8 Outlook

This final chapter of the lecture covers a number of topics that are related to the practical relevance of formal specification.

8.1 IT Security and Formal Specification

8.1.1 IT Security Criteria

In the last 20 years, much effort has been spent by governmental organisations of many countries in order to advance the state of information technology security. IT security covers here the following broad areas of properties of a computer system:

- Correct functionality
- Reliability and availability
- Data confidentiality
- Data integrity

In order to make these rather vague terms measurable, standards for more concrete criteria were developed, which are to be used in a certification process for IT products. For formal specification of software, the important aspect is that formality is officially required in these specifications for some “evaluation assurance levels”.

Historically, the main steps were as follows:\footnote{1}{For more information, see the following Web addresses: http://www.bsi.de (German national information), http://www.itsec.gov.uk (UK national information), http://www.nist.gov/cc (US and international information, Common Criteria).}

- The US Department of Defense published the first set of criteria in 1983 as the “Trusted Computer Security Evaluation Criteria (TCSEC)”, more popularly known as the “Orange Book”. Follow-up efforts led to the “US Federal Criteria”, but these were never formally adopted.

- During the 1980s, four European countries (France, Germany, United Kingdom and the Netherlands) produced versions of their own national criteria. In Germany, the so-called “IT-Sicherheitskriterien” were published in 1989, by the “Bundesamt für Sicherheit in der Informationstechnik (BSI)”.}
• The four national criteria sets in Europe were harmonised and published as the “Information Technology Security Evaluation Criteria (ITSEC)”. The current issue, version 1.2, was published by the European Commission in June 1991. In 1993, it was followed by the “IT Security Evaluation Manual (ITSEM)”, which specifies the methodology to be followed when carrying out ITSEC evaluations.

• In the 1990s, Europe (specifically the four countries with national standards), USA and Canada started a project to harmonise their security criteria with the goal of an international standard. This harmonisation was called the “Common Criteria (CC)”. Version 1.0 of the CC was published in 1996 and version 2.0 in December 1997.

• CC version 2.0 was submitted to the Joint Technical Committee 1 of the ISO (International Organisation for Standardization) and the IEC (International Electrotechnical Commission). In June 1999, the ISO accepted this text as an International Standard (IS). So the CC are now renamed to IS 15408 (parts –1, -2 and -3), which will be officially published in fall 1999. Afterwards this standard will become valid also as a national standard worldwide (in Germany as a DIN standard).

As a warning, it should be noted that these standards are extremely complex (and bureaucratic) in their terminology and structure

8.1.2 Evaluation Assurance Levels (EAL)

The Common Criteria define 7 different levels for measuring the criteria. An evaluation usually is carried out for a specifically defined EAL, where higher EALs mean stronger security requirements.

The following table gives the EALs of the CC plus the corresponding terms from the older criteria sets:

<table>
<thead>
<tr>
<th>Common Criteria (IS 15408)</th>
<th>Brief definition</th>
<th>US TCSEC</th>
<th>European ITSEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAL0</td>
<td>(minimal protection)</td>
<td>D</td>
<td>E0</td>
</tr>
<tr>
<td>EAL1</td>
<td>functionally tested</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EAL2</td>
<td>structurally tested</td>
<td>C1</td>
<td>E1</td>
</tr>
<tr>
<td>EAL3</td>
<td>methodically tested and checked</td>
<td>C2</td>
<td>E2</td>
</tr>
<tr>
<td>EAL4</td>
<td>methodically designed,</td>
<td>B1</td>
<td>E3</td>
</tr>
<tr>
<td>EAL</td>
<td>Design and Verification</td>
<td>Result</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-------------------------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>semiformally designed and tested</td>
<td>B2 E4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>semiformally verified design and tested</td>
<td>B3 E5</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>formally verified design and tested</td>
<td>A1 E6</td>
<td></td>
</tr>
</tbody>
</table>

The last entry of this table relates to formal specification and verification. This level is described as follows:

“For an EAL7 evaluation the formal model is supplemented by a formal presentation of the functional specification and high level design showing correspondence. Evidence of developer “white box” testing and complete independent confirmation of developer test results are required. Complexity of the design must be minimised.”

