Formal Specifications of Complex Systems

מפרטים פורמליים למערכות מורכבות

236368
Bureaucracy

• Lecturer: Dr. Sarai Sheinvald
  Taub 618,
  sarai@braude.ac.il
  Office hour: by appointment

• Background materials: from the course site
• Lecture/exercise slides + Book chapters
• The lectures have material not on the slides (and the exam can include that material).
• Grade:
  if examgrade < 51,
    then finalgrade=examgrade
  else  finalgrade= .75*examgrade + .25*homework
Basic principles

• Use precise notations to describe real requirements of complex systems
• Allows formal analysis using tools
• Clarifies many issues before they become implementation problems
• Helps explain concepts like “refinement”, “concurrency”, “closed/open” systems, “fault tolerance”, “real time”
• Gives criteria for evaluating spec. methods
• Links SE to CS theory
What are specifications?

- Requirements specification: a contract between the client and the developers; Includes:
  - The operations and their meaning
  - External constraints and assumptions
  - Descriptions of results/output
  - Interactions/synchronization/interference among operations
  - Forbidden situations, Guaranteed situations

- Design specification: a contract among the developers
  - internal divisions, interfaces, assumptions and guarantees, plus the above

- Formal Specifications (usually) deal with functional requirements, and the behavior of the desired system
What are we specifying?

• The collection of possible computations with the (usually nonexistent) system
  • Emphasizing the permitted, forbidden, and required states
    (Bank balance is never negative)
  • Emphasizing the permitted, forbidden, and required sequences of operations
    (Need Initialize before Add)

• **Terminating systems** with a “final result”
  • Mathematical functions (matrix multiplication)
  • Compilers that produce assembly code

• **Reactive systems**: react to requests/input, ongoing
  • Operating systems
  • Email systems
  • Control systems (air traffic, elevators, nuclear reactors..)
  • Telephone switches
Advantages of precise specifications

• Understand open questions/contradictions at an early stage of development

• Prevent misunderstandings

• Essential input to tools
  • For prototyping and simulation
  • For detecting inconsistencies, gaps
  • For testing and verifying implementations
  • For detecting reuse possibilities
Evaluating specification methods--
A good method should be:

• Precise (= unambiguous)
• Accessible (= easy to understand and explain)
• Expressive (= can write all requirements )
• Modular (= can easily add new requirements and extend old ones)
• Hierarchical (= “zoom” capabilities to consider different levels of detail)
• Executable (= tools for simulation/prototyping)
• Minimal (= does not overly restrict the implementation)
Kinds of Properties

• **Safety**: properties maintained in the history of the computation so far
  • Which states (and histories) are allowed and which are forbidden
  • Doing nothing satisfies all safety properties!
  • = nothing bad ever happens

• **Liveness**: properties on guaranteed progress in computations
  • Which states must occur
  • Can be replaced by real-time properties
  • = good things do happen
Examples: safety or liveness?

• Mutual exclusion of critical sections?
• No deadlock (some operation is always possible)?
• No starvation (every active process actually executes)?
• Termination (computations reach halt)?
• Partial correctness?
• Responsiveness (every request is answered)?
Why do we classify?

• Division to kinds of properties (with more divisions given later) indicates which specification tool is appropriate
• Different kinds of properties have different verification and testing techniques
• Connect to extensions (we will see) for
  • Real time specification
  • Fault tolerance in computer networks
Three kinds of specifications

• **Data and transition modeling:**
  • For individual operations or procedures
  • Hoare logic/annotations: I/O with logic
  • OCL for object-oriented systems and classes
  • Z-system specification based on sets; each operation relative to a system state
  • Larch-specification based on word algebra: relations among operations… event based
  • Emphasizes safety properties (plus termination)
Kinds of Specifications (cont.)

• **Control: concurrency, overlap, synchronization**
  • State machines, transition systems
  • Statecharts – hierarchical, concurrent automata
  • LOTOS – a calculus of communicating processes (also CCS, CSP, IP,…)

• **Global liveness (and safety too):**
  • Temporal logic in many versions (CTL, linear, anchored, future/past)
Theory or Practice?

• (Almost) all methods in the course have associated tools, and have been used on real, large-scale systems
• Some connect with techniques for OO modeling (UML) and hardware/firmware verification
• Some are part of industry standards for communication or mobile
• But...the methods are not typically used for “regular” software specification
• Will see plusses and minuses of methods, gain insight into what we mean by specification, modularity, refinement, implementation,...
Course outline

• Using assertions in logic and OCL for OO systems
• Logic and Set theory in Z to describe properties, and for high level modeling of systems and their operations
• Modelling control with state machines and statechart diagrams
• Temporal logics for global properties
• LOTOS for coordination and synchronization in parallel processes—a process algebra
• Algebraic specifications showing relations among operations
• Realtime and fault tolerance
• Using formal specifications in practical tools
Extending Logic for assertions and modelling

• Classic predicate calculus (first order logic) can express many properties used in specification

• Problem #1: Does it really capture our intentions?
  \( \forall i \, \exists j : a[j] = a0[i] \land \forall i : 1 \leq i < n. \, a[i] \leq a[i + 1] \)

• Problem #2: many assertions natural for modelling and requirements are hard to write in logic
  • Push on to a stack
  • Traverse a tree
Input/output specifications

