CSE584: Software Engineering

Lecture 2: Requirements & Specification (A)

David Notkin
Computer Science & Engineering
University of Washington
http://www.cs.washington.edu/education/courses/584/

Requirements & specification

- More software systems fail because they don't meet the needs of their users than because they aren't implemented properly
- Boehm:
  - Verification: Did we build the system right?
  - *Validation: Did we build the right system?*

Our plan of attack: this week

- An overview of the key problems in requirements and specification
- A brief history in proving programs correct
  - An expected panacea for software that didn't pan out
  - But has provided some benefits
- A look at formal specifications, with a focus on two forms
  - Program correctness (as a basis for model-based specifications)
  - Model-based specifications (Z)
  - Overview of state machine based specifications

Our plan of attack: next week

- Analysis of state machine based specifications (model checking)
- A brief overview of requirements engineering issues
- Michael Jackson on video: "The World and the Machine"

Non-functional requirements

- We're simply going to ignore non-functional requirements
  - Performance, ease of change, etc.
- I'm not proud of this, but there is relatively little known about this issue
  - Worthwhile concrete discussion: should an interface's specification (documentation) specify the performance of the operations?
    - Pro: Sure, it's a key property (and people will find it out anyway)
    - Con: No way, since I'm supposed to be able to change an implementation as long as it behaves the same

A key problem: ambiguity

- (You'll have your own favorites along these lines; this is from Jackson's book on our reading list)
- In an airport at the foot of an escalator

Shoes must be worn

Dogs must be carried

In logic it's clear

$\forall \mathtt{x} \bullet (\mathtt{OnEscalator(x)} \Rightarrow$
$\quad \exists \mathtt{y} \bullet (\mathtt{PairOfShoes(y)} \wedge \mathtt{IsWearing(x,y)})$

$\forall \mathtt{x} \bullet ((\mathtt{OnEscalator(x)} \wedge \mathtt{IsDog(x)}) \Rightarrow$
$\quad \mathtt{IsCarried(x)})$

Formalization still leaves open questions

- **Do dogs have to wear shoes?**
- **What are "shoes"? What are "dogs"? What does it mean to wear shoes?**
- **Why do the formalizations say "dogs are carried" and "shoes are worn" while the signs say "must be"?**
- **As Jackson will say in the video (with a different example)**
  - **The formalizations are in the *indicative* mood: statements of fact**
  - **The signs are in the *optative* mood: statements of desire**
  - **This kind of "mood mixing" increases confusion**

"dog" (noun)

- **OED has 15 definitions**
  - **11K words in the full definition**
- **Webster's 11 definitions include**
  - **a highly variable domestic mammal (*Canis familiaris*) closely related to the common wolf**
  - **a worthless person**
  - **any of various usu. simple mechanical devices for holding, gripping, or fastening that consist of a spike, rod, or bar**
  - **FEET**
  - **an investment ... not worth its price**
  - **an unattractive girl or woman**

"shoe" (noun, Webster's): six definitions including

- **an outer covering for the human foot usu. made of leather with a thick or stiff sole and an attached heel**
- **another's place, function, or viewpoint**
- **a device that retards, stops, or controls the motion of an object**
- **a device (as a clip or track) on a camera that permits attachment of accessory items**
- **a dealing box designed to hold several decks of playing cards**

A students' view

Another high-level issue

- **Specification languages that are "closer" to the user decrease the change of building the wrong system**
  - **But increase the chance of building the system wrong**
- **And specification languages that are "closer" to the program do the opposite**

Specification

Program

Formalism

- In the mid-1960's, there was a set of software research — today we call it programming methodology — that was intended (in my view) to solve two problems
  - Decrease ambiguity through the use of mathematics to specify programs
  - Allow us to prove programs correct by showing that a program satisfies a formal specification
- Turing Awards in this area include: Dijkstra (1972), Floyd (1978), Hoare (1980), Wirth (1984), Milner (1991), Pnueli (1996)

Don't be confused…

- I don't believe that this is a practical approach
  - It may be applicable in some rare situations
- But it's a useful basis for some other work
  - And the historical context is important
  - And the technical material of value

Basics of program correctness

- In a logic, write down (this is often called the *specification*)
  - the effect of the computation that the program is required to perform (the *postcondition* Q)
  - any constraints on the input environment to allow this computation (the *precondition* P)
- Associate precise (logical) meaning to each construct in the programming language (this is done per-language, not per-program)
- Reason (usually backwards) that the logical conditions are satisfied by the program S
- A Hoare triple is a predicate {P}S{Q} that is true whenever P holds and the execution of S guarantees the Q holds

