6SENG001W Reasoning about Programs

Lecture 1

Introduction to Formal Methods

&

the B-Method

Overview of Lecture 1: Introduction & B-Method

The aim of this lecture is to:

- Explain the need for *Formal Methods*:
  - describe the *"software crisis"*,
  - possible solutions, i.e. *"software engineering"*,
  - outline why formal methods (i.e. formal specification) is a *good or necessary* solution,
  - advantages of using the B-Method for formal specifications.
- 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:
  - **Atelier B** – supports all stages of B-Method, e.g. syntax & type checking.
  - **ProB** – a B specification "animator".

PART I

The Need for Formal Methods

Is there a “Lack of Confidence” in Software?

Consider:

A current “hot” topic of research is *“driver-less”* cars.

Question:

How do the public feel about getting into a *driver-less car*?

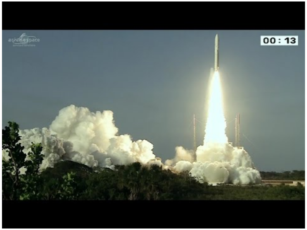
So lets look at some of the responses from a small sample of the public.

“Fake News” Fan

“I’ve never heard of any software ever going wrong.
Software bugs – fake news. Hilary loves software bugs.
Make Software Great Again!”

“Fake News” Fan

*“I’ve never heard of any software ever going wrong.
Software bugs – fake news. Hilary loves software bugs.
Make Software Great Again!”*



ESA’s Ariane 5 rocket exploded just after takeoff on its maiden flight!

Due to a **trivial software bug** – converting a 64-bit float to a 16-bit integer!

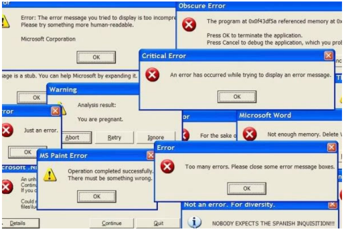
Unfortunately, its **not** “Fake News”.

The Microsoft Employee?

"No problem. I'm sure they'd have got rid of any bugs before they go on sale to the public. Just like when they release a new version of Windows."

The Microsoft Employee?

"No problem. I'm sure they'd have got rid of any bugs before they go on sale to the public. Just like when they release a new version of Windows."



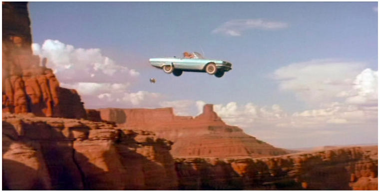
Is your name Bill Gates by any chance?

Extreme Sports Enthusiast

"I live near some cliffs & a railway crossing, but what's a few minor accidents between friends?"

Extreme Sports Enthusiast

"I live near some cliffs & a railway crossing, but what's a few minor accidents between friends?"



Ouch – that's got to hurt!

Clinically Insane?

“Playing Russian Roulette is my No. 1 hobby, so I just can’t wait!”

Clinically Insane?

“Playing Russian Roulette is my No. 1 hobby, so I just can’t wait!”

Safer to take up chainsaw juggling, unless you’re the world’s favourite rabbit.

Current Perception of Software & Computer Systems

The general perception of computer systems (software) is that it is — inherently unreliable & full of software bugs.

- In the past, this was mainly due to **hardware faults**.
- Now mainly/exclusively due to **unreliable software**.

Question:

So based on what you know about software would you use a driver-less car?

YES or NO

Impact of Poor Software

Today Software controls virtually all our systems & activities.

So if this software is not of a high quality, (e.g, correctness, robustness, etc) it can be a really BIG PROBLEM.

When the software is faulty, then a range of problems can arise:

- a simple nuisance, e.g. laptop crashes,
- financial loss, e.g. unauthorised financial transfers,
- safety of property, e.g. Ariane 5 rocket explosion 1996,
- loss of life, e.g. nuclear power plant accident, plane crashes, etc.

Cost($\pounds$) of Poor Software

Many (most?) large complex software systems:

- cost far more than initially budgeted for,
- are delivered late to the customer or are never delivered at all,
- faulty when declared finished & delivered.

Examples

- Most/all Government software systems, e.g. $\pounds 12.4^{+} bn$ NHS IT System.
- Microsoft OS's *bugs* & *viruses* – financial impact on huge number of individuals & companies.
- London Ambulance Service's Ambulance dispatcher system. (Cost in the 10's of millions of $\pounds$.)
- Cancellation of BBC's $\pounds 100m$ Digital Media Initiative (DMI) system.

