Formal Methods in Programming

Lecture 1

Introduction to Formal Methods.
Overview of Formal Methods techniques & tools

Formal Methods in Programming

„A more mathematical approach is inevitable. Professional software development—not the everyday brand practiced by the public at large—will become more like a true engineering discipline, applying mathematical techniques. I don't know how long this evolution will take, but it will happen. The basic theory is there, but much work remains to make it widely applicable."

[Bertrand Meyer, 2001]

Lecture Outline

- References
- The need for Formal Methods (FM)
- What are FM?
- Why using FM?
- FM in software lifecycle
- FM levels
- Formal specification: advantages, taxonomies, examples
- Formal verification

References

[1] Almeida, J.B., et al., *Rigorous Software Development: An Introduction to Program Verification*, Springer, 2011.

[2] Bowen, J.P., Hinchey, M.G., *Seven More Myths of Formal Methods*, IEEE Software, 12(4):34-41, 1995.

[3] Bowen, J.P., Hinchey, M.G., *Ten Commandments of Formal Methods*, IEEE Computer, 28(4):56-63, 1995.

[4] Bowen, J.P., Hinchey, M.G., *Ten Commandments of Formal Methods ... Ten Years Later*, IEEE Computer, 39(1):40-48, 2006.

[5] Clarke, E.M., Wing, J.M., et al., *Formal Methods: State of the Art and Future Directions*, ACM Computing Surveys, 28(4):626-643, 1996.

[6] Hall, A., *Seven Myths of Formal Methods*, IEEE Software, 7(5):11-19, 1990.

[7] Haxthausen, A.E., *An Introduction to Formal Methods for the Development of Safety-critical Applications, 2010.*

[8] Holloway, C.M., *Why Engineers Should Consider Formal Methods*, Proceedings of the 16th Digital Avionics System Conference, 1997.

The need for FM

- Today's trend: *ubiquitous computing*
  - computers and software are everywhere, being used in almost all aspects of daily life
  - e.g. transportation, finance, health care, aviation, telecomunications, etc.
- Major requirement: *software reliability*

  „Reliability is defined as the combination of correctness and robustness or, more prosaically, as the absence of bugs." [Bertrand Meyer]

  Correctness = software's ability to behave according to the specification

  Robustness = software's ability to react appropriately to cases outside the specification
  - especially when failures may lead to catastrophes, where people die or values/money are lost
  - e.g. software controlling trains should be correct, so as to avoid train collisions; banking software should be correct, so as to perform transactions as required and secure, so as to prevent unauthorized access
- Critical issue*: software complexity*
  - such software is rather complex and it is not an easy task to make it bug-free

The need for FM

- State of facts

  „Software and cathedrals are very much the same. First we build them, then we pray." [Sam Redwine, 1988]

  „Despite 50 years of progress, the software industry remains years - perhaps decades – short of the mature engineering discipline needed to meet the needs of an information-age society." [Scientific American, 1994]

  - software development surveys show that software is often full of bugs, leading to delays, cost overruns, usability problems
  - bugs are so common in commercial products, that companies now systematically include a bug report system with each release
- Cost of bugs
  - in 2002, the North American Institute for Standards and Technologies estimated the cost of bugs in the American economy to ascend to 59 billion $
  - the cost of maintaining software is about 2/3 of the total cost and a software specification fault is about 20 times more costly to repair if detected after production than before

The need for FM

- Famous errors
  - *Ariane 5 (1996)*
    - ESA rocket that exploded on June 4, 1996, less than 40 seconds after its takeoff
    - lost: rocket and cargo valued at 500 million $ + a decade of development estimated to about 7 billion $
    - root causes:
      - [Inquiry Board report] errors in the software of the inertial reference system (SRI)
        - uncaught exception raised by a failed conversion from a 64-bit float to 16-bit integer, due to an overflow
      - [Bertrand Meyer] reuse specification error
        - module reused from Ariane 4, in the absence of a proper specification

