Formal Methods and Software Engineering

Motivation

Last decades many technical/theoretical results on Formal Methods (FM).

Still not widespreadly used in industry.

Main problems:

– Management attitude

– How to embed FM in Software Engineering ?

Viewpoint: (future) SE manager

Warning

The role of FM in SE is still largely an open question...

– this lecture does not provide simple receipes

– ”checklist” of relevant issues

– managerial creativity needed for decisions !

Main sources:

- experiences reported in papers
- An International Survey of Industrial Applications of Formal Methods (US Dept. of Commerce)

Software Problem

Last 30 years: software crisis/affliction

Growing demands on software (quantity/quality).

Big economic importance !

Still software production a craft, not an engineering discipline. Result:

- development times long
- costs high
- quality low (e.g. many errors)
- management very difficult

Other engineering disciplines matured after a long period of development.

Why not simply copy them ?

Software characteristics

Comparison with traditional engineering disciplines:

Software is

Complex

– many different behaviours (cf. bridge, engine)

Discontinuous

– small design error can have big consequences: software is hard to predict

– impossible to provide "design margins" by "overengineering"

Two counterarguments

Modular redundancy (multiple versions)

Controversial:

– same mistake might be replicated

– redundancy management software complex

Statistics

– quantification of errors difficult

– for small failure rates: experimental approach infeasible

Need for correctness

Design errors eliminated by

- design process
- quality assurance

Software configuration

In the early days: fixation on ”working program”
(still present in non–professionals !)

A working program is just a small part of the software *configuration*, e.g. :

- production plan
- requirements specification
- design
- data structures
- test specification
- manual

Engineering ingredients

Cascade and incremental design *conceptual* frameworks.

Actual practice can be more complicated !

Roughly the following ingredients always occur:

- requirements analysis
- planning
- design
- coding
- testing/validation
- correction/adaptation

What can be the role of formal methods ?

Why formal methods

A formal specification is

precise, unambiguous, structured, consistent, abstract.

Benefits:

- facilitates precise recording/communication of ideas
- enables mathematical analysis (e.g. correctness)
- forces intellectual control of a problem
- bridges the gap between the "informal world" (requirements, design) and the code
- helps clarifying requirements

Important aspects

Some aspects that play a role:

- level of formality
- choice of formal method
- formal analysis
- tool support
- education
- organization of people and process
- organization of design stages
- selected components/properties
- transformational design
- testing
- software metrics
- maintenance

Managerial decisions

Decisions have to be taken for all these aspects!

A good project depends on the right decisions...

Unfortunately there is no formula or receipe – the manager is important !

"A good manager can manage anything" – probably not true for software engineering !

Levels of formality

1. **Loose**
   – English, notations, diagrams and mathematics combined
   – analysis: arguments to persuade reviewers
2. **Formal**
   – specification language (syntax/semantics)
   – analysis: by hand, but using explicit axioms or proof rules
3. **Mechanized**
   – specification language (syntax/semantics)
   – analysis: with the help of automated analysis tools

Selecting levels

Several possibilities:

– Going from "informal" to some level of formality in one step

- small, well-structured problems
- highly trained personnel

– increasing the level of formality

- loose level for exploring and discovering
- formal or mechanized level when problem is clear

– using different levels at different stages or even in parallel

- use loose level to communicate with customers

Choice of FM

Many different formalisms. Some types:

- model oriented (e.g. Z)
- algebraic (e.g. LOTOS)
- property oriented (e.g. logics)
- functional (e.g. ML, Miranda)
- concurrency (e.g. PROMELA, Petri Nets)

This classification is not exhaustive, and formalisms can belong to different types.

Analysis tools

Tools that help in assessing the correctness of a formal specification:

theorem provers/proof checkers
: still much expertise is needed for using them

model checking tools
: check properties on a model of a system, e.g. by exploring the state space

equivalence/preorder checkers
: check whether two specifications are in some correctness relation w.r.t. each other (verification, validation)

simulators
: for animating a specification, often interactively (validation, checking requirements)

Other FM tools

(apart from general CASE tools):

- syntax, type, and consistency checking tools
- natural to formal language tools, knowledge based support
- general analysis tools (e.g. for data flow analysis)
- design support tools (e.g. performing transformations)
- compilers (for coding, or translating into another formalism)
- test generation, selection, and execution tools

Tool decisions

- What tools are needed ?
- How to obtain them ?
  - prototypes, academic research tools
  - commercial tools
  - own development

Tools are very important!

