Formal Methods - An Introduction*

Mairead Meagher

January 12, 2019

Contents

*Much of the introductory material contained in this chapter is based on [Rep13]

1 Motivation for Formal Methods

Since the introduction of early commercial computers in the 50s, we depend on computers more and more. From the final goods used in everyday life (watches, consumer electronics, telephones, cars, etc.) to the largest national and international infrastructures (energy, transportation, etc.), many functions that were previously performed mechanically or electrically are now handled digitally. As a consequence, the number of microcontrollers and microprocessors now far exceeds (and grows faster than) the total human population on the Earth. The emergence of '*The Internet of Things*' is leading to a huge explosion of embedded software installations. This importance of computing artefacts in our daily live has been made possible by a combination of major advances in all facets of computer science:

- Increase in computing power, as illustrated by Moore's and Koomey's laws, which state that the computation power of a processor and the number of operations that can computed with a given amount of energy double every 18–24 months;
- Increase in data storage capabilities, as illustrated, e.g., by Kryder's law, which states that the number of bits that can be stored on magnetic disks doubles every 12 or 18 months;
- Increase in connectivity, as illustrated by the growth of telecommunication bandwidth and mobile traffic;
- Increase in software productivity, which enabled the development of large amounts of software, the growth of which is estimated to be exponential, at least in the case of open source software.

1.1 Safety-critical and Mission-critical systems

In many cases, computer automation delivers more flexible and reliable devices and infrastructures by enabling repetitive tasks previously done by humans, often in a sporadic manner, to be accomplished with precision and regularity. There is, however another side to this. Any error in these systems are repeatable and potentially more catastrophic. There are two kinds of systems which are particularly sensitive to these potential problems:

- *Safety-critical systems* (or *life-critical systems*). Failures or malfunctions in these may threaten human lives. Examples include transport(cars, trains, airplanes etc.), energy(nuclear plants etc) and medicine (assisted medical devices, assisted surgery). The malfunction of the Therac 25 radiotherapy engine killed five people in the 1980's due to bad software design.
- *Mission-critical systems* (or *business-critical* or *correctness-critical systems*). Malfunctions in this category may lead to financial losses (e.g.

banking applications). Another example would be electronic-voting systems where the correctness of the system is necessary to protect the running of elections and therefore democracy. Some hardware failures here include the Pentium fLoating-point division bug(1994) (cost Intel 475 million dollars) and Cougar Point chipset flaw (cost one billion dollars).

1.2 Why Systems Fail

There are different reasons for failure or malfunctioning of computer systems:

- *Design errors* that prevent a system from achieving its intended functionality, may include :
  - Inappropriate capture of systems requirements. This can be due to the lack of, or bad communication between the experts in the system and the systems analyst;
  - Inaccurate modelling of the actual environment in which the system should function;
  - Unexpected interactions between several concurrent functionalities
- *Hardware faults* including physical or logical issues in microprocessors, micro-controllers, integrated circuits, sensors etc. Certain issues happen due to age of components and cannot be avoided but must be detected and safely recovered from.
- *Software Bugs* - these are logical mistakes when implementing the software, These can be
  - run-time errors;
  - infinite loops
  - deadlocks etc.
- *Security issues* occur when the system is not robust enough to resist outside attacks. This is becoming more and more important.
- *Performance issues* occur when a system cannot deliver its expected quantitative performance (maybe too slow, using too much memory etc.)

1.3 Why writing correct software is difficult

In an ideally simple world, designing and implementing correct and robust computer-based systems should not be a impossible task. But there are practical reasons that make this task more difficult than it should be. In addition to the permanent needs for reducing costs and shortening time-to market, five key factors contribute to make system design more complex

- Certain problems in hardware, software, and system design are inherently difficult. This is the case of fault-tolerant systems, which have to recover from physical or logical failures, and concurrent systems, which rely on the co-operation and coordination of multiple agents executing simultaneously.
- Because of economical competition, new functionalities are constantly added to systems in order to deliver better value to the customers.This race to expanding functionality (*feature creep*) is a major cause for the explosion of the software size (*software bloat*).
- System complexity also derives from the existence of economical competition, as systems often must support or interact with multiple platforms (e.g., hardware architectures, processors, operating systems, middleware, computer languages, etc.) and to handle legacy applications.
- The quest for performance drives system designers and implementers into inventing optimized algorithms that deliver enhanced performance at the expense of increased complexity.
- Finally, the need for security, which comes along with the growing role devoted to computers, forces system designers to introduce new features (e.g., authentication and authorization procedures) that increase complexity and may raise new issues, such as privacy concerns.

