Formal Methods and their role in Software and System Development

© Riccardo Sisto, Politecnico di Torino
What are Formal Methods?

• Rigorous (mathematical) methods for modelling and analysing (computer-based) systems
  – Formal specification
    • Build a mathematical model of the system
    • Express properties (requirements)
  – Formal verification
    • Check that the model satisfies its requirements
What Kinds of Systems are dealt with by FMs?

- Hardware (Logical circuits)
- Software
  - Formal Methods: a branch of software engineering
- Distributed Systems
- ...

DIISFV  Formal Methods  3
Formal Specification

• **Abstract** mathematical model of the system
  – Unambiguous
  – Consistent
  – Complete

• Examples
  – Combinational circuit -> boolean function
  – Sequential circuit -> FSM
  – Less immediate for software and systems
Validation and Verification

• Validation
  – Check that the model fulfils user requirements
  – Is NOT formal (user requirements are subjective)

• Verification
  – Check internal model consistency
  – Compare model wrt higher-level models
  – Can be formal (as long as models are formal)
Validation and Verification

- Informal Requirements
- Requirement Analysis/Spec.
- Design
- Design spec.
- Implementation
- Requirements spec.
- Validation
- Verification
Why Formal Methods?

• Avoid ambiguous/incomplete/inconsistent descriptions
  – e.g. in standard definitions, in contracts

• Get better confidence of system correctness
  – e.g. give a mathematical proof that a model fulfils some requirements
    • Not possible using only informal techniques
Complexity: The Main Challenge

• Complex models are undecidable
  – e.g. does a C program always terminate?

• Even when decidable, complex models can be too large to be analyzed in reasonable time and space

• Possible ways out:
  – Semi-decision procedures
  – Approximate/non-exhaustive modelling/analysis
  – Correctness by construction (rather than correctness analysis)
## Example: SW Validation and Verification Techniques

<table>
<thead>
<tr>
<th></th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Manual</strong></td>
<td>Review, Inspection</td>
<td></td>
</tr>
<tr>
<td><strong>Computer-Aided</strong></td>
<td>Formal proofs (Theorem proving)</td>
<td>Debugging</td>
</tr>
<tr>
<td><strong>Automatic</strong></td>
<td>Compilation and Type Checking</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data/Control Flow Analysis</td>
<td></td>
</tr>
<tr>
<td></td>
<td>State exploration (Model Checking, etc.)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Formal Proofs

• Some properties of a program (or model) are proved using a formal *deductive system*

• The proof is developed with the aid of a *theorem-prover*
  – Checks correctness of each proof step and proof completeness.
  – Can even search for possible proof steps

• Not so practical
  – Generally not fully automatic
  – Very high level of expertise required
Control/Data Flow Analysis

• Technique used by compilers for optimization (e.g. to look for unreachable code, unused variables, etc)
  – Based on formal program execution model (Control Flow Graph)

• Computes simple information about all possible executions of a program (or model) without executing it and independently of program input:
  – **Data Flow Analysis** => information about the values taken by program variables in various program locations (e.g. initialization status, sign, etc.)
  – **Control Flow Analysis** => information about the (real) control flow (e.g. unreachable code)
Control/Data Flow Analysis

- Theories for classical analyses already available (are those useful for optimization and type checking)
- Frameworks for developing new analyses available
  - But developing new analyses requires high expertise
- Mostly useful for sequential programs
  - Very efficient for single functions (intra-procedural analysis)
  - Much heavier for whole programs (inter-procedural analysis)
  - Extensions for parallel programs available, but complexity is similar to the one of model checking
State Exploration/Model Checking

• Model Checking is a state space exploration technique:
  – Based on a (finite state) state/transition program model (not just a control flow graph)
  – Checks if a user-defined or pre-defined property holds by exploring all possible model execution paths.

• In order to keep the analysis simple/finite
  – Abstraction techniques (=> neglect some details, e.g. the ones that are irrelevant for the property to be checked)
  – Non-exhaustive verification (the check is done only on a subset of all the possible model runs)
Model Checking

• Especially used for concurrent systems
  – Testing and debugging are difficult and give poor coverage (nondeterminism makes it difficult to reproduce situations)
  – Control/data flow analyses for concurrent systems are equally complex but more difficult to implement

• Most promising technique for concurrent/distributed systems
  – Fully automatic (push button)
  – Complexity is exponential in the number of parallel components, but models of significant size can be analysed
How much do FMs cost?