Examples

- ```
  {true}
    y := x * x;
  {y >= 0}
  ```
- ```
  {x <> 0}
    y := x * x;
  {y > 0}
  ```
- ```
  {x > 0}
    x := x + 1;
  {x > 1}
  ```

Sequential execution

- What if there are multiple statements
  - {P} S_1;S_2 {Q}
- We create an intermediate assertion
  - {P} S_1 {A} S_2 {Q}
- We reason (usually) backwards to prove the Hoare triples
- A formalization of this approach essential defines the ; operator in most programming languages

- ```
  {x > 0}
    y := x*2;
    z := y/2
  {z > 0}
  ```
- ```
  {x > 0}
    y := x*2;
  {y > 0}
    z := y/2
  {z > 0}
  ```

Conditional execution

- {P}
  if C then S_1
  else S_2 fi
  {Q}
- Must consider both branches
- Ex: compute the maximum of two variables x and y

```
{true}
  if x >= y then
    max := x
  else
    max := y
  fi
{max >= x ∧ max >= y}
```

$$\{P\}\ \texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{Q\}$$
$$\equiv$$
$$\{P \wedge C\}S_1\{Q\} \wedge \{P \wedge \neg C\}S_2\{Q\}$$
```

Be careful!

- `{true}`
  `max := abs(x)+abs(y);`
  `{max >= x ∧ max >= y}`
- This predicate holds, but we don't want it to
  - The postcondition is written in a way that permits satisfying programs that don't compute the maximum of `x` and `y`
  - Side observation: (almost) every specification is satisfied by an infinite number of programs and *vice versa*
- The right postcondition is
  - `{max = x ∨ max = y} ∧`
    `{max >= x ∧ max >= y}`

Another classic mis-specification

- Postcondition for sorting an array
  - `∀ i,j • i < j ⇒ a[i] <= a[j]`
-
```
for i := 1 to n do
  a[i] := i
endfor
```
- `(∀ i,j • i < j ⇒ a[i] ⇒ a[j]) ∧`
  `A = permutation(A')`
  - It's even more complicated if you want to define a stable sorting specification
    - Stable sorting leaves equal keys in the same order as they were

Assignment statements

- We've been highly informal in dealing with assignment statements
  - "Associate precise (logical) meaning to each construct in the programming language …"
- What does the statement `x := E` mean?
  - `{Q(E)} x := E {Q(x)}`
  - If we knew something to be true about `E` before the assignment, then we know it to be true about `x` after the assignment
    - Assuming no side-effects

Examples

- `{y > 0}`
  `x := y`
  `{x > 0}`
- `{x > 0}` [Q(E) ≡ x + 1 > 1 ≡ x > 0]
  `x := x + 1;`
  `{x > 1}` [Q(x) ≡ x > 1]
- `{ ? }`
  `x := y + 5`
  `{x > 0}`
- `{x = A ∧ y = B }`
  `t := x;`
  `x := y;`
  `y := t`
  `{x = B ∧ y = A }`

Loops

- `{P} while B do S {Q}`
- We can try to unroll this into
  - `{P ∧ ¬ B} S {Q} ∨`
    `{P ∧ B} S {Q ∧ ¬B} ∨`
    `{P ∧ B} S {Q ∧ B} S {Q ∧ ¬B} ∨ …`
- But we don't know how far to unroll, since we don't know how many times the loop can execute
  - We're essentially looking for a fixed point of the loop
- For the moment, let's ignore the newest problem we have: dealing with termination

Loop invariants

- The most common approach to this is to find a *loop invariant (*`I`*)*, a predicate that
  - Is true each time the loop head is reached
  - And helps us prove the properties we need
- (It approximates the fixed point of the loop)
- If we have `{P} while B do S {Q}` then find `I` such that
  - `P ⇒ I` -Invariant is correct on entry
  - `{B ∧ I} S {I}` -It's invariant
  - `{¬B ∧ I} ⇒ Q` -Loop termination proves Q

Example