1.4 Organizational and Technical Approaches

So how do we approach these systems to ensure that systems are written to conform to what is expected of them? There are two different approaches:

- *Organizational approaches* consider the problem as a particular instance of the more general product quality problem: how to build computer-based systems with zero defects? Various methodologies and standards have been proposed for quality enhancement, such as ISO 9001 (Quality management systems – Requirements), CMMI (Capability Maturity Model Integration) etc.
- *Technical approaches* address the problem by putting the focus primarily on the system itself and on computer-science aspects. In particular, much attention is granted to software aspects, often with careful examination of source code. Many such techniques have been developed for producing, testing, and validating computer-based systems. Many of them (often the less costly and less disruptive ones) have been already adopted by industry and integrated in product development methodologies. These techniques enable to prevent, or detect and eliminate a majority of mistakes in a given product. However, certain mistakes still remain undetected, particularly in the case of complex systems. The existence of such residual mistakes (sometimes called *high-quality bugs*) is a major concern for life-or mission-critical systems. For this reason, alternative and/or complementary techniques have to be investigated.

Formal methods are considered to be the best candidates for going beyond those techniques commonly adopted by industry, and which constitute a promising step towards zero-defect computer-based systems.

2 What are Formal Methods?

> Formal methods are a key enabling technology for building safe and secure computer-based systems. They help to fight the software quality crisis, in conjunction with related approaches, such as better technical education, design methodologies, computer languages, and development tools.

There are many ways to describe Formal Methods:

- One categorization of formal methods is that formal methods are made up of two parts:

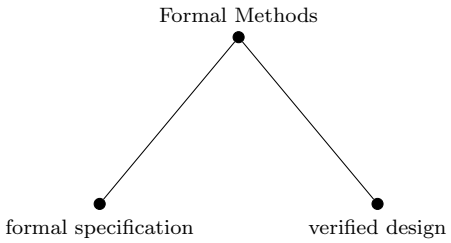

Figure 1: Formal Methods

- Another categorization states that

> "Formal methods in a broad sense are mathematically well-founded techniques designed to assist the development of complex computer-based systems; in principle, formal methods aim at building zero-defect systems, or at finding defects in existing systems, or at establishing that existing systems are zero-defect".

2.1 Languages, Tools and Methodologies

The following three general traits common to most formal methods:

– *Languages*: Formal methods are often associated with mathematical notations that can describe the properties expected from a system and/or the particular ways in which the system is designed (e.g., architecture, algorithms, etc.). Depending on the formal method considered, such descriptions can concern various phases of system

development, from requirements, specification, and design to implementation and run-time execution. Whatever the phase considered, a central idea of formal methods is to consider systems, hardware, and/or software as mathematical objects that can be described and analyzed rigorously. We will be using the Z Notation as our specification language.

- *Tools*: Formal methods often come with software tools that help the developer to ensure that the system will function as expected (obviously, under certain assumptions).

  An important difference between formal methods and traditional testing techniques is the emphasis of formal methods on analyzing (ideally) all possible executions of the system, and not only a few ones. This is essential if the proper functioning of the system has to be mathematically demonstrated, and not only estimated with probabilities.

- *Methodologies*: To be effective, formal methods should be well-integrated within industrial practice. For this reason, most formal methods are equipped with methodological guidelines for a proper use in real-size system development.

2.2 Application Domains and Environment Assumptions

We categorise formal methods under the following headings:

- *Application Domains.* These are the categories of domains to which we can apply formal methods.
  - *Systems Design and Engineering.* We mean computer-based-systems that have a hardware/software component.
  - *Protocol design and engineering.* This is a very suitable application area for formal methods, e.g. *cryptographic protocols.*
  - Software Design and Engineering. This is where the system under design is mostly or entirely a software system, or where a real system is abstracted in such a way that only its software aspects are considered. This is where our area of interest is.
  - Hardware design and engineering. The need for formal methods in this area steps from the cost of making mistakes. Different terminology used, e.g. testing can mean testing of physical components, etc.
- *Environment Assumptions.* A system has an associated environment, which corresponds to the "rest of the universe" with which the system interacts, e.g. human users, other hardware elements, other software components in the system. We distinguish between three kinds of environments:

* A *nominal environment* - this is well-understood and predictable.
* A *faulty environment* - mostly understood and largely predictable but its behaviour can be altered by faults, e.g. hardware malfunction.
* A *hostile environment* - neither totally understood nor predictable; its behaviour cannot be trusted because of the potential presence of a number of adversaries (or attackers or intruders) who can take control of the environment (at least, in part) and modify its expected behaviour.
* *Emerging environments* - these are not well understood and so not fully predictable.

