CS445 / SE463 / ECE 451 / CS645

Software requirements specification & analysis

A reference model for requirements engineering

Spring/Summer 2023

Mike Godfrey & Daniel Berry & Richard Trefler

Overview

Goal: A clear understanding of how requirements relate to the SUD (System Under Development) and its environment

[Much of this is based on the work of P. Zave and M. Jackson with C. and E. Gunter "A Reference Model for Requirements and Specifications" *IEEE Software* 17:3, 37-43, 2000]

Overview

- Topics:
  - Reference model for requirements engineering
  - System, environment, interface
  - Context diagrams
  - Deriving specifications from requirements
  - Zave–Jackson Validation Formula (ZJVF)
  - Domain knowledge

Overview

The parts of the RE Reference Model give you the mental tools to form a domain (context) model for any system you have to build.

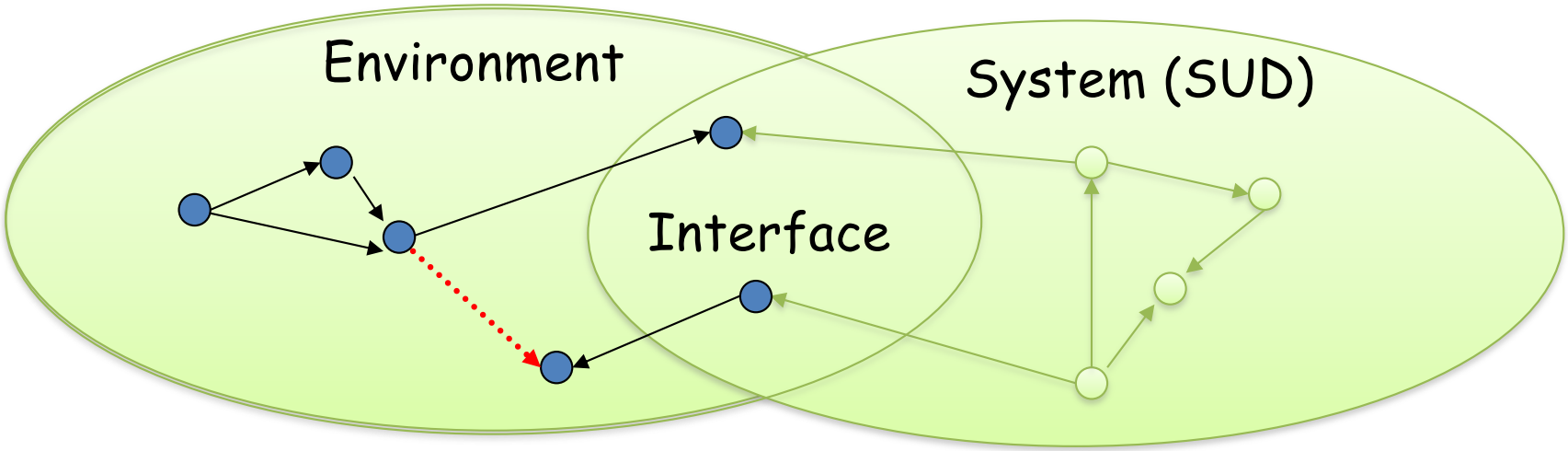
It helps you decide what are the components and where they go in the model, which it turn tells you what you implement and what you don’t.

Reqs, specs, and programs

Hard reality (domain model)

Shared phenomena

Data structures and algorithms



"The world"

The boundary of the Environment is fuzzy.

Reqs, specs, and programs

We view the hardware and software as building a *System*

- that operates within a specified *Environment*
- interacting with it through a set of *Shared Phenomena:*
  - a *Sensor* senses phenomena in the environment (e.g., keyboard clicks)
  - an *Actuator* causes change in the environment (e.g., screen display)

Reqs, specs, and programs

Both the System and the Environment sit in the *World*.

- A System is built to improve the Environment in some way.
- Therefore, everything in the World that is involved in the improvement is part of the Environment.

The system

A *System* can be any socio-technical artifact that is to be constructed

- It can be composed of some mix of software and hardware, processes, and possibly humans.
- If humans are part of the System, then their behavior must be specified and we must accept that they will not behave as specified.

The system

However, in general only the software is easily modifiable, and that is what we will concentrate on.

Hence, we will talk about
Software Engineering and *Software Requirements*,
but we really mean
System Engineering and *System Requirements*.

The environment

We scope the *Environment* to include only those aspects of the real world that are relevant to the particular problem at hand.

Shared phenomena

Shared Phenomena are visible to both the Environment and the System, and form the *Interface* between the two.

- A given interface entity may be sensed or controlled by the Environment or the System, but generally not by both.
  - It serves as a communication bridge from the one to the other.

Shared phenomena

Shared Phenomena...

- Anything that *has* to be in the Interface *has* to be shared by the Environment and the System.
- Anything that *has* to be shared by the Environment and the System *has* to be in the Interface.

We use both ways to make decisions.

A Look Back

The parts of the RE Reference Model give you the mental tools to form a domain (context) model for any system you have to build.

It helps you decide what are the components and where they go in the model:

- in only the environment: you don’t develop
- in the system: you develop
- in both, i.e., in the interface

Park example

Suppose that that the city of Waterloo decides to raise funds by instituting users fees for public parks.

We must implement a complete system of money collection, security, etc.

Park example: Requirements

Informal requirement:

Collect $1 fee from each human park user on entry to park.

- Ensure that no one may enter park without paying.
- Ensure that anyone who has paid may enter the park.

These are requirements, stated in terms of only the environment, independent of any System.

Park example

There are multiple levels of *how*:

- Solution level
- Design level
- Coding level

These names are arbitrary; understand them by example.

Park example: Possible solutions

Solution #1:

- Employ human fee collectors.
- Enforce perimeter security by instituting the Waterloo Park Militia, armed guards who ensure that no one uses a park w/o paying a user fee.

Park example: Possible solutions

Solution #2:

- Use barriers with automated coin collection.
- Use chain link fences for security.
  - There is a barrier (turnstile) through which to enter a park.
  - A person inserts a coin, the barrier unlocks, allowing the person to push the barrier and enter the park.

Park example: Possible solutions

Solution #2:

After some research, we find appropriate barrier and coin collection hardware, but it's all brand new technology. So, we must create the embedded software system.

This solution admits of three different designs based on barrier hardware characteristics:

Park example: Possible designs

Design #1:

Barrier can be turned as many times as desired unless it is locked by control.

Design #2:

Barrier is locked unless it is unlocked by control, when it can then be turned once.

Design #3:

Barrier is locked unless it is unlocked and then turned by control.

Park example: Possible designs

Each of these designs admits of many implementations, each with its own algorithm and programming language.

