FMSE: Formal Methods in Software Engineering

1 Formal Methods in Software Engineering

Summary

This document describes a 30 ECTS curricular unit offered by the Logic and Formal Methods Group of Minho University according to the Bologna recommendations (2nd cycle).

This unit stems from more than 20 years of experience at Minho in teaching, researching and applying rigorous methods in the construction of software. Its main aim is to provide a consistent and solid answer to the requirements and challenges of the information age in the near future.

By resorting to the mathematical foundations of modelling, reasoning, programming and testing, this offer bears a special mark in the way future professionals are trained to meet high-quality standards in the design of software solutions to real-life problems.

This unit is complemented by another offer of the same group entitled Programming and Verification*, which addresses complex algorithms and machine-assisted verification, and applications.*

Staff

Luís S. Barbosa, José B. Barros, Manuel Alcino Cunha, José N. Oliveira, Joost Visser.

Contents

1 Formal Methods in Software Engineering **1**
1.1 Context and Aims . . . 3
1.2 Overall Unit Structure . . . 3
1.2.1 Attendance modalities . . . 3
1.2.2 Keywords . . . 4
1.2.3 Required background . . . 4
1.2.4 Learning outcomes . . . 4
1.2.5 Teaching methodology . . . 5
1.2.6 Assessment . . . 5
1.3 Syllabus . . . 5
1.3.1 Formal Methods (FM) . . . 5
1.3.2 Calculus of Information Systems (CIS) . . . 8
1.3.3 Processes and Software Architectures (PSA) . . . 9
1.3.4 Software Analysis and Testing (SAT) . . . 10

1.3.5 Cohesive Project (CP) 11

1.1 Context and Aims

The *Formal Methods in Software Engineering* unit (FMSE unit for short) consists of four 5 ETCS modules on specialized (theory) topics which are linked together by a 10 ECTS lab module which acts as knowledge integration *bus* (Fig. 1). This structure is intended to support the unit's overall aim of ensuring that solid theoretical knowledge finds its way to practice and real-life application.

This strong blend between theory and practice is the one side of this unit's *mission statement*. The other side is captured by the following "stamp", which software professionals should endeavour to be able to stick to their products and services in the future:

This entails the need for knowledge representation notations and languages amenable to mathematical reasoning under a careful balance between *descriptive* power and *calculational* (reasoning) power, thus meeting the $e = m + c$ equation: *engineering* is nothing but *modelling* allied to *calculation*.

As a complement of this equation, the unit is particularly concerned with testing of both models and implementations. The former is better known as *model animation* and requires background in runnable declarative programming notations. Model animation is useful in as-early-as-possible quality control of the intended design, in particular in what concerns adjusting the level of abstraction, filtering childish errors and sorting subtle misinterpretation of requirements. Implementation testing is akin to program analysis and can also be performed on a sound and effective basis, well far way from primitive trial-on-error debug.

As a final target of the FMSE unit we find *software architecture*, a theme which bringing about such important aspects of software design as modularization, objectification, reuse, simulation, and so on. This leads to a *programming-in-the-large*, component-oriented approach to software design which is of enormous relevance nowadays.

1.2 Overall Unit Structure

1.2.1 Attendance modalities

The FMSE unit is offered on a semestral / annual basis depending upon the target students. The annual regime is recommended as default for students wishing to complete two 30ECTS units in one year. The semestral alternative, which is more intensive and focused in time, is more adequate for contiunous education or wherever required by its integration in Erasmus-Mundos projects involving other academic institutions.

The 4+1 module structure of the unit suggests a weekly load of 8T + 8TP + 8P hours, where T stands for theory class, TP for an interactive class and P for labwork.

1.2.2 Keywords

Keywords: Informatics (30 ECTS)

Keywords (ACM Computing Classification System): The 30ECTS of the FMSE unit distribute as follows across the ACM Computing Classification System of the ACM *Computing Curricula* (1998), in lexicographic order:

- Software/SOFTWARE ENGINEERING/Metrics — 5
- Software/SOFTWARE ENGINEERING/Requirements/Specifications — 5
- Software/SOFTWARE ENGINEERING/Software Architectures — 5
- Software/SOFTWARE ENGINEERING/Software/Program Verification — 5
- Software/SOFTWARE ENGINEERING/Testing and Debugging — 5
- Theory of Computation/LOGICS AND MEANINGS OF PROGRAMS/Specifying and Verifying and Reasoning about Programs — 5

1.2.3 Required background

- Experience in programming, including familiarity with declarative programming (logical, functional):
- Background in predicate logic, lambda calculus and discrete mathematics at first cycle level.