The detailed criteria for the design and development of the system prescribe that:

“A formal specification is written in a notation based upon well-established mathematical concepts, and is typically accompanied by supporting explanatory (informal) prose. These mathematical concepts are used to define the syntax and semantics of the notation and the proof rules that support logical reasoning. The syntactic and semantic rules supporting a formal notation should define how to recognise constructs unambiguously and determine their meaning. There needs to be evidence that it is impossible to derive contradictions, and all rules supporting the notation need to be defined or referenced.”

For formal software specification, this means that:

- a complete formal specification has to be given,
- formal consistency proofs have to be carried out,
- verification of implementations against the formal specification have to be carried out.

Given the current state of the art of software specification, it can be stated that

---

2 From: Common Criteria Evaluation Board: Common Criteria – An Introduction. (This brochure is available for download at the lecture home page.)

3 ISO/IEC 14408-3:1999(E), p. 97
• the technology for consistency proofs is well-developed only for relatively limited specification languages, and that
• consistency proofs and verification require very high investment in training of developers and development time.

8.1.3 Economics of EAL7-Level Software

It is still not proven that the additional effort spent for formal specification and verification actually pays back over the lifetime of a software system. Some scientific studies provide results which point into this direction, however industry is not convinced of the economics of formal specification. So current state of practice is as it is stated in a guidebook on the Common Criteria:

“EAL7 represents an achievable upper bound on evaluation assurance for practically useful products. It should only be considered for experimental application to all but conceptually simple and well-understood products. EAL7 is applicable to the development of specialised security products, for application in extraordinarily high-risk situations which justify the extraordinary additional costs. Practical application of this level is currently limited to products with tightly focused security functionality which is amenable to formal analysis.”

Some figures from one of the most advanced projects on formal software verification may be interesting in this context.

The project “KIV” (Karlsruhe Interactive Verifier) is officially approved by the BSI for E6 certification in Germany. The largest application that was fully verified was a disposition system for radio stations of the following size:

• 5000 lines of formal specification
• 7000 lines of code
• 2 person years of verification effort

The KIV group generally estimates that 2000-3000 of lines of code per person year is the currently achievable throughput in formal verification.

---

However, there are also experience reports in the literature\(^6\) which give an estimate of 200-400 lines of fully formally developed code per person year. The difference seems to be caused by the very sophisticated and automated tool KIV. For comparison: The average productivity in traditional software construction is estimated at around 10000 lines of code per person year\(^7\).

So it is possible or even likely that formal specification and verification becomes economically feasible for highest-quality software projects in the mid-term future, assuming that more powerful tools become available.

A serious problem in the industrial uptake of formal methods is the immaturity and diversity of tools. Moreover, many experiments were carried out in the industry at a time where virtually no tool support existed for formal methods, and this experience led to very serious prejudices against formal methods\(^8\). The recent developments around OCL may show a way for a “fresh approach” to formal methods for the industry.

### 8.1.4 Social Responsibility

The community of engineers and scientists in any technical discipline has a special responsibility for ensuring that negative effects of their technology to the health of people, to the natural environment and to economies are limited as far as technically possible. Many engineering societies have established their own “code of ethics” which covers this commitment among others.

For an engineer, this means to apply the “best practices” of his/her discipline to the actual development tasks he/she is working on. For safety-critical software, best practice may in some cases be formal specification and/or verification.

For a scientist, there is an obligation to advance the state of the art in order to achieve the mentioned goal in a better way. It is obvious that

---


research in formal methods for software specification and development is necessary from this point of view, although progress in this area seems to be rather slow compared to other disciplines of computer science.

