Formal Methods: Introduction

Klaus Draeger

Lecture 1: Introduction; the B-Method

The aim of this lecture is to:

- Explain the need for *Formal Methods*:
  - describe the issue with *software bugs*,
  - outline why formal methods (i.e. formal specification) are a *good or necessary* solution,
- Provide an overview of the *B-Method*:
  - outline the structure of a B specification – *Abstract Machines*,
  - present a simple example of an *Abstract Machine*.
- Introduce B-Method CASE tools:
  - **AtelierB** – supports all stages of B-Method, e.g. syntax and type checking.
  - **ProB** – a B specification “simulator”.

Impact of Poor Software

- Today Software controls virtually all our systems and activities.
- Software bugs can be a really **big issue**.
- The consequences could be
  - a simple nuisance, e.g. laptop crashes,
  - financial loss, e.g. unauthorised financial transfers,
  - loss of property, e.g. Ariane 5 rocket explosion 1996,
  - **loss of life**, e.g. Therac-25 radiotherapy machine, self-driving vehicles

Lessons from Complex Software Projects

Experience shows that **much of the cost** of a complex software projects is spent on **fixing errors.**
Most of these errors are:

- Introduced **early**, during requirements, specification and design stages.
- Only found during *implementation*, *testing* or *maintenance*

Errors are usually caused by:

- lack of precision at the requirements stage,
- incomplete or omitted specification stage,
- making poor design decisions

Attempted Solution: Natural Language

The first software development techniques were pretty ad-hoc and very informal

Natural language approach or some structured subset is used to try to produce a “precise” description of the system.

But natural language is ambiguous, imprecise, etc.

Example: consider the variety of programs you can produce from the same coursework specifications

Partial solution UML

A widely used Software Engineering methodology is **UML**

- based on “**non-formal structured**” diagrams,
- breaking down the problem into **sub-systems (objects)**,
- capturing aspects like
  - the relationship between **data**
  - the **flow of information** through a system.

Showing correctness relies on extensive **testing.**

Limitations of testing

- Thorough testing is **good** for reducing bugs
- I cannot ensure their **absence.**
- Example:
  - After the end of lifefor Windows XP, Microsoft performed a formal analysis
  - This a large number of previously undetected errors including 10000s of null pointer exceptions
  - This despite years of updates and 100s of millions of users
- This is acceptable for some systems, but **not** all of them

Safety and Security Critical Systems

The **Safety and Security Critical Systems** sectors of the software industry has are several recognised **certification standards** for software quality:

- **Evaluation Assurance Level** (EAL1 – EAL7): for the **security** of information systems, e.g. banking sector.
- **Safety Integrity Level** (SIL1 - SIL4): for the **safety** of railway systems, automotive, chemical systems, etc. SIL4 **"Mean Time to Failure"** is about **100,000 years**.

For the highest levels of certification, **"formal methods"** are either essential or legally required.
Companies that produce this type of software (e.g. Siemens, Quinetic) have to use formal methods to **guarantee its quality**.

Formal Specifications

Before we can **check** if software is correct, we first have to **define** what that means

This description will be some kind of **specification** which is

- More abstract than actual code
  - Data specified using mathematical structures like **sets**
  - Focusing on **what** an operation should do, not **how**
- More detailed than e.g. UML
  - Describes **properties** which the data should always satisfy
  - Decribes what **changes** operations make to the data
- Precise, using logic and mathematics

Formal Specifications: The B Method

The building block of the B-method is the concept of an **Abstract Machine (AM).**
It is a concept similar to the programming concepts of: **modules** or **class definition** (e.g. Java).
An abstract machine (model) is a specification of:

- **what** (part of) a system should be like (data), and
- **how** it should behave (operations).

An abstract machine has a:

- **name** to let the rest of the system refer to it,
- **local state**, represented by its variables,
- **interface**, i.e. a set of operations to access and update the state variables.

B Machine Structure

Abstract machines have several main parts, though not all parts are always present.
(Don't worry about notation yet, we will explain it soon)

- SETS:
  Defines entirely new types of values, e.g.
  $Days = \{Mon, Tue, Wed, Thu, Fri, Sat, Sun\}$
  (similar to enums)
- CONSTANTS and PROPERTIES:
  Define derived types based on existing ones, e.g.
  $Weekend \ <: \ Days \ \& \ Weekend \ = \ \{Sat, Sun\}$
  $badDay \ : \ Days * NAT \ \& \ badDay \ = \ (Fri, 13)$

B Machine Structure

Abstract machines have several main parts, though not all parts are always present.
(Don't worry about notation yet, we will explain it soon)

- VARIABLES: Defines the machine's variables
- INVARIANT: Defines the variable properties, e.g.
  $spend : NAT \; \& \; budget : NAT \; \& \; spend <= budget$
- INITIALISATION: Given initial values, e.g.
  $spend := 0 \; || \; budget := 1000$
- OPERATIONS:
  Contains the operations which can query or change the variable values

