Course Housekeeping

- Class meeting schedule:
  - Tuesday, Feb. 19
  - Thursday, Feb. 21
  - Friday, Feb. 22
  - Tuesday, Feb. 26
  - Thursday, Feb. 28
- Assignment #1
- Assignment #2
Class Meeting Schedule

- Tuesday, Feb. 19
  - NO CLASS (UMass Monday)
- Thursday, Feb. 21
  - Class as usual
- Friday, Feb. 22
  - At 12 Noon: Recording the Feb. 26 lecture
  - In R. 142
- Tuesday, Feb. 26
  - Lecture will be playback of prerecorded lecture
- Thursday, Feb. 28
  - Class as usual

Assignment #1

- Being graded now
- Evaluations to be returned electronically
  - Probably by the end of the weekend
- There will be a letter grade
  - Relatively lightly weighted in final grade
- Written comments and guidance are important
  - Suggestions for widening/narrowing scope
  - Suggestions for adding new stakeholder types
  - Suggestions for reconsidering their stakes
Stakeholders and Their Stakes

• For this assignment, a Stakeholder is a party that has a stake/interest/concern about the SOFTWARE
  – We are not particularly interested in the business for which the software was created
• What stake does an investor have in the SOFTWARE?
  – Not the business
• We will see that stakeholder stakes in the SW translate into software requirements
  – Business stakes translate into business requirements
• We will see that the stakeholder stakes also translate into testing and analysis requirements

Example: An ecommerce system

• Investor wants to make money from the business
  • Incorrect software could cause items to be sold below cost, losing the business money
    • Thus, investor has a stake in correctness
      » The cost of expensive software could cut into investor’s return on investment
        • Thus, investor has a stake in software being developed quickly, cheaply (e.g. less testing)
  • Investor wants to gain market share
    » Fragile software could go down often, causing loss of market share
      • Thus, investor has a stake in robustness
Assignment #2

• Formalize some aspects of the system described informally in assignment #1
  – Provide rationale for choice: e.g. what stakeholder issues are to be addressed
• Focus on evaluation of suitability of formalism for describing what you had in mind
• Quantity of graphs is not main goal of assignment
• Appropriateness of notation is important
• Evaluation is VERY important

Some Examples of Diagrams seen in Assignment #1
Introduction

Millions of Americans watch cable television every evening and billions of dollars are spent by advertisers to sell their products on the air. What is often overlooked during this process are the software systems that lay at the heart of this system. There are two high-level components of the system: the component that transfers data from content providers (television networks) to the content service (cable company), and the component that transfers said data from the content service to a viewer’s (cable customer) home and allows them to navigate and manage the content. An overview of this process can be seen in figure one. The task is inherently distributed; viewers reside along a wide geographic area, as do content providers.

Description of Stakeholders

Throughout the process there are many stakeholders who have great diversity in their interests and requirements. For this document we will define the following stakeholders:

Content providers

Companies such as GE and Viacom who control the channels, provided such as CNN, ESPN and NBC. The principal interest of content providers is to maximize the number

Building Automation Systems & Controls

CS 520/620 Spring 2013 Univ. of Massachusetts
Copyright L. Osterweil, all rights reserved
These are strongly evocative
Leave room for different interpretations

How to be:
  Evocative
  Precise
  Detailed
  Complete

Modeling Data and Types
Representation of Data/Objects

- Complement to emphasis on representation of activities
  - Foregoing representations all focussed on activities
  - Weak capabilities for describing data and objects
  - Seen mostly as effects of activities
  - Numerous places where data descriptions were needed
    » eg. Request List in elevator example
      ▪ Supposed to be sorted (which way?)
      ▪ Elements had fields (what types?)

