CSE P503: Principles of Software Engineering

David Notkin
Autumn 2007

Requirements and Specifications … or

You can't always get what you want
But if you try sometime, yeah,
You just might find you get what you need!
--Jagger & Richards

When I want to know what France thinks, I ask myself. --de Gaulle

If you don’t know where you’re going, it doesn’t matter how you get there. --Anonymous

Tonight: an experiment

- A few interludes on questions and concerns that have been raised by the one-minute papers, email, etc.

- What is an engineer?
- An empirical software engineering research result
- A cool tool

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
  - Is a basis for model-based specifications (below)
- A look at formal specifications, with a focus on two forms
  - 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
- MAYBE: 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
- Topic worthy of a research paper

Dogs and shoes

$\forall$**x • (OnEscalator(x)**$\Rightarrow$
$\exists$y •(PairOfShoes(y)$\wedge$ **IsWearing(x,y))**

$\forall$**x • ((OnEscalator(x)**$\wedge$ **IsDog(x))** $\Rightarrow$
IsCarried(x)

- Do dogs have to wear shoes?
  - What are the types of the variables?
- What are dogs? What does it mean to wear shoes?
  - Designation: "a recognition rule for recognizing some class of phenomenon that you could observe in a domain." [M. Jackson]
- Why do the formalizations say "dogs are carried" and "shoes are worn" while the signs say "must be"?
  - The formalizations are in the *indicative* mood: statements of fact
  - The signs are in the *optative* mood: statements of desire
  - Separating facts from desired behavior can reduce confusion

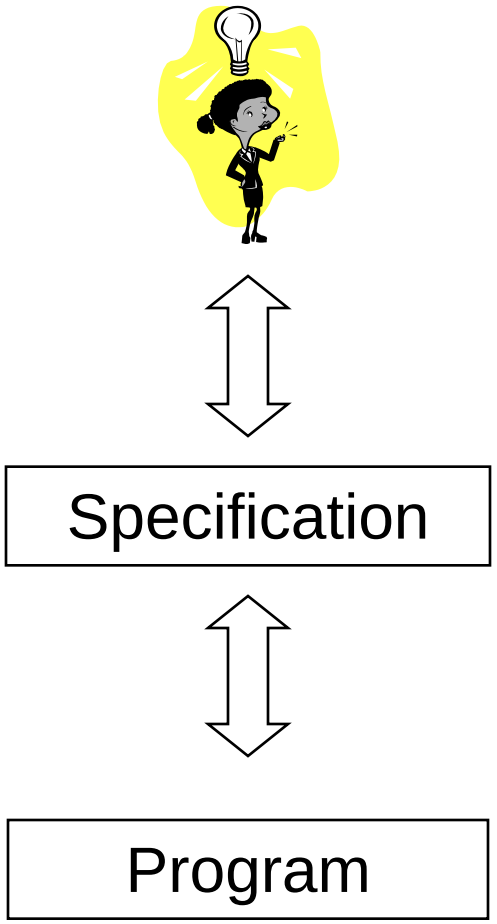
Calvin and Hobbes: on designations

- “Explain Newton’s First Law of Motion in your own words.” … “I love loopholes!”

Pick your poison

- 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

- Why might you pick one over the other?



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 in most situations
  - It may be applicable in some situations
- But it’s a useful basis for some other work
  - And the historical context is important
  - And the technical material is of value

Interlude: what is engineering?

Merriam-Webster

> **"2 a: the application of science and mathematics by which the properties of matter and the sources of energy in nature are made useful to people b: the design and manufacture of complex products <software *engineering*>"**

National Academy of Engineering

> **"Engineering has been defined in many ways. It is often referred to as the 'application of science' because engineers take abstract ideas and build tangible products from them. Another definition is 'design under constraint,' because to 'engineer' a product means to construct it in such a way that it will do exactly what you want it to, without any unexpected consequences."**

“Engineer”