1.2.4 Learning outcomes

- Create, analyse, refine, classify, animate, test, transform and calculate with abstract models of requirements in software engineering.
- Transform specifications of complex information systems into efficient implementations on diverse technologies and platforms.
- Model, analyse, classify and transform component interaction patterns, modular strategies (components, objects, services) and software architectural schemes.
- Select and use tools for program analysis

- Perform software quality control and plan / execute projects in software testing.
- Document and justify software design decisions based on accurate software modelling and reasoning.
- Manage software engineering projects in an integrated way from conception to implementation, testing and deployment.

1.2.5 Teaching methodology

- Introduction and explanation of concepts interleaved with case study analysis.
- Classroom group-work on exercises and small case-studies, adding tool support to pen-and-paper work wherever possible.
- Lab assignments and offline group projects on medium scale case-studies under direct supervision of the lecturing staff.

1.2.6 Assessment

Student assessment is targetted at measuring not only individual learning gains but also group-level skills, including integration, determination and perseverance in meeting requirements and surviving adversity.

There is only one final examination paper whose mark is weighted-averaged with continuous assessment of lab work and its deliverables (quality of written reports, professionalism in tool development, oral presentation skills etc.)

1.3 Syllabus

The FMSE unit consists of 4 thematic modules worth 5 ECTS each,

FM Formal Methods

CIS Calculus of Information Systems

PSA Processes and Software Architectures

SAT Software Analysis and Testing

linked together by a laboratorial "bus" of 10 ECTS (the so-called *Cohesive Project*) which ensures experimentation and practical application of the units main learning outcomes. This overall structure is depicted in Fig. 1.

1.3.1 Formal Methods (FM)

Description. The main purpose of formal modelling is to identify properties of real-world situations which, once expressed by mathematical formulæ, become abstract models which can be queried and reasoned about. This often

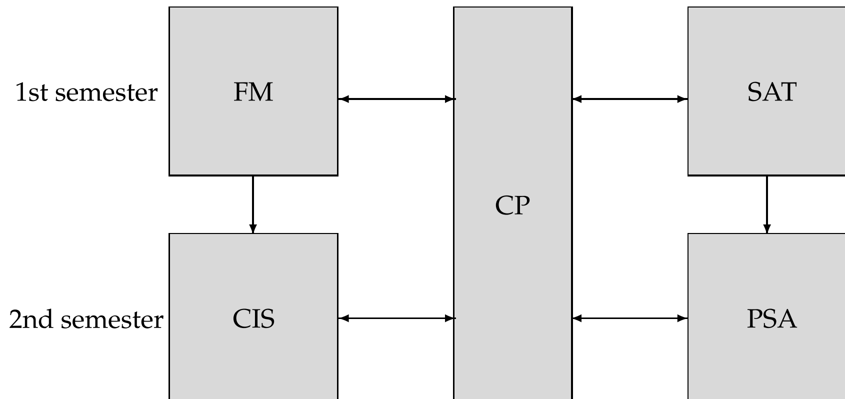

Figure 1: Internal Structure of the FMSE 30ECTS unit.

raises a kind of *notation* conflict between *descriptiveness* (ie., adequacy to describe domain-specific objects and properties, inc. diagrams or other graphical objects) and *compactness* (as required by algebraic reasoning and solution calculation).

Classical *pointwise* notation in logics involves operators as well as variable symbols, logical connectives, quantifiers, etc. in a way which is hard to scale-up to complex models. This is not, however, the first time this kind of notational conflict arises in mathematics. Elsewhere in physics and engineering, people have learned to overcome it by changing the "mathematical space", for instance by moving (temporarily) from the time-space to the s-space in the *Laplace transformation*.

The *pointfree (PF) transform* adopted in this module is at the heels of this old reasoning technique. Standard set-theory-formulated concepts are regarded as "hard" problems to be transformed into "simple", *subsidiary equations* dispensing with points and involving only binary relation concepts. As in the Laplace transformation, these are solved by *purely algebraic* manipulations and the outcome is mapped back to the original (descriptive) mathematical space wherever required.

Note the advantages of this two-tiered approach: intuitive, domain-specific descriptive formulæ are used wherever the model is to be "felt" by people. Such formulæ are transformed into a more *elegant*, simple and compact — but also more cryptic — algebraic notation whose single purpose is easy manipulation.

According to this line of thought, this module structures itself in two levels:

description and calculation. At the former, students learn how to capture informal requirements into abstract models written in VDM-SL notation (SO/IEC standard 13817-1). At the latter, a PF-semantics of VDM-SL is set up which is used in reasoning and calculating with abstract models, eg. in proving invariant properties, in converting abstract models to and from ER-diagrams, etc.

Given its basic nature, this module is central to the whole unit. It binds to the Cohesive Project (CP) via lab sessions on VDM-SL model animation and testing (cf. rapid-prototyping).