“Software Crisis”

These problems of poor quality software, i.e. buggy & unreliable, with the associated problems of:

- a negative impact on our daily lives,
- the excessive cost of developing “working” systems,
- the total waste of money on NOT developing “working” systems,

are usually referred to as the – “Software Crisis”.

Conclusion

So anything that can improve the quality of the software we use will have a significant impact on all of our lives & bank balances.

Lessons from Developing Large Complex Software Projects

Experience shows that the majority of the cost of a developing a large complex software project is spent on fixing errors.

Most of these errors are:

- Introduced at the start of a software project, e.g. requirements, specification & design stages.
- Only found during *implementation*, *testing* or *maintenance*
  E.g. a system crash!
- **Never detected**.

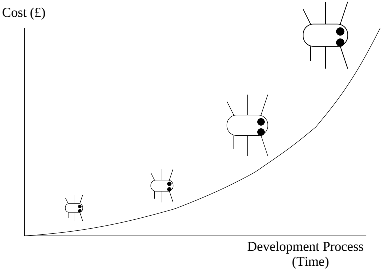
Errors are usually caused by:

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

Outcome – We are Engaged in a Bug War!

Bugs are introduced at the *start of a project*, but are only detected *towards the end* or *not at all!*

Bugs tend to breed, so the longer they are around, i.e. earlier they appear, the more time they have to breed. (Consider your own software projects.)



Important implication — cost of fixing bugs grows very dramatically the later in the software project they are found & corrected, e.g. BBC DMI system.

This is the (hidden?) cost of the software crisis.

Attempted Solution I: Natural Language

The first software development techniques were pretty ad-hoc, very informal & did not use any Formal Methods.

Natural language approach or some structured subset is used to try to produce a:

> *“precise” description of the system.*

Does not work since natural language is ambiguous, *imprecise*, etc.

The use of natural language can **never** result in an unambiguous, precise & complete description of a system.

So it could never be good enough to use to as the starting point for developing good quality software.

Example: consider the variety of programs[1] students produce from the same coursework specification such as *input student marks & output a summary*.

---

[1] Assuming no plagiarism of course!

Solution II: Software Engineering

Recognition of these problems has led to attempts to improve software development techniques, through the use of *software engineering*.

All very well to apply *”engineering principles”* approach, **BUT** compared to other engineering activities, e.g. building & manufacturing, engineering large software systems is very different:

- A large creative design phase, a trivial manufacturing phase, but requires a substantial testing phase.
- Software is far too easy to modify (i.e. *hackable*) & this contributes to many problems, aka. *“feature creep”*.
- Installed software is *“maintained”*, i.e. *“patches”*, *“updates”* & new features added.
- Eventually these changes result in a *loss of structure & understanding*, etc. of the original system.

Exercise: Compare this to building & maintaining a bridge or a plane.

Consider the “Standard” Software Development Life-Cycle Phases

Requirements Analysis	Identify, understand & define the system & problems that need to be solved.
System Specification	Write the *requirements* in an agreed form, e.g. functions, performance, reliability & usability.
Design	Alternative ways of satisfying the specification are investigated. *“Bridge”* from the start – *What is to be done?* to the finish – *How is it to be done?*
Prototyping & Implementation	Chosen *design* translated into *code*. Requires all details about functions & data to be sorted out.
Verification, Testing & Integration	Testing of individual components & the whole system.
Maintenance & Enhancement	Finished system delivered & from then on, bugs corrected & new features added.

Solution II.1: Software Engineering Methodology – UML

Currently, the most widely used Software Engineering methodology is *UML (+ Design Patterns)*.

Consider UML:

- based on "non-formal structured" diagrams,
- aimed at breaking down the problem into *sub-systems (objects)*,
- capturing the relationship between *data* & the *flow of information* through a system.
- etc. etc.

The way the software produced is shown to be correct relies *solely on extensive testing.*

It does not support any notion of *"mathematical proof"*, so can not *"prove"* that the software is correct.

Solution II.2: Software Testing

Software "bugs" are universally accepted & expected.

Encourages sloppy standards in software production, that would not be tolerated in other engineering disciplines, e.g. bridges, planes.

UK software suppliers are legally liable for the software they produce, under *Civil Liability* law & *Consumer Protection Act of 1987*.

A consequence of this is the reliance on *software testing*, but unfortunately the 1972 *Turing Award* winner Edsger Dijkstra pointed out that:

> *"... testing can **only** reveal errors but **never** demonstrate their absence ..."*

BUT – to function correctly a large software system has to make thousands or millions of: inputs, tests, decisions & calculations etc.

In order to *"prove"* such a system correct a *test plan* would have to consider every possible combination of these – **not feasible**.

