IV. Foundations

IV.1 Specifications & Models


model A simplified representation of something (the referent). The representation may be physical or abstract, and may be restricted to certain properties of the referent. In computing, models are usually abstract and are typically represented in a diagramming notation, such as dataflow diagrams, behavioural diagrams, UML diagrams, or formal specifications (

specification A formal description of a system, or a component or module of a system, intended as a basis for further development. The expression of the specification may be in text in a natural language (e.g. English), in a specification language, which may be a formal mathematical language, and by the use of specification stages of a methodology, which includes a diagrammatic technique. Characteristics of a good specification are that it should be unambiguous, complete, verifiable, consistent, modifiable, traceable, and usable after development.

model-based specification A form of specification, usually software specification, that is developed by creating a mathematical model of that system. Typically the mathematical model is expressed in terms of objects and operations, and these are defined using such mathematical concepts as sets, relations and functions.

specification language. A language, often a machine-processible combination of natural and formal language, used to express the requirements, design, behavior, or other characteristics of a system or component. For example, a design language or requirements specification language.

Contrast with: programming language; query language.

[IEEE-610.12-1990]
My Definitions

- What is a model?
  A model is an abstract description of an artefact that reflects all for a given problem/perspective relevant details.

- What is a specification?
  A specification is a description of the required properties that reflects all for a given problem/perspective all relevant details an artefact has to ensure.

What is the difference?

Models vs. Specifications

- Model \( M \):
  abstract description of an artefact
- Specification \( \varphi \):
  a description of required properties an artefact has to ensure

Relation (usually only considered for formal …):
- A model can fulfil a specification: \( M \models \varphi \)
- A model \( M \) is also a specification \( (M \models \varphi \iff \forall \varphi : M \models \varphi) \)
- All models fulfil the empty specification: \( \models true \)
- A given specification \( \varphi \) can be inconsistent and no model fulfilling it exists: \( \lnot \exists M : M \models \varphi \)

Formal Models

- What is a conceptual model?
  A conceptual model is an abstract description of an artefact that reflects all for a given problem/perspective relevant details.

- What is a formal model?
  A formal model is an abstract description with defined syntax and semantics for an artefact that reflects all for a given problem/perspective relevant details.

Formal Specification

- What is an informal specification?
  An informal specification is a set of required properties that reflects all for a given problem/perspective all relevant details an artefact has to ensure.

- What is a formal specification?
  A formal specification is a set of required properties with defined syntax and semantics that reflects all for a given problem/perspective all relevant details an artefact has to ensure.

Formal Models & Specifications (1/2)

Different kind of per Model/specification analysis:
- The model is syntactically correct
  - no typing errors, ...
- The model is semantically correct
  - General requirements (e.g., no deadlocks)
- The specification is syntactically correct
  - no syntax errors, ...
- The specification is semantically correct
  - not inconsistent, ...

Formal Models & Specifications (2/2)

Different kind of analysis possible (formal methods):
- Model & specification
  - model fulfils specification \( (M \models \varphi) \)
- Model & model
  - Refinement \( M \subseteq M' \): \( M' \) is a refinement of \( M \)
    - Abstraction \( M \supseteq M' \): \( M' \) is an abstraction of \( M \)
  - The considered set of all specifications \( \varphi \) determines what notion of refinement is required
- Specification & specification
  - Refinement \( \varphi \Rightarrow \varphi' \): \( \varphi' \) is stronger than \( \varphi \)
    - Abstraction \( \varphi \Leftarrow \varphi' \): \( \varphi \) is weaker than \( \varphi' \)

Why also the informal/conceptual stuff?
Informal Specifications

Why not formal?
- The intention of an informal specification is often to foster communication between the customer and the requirement engineer.
- The mental distance between the mental model of the modeller and the formalism is often rather large:
  - Efforts to do a correct formalization are often costly.
  - A formalization can result in subtle specification errors (which are difficult to detect).

Examples for informal specifications in SWE:
- requirements documents
- Specification documents

Conceptual Models

Why not formal?
- The intention of the model is to understand rather than determine all “details”.
- Misunderstanding about underspecification (see refinement).
- Modelling at the right level of abstraction is a complex task.
- Efforts to do a correct formalization are often too high.