3 Stages of the Formal Process

As seen in Figure 1, the formal methods process can be broken down into

- Formal Specification - Statement of what the software is to do in a precise and unambiguous manner. For the rest of this chapter, we will examine the Formal Specification part.
- Verified design - Using rules of logic, it can be proved formally that a program matches its specification. This is potentially (not usually!) mechanical and therefore can be automated using automated program provers. We will return to the Verified Design later.

3.1 The use of Mathematics in the Formal Process

The use of mathematics:

- Mathematics is used to define the behaviour of the system.
- Mathematics makes it possible to reason about the system.

Why is mathematics used?

- Mathematics is a precise language. There is no ambiguity. This gives the software engineer the precision needed.
- Mathematics is concise. Complex ideas can be expressed very concisely.
- Mathematics is expressively very powerful. Most ideas are capable of being expressed in terms of mathematics.
- Mathematics facilitates formal reasoning. The consistency of what has been written can be checked.
- Using mathematics, simple, coherent, unifying features can be found.

4 Formal Specification

One of the most important documents that is produced as part of a systems life cycle is the specification. It models the behaviour of the system and is referred to constantly throughout the development process. It is important that the specification must be precise and unambiguous and ideally, therefore, should be written in mathematics. Such a specification is called a formal specification. A formal specification is the mathematical encapsulation of informal requirements.

Practitioners have found that writing and debugging a formal specification leads to a fuller understanding and knowledge of the application. It is also very effective in exposing gaps and inconsistencies in that knowledge.

The formal specification uses the notion of abstraction:

- Says **what** is to be achieved, not **how** it is to be achieved;
- Defines whatever level of detail is necessary;
- Uses application- oriented data types and structures, not computer-oriented ones.

Producing high quality specifications is difficult and expensive, as one must carefully avoid the *seven sins of the specifier* [Mey85], which may affect not only specifications in natural language but, for a part, formal specifications as well. We briefly mention these seven sins below and briefly comment their meaning in terms of properties:

1. *Noise*: "The presence in the text of an element that does not carry information relevant to any feature of the problem" — Irrelevant properties.
2. *Silence*: "The existence of a feature of the problem that is not covered by any element of the text" — Missing properties, thus allowing certain "invalid" implementations to be accepted (mathematically, the specification is said to be incomplete).
3. *Over-specification*: "The presence in the text or an element that corresponds not to a feature of the problem but to features of a possible solution" — Superfluous properties, thus prohibiting certain "valid" implementations. We try to write "weak" specifications.
4. *Contradiction*: "The presence in the text of two or more elements that define a feature of the system in an incompatible way" — Unsatisfiable properties, for which no "valid" implementation can exist (mathematically, the specification is said to be inconsistent).
5. *Ambiguity*: "The presence in the text of an element that makes it possible to interpret a feature of the problem in at least two different ways" — Imprecise properties, thus allowing divergent implementations.

6. *Forward reference*: "The presence in the text of an element that uses features of the problem not defined until later in the text" — Properties that depend on each other; in absence of circular dependencies, forward references can always be eliminated by proper reordering of properties using topological sort; the presence of circular dependencies between properties is often a mistake, but can be appropriate in certain cases.

7. *Wishful thinking*: "The presence in the text of an element that defines a feature of the problem in such a way that a candidate solution cannot realistically be validated with respect to this feature" — Unreasonable properties that cannot be realistically implemented.

4.1 Specification Styles

There are some different categorizations of specification styles:

4.1.1 Declarative vs operational specifications

- **Declarative specifications** define what a system or component should do, but not how it should do it. Usually, they express objectives and constraints that any correct implementation of the system or component should satisfy, but they are non-constructive, in the sense that it would be impossible (or, at least, very difficult) to automatically derive from these constraints an efficient implementation of the system or component.

- *Operational specifications* possibly define what a system or component should do, and definitely define (at least, partly) how it should do it. Such specifications are constructive, meaning that one can use them to generate automatically an implementation of the system or component (or, at least, a skeleton of such implementation).

The Z Notation uses the Declarative Specification approach.