Question: Does Software Testing Work?

Several years ago, Microsoft, as part of an attempt to improve the quality of their operating systems performed a *“formal analysis”* of *Windows XP*.

As expected, Windows XP was found to contain many different types of errors.

In particular, it contained more than **10,000** of a particular type of error known as *“null pointer exceptions”*!

The significance of this is that:

- these bugs had **not been detected**, let alone fixed before it reached the *end of its life*,
- even after being used by 100s of millions of users for many years & numerous “patches” & “updates”.

Answer: **NO**

The real *“Fake News”* is that software testing **“proves”** that software doesn’t contain any bugs!

What about the Highest Quality Software?

In the *"Safety or Security Critical" (SSC)* industries, e.g. nuclear, aviation, transport, health, finance & defence, the highest quality software is required.

For SSC software either errors/bugs are NOT acceptable, or errors are so rare that it is considered an *acceptable risk*.

Problem: most/all large complex software projects that use standard methods e.g. UML, usually end up:

- containing many errors, or
- way over budget, or
- delivered late, or
- never delivered at all, or
- take so long to be delivered they get cancelled, or
- a combination of the above.

UML etc. is okay for *"bog-standard"* software, but its **not** okay (or even legal) for *Safety or Security Critical* systems.

“Quality Standards” for Safety & Security Critical Systems

Within the *Safety & Security Critical Systems* sectors of the software industry there are several recognised *certification standards* for software quality:

- *Evaluation Assurance Level*: for the *security* of information systems, e.g. banking sector.
  “Levels of assurance”: EAL1 – EAL7, (7 is highest).

- *Safety Integrity Level*: for the *safety* of railway systems, automotive, chemical systems, etc.
  “Levels of Integrity”: SIL1 - SIL4, (4 is highest).
  SIL4 *“Mean Time to Failure”* is about **100,000 years**.

For systems to achieve the highest levels of certification, the use of *“formal methods”* is 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*.

Problems with Solutions I & II

- Non-formal software development methods – *natural language*, *UML*, etc, **can not result in or guarantee** the production of high quality safety/security critical software.
- Since they all allow *errors to be introduced* far too easily, e.g. specification stage.
- Do **not** provide a satisfactory way to get rid of them, e.g. just *testing*.
- *None* has any notion of *"mathematical proof"*, which is the *ultimate bug killer*.
- So we need some other approach that is better.
- So the only thing that will do the job is *formal (mathematical) methods*.

Solution III: Software Development using *Formal Methods*

Definition

A *formal method* is a systematic mathematically based approach used to determine whether a program has certain desirable properties.

Formal Method Development Stages:

- Starts with the construction of an initial *"abstract model"* (mathematical specification) of the software system.
- It then concentrates on the construction of a *sequence of models* by successive *"refinements"* starting from the initial model.
- The *"desirable properties"* to be proved are parts of the models: *"invariants"* & *"refinements"*.
- At the end of the process, the *"most refined model"* (*"implementation"* model) is *automatically translated* into *program code*.

Overview of Formal Methods Refinement

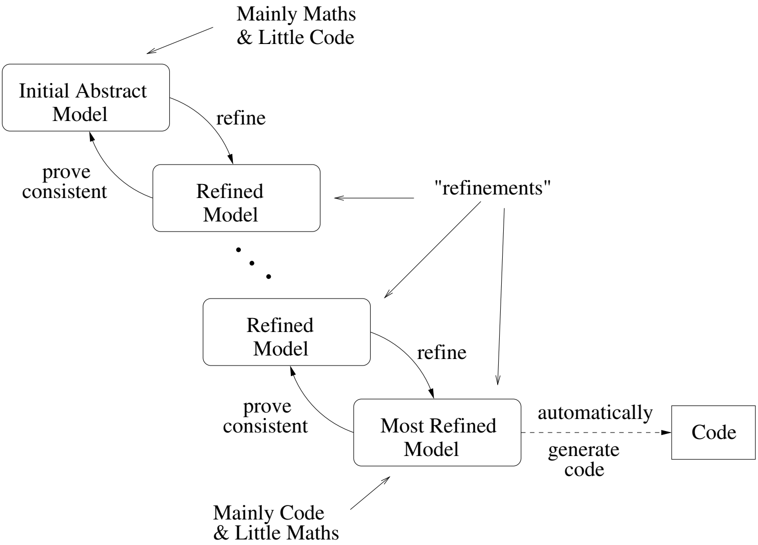


Figure : 1.1 Formal Methods Refinement

Solution III: Formal Specifications

We must use some kind of formal (i.e. mathematical) language to produce a *formal specification* of a system.

