Introduction to Formal Methods

October 6, 2005

- Errors and their sources
- What are formal methods ?
- Techniques and applications

Course objectives

– be able to verify correct behavior of designed systems
– detect main error types and sources
– use formal methods as an alternative to simulation and testing
– use rigor in the description of systems
– build appropriate models for the systems under design
– unambiguously express specifications for desired properties
– evaluate applicability of formal methods for a particular design
– know and be able to use several verification tools

Famous errors: Therac-25

– medical radiation therapy machine
– 6 massive overdoses leaving several dead (1985-87, USA Canada)
– cause: errors in the control program, no hardware safety backup

Analysis [Leveson 1995]:

- excessive trust in software when designing system
- reliability $\neq$ safety
- lack of hardware interlocks
- lack of appropriate software engineering practices (defensive design, specification, documentation, simplicity, formal analysis, testing)
- correcting one error does not necessarily make system safer !

Famous errors: Ariane 5 space rocket

– Self-destructed due to malfunction 40 seconds after launch (1996)
– Cause: 64-bit float $\rightarrow$ 16-bit int conversion generated uncaught exception in its ADA program
– Cost: $500 M (rocket), $7 billion (project)

Analysis

- main cause: inappropriate software reuse
- code taken over from the Ariane 4, without judicious analyis
  - execution was no longer necessary at moment of error
  - no analysis of overflow for unprotected variables

  $\Rightarrow$ necessity of specifying and observing an interface
- bad design of system fault tolerance: the inertial reference system and the backup system affected by the same error

Famous errors: The Pentium FDIV bug

Error in the floating point division unit (1994)

- SRT division algorithm, generates 2 quotient bits per cycle (base 4)
- uses a lookup table to determine next quotient digit
- a few entries erroneously marked as "don't care" ⇒ wrong values
- Cost: ca. $500 million

Analysis

- Circuit could have been formally verified at that time
  - by automated theorem proving [Clarke, German & Zhao]
  - or with special data structures for multiplication [Bryant & Chen]
- but other more complex components were verified instead (instruction execution, cache coherence)

Famous errors: Mars space probes

Mars Pathfinder, 1997

- Problem: on Mars, space probe was resetting frequently
- Cause:
  priority inversion between processes sharing common resources
- Issue and solution were well known in literature !
  [Sha, Rajkumar, Lehoczky. Priority Inheritance Protocols, 1990]

1. Process A (low priority) requests resource R
2. A interrupted by C (high priority)
3. C waits for R to be freed; switch back to A
4. A interrupted by B (medium priority, $A < B < C$)

$\Rightarrow$ C waits for lower priority B, without directly depending on it !

Solution:
raising the priority of a process (A) that obtains a resource to the level of the highest priority process (C) that can request the resource

Famous errors: Mars space probes

Mars Climate Orbiter, 1998

- disintegrated uppon enty to Mars atmosphere
- technical error: mismatch between anglo and metric units
- multiple process errors: lack of formal interfaces between modules

Mars Polar Lander, 1998

- landing gear prematurely activated upon entry to atmosphere
- resulting shock is interpreted as landing, engines are stopped
- error: lack of integration testing

How can one detect errors?

Testing

+ directly on the product ⇒ tests have immediate relevance
– errors detected late are costly
– diagnosis needs complete observability

Simulation

+ can be performed through the design stage
– simulator can be significantly slower than real system

– Exhaustive testing and simulation is often impossible

Program testing can be used to show the presence of bugs, but never to show their absence!" (E. W. Dijkstra, 1979)

What are formal methods ?

" ... mathematically-based languages, techniques and tools for specifying and verifying [...] systems" [Clarke & Wing, 1996]

Or, in more detail: "a set of tools and notations
– with a formal semantics,
– used to unambiguously specify the requirements of a systsm
– that allow proving properties of that specification
– and proving the correctness of an implementation with respect to that specifciation"
[Hinchey & Bowen, *Applications of Formal Methods*, 1995]

What can formal methods guarantee ?

- there are no absolute guarantees
- a formal method cannot be better than the employed model and the specifications
  - *model and specifications* have to be validated

However, formal methods can offer:

- a logically consistent way of reasoning
- exhaustive coverage, often impossible to achieve by other means
- mechanization and automation $\Rightarrow$ performance and correctness

They can complement successfully simulation, testing, etc.

Formal methods: Necessity and difficulties

Usefulness especially in case of:

– *complexity*: abstraction / approximation techniques
– *concurrency*: difficult to reproduce and analyze otherwise
– *criticality*: (avionics, banking, medicine, security)

Error synamics in software development [John Rushby, SRI]

- 20-50 errors/kloc before testing → 2-4 errors/kloc after
- formal code inspection can reduce before-testing errors 10-fold !

Case study on 10kloc distributed real-time code:

- verification and validation: 52% cost (57% time)
- of this, 27% cost in inspection, 73% in testing
- 21% due to 4 defects uncovered in final testing (one of these originated in design phase)
- error elimination in detailed code inspection: 160 times more efficient than in testing !

