Formal methods

3.1 INTRODUCTION: In computer science, specifically software engineering and hardware engineering, **formal methods** are a particular kind of mathematically based techniques for the specification, development and verification of software and hardware systems.[1] The use of formal methods for software and hardware design is motivated by the expectation that, as in other engineering disciplines, performing appropriate mathematical analysis can contribute to the reliability and robustness of a design.[2]

Formal methods are best described as the application of a fairly broad variety of theoretical computer science fundamentals, in particular logic calculi, formal languages, automata theory, and program semantics, but also type systems and algebraic data types to problems in software and hardware specification and verification.[3]

Taxonomy

Formal methods can be used at a number of levels:

Level 0: Formal specification may be undertaken and then a program developed from this informally. This has been dubbed *formal methods lite*. This may be the most cost-effective option in many cases.

Level 1: Formal development and formal verification may be used to produce a program in a more formal manner. For example, proofs of properties or refinement from the specification to a program may be undertaken. This may be most appropriate in high-integrity systems involving safety or security.

Level 2: Theorem provers may be used to undertake fully formal machine-checked proofs. This can be very expensive and is only practically worthwhile if the cost of mistakes is extremely high (e.g., in critical parts of microprocessor design).

Further information on this is expanded below.

As with programming language semantics, styles of formal methods may be roughly classified as follows:

- Denotational semantics, in which the meaning of a system is expressed in the mathematical theory of domains. Proponents of such methods rely on the well-understood nature of domains to give meaning to the system; critics

point out that not every system may be intuitively or naturally viewed as a function.
- Operational semantics, in which the meaning of a system is expressed as a sequence of actions of a (presumably) simpler computational model. Proponents of such methods point to the simplicity of their models as a means to expressive clarity; critics counter that the problem of semantics has just been delayed (who defines the semantics of the simpler model?).
- Axiomatic semantics, in which the meaning of the system is expressed in terms of preconditions and postconditions which are true before and after the system performs a task, respectively. Proponents note the connection to classical logic; critics note that such semantics never really describe what a system *does* (merely what is true before and afterwards).

Lightweight formal methods

Some practitioners believe that the formal methods community has overemphasized full formalization of a specification or design.[4][5] They contend that the expressiveness of the languages involved, as well as the complexity of the systems being modelled, make full formalization a difficult and expensive task. As an alternative, various *lightweight* formal methods, which emphasize partial specification and focused application, have been proposed. Examples of this lightweight approach to formal methods include the Alloy object modelling notation,[6] Denney's synthesis of some aspects of the Z notation with use case driven development,[7] and the CSK VDM Tools.[8]

Elements

Formal Methods consists of the following basic elements :-

Requirement Specifications

These are usually developed in close collaboration with the customer. They are of a general nature regarding implementation but a specific nature about elements of the project such as :-

1. physicality.
2. performance.
3. inter-connectivity.
4. functionality.

Project Specifications

These are usually developed by the Project Architect. They include the specification of the project framework (how the components are tied together - how they communicate etc.), the component interfaces (e.g. function call based structures) and the component functionality. They form the specific details required by the engineers implementing the components of the project.

Project Implementation Details

These are usually constructed by the engineers constructing the project. These form a description of how the project was implemented on an individual component basis and on a system wide basis.

Test Specifications

These have a 1:1 relationship to the Requirement Specifications. These are often constructed by a Test Engineer and can consist of test scripts. Additional tests may also be specified.

Test Results

These indicate the results of the tests (software is usually tested by an automated test harness) and are used to verify the project.

Formal Methods projects are specification and test driven.

With any Formal Methods project there is an additional layer involving implementation standards and associated documentation. With software this is Coding Standards (not to be confused with coding styles).

Uses

Formal methods can be applied at various points through the development process.

Specification

Formal methods may be used to give a description of the system to be developed, at whatever level(s) of detail desired. This formal description can be used to guide further development activities (see following sections); additionally, it can be used to verify that the requirements for the system being developed have been completely and accurately specified.

The need for formal specification systems has been noted for years. In the ALGOL 58 report,[9] John Backus presented a formal notation for describing programming language syntax (later named Backus Normal Form then renamed Backus-Naur Form (BNF)[10]). Backus also wrote that a formal description of the meaning of syntactically valid ALGOL programs wasn't completed in time for inclusion in the report. "Therefore the formal treatment of the semantics of legal programs will be included in a subsequent paper." It never appeared.

Development

Once a formal specification has been produced, the specification may be used as a guide while the concrete system is developed during the design process (i.e., realized typically in software, but also potentially in hardware). For example:

- If the formal specification is in an operational semantics, the observed behavior of the concrete system can be compared with the behavior of the specification (which itself should be executable or simulateable). Additionally, the operational commands of the specification may be amenable to direct translation into executable code.
- If the formal specification is in an axiomatic semantics, the preconditions and post-conditions of the specification may become assertions in the executable code.

