Formal Methods of Software Engineering

Robert Dewar and Amir Pnueli
Course: G22.3033.007

Thursdays, 5-7 PM

Copies of presentations and Lecture Notes will be available at
http://cs.nyu.edu/courses/fall01/G22.3033-007/index.htm
Introduction

Software Engineering is a collection of techniques and methodologies which enable programmers and system designers to construct large software systems in a systematic, effective, and reliable manner.

They are particularly useful for projects which

- Involve large teams of designers/programmers.
- Expected to last over long periods, will require support and maintenance, and may give rise to multiple cycles of evolution and versions.

They are absolutely essential for systems which are identified as

- Safety Critical, i.e., systems where a single failure may result in loss of human life, or
- May result in an unacceptable financial loss.
Software Engineering Approaches

Every SE methodology is based on a recommended development process proceeding through several phases which can be roughly listed as:

- **Analysis** — Identification of the essential system requirements and sometimes the available or desired architecture on which it should be implemented.
- **Specification** — An integrated formulation of what the system should do — functional and non-functional.
- **Design** — Deciding how the system should achieve its goals — identification of a hierarchy of modules and their expected interface specification.
- **Coding** — the actual programming.
- **Unit-Testing** — Testing for functional correctness at a module level.
- **Integration** and **System-Testing** — Testing on the actual architecture.
- **Maintenance**.
Different Variants and Concerns

Earlier formulations viewed the process as proceeding **linearly** – the **waterfall** sequence. Each phase may lead to additional iterations of preceding phases.

A more recent view is the **V-shaped** process:
Moving from Verbs to Nouns

A development methodology which was popular in the 70/80’s was Yourdon’s Structured Analysis approach. It based the specification of a system on a functional decomposition of its activity.

A possible guideline to its application could be:

From a textual description of the system, extract a process for each verb.

The more recent object oriented methodology, can be applied by:

From a textual description of the system, extract an object for each noun.

The difference is not only grammatical, it was also motivated by the difficulties inherent in restructuring a conceptual specification into a design.

Current trends are towards executable specifications and their automatic translation into running code.
Formal Methods

The Formal Methods approach to software construction is based on viewing a program and its execution as mathematical objects and applying mathematical and logical techniques to specify and analyze the properties and behavior of these objects.

The main advantages of the formal approach to software construction is that, whenever applicable, it can lead to an increase of the reliability and correctness of the resulting programs by several orders of magnitude. Again, the main potential beneficiary of such an increase is the class of safety critical systems.
The Questions Asked by Formal Methods

Formal methods deal with two distinct descriptions of the same system. A more abstract description $S$, called a specification, and a more detailed (concrete) description $I$, called an implementation. The main questions asked are:

- **Verification** — Given $S$ and $I$, validate that $I$ is a correct implementation of $S$ or, more generally, that $S$ and $I$ are compatible.
- **Synthesis** — Given one of the descriptions, produce the other at the appropriate level of abstraction. Usually, we are given $S$ and asked to construct $I$. However, in some reverse engineering applications, we may be asked to produce a specification $S$ for a given implementation $I$.

These questions can be asked in

- A **Two Languages** Framework — The specification is presented in a specification language, usually a logic or an equivalent declarative formalism. The implementation is presented in a programming-like language.
- A **Single Language** framework — $S$ and $I$ are presented in the same language. We then talk about establishing refinement, abstraction, equivalence, etc. This framework can support a multistep development process:
  $$S_1 \sqsubseteq S_2 \sqsubseteq \cdots \sqsubseteq S_n = I$$
# A Scandalously Incomplete Classification of Formal Methods

<table>
<thead>
<tr>
<th>Type</th>
<th>Logical</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>VDM, Z, the B-method, PVS</td>
<td>Larch, OBJ</td>
</tr>
<tr>
<td>Reactive/Concurrent</td>
<td>LTL, CTL, TLA, model-checking,</td>
<td>CSP, CCS, the $\pi$-calculus</td>
</tr>
<tr>
<td></td>
<td>deductive verification.</td>
<td></td>
</tr>
</tbody>
</table>
So, How Successful is the Application of Formal Methods to System Development?

- Much more successful in the development of hardware. Many of the large semi-conductors manufacturers: Intel, IBM, AMD, use formal verification (mainly model checking) as part of their development process.
- Partially, in some large software projects.

One of the expected benefits of studying formal methods is the acquired state-of-mind by which every software component should have its own well defined interface specification, independently of whether we attempt to formally verify it, or even write it down explicitly.
An Example of A Partially Formal Approach to the Development of Safety Critical Systems

Developing a methodology and a tool suite supporting the development of safety critical real-time embedded systems.