4.1.2 Partial vs Total models

When a system is too large, one may apply restrictions to avoid complexity issues by modelling only one or several part(s) of the system. In such case, the model is said to be partial (opposite: total or complete). For instance,

- You might choose to model only the most "difficult" parts of the system, i.e. those worthy of verification.

- You might also model only a representative sample of identical components.

- You might choose stronger environment assumptions to simplify the model.

Obviously, these simplifications should be introduced carefully.

4.1.3 Deterministic vs non-deterministic models

The question of nondeterminism is central in formal specification.

- A model or program written in these languages is said to be *deterministic* if its executions are reproducible, i.e., the same inputs produces the same outputs.
- A model or program is said to be *non-deterministic* if its executions are unpredictable even when performed in identical conditions, i.e., the same inputs may produce different outputs.

  $$x = \pm\sqrt{4}$$

  In this case, there are two possible correct answers. When implementing, we would need to decide on a decidable answer, but at specification stage, it is appropriate to leave that decision until later in the process.

4.1.4 Abstract vs concrete models

When a system is too complex, we usually apply abstractions to simplify the specification. Abstraction is a fundamental concept of formal methods. It consists in replacing a concrete model by an abstract model: both models describe the same system or component, but the abstract model is less detailed and hides (namely, abstracts away) a part of the complexity of the concrete model. The underlying motivation for abstraction is that verification may become possible on the abstract model even if it was intractable on the concrete model. Notice that the notion of abstraction in formal methods is related to the classical notion of abstraction in computer science. The different types of abstraction are:

- Behavioural abstraction. Omit certain, simple, behaviours.
- Data abstraction. One way of affecting this is to use *"application-oriented"* types in the specification.
- Variable abstraction - also know as variable slicing, you reduce the areas of interest to focus on the important area.
- Time abstraction - remove real-time aspects.

4.2 Types of Formal Specification Notations

4.2.1 Model-Based Specifications

(Often called State-Based)
Examples of notations : Z([Spivey]), VDM([VDM90]) , B([Abrial96])

Data types derived from set theory, such as sets, relations, functions and sequences are used to represent the components of the system. Logic is used to define what must be true about these components and the relationship between them. Operations are defined in terms of effect on state. Each operation is defined by the relationship which must hold between the state of the system before and after the operation.

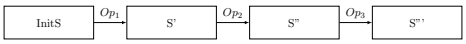

Figure 2: Model-Based / State-Based Systems

4.2.2 Algebraic Specification Languages

Examples are Clear , ACT ONE , Larch .

4.2.3 Process Algebra

Examples are *CCS* (Calculus of Communicating Systems), *CSP*(Communicating Sequential Processes). The programming language occam is based on CSP.

These allow a system to be modelled by a collection of processes which communicate with each other. Process Algebra's are therefore useful for the study of concurrent systems.

These notations allow the description of a system and its constituent parts by means of sets of equations which capture behaviour directly. For large systems, mechanisms need to be provided for allowing the re-use of common components.

5 The Z Notation

5.1 What is a Z Specification?

The main notation that we use for this module is *the Z Notation*. A Z Specification is a mixture of

- Formal mathematical statements
- Explanatory text

The formal mathematical statements have a **structure** imposed and there are **conventions** used. The mathematical component of the specification is based on set theory and one of Z's most distinguishing features is a structuring device called a ***schema***.

5.2 Where did Z come from?

The Z notation was developed at Oxford University's Programming Research Group in the late 1970's and early 1980's by Jean-Raymond Abrial, Bernard Sufrin and Ib Sorenson. From the start, Z was used in industry. IBM used it to re-specify their Customer Information Control System (CICS) . After experimentation, work in large systems in an industrial environment and theoretical investigation, a standard core language has been defined and can be found in Spivey [2].

5.3 The Mathematics of Z

Z is based on standard mathematical notation.

- Logic
  - First order predicate calculus (the most basic form of formal logic).
- Set theory
  - Strongly typed.
  - Relations, functions, sequences all viewed as sets of couplets.
  - Sets can be of any application-oriented type. This means that we can introduce sets at any level of abstraction. e.g. we could introduce the sets [NAME, BOOK] without further specifying what internal structures are involved.
- Extensibility
  - Minimal built in constructs.
  - Extensibility mechanisms.
  - Generic definitions.
  - Basic library.

5.4 Structuring Specifications

On of Z's characteristic features is the structuring of specifications. The schema is one of the basic constructs. We can develop partial specifications and their combination. Fragments of mathematics can be named and later referred to. By using calculus of schemas, i.e. building up complex constructs from more simple schemas, we can build up large and complex specifications using simple building blocks.