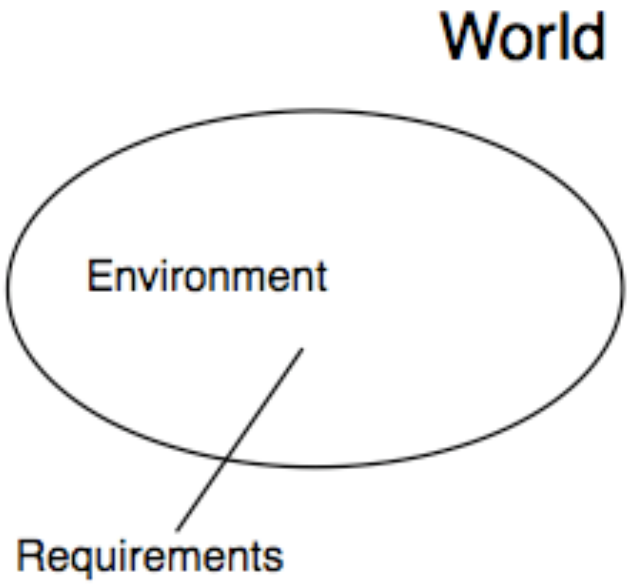
Since this course goes no lower than the solution level, we ignore these implementations.

In any case, by now, you all should be able to program these implementations in your sleep ☺!!!

Reqs, Sol'n, Design, & Implementation

- Reqs: If a user presses the "K" key, then E sees "K" on the screen.
- Sol'n: If the "K" key is pressed, then display "K" on the screen.
- Design: A press of any key causes emission of an ASCII code that is used as an index into a table of bitmaps in a font table, the bit map that is put on screen.
- Implementation: C realization of the Design.

Requirements



Requirements (Reqs) are desired changes to the Environment

So, Requirements are expressed in terms of only Environment phenomena.

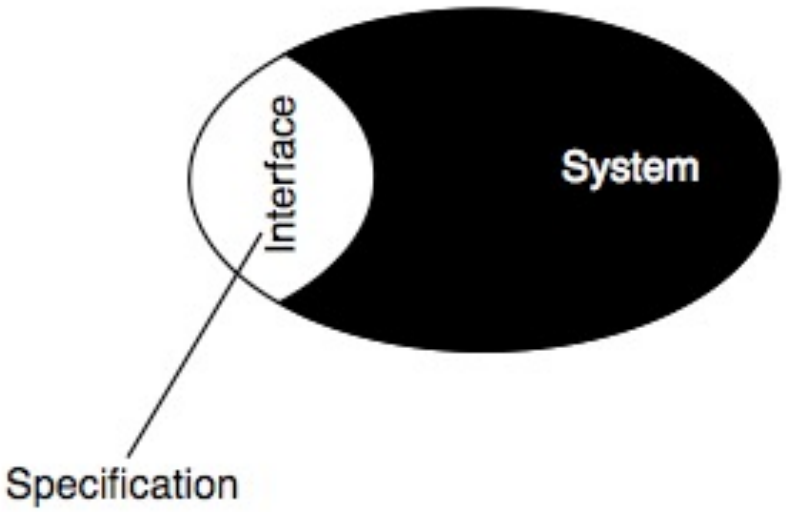
Requirements

Requirements are expressed in terms of only environmental phenomena:

- No one should enter the park without paying.
- Anyone who has paid should be allowed to enter the park.

There is *no* notion of any solution, design, or implementation here.

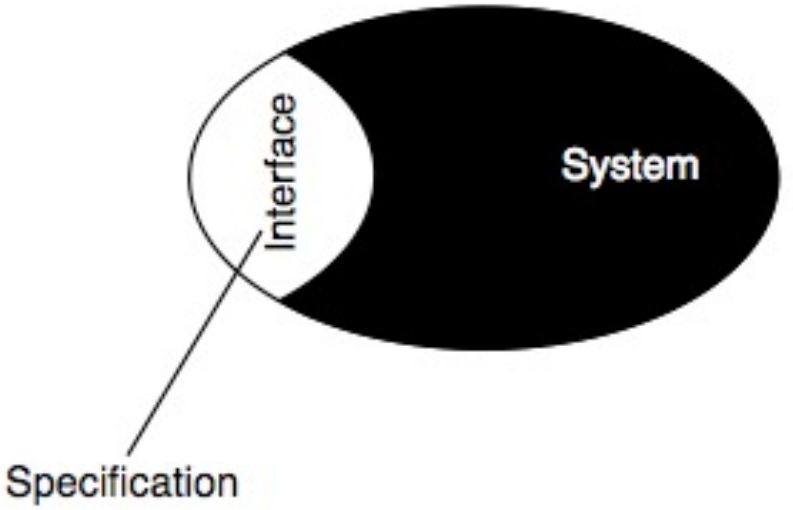
Specification



A *Specification* (Spec) is a description of the proposed behavior of System.

Want to avoid design and implementation bias in the Spec. So, a Spec is expressed in terms of only Interface, i.e., shared, phenomena.

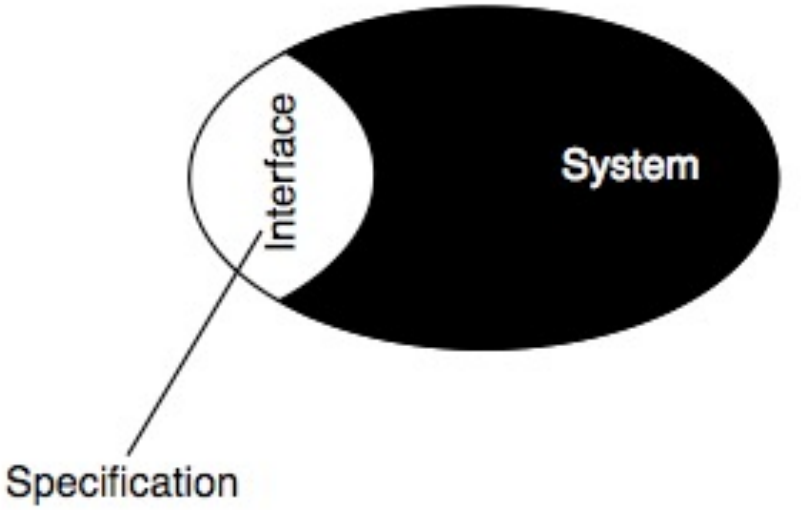
Specification



A Spec is expressed in terms of only Interface phenomena.

A Spec describes how the System should react to Environment events that it can sense. The reaction includes actuating Environment events.

Specification



A Spec is expressed in terms of only Interface phenomena.

A Spec describes how the System should react to Environment events that it can sense. The reaction includes actuating Environment events.

Any Environment entity that must be in a Spec is in the Interface.

Specification

A Spec describes how the system should react to Environment events that it can sense:

For Solution #2:

> If the number of coins inserted is greater than the number of times the barrier has been turned, then the barrier is unlocked.

