Welcome to CSC2108F

Automated Verification

Instructor: Marsha Chechik

Introduction to Formal Methods

Overview

- What are formal methods?
- Industrial applications of FMs
- Characteristics of FMs
- A closer look at several examples
- Summary

References: [1]-[3]

Formal Methods - Definition

Formal methods for computer systems are mathematically based techniques for describing system properties.

Such formal methods provide framework within which people can specify, develop, and verify systems in a systematic, rather than ad hoc, manner.

Formal Methods - Definition

A method is *formal* if it has a sound mathematical basis, typically given by a formal specification language.

Can precisely define
- Consistency and Completeness notions
- Specification
- Implementation
- Correctness

Can reveal design flaws that otherwise might be discovered only during testing and debugging.

Can help determine correctness of a system implementation and the equivalence of difference implementations.

What does formal method consist of?

A set of guidelines or a "style sheet" that tells the user the circumstances under which the method can and should be applied and how it can be applied most effectively.

May or may not have tool support.

If syntax of formal method's specification is made explicit, can have *syntax analysis* tools.

If language semantics is restricted, can have *semantics analysis* tools.

So, formal specifications are amendable to machine analysis and manipulation

What is a Specification Language?

It provides ...
- a notation (its syntactic domain)
- a universe of objects (its semantic domain)
- a precise rule defining which objects satisfy each specification.

A *specification* is a sentence written in terms of the elements of the syntactic domain. It denotes a specificand set, a subset of the semantic domain.

A *specificand* is an object satisfying a specification. The *satisfies* relation provides the meaning, or interpretation, for the syntactic elements.

Syntactic Domains

Defined in terms of
- a set of symbols (constants, variables, logic connectives)
- a set of grammatical rules for combining these symbols into well-formed sentences.

Example:
$\forall x, P(x) \Rightarrow Q(x)$ - well-formed in PL
$\forall x, \Rightarrow P(x) \Rightarrow Q(x)$ - not well-formed in PL

Doesn't have to be text. Graphical elements can also have formal semantics.

Semantic Domains

What can be specified using formal methods?
A great number of things!

- ADT specification languages - for algebras, theories and programs.

- Concurrent and distributed systems specification languages - for state sequences, event sequences, streams, synchronization trees, state machines, etc.

- Programming languages - for functions from input to output, computations, predicate transformers, relations, etc.

Programming vs Specification Languages

$$\text{PL} \rightarrow \text{SL}$$
$$\text{SL} \not\rightarrow \text{PL}$$

- Specifications do not have to be executable
- Are not restricted to expressing only computable functions

- All programs are formal objects (and thus can be formally manipulated - compiled and executed)
- Programmers can either work with informal requirements and formal programs or use additional formalism for requirements specification

Often use terms *implementation* instead of *specificand* and *implements* instead of *satisfies*.

Satisfies Relation

Behavioral specifications describe only constrains on the observable behavior of specificands.

- required functionality of the system (input → output)
- fault tolerance, safety, security, response time, space efficiency - addressed in current research

Often behavioral aspects are included as part of a system's functionality (e.g., "compute this function every 30 ms").

Structural specifications describe constraints on the internal composition of specificands (e.g., module interconnection languages).

- various kinds if hierarchical and uses relations
- procedure-call graphs, data-dependency diagrams, definition-use chains

Properties of Specifications

- Well-formed specification should be unambiguous (i.e., have exactly one meaning)
  - any specification language incorporating a natural language is not formal
  - visual specification languages that permit multiple interpretations of a box and/or arrow is not formal

- A specification should be consistent
  - its specificand set is non-empty.
  - there is some implementation that will satisfy the specification.

- A specification need not necessarily be complete (in mathematical sense)
  - some things may be unspecified (design decision)
  - specifiers cannot predict all possible scenarios
  - but should state which aspects are intentionally underspecified

Proving properties of specificands

Usually have a specification language with well-defined logical inference system.

- Logical inference system defines a consequence relation
- Use it to prove properties from the specification (used as a set of facts)

Inference system gives users a way to predict a system's behavior without having to execute or even build it!

- Can ask questions (in form of conjectures) about a system cast in terms of specification itself
- If can prove a surprising result - then the specification is wrong

A formal method with defined inference system usually can be completely mechanized, if it has a finite set of finite rules.