Verification

Once a formal specification has been developed, the specification may be used as the basis for proving properties of the specification (and hopefully by inference the developed system).

Human-directed proof

Sometimes, the motivation for proving the correctness of a system is not the obvious need for re-assurance of the correctness of the system, but a desire to understand the system better. Consequently, some proofs of correctness are produced in the style of mathematical proof: handwritten (or typeset) using natural language, using a level of informality common to such proofs. A "good" proof is one which is readable and understandable by other human readers.

Critics of such approaches point out that the ambiguity inherent in natural language allows errors to be undetected in such proofs; often, subtle errors can be present in the low-level details typically overlooked by such proofs. Additionally, the work

involved in producing such a good proof requires a high level of mathematical sophistication and expertise.

Automated proof

In contrast, there is increasing interest in producing proofs of correctness of such systems by automated means. Automated techniques fall into two general categories:

- Automated theorem proving, in which a system attempts to produce a formal proof from scratch, given a description of the system, a set of logical axioms, and a set of inference rules.
- Model checking, in which a system verifies certain properties by means of an exhaustive search of all possible states that a system could enter during its execution.

Some automated theorem provers require guidance as to which properties are "interesting" enough to pursue, while others work without human intervention. Model checkers can quickly get bogged down in checking millions of uninteresting states if not given a sufficiently abstract model.

Proponents of such systems argue that the results have greater mathematical certainty than human-produced proofs, since all the tedious details have been algorithmically verified. The training required to use such systems is also less than that required to produce good mathematical proofs by hand, making the techniques accessible to a wider variety of practitioners.

Critics note that some of those systems are like oracles: they make a pronouncement of truth, yet give no explanation of that truth. There is also the problem of "verifying the verifier"; if the program which aids in the verification is itself unproven, there may be reason to doubt the soundness of the produced results. Some modern model checking tools produce a "proof log" detailing each step in their proof, making it possible to perform, given suitable tools, independent verification.

Applications

Formal methods are applied in different areas of hardware and software, including routers, Ethernet switches, routing protocols, and security applications. There are several examples in which FMs have been used to verify the functionality of the hardware and software used in DCs. IBM used ACL2, a theorem prover, in AMD

x86 processor development process. Intel uses FMs to verify its hardware and firmware (permanent software programmed into a read-only memory). There are several other projects of NASA in which FMs are applied, such as Next Generation Air Transportation System, Unmanned Aircraft System integration in National Airspace System,[11] and Airborne Coordinated Conflict Resolution and Detection (ACCoRD).[12]

B-Method with AtelierB is used to develop safety automatisms for the various subways installed throughout the world by Alstom and Siemens, and also for Common Criteria certification and the development of system models by ATMEL and STMicroelectronics.

Formal verification has been frequently used in hardware by most of the well-known hardware vendors, such as IBM, Intel, and AMD. There are many areas of hardware, where Intel have used FMs to verify the working of the products, such as parameterized verification of cache coherent protocol,[13] Intel Core i7 processor execution engine validation [14] (using theorem proving, BDD's, and symbolic evaluation), optimization for Intel IA-64 architecture using HOL light theorem prover,[15] and verification of high performance dual-port gigabit Ethernet controller with a support for PCI express protocol and Intel advance management technology using Cadence.[16] Similarly, IBM has used formal methods in the verification of power gates,[17] registers,[18] and functional verification of the IBM Power7 microprocessor.[19]

Formal methods and notations

There are a variety of formal methods and notations available.

Specification languages

- Abstract State Machines (ASMs)
- A Computational Logic for Applicative Common Lisp (ACL2)
- ANSI/ISO C Specification Language (ACSL)
- Alloy
- Autonomic System Specification Language (ASSL)
- B-Method
- CADP
- Common Algebraic Specification Language (CASL)
- Java Modeling Language (JML)
- Knowledge Based Software Assistant (KBSA)
- Process calculi

    - CSP
    - LOTOS
    - π-calculus
- Actor model
- Esterel
- Lustre
- mCRL2
- Perfect Developer
- Petri nets
- Predicative programming
- RAISE
- SPARK Ada
- Spec sharp (Spec#)
- Specification and Description Language
- Temporal logic of actions (TLA)
- USL
- VDM
    - VDM-SL
    - VDM++
- Z notation
- Rebeca Modeling Language

Model checkers

- SPIN
- PAT is a powerful free model checker, simulator and refinement checker for concurrent systems and CSP extensions (e.g. shared variables, arrays, fairness).
- MALPAS Software Static Analysis Toolset is an industrial strength model checker used for Formal Proof of safety critical systems
- UPPAAL