Specification Methods
Specifikationsmetodik
6559

Winter 2006

Specification Methods

- 3 credit course
- 09.01 – 20.02.2006 (13 lectures)
- exams on 20.03.2006 and 10.04.2006
- exercise sessions
  supervised by Andreas Enbacka
- literature:

  1. S.Schneider. "The B-Method. The Introduction"
  2. K.Lano, H.Haughton. "Specifications in B"
  3. Wordsworth. "Software Engineering with B"

- the course web page:
  www.abo.fi/~Linas.Laibinis/SpecMeth/index.html

The overview of the course

- we will see advantages of using (formal) specifications in software development
- we will learn an industry-strong specification method called "B" which has
  - a language
  - a tool called the B-Toolkit
- we will do a lot of exercises during this course using B-Toolkit

Introduction to B

- The B Method (or simply B) – a formal approach for industrial development of highly reliable software
- It covers the complete software lifecycle, from requirements (*specification), through design (*refinement) to implementation and code generation
- In this course we will be concerned with the use of B for specification

What do we mean by Formal Method?

- We mean the application of mathematics (set theory and logic) to specify, design and implement software in a such a way that the resulting code has been *proved to be consistent with the original specification*
- *In B, a specification is a mathematical (abstract) model of the required behaviour of the system*
- *B specifications are transformed into concrete implementations by a sequence of formally defined* refinement (i.e. design) steps

Connection with "conventional" methods

- In conventional methods the requirements are usually expressed informally, in either natural language, or using some structured notation (e.g. UML)
- Specifications are frequently expressed directly in programming code (in the form of comments or annotations)
- In contrast, using B, the specification is abstract description of the requirements, expressing what behaviour is required, rather than how to produce that behaviour

Traditional engineering disciplines

- In the traditional engineering disciplines, designs are based on a mathematical theory of the materials, components, structures to be used in the implementation of ,e.g., bridges, buildings, electronic circuits
- Testing consists of physical testing of implementation or its model
- This is succesful strategy because the domains can be described by continuous mathematics: if a model conforms for some specific test input, it will also conform for input that is 'less than' that input
- This strategy does not work for discrete valued domains

Testing of software

- Software executes over discrete domains, and testing usually consists of probing points within that space
- Thus testing only confirms conformance of behaviour (e.g. correctness of execution) at specific points. Testing is incapable, in general, to demonstrate conformance over the the complete application domain
- *Thus testing may confirm the presence of bugs, not their absence*

Advantages of formal software development

- We build a formal model using constructs that are defined by precise mathematical theories. These models capture the behaviour in a complete application domain

- As we develop our specifications into implementations, the formal method produces proof obligations that basically describe the complete set of tests. These tests confirm that the behaviours of the specification and the design are consistent

- *The formal proof validates behaviour in a complete domain, not simply at a single point*

Applications of formal methods

- Formal methods have been used in various mission critical applications, like train control systems and smart cards. In some countries the use of formal methods is mandatory for certain critical systems

- The most famous use of B is for the control system of the Meteor line (new driveless line of Paris metro) opened in October 1998. See
  www.siemens-ts.com/pagesUS/realisations/Paris.htm
  www.siemens-ts.com/pagesUS/produits/Meteor.htm
  The distributed control system handled the critical parts of the central control room, the equipment along the track, the onboard train control. The Meteor system was developed by Mantra Transport, now owned by Siemens

- For another application (smart cards by GEMPLUS) see
  www.gemplus.com/smart/r_d/trends/system_model/b_method/

Increase of formality

- The increase of the use of formality in software development has been continuous, from formal grammars to specify programming language syntax, to the semi-formal application of translator generators in compiler implementations
- High-level languages themselves are an instance of increased formality, over machine level (assembler) programming in this case
- Everywhere rigour and formality has been used, there has been an increase in the reliability of implementations
- *There is no reason to believe that this "progress" will not continue*

Course Objectives

- The objective of the course is to get you to think more carefully about the specification phase of software development
- The objective of the use of a formal method (B) is to encourage you to think more rigorously about specification in particular, and to extend that to other phases of software system development

Origins of B

The author of the B Method is Jean-Raymond Abrial. However, B is built on the foundations established by

- Tony Hoare, Edsgar Dijkstra: *Weakest preconditions*
- *Cliff Jones:* Pre- and Post- conditions, VDM
- Ralph Back, Carroll Morgan: *Refinement Calculus*
- *Jean-Raymond Abrial: Z specification notation and mathematical toolkit*

B Method and Toolkit

- As its name implies, B (the B Method) is a method, not simply a notation. B is supported by a toolkit, which should be regarded as essential requisite for using the method. There are two toolkits:
  - AtelierB (distributed by Steria):
    http://www.atelierb.societe.com
  - B-Toolkit (distributed by B-Core):
    http://www.b-core.com

We will be using the B-Toolkit.

The B-Toolkit

The B-Toolkit is a project management tool that provides the following facilities:

- syntax and type analysis of specifications
- animation of specifications
- support of design (refinement) transformations
- generation of proof obligations
- automatic and interactive proof
- code generation
- etc.

Abstract Machines

An unit of B specification is called an *abstract machine. It is a module that encapsulates*

- *state, consisting of a set of variables constrained by* invariant property
- *operations, that may change the state, while maintaining the invariant, and may return a sequence of results*

Object based

- Abstract machines are sometimes described as *object-based, rather than object-oriented*
- *An abstract machine can be compared with an object, an instance of a class*
- *A machine does not behave as a class, although it is possible to model a class*

A simple specification

- We will model a "one-account" bank into which we can put money and also take money out.
- In our model we will use a variable `account` whose value is a natural number, representing the amount in cents.
- Also, we have 3 operations:

  `FeedBank(amount)` – add `amount` to the bank

  `RobBank(amount)` – take `amount` from the bank

  `money <-- CashLeft` – en enquiry operation that returns the amount of money left

A simple specification (cont.)

```
MACHINE OneAccountBank            the name of a machine
VARIABLES account                 we need a variable
INVARIANT account:NAT             account is a natural number
INITIALISATION account:=0         we start with zero account
OPERATIONS
  FeedBank(amount)=               putting money
  PRE amount:NAT                  preconditioned operation
  THEN
    account := account + amount   updating the account
  END
```


A simple specification (cont.)

```
  RobBank(amount)=                withdrawing money
  PRE amount:NAT                  preconditioned operation
  THEN
    account := account - amount   updating the account
  END;

  money <-- CashLeft=             how much money left?
  BEGIN
    money := account              setting the return value
  END
```