Specification

A Spec describes how the system should react to Environment events that it can sense:

For Solution #2:

> If the number of coins inserted is greater than the number of times the barrier has been turned, then the barrier is unlocked.

Note how internal System details are not mentioned and …

how it avoids Environment entities not in the Interface.

Specification

A spec describes how the system should react to Environment events that it can sense:

Even Solution #1 has a Spec mentioning only Interface phenomena for that solution:

> If the fee collector, F, receives $1 from a person, P, trying to enter the park, F tells the militia officers not to shoot P; otherwise, F tells the militia officers to shoot P.

Specification

Solution #1 Spec mentions only Interface phenomena:

> If the fee collector, F, receives $1 from a person, P, trying to enter the park, F tells the militia officers not to shoot P; otherwise, F tells the militia officers to shoot P.

The Interface consists of the fee collector, the militia officers, and the persons.

The hidden part of the System, not in the Environment, is the *brains* of the Interface entities.

Specification

Solution #1 Spec mentions only Interface phenomena:

> If the fee collector, F, receives $1 from a person, P, trying to enter the park, F tells the militia officers not to shoot P; otherwise, F tells the militia officers to shoot P.

The Interface consists of the fee collector, the militia officers, and the persons.

Why must the persons be in the Interface?

Does this make sense in the real world?

Specification

Solution #1 Spec mentions only Interface phenomena:

> If the fee collector, F, receives $1 from a person, P, trying to enter the park, F tells the militia officers not to shoot P; otherwise, F tells the militia officers to shoot P.

The Interface consists of the fee collector, the militia officers, and the persons.

Does this make sense in the real world?

The problem is the park militia who can shoot persons who enter the park without paying. They have to shoot only the right people.

Park example: Possible solution

If the solution had been

Solution #1':

- Employ human fee collectors that give out tickets to those who pay the fee.
- Enforce perimeter security by a chain link fence and ticket collectors that allow in only those with tickets.

Park example: Possible solutions

Spec of Solution #1’

- The fee collector issues a ticket for each $1 E receives.
- The ticket collector allows in only any ticket bearer.

Just as the system does not care what pushes a key on a keyboard, the system here does not care who bears a ticket.

Requirements vs. Specification

- *Requirements* are statements of *desired* properties of an SUD, *what* it must do
  - trying to avoid any mention of any solution.
- A *specification* is a description of a *solution* explaining *how* the SUD will satisfy those properties, in terms of the shared phenomena
  - more concrete and detailed, but not as much as a design or implementation.

Requirements vs. Specification

Requirements are statements of *desired* properties of an SUD, *what* it must do

- trying to avoid any mention of any solution.

Reqs mention only Environment entities.

If they mention the Interface entities of some solution, that solution becomes required and other solutions are precluded!

Requirements vs. Specification

A *specification* is a description of a *solution* explaining *how* the SUD will satisfy those properties, in terms of the shared phenomena

- more concrete and detailed, but not as much as a design or implementation.

Requirements vs. Specification

A *specification* is a description of a *solution* explaining *how* the SUD will satisfy those properties, in terms of the shared phenomena

- more concrete and detailed, but not as much as a design or implementation.

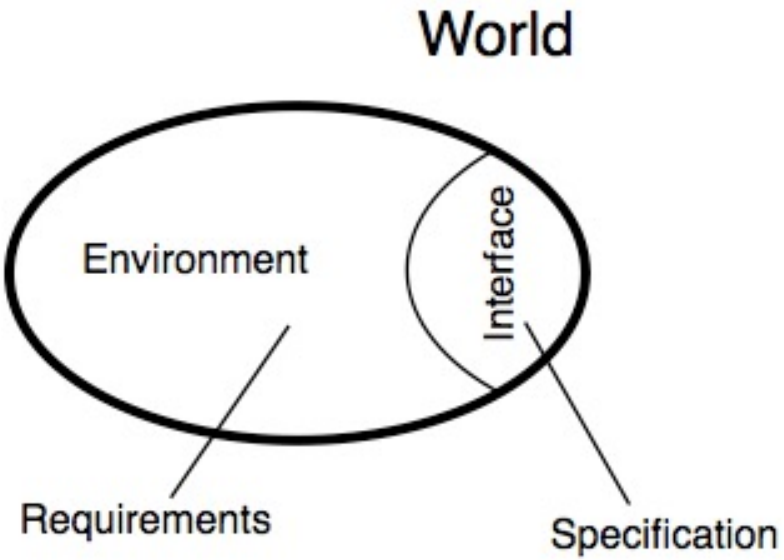
A spec mentions only Interface entities.

If it mentions some other Environment entity, that entity becomes part of the Interface.

R, S, Design, & Code

- R: If a user presses the “K” key, then E sees “K” on the screen.
- S: If the “K” key is pressed, then display “K” on the screen.
- Design: A press of any key causes emission of an ASCII code that is used as an index into a table of bitmaps in a font table, the bit map that is put on screen.
- Code: C realization of the Design.

Scoping the Environment

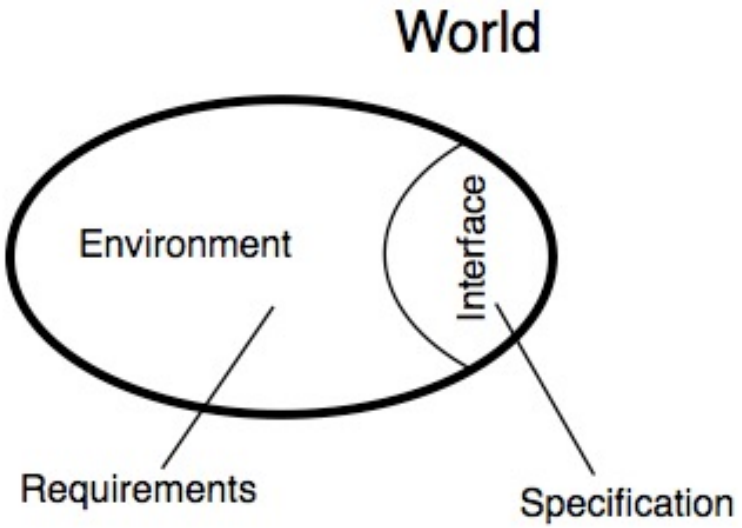


The Environment defines the area of discourse.

The Environment

- It is a subset of the World.
- Want to model only as much of the World as is necessary to express the reqs and the spec – no more and no less.

Scoping the Interface



The Interface constrains the interaction and the spec.

The Interface

- It is a subset of the Environment.
- Want to put into the Interface only as much of the System and the Environment as is necessary to express the spec – no more and no less.

Context diagram

A *Context Diagram* is a graphical model of the Environment plus System and their components.

- Each Environment component that is connected to the System shares phenomena with the System.
- Sometimes need to have Environment components that don't interact directly with the System, to be able to write the Reqs and Spec.