- The title “engineer” is controlled by countries and, within the U.S., by states, in various ways
- In the U.S., with some differences between states, *Professional Engineers* are registered or licensed, based on
  - a degree from an accredited four-year university engineering program
  - passing a standard Fundamentals of Engineering (FE) test on basic engineering principles
  - gaining engineering experience, of about four years, under the supervision of a Professional Engineer
  - passing the Principles and Practice in Engineering (PE) test in a specific engineering discipline (plus engineering ethics)
- A Professional Engineer is authorized to "stamp" engineering documents for a studies, designs, etc.
  - This formally takes legal responsibility
- Some states license per discipline, others in general, but…

Software engineer

- Some states protect all “engineer” titles, some only “Professional Engineer”
- Washington State RCW 18.43.010

  “In order to safeguard life, health, and property, and to promote the public welfare, any person in either public or private capacity practicing or offering to practice engineering or land surveying, shall hereafter be required to submit evidence that he is qualified so to practice and shall be registered as hereinafter provided; and it shall be unlawful for any person to practice or to offer to practice in this state, engineering or land surveying, as defined in the provisions of this chapter, or to use in connection with his name or otherwise assume, use, or advertise any title or description tending to convey the impression that he is a professional engineer or a land surveyor, unless such a person has been duly registered under the provisions of this chapter.”

- Texas is the only state that requires, under some conditions, software engineers to be licensed
  - About two years ago there were 50,513 licensed professional engineers in Texas (some inactive)
  - Of those only 60 are primarily in software engineering, about .1% of the total – and of those, a fifth are in universities

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 that **`Q`** holds

Examples

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

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

- ```
  {x > 0}
    x := x + 1;
  {x > 1}
  ```
  ```

More examples

- ```
  {x = k}
    if (x < 0) x := -x endif;
  {    ?    }
  ```

- ```
  {    ?    }
    x := 3;
  { x = 8 }
  ```

Strongest postconditions

[example from Aldrich and perhaps from Leino]

The following are all valid Hoare triples

- `{x = 5} x := x * 2 { true }`
- `{x = 5} x := x * 2 { x > 0 }`
- `{x = 5} x := x * 2 { x = 10 || x = 5 }`
- `{x = 5} x := x * 2 { x = 10 }`

- Which is the most useful, interesting, valuable? Why?

Weakest preconditions

[example from Aldrich and perhaps from Leino]

Here are a number of valid Hoare Triples

- **{x = 5 && y = 10} z := x / y { z < 1 }**
- **{x < y && y > 0} z := x / y { z < 1 }**
- **{y ≠ 0 && x / y < 1} z := x / y { z < 1 }**

- The last one is the most useful because it allows us to invoke the program in the most general condition
- It is called the *weakest precondition*, **wp(S,Q)** of **S** with respect to **Q**
  - If **{P} S {Q}** and for all **P′** such that **{P′} P′ ⇒ P**, then **P** is **wp(S,Q)**

Sequential execution

- What if there are multiple statements
  - `{P} S1;S2 {Q}`
- We create an intermediate assertion
  - `{P} S1 {A} S2 {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 S1
  else S2
  endif
  {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)}
```

Hoare logic rule: conditional

$$\{P\}\ \texttt{if C then S1 else S2}\ \{Q\}$$

$$\equiv$$

$$\{P \wedge C\}S1\{Q\} \wedge \{P \wedge \neg C\}S2\{Q\}$$

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
  - In essence, 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)}

Out of sorts (example from last time)

- $(\forall\ i,j \bullet i < j \Rightarrow a[i] \Rightarrow a[j])$
  $\wedge\ \ A = permutation(A')$
- It's even more complicated if you want to define a stable sorting specification – one that leaves equal keys in the same order as they were in the original array

Assignment statements

- We've been highly informal in dealing with assignment statements
- 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]
```

More examples

```
{     ?     }
  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
- The most common approach to this is to find a loop invariant, which is a predicate that
  - is true each time the loop head is reached (on entry and after each iteration)
  - and helps us prove the postcondition of the loop
  - It approximates the fixed point of the loop

Loop invariant for **{P} while B do S {Q}**

- Find I such that
  - **P ⇒ I** – Invariant is correct on entry
  - **{B ∧ I} S {I}** – Invariant is maintained
  - **{¬B ∧ I} ⇒ Q** – Loop termination proves Q
- Example