B Machine Example 1

MACHINE *FamilyMeeting*

SETS

Family = {Amber, Ben, Claudia, Derek, Emily}

CONSTANTS

Nuisances

PROPERTIES

$Nuisances <: Family \ \& \ Nuisances = \{Ben, Claudia\}$

VARIABLES

Invited

INVARIANT

$Invited <: Family - Nuisances$

INITIALISATION

$Invited := \{\}$

OPERATIONS

$addInvite(new) = BEGIN \ Invited := Invited \ \backslash/ \ \{ \ new \ \}$ END

END

B Machine Example 2

```
MACHINE Sieve
    CONSTANTS
        maxNumber
    PROPERTIES
        maxNumber : NAT & maxNumber = 1000000
    VARIABLES
        sieve, primes
    INVARIANT
        sieve <: NAT & primes <: NAT & sieve/\primes = {}
    INITIALISATION
        sieve := 2..maxNumber || primes := {}
    OPERATIONS
        nextPrime = PRE sieve /= {} THEN
            primes := primes \/ min(sieve) ||
            sieve := {nn|nn : sieve & nn mod min(sieve) /= 0}
        END
END
```

Tools and notation

- Throughout this module we will use two tools to write specifications:
  - Atelier B, an editor and type checker
  - ProB, a Simulator
- These will use an ASCII based notation for mathematical symbols
  - For example, the **union** of two sets like $A \cup B$ will be written *A\/B*
  - Atelier B has a sub-window for browsing this notation
  - There is a pdf with the notation on blackboard (and you get to use it in the ICT)

Sets

- Sets are essentially "bags" of values.
  - No inherent order
  - Duplicates don't count: any value is either in the set or not
  - For example $\{1, 2\} = \{2, 1\} = \{1, 1, 2, 1, 2\}$
- They are a fundamental structure in maths
  - All maths can be defined in terms of sets
  - Additional structures (orderins, functions, ...) are really just more sets
  - But it can be more convenient to assume other pre-defined types like numbers

Sets in mathematics

- Sets can contain essentially anything.
  - $\mathit{Colours} = \{\mathit{red}, \mathit{green}, \mathit{blue}\}$
  - $\mathit{SmallPrimes} = \{2, 3, 5, 7\}$
  - Nested sets: $\mathit{Multiples} = \{\{2, 4, 6, 8\}, \{3, 6, 9\}, \{5\}, \{7\}\}$
    - Note that $\{2\}, \{\{2\}\}, \{\{\{2\}\}\}$ etc are all **different** things
- Sets used in the B method cannot be "mixed" like $\{\mathit{green}, 8, \{\}\}$
  - This restriction does not exist in more general theories
  - But it makes it easier to reason about the sets

Set operations

There are a number of common operations we can apply to sets

Operation	Math notation	ASCII notation
Union	$A \cup B$	A \/ B
Intersection	$A \cap B$	A /\ B
Set difference	$A \setminus B$	A - B
Member	$x \in A$	x : A
Non-member	$x \notin A$	x /: A
Subset / strict subset	$A \subseteq B$ / $A \subset B$	A <: B / A <<: B
Not a subset / strict subset	$A \nsubseteq B$ / $A \not\subset B$	A /<: B / A /<<: B
Number of elements	$\lvert A \rvert$	card(A)

Set definitions

New sets can also be defined in a number of ways

Definition	Math notation	ASCII notation
Empty set	$\emptyset$ or $\{\}$	$\{\}$
Explicit enumeration	$\{1, 2, 5\}$	$\{1, 2, 5\}$
Integer range	$\{2, \ldots, 7\}$	2..7
Power set	$\mathbb{P}(A)$ or 2^A	$POW(A)$
Product / set of pairs	$A \times B$	A * B
Set comprehension	$\{x \in \mathbb{N} \mid x < 15\}$	$\{x \mid x : NAT \mathbin{\&} x < 15\}$

These can be combined, e.g. $\{A \mid A : POW(NAT) \mathbin{\&} 2 : A\}$

Numbers

There integers and their subsets have the usual arithmetic operations

Definition	Math notation	ASCII notation
Integers	$\mathbb{Z}$	INTEGER
Natural numbers from 0	$\mathbb{N}$	NAT
Natural numbers from 1	$\mathbb{N}_1$	NAT1
Addition	$x + y$	x + y
Subtraction	$x - y$	x - y
Multiplication	$x * y$	x * y
Division	x/y	x / y
Remainder	$x \mod y$	x mod y
Power	x^y	x ** y

Numbers

There integers and their subsets have the usual arithmetic operations

Definition	Math notation	ASCII notation
Previous number	$pred(x)$	pred(x)
Next number	$succ(x)$	succ(x)
Equality	$x = y$	x = y
Inequality	$x \neq y$	x /= y
Less	$x < y$	x < y
Less or equal	$x \leq y$	x <= y
Greater	$x > y$	x > y
Greater of equal	$x \geq y$	x >= y

Formal Specifications: The B Method

Formal Specifications: The B Method