Examples for conceptual models in SWE:
- OO analysis models (no semantics!)
- Architectural drawings
- UML \(\Rightarrow\) today often a formal interpretation is possible!

IV.1.1 Fault Models

A fault model determines which possible effects of faults on the behaviour of a system model are considered.

- Usually explicit considered only for random hardware faults.
- Often no perfect representation of all possible physical effects.
- But can assist in the simulation of possible fault, the design of fault tolerant components, ...

Single Stuck-at Fault Model

Assumptions:
- Basic hardware function seen as “black-box”.
- Fault modelled as:
  - Input or output error;
  - Stuck at either 1 or 0.
- Models only permanent faults.

Bridging Fault Model

Assumptions:
- Basic hardware function seen as “black-box”.
- Fault modelled as:
  - Accidentally joined two or more nodes in a circuit.
- Models only permanent faults.

Hardware Fault models

Fault Models:
- Single-stuck-at fault model
- Bridging fault model
- Stuck-open fault model (CMOS)

General Concept:
- Derive faults by altering system model.

Similar concept for software:
- Mutation testing includes a similar approach, because the faults are assumed to be simple changes of the program code (no assumption that these are all possible faults).
**Problem of Software Fault Models**

**Hardware:**
- Physical defects affect only a part of the system
  - One mechanical component
  - Limited area on a chip
- Alter only one logic element

**Software:**
- Changed variable can possibly affect the correctness of any routine of a component accessing it (only limited by encapsulation)
- Corrupted op-code can result in arbitrary behaviour of the whole component

Consider **software modules**

---

**Faults of Software Modules**

Possible component faults:
- Crash (implicit fail silent assumption)
- Corrupted behaviour with maximal duration (watchdog)
- Corrupted behaviour without duration limit

Possible communication faults:
- Reordering of messages
- Lost massages
- Corrupted messages
- Repeated messages
- Arbitrary faked messages (Byzantine faults)
  - Agreement for $3N+1$ works only when at most $N$ cheat!

---

**Fault Models: Discussion**

Use a coarse grain fault model:
- Efficient algorithms for test pattern generation
- Cheap design techniques for fault tolerance

More fine grain fault model:
- More relevant faults are covered
- Only inefficient algorithms for test pattern generation
- Only expensive design techniques for fault tolerance
- Sometimes too much and complex faults have to be considered

---

**IV.1.2 Accident Models**

Requirements of an accident model:
- How does an accident happen?
- Which activities can prevent it from happening?

Examples:
- Basic energy model
- Domino and single event models
- Chain of event models
- Perturbation theory of accidents
- System theory
- Control theory

---

**Basic Energy Model**

**Engineering view:**
- Accident: result of an uncontrolled and undesired release of energy
- Prevention: altering or controlling the path of energy flow

More detailed considerations:
- Two cases (1) exceed resistance (2) from normal exchange
- An energy transformation accident occurs when transformation of energy injures people or damages property and an energy deficiency accident happens when the required energy is missing

---

**Domino Model**

Causality requires a sequence of five steps (dominos):
1. Anomaly or social environment, leading to
2. Fault of person, which is the proximate reason for
3. An unsafe act or condition (mechanical or physical hazard), which results in
4. An accident, which leads to
5. An injury
- Accident can be prevented at any step

Invalid assumption: the accident is always the result of a single event!
Chain of Events Model

- View:
  - sequence of events (not single event)
  - problem: where to stop when looking backwards?
  - Subjective choice of what should be included
  - AND/OR combination (see fault trees)
  - Accident prevention might include multiple actions

Multilinear Event Sequence Model (1/2)

- Accident prevention might include multiple actions
- the timing of events is not considered

Multilinear Event Sequence Model (2/2)

- Perturbation theory of accidents:
  - Perturbations = external influences that vary from what is usual or expected
  - stable state as long as actors adapt to the perturbations without being stressed beyond their capacity to adapt or recover. The resulting accident sequence:
    - If one of the actors fails or is unable to adapt, the perturbation initiates an accident sequence
    - This may overstress other actors, causing injury or damage, which in turn overstress subsequently exposed actors
    - Cascading injury or damage are possible until the actors are able to accommodate the stresses without harm