• Quite a lot…
  – Time-consuming activities
  – Expert developers needed

• But cost is progressively mitigated by research
  – Better (faster, more automated) tools
  – Patterns
When is it **worth** using FMs?

- **Critical systems**
  - **Safety critical**: e.g. transportation (aerospace, automotive, etc.), control (industrial plants), …
  - **Mission/security critical** (e.g. e-banking, e-commerce, …)

- **Systems where early error detection saves lots of money** (silicon chips)

- **Note**: FMs are just one technique (fault prevention)
  - Other techniques can be used for improving safety/security:
    - Fault tolerance
Some interesting data on software development

Taken from

- CeBASE (www.cebase.org)

• For large, critical systems, error correction after software delivery may cost up to 100 times more than early error correction

• Errors are not uniformly distributed (80% of defects comes from 20% of modules and 50% of modules are error-free).
Some interesting data on software development

• The most used techniques are those with the best performance-to-cost ratio. In particular:
  – review (removes 60% of defects, but may get to 90% in particular cases)
  – testing
  Note: review and testing typically reveal different error classes, so they are complementary.

• Formal methods are currently used in 10-15% of projects, on critical (very often concurrent or distributed) software
When are FMs **mandatory**?

- **Development standards**
  - Sometimes prescribe the use of FMs for the most critical parts of a system
  - Risk analysis is used to identify these parts

- **Certification**
  - Process certification (product has been developed using certain FMs)
  - Product certification (product has been assessed using FMs)
Examples of Reference Standards

• **IEC 61508** - Functional safety of electrical/electronic/programmable electronic safety-related systems
  – Reference for developing safety-critical embedded systems
  – Especially adopted in the industrial automation and control fields

• **DO-178B** - Software Considerations in Airborne Systems and Equipment Certification
  – Used as guidance for certification of airborne systems by FAA
IEC 61508 Overview

• Applies to safety-related systems

• Each system must be assigned a SIL (Safety Integrity Level), according to the risks associated with its use

• IEC 61508 prescribes a general development lifecycle and SIL-dependent requirements for analysis, design, implementation and operation of systems

• Separate requirements are expressed for hardware and software sub-systems
IEC 61508 SIL Levels

• 4 SIL (safety integrity levels)
  – SIL 1: least dependable
  – SIL 4: most dependable

• A SIL represents a target risk-reduction level which can be achieved by a combination of quantitative and qualitative factors
  – Requirements for probabilistic failures
  – Requirements for systematic failures
Example: SIL Requirements based on Probabilistic Analysis

For Low Demand Mode of operation

<table>
<thead>
<tr>
<th>SIL</th>
<th>Probability of Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIL4</td>
<td>$10^{-5}$ to $10^{-4}$</td>
</tr>
<tr>
<td>SIL3</td>
<td>$10^{-4}$ to $10^{-3}$</td>
</tr>
<tr>
<td>SIL2</td>
<td>$10^{-3}$ to $10^{-2}$</td>
</tr>
<tr>
<td>SIL1</td>
<td>$10^{-2}$ to $10^{-1}$</td>
</tr>
</tbody>
</table>
Software Requirements

• Qualitative factors
  – Software development lifecycle
  – SIL-dependent Techniques and Measures
IEC 61508
Software Safety Lifecycle

Diagram showing the software safety lifecycle with steps including:
- 9.1 Software safety requirements specification
  - 9.1.1 Safety functions requirements specification
  - 9.1.2 Safety integrity requirements specification
- 9.2 Software safety validation planning
- 9.3 Software design and development
- 9.4 PE integration (hardware/software)
- 9.5 Software operation and modification procedures
- 9.6 Software safety validation

To box 12 in figure 2 of IEC 61508-1
To box 14 in figure 2 of IEC 61508-1
IEC 61508
Software Safety Lifecycle
Examples of Recommended Techniques and Measures