The need for FM

- Famous errors
  - *Intel's Pentium processor (1994)*
    - errors in the Floating Point Unit (FPU) instructions for division
    - cost: around 500 million $
  - ***Therac-25 (1985-1987)***
    - medical device used in radiation therapy for cancer patients
    - involved in 6 accidents between 1985 and 1987, with pacients receiving major overdoses of radiation (more than 100 times the intended dose)
    - root causes:
      - bad software design and development practices
        - several coding errors in the control program
        - inappropriate software reuse

The need for FM

- Major SE goal**:** *Enable developers to build systems that operate reliably, despite their complexity*
- Reliability approaches
  - *Testing and simulation*
    - classic approach, well understood, mature and well integrated in industry
    - produce a set of representative data, input it to the sistem/model and compare the outputs to the expected ones
    - drawbacks:
      - software systems are discrete, not continuous and the great majority have infinite value spaces => testing cannot be exhaustive

        „Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence." [Edsger W. Dijkstra]
      - errors detected late in the lifecycle => high debugging costs

What are FM?

- Reliability approaches
  - *Formal Methods*
    - alternative, mathematical approach, gaining acceptance in industry
    - FM = the mathematics of Software Engineering

      „mathematically based techniques for the specification, development and verification of software and hardware systems." [FOLDOC – Free On-Line Dictionary of Computing]

      „ ... mathematically based languages, techniques and tools for specifying and verifying systems" [2]

      „ ... mathematically based techniques and tools for the synthesis (i.e. development) and analysis of software systems" [3]
    - A *formal method* is a *formal language* accompanied by a formal reasoning system
      - formal language = language with a formal syntax and semantics
      - formal reasoning systems examples: theorem proving, model checking, refinement, correctness verification of an algorithm with respect to its formal specification

Why using FM?

- As in other engineering disciplines, appropriate mathematical modeling and analysis can contribute to the correctness of the final product

  „Software engineers strive to be true engineers. True engineers use appropriate mathematics. Herefore, software engineers should use appropriate mathematics. Thus, given that formal methods is the mathematics of software, software engineers should use appropriate formal methods." [Michael Holloway - NASA, 1997]

  - FM **do not** a priori **guarantee** correctness

    ... **but** their use may **increase** the level of correctness!
  - formalizing requirements allows one to discover incompleteness, inconsistencies, ambiguities
  - FM increase system understanding
  - used in the specification and design phases, FM not only allow more flows to be found, but also expose them earlier in the development process
  - FM allow reasoning about complete data space and configurations of a system, rather than samples of these sets
  - FM **do not replace testing**, they **complement** it!
  - Formal verification is the only way to provide assurance that catastrophic failure will not happen
  - FM enable automation
  - FM can lead to lower overall development costs, although the cost of the initial phases increases
  - FM can lead to a clearer documentation

FM in software lifecycle

- Requirements analysis
  - identification of inconsistencies, ambiguities, incompleteness
  - system specification
- Design
  - specification of interfaces
  - design by refinement
- Verification
  - Formal verification
  - Testing & debugging
    - generation of test cases from formal specifications
- Reverse engineering

FM levels

- 4 levels
  0. Mathematical notations & thinking
  1. Formal specification
  2. Formal specification and semi-formal verification
  3. Formal specification and formal verification
- Choice of a level depends on
  - critical nature of the application
  - available resources
    - time, money, skilled people
- Different levels for different components
  - formal specification of the whole system + formal verification of only critical functions

FM's core problem

- To be able to guarantee software behavior following some rigorous approach
- At the core of FM there is the concept of *formal specification* (formal model of the system containing a description of its intended behavior)
- The core problem can be decomposed into two sub-problems

  1. How to enforce the desired behavior at the specification level? (a.k.a. *model validation*)

     - animation, proving properties, transformation

  2. How to obtain, from a specification, an implementation with the same behavior?, or, conversely, given an implementation, how can it be guaranteeed that it has the same behavior as the specification? (a.k.a. *formal relationship between specification and implementation*)