Syllabi

- Introduction to high-quality control standards in the software field. Safety-critical systems.
- Formal methods and the formal method life-cycle. Specification versus implementation (and refinement).
- The role of abstraction in formal modelling. Sub-specification and non-determinism.
- Languages for formal specification and systems modelling. From the VDM method to the SO/IEC 13817-1 (VDM-SL) standard. Abstract data models: sets, sequences and mappings. Notation and properties.
- Reasoning about models. Introduction to the "Pointfree-transform". Relations versus functions. Relational calculus. Semantics of the VDM-SL notation expressed in the relational calculus. Galois connections. Point-free reasoning. Taxonomy of binary relations. Wellfoundedness and termination. Structural induction.
- Formal modelling in the IS context. VDM-SL semantics of ER-diagrams.
- Lab sessions: experience with CSK's VDMTOOLS for Quality Software on Schedule. Use of Haskell to animate abstract models.

Learning Outcomes

- Learn what an abstract model of a problem or situation is
- Create, analyse, refine, classify, animate, test, transform and calculate with formal models
- Apply the PF-transform and the relational calculus to the reasoning about abstract models

1.3.2 Calculus of Information Systems (CIS)

Description. This module applies the learning outcomes of its predecessor (FM) to a specific and particularly relevant software engineering problem: that of calculating *correct* implementations from abstract models. As a particular application, this refinement-by-calculation approach is instantiated to the area of database design. In order words, database schema design is regarded as a special case of *"do it by calculation"* data refinement. This provides a calculational alternative to state-of-the-art casuistic practice stemming from set-theoretic "normalization theory".

Another learning outcome of the CSI module has to do with *objectification*, the process of evolution of a purely declarative (eg. functional) model into an object, or abstract state machine (ASM) exhibiting a public API which can be accessed in client-server architectural mode. At the level of model animation, this corresponds to switching from VDM-SL / Haskell to VDM++ / OOHaskell and extensions.

ASM refinement is studied in two steps, as suggested by their levels of service, *dataware* e *middleware*. Concerning the first, emphasis is put on a calculus of data-structures which stems from the relational calculus studied in the FM module. This is supported by locally developed tools such as VooDoMM and 2LT, which interface XLM and VDM with SQL via Haskell. Middleware refinement is calculated in terms of the algorithmic refinement order which reduces nondeterminism and increases definition. This includes derecursivation and calculation of while/for loops from recursive specs.

At lab assignment level, students assemble themselves in groups and carry out the development of applications (typically in the form of a webservice) which stems from a VDM++ model by addition of a client, the server being transformed into an SQL server once prototype animation is over. Linking these experiments with the *Process Calculi and Software Architecture* (PCSA) and *Software Analysis and Testing* (SAT) modules is part of the overall aim of CSI.

Syllabus

- Introduction to program development by calculation. Foundations of data refinement. Abstraction and representation functions/relations. Invertibility. Surjectivity and/or injectivity. Isomorphic data. Data migration using formal methods.

- Data refinement by calculation. Repository of laws for data refinement. Recursion removal theorem. Encoding of recursive polynomial data models as pointer-structures in C/C++ style. On grammar-based representations (XML). Object-oriented representation.

- Algorithimc data refinement. Simulation and reduction of nondeterminism. Deterministic (functional) implementations. Change of real/virtual data-structures. Fusion laws. Hylomorphisms. Derivation of while/for-loops as hylomorphisms.

- Project: from functional specification animation to service prototyping. Objectification: from algebras to objects. Reactive behaviour. API. Client-server arquitecture. Embedding formal models into foreign language contexts. Experience with VDM++ and the VDMTOOLS← API.

Learning Outcomes

- Understand what stepwise refinement of software is.
- Distinguish specifications from implementations and abstractions from representations.
- Transform specs into programs.
- Animate design prototypess in the client-server architecture.
- Calculate relational database schemas from abstract models.

1.3.3 Processes and Software Architectures (PSA)

Description. This module is designed as an introduction to software systems' architecture from two complementary perspectives: the *structure of their interactions* and their *emergent behaviour*. From a methodological point of view it combines a foundational level (essentially based on process calculi) with a lab project integrating both methodological and technological issues (e.g., web-services, coordination middleware, specific platforms over e.g. JAVA or C#).

The first part of the module is concerned with behaviour and interaction (formal) modelling. Particular emphasis is placed on calculi of mobile architectures and dynamic reconfiguration.