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

Termination

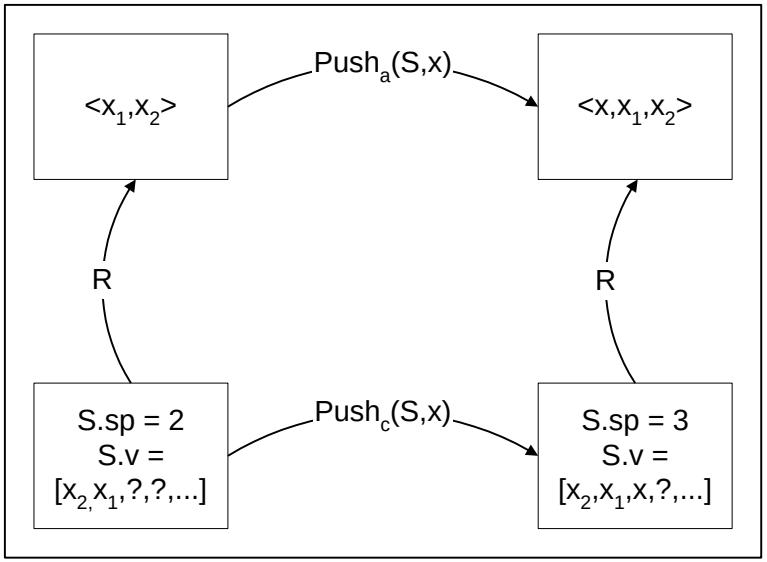
- Proofs with loop invariants do not guarantee that the loop terminates, only that it does produce the proper postcondition **if** it terminates – this is called *weak correctness*
- A Hoare triple for which termination has been proven is *strongly correct*
- Proofs of termination are usually performed 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: but many important programs now are intended not to terminate

What else?

- 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)**
- Need logic rules for procedure calls (with different parameter passing mechanisms), pointers, gotos, concurrency, dynamic dispatch, …

Correctness of data structures

- 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., $\texttt{R}$), the representation mapping



Example

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

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

So what?

- I just spent time 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

Interlude: an empirical result (story)

- Reliability in hardware is often improved by replicating components
- N-version programming was an attempt by Avizienis and colleagues to improve software reliability in a similar way
- In particular, the idea was to have independently produced software act as replicated components with the expectation that there would be a low probability of identical software faults occurring in different versions

Knight and Leveson experiment

- Knight & Leveson evaluated the assumption of independence in N-version programming through a careful experiment
- They found that the assumption of independence of failures failed statistically

Another empirical result: Votta

"Does every inspection need a meeting?"

"At each step in large software development, reviewers carry out inspections to detect faults. These inspections are usually followed by a meeting to collect the faults that have been discovered. However, we have found that these inspection meetings are not as beneficial as managers and developers think they are. Even worse, they cost much more in terms of products development interval and developer's time than anyone realizes.

"Analysis of the inspection and collection process leads us to make the following suggestions. First, at the least, the number of participants required at each inspection meeting should be minimized. Second, we propose ..."

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
- 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 ("proving little theorems about big programs instead of big theorems about little programs" –B. Scherlis) and incremental benefit
  - Tool support (making specifications "electric" — D. Jackson) and automated analysis
  - 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