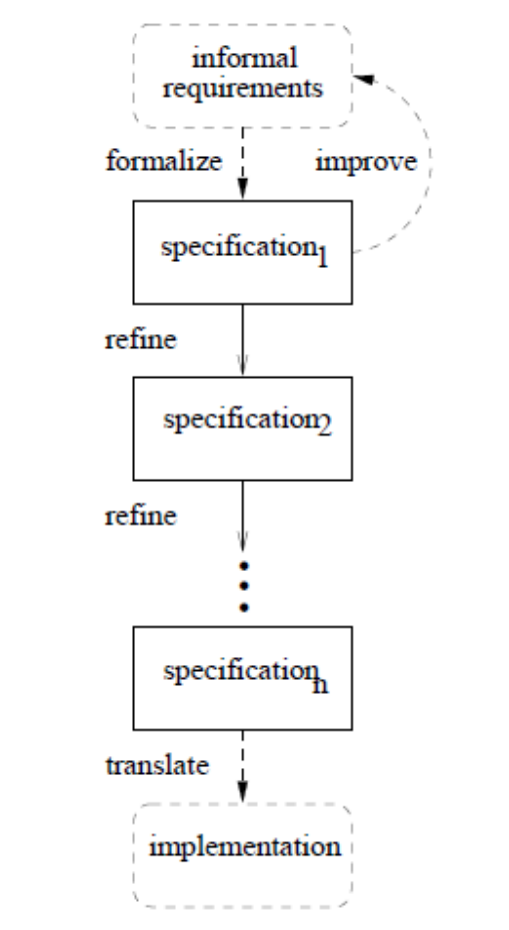
     - implementations derived/refined from specifications or guaranteed correct by construction or verification (formal proof)

Formal specification

- A *specification* is said to be *formal* if it is expressed in a formal specification language (a language with precise syntax and semantics)
- Advantages of formal specifications
  - abstract (refinable)
  - precise
  - enable formal analysis

Abstraction and refinement

- Specifications should be abstract, they should leave out irelevant (implementation-related) details
- A good requirements specification describes *what* a system should do, not *how* it should do it
- As a consequence of abstraction, a specification may have several implementations satisfying it
- For complex products, development takes place by stepwise refinement: at each step a refined specification is built - by introducing new aspects and limiting choices - and verified against the previous one. The last specification is easily translatable into a programming language (code generators may be used).



Precision

- Informal specifications are most often
  - ambiguous
    - formal specifications are unambiguous due to the precise semantics of formal specification languages
  - incomplete
    - the use of a formal language forces the specifier to think deeply and consider all the posible cases
  - inconsistent
    - formal specifications allow mathematical reasoning that uncovers inconsistencies
- However, informal specifications are more intuitive
- Formal specifications should complement informal ones, not replace them

Formal analysis

- It is possible to prove properties in order to *validate* a formal specification, or to *verify* a refinement against it
  - *verification* is the act of investigating whether a product is correct (satisfies its specification)
  - *validation* is the act of investigating whether a specification correctly describes the problem to be solved
- If stepwise refinement is used, verification also takes place stepwise
  - formal specifications allow to mathematically define what it means for a specification to satisfy another specification
  - at each step, it is verified whether the current specification satisfies or not the previous one
- Validation may be performed
  - by stating some properties (mathematical statements) that are expected to be consequences of the specification and then using mathematics to prove that.
  - by testing, if they are executable

Taxonomies

- A specification describes
  - the manipulated data
  - the way it evolves (the operations that transform it)
- According to the focus given to the above aspects, there are two main specification styles
  1. *model-oriented* (a.k.a. *state-oriented*) specifications
     - focus on the operations and how they change the internal state
  2. *property-oriented* specifications – *algebraic/axiomatic*
     - focus on the manipulated data, the way they are related and evolve
- According to the kind of system specified, there are
  - languages for sequencial programs
  - languages for reactive, concurrent systems