```
                    2 ≤ i ≤ n + 1
                          ∧
                    x = max(a[1],…,a[i-1])
{n > 0}
  i := 2;
  x := a[1];
  while i <= n do
    if a[i] > x then x := a[i];
    i := i + 1;
  end;
{x = max(a[1],…,a[n])}
```

Termination

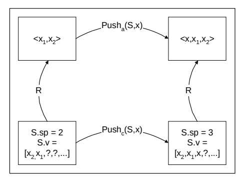
- A Hoare triple for which termination has been proven is ***strongly correct***
- A Hoare triple that is true if termination occurs is ***weakly correct***
- Proofs of termination are usually performend separately from proofs of correctness, and they are usually performed through well-founded sets
  - In this example it's easy, since `i` is bounded by `n`, and `i` increases at each iteration
- Historically, the interest has been in proving that a program *does* terminate
  - Many important programs now are intended *not* to terminate

What's left in classic program correctness?

- Dijkstra's weakest precondition (`wp`) formulation is a more popular alternative to Hoare triples
  - `wp(S,Q)` is the weakest precondition such that if `S` is executed, `Q` will be true
    - `{P}S{Q}` ≡ `P` ⇒ `wp(S,Q)`
- Oh yeah, and procedure calls (with different parameter passing mechanisms), pointers, gotos (!!), concurrency, and other real programming language constructs (not even counting OO features like dynamic dispatch)

Correctness of data structures

- This is a biggie, which you've already seen in Z
  - Showing the implementation of the *BirthdayBook* satisfied the specification
  - Primarily due to Hoare; figures from Wulf *et al.*
- Prove the specifications on the abstract operations (e.g., $\texttt{Push}_a$)
- Prove the specifications on the concrete operations (e.g., $\texttt{Push}_c$)
- Prove the relation between abstract and concrete operations (e.g., `R`), the representation mapping



Example

$\{\neg\texttt{full}(S_a)\}$ $\texttt{Push}_a(S_a,\texttt{x})$ $\{S_a = \langle x\rangle \,||\, S'_a\}$

$\{\neg\texttt{full}(R(S_c))\}$ $\texttt{Push}_c(S_c,\texttt{x})$ $\{R(S_c) = \langle x\rangle \,||\, R(S'_c)\}$

So what?

- I just spent about an hour showing you stuff that I said isn't especially useful
  - It's tedious and error-prone
    - If we can't get our programs right, why should we believe we get our detailed proofs right?
    - One answer: tools, such as proof assistants
  - It's hard with real programming languages and programs
- But it does lay a foundation for
  - Thinking about programs more precisely
  - Applying techniques like these in limited, critical situations
  - Development of some modern specification and analysis approaches that seem to have value in more situations

Formal methods

- The failure of proof of correctness to meet its promises caused a heavy decrease in interest in the late 1970's and the 1980's
- But there has been a resurgence of interest in formal methods starting in the late 1980's and through the 1990's
  - Mostly due to potential usefulness in specification and a few success stories
  - Still not entirely compelling to me, in a broad sense, but definitely showing more promise
  - Key issues to me include
    - Partial specifications
    - Tool support (making specifications "electric" — D. Jackson)
    - What domains, and applied by whom?

Potential benefits

- Increased clarity
- Ability to check for internal consistency
  - This is very different from program correctness, where the issue was to show that a program satisfied a specification
- Ability to prove properties about the specification
  - Related to M. Jackson's refutable descriptions
- Provides basis for falsification (a fancy word for "debugging")
  - Perhaps more useful than verification

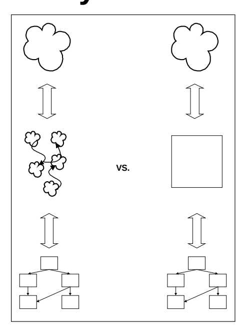
C.A.R. Hoare, 1988

Of course, there is no fool-proof methodology or magic formula that will ensure a good, efficient, or even feasible design. For that, the designer needs experience, insight, flair, judgement, invention. Formal methods can only stimulate, guide, and discipline our human inspiration, clarify design alternatives, assist in exploring their consequences, formalize and communicate design decisions, and help to ensure that they are correctly carried out.

Observation

- From a specification of a small telephone system
  - `"…a subscriber is a sequence of digits. Let Subs be the set of all subscribers … ...certain digit sequences correspond to unobtainable numbers, and some are neither subscribers, nor are they unobtainable."`
- "Only a mathematician could treat the real world with such audacious disdain." —M. Jackson

Anthony Hall's view



And Martyn Thomas sez…

Styles of specifications

- *Model-oriented (e.g. Z, VDM)*
- Algebraic (e.g. OBJ, Larch)
- Process Model (e.g. CCS, CSP)
- *Finite state-based (e.g. Statecharts, RSML)*
- Logical, constructive, multi-paradigm, broad spectrum, ...

Model-oriented

- Model a system by describing its state together with operations over that state
  - An operation is a function that maps a value of the state together with values of parameters to the operation onto a new state value
- A model-oriented language typically describes mathematical objects (e.g. data structures or functions) that are structurally similar to the required computer software

Z (“zed”)

- Perhaps the most widely known and used model-based specification language
- Good for describing state-based abstract descriptions roughly in the abstract data type style
  - Real ADT-oriented specifications are generally does as algebraic specifications
- Based on typed set theory and predicate logic
- A few commercial successes