- Theorem provers, proof checkers

Where to use formal methods?

- Can apply them in all phases of system development
- Greatest benefit - from process of formalizing rather than from end result

Requirements analysis

- helps clarify a customer's set of informally stated requirements
- reveals contradictions, ambiguities and incompleteness
- customers and specifiers can pose and answer questions based on specification to see if customer's intentions are met and whether the specificand has the desired set of properties

FM during System Design

Most important activities are decomposition and refinement
Most suitable FMs - VDM, Z, Larch, etc.

- *Decomposition* is the process of partitioning a system into smaller modules.
- Can write specifications to capture precisely the interfaces between these modules.
  - clients can use the module without knowing how it is implemented
  - module can be implemented regardless of who uses it

- *Refinement* involves working at different levels of abstraction, refining one module into several, choosing a representation type for abstract, etc.
- Each refinement step requires proof that properties are preserved

Formal Methods in V&V

Verification is the process of showing that a system satisfies its specification.

- Formal verification is impossible without a formal specification.
- Can verify critical pieces of the system.

Tricky part is to explicitly state the assumptions about the environment in which each critical piece is placed.

Formal methods can aid in system testing and debugging

- Can generate test cases for black-box testing from specs
- Specs can be used to explicitly state assumptions on use of module for boundary conditions
- Specs + implementation can be used for path testing, unit testing, integration testing.

Need to not only cover all paths but insure that there are no missing paths

FM in System Analysis

Specification serves as a reference point in analysis of built system.

Recent work in formal methods - on specifying a system already built, running and used

- Reveals bugs in published algorithms and circuit designs
- Finds unstated assumptions, inconsistencies, and unintentional incompleteness in systems.
- Applied to:
  - Medium-sized systems (VLSI circuits, microprocessors, operating systems kernels)
  - Some large systems (A-7, TCAS II)

Problem of scale:

- size of specification (can be managed using better tools)
- complexity of specificands (internal complexity, interface complexity)

Other Uses for Formal Methods

System Documentation.

Specification:
- a description alternative to system implementation.
- serves as a communication medium between a client and a specifier, between an implementer and specifier, between implementors, etc.
- can capture "what" rather than "how"

Specifications

- **CICS** (Oxford University and IBM Hursley Laboratories)
  - 1980s, using Z
  - formalized part of IBM's Customer Information Control System
  - measured overall improvement in the quality and 9% reduction in total development cost
  - Got the Queen's Award for Technological Achievement

- **CDIS** (1992, Praxis for UK Civil Aviation)
  - used VDM and CCS to specify Display Information System - a part of the new air traffic management system for London's airspace
  - Formal methods were used as an integral part of development process
  - Productivity was same or better than on comparable projects but defect rate of 0.75 faults/KLOC - 2 to 10 times improvement!

Specifications (Cont'd)

- **Lockheed C130J** (1995, Lockheed and Prexis)
  - Analyzed the code for avionic software supplied to US Air Force and RAF
  - software coded in SPARK-annotated subset of Ada
  - Specifications written in Software Productivity Consortium's CORE notation, based on SCR
  - The added quality decreased the overall cost of software because of savings in testing
  - Software was "correct by construction", passing all tests

Specifications (Cont'd)

- **TCAS** (early 90s, Irvine - U. of Washington)
  - Used RSML to specify requirements for Traffic Collision Avoidance System, required on all commercial aircraft flying in US airspace
  - RSML is based on Statecharts
  - TCAS II specification has been automatically checked for completeness and consistency
  - Provably correct code can be automatically generated from RSML spec
  - Demonstrated practicality of writing formal spec for a complex system and building formal model of a system that is readable and reviewable by application experts without special training

Formal Methods in Commercial and Safety-Critical Systems

Databases
- HP Medical Instruments real-time database for storing patient monitoring information (1991)

Devices
- Tektronix family of oscilloscopes (1990)
- Schlumberger line of household electricity meters (1996)

Medical
- Clinical Neutron Therapy System at U. of Wash (1995)

Nuclear
- Argonne National Laboratories' work on Reactor Safety System for the Experimental Breeder Reactor-II (1987-89)

Formal Methods - (Cont'd)

- Shutdown system of the Darlington Nuclear Generating System, Canada (1990)