8.2 Hybrid Formal/Informal Development of Software

There are ways how formally specified and/or verified software can be combined with traditionally produced software. There are at least three directions for such attempts:

- Using formal specifications at system runtime
- Using formal specifications for test case derivation
- Combining formally verified components of a system with less reliable components

8.2.1 Formal Specifications at Runtime

Formal specifications are often difficult to analyse on model level, but can be effectively checked on instance level. The most prominent example of a specification language designed for this usage is OCL.

From a formal specification, assertions can be derived which are inserted into the actual program code, like the following schema:

```java
void assertP () {
    if not P
        throw new AssertException("P violated");
}
```

There are various strategies how assertions can be used, e.g.

- Checking pre- and postconditions at method/procedure invocation
- Checking invariants after any method/procedure invocation
- Checking invariants at specific “checkpoints”
- Using pre- and postconditions in automated test environments

This usage avoids verification and instead augments testing by formally based checks. Please note that there is no need that the formally specified conditions cover the full functionality of the system. For instance, an invariant may just check a very important data integrity condition on a few variables or objects, but ignore most of the system complexity.
Example:

As a small example of the current state of the art, a few screenshots from the "Dresden OCL Toolset"\(^9\) may be helpful. Here is a version of the UML tool *ArgoUML* showing a special editor for OCL constraints (invariants, in this case).

After generating the code skeleton for this very small model, a simple Java file is created (which is available from the Web pages of the lecture under the name *Person1.java*). It looks very much like the code skeletons generated from other CASE tools. The main difference is that the file contains a special Java comment line which includes the OCL constraint text.

\(^9\) For more information see http://www-st.inf.tu-dresen.de/ocl
After completing the code skeleton with method bodies, we have an compilable and executable Java program (*Person2.java*). The sample run below shows that is possible, however, to create objects which violate the constraints (an invariant was provided that married persons should be at least 18 years old).

Now the *OCL Compiler* tool is applied to actually create assertions from the constraints. The tools is invoked from a standard command shell, since this is the stage of development, where developers not necessarily use CASE tools anymore. The resulting file (which is not intended to be understood by human readers) is available as *Person3.java*. 
When compiling and executing the enhanced program, we get a runtime error message from the constraint violation.

End of example

8.2.2 Formal Specifications for Test Case Derivation

Formal specifications provide a framework to systematically derive test cases from specifications. The basic idea is that a class of test cases is described by a specification which covers only a part of the input/output relations, so it is an abstraction of the original specification. This idea was developed in detail for Z specifications\(^\text{10}\) but it is transferable to other languages.

Let \(C\) be the given specification, and \(T_1, \ldots, T_n\) be the specification of the test cases. Then several criteria for tests can be specified (and verified) formally, e.g.

- **Soundness:**
  \[
  \forall i \in \{1, \ldots, n\} \cdot C \Rightarrow T_i
  \]

- **Completeness:**
  \[
  T_1 \land \ldots \land T_n \Rightarrow C
  \]

Also other conditions, like independence of test cases, can be treated in such a framework very precisely.

---

Please note that the proof of soundness and completeness deals with classes of test cases, and not individual test cases. Only in very special cases, a finite set of test cases can actually prove the correctness of the implementation.

8.2.3 Formally Verified Components

A software architecture for a safety-critical system can be designed in such a way that it combines formally verified components and traditionally built components. In the following, two examples of architectural patterns for this purpose are described.

Insulation Layer

The insulation layer architecture is a variant of the well-known layered architecture where a special layer is inserted between trusted (formally verified) and untrusted components. Trusted parts are shown in grey.

![Insulation Layer Diagram]

The purpose of the insulation layer is to ensure that all calls to the trusted part of the system are according to the required preconditions. For this purpose, e.g. assert statements derived from the precondition may be used. Any other interaction of the untrusted part with the trusted part is prohibited.