Formal Specification is seen as a "real" solution to the software crisis.

Since it allows the requirements of a software system to be described:

- *succinctly*,
- *precisely*,
- *unambiguously*
- allows essential & desirable properties to be *proved*,
- allows checking of *consistency* of the specification.

The consequences of **NOT** using *Formal Specifications* & using an *"informal specification or set of requirements"* (i.e. non-mathematical), is that we do not have a sound way of checking:

> *"If the final software system does what it was actually required to do".*

Essential Role of "Proofs" about Specifications (Models)

Mathematical Proofs about a *formal specification* of a software system play an *essential* role in developing quality software.

Proof that a *"desirable property"* holds about the system specification, can mean that the same property holds about the *implemented system*.

Example

To be able to prove that:

> *"Thermostat control software does not crash if it inputs unexpected temperature readings, but closes the system down safely."*

This is obviously a **VERY** desirable property to prove for a thermostat in a nuclear power plant!

Note: the proof of desirable properties about a system at the *specification stage* can also *eradicate errors* that could continue in the system until it was *delivered* or more likely *never found*.

Which Formal Method?

So, what we need is a *Formal Method* & *Specification Language* which offers:

Precision:
- be as *precise as necessary*,
- so as to *remove any ambiguity* about what is required.
- Similar to precision of a programming language.

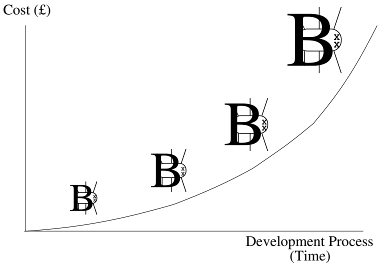
Abstraction:
- allows us to *abstract away from unnecessary detail*, e.g. algorithms & data structures,
- until we are *ready to deal with it*.

Proof:
- ability to *"prove" desirable properties* about our specifications.
- *"proofs"* are best way to eliminate errors & bugs.

Answer: *B-Method*

Killing Bugs with B

For the reasons outlined above, B can be seen as an effective *bug killer*!



There is an additional cost when using B to eradicate bugs, & that is, that much more effort is spent on the specification stage of a project.

It also usually requires either an investment in staff development, i.e. teaching them B; or employing trained B practitioners.

For this reason companies are often not willing to make this “up front” investment, but usually end up having to *spend much more on the testing stages* of a project getting rid of bugs.

PART II

Introduction to the B-Method

So what is the *B-Method?*

- History: created by Jean-Raymond Abrial (80's & 90's).
  Successor to Abrial's Z Specification Language.
- Its a *formal software development method* that supports *all* stages of the *software life-cycle* in a uniform and formal way.
- Incorporates & combines many ideas from a range of formal methods.
- The main elements are: *B models* (specifications) & *proofs*.
- A *B model* is a collection of *"components"*, called *"machines"*.
- B specifications are written in its formal specification language: *Abstract Machine Notation (AMN)*.
- AMN is a "wide spectrum" pseudo-programming language that covers both *abstract specifications* & *implementation level code*.
- Supported by CASE tools, we shall use two B-Method tools:
  - **Atelier B** – syntax & type checking, theorem prover. (ClearSy, France).
  - **ProB** – animator, (Heinrich-Heine-University, Germany).

Overview of B-Method Software Development

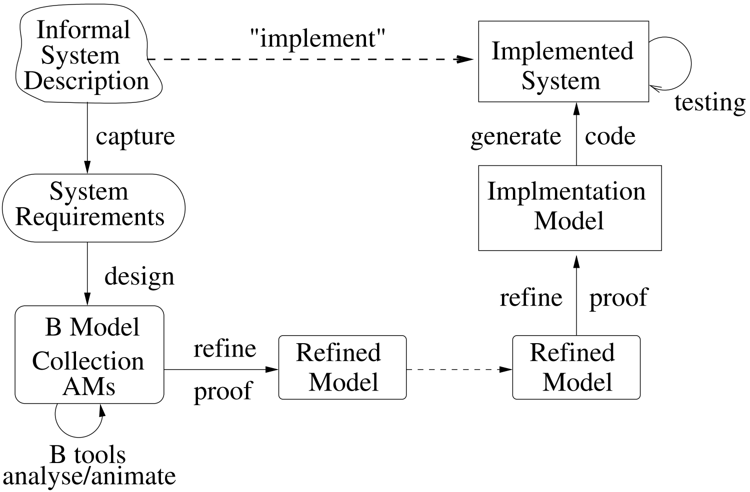


Figure : 1.2 B-Method Software Development

B-Method Software Development Stages

The B-Method's formal software development process can be broken down into several "phases" or "stages":

- *Capture the System Requirements*

  — Initial *"vague system description"* turned into a list of *requirements*. (Pre B-Method.)

