; Q:=Q+1 END
 END
{X = R+Y×Q ∧ R<Y}
```

WHILE annotation

- **A correctly annotated total correctness specification of a `WHILE`-command thus has the form**

$$[P]\ \mathtt{WHILE}\ S\ \mathtt{DO}\ \{R\}[E]\ C\ [Q]$$

where R is the invariant and E the variant

- **Note that the variant is intended to be a non-negative expression that decreases each time around the `WHILE` loop**

- **The other annotations, which are enclosed in curly brackets, are meant to be conditions that are true whenever control reaches them**

Some exercises

- **When is $[\mathtt{T}]\ C\ [\mathtt{T}]$ true?**

- **Write a partial correctness specification which is true if and only if the command C has the effect of multiplying the values of $\mathtt{X}$ and $\mathtt{Y}$ and storing the result in $\mathtt{X}$**

- **Write a specification which is true if the execution of C always halts when execution is started in a state satisfying P**

From imperative to OOP specifications

- Imperative part in OOP concerns methods
- Method specifications in JML:

```
method-specification ::= specification | extending-specification
extending-specification ::= also specification
specification ::= spec-case-seq [ redundant-spec ]| redundant-spec
spec-case-seq ::= spec-case [ also spec-case ] . . .
```

- Method-specification can include any number of spec-cases, joined by the keyword `also`, as well as a `redundant-spec`
- Each of the spec-cases specifies a behavior that must be satisfied by a correct implementation of the method
- Whenever a call to the specified method or constructor satisfies the *precondition* of one of its spec-cases, the rest of the clauses in that spec-case must also be satisfied by the implementation

Method specification

- The spec-cases in a method-specification can have several forms:

```
spec-case ::= lightweight-spec-case | heavyweight-spec-case| model-
program
```

- heavyweight specification cases, which start with one of the keywords:

```
behavior, normal_behavior or exceptional_behavior
```

- lightweight specification cases do not contain these `behavior` keywords

Access Control in specification Cases

- Heavyweight specification cases may be declared with an explicit *access modifier*:

```
privacy ::= public | protected | private
```

- The access modifier of the case cannot allow more access than the method being specified.
- Example:
  - a public method may have a `private` behavior specification,
  - but a `private` method may not have a `public` specification.
- A heavyweight specification case without an explicit access modifier is considered to have default ***package*** access.

Lightweight specification cases

- Do not specify an access modifier,
- their access modifier is implicitly the same as that of method being specified.
- For example,
  - a lightweight specification of a *public* method has *public* access, implicitly,
  - a lightweight specification of a *private* method has *private* access, implicitly.
- This is a different default than that for heavyweight specifications, where an omitted access modifier always means *package* access.
- A lightweight specification case can be understood as *syntactic sugar* for a *behavior specification case*

Lightweight Specification Cases: Syntax

```
lightweight-spec-case ::= generic-spec-case

generic-spec-case ::= [spec-var-decls] spec-header [generic-spec-body] |
      [spec-var-decls] generic-spec-body

generic-spec-body ::= simple-spec-body |{|generic-spec-case-seq |}

generic-spec-case-seq ::= generic-spec-case [also generic-spec-case] ...

spec-header ::= requires-clause [requires-clause] ...

simple-spec-body ::= simple-spec-body-clause [simple-spec-body-clause] ...

simple-spec-body-clause ::= diverges-clause | assignable-clause |
      accessible-clause| captures-clause | callable-clause | when-clause |
      working-space-clause| duration-clause | ensures-clause | signals-only-
      clause | signals-clause | measured-clause
```

Example: Lightweight *vs* Heavyweight Spec. Cases

Lightweight	Corresponding Heavyweight (with explicit defaults)

`package org.jmlspecs.samples.jmlrefman;`
`public abstract class Lightweight {`
`protected boolean P, Q, R;`
`protected int X;`
	`package org.jmlspecs.samples.jmlrefman;`
	`public abstract class Heavyweight {`
	`protected boolean P, Q, R;`
	`protected int X;`

`/*@ requires P;`	`/*@ protected behavior` `@ requires P;`
`@ assignable X;`	`@ diverges false;` `@ assignable X;`

	`@ when \not_specified;`
	`@ working_space \not_specified;`
	`@ duration \not_specified;`

`@ ensures Q;`	`@ ensures Q;` `@ signals_only Exception;`

`@ signals (Exception) R;`
`@*/`
`protected abstract int m() throws Exception;`
`}`
	`@ signals (Exception) R;`
	`@*/`
	`protected abstract int m() throws Exception;`
	`}`

Example explanation:

- the default for an ***omitted*** clause in a lightweight specification is `\not_specified` for all clauses, except `diverges`, which has a default of `false`, and `signals`.
- The default for an omitted `signals` clause is to only permit the exceptions declared in the method's header to be thrown

Heavyweight specification Cases

- There are three kinds of heavyweight specification cases, called behavior, normal behavior, and exceptional behavior specification cases, beginning (after an optional privacy modifier) with the one of the keywords `behavior`, `normal_behavior`, or `exceptional_behavior`, respectively.