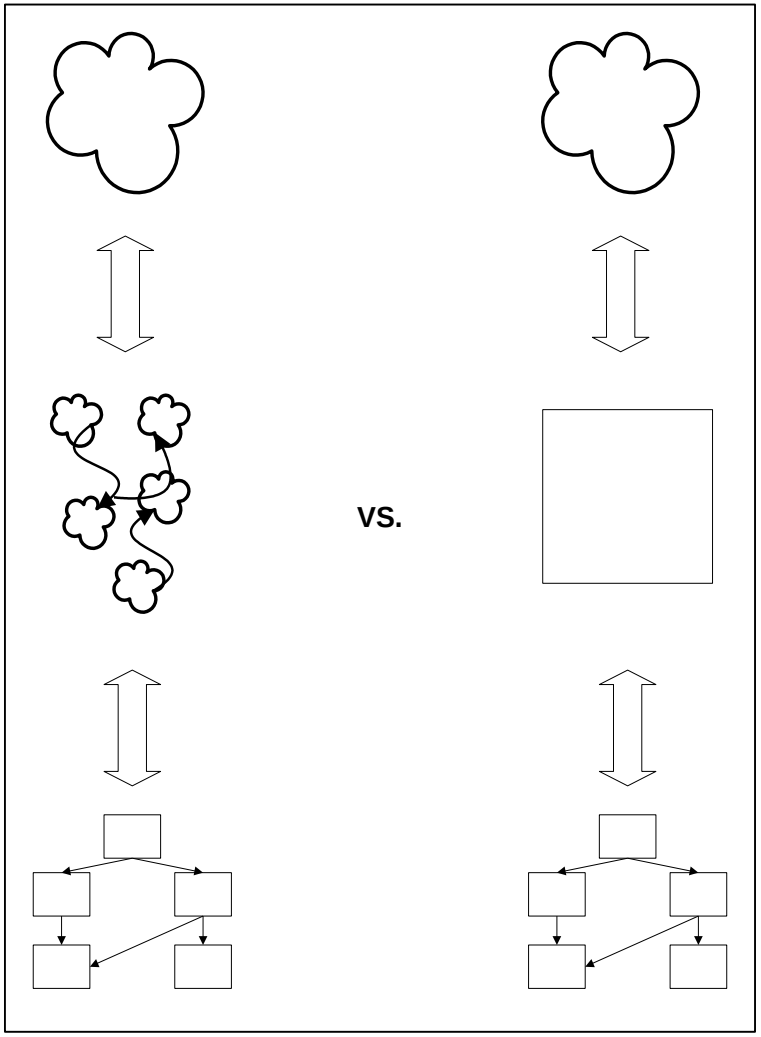
C.A.R. Hoare, 1988

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…

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

- 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.

Off to the whiteboard …

- …for the classic Z BirthdayBook example

Points about Z

- This isn't proving correctness between a specification and a program
  - There isn't a program!
- Even the specification 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?}
- 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

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 – largely functional languages – come closer than imperative and OO languages

Z used to improve CICS/ESA_V3.1

- A broadly used IBM 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 $\Rightarrow \bullet \oplus \Xi \Delta \notin \varnothing \theta$ characters)
  - html extensions
  - ZML
- Type checkers
- Proof editors, proof assistants, provers
- Specification animations
- …

Analyzing specifications

- It is easy to write specifications that are inconsistent
- Daniel Jackson and colleagues have developed a sequence of tools that check Z-like specifications for inconsistencies
- You feed a specification 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
- His Alloy system is the most recent of these tools – we'll play with it some for an assignment (most likely)

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

1. No cells are added
   - `l.*next' in l.*next`
2. No cell with value v afterwards
   - `no c:l.*next'|c.val'=v`
3. All cells with value v removed
   - `l.*next' = l.*next-{c|c.val=v}`
4. No cells mutated
   - `all c|c.val = c.val'`
5. 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)
- They generally use some form of bounded checking based on the *small scope hypothesis*, which "argues that a high proportion of bugs can be found by testing the program for all test inputs within some small scope. … If the hypothesis holds, it follows that it is more effective to do systematic testing within a small scope than to generate fewer test inputs of a larger scope." [Andoni et al.]

Interlude: mylyn from tasktop.com

- Note: co-developed and co-founded by my former student, Gail Murphy
- “Mylyn is the Task-Focused UI for Eclipse that reduces information overload and makes multi-tasking easy. It does this by making tasks a first class part of Eclipse, and integrating rich and offline editing for repositories such as Bugzilla, Trac, and JIRA. Once your tasks are integrated, Mylyn monitors your work activity to identify information relevant to the task-at-hand, and uses this task context to focus the Eclipse UI on the interesting information, hide the uninteresting, and automatically find what's related. This puts the information you need to get work done at your fingertips and improves productivity by reducing searching, scrolling, and navigation.”