  - I’ll come back to one reengineering story afterwards

Basics: you already read this

- Static schemas
  - States a system can occupy
  - Invariants that must be maintained in every system state
- Dynamic schemas
  - Operations that are permitted
  - Relationship between inputs and outputs of those operations
  - Changes of states

The classic example

- A “birthday book” that tracks people’s birthdays and can issue reminders of those birthdays
  - There are tons of web-based versions of these now
- There are two basic types of atomic elements in this example
  - [NAME,DATE]
  - An inherent degree of abstraction: nothing about formats, possible values, etc.

Points about the Z reading

- This isn’t proving correctness between a spec and a program
  - There isn’t a program!
- Even the spec without the implementation has value
- The most obvious example is when a theorem is posited and then is proven from the rest of the specification
  - $known' = known \cup \{name?\}$

More points about the Z reading

- The actual notation seems more effective that some others
- The Z is intended to be in bite-sized chucks (schema), interspersed with natural language explanations
  - cf. M. Jackson in the video next week
- ZF (Zermelo-Fraenkel Set Theory), of which the set comprehension operator • is a part, helps increase clarity

Schema calculus: sweet!

- The schema calculus allows us to combine specifications using logical operators (e.g., $\wedge$, $\vee$, $\Rightarrow$, $\neg$)
  - This allows us to define the common and error cases separately, for example, and then just $\wedge$-ing them together
- In some sense, it allows us to get a cleaner, smaller specification

But don’t try this on programs!

- Wouldn’t it be fantastic if we had the equivalent of the schema calculus on programs?
  - Write your error cases separately and then just $\wedge$ them together
  - Write a text editor and a spell checker and integrate them by $\wedge$-ing them together
  - So you want to build a program that doesn’t blow up a nuclear power plant?
    - Just build one that does, and then negate it!
- Programs are not logic
  - Some classes of programming languages come closer than imperative and OO languages

Z/CICS

- Z was used to help develop the next release of CICS/ESA_V3.1, a transaction processing system
  - Integrated into IBM’s existing and well-established development process
  - Many measurements of the process indicated that they were able to reduce their costs for the development by almost five and a half million dollars
  - Early results from customers also indicated significantly fewer problems, and those that have been detected are less severe than would be expected otherwise

1992 Queen’s Award for Technological Achievement

- “Her Majesty the Queen has been graciously pleased to approve the Prime Minister's recommendation that The Queen's Award for Technological Achievement should be conferred this year upon Oxford University Computing Laboratory.
- “Oxford University Computing Laboratory gains the Award jointly with IBM United Kingdom Laboratories Limited for the development of a programming method based on elementary set theory and logic known as the Z notation, and its application in the IBM Customer Information Control System (CICS) product. …”

...

- “The use of Z reduced development costs significantly and improved reliability and quality. Precision is achieved by basing the notation on mathematics, abstraction through data refinement, re-use through modularity and accuracy through the techniques of proof and derivation.
- “CICS is used worldwide by banks, insurance companies, finance houses and airlines etc. who rely on the integrity of the system for their day-to-day business.”

Other success stories

- There are a few other success stories, too (not all Z!)
  - Ex: Garlan and Delisle. "Formal Specification of an Architecture for a Family of Instrumentation Systems" (1995)
  - Aided Tektronix in unifying their understanding and development processes for a broad range of oscilloscopes, function generators, etc.
- Clarke and Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys 28*(4), 1996.
- Craigen, Gerhart, Ralston. An International Survey of Industrial Applications of Formal Methods, Volumes I & II (Purpose, Approach, Analysis and Conclusions; Case Studies), NIST, 1993.

Tool support for Z?

- Some commercial, some freeware
- Formatting (handling all those $\Rightarrow\bullet\oplus\Xi\Delta\notin\emptyset\theta$ characters)
  - And now some html extensions
- Type checkers
- Proof editors, proof assistants, provers
- Specification animations
- …
- Most found through http://archive.comlab.ox.ac.uk/z.html

Analyzing specifications

- It is easy to write specification that are inconsistent
  - This means that you can prove a given theorem both true and false given the specification
  - Alternatively, one can check a hypothesis against the specification and see if it holds (we saw this in Z)
- Daniel Jackson and colleagues have developed a sequence of tools that check Z-like specifications for inconsistencies
- You feed a spec to the tool and it says either
  - Here's a problem, and here's a specific (counter)example of it, or
  - I can't find one (although there may be one)
- Examples include paragraph style mechanisms, telephone switch structures, many more (generally relatively small)
  - Pieces of the ideas appear in Jackson and Chapin. Redesigning Air-Traffic Control: A Case Study in Software Design. *IEEE Software*, May/June 2000

An example (skipping *lots* of steps): Jackson & Vaziri