- Problems in doing this well
  - What information needed/what questions need answers?
    » Hierarchical decomposition of data
    » Legal actions on data
    » Typing information
  - What forms of representation will be useful?
    » Natural language
    » Diagrammatic
    » Formal language

Primitive Data Representations

Record Structures

Array Structures

Linked List Structures
Specification without Graphs and Diagrams

• The graphs are pictures of relations
• The semantics are provided by the relations
• What do we need the graphs for?
  – Clarity
  – Communication with non-CS people
• Do we need the graphs for communication with CS types?
• Might be more precise, concise
• Not everyone is “visual”

Use of Formalism Alone

• Main advantages are precision and rigor
• Semantic breath of scope is possible.
  – Some things are hard to draw pictures of
• Main drawback is lack of clarity
  – At least for non-technical people
Formal Languages Approach

• Represent (model) specific aspects of software system using established formalisms (eg. set theory, logic, algebra) to provide the semantics
• Focus on some aspect(s), ignore the others
  – To provide clear understanding of that aspect
  – Avoid clutter
  – Provide rigor and detail about modeled aspect(s)
• We have already seen example(s) of this:
  – Use of finite mathematics, logic, graph theory to provide semantics for diagrams

Advantages of Formal Language

• Diagrams support clarity, good for customers, ??
  – Pictures support intuitive reasoning
  – Help identify gaps, shortcomings, weaknesses
  – Suggest truths, theorems, facts
  – But are generally based upon very weak semantics
    » Lack breadth of semantics
    » Often lack precision and detail
• Formal Languages, good for developers, ???
  – Strength is precision and rigor
  – Broad semantics are possible
  – Often feature considerable detail (that may interfere with clarity)
Defining Data Semantics with Formal Languages

- Focus on defining data types
  -- Data type is a set of data instances all having some common characteristics and properties
  -- Define the set, and characteristics

- User's (client's)-eye view of the data types to be used

- Describe the "accessing primitives" / "operators", "methods", functions providing the only mechanisms for manipulating instances of a given type
  -- Dual notion to describing functions in terms of their data inputs and outputs

- Goal: Specify the types without specifying their implementation

- Being rigorous helps separate (even slightly) different notions of a data type from each other

Abstract Data Type Definition Approaches

- Natural language
- Diagrams
- Finite State Machines
- Axiomatic Set Theory
- Algebras

Algebraic Specification

- Draws upon the semantics of algebra to form the basis of the semantics of data types
- Define a type as being the elements of an algebra
- Define the type in terms of how its functions interact with each other
- Consists of two parts:
  --Function list
  --Function interrelations:

  - Function list: function templates
  - Function interrelations: how the functions interact with each other

Algebraic Specification of a Stack

FUNCTION LIST:

RELATION LIST:
Algebraic Specification of a Stack

FUNCTION LIST:

CREATE: \rightarrow \text{STACK}

PUSH: \text{STACK} \times \text{INTEGER} \rightarrow \text{STACK}

TOP: \text{STACK} \rightarrow \{ \text{INTEGER} \cup \text{INTERR} \}

POP: \text{STACK} \rightarrow \{ \text{STACK} \cup \text{STACKERR} \}
Algebraic Specification of a Stack

FUNCTION LIST:

CREATE: \rightarrow \text{STACK}

PUSH: \text{STACK} \times \text{INTEGER} \rightarrow \text{STACK}

TOP: \text{STACK} \rightarrow \{ \text{INTEGER} \cup \text{INTERR} \}

POP: \text{STACK} \rightarrow \{ \text{STACK} \cup \text{STACKERR} \}
Algebraic Specification of a Stack

FUNCTION LIST:

CREATE: \( \rightarrow \) STACK

PUSH: STACK \( \times \) INTEGER \( \rightarrow \) STACK

TOP: STACK \( \rightarrow \) \{ INTEGER \( \cup \) INTERR \}

POP: STACK \( \rightarrow \) \{ STACK \( \cup \) STACKERR \}

RELATION LIST:

TOP (PUSH(s,i)) = i
TOP (CREATE) = INTERR
A Different Stack

FUNCTION LIST:

CREATE: \( \rightarrow \text{TSTACK} \)

PUSH: TSTACK \( \times \) INTEGER \( \rightarrow \) TSTACK

TOP: TSTACK \( \rightarrow \) \{ INTEGER \( \cup \) INTERR \}

POP: TSTACK \( \rightarrow \) \{TSTACK \( \cup \) STACKERR\}

ISEMPTY: TSTACK \( \rightarrow \) \{TRUE, FALSE\}

RELATION LIST:

TOP (PUSH(\(s,i\))) = \(i\)

TOP (CREATE) = INTERR

POP (PUSH \((s,i)\)) = \(s\)

POP (CREATE) = STACKERR

ISEMPTY(CREATE) = TRUE, ELSE = FALSE

Yet Another Stack

FUNCTION LIST:

CREATE: \( \rightarrow \text{LSTACK} \)