Accident Model & System Theory

- Models based on system theory consider accidents as arising from the interaction between components of the system
- Safety is an emergent property that arises when the components interact within an environment as restricted by its safety constraints
- Accident can result from interactions when one component violates these constraints
- Such an interaction of components leading to a hazard is named dysfunctional interaction
  - Software can contribute to an accident (e.g., controller)

Two types of dysfunctional interaction:
- Problems in articulating and coordinating subsystems
  - Boundary, overlapping control, asynchronous evolution of subsystems
- Lack of linkage
  - E.g., mental model mismatch (user ↔ system)

Accident Models: Summary

- Examples:
  - basic energy model
  - domino model
  - chain of events model
  - perturbation theory of accidents
  - System theory
  - Control theory

Choice of accident model:
- Restrict ways how hazards can be excluded
- Limits the ways hazards may be produced
IV.1.3 Formal Methods

- Formal methods are the use of mathematical techniques in the specification, design and analysis of computer hardware and software.
- CASE tools and graphic or diagrammatic methods to describe the requirements or specification of a system are formalized methods, only.

Formal methods are based on:
- Formal specifications and formal models
- Techniques to prove equivalence or refinement

Exploit Formal Models

General observation:
- Mathematical modeling of the (continuous) system behavior is at the heart of all forms of control engineering.
- Computer hardware and software works in contrast has a discrete (digital) system behavior and therefore discrete mathematics and mathematical logic has to be employed.
- The term formal methods covers a wide range of techniques that use discrete mathematics and mathematical logic.

Levels of Rigour (1/2)

Four levels of rigour ([Rushby1993]):

**Level 0**: No use of formal methods

**Level 1**: Use of concepts and notations from discrete mathematics

**Level 2**: Use of formalized specification languages with some mechanized support tools

**Level 3**: Use of fully formal specification languages with comprehensive support environments, including mechanized theorem proving or proof checking

Levels of Rigour (2/2)

<table>
<thead>
<tr>
<th>Documents</th>
<th>Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Level 0</strong>: Natural language, pseudo code, augmented with diagrams and equations. Manual process of review and inspection</td>
<td></td>
</tr>
<tr>
<td><strong>Level 1</strong>: Logic and discrete mathematics is used to replace some natural language parts (less ambiguous). Only informal proofs if any</td>
<td></td>
</tr>
<tr>
<td><strong>Level 2</strong>: Specification languages provide a standardized notation for discrete mathematics which usually provide some automated support for checking for certain classes of faults. Only rigorous proofs, but often still not formal proofs</td>
<td></td>
</tr>
<tr>
<td><strong>Level 3</strong>: Specification language that employs a strictly defined logic and provides techniques for formal proofs. Only formal proofs which can be developed using theorem provers and checked using proof checkers</td>
<td></td>
</tr>
</tbody>
</table>

Extent of the Application

- Formal methods can be employed for all, some or none of the various verification activities between the process stages.
- Formal methods may be applied for the complete system or only selected components.
- Only certain characteristics (e.g., those related to safety) may be verified using formal methods.

Observation:
- Different formal methods may be employed.
- Translation between them is often "impossible".

Areas of Application

- **Requirements analysis**
  - Limited value, because good communication with the customers is most important.
- **Specification**
  - Use of formal specification languages
- **Design and Implementation**
  - Proof correct realization step
  - Stepwise refinement
- **Verification & Validation**
  - Use tools to verify specific properties
Formal Specification Languages (1/2)
Algebraic specification languages (level 2)
- OBJ
  - Executable specification language
  - Behaviour is expressed by rules (equations)
Model-based specification languages (level 2)
- Vienna Development Method (VDM)
  - Operations, states
  - Pre- and post-conditions
  - Programming constructs: while, if, then, else, ...
  - No time concept, but extensions to cope with time possible
  - Only sequential systems

Formal Specification Languages (2/2)
- Z (University of Oxford) (level 2)
  - Based on typed set theory
  - Variables and axioms
  - No time concept, but extensions to cope with time possible
  - Only sequential systems
Formal methods for concurrent systems (level 2)
- Temporal logics (CTL, LTL)
- Process algebras (LOTOS, CCS)
Level 3 approaches:
- Higher Order Logic (HOL)
- PVS
- Boyer-Moore
Integration of process algebras in HOL/PVS