This architecture is particularly useful for enforcement of data integrity.

Simplex Architecture

The Simplex architecture\(^\text{11}\) is suggested for general use in safety-critical software; it is not directly connected to formally verified components but it can be applied in this context.

\(^\text{11}\) See: [http://www.sei.cmu.edu/activities/simplex/](http://www.sei.cmu.edu/activities/simplex/)
The basic concept of the Simplex architecture is the replacement unit. A mechanism is realised which aims at the replacement of a software component with another component, the replacement unit. Replacement is done after one has gained some trust in the functionality of the new unit by comparing its output with the output of a trusted unit. This architecture requires a parallel execution of two units for the same functionality.

An example for a trusted component may be a formally verified, but very simple and therefore inefficient implementation of an algorithm, whereas the untrusted component may be a highly optimised and efficient implementation of the same algorithm, which is too complex to verify formally. In such a case, the simplex architecture for some time invokes the untrusted component in parallel to the trusted component but uses the output of the trusted component only. The output of the untrusted component is just used for comparison with the correct output. After some time or after a number of tests with sufficient coverage, the whole composite component is replaced by the untrusted component.

The advantages of such an approach are that it can be applied equally well to component which are trusted for another reason than formal verification, and that it is well suited to embedded systems where testing under realistic conditions is rather difficult. (An example, consider the Ariane 5 disaster, where software failed which was never tested in its actual hardware environment.)
8.3 Seven myths of formal methods

There is a paper\textsuperscript{12} by Anthony Hall (Praxis Systems) that summarises very well a number of common misconceptions about formal methods. These “seven myths” provide a good conclusion to this lecture.

8.3.1 Myth 1: Formal methods can guarantee that software is perfect.

This is not the case, since

- specifications may be incorrect
- also formal proofs may contain errors
- underlying theory may be erroneous or inadequately applied
- actual behaviour of programming languages, operating system or hardware may deviate from the abstract expectations

8.3.2 Myth 2: Formal methods are all about program proving.

This is not the case. Writing formal specifications is a precondition to formal verification. Sometimes, formal specification alone is sufficient to improve the quality of software.

8.3.3 Myth 3: Formal methods are only used for safety-critical systems.

This is only partially true. Formal methods are well justified for safety-critical systems, but large application cases of formal methods are known which are not safety-critical. An example is standard software used in many applications: IBM, for instance, spent the effort to formally verify its CICS transaction monitor system\textsuperscript{13}.

8.3.4 Myth 4: Formal methods require highly trained mathematicians.

In fact, mathematics stays much in the background when actual specifications are written. However, special skills are required to effectively apply formal methods. Close integration into semi-formal notations and avoidance of mathematical notation, as done in OCL, will make formal specifications accessible for a larger group of engineers.


\textsuperscript{13} C.J. Nix, B.P. Collins, The Use of Software Engineering, Including the Z Notation, in the Development of CICS. Quality Assurance, September 1988, 103-110.
8.3.5 Myth 5: Formal methods increase the cost of development.

In fact, there is not much evidence yet whether this is true or not. Anthony Hall claims (in 1990!) that formal methods already decrease the development cost. The central argument is here that formal specifications help in reducing the cost for testing and maintenance, which are much higher than the actual development cost in traditional development.

8.3.6 Myth 6: Formal methods are unacceptable to users.

There are ways to make formal specifications useful also for users with no computer science knowledge at all. These are the same techniques that have to be applied to highly sophisticated semi-formal notations like UML:

- Paraphrase specification in natural language
- Demonstrate consequences of the specification
- Animate the specification

8.3.7 Myth 7: Formal methods are not used on real, large-scale software.

There are more formal methods applications than publicly known. In safety-critical systems, including military systems, the used technology is often kept secret and therefore not known. Moreover, formal methods have started to slowly diffuse into traditional software engineering, as can be seen e.g. from the notation UML, and in particular OCL.