```
class List {List next; Val val;}
…
void static delete (List l, Val v) {
  List prev = null;
  while (l != NULL)
    if (l.val == v) {
      prev.next = l.next ;
      return; }
    else {
      prev = l ;
      l = l.next ;
}
```

- Procedure for deleting all elements with a given value from a singly linked list
- Relational formulae are automatically extracted
- Fields of List treated as binary relations
  - `next: List → List`
  - `val: List → Val`

Desired properties of delete

- No cells are added
  - `l.*next' in l.*next`
- No cell with value v afterwards
  - `no c:l.*next'|c.val'=v`
- All cells with value v removed
  - `l.*next' = l.*next-{c|c.val=v}`
- No cells mutated
  - `all c|c.val = c.val'`
- No cycles introduced
  - `no c:l.*next|c in c.+next -> no c:l.*next'|c in c.+next'`

Running the tool shows that
- Properties 1, 4 and 5 appear to hold
- But not properties 2 and 3
  - Property 2 fails because the first list cell cannot be deleted
  - Even a simple fix shows another error, in which the last two cells share a value equal to v

Underlying technologies

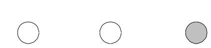
- The Jackson et al. tools have been based on (primarily) two different technologies
  - Model checking
    - Explicit state space enumeration
    - BDD-based symbolic model checking
  - Constraint satisfaction (boolean satisfiability)
    - Stochastic (WalkSAT)
    - Deterministic (Davis-Putnam, SATO, RelSAT)

Algebraic specifications (in one slide)

- The formal model used most frequently for abstract data types
- Define an algebra (over a set of types or "sorts") that gives the semantics of the operations
- Classic example
  - `pop(push(S,x)) = S`
  - `top(push(S,i)) = i`
  - …
- Can define notions of consistency and completeness

Finite state machines

- There is a large class of specification languages based on finite state machines
- Often primarily for describing the control aspects of reactive systems
- The theoretical basis is very firm



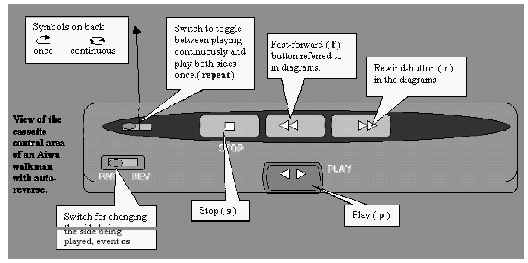
- A finite set of states
- A finite alphabet of symbols
- A start state and zero or more final states
- A transition relation

Many, many models

- Petri nets
- Communicating finite state machines
- Statecharts
- RSML
- …

Walkman example

(due to Alistair Kilgour, Heriot-Watt University)

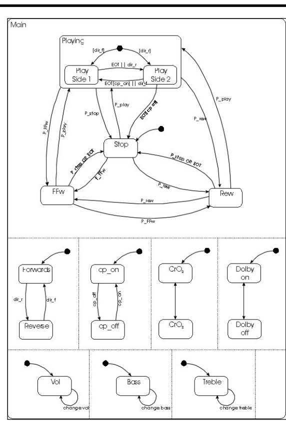


A common problem

- It is often the case that conventional finite state machines blow-up in size for big problems
  - This is especially true for deterministic machines (which are usually desirable)
  - And for machines capturing concurrency (because of the potential interleavings that must be captured)

Statecharts (Harel)

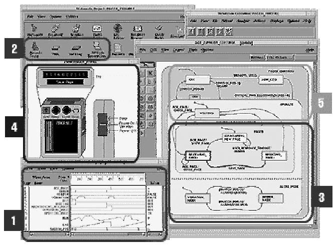
- A visual formalism for defining finite state machines
- A hierarchical mechanism allows for complex machines to be defined by smaller descriptions
  - Parallel states (AND decomposition)
  - Conventional OR decomposition



Tools

- Statecharts have a set of supporting tools from i-Logix (STATEMATE, Rhapsody)
  - Editors
  - Simulators
  - Code generators
    - C, Ada, Verilog, VHDL
  - Some analysis support
- Statecharts (roughly) are a central part of UML

i-Logix screenshot



Analysis

- Given a Statecharts description, how can one tell if it has some desirable properties?
  - For instance, is it deterministic?
  - Are there deadlocks?
  - And domain-specific properties, too
- Perhaps the most promising technology for helping with this is model checking

Which we'll look at next week

- At the beginning of class