Formal Specification Languages (3/3)
Adantages:
- Automatic consistency checks
- Rigorous typechecking
- Executable specifications (OBJ, subset of VDM)
- Validation is a trusted unambiguous system description
- Formal proofs for the design and implementation are possible
Disadvantages:
- More specification efforts
- Well trained experts are required
- Detection of subtle errors is hard due to the larger mental distance

Design and Implementation
- A given specification is transformed into
  - An architectural design
  - A detailed design
  - An implementation
- The models become more detailed with each step
  - Often structural similarities with specification
  - But also structural difference (e.g., fault tolerance)
Steps required:
- Realize the more detailed model
- Proof that the model fulfills the model specification

Stepwise Refinement (1/2)
- A series of correct model-transformations (stepwise refinement) can be used to realize the required system
- Techniques that assist the required proof are often provided (less effective for significant structural changes)

Stepwise Refinement (2/2)
Stepwise refinement: A software development technique in which data and processing steps are defined broadly at first and then further defined with increasing detail.

[IEEE-610.12-1990]
Observation:
- In line with most development techniques
- But, in practice changes in later stages will enforce to also change earlier stages to be correct!
Automatic Checking

- Instead of a „complete“ specification use only one that consists of relevant properties (e.g., for safety)
- Usually only restricted notions for formal models
  - Finite automata (or similar restricted models)
  - Prepositional logic
  - Temporal logic (Model-Checking)

Benefits:
- Counterexample when a property is not fulfilled
- For too large models not feasible (state explosion)
- For too complex formal properties not feasible

Problems:
- Restricted notions for formal models
- Finite automata (or similar restricted models)
- Prepositional logic
- Temporal logic (Model-Checking)

Industrial Applications (1/2)

- Traffic Alert and Collision Avoidance System (TCAS)
  - After flaws have been found in the original standard formal methods have been used to produce a new formal specification
- Central control function display information system (CDIS)
  - Requirements: world model of the system and its environment (ER), processing requirements (real-time Youdan technique), core specification of all handled data (VDM)
  - Reduced train separation for the Paris Métro (SACEM)
    - A specification written in the B language and reverse engineering to validate the operational code

Benefits:
- Counterexample when a property is not fulfilled
- For too large models not feasible (state explosion)
- For too complex formal properties not feasible

Problems:
- Restricted notions for formal models
- Finite automata (or similar restricted models)
- Prepositional logic
- Temporal logic (Model-Checking)

Industrial Applications (2/2)

- T800 Transputer floating-point unit (INMOS)
  - Work with Z was that successful, that even uncovered faults in the IEEE floating-point standard gave been detected
- AAMPS Microprocessor (Rockwell)
  - The operations and the related microcode have been specified using PVS at the instruction set and register transfer level to proof that the microcode correctly implements the operations
- Several railway, avionics and nuclear applications (ABB)
  - Prepositional logic and the Prover tool have been used for these projects and verification time has been reduced by 80% and the overall development time was cut by 20%

Formal Methods: Summary

- Also less beneficial cases:
  - Flight warning computer A330/340 (Airbus)
    - A second implementation with formal specification written in LOTOS took about 30% longer, because the design was to large for the tools and the generated code was to slow

Current Status:
- Formal methods are at most used for parts of the development
- Machine supported formal verification remains specialized activity
- The use of formal specification languages is now widespread for safety-critical systems

Remember:
- Safety ≠ Correctness (environment assumptions correct?)

IV.1.4 Summary

Fault Models: What kind of faults do we consider?

Accident Models: How does an accident happens and how can we prevent it?

Formal Methods: How exactly does the system work?

Further models:
- Time models (discrete, continuous)
- Environment models (see environment simulation)
- Human task and error models (Individual, Groups)
- ...

Summary

Inherent limitation of all (formal) models:
- Effects not covered by the model (abstraction) cannot be detected when considering the model only (validation is always required)
- Models are often only useful, when abstraction is used to hide tedious details not relevant for the problem at hand (building adequate models is crucial)