Error causes and costs

Errors in programs

[NASA JPL (Voyager and Galileo probes)]

- majority: deficiencies in requirement and interface specification
- 1 error in 3 pages of requirements and 21 pages of code
- only 1 in 3 were programming errors
- 2/3 of functional errors: omissions in requirement specifications
- majority of interface errors: due to bad communication

Summarizing: an overall view

- Most frequent error causes:
  conceptual errors, simultaneous defects, unforeseen interactions
  – main shortcomings: in timely application of formal methods
  – main cost: late error removal
- Maximum potential of formal methods:
  – in high-level modeling and verification
  – for complex, concurrent, distributed, reactive, real-time, fault-tolerant systems

Formal methods in the development cycle

- Requirement analysis.
  - can identify contradictions, ambiguities, omissions
- Design
  - decomposing into components and specifying interfaces
  - design by successive refinement
- Verification
- Testing and debugging
  - model-based test case generation
- Analysis
  - abstract model, less complex than real system

Applications

Formal verification of:

- Hardware
  - Combinational circuits
  - Sequential circuits
- Software (generally speaking)
- Communication protocols
- Security protocols
- Real-time systems
- Concurrent and distributed systems

Verification approaches

Two main categories:

Model checking (state space exploration)

– system is represented as a finite-state machine
– specification: reachability (no error state reached),
  or more complex (temporal logic formula)
– uses exhaustive state space exploration algorithms
  answer: “correct— or counterexample execution sequence

Theorem proving

– model represented in logical system with axioms and deduction rules
– application/analysis domain represented likewise (a *theory*)
– mechanized theorem proving: automated or manual

Techniques

- *Abstraction*: most important, reduces verification complexity
- On-the-fly state space construction and state space reduction
- Symbolic state space representation
- Refinement checking
- Compositional verification
- Assume-guarantee reasoning

Applications: Hardware design

- Verification of combinatorial equivalence
  – major success, became standard in all CAD tools
- Verification of sequential designs
  – large companies have dedicated research groups
  (IBM, Intel, Motorola, Fujitsu, Siemens, etc.)
  – use publicly available verifiers or their own in-house tools
- cache coherence protocols: Gigamax, IEEE Futurebus+
- Motorola 68020: modeled in Boyer-Moore theorem prover; verification of binary code produces by compilers
- AAMP-5 (avionics processor): modeled in PVS theorem prover; verification of microcode for instruction execution
- modeling/verification of DLX-type pipelined / superscalar processors

Applications: Avionics

Lockheed C130J

– ADA code with annotations in SPARK language analyzed

– result: "correct by construction" software, reduced cost

TCAS-II (Traffic Collision Avoidance System)

- mandatory on all U.S. commercial aircraft
- implements automatic alert and course change if dangerously close
- specification expressed in a formal language (RSML)
- completeness and consistency were verified [Heimdahl, Leveson '96]
- result: English-language description abandoned in favor of completely formal specification

Airbus A340

– Cousot et al. (1993) proved complete absence of runtime errors in main flight control software using a static program analyzer

⇒ formal models of complex systems are feasible ⇒ can be analyzed by experts from the application domain

Other Applications

- Telephony. Specification and analysis of interactions between various features of the telephone system.
- Consumer electronics. Manual and later automatic verification of a control protocol from Philips audio components.
- Control systems in automotive electronics.
- Communication protocols (untimed and timed).
- Security protocols. Analysis using special logics to reason about encrypted messages, intruders, etc.
- System software. Verification of device drivers.

Formal methods: Specification

- Specification is needed in any formal method
  can be the only aspect of the method (no analysis or verification)
- requires a language with formally (mathematically) defined syntax and semantics

A specification language defines:

– a syntactic domain (the formal notation)
– a semantic domain (the universe of regarded objects)
– a precise definition of objects that satisfy a specification

[M. Chechik, *Automated Verification*, lecture notes, U. Toronto]

Syntax and semantics

Syntax

– an alphabet of symbols (e.g. propositions, logical operators)
– grammar rules for creating well-formed formulas

Semantics

The semantic domain varies according to the language:
– state sequences, event sequences, traces, synchronization structures (in specification languages for concurrent systems)
– input/output functions, relations, computations, predicate transformers
(for programming languages)

Types of specifications

- declarative (need not represent a computable function)
- executable (e.g. programming languages)

- behavioral (property-oriented) (e.g., functionality, reactivity)
  – describe system behavior with respect to properties that must be satisfied
- structural (model-oriented) (e.g. diagrams, connectors, hierarchy)
  – build a model of the system using precise mathematical notions (sets, functions, predicate logic)

Sometimes, the same language is used for specification and model (implementation)
$\Rightarrow$ it is possible to do refinement with successive abstraction levels

Properties of specifications

- unambiguous: has a well-defined meaning (NOT: language without formal semantics, natural language, graphical schemes with several interpretations)
- consistent (non-contradictory)
  - there exists at least an object that satisfies it
- may be incomplete
  - can be nondeterministic or leave behavior up to implementation

If the language has a system for *logical inference*, one can prove properties starting from the specification (before building a model)

Specification: the Z language

– based on first-order logic and set theory
– functional, declarative description
– used extensively for industrial projects in the U.K.