Model-oriented specifications

- The corresponding languages are able to describe the internal state of the system and focus on the way operations change that state
- Underlying foundation: discrete mathematics, logic, set/category theory
- Flavors
  - formalisms based on Abstract State Machines (ASM)
    - B method and language (tools: AtelierB, B-Toolkit, ProB, Rodin)
  - formalisms based on set/category theory
    - Z, VDM, B, Alloy, Specware, Charity
  - formalisms based on automata modelling
    - used for reactive, concurrent, communicating systems & protocols
  - modeling languages for real-time systems
    - extensions of the simple automata framework, to be able to handle, time, temperature, inclination, altitude, etc.

- Example: B abstract machine

MACHINE Car_status
SETS
$STATUS = \{sold, available\}$
USES $Cars$
VARIABLES $status$
INVARIANT
$status \in CARS \nrightarrow STATUS$
INITIALISATION $status := \emptyset$
OPERATIONS
$set\_status(x, m) \mathrel{\widehat{=}}$
PRE $m \in STATUS \wedge x \in CARS$
THEN
$status(x) := m$
END
END;

Algebraic specifications

- Based on multi-sorted algebras
  - a collection of data grouped into sets (one set for each datatype)
  - a collection of functions on these sets (functions of the modelled program)
  - a set of axioms specifying the properties of functions
- Abstract away from the algorithms, focus on data and input-output behavior of functions
- CASL, OBJ, Clear, Larch, ACT-ONE

- Example: List datatype

Spec: $LIST_0\ (ELT)$
Extends: Nat_0
Sorts: *list*
Operations:

nil	:	$\rightarrow$ *list* // constructor
cons	: *elt* $\rightarrow$ *list*	$\rightarrow$ *list* // constructor
length	: *list*	$\rightarrow$ *nat*
hd	: *list*	$\rightarrow$ *elt*
tl	: *list*	$\rightarrow$ *list*
append	: *list* $\rightarrow$ *list*	$\rightarrow$ *list*
rev	: *list*	$\rightarrow$ *list*

Axioms: $xs, ys : list,\ x : elt$

$length(nil)$	$= 0$
$length(cons(s, xs))$	$= 1 + length(xs)$
$hd(cons(x, xs))$	$= x$
$tl(cons(x, xs))$	$= xs$
$append(nil, ys)$	$= ys$
$append(cons(x, xs), ys)$	$= cons(x, append(xs, ys))$
$rev(nil)$	$= nil$
$rev(cons(x, xs))$	$= append(rev(xs), cons(x, nil))$

Concurrent, mixed, declarative

- Specification of concurrent systems
  - process algebras: CCS, CSP
- Mixed languages
  - LOTOS
    - extends an algebraic framework with CCS primitives
  - RAISE
    - integrates model-oriented specification, property-oriented specification and process algebra
    - evolution of VDM with a CSP-like concurrency layer
- Declarative modelling
  - logic-based languages
    - modeling approach based on the concept of *predicate*, e.g. Prolog
  - functional languages
    - based on lambda calculus, e.g. languages - Scheme, SML, Haskel, Ocaml & proof assistants – ACL2, Coq, PVS, Isabell, HOL, Agda
  - rewritting systems
    - closed to algebraic languages, axioms replaced by equations, e.g. ELAN, SPIKE

Formal verification

- Traditionally, software verification & validation has been mostly done by code inspection and testing
- The use of a formal specification though enables employment of formal verification techniques
- Advantages of formal verification
  - they consider all situations, unlike testing
  - bugs are discovered earlier than implementation testing
- Disadvantage
  - it is performed on the model, not the system itself, therefore it does not replace testing, but complements it
- State of the art approaches in formal verification
  - theorem proving
  - model checking