Tool availability may have crucial impact on deciding upon a formalism.

(Still the surveys indicate that also without tools good results can be obtained)

Human resources

Is there sufficient FM know-how in the project?

If not:

- extra training and education (see next sheets)
- hire temporarily some external experts (e.g. for theorem proving)
- make use of external consultants
- delegate activities to a third party (e.g. validation/verification)

Personnel

An analysis should be made of *who* needs *what* skills.

Example:

A **specifier** should be able to **write** a formal spec.

A **developer** should be able to **read** a formal spec in detail.

A **verifier** should be able to **analyze** a formal spec.

A **customer** should be able to **understand** a formal spec.

A **reviewer** should be able to **understand analysis** results.

Education

Age and educational level are important !
(cf. 50 year old programmer with 27 year old math PhD)

Some typical figures:

– discrete mathematics:
course several days

– formal specification (e.g. Z):
course 1 or 2 weeks

– tutoring/consultation in real projects:
attending workshops, hiring consultant during early project phase

Experience shows that after learning a formal specification language, a system developer needs about three months of practice before his skills can be used in real projects...

Planning

Important inputs:

- human resources
- computer/tools resources
- estimates, based on software metrics (see further in this course)

Make use of standard SE planning techniques.

e.g. network planning (tools exist for producing timeline charts and resourse allocation tables)

Important: **risk analysis**. What might go wrong, and how do we act then?

Planning decisions (1)

- What parts of the system will be verified/validated/tested ?
  often impossible to assess the whole system;
  then only those components that are critical or hard to design
- What properties will be verified/validated/tested ?
  those that are critical, or can be effectively dealt with
- What level of abstraction ?
  too abstract: problems disappear
  too concrete: assessment infeasible

Planning decisions (2)

- Who performs the assessment ?
  the specifier, another team, external experts ?
- At what stage in the design process ?
  the earlier a mistake is found, the less costly its removal !

(J. Bowen: a defect removed at service time is 1000 times costlier then at the requirements capturing phase)

Requirements capturing

Errors made at this stage most costly:
assessment very important !

Problems:

- Initially the requirements are always informal, ”in the head of the customer”
  important to involve customers in specification assessment, e.g. by prototyping, simulation/animation, natural language paraphrasing
- For complex systems the requirements will only gradually become clear
  prototyping, incremental design (gradually adding more functionality)

Cleanroom (1)

Cleanroom: software engineering method (IBM)

Different teams:
specification – development – certification

Not tied to a specific formal method. Main point: the development team does not perform debugging or even compilation !

No unit testing !

Motivation:

- debugging often introduces new errors (15% of the cases)
- these errors are deep and hard to find

Cleanroom (2)

Based on SQC: Statical Quality Control as used in e.g. manufacturing industry

- In an assembly line, at several stations statistical measurements are taken
- If any partial product fails, the entire assembly line is stopped, to take care of the production problem

This provides an incentive for doing accurate work.

The idea sounds counterintuitive, but so did touch typing (Dutch: blind typen) when it was introduced !

Cleanroom (3)

Testing: statistical usage testing (based on external system behaviour)

- specify usage probabilities
- derive from this randomly generated tests
- execute test cases, compute quality measures

Quality too low: the process should be improved.

– Success of Cleanroom seems largely based on group responsibilities and discussion of specifications

– Seems to yield improvements in cases where quality is initially low.

Cleanroom not a definite answer to all problems, but certainly contains some interesting ideas

Transformational design

Idea:

- start with a formal specification of the requirements:
  Initial Specification
- stepwisely transform this using correctness preserving transformations:
  Intermediate Specifications
- finally a formal specification is obtained, that is structured in such a way that it is easy to code:
  Final Implementation

Transformational design (2)

Prerequisites:

– a specification formalism that allows alternative synctactical structures (specification styles)

– formal design transformations:

- serve a specific design goal
- preserve correctness (in some precise sense)
- are ideally tool supported

The transformational design approach will be illustrated later in the context of process algebra.

Testing

Testing is an *empirical* validation method to establish/estimate the correctness of implementations and realizations.

- **product testing:**

  validation of the realization wrt the *final* implementation, which serves as the *logical blueprint* for the realization.

- **conformance testing:**

  validation of the realization wrt the *initial* specification, which serves as the *design obligation* (contract, standard) for the realization.