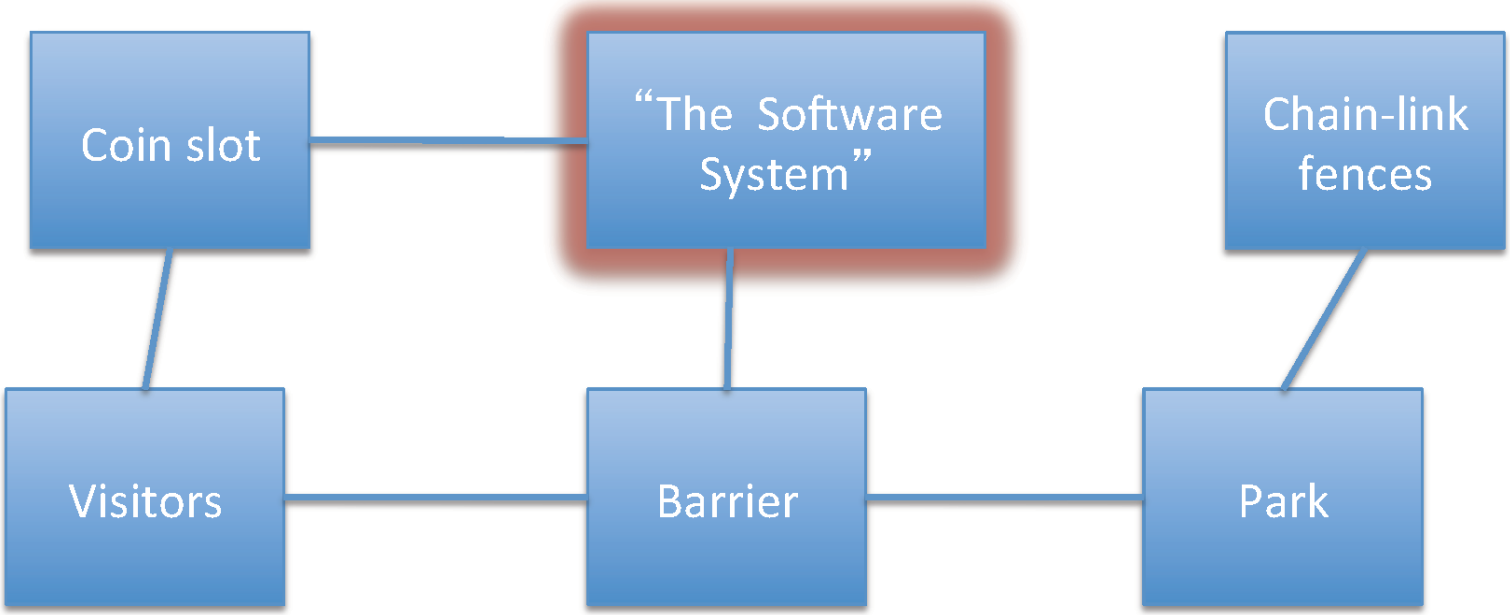
Context diagram

A *Context Diagram* is known in the literature also as

- a Domain Model
- a Class Model
- a Class Diagram

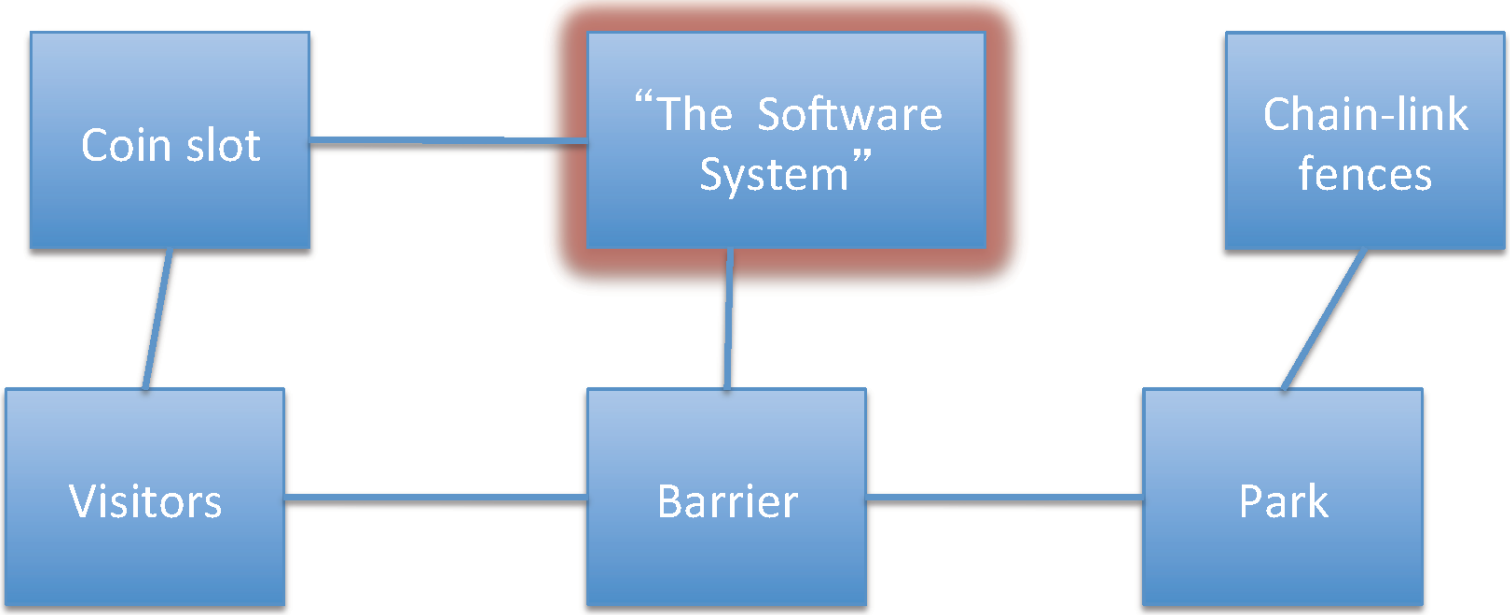
Just so that you are aware

Context diagram



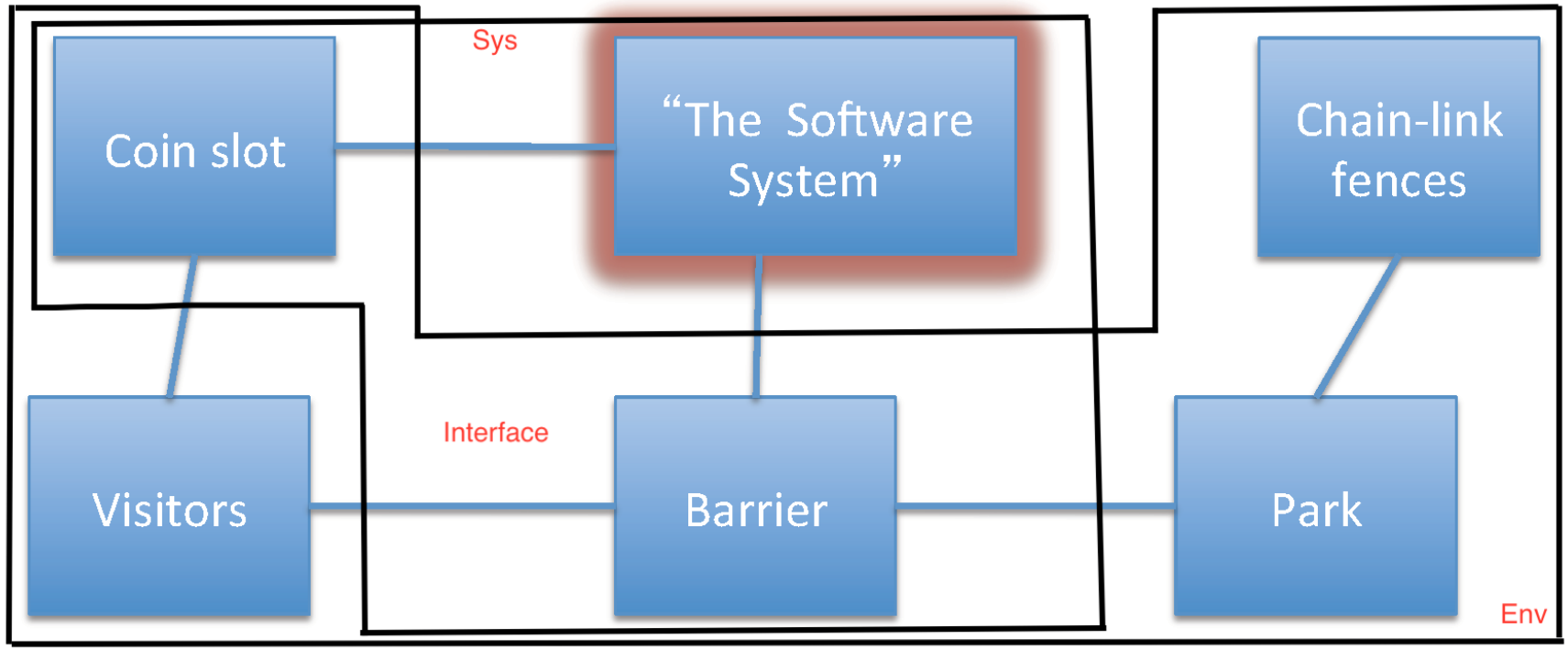
- Each Environment component that is connected to the System shares phenomena with the System.
- Sometimes need to have Environment components that don't interact directly with the System, to be able to write the Reqs and Spec.

Context diagram



- Each Environment component that is connected to the System shares phenomena with the System.
- Sometimes need to have Environment components that don't interact directly with the System, to be able to write the Reqs and Spec.

Context diagram



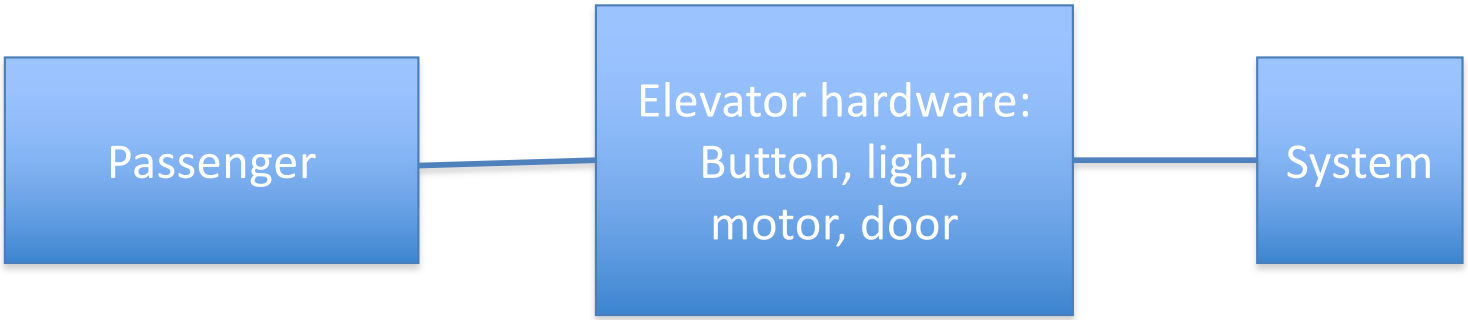
Superimposing World model on context diagram

Example: Elevator