Finite state machines

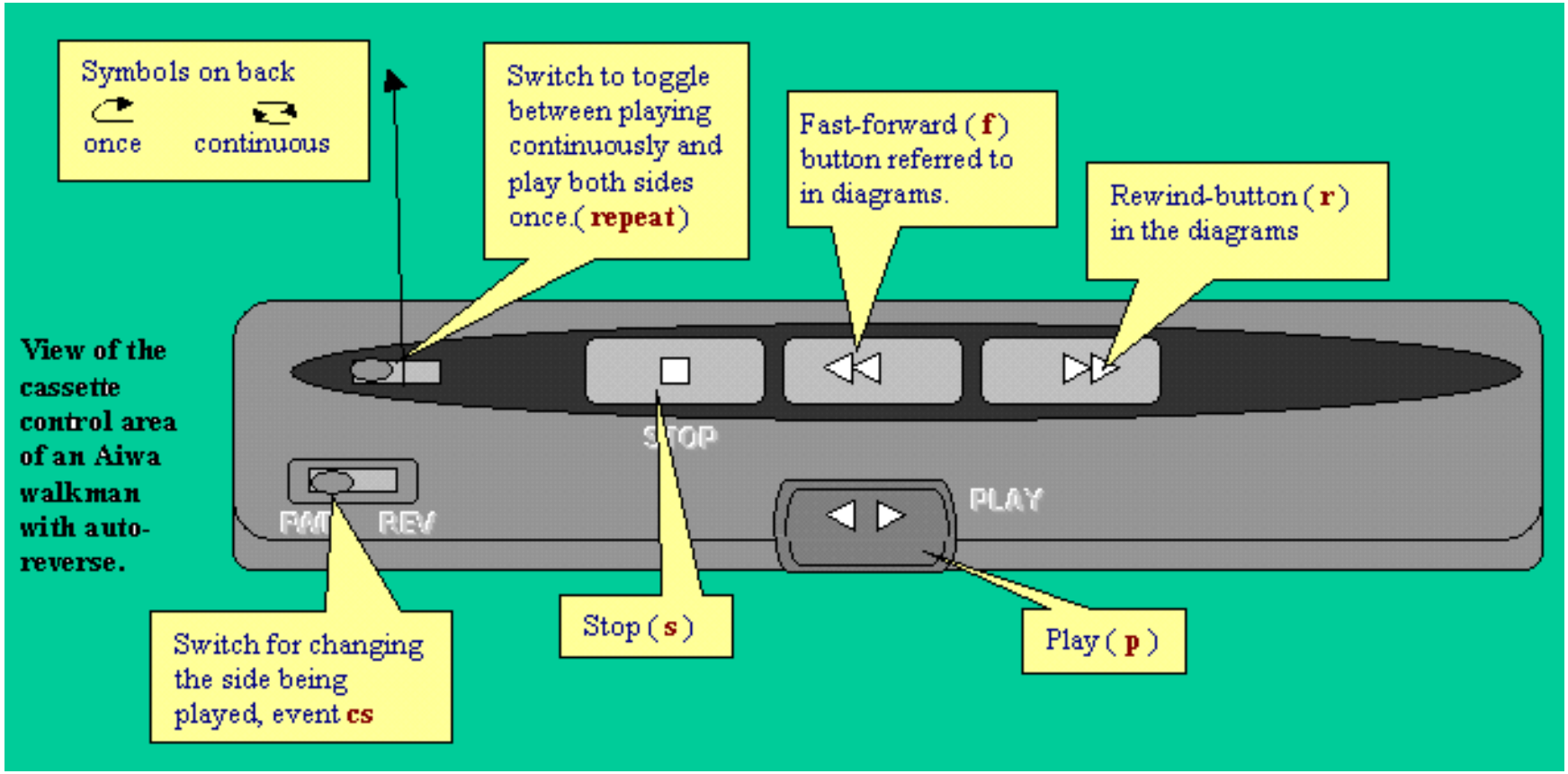
- There is a large class of specification languages based on finite state machines
  - A finite set of states
  - A finite alphabet of symbols
  - A start state and zero or more final states
  - A transition relation
- Often used for describing the control aspects of reactive systems (and much, much more!)
- The theoretical basis is very firm