5.4.1 Top-level structure

A Z formal specification is made up of formal text and explanatory English text. The formal text is made up of :

- basic type definitions, which define the values which constants and variables may take;
- axiomatic descriptions, which introduce global constants and variables;
- constraints on (previously declared) global variables;
- schemas, which define the state and operations of the system;
- abbreviation definitions, which introduce new global constants.

Each construct in the sequence may use names that are in scope.

5.5 Limitations of Formal Specifications using the Z Notation

There are some limitations on the range of properties that specifications using the Z Notation have:

- They do not define non-functional attributes
  - Performance
  - Usability
  - Size
  - Availability
- Do not tell you how to design the system. In fact, It is possible to specify systems which cannot be implemented.

5.6 A short example

Look at a simple Birthday Book system , i.e. a system which records peoples' birthdays and is able to issue a reminder on a person's birthday. We need to deal with peoples' names and dates. We don't have to worry about the form the names and dates take, we can introduce them as basic types of the specification:

$$[NAME, DATE]$$

We need to describe the system's state space and we do this using a schema:

$$\begin{array}{|l} \textit{BirthdayBook} \\ \hline known : \mathbb{P}\, NAME \\ birthday : NAME \pfun DATE \\ \hline known = dom birthday \end{array}$$

We have just described a system with two components:

- known - the set of known names
- birthday - a function which links a name to a date (that person's birthday)

We have also defined a relationship between these two components, a relationship which will not change for any state of the system. We say that the set known is the same as the *domain* of the birthday or the set of names for which we can validly check birthdays. We call this special relationship the invariant.

5.6.1 Conventions

Conventions make reading and writing specifications easier They allow compact definitions of state changes, inputs and outputs.

Remind Example

Remind

$$\Xi BirthdayBook$$
$$cards! : \mathbb{P}\, NAME$$
$$today? : DATE$$

$$card! = \{n : known \mid birthday(n) = today?\}$$

This shows the use of several conventions which we will see later.[**?**]

- Input parameters have a *?*
- Output parameters have a *!*

5.7 Example of a Formal Specification

5.7.1 Introduction

The *RDS* (Revised Defence Standard) describes the process which a *Contractor* must follow in order to supply a system with safety-critical attributes. The main definition in this specification is given by the schema *CompleteProjectState* which comprises all the objects which a Contractor must define. The state is built by introducing several schemas, named *ProjState1* through to *ProjState8*, each of which includes its predecessor. A later section summarizes the requirements.

5.7.2 Hazard Identification

The Contractor must first identify all hazards that are applicable to the system under analysis, together with the associated accidents and accident sequences. Hazards, accidents and events are modeled by abstract sets:

$$[HAZARD, ACCIDENT, EVENT]$$

The sets *hazards* and *accidents* model the identified hazards and accidents respectively; the accident sequences are modelled by the function *accSeqs*, and the identified accidents are those caused by at least one accident sequence.

*ProjState*1

$$hazards : \mathbb{P}\, HAZARD$$
$$accidents : \mathbb{P}\, ACCIDENT$$
$$accSeqs : ACCIDENT \pfun (\mathrm{seq}_1\, EVENT)$$

$$accidents = domaccSeqs$$

Thus, for each accident, we record at least one non-empty sequence of events that could cause it.

..... and so on .....

References

[Abrial96] J.R. Abrial *The B-Book: Assigning Programs to Meanings,* Cambridge University Press, 1996. ISBN 0-521-49619-5.

[ALG85] Ehrig and Mahr *Fundamentals of Algebraic Specification,* Springer-Verlag, 1985.

[Hoare85] Hoare *Communicating Sequential Processes,* Prentice Hall, 1985.

[Mey85] Bertrand Meyer *On Formalism in Specifications,* IEEE Software, 2(1):6–26,1985.

[Milner] Milner *A Calculus of Communicating Systems,* Lecture Notes in Computer Science vol. 92, Springer-Verlag, 1980.

[Rep13] Editor: Dr. Hubert Garavel *Formal Methods for Safe and Secure Computers Systems. BSI Study 875,* 2013.

[Spivey] Spivey *The Z Notation - A Reference Manual, 2nd Edition,* 1992:Prentice Hall.

[VDM90] Jones *Systematic Software Development Using VDM, 2nd Edition,* 1990, Prentice Hall.