An elevator passenger who wants to travel from one floor to another (higher) floor presses the "up" button at his current floor. The light beside the button must then be lit, if it was not lit before. The elevator must arrive reasonably soon, travelling in an upwards direction. The direction of travel is indicated by an arrow illuminated when the elevator arrives. The doors must open, and stay open long enough for the passenger to enter the elevator. The doors must never be open except with the elevator is stationary at a floor.

Michael Jackson, *Software Requirements and Specifications, Addison-Wesley, 1995.*

Example: Elevator



A Look Back

The parts of the RE Reference Model give you the mental tools to form a domain (context) model for any system you have to build.

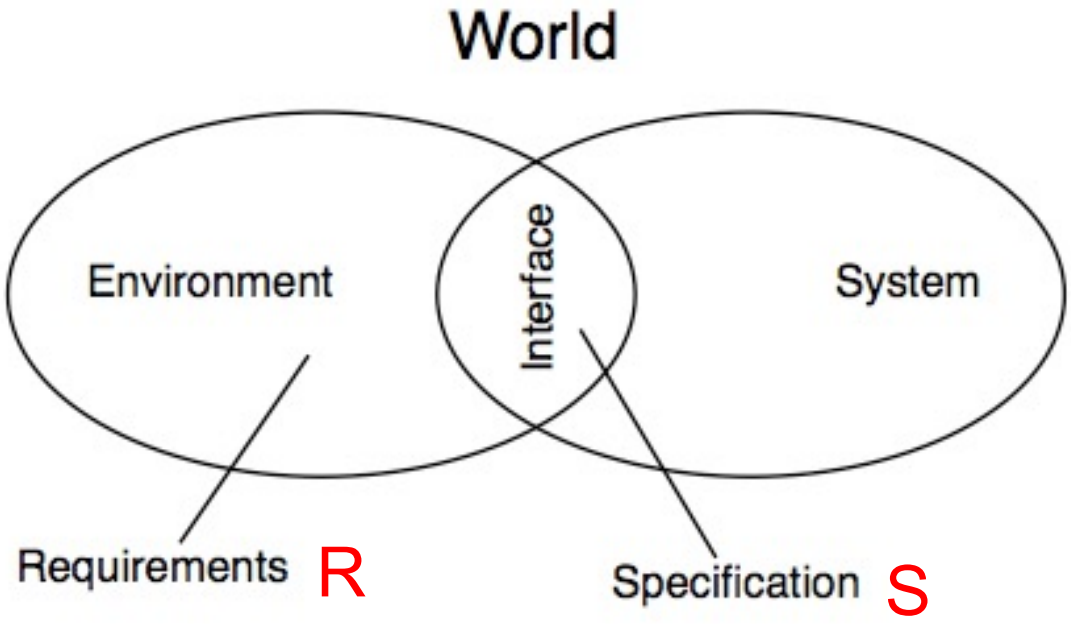
It helps you decide what are the components and where they go in the model:

- in the environment you don't develop
- in the system you develop
- in both, i.e., in the interface

It tells you the questions you must ask, and

it tells you what you have to build and what you don't.