PUSH: LSTACK \( \times \) INTEGER \( \rightarrow \) LSTACK

TOP: LSTACK \( \rightarrow \) \{ INTEGER \( \cup \) INTERR \}

POP: LSTACK \( \rightarrow \) \{LSTACK \( \cup \) STACKERR\}

LENGTH: LSTACK \( \rightarrow \) INTEGER

RELATION LIST:

TOP (PUSH(\(s,i\))) = \(i\)

TOP (CREATE) = INTERR

POP (PUSH \((s,i)\)) = \(s\)

POP (CREATE) = STACKERR

LENGTH (LSTACK) = ........
What is this good for?

- Focus on the computational aspects of this type
- Provides simplification rules
  - E.g. defines inverses
  - Helps support reasoning about execution traces involving stacks

Example

Create stack;
sum := 0;
I := read ( );
push (stack, I);
Do forever
  if ( top (stack) = sentinel) then exit;
  sum := sum + top (stack);
  pop (stack);
  I := read ( );
  push (stack, I);
end Do;
Print (sum);
Example

Create stack;
sum := 0;
I := read ( );
push (stack, I);
Do forever
  if ( top (stack) = sentinel) then exit;
  sum := sum + top (stack);
  pop (stack);
  I := read ( );
push (stack, I);
end Do;
Print (sum);

Is stack empty when Print Statement is encountered?

Example: push(pop(push(pop(create, I)),I)),I)
Example

Create stack;
sum := 0;
I := read ( );
push (stack, I);
Do forever
    if ( top (stack) = sentinel) then exit;
    sum := sum + top (stack);
    pop (stack);
    I := read ( );
push (stack, I);
end Do;
Print (sum);

Example: push(pop(push(pop(create, I),I),I))

Example

Create stack;
sum := 0;
I := read ( );
push (stack, I);
Do forever
    if ( top (stack) = sentinel) then exit;
    sum := sum + top (stack);
    pop (stack);
    I := read ( );
push (stack, I);
end Do;
Print (sum);

Example: push(pop(push(create, I),I),I)
Example

Create stack;
sum := 0;
I := read ( );
push (stack, I);
Do forever
  if ( top (stack) = sentinel) then exit;
  sum := sum + top (stack);
  pop (stack);
  I := read ( );
push (stack, I);
end Do;
Print (sum);

Is stack empty when Print Statement is encountered?
Example: push(pop(push(create, I)), I)

Example

Create stack;
sum := 0;
I := read ( );
push (stack, I);
Do forever
  if ( top (stack) = sentinel) then exit;
  sum := sum + top (stack);
  pop (stack);
  I := read ( );
push (stack, I);
end Do;
Print (sum);

Is stack empty when Print Statement is encountered?
Example: push(create, I)
Example

Create stack;
Sum := 0;
I := read ( );
push (stack, I);
Do 10 times
   if ( top (stack) = sentinel) then exit;
   sum := sum + top (stack);
   I := read ( );
   push (stack, I);
end Do;
Print (sum);

Other Types?

• Polynomial
• Set/bag
• Dequeue
• Tree
• Editor
Previous definition doesn’t address some characteristics

Other formalisms make it easier to specify other characteristics

Axiomatic Set Theory ADT Specification

• Semantics of the data type derived from semantics of axiomatic set theory
• Supports rigorous reasoning about design and code
• Describes data type in terms of its abstract behaviors
• Describes accessing functions in terms of relations to each other
• Hard to write correct specifications (and know it)
• Reading them is hard too
Axiomatic Specification of a Stack

STACK(CREATE)

It is true that you can create instances
Of objects of type “STACK”
Axiomatic Specification of a Stack

STACK(s) ∨ INTEGER(i) \rightarrow PUSH(s,i) \neq CREATE

Push makes a non-empty stack
Axiomatic Specification of Stack

\[
\text{STACK}(s) \land \text{STACK}(s') \land \text{INTEGER}(i) \implies [ \text{PUSH}(s,i) = \text{PUSH}(s',i) \implies (s = s') ]
\]

Pushing doesn’t change what was already there (not covered by algebraic specification)
Axiomatic Specification of Stack