Security
- Security policy model for the NATO Air Command and Control System (1995)
- The secure transmission of datagrams in the Multinet Gateway System (1984)
- Token-based Access Control System of the US NIST (1990)

Telephony
- Various features of AT&T's 5ESS telephone switching system using Estrel (1996) and combinations of Z and CSP (1995-96)
- University of Passau and Siemens Nixdorf's work on customizable telephone services and features (1996), for Deutsche Telekom

Formal Methods - (Cont'd)

Transportation
- Automatic train protection system for the Paris Metro (1992)
- British Rail's signaling rules (1994)
- On-board avionics software for Levy aircraft (1992)

Characteristics of Formal Methods

Characteristics of formal methods influence the style in which a user applies it.

Some characteristics:
- Property- vs model-oriented
- Executable
- Tool-supported
- Visual

Model- vs Property-Oriented

- *Model-oriented* method - specification defines a system's behavior directly by constructing a model of the system in terms of mathematical structures (sets, tuples, etc).

Examples:
Z, VDM, SCR, Petri Nets, I/O automata, Unity...

- *Property-oriented* method - state no more than the necessary minimal constraints on the system's behavior.

Examples:
algebraic specifications of ADTs, Hoare-axiom method, temporal logic, Larch...

Visual Languages

Languages which contain graphical elements in their syntactic domains.

Examples:
- Petri nets (specify behavior of concurrent systems)
- Statecharts (specify state transitions in reactive systems)
  - can represent a hierarchy of states
  - has formal semantics
  - has tool support
- Miro (like Statecharts, plus security constraints)

Buffer of Size 1 in StateCharts

Executable Specifications

- More restricted (its functions must be computable and defined over domains with finite representations)
- Suffer from implementation bias

Uses:
- To gain immediate feedback about the specification itself
- For rapid prototyping
- To test a specificand through symbolic execution of the specification

Examples:
- Statemate, SCR*
- Prolog
- PAISley - a model-oriented language, based on event sequences, used to specify functional and timing behavioral constraints for parallel processes

Bounds of Formal Methods

- Between the math and real worlds (customer vs specifier)
- Between the real world and its abstract representation (by formal specification language)
  Example: real arithmetic, abstracted from computer representation
- Between the real system and its environment

Environment is a set of assumptions. System's behavior is conditional on the environment's behavior:

Environment $\Rightarrow$ *System*

Specification can make assumptions about the environment but not place constraints over it. Although cannot state all assumptions, need to make explicit as many as possible

Examples

- Abstract data types - a symbol table
  - Z
  - VDM
  - Larch

- Concurrency - Unbounded buffer
  - Temporal logic (Pnueli)
  - Communicating Sequential Processes (CSP - Hoare)

Overview of Z

- a formal method based on set theory
- can be used in both model-oriented and property-oriented styles
- has a number of tools supporting it, e.g., B - proof checker.

Some notation:
- $X \mapsto Y$ - a set of partial mappings from set X to set Y
- unprimed variables stand for the state before an operation and primed - after
- $\lhd$ - domain subtraction operator

Symbol Table in Z

Overview of VDM

- supports a model-oriented specification style
- defines a set of built-in data types, e.g. sets, lists, and mappings
- specifications explicitly note pre- and post-conditions

Notation:
- similar to Z
- **rd** - read-only access
- **wr** - write-and-read access

Symbol Table in VDM

Overview of Larch

- property-oriented method, two-tiered specification: interface specification and a trait
- has syntax analyzers and LP - to perform semantic analysis on traits
- has no built-in notation or built-in types
  - user does not have to learn any special vocabulary
  - but has to provide a large set of user-defined symbols and equations defining their meaning
  - have libraries of traits that define many general concepts

Overview of Larch - Interface Specification

- specifies state-dependent behavior (side-effects, exceptional termination)
- looks similar to Z and VDM
- **requires** and **ensures** are pre- and post-conditions
- **modifies** - objects whose value may possibly get changed
- **signals** - raise exceptions

Overview of Larch - Traits

- specifies state-independent properties of data accessed by programs
- contains a set of function symbol declarations and a set of equations
- **generated by** - defines an inductive rule of inference and is useful for proving properties about all values
- **partitioned by** - adds equivalences between items
- **exempting** - specifies exception-handling
- **converts** and **exempting** - used to state that the algebraic specification is sufficiently complete.