Deriving specifications



Deriving specs (S) is the process of identifying actions, functions, operations, and constraints on shared phenomena, i.e., S, that achieves or entails the reqs (R):

$$S \vdash R$$

Domain knowledge

Reqs are concerned with describing things that we want the System to help make true.

- The System might not be able to accomplish these things by itself.
- Guaranteed properties of the Environment might be necessary for the System to actually and fully meet the Reqs.

These properties are called Domain Knowledge, D.

Domain knowledge

These properties are called Domain Knowledge, D.

Domain Knowledge is thus the set of properties that we know, or assume, to be true of the Environment that are relevant to the problem.

Domain knowledge

Without domain knowledge, you could not ensure that any system you designed would be capable of satisfying the stated requirements!

That is, without D, you could not ensure that $S \vdash R$.

That is, you will need to show that $D, S \vdash R$,

the Zave—Jackson Validation Formula (ZJVF).

Park domain knowledge

- *There is no way to get into or out of the park except through the barrier, i.e.,*
  - *the fence is tall enough that no one can jump over it, and*
  - *the fence is inserted deep enough into the ground that no one can burrow underneath it.*
- *…*

Park domain knowledge

- *…*
- *The power that the barrier and coin slot need to operate is always available.*
- *The coin slot is able to hold as many coins as have been inserted.*
- *….*

Park Spec

- The park turnstile (barrier + coin slot) system spec will be expressed in terms of the shared phenomena:
  - coin insertion, barrier locking and unlocking, barrier pushing, …
- Without the stated domain knowledge, you could not ensure that any turnstile system you designed would be capable of satisfying the stated reqs!

Park example: Requirements

Informal requirement:

Collect $1 fee from each human park user on entry to park.

- Ensure that no one may enter park without paying.
- Ensure that anyone who has paid may enter the park.

Elevator domain knowledge

- *The elevator is constrained to move in a shaft, so that it never goes from one floor to another without passing all the intermediate floors.*
- *If the motor polarity is set to "up" and the motor is activated, then the elevator will rise.*
- *…*

Elevator domain knowledge

- *…*
- *If the elevator arrives at a floor travelling upwards, the floor sensor switch is set on when the elevator is nine centimeters below the home position at the floor.*
- *The lift doors take 2250 msec to reach the fully closed state from the fully open state.*

Elevator Spec

- The elevator spec will be expressed in terms of the shared phenomena:
  - states of the sensor switches, button pressings, setting and activations of the motor and doors, …
- Without the stated domain knowledge, you could not ensure that any elevator system you designed would be capable of satisfying the stated reqs!

R, S, D, Design, & Code

- R: If a user presses the “K” key, then E sees “K” on the screen.
- S: If the “K” key is pressed, then display “K” on the screen.
- D: The user has a way to press keys and a way to see what is on the screen.
- Design: A press of any key causes emission of an ASCII code that is used as an index into a table of bitmaps in a font table, the bit map that is put on screen.
- Code: C realization of the Design.

Traffic light example

- S = spec of traffic light that guarantees that perpendicular directions do not show green at same time
- R = perpendicular traffic does not collide

Can you show that $S \vdash R$?

Traffic light example

- S = spec of traffic light that guarantees that perpendicular directions do not show green at same time
- R = perpendicular traffic does not collide

Can you show that $S \vdash R$?

No! What's missing? Some D!

Traffic light example

- S = spec of traffic light that guarantees that perpendicular directions do not show green at same time
- R = perpendicular traffic does not collide
- D = drivers behave legally and cars function correctly

Now you can show that D, S $\vdash$ R?

Traffic light example

- S = spec of traffic light that guarantees that perpendicular directions do not show green at same time
- R = perpendicular traffic does not collide
- D = drivers behave legally and cars function correctly

Now you can show that D, S $\vdash$ R?

Yes… sort of, in a hand-wavy way! ☺ ☹

Traffic light example

- S = spec of traffic light that guarantees that perpendicular directions do not show green at same time
- R = perpendicular traffic does not collide
- D = drivers behave legally and cars function correctly

Now you can show that D, S $\vdash$ R?

Yes… sort of, in a hand-wavy way! ☺ ☹

It ain't pure and proper math, like is done in MC!

But you (I hope!) and I can see the logic.

We'll see soon that there is a good reason why it will never be pure and proper math.

Traffic light example

- S = spec of traffic light that guarantees that perpendicular directions do not show green at same time
- R = perpendicular traffic does not collide
- D = drivers behave legally and cars function correctly

Now you can show that D, S $\vdash$ R?

Yes… sort of, in a hand-wavy way! ☺ ☹

Or can we?

It all depends on all hardware's operating correctly and power's being always available!

Traffic light example

- S = spec of traffic light that guarantees that perpendicular directions do not show green at same time
- R = perpendicular traffic does not collide
- D = drivers behave legally and cars function correctly

Now you can show that D, S ⊢ R?

Yes… sort of, in a hand-wavy way! ☺ ☹

It all depends on all hardware's operating correctly and power's being always available!

We choose to ignore hardware failures here.

In this part of the world, the probability of power failures is low enough to ignore.

Traffic light example

In parts of the world, in which the probability of power failures is too high to ignore:

In such a place, the Spec might provide that:

> In the event of a power failure, a back-up battery flashes the red lights in all directions.

This makes the traffic light a 4, or whatever, way stop sign.

A look back.

Normally, you will have to examine *each* domain assumption, i.e., conjunct of D, to decide…

> if, given the *context* of your SUD, it is worth for the system to protect against, i.e., detect and react to, the failure of the assumption to be true.

Take into account the probability of, and the cost of the failure.

Traffic light example

- S = spec of traffic light that guarantees that perpendicular directions do not show green at same time
- R = perpendicular traffic does not collide
- D = drivers behave legally and cars function correctly

Problem: make D unnecessary!

Traffic light example

- S = spec of traffic light that guarantees that perpendicular directions do not show green at same time
- R = perpendicular traffic does not collide
- D = drivers behave legally and cars function correctly

Problem: make D unnecessary

Steel walls pop up on red.

Traffic light controls cars by radio!