Many, many models

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

Walkman example

(due to Alistair Kilgour, Heriot-Watt University)

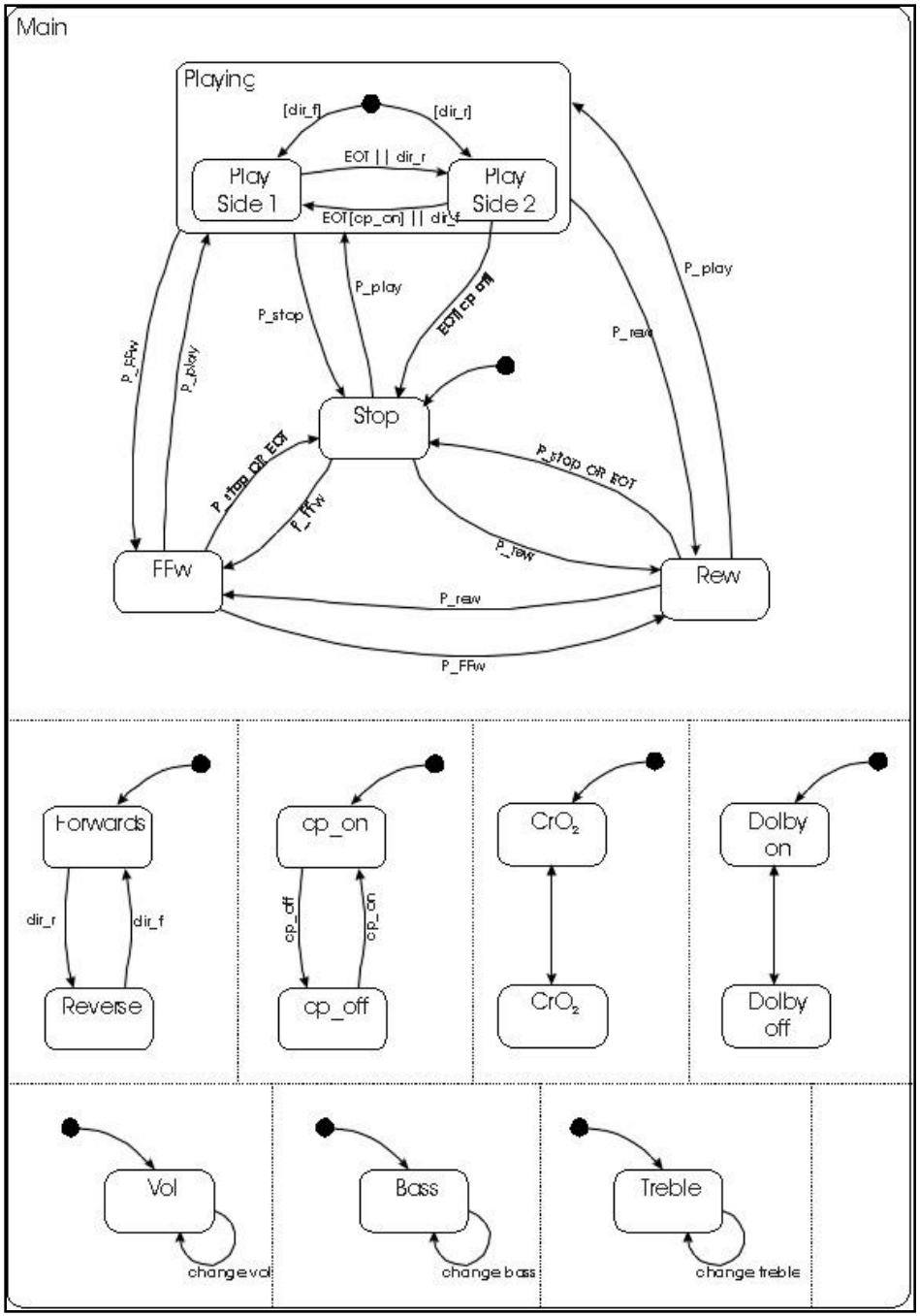


A common problem

- It is often the case that conventional finite state machines blow-up in size for big problems, in two senses
  - The actual description of the machine can get very large
  - The state space represented by the machine can get to be enormous
- This is especially true for
  - deterministic machines (which are usually desirable) and
  - machines capturing concurrency (because of the potential interleavings that must be captured)