On top of such a formal foundation, the second part of the module characterises the discipline of software architecture, emphasising description languages, conceptual frameworks, standards and documentation issues. The module underlines the differences between static (compile-time) software interconnection, oriented towards the application macro structure and resource control, and *dynamic*, run-time software orchestration on top of heterogeneous and global computational platforms. Students are made aware of a subtle, but fundamental, change in the conception of software, more and more regarded as a *service* rather than a *product*.

In such a context, the module characterises a number of concepts (e.g. contract, interface, connector, *glue code*, architectural pattern, configuration, etc) as well as typical architectural styles. In particular, the following paradigms are discussed in detail:

- *object*-oriented architectures,
- *component*-oriented architectures and
- *service*-oriented architectures.

Syllabus

1. Components, services and processes
   - Introduction: software structure *vs* behaviour; construction *vs* observation; inductive *vs* coinductive reasoning.
   - Software components and services as interactive processes.
   - Interaction and process calculi.
   - Software mobility and dynamic reconfiguration. Introduction to π-calculus.
   - Case studies in the software laboratory.
2. Project and Analysis of Software Architectures
   - From components to architectures. Notion of Software Architecture.
   - Architectural description: ADLs (*Architecture Description Languages*); architectural patterns and styles; methodologies.
   - Object-oriented architectures: object composition; models, languages and tools.
   - Component-oriented architectures: component coordination; models and platforms.
   - Service-oriented architectures: service orchestration; models and languages.
   - Architectural documentation, analysis and re-engineering.
   - Case studies in the software laboratory.

Learning Outcomes

- Clear understanding of core notions on software architectural design: interaction, process, component (object, service, resource, ...). Ability to distinguish among different forms of software composition and coordination.
- Ability to identify in real software products different architectural styles and interaction patterns and use them in software projects.
- Ability to create, analyse, test and transform architectural models and styles.

1.3.4 Software Analysis and Testing (SAT)

Description. In this course, students are invited to switch away from the usual perspective on *software* of programmer, designer, or architect, which is geared at *construction*. Instead, they must take the perspective of tester, quality manager, and re-engineer, which is geared at *evaluation*.

They will employ this perspective at all levels of granularity, from individual routines up to *software* portfolios, and with varying levels of rigour, including both heuristic and formal methods. In the accompanying lab excercises, the students will not create software, but rather employ a range of methods and tools to perform evaluation of existing software.

This module does not introduce program analysis and testing as an *ad-hoc* collection of techniques. Rather, it provides a *coherent theoretical framework* in which program analysis and testing are bound together, and treated explicitly as complementary to *software* construction techniques. In this framework, test cases, for example, are the *extensional* counterparts of the *intensional* properties derivable from specifications. Consequently, specification-driven analysis and generation of tests emerges parallel to reverse engineering and calculational refinement of specifications into programs. Such theoretical embedding of program analysis and testing will equip students with a strong and systematic grip on techniques that are commonly used and understood only on ad-hoc basis.

Syllabus

1. Unit testing, Functional testing, Test coverage analysis.
2. Model-driven testing, Test generation, Model checking, Fault injection.
3. Software metrics, Codings standards, Style checking.
4. Program understanding, Reverse engineering.
5. Verification, Security analysis, Risk assessment.
6. Complexity analysis, performance analysis.

Learning Outcomes

- Derive test sets from abstract software models
- Perform complexity and performance analysis.
- Derive metrics and assess quality of code
- Plan and implement software testing projects

1.3.5 Cohesive Project (CP)

Description. This module is exclusively made of practical, lab assignments. It runs parallel to the other four (thematic) modules and integrates with them by accepting requirements such as assignments, tools to use / enhance / develop, etc. It is also regarded as the right place to address projects and themes proposed by industrial partners, much in the same way some (optional) courses already run in the department.

By default, a project is put forward at the beginning of the academic year which is carfully planned so as to fullfill the role of enhancing cohesion of the student's individual learning gains. This is addressed and implemented in a stepwise manner, with increased sophistication along with students increase of background acquired in the thematic modules.

This project does not exclude parallel development of tool support, be it enhancing existing tools or implementing them anew. As has happened in the past, this adds to the LMF group's patrimony which is endowed to future editions of the unit. Current examples are the *UMinho Haskell libraries*, the *Camila Revival* innitiative, etc.

Interoperability with other technologies is another main ingredient of this module's training, meaning that there is no such thing as a *wrong* or *old* technology: all platforms will be under consideration provided one can find a way of integrating with them. We believe this to be important for students to accept reading legacy code and be prepared to reuse it.

Last but not least, students are instilled the practice to measure their productivity in terms of effort/outcome ratios, time cards and other devices which prepare them for the need to measure and control costs in engineering projects.

Learning Outcomes

- Plan, execute and assess software design projects
- Learn how to use a tool effectively
- Get acquainted with different technology integration.
- Learn what to do cooperative work and management of costs.