Traffic light example

- S = spec of traffic light that guarantees that perpendicular directions do not show green at same time
- R = perpendicular traffic does not collide
- D = drivers behave legally and cars function correctly

Problem: make D unnecessary

Steel walls pop up on red.

Traffic light controls cars by radio!

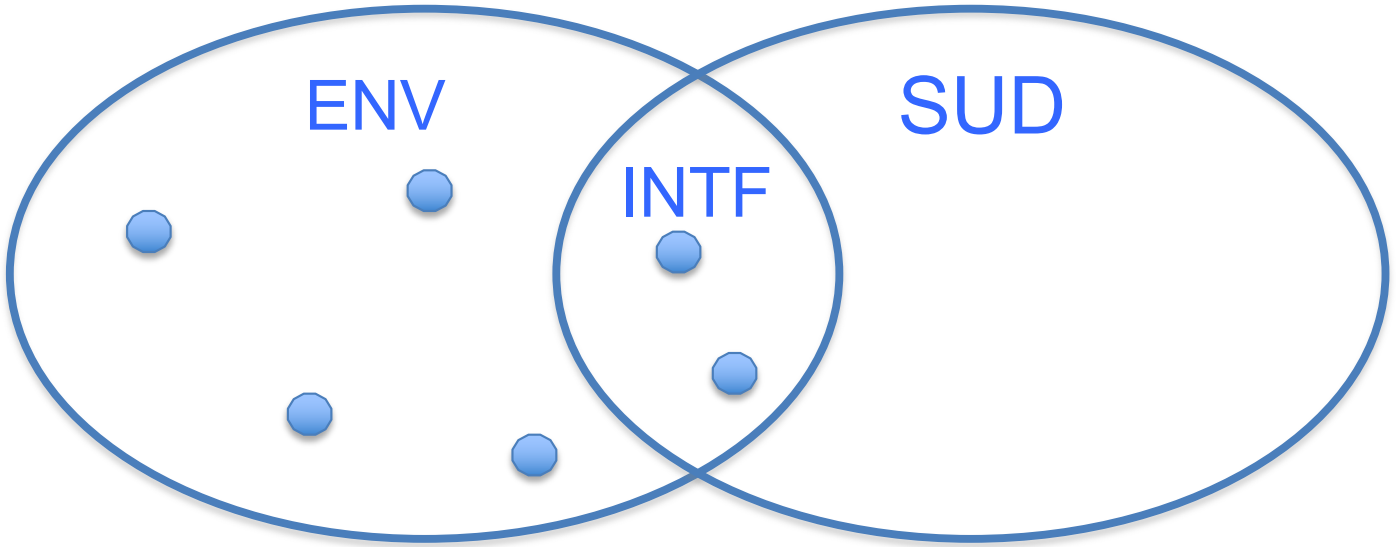
Even so, it still all depends on all hardware's operating correctly and power's being available!

A Look back

This part of the RE reference model, the ZJVF, helps you decide

- the features that you need to meet the reqs of the SUD you have to develop and
- the exceptions these features must check for to deal with the surprises that the real world delivers.

Reference model



R – Reqs live in ENV (incl. INTF)
S – Spec lives in INTF, describes behavior of SUD
D – Domain knowledge lives in ENV (incl. INTF)

Reference model

Thus, if we enlarge our model to include domain knowledge, then the ZJVF must hold:

$$\mathbf{D, S \vdash R}$$

- D is domain knowledge
- S is the spec
- R is the reqs

Reference model

$$\mathbf{D, S \vdash R}$$

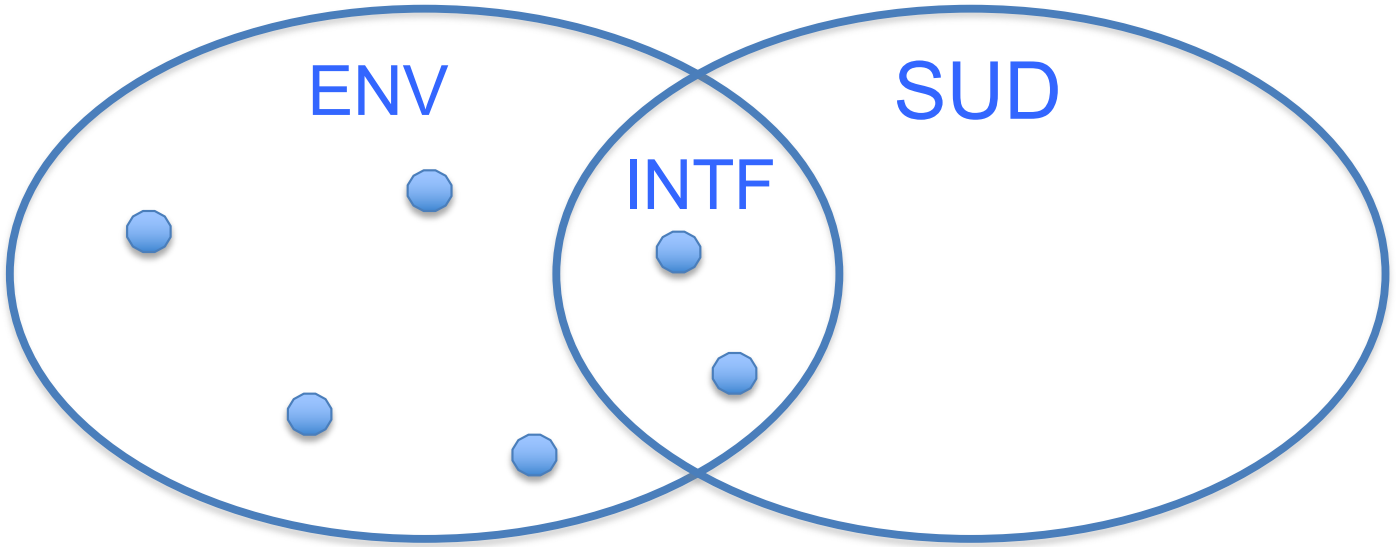
- The spec describes the behavior of a system that is supposed to realize the reqs.
- The domain assumptions are needed to argue that any system that meets the spec, and that manipulates the interface phenomena, will satisfy the original reqs.

Reqs that live in only ENV – INTF?

Is there some notion of requirements that live in only ENV, saying only what is desired in the world, independent of any system that might achieve it?