STACK(s) ∧ INTEGER(i) ⇒ TOP(PUSH(s,i)) = i

Last in, first out
Axiomatic Specification of Stack

STACK(s) ∧ INTEGER(i) ⇒ POP(PUSH(s,i)) = s

Pop and Push are inverses of each other
Axiomatic Specification of Stack

\[
\begin{align*}
\text{TOP(CREATE)} &= \text{INTEGERERROR} \\
\text{POP(CREATE)} &= \text{STACKERROR}
\end{align*}
\]

Expected behavior of empty stack
Axiomatic Specification of a Stack

STACK(s) ∧ INTEGER(i) ⇒

STACK(PUSH(s,i)) ∧

[POP(s) ≠ STACKERROR ⇒ STACK(POP(s))] ∧

[TOP(s) ≠ INTEGERERROR ⇒ INTEGER(TOP(s))]
Axiomatic Specification of a Stack

∀ A [ A(CREATE) ∧ (∀ s)(∀ i)

[STACK(s) ∧ INTEGER(i) ∧ A(s)

⇒ A(PUSH(s,i)) ∧ [s ≠ CREATE ⇒ A(POP(s)) ] ]

⇒ ∀ s [STACK(s) ⇒ A(s) ]

Anything that uses these three functions this way is a stack--although more elaborate stacks might have more functions
Full Axiomatic Specification of a Stack

\[
\begin{align*}
\text{STACK}(\text{CREATE}) & \\
\wedge \text{STACK}(s) \wedge \text{INTEGER}(i) & \implies \text{PUSH}(s,i) \neq \text{CREATE} \\
\wedge \text{STACK}(s) \wedge \text{STACK}(s') \wedge \text{INTEGER}(i) & \\
& \implies [\text{PUSH}(s,i) = \text{PUSH}(s',i) \implies (s = s')] \\
\wedge \text{STACK}(s) \wedge \text{INTEGER}(i) & \implies \text{TOP}(\text{PUSH}(s,i)) = i \\
\wedge \text{STACK}(s) \wedge \text{INTEGER}(i) & \implies \text{POP}(\text{PUSH}(s,i)) = s \\
\wedge \text{TOP}(\text{CREATE}) & = \text{INTEGERERROR} \\
\wedge \text{POP}(\text{CREATE}) & = \text{STACKERROR} \\
\wedge \text{STACK}(s) \wedge \text{INTEGER}(i) & \implies \\
& \text{STACK}(\text{PUSH}(s,i)) \wedge \\
& [\text{POP}(s) \neq \text{STACKERROR} \implies \text{STACK}(\text{POP}(s)) ] \wedge \\
& [\text{TOP}(s) \neq \text{INTEGERERROR} \implies \text{INTEGER}(\text{TOP}(s)) ] \\
\wedge \forall A \big[ A(\text{CREATE}) \wedge (\forall s)(\forall i) \\
& [\text{STACK}(s) \wedge \text{INTEGER}(i) \wedge A(s) \\
& \implies A(\text{PUSH}(s,i)) \wedge [s \neq \text{CREATE} \implies A(\text{POP}(s)) ] \\
& \implies \forall s [\text{STACK}(s) \implies A(s) ] \\
\end{align*}
\]

What is this good for?

- Providing rigorous specifications about types to be built
- Providing rigorous specifications for modules to be built
- Supports logical reasoning about software using this type
- Proving rigorous theorems about software
- We will see more about this soon
Course Housekeeping

• Class meeting schedule:
  – Thursday, Feb. 21 (today)
  – Friday, Feb. 22
  – Tuesday, Feb. 26
  – Thursday, Feb. 28
• Assignment #1
• Assignment #2

Class Meeting Schedule

• Thursday, Feb. 21
  – Class as usual
• Friday, Feb. 22
  – At 12 Noon: Recording the Feb. 26 lecture
  – In R. 142
• Tuesday, Feb. 26
  – Lecture will be playback of prerecorded lecture
  – Bobby Simidchieva and Jonathan Leahey will be here
• Thursday, Feb. 28
  – Class as usual
Assignment #1

• Evaluations were returned electronically over the weekend
• Letter grade relatively lightly weighted in final grade
  – Take suggested improvements seriously
  – Suggestions for widening/narrowing scope
  – Suggestions for adding new stakeholder types
  – Suggestions for reconsidering their stakes