- *Design a "B Model" (Abstract Model) from the System Requirements*

  — B model is a *"high level"*, *"abstract" specification*.

- *Refine the"Abstract Model" into an "Implementation Model"*

  — Translate the *"maths"* descriptions into *"program code"*.

- *Generate "Program Code" from Implementation Model*

  — **Atelier B** tool generates: C, C++, Ada code.

Industrial Use of the B-Method

The B-Method's approach to formal software development has been applied in large industrial projects by the following companies:

- Qinetic – very high quality s/w systems, e.g. defence, avionics
- Siemens (Transportation Systems) – train control systems, e.g. Paris METRO Line 14.
- Volvo, Peugeot, Renault – electronics in cars
- Nokia – microelectronics
- France Telecom – electronics
- Gemplus – smart card security, e.g banking sector
- BOSCH – electronics
- EDF – nuclear control system design

Possible current & future areas – Google driver-less cars, Apple iCar??

PART III

Introduction to the B-Method's Abstract Machines

B-Method's Abstract Machines (AM)

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*, *class definition* (e.g. Java) or *abstract data types*.

An *abstract machine (model)* is a *specification* of:

- **what** a system should be like, or
- **how** it should behave (*operations*);
- but **not how** a system is to be built, i.e. no implementation details.

An *abstract machine* has a:

- **name**,
- **local state**, represented by *"encapsulated" variables*,
- **interface**, i.e. a collection of operations, that can access & update the state variables.

Example: `PaperRound` Abstract Machine

A *paper round manager* keeps track of houses that receive deliveries, it uses a state variable $houseset$ & has two operations: add & $number$.

MACHINE $PaperRound$

  VARIABLES $houseset$

  INVARIANT $houseset \subseteq \mathbb{N}_1$

  INITIALISATION $houseset := \{\}$

  OPERATIONS

    $add(new) =$

      PRE $new \in \mathbb{N}_1 \ \wedge \ new \notin houseset$

      THEN $houseset := houseset \cup \{ \ new \ \}$

      END ;

    $ans \leftarrow number =$

      BEGIN

        $ans := card(houseset)$

      END

END

Abstract Machine Notation (AMN): B Symbols & Machine Readable ASCII

The B tools require machine readable ASCII versions of the various B symbols & AMN notation.

B symbols & AMN used in `PaperRound`:

Symbol	ASCII	Description
$\mathbb{N}_1$	`NAT1`	Set of natural numbers from 1
$x \in A$	`x : A`	x is an element of A
$x \notin A$	`x /: A`	x is not an element of A
$A \subseteq B$	`A <: B`	A is a subset or equal to B
$\varnothing$, $\{\ \}$	`{}`	Empty set
$A \cup B$	`A \/ B`	Union of A and B
$\mathrm{card}(A)$	`card(A)`	Number of elements in set A
$P \wedge Q$	`P & Q`	Logical Conjunction (“P and Q”)
$var := exp$	`var := exp`	Assignment: `var` set to `exp`
$out \leftarrow op$	`out <-- op`	Operation `op` with output `out`
`BEGIN` .. `END`	`BEGIN` .. `END`	Statement Block
`PRE` .. `THEN` .. `END`	`PRE` .. `THEN` .. `END`	Operation Pre-Condition

Atelier B & ProB Machine Readable PaperRound

PaperRound

```
MACHINE PaperRound

  VARIABLES
    houseset

  INVARIANT
    houseset <: NAT1

  INITIALISATION
    houseset := {}

  OPERATIONS

    add(new) =
      PRE  new : NAT1  &  new /: houseset
      THEN
           houseset := houseset \/ { new }
      END ;

    ans <-- number =
        BEGIN
             ans := card( houseset )
        END
END
```

B-Method Tools: **Atelier B** & **ProB**

In the tutorials you will use the two B-Method CASE tools to develop B specifications:

- **Atelier B** – supports all stages of B-Method:
  syntax & *type checking*, *theorem proving*, *refinement* &
  code generation into C, C++ & Ada.

- **ProB** – is an *"animator"*.
  It allows you to animate (i.e. execute) the operations defined in a B specification.

First Tutorial

The first tutorial is intended to introduce you to the B tools & *AMN*.

You will be required to type in the `PaperRound` machine into **Atelier B**, then syntax & type check it using **Atelier B**.

Finally, animate it using **ProB**.