• R. Floyd, 1967. For flowcharts
• C.A.R. Hoare, 1969. For while programs
• Assumptions:
  • a single sequential module
  • computes a function of the input
• Uses first-order predicate calculus over program variables, with standard mathematics.
Standard Mathematic Symbols

- **inequalities**: $<, \cdot, =, >, \geq, \neq$
- **operations**: $+, -, *$
- **aggregations**: $\Sigma, \Pi, n!$
- **standard data types**: $\text{integer}(x), \text{real}(y)$
- **range**: $1..n$
- $x_0$ for the initial value of $x$
- Sometimes, $x'$ for the value of $x$ after an operation
- **can use logical** `one-time variables`
- **Problem**: logic might not really express what we intend---will see throughout the course
Hoare logic notation

\{ P \} \ S \ { Q \}

• Interpretation: if predicate $P$ is true in the state before the code $S$ is executed, and the code executes and terminates, then the predicate $Q$ will be true of the resultant state.

• The idea: a program logic, sound and complete under this interpretation.
Originally: mixes code and logic

• Claim: due to the assumptions, only the initial and final states are significant.
  • \( P \) **Precondition**: required inputs or states.
  • \( Q \) **Postcondition**: relates inputs and outputs.
• A partial correctness assertion
• Is it safety or liveness?
• Total correctness: partial correctness + termination
Hoare Logic Deduction Rules

• Example: conditional statement

\[
\{ P \land B \} S_1 \{ Q \} \\
\{ P \land \neg B \} S_2 \{ Q \}
\]

\[
\{ P \} \text{ if } B \text{ then } S_1 \text{ else } S_2 \{ Q \}
\]

Used by Hoare to prove partial correctness — can be seen as the **specification** of the conditional statement, to be satisfied by any translation to assembly code.
While loop Hoare logic rule

\[
\{ P \land B \} \ S \ { P } \\
(P \land \neg B) \implies Q \\
\underline{\{ P \} \ while \ B \ do \ S \ \{ Q \}}
\]

The purpose of a loop: maintain the loop invariant while taking a step towards the termination condition of the loop.
Annotation

• an assertion between, before, or after statements of code.

• Examples
  
  \[
  \{ a0[1..n] : \text{integer} \}
  
  S
  
  \{ \forall i \exists j: a[j] = a0[i] \land \forall i: 1 \leq i < n. a[i] \leq a[i + 1] \}
  
  What is specified? Is it our intention?
  
  \[
  \{ \text{integer}(x) \land x \geq 1 \}
  
  T
  
  \{ \text{integer}(y) \land y^2 \leq x \land (y + 1)^2 > x \}
  
  Can we write it differently?
Invariant

- An assertion that is true whenever the control is where the assertion appears (also called a local invariant).
- The intended interpretation of an annotation is as an invariant.
- Input/output assertions should be invariants.
- Can annotate unwritten code, as its specification.
- **Global invariant**: true throughout every execution.
An Annotated Linear Search

\{ n \geq 1 \}

\begin{align*}
    i &:= 1; \text{found} := \text{false} ; \\
    \{ n \geq 1 \land i = 1 \land \text{found} = \text{false} \} \\
    \text{while } ( ( i \leq n ) \text{ and not found } ) &\text{ do} \\
    \{ \forall j. 1 \leq j < i. a[j] \neq x \land \neg \text{found} \} \\
    \text{if } a[i] = x &\text{ then found} := \text{true} \\
    \text{else } i &:= i + 1 \\
    \text{od} \\
    \{ ( \text{found} \Rightarrow a[i] = x ) \land ( \neg \text{found} \Rightarrow \forall j. 1 \leq j \leq n. a[j] \neq x ) \} 
\end{align*}
Another possible view

{ n \geq 1 }
i := \ ? ; \ found := \ ? ;
{n \geq 1 \land i = 1 \land \text{found} = \text{false} }\
\text{while } (( i \leq n ) \text{ and not } \text{found} ) \text{ do}
{ \forall j. 1 \leq j < i. a[j] \neq x \land \neg \text{found} }\
...\
{ ( \text{found} \Rightarrow a[i] = x ) }\
\text{od}
{ ( \text{found} \Rightarrow a[i] = x ) \land ( \neg \text{found} \Rightarrow \forall j. 1 \leq j \leq n. a[j] \neq x) }
Annotations as specifications

• Can use the pre- and post conditions as the design specification, and the annotations as restrictions on the implementation

• Have annotations with holes (= program skeleton)

• (The purpose of a loop: maintain the loop invariant while taking a step towards the termination condition of the loop)
Tricks

• Logical/rigid/specification variables: to "remember" values before a code segment
  \[ \{ y = Y \} \ y := y - 7 \ \{ y < Y \} \]

• Auxiliary variables and programs: adding variables and statements just for expressing specifications
  \[
  \begin{align*}
  c &:= 0; \\
  x &:= ... \text{/* a key statement} \ ; c := c + 1; \\
  y &:= .... \text{/* another} \ ; c := c + 1; \\
  ... \\
  \{ c < K \} \text{ “there are less than K key statements”}
  \end{align*}
  \]

• Problem: formalizing exactly what it means...