- **refinement testing:**

  validation of an *executable* implementation wrt to its specification, i.e. an empirical test of the validity of the correctness relation concerned.

Testing Strategies

- **black box testing:**

  Only the *external behaviour* of the object (implementation or product) under test can be observed. All internal structure (e.g. the state space) is hidden.

  This typically holds for conformance testing.

- **glass box testing:**

  Both the external behaviour and the internal structure of the object under test can be observed/ are known.

  This typically holds for refinement testing.

- **grey box testing:**

  The external behaviour and some of the internal structure can observed/are known.

  This typically holds for product testing.

FM and testing

Testing within FM framework can have the following advantages:

- formal test validation
  Does a test really test what it is intended for ?
  Is it really a failure to fail the test ?
- test derivation algorithms
  Manual test derivation time consuming; if the design changes, the tests have to change too !
- precise evaluation of test outcomes
  What has caused the error ? How to repair it ?
- test coverage measures
  Quantification of how much of the system behaviour has been covered by a test suite.

Software metrics

Quantative data about the design process important for

- estimation (planning !)
- risk analysis
- scheduling
- tracking and controlling the process

Often such data are absent. It is then impossible to assess the influence of adopting FM in the design process !

Software metrics: *great strategic importance* !

Quantification makes sense only over a long period of time.

Types of metrics

Code-oriented: Kilo Lines Of Code

E.g.:
productivity: KLOC/person-month
quality: defects/KLOC
costs: $/KLOC
documentation: pages/KLOC

problem: too much fixated on the final code !

Alternatives:

– function–oriented metrics
function points: number of inputs, outputs, interfaces etc.

– feature points
general properties, e.g. communication, distribution, reusability

both combined with heuristical weighing factors

Metrics and FM

– fomal specifications are more suitable for quantification than natural language specifications

– such a quantification could be automated (incorporation in CASE environment)

– measurements can be made at an early stage in the design trajectory, facilitating planning

– possibly more intelligent metrics can be defined

Currently research on FM metrics is ongoing

example: structural metrics on Z specifications, e.g. flowgraph–like metrics

Maintenance and FM

Maintenance benefits from FM:

– all components formally specified, so modifications easier

– effects of changes on other components: redo the verification/testing/validation and check the effects

Retrospective specification ("backwards engineering"):

- of just code: nearly impossible
- of informal specifications: difficult, as informal specs usually structured in a messy way (e.g. mixing levels of abstraction) that hampers clear formalization

Safety–critical systems

e.g. nuclear plants, train switching systems, aviation systems

Benefits of FM here widely acknowledged. Design errors should be absent at all costs...

- failure rates higher then 10^{-9} per hour can no longer be empirically established; hard to quantify software errors
- hardware failures cannot be avoided but should be quantified
- the system should be designed such that a reliability model can be constructed

FM are not restricted to safety–critical systems! E.g.....

Embedded software

– controls consumer/industrial products

– usually resides in ROM

E.g. tv (1 Mg), micowave oven, shaver (8 K !), car

Software usually not very complex or safety–critical BUT:

- errors very costly; modifications costly
- errors can usually not be dealt with by adjusting user behaviour
- high replication makes efficiency an important economic issue

previsions are that the software need in this area will grow explosively

Therefore: important role for FM !

Miscellaneous remarks

– coding (from a formal spec) is a well–understood discipline and can often even be reliably automated. (Still often managers panic if at 2/3 of the project there is still no code...)

– Performance analysis should be integrated with FM as early in the design trajectory as possible

– Reviewing important; formal assessment results valuable input to review. (Let the reviewer explain the design to the designers !)

– FM: analysis and specification stages usually longer, later development stages much shorter (since there will be less integration problems and redevelopment)

Industrial acceptation

Starters:

- technology explorations (pilot projects, research projects)
- recognition of difficulties

Boosters:

- difficult technical problems recognized
- successful applications
- contractual requirements

Years 1, 2, 3: exploring methods/applications

Vears 4, 5, 6: serious experimentation, measurements, products

Years 7, 8, 9: developing appropriate tools/processes around successful methods

Concluding remarks

- We have briefly touched many points concerning FM and SE. Not exhaustive !
- Not yet very systematic; more research needed (welcome !)
- Use these notes as a checklist. Be creative !
- Most important point: in the design process, a good formal specification is much more important than working final code !