Assignment #2

• Should be working on that now
• Adopt stakeholder perspective(s)
• Think about specific stake(s)
• Select specification approach(es) that help make clear to your selected stakeholder(s) what you have in mind for this system in the area of the selected stake
• Due on Friday, 1 March at 11:59PM
Review of Some Past Approaches

- Diagrammatic/Pictorial Activity specifications
  - DFG, CFG, FSA, MSC (Ladder chart)
  - And combinations of these (e.g. Statemate, UML)
- Data/artifact specifications
  - Algebraic and Axiomatic
- Something that combines elements of the above is next

Other Formal Approaches

- System structure: Its modules, their relations,
  - Z (pronounced "zed")
  - Larch
  - VDL
- Concurrency structure
  - CSP (Cooperating Sequential Processes)
  - TSL (Task Sequence Language)
Z (Pronounced “Zed”)

• Developed at Oxford by Hoare, Spivey, etc.
• Represents WHAT software systems do without specifying HOW
• Uses set theory and function notation
• Describe systems as collections of SCHEMAS
  – inputs and outputs to functions
  – Invariants: statements whose truth is preserved by the functions
• Lots of idiosyncratic notation
• Intent is to make specifications brief, yet clear and precise

The “Birthday Book” Example

• Maintain a repository of information about birthdays
• Consists of (name, birthday) pairs
• Want to add pairs for people whose birthdays are to be remembered
• Want to know whose birthday falls on a given date
• Don’t care about how this is implemented
Example Schema

BirthdayBook

known : \( \mathcal{P} \) NAME

birthday : NAME \( \rightarrow \) DATE

known = dom birthday

This schema describes the STATE SPACE of the system: the space of all states that the system can be in.

Example Schema

Name of the schema

BirthdayBook

known : \( \mathcal{P} \) NAME

birthday : NAME \( \rightarrow \) DATE

known = dom birthday

This schema describes the STATE SPACE of the system: the space of all states that the system can be in.
Example Schema

Name of the schema

“Set of” symbol: This line means known is a set of elements of type NAME

BirthdayBook

known : \( \mathcal{P} \) NAME

birthday : NAME \( \rightarrow \) DATE

known = dom birthday

This schema describes the STATE SPACE of the system: the space of all states that the system can be in
Example Schema

Name of the schema

"Set of" symbol: This line means
known is a set of elements of type NAME

BirthdayBook

known : \( \mathcal{P} \) NAME

birthday : NAME \( \rightarrow \) DATE

known = dom birthday

The invariant part of the schema:
This line means
The set known is the domain of definition
of the function birthday

"function" symbol used to denote a
function from a set to a set
A function from an element to an
element is denoted by

This line means
birthday is a function defined on a
set of NAME and mapping into a set
DATE

This schema describes the STATE SPACE of the system: the
space of all states that the system can be in

Another Schema

This Delta symbol indicates that this schema will describe a state change

AddBirthday

\( \triangle \) BirthdayBook

name? : NAME
date? : DATE

name? \# known
birthday' = birthday U \{name? \rightarrow date? \}
Another Schema

This Delta symbol indicates that this schema will describe a state change

AddBirthday
\[ \triangle BirthdayBook \]

name? : NAME
date? : DATE

When the schema describes a change of state then it is necessary to distinguish between the value of an element before the state change and the value after. The ' denotes the value after the state change.

name? # known
birthday' = birthday U \{name? \rightarrow date? \}

The ? after these symbols indicates that the element will be an input.
Two more Schemas

FindBirthday

Ξ BirthdayBook

name? : NAME
date! : DATE

name? ∈ known
date! = birthday (name?)

Remind

Ξ BirthdayBook

today? : DATE
cards! : ℘ NAME

cards! = \{ n : known \mid birthday(n) = today? \}

Indicates there will be no change in schema state
Two more Schemas

FindBirthday

BirthdayBook

name? : NAME
date! : DATE

name? ∈ known
date! = birthday (name?)

Remind

BirthdayBook

today? : DATE
cards! : ∩ NAME

cards! = { n : known | birthday(n) = today? }

This denotes the set of all elements n, drawn from the set known, such that birthday(n) = today?
What is Tokeneer?

- Software system for controlling access to a high-security area
- Developed so that security and other requirements could be formally verified
- Requirements specification is contained on a large document
  - Many different specification approaches are used
  - We saw some use cases
  - Now see some Z specifications

3 Domain Information

This section describes the existing Tokeneer system, and the existing ID Station. This allows the proposed variant system structure to be understood in context (see section 4).

