Formal Methods for Safety Assessment

- formal method ⇒use of formalisms and techniques (based on mathematics) for specification, design, analysis
- formalized methods: several notations, different sorts of diagrams, pseudocode, formality, but no formal foundation

Advantages of Formal Methods

- growing complexity of computer-based system
- remove ambiguities in specification of requirements and models
- *verification* checking, whether a system correctly implements requirements
- *validation* check, whether the started requirements are the intended ones
- traditional techniques: testing, simulation (not exhaustive)
- formal methods: exhaustive, automation possible

Formal Methods in the Development Process

- formal methods can have high level ob abstraction
- Rushby's four different levels:

  Level 0 *no use of formal methods* standard practice

  Level 1 *use of concepts and notation from discrete mathematics* concepts and notations from discrete mathematics and mathematical logic

  Level 2 *use of formalized specification languages with some mechanized support tools* formal notations with fixed syntax, at least limited tool support, proof performed manually

  Level 3 *Use of fully formal specification languages with comprehensive support environments, including mechanized theorem proving or proof checking* fully formal specification languages, supported by tools, automatic generation of proofs, proofs are fully formalized

- lower levels of rigor: not so safety relevant
- high levels of rigor ⇒fully safety critical systems
- formal methods need not be applied everywhere ⇒overall correctness not guaranteed
- apply formal methods early or late?

  early specification bugs are the worst, work better at early stages than testing and simulation (which work good at code/ gate level)

  late completed system will be verified

- limit formal methods to certain subsystems
- formal methods can be used for *certification* of safety-critical systems

Problems and Limitations

- ↯ formal methods are expensive
- ✚ formal methods can find bugs in very early stages ⇒save money in long run
- ↯ difficult to use, requires expertise

- ↯ maximum benefits only, if applied in every stage of development process
- ✚ use lightweight formal methods (produce specifications of system, although no full coverage)
- ↯ single formalism not suited for whole process, use different formalisms ⇒may cause problems when verification has to be assembled
- ↯ not fully automatic, problems often huge, computing power not enough
- ↯ formal languages are often hard to understand
- ↯ proof (by theorem prover/model checker may not be enough to convince certification authority)
- ✚ counterexamples
- ↯ verifier may produce wrong results

Formal Models and Specifications

- categories for formal specification languages

  algebraic languages

  model-based languages

  process algebras and calculi

  logic-based languages

- **property-oriented** modeling in terms of properties, that system must satisfy

  model-oriented mathematical model corresponding to particular implementation, closer to final implementation (are used in later stages of design)

0.0.1 Algebraic Specification Languages

- characterize objects to be specified in terms of algebraic relationships between them
- programs are many sorted algebras (collections of data, operations over data)
- typically property oriented

Model-Based Specification Languages

- model-based style of specification, explicit mathematical model of system state and corresponding operations
- state consists of set of external variables, preconditions, postconditions
- Z, VDM (Vienna Development Method), Larch, Alloy
- *Z* based on typed set theory + first-order predicate logic, specification via set of schemas, typically based on a hierarchy of schemas
- related to Z ⇒B language and method
  - a system is modeled as set of interdependent abstract machines

Process Algebras and Calculi

- set of logical formalisms ⇒describe concurrent and distributed systems, explicit model interaction, inter-process communication, synchronization
- behavior is described using axiomatic approach and formalized terms (algebraic laws)
- *bisumulation equivalence* ⇒formal reasoning about equivalence between processes
- CSP (Communicating Sequential Processes), CCS (Calculus of Communicating Systems), ACP (Algebra of Communicating Processes), π-calculus, ambient-calculus, LOTOS

Logic-Based Languages

- logic in broad sense to describe systems and system properties
- theorem provers, temporal (LTL, CTL) and interval logic (e.g. Duration Calculus)
- state machines

  moore machines output depends only on state
  mealy machines output depends on state and input

- *flowcharts* explicitly model program statements and control flow, hoare approach, dijkstras weakest transitions
- *petri nets*
- *kripke structures*

State Transition Systems

$\mathcal{P}$ is a set of proposition, state transition system is tuple $\langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$

- $\mathcal{S}$ finite set of states
- $\mathcal{I} \subseteq \mathcal{S}$ set of initial states
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ transition relation
- $\mathcal{L} : \mathcal{S} \longrightarrow 2^{\mathcal{P}}$

- *trace* start in $\mathcal{I}$, repetedly append states reachable through $\mathcal{R}$
- *paths* infinite traces
- state s is *reachable*, if there exists a path from $\mathcal{I}$
- *NuSMV*
  - combination of components via

    synchronous composition (default in NuSMV), components evolve in parallel, perform transitions simultaneously
    asynchronous composition interleaving model for evolution of component

Temporal Logic

- express properties of reactive systems modeled as Kripke structures

  safety properties nothing bad ever happens
  liveness properties something desirable will eventually happen

- most common: LTL (linear time temporal logic), CTL (computation tree logic)
- *PSL* (Property Specification Language), high level language for expressing temporal requirements
  - origins in the hardware domain
  - superset of CTL and LTL

Formal Methods for Verification and Validation

Testing and Simulation

- testing and simulation ⇒commonly used for verification and validation
- model must be executable

  statically ~~execution of model~~

dynamically operation of real system

- *test-oracle* predict what system should do ⇒from system requirements

simulation use a model

- environmental simulation: simulate environment
- good, when testing with environment is not possible (impossible: space flight, dangerous: nuclear power plant)

forms of testing **static testing** test without operating the system

- **walk-throughs, design reviews** a form of code inspection, typically by peer-review
- **fagan inspections** systematic methodology to find defects and omissions in development process
- **formal proofs**
- **type analysis** statically analyze type information, ensure that type errors don't occur at run-time
- **control flow analysis** analyze control structure of program, formalized as graph
- **data flow analysis** analyze flow of data of program, analyze operations on data, different forms of static analysis
- **symbolic execution** run program using symbolic inputs, compare with expected result
- **pointer analysis** analyze how program accesses pointers or heaps ⇒memory leaks
- **shape analysis** generalization of pointer analysis ⇒determine information about heap-allocated data structures, also produces information for debugging

dynamic testing actual execution of system

- **behavioral testing** intended to evaluate behavior of system form perspective of external user
  - **functional testing** evaluates compliance with functional requirements
  - **nonfunctional testing** evaluates nonfunctional characteristics (performance, reliability, safety)
- **structural testing** test various routines and different execution paths, requires knowledge of internal implementation ⇒white-box testing
- **random testing** random choice about test cases

black-box testing tester has no knowledge about internal implementation of system, also *requirement based testing*

white-box testing performed by test-engineers, knowledge of internal implementation

test vector set of inputs to be applied

test coverage **coverage-based testing** plan test activity according to test coverage

requirements-based coverage for how many requirements has been tested

equivalence partitioning partition test vectors into equivalence classes ⇒similar, qualitative behaviors (black box view)

boundary analysis test vectors at extreme boundaries of equivalence class (at either side of boundary)

state-transition testing stimulate each transition of system ⇒finite state machine

structure based coverage based on data-flow or control flow

- **statement coverage** measures portion of statements to be executed (often minimal requirement for coverage testing)
- **branch coverage** measure coverage of paths
- **call-graph coverage** each possible invocation tree of subroutines

Theorem Proving

- system and properties are expressed in common language
- based on formal system ⇒verify a property ≡ general proof

- generate lemmas as intermediate proofs
- different theorem provers are based on different inference rules

  automated theorem provers fully automated

  interactive theorem provers require human guidance

  proof checkers no proof construction, verify existing proof to be valid