Software Safety Requirements Specification

<table>
<thead>
<tr>
<th>Technique/Measure*</th>
<th>Ref.</th>
<th>SIL1</th>
<th>SIL2</th>
<th>SIL3</th>
<th>SIL4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Computer-aided specification tools</td>
<td>B.2.4</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>2a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Semi-formal methods</td>
<td>Table B.7</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>2b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Formal methods including for example, CCS, CSP, HOL, LOTOS, OBJ, temporal logic, VDM and Z</td>
<td>C.2.4</td>
<td>---</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
</tbody>
</table>

NOTE 1 – The software safety requirements specification will always require a description of the problem in natural language and any necessary mathematical notation that reflects the application.

NOTE 2 – The table reflects additional requirements for specifying the software safety requirements clearly and precisely.

* Appropriate techniques/measures shall be selected according to the safety integrity level. Alternate or equivalent techniques/measures are indicated by a letter following the number. Only one of the alternate or equivalent techniques/measures has to be satisfied.
Examples of Recommended Techniques and Measures

Software Design and Development

<table>
<thead>
<tr>
<th>Technique/Measure*</th>
<th>Ref</th>
<th>SIL1</th>
<th>SIL2</th>
<th>SIL3</th>
<th>SIL4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a Structured methods including for example, JSD, MASCOT, SADT and Yourdon</td>
<td>C.2.1</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>1b Semi-formal methods</td>
<td>Table B.7</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>1c Formal methods including for example, CCS, CSP, HOL, LOTOS, OBJ, temporal logic, VDM and Z</td>
<td>C.2.4</td>
<td>---</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>2 Computer-aided design tools</td>
<td>B.3.5</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>3 Defensive programming</td>
<td>C.2.5</td>
<td>---</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>4 Modular approach</td>
<td>Table B.9</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>5 Design and coding standards</td>
<td>Table B.1</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>6 Structured programming</td>
<td>C.2.7</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>7 Use of trusted/verified software modules and components (if available)</td>
<td>C.2.10 C.4.5</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
</tbody>
</table>

* Appropriate techniques/measures shall be selected according to the safety integrity level. Alternate or equivalent techniques/measures are indicated by a letter following the number. Only one of the alternate or equivalent techniques/measures has to be satisfied.
Examples of Recommended Techniques and Measures

Software Verification

<table>
<thead>
<tr>
<th>Technique/Measure*</th>
<th>Ref</th>
<th>SIL1</th>
<th>SIL2</th>
<th>SIL3</th>
<th>SIL4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  Formal proof</td>
<td>C.5.13</td>
<td>---</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>2  Probabilistic testing</td>
<td>C.5.1</td>
<td>---</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>3  Static analysis</td>
<td>B.6.4</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td></td>
<td>Table B.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4  Dynamic analysis and testing</td>
<td>B.6.5</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td></td>
<td>Table B.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5  Software complexity metrics</td>
<td>C.5.14</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>
Examples of Recommended Techniques and Measures

Table B.8 – Static analysis
(referenced by table A.9)

<table>
<thead>
<tr>
<th>Technique/Measure*</th>
<th>Ref</th>
<th>SIL1</th>
<th>SIL2</th>
<th>SIL3</th>
<th>SIL4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Boundary value analysis</td>
<td>C.5.4</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>2 Checklists</td>
<td>B.2.5</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>3 Control flow analysis</td>
<td>C.5.9</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>4 Data flow analysis</td>
<td>C.5.10</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>5 Error guessing</td>
<td>C.5.5</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>6 Fagan inspections</td>
<td>C.5.15</td>
<td>---</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>7 Sneak circuit analysis</td>
<td>C.5.11</td>
<td>---</td>
<td>---</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>8 Symbolic execution</td>
<td>C.5.12</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>9 Walk-throughs/design reviews</td>
<td>C.5.16</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
</tbody>
</table>

* Appropriate techniques/measures shall be selected according to the safety integrity level.
Common Criteria (CC)

- Information Technology Security Evaluation Standard
- Fusion of similar national standards (Canada, France, Germany, Holland, Great Britain, United States)
- Also standardized by ISO (ISO 15408)
- Standard documents can be downloaded from the site
  
  www.commoncriterion.org
Common Criteria: History


ITSEC 1991

UK Conf. Levels → Canadian Criteria 1993

German Criteria

French Criteria
Common Criteria: Objectives

• A common standard reference for evaluating/certifying IT system security.