The ID Station is part of the larger Tokeneer system, as depicted below. Analysis of the interactions within this larger system has not been carried out, and would normally be done to ensure that a full understanding of the system’s true requirements is obtained. Within the scope of this project, and given that the functionality of the system is already well-defined, this step will not be carried out.
There are two system boundaries of interest in the development of this variant: the boundary between the ID Station machine and its environment (including its peripherals); and the boundary between the ID Station core functions and its support functions. These boundaries are expanded below.

Coating ID Station Structure:
The existing ID Station has four connected peripherals and a number of internal drivers and libraries.

The ID Station interfaces to four different physical devices.

---

Praxis

High Integrity
Systems

Tokeneer ID Station

Formal Specification

Reference SP.1229.41.2
Issue 1.4
Page 49

> See: UserEntryContext (p. 44), UserToken (p. 23), DoorLatchAlarm (p. 22), Status (p. 23), AddElementsToLog (p. 33), present (p. 8), UserTokenWithOKAuthCerti (p. 46), UserTokenOK (p. 47), waitingRemoveTokenFail (p. 24)

THEValidateUserToken ≡ ValidateUserTokenOK ∨ ValidateUserTokenFail

> See: ValidateUserTokenOK (p. 48), ValidateUserTokenFail (p. 48), UserTokenToken (p. 45)

6.4 Reading a fingerprint
6.4 Reading a fingerprint

FS/UserEntry.ReadFingerOK

A finger will be read if the system is currently waiting for it and the user Token is in place.

- UserEntryContext
- DoorLatchAlarm
- UserToken
- Stats
- AddElementsToLog

\[
\begin{align*}
\text{status} &= \text{waitingFinger} \\
\text{fingerPresence} &= \text{present} \\
\text{userTokenPresence} &= \text{present} \\
\text{currentDisplay}' &= \text{wait} \\
\text{status}' &= \text{gotFinger}
\end{align*}
\]

> See: UserEntryContext (p. 44), DoorLatchAlarm (p. 22), UserToken (p. 23), Stats (p. 21), AddElementsToLog (p. 33), waitingFinger (p. 24), present (p. 8), wait (p. 15), gotFinger (p. 24)

FS/UserEntry.NoFinger

If there is no finger present then either nothing happens, since we have not allowed sufficient attempts to eat and validate a finger.

FS/Door.UnlockDoor

The door is unlatched by updating the timeouts on the door latch and alarm.

- UnlockDoor
  - DoorLatchAlarm
  - Config

\[
\begin{align*}
latchTimeout' &= \text{currentTime} + \text{latchUnlockDuration} \\
alarmTimeout' &= \text{currentTime} + \text{latchUnlockDuration} + \text{alarmSilentDuration} \\
\text{currentTime}' &= \text{currentTime} \\
\text{currentDoor}' &= \text{currentDoor}
\end{align*}
\]

> See: DoorLatchAlarm (p. 22), Config (p. 19)

FS/Door.LockDoor

The door is explicitly latched and timeouts on the door latch and alarm are reset. Resetting the timeouts to the current time will ensure that the door will be latched directly and the alarm sound if there is a breach of security.

- LockDoor
  - DoorLatchAlarm

\[
\begin{align*}
\text{currentLatch}' &= \text{latched} \\
latchTimeout' &= \text{currentTime} \\
alarmTimeout' &= \text{currentTime} \\
\text{currentTime}' &= \text{currentTime} \\
\text{currentDoor}' &= \text{currentDoor}
\end{align*}
\]
Much More to come in a couple of weeks

**Z Features**

- Z specifies the state of the domain being changed by the software system being described
  - And the ways in which it can be changed
- Like FSA in specifying state changes
- But with strong focus on data
  - Like ADT specifications in that way
- Schemas can be grouped and composed
  - Allows specifying large and complex systems
Z Stakeholders

- Emphasis on what a system is supposed to do
- Indication of how it looks externally
- Provides information of concern to
  - Users, Customers
    » Who want to know what the system does
    » But have to be mathematically sophisticated
  - Implementers
    » Who need to know what they have to build
  - Regulators
    » Who want to be able to reason to be sure that the system is well-conceived
- Less useful to
  - Technically unsophisticated