Statecharts (Harel)

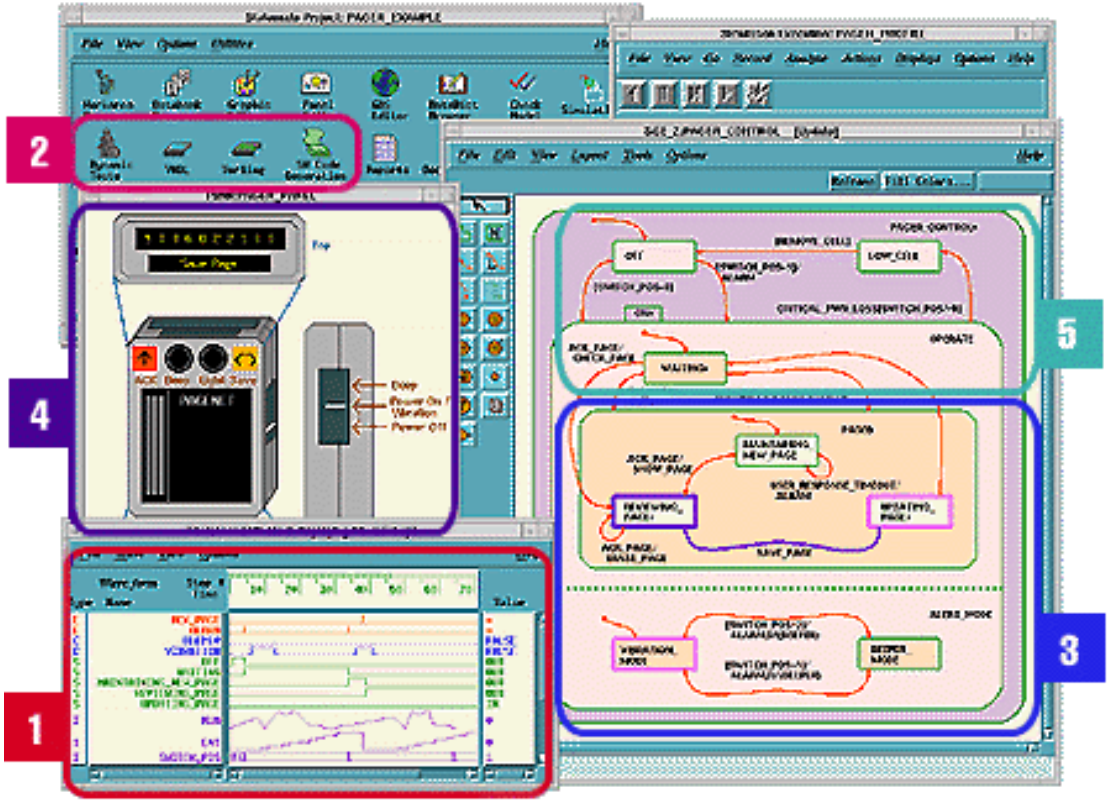
- A visual formalism for defining finite state machines
- A hierarchical mechanism allows for complex machines to be defined by smaller descriptions
  - Parallel states (AND decomposition)
  - Conventional OR decomposition
- Now part of UML


Main
Playing
[dir_f]
[dir_r]
Play Side 1
Play Side 2
EOT || dir_r
EOT[cp_on] || dir_f
P_play
P_stop
P_play
EOT[cp off]
P_rew
P_play
P_FFw
P_play
Stop
P_stop OR EOT
P_stop OR EOT
P_FFw
P_rew
FFw
Rew
P_rew
P_FFw
Forwards
dir_r
dir_f
Reverse
cp_on
cp_off
cp_on
cp_off
CrO2
CrO2
Dolby on
Dolby off
Vol
change vol
Bass
change bass
Treble
change treble

Tools

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

i-Logix screenshot (old)



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
- The most promising technology for helping with this is model checking, which we'll look at next week
  - Model checking has also moved into applications to code as well as specifications