• “Confidence in the security of a product, system or service is very much a state of mind. The CC can be used to build such confidence by providing a means of quantifying or measuring the extent to which security has been assessed” (CC User Guide).
Common Criteria: Structure

• Part 1: Introduction and General Model
  – general concepts and principles of IT security evaluation, general model of evaluation, constructs for writing high-level specifications of products and systems.

• Part 2: Security Functional Requirements
  – standard way of expressing security functional requirements

• Part 3: Security Assurance Requirements
  – standard way of expressing assurance requirements
General Approach

• Confidence in the security of a product can be achieved by means of actions taken during:
  – development
  – evaluation
  – operation

• CC do NOT define
  – a particular development process (but they refer to typical development phases)
  – a particular evaluation methodology
The Context of an Evaluation

Evaluate

Evaluation Criteria (the CC)

Evaluation Methodology

Evaluation Scheme

Final Evaluation Results

Approve/Certify

List of Certificates/Register
The Evaluation Process

- Security requirements (PP and ST)
- Develop TOE
- TOE and Evaluation Evidence
- Evaluate TOE
- Evaluation Results
- Operate TOE

Evaluation criteria
Evaluation Methodology
Evaluation Scheme
Feedback
Security Requirements Classification
Functional Requirement Samples

- Class “Identification and authentication”
  - Family “User authentication”
    - FIA_UAU.1 Timing of authentication, allows a user to perform certain actions prior to the authentication of the user’s identity.
    - FIA_UAU.2 User authentication before any action, requires that users authenticate themselves before any action will be allowed by the TSF.
    - FIA_UAU.3 Unforgeable authentication, requires the authentication mechanism to be able to detect and prevent the use of authentication data that has been forged or copied.
  - ...

Formal Methods
Evaluation Assurance Levels (EAL)

- EAL1 - functionally tested
- EAL2 - structurally tested
- EAL3 - methodically tested and checked
- EAL4 - methodically designed, tested and checked
- EAL5 - semiformally designed and tested
- EAL6 - semiformally verified, designed and tested
- EAL7 - formally verified, designed and tested
The High Levels

• **EAL5** permits a developer to gain maximum assurance from security engineering based upon rigorous commercial development practices supported by moderate application of specialist security engineering techniques.

• **EAL6** permits developers to gain high assurance from application of security engineering techniques to a rigorous development environment in order to produce a premium TOE for protecting high value assets against significant risks.

• **EAL7** is applicable to the development of security TOEs for application in extremely high risk situations and/or where the high value of the assets justifies the higher costs. Practical application of EAL7 is currently limited to TOEs with tightly focused security functionality that is amenable to extensive formal analysis.
FM prescriptions in EAL7

- EAL7 requires an extended use of formal methods, especially in the class that refers to development methodologies (Development class):
  - **Functional specification**: “The functional specification shall describe the TSF and its external interfaces using a formal style, …”
  - **High level design**: “The presentation of the high-level design shall be formal.”
  - **Low level design**: “The presentation of the low-level design shall be formal.”
  - **Representation correspondence**: “For those corresponding portions of representations that are formally specified, the developer shall prove that correspondence”.
Future Trend

• The role of FM in system and software development is increasing:
  – System and software are more and more critical (safety criticality, security criticality, etc.)
  – System and software complexity is increasing
  – Some concurrency-related errors are likely to go undetected in reviews, testing and simulation
  – Formal techniques are making much progress (model checking in particular)
  – Research is working towards making FMs user-friendly (and then less expensive…)
State of the art (model checking)

• Model checking is a PSPACE hard problem
  ⇒ Scalability towards large problems is quite limited

• But we should consider that:
  ▪ It is possible to \textit{gradually} pass from exhaustive verification to approximate/partial verification
  ▪ The time when only toy-examples could be verified have definitely passed
A significant example

Memory and Time requirements for verifying a telephone switch (Bell Labs)

Memory (Mb)

Time (s)

required to solve problem

available to solve problem