	$FindPhones$
$PhoneDB$	$\Xi PhoneDB$
$members : \mathbf{P} Person$	$name? : Person$
$telephones : Person \leftrightarrow Phone$	$numbers! : \mathbf{P} Phone$
$\text{dom}\ phones \subseteq members$	$name? \in \text{dom}\ phones$
	$numbers = phones(\lvert\{name?\}\rvert)$

– a *schema* (PhoneDB) (states + possibly transitions),
and an *invariant*
– operations that change the state (Δ) or don't (Ξ)

Specification: the Larch language

[Guttag, Hornig, Garlan, MIT/DEC SRC]:
description with 2 parts/languages

1. language-independent abstraction (specification)
2. interface specification for modules in a given language

```
Table: trait
  includes Integer
  introduces
    new: -> Tab
    add: Tab, Ind, Val -> Tab
    lookup: Tab, Ind -> Val
  asserts \forall i, i1: Ind, v: Val, t: Tab
    \not (i \in new);
    i \in add (t, i1, v) == i = i1 \/ i \in t
    lookup(add(t, i, v), i1) ==
        if i = i1 then v else lookup(t, i1)
```

The Larch language (cont.)

Interface specification for the C language

```
mutable type table
uses Table(table for Tab, char for Ind,
         char for Val, int for Int);
constant int maxTabsize;
table table_create(void) {
  ensures result' = new /\ fresh(result);
}
char table_read(table t, char i)
  requires i \in t^;
  ensures result = lookup(t^, i);
}
```

– defines preconditions and postconditions

– interface stays at abstract level (without algorithms)

Specification other languages

VDM (Vienna Development Method

– originates from the efforts of the IBM Vienna group in the 70's
– similar and related to Z

B

– developed by Jean-Raymond Abrial (France)
– as opposed to Z, has strong automated tool support
– preconditions / postconditions, invariants, refinement
– support for automated code generation
– industrial usage (Paris metro, Alsthom, $n \cdot 10$kloc)

Interface specification notions have been directly incorporated in some programming languages, e.g., Eiffel (design by contract)

Modeling of concurrent systems

Two main approaches:
– traditional imperative programming + add-ons for concurrency (semaphores, monitors, rendezvous communication, etc.)
– concurrent computation model, based on process interaction ("*indivisible interaction*")

Communication and concurrency are complementary notions [Milner]

- Communicating Sequential Processes [Hoare]
- Calculus of Communicating Systems [Milner]

Modeling: Communicating Sequential Processes (CSP)

Example [Hoare]: chocolate vending machine with coins

Alphabet: $\alpha_V = \{in1p, in2p, small, large, out1p\}$
Behavior:

$$V = (in2p \rightarrow (large \rightarrow V | small \rightarrow out1p \rightarrow V) \\ \quad | in1p \rightarrow small \rightarrow V)$$

or, formally:

$$V = \mu X.(in2p \rightarrow (large \rightarrow X | small \rightarrow out1p \rightarrow X) \\ \quad | in1p \rightarrow small \rightarrow X)$$

(unique solution of above equation)

CSP: formalism (process algebra) centered on actions
with nondeterminism, synchronous composition, etc.

Modeling: finite-state automata

- Variants:
  - labels on states or on transitions
  - transitions specified as functions or relations
  - augmented or not with variables (data)
- Kripke structure:

  = automaton labeled with *atomic propositions* from a set AP:

$$M = (S, S_0, R, L)$$

- S: finite set of states
- S_0: set of initial states
- $R \subseteq S \times S$: total transition relation
- $L : S \to 2^{AP}$: state labeling function

The notion of correctness

- Generally: the system satisfies a property (specification)
- Behavior is functionally correct.
  - system is seen as implementing an input/output function
  - example formalism: Hoare triplets

$$\{P\} \quad S \quad \{Q\}$$

{ precondition } program(system) { postcondition }

Sample reasoning:

$$\frac{\{P\}\ S_1\ \{Q_1\} \qquad Q_1 \Rightarrow Q_2 \qquad \{Q_2\}\ S_2\ \{R\}}{\{P\} \quad S_1; S_2 \quad \{R\}}$$

The notion of correctness (cont.)

Temporally correct behavior

- for *reactive* systems: conceptually infinite execution
- behavior is defined by a reaction to an input sequence
- specification: e.g. temporal logic
- properties: absence of deadlock, time-bounded reaction, etc.

Examples:

– any request is followed by a response within at most 5 seconds
– any process obtains the resource an infinite number of times
– on any trajectory, at some point a stable state is reached

Verification techniques

Two main categories / approaches:

State space exploration (model checking)

– specification usually given in temporal logic

– exhaustive state-space exploration algorithms verify the truth value of the formula or produce an execution trace as counterexample

– equivalence checking: specification is also a (more abstract) model

Theorem proving

– representation in a logical system with axioms and deduction rules

– the analyzed domain is also represented by axioms and rules (a *theory*)

– mechanized theorem proving: manually guided or automated