Theorem proving

- It is the act of constructing a mathematical proof for a mathematical statement to be true; has its roots in mathematical logic
- According to their rigor, mathematical proofs can be either
  - semi-formal proofs
    - use a mixture of mathematical formulas and natural language
  - formal proofs
    - expressed in a symbolic language (proof language), with a precise syntax
    - a proof consists of a sequence of argumentation steps. A step consists of some premises and a conclusion that can be drawn from premises. The underlying mathematical logic provides a number of proof rules of how to draw conclusions from premises. The conclusion of one step can be used as a premise for the following. Eventualy, this leads to a conclusion which is the statement to be proved

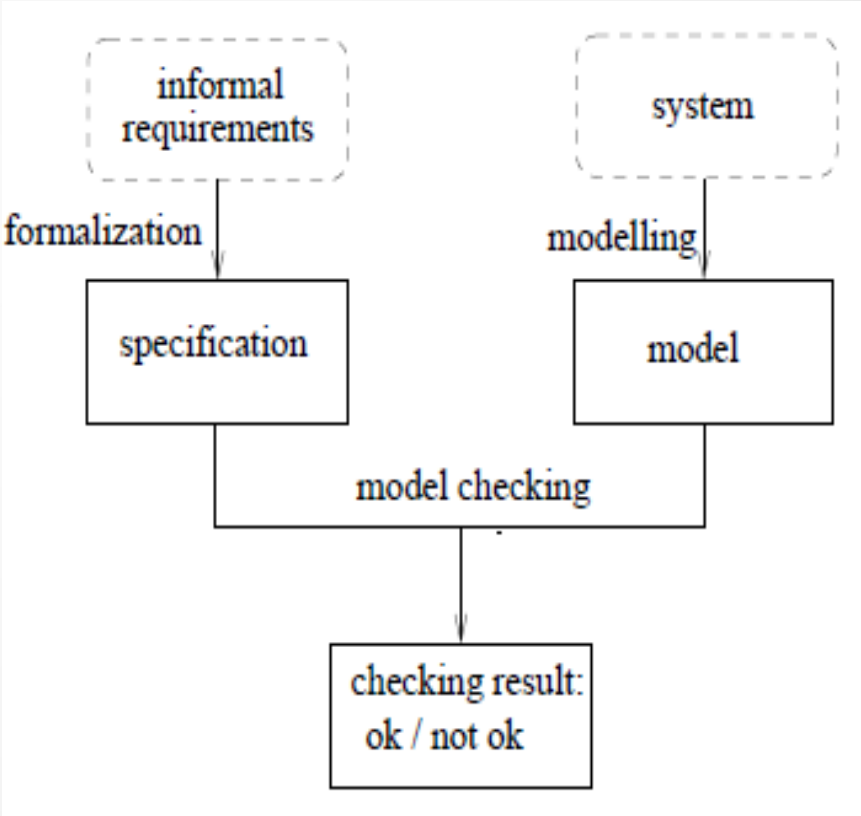
Theorem proving

- Computer tools involved in theorem proving
  - proof checkers
    - the most simple
    - e.g. MetaMath, Mizar
  - interactive theorem provers
    - the most common
    - e.g. PVS, Isabelle/HOL, ACL2, Coq
  - automated theorem provers
    - difficult to create
    - e.g. Prover9, SPASS

Model checking

- It is an automated approach to verify that a model of a (reactive, concurrent) finite state system satisfies a formal requirements specification to the system
- Models are typically expressed in terms of finite-state automata
- Property specifications are expressed using a temporal logic language (e.g. LTL, CTL)
  - safety properties
  - liveness properties
- Model checkers
  - e.g. SPIN, SAL, NuSMV

- Model checking process



Model checking vs. theorem proving

- Model checking

  + fully automatic, easier, faster to use

  - state-space explosion problem

- Theorem proving

  + applicable in cases where model checking cannot be employed

    - state space explosion
    - sistems that cannot be modelled as finite state automata