```
heavyweight-spec-case ::=  behavior-spec-case
                         | exceptional-behavior-spec-case
                         | normal-behavior-spec-case
```

- Like lightweight specification cases, normal behavior and exceptional behavior specification cases can be understood as ***syntactic sugar for special kinds of behavior specification*** cases

Semantics of flat behavior specification cases

- Behavior-spec-case consists of any number of following clauses:
  - `requires`
  - `diverges`
  - `measured_by`
  - `assignable`
  - `accessible`
  - `callable`
  - `when`
  - `ensures`
  - `duration`
  - `working_space`
  - `signals_only`
  - `signals`
- There are defaults that allow any of them to be omitted.

Example: non-helper method *m* specification case

```
behavior
    forall T1 x1; … forall Tn xn;    % For every possible value of the variables following holds
    old U1 y1 = F1; … old Uk yk = Fk;  % In augmented pre-state,  the pre-values are made explicit
    requires P;                          % Preconditions and all invariants should hold in the pre-state of the call
    measured_by Mbe if Mbp;   % Mbe is variant of recursive call if Mbp, is true in the augmented pre-state
    diverges D;      % D becomes true if recursive call never terminates, otherwise terminates in post-state s.t.
    when W;                              % executes as long the W holds
    accessible R;                        % Locations that are readable from
    assignable A;                        % Locations which can be assigned to during method execution
    callable p1(…), …, pl(…);            % Methods and constructors called during method execution
    captures Z;          % param.-s of reference type assigned to fields of some object or to array elements
    ensures Q;                           % guarantee if method terminates normally
    signals_only E1,…,Eo;              % throwing an exception of type Ea it must be one of E1,…,Eo
    signals (E e) S;    % exceptional cond R must hold in post-state, augmented by a binding from variable e
    working_space Wse if Wsp;      % restriction placed on the maximum space the method call may have
    duration De if Dp;          % if Dp=true in the  pre-state, then the method execution takes De timeunits
```

Method Specification Clauses (1): `variable declarations`

- Specification Variable Declarations clause

```
spec-var-decls ::= forall-var-decls [old-var-decls] | old-var-decls
forall-var-decls ::= forall-var-declarator [ forall-var-declarator ] . . .
forall-var-declarator ::= forall [bound-var-modiers]
                                        type-spec quantied-var-declarator ;
old-var-decls ::= old-var-declarator [ old-var-declarator ] . . .
old-var-declarator ::= old [ bound-var-modiers ]
                                        type-spec spec-variable-declarators ;
```

- `old-var-declarator` allows abbreviation within a specification case.
- The names defined in the `spec-variable-declarators` can be used throughout the spec case for the values of their initializers.
- The expressions are evaluated in the method's pre-state

Method Specification Clauses (2): `requires`

- A `requires` clause specifies a precondition of method or constructor

```
requires-clause ::= requires-keyword pred-or-not ;
      | requires-keyword \same ;
requires-keyword ::= requires | pre
      | requires_redundantly | pre_redundantly
pred-or-not ::= predicate | \not_specified
```

- The predicate in a `requires` can refer to any visible fields and to the parameters of the method
- Any number of `requires` clauses can be included a single specification case.
- Multiple `requires` clauses in a specification case means the same as a single requires clause whose precondition predicate is the conjunction of these precondition predicates in the given requires clauses.
- `\same` stands for the disjunction (with ||) of the preconditions in all spec cases from the current class together with the inherited spec cases defined in its supertypes.

Method Specification Clauses (3): `ensures` clauses

- Specifies a property that is guaranteed to hold at the end of the method (or constructor) invocation in the case that this method (or constructor) invocation returns without throwing an exception.

```
ensures-clause ::= ensures-keyword pred-or-not ;
ensures-keyword ::= ensures | post| ensures_redundantly | post_redundantly
```

- A predicate in an ensures clause can refer to
  - any visible fields,
  - the parameters of the method,
  - `\result` if the method is non-void, and
  - may contain expressions of the form `\old(E)`.
- Multiple `ensures` clauses in a specification case mean the same as a single ensures clause whose postcondition predicate is the conjunction of the postcondition predicates in the given ensures clauses.
- The default precondition for a lightweight specification case, is **`\not_specified`**.
- The default precondition for a heavyweight specification case is **`true`**

Summary

- We have given a notation for specifying
  - partial correctness of programs
  - total correctness of programs
- It is easy to write incorrect specifications
  - and we can prove the correctness of the incorrect programs
- It is recommended to use testing, simulation and formal verification hand in hand.