Symbol Table in Larch

Temporal Logic - Overview

Here, model a system's behavior as a set of linear sequences of states and associated events (linear-time logic)

Two categories of properties:
- *safety* - "nothing bad ever happens" (functional correctness)
- *liveness* - "something good will eventually happen" (termination)

Temporal logic
- property-oriented method
- modal operators specify concisely assertions about system behavior
- can refer to past, current and future states (and events)
- specifications are terse, unstructured sets of predicates, all of which must be satisfied by a given implementation

Temporal Logic - Overview (Cont'd)

Some modal operators:
- $\bigcirc P$ - in the next state P will hold
- $\Box P$ - in all future states, P will hold
- $\Diamond P$ - in some future state, P will hold
- P - P held in the previous state
- $\bigcirc$ P - P held in some state in the past

-Buffer has a left input channel and a right output channel.
- $\langle c!m \rangle$ denotes the event of placing message m on channel c

Unbounded buffer in temporal logic

Meaning of Properties

1) any message transmitted to the right channel must have been previous placed on the left channel

2) Messages are transmitted first in, first out

3) All messages are unique. This is not a property of the buffer but an assumption of the environment, essential to the validity of the specification

4) (liveness property) Each incoming message will eventually be transmitted

Overview of CSP

- model-oriented method for specifying concurrent processes
- property-oriented method for stating and proving properties about the model
- based on model of *traces* (event sequences)
- assumes that processes send messages across channels and synchronize on events, so sending message m on channel c is synchronized with receiving m on c
- can specify programs - state and prove properties about the traces it demotes, can formally verify that a given CSP program satisfies a specification on races

Overview of CSP (Cont'd)

Notation:
- $x \to P$ - a process first engages in the event x and then behaves exactly as described by P
- *left?m* - message m is in *left* buffer
- *s*ˆ *t* - concatenation of sequence s to sequence t
- $x \to P \mid y \to Q$ - process that engages in either x or y
- $s \leq t$ - sequence s is a prefix of sequence t

Unbounded buffer in CSP

Meaning of Properties

1) If the buffer is empty, in the event that there is a message m on the left channel, it will input it.

2) If the buffer is non-empty, then either the buffer will input another message n from the left channel, appending it to the end of the buffer, or output the first message in the buffer to the right channel

BUFFER is a CSP program. Describes a set of traces, each of which satisfies the predicate on the RHS of **sat**. Input messages will eventually be delivered, and only in the order of arrival.

Overview of Transition Axiom

- Leslie Lamport, 1983
- Combines an axiomatic method for describing the behavior of individual operations with temporal logic assertions
- Uses keywords and distinct clauses for highlighting a mode of state, state initialization and state changes
- Uses a combination of linguistic features borrowed from formal methods for sequential and concurrent systems

Transition axiom notation

- invocations of different operations can be active concurrently but only one invocation of a given operation can be active at once
- *at(OP)*, *in(OP)*, *after(OP)* - where control is in relation with operation *OP*
- $*$ - concatenation
- $\leadsto$ - "leads to"

Bounded buffer in TA

Advantages of Formal Methods

Within some well-defined mathematical framework, formal methods let system developers couch their ideas precisely.

The more rigor:
- the more likely will requirements be stated correctly
- design decisions will be done correctly
- the more precisely we can argue about the correctness of the implementation

Summary. What can FM do?

- identify many deficiencies in a set of informally stated requirements
- detect discrepancies between a specification and an implementation
- find errors in existing programs and systems
- specify medium-sized and nontrivial problems, especially the functional behavior of sequential programs, ADTs and hardware
- provide a deeper understanding of the behavior of large, complex systems

Challenges in Formal Methods Research

- Specifying nonfunctional behavior s.t. reliability, safety, real time, performance, human factors
- Combining different methods
- \- Building more usable and more robust tools to manage large specifications and perform more complicated semantic analysis of specifications more efficiently
- \- Building specification libraries so systems and their components can be reused based on information captured in their specification

Challenges in Formal Methods Research (Cont'd)

- - Integrating formal methods with the entire system development effort (e.g., recording design rationale formally)

- - Demonstrating that existing techniques scale up to handle real-world problems (and to scale up the techniques themselves)

- - Educating and training more people in the use of FM

food for slide eater

food for slide eater

food for slide eater