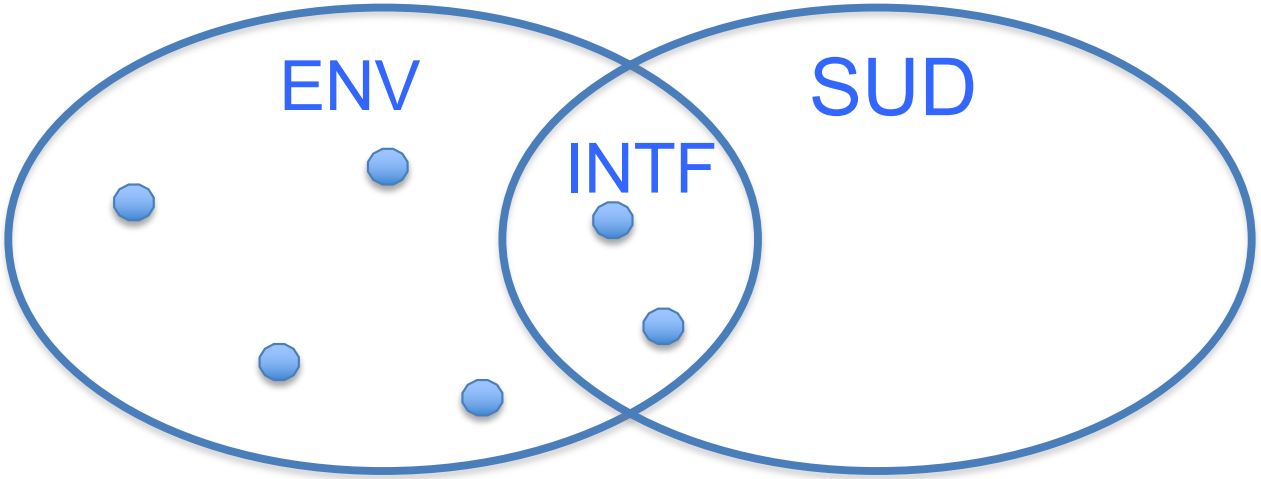
We could call these “high-level requirements” or “goals”!

Add Goal = G, with R ⊢ G



G – High Level Reqs, Goals live in ENV – INTF
R – Requirements live in ENV (incl. INTF)
S – Spec lives in INTF, describes behaviour of SUD
D – Domain knowledge lives in ENV (incl. INTF)

Where is the User Interface?



All this makes it very clear where the user interface of the SUD is and…
Why it is described in the Spec, and it is NOT just another implementation detail!

Reference model

If you can't prove **D, S** ⊢ **R**, then at least one of 3 things must be wrong:

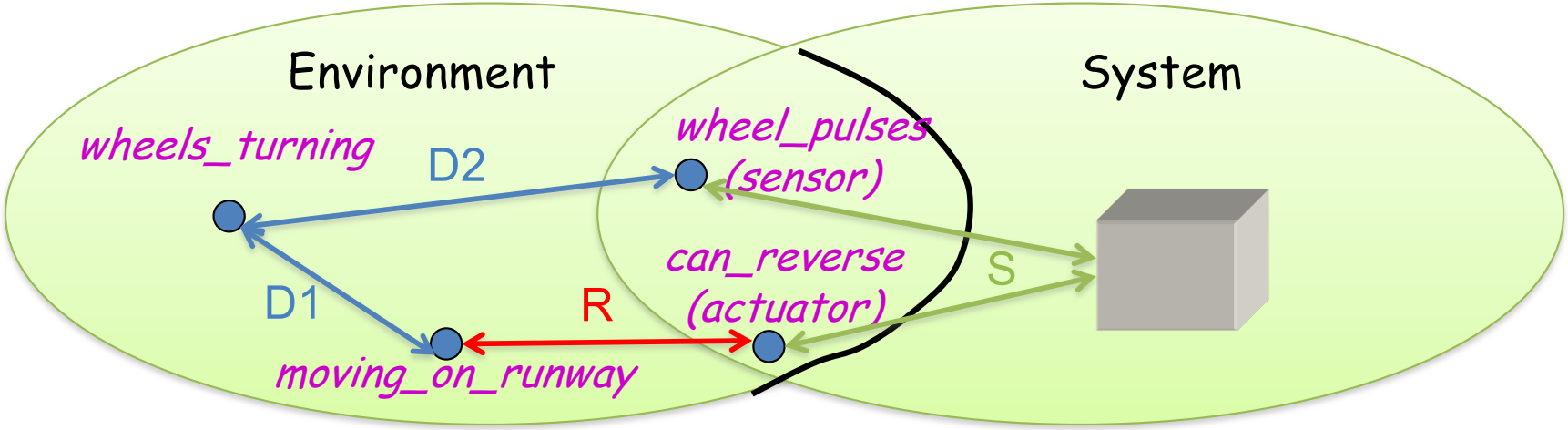
- reqs are incorrect or unreasonable
- system doesn't do enough
- we aren't assuming enough about the environment

Reference model

A real world example: [M. Jackson]

- An airplane overshot the runway on landing.
  The pilot had tried to engage reverse thrust, but the airplane's avionics system wouldn't permit it.
- What's wrong?

A real world example



R: An airplane may engage reverse thrust iff it's moving on the runway
D1: Moving on runway iff wheels turning
D2:Wheel pulses detected iff wheels turning
S: Can reverse iff wheel pulses detected

A real world example

The reason for the crash that the runway was wet, and the wheels were hydroplaning instead of turning.

- Reverse thrust could be engaged only if pulses from the wheel sensors indicated that the wheels were turning.

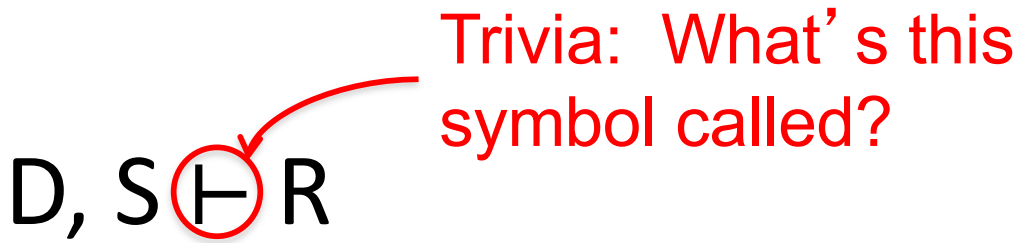
A real world example

The developers made domain assumptions, but D1 was wrong.

- If airplane is hydroplaning, then MOVING_ON_RUNWAY is true (and would like to engage reverse thrust), but WHEELS_TURNING is false.
- The error was in the step of deriving spec from reqs.

Correctness

To validate:



Must be able to argue that the spec plus the domain assumptions are enough to satisfy the reqs.

Correctness

$$D, S \vdash R$$

If you can't make this argument successfully, then you need to do one (or more) of:

1.

2.

3.

Correctness

1. strengthen the specification
2. strengthen the domain knowledge
3. weaken the requirements

Example: Train crossing

Req: train is in crossing ⇒ gate must be down

S1: if approaching train is 200m away, lower gate

Example: Train crossing

Is S1 enough?

No, then give D1, then D2.

- D1: gate can be lowered in 10 sec
- D2: trains move more slowly that 200m/10s

Example: Train crossing

- D1: gate can be lowered in 10 sec
- D2: trains move more slowly that 200m/10s

Yes, this is enough now ... but ...

- Is this enough to be *safe?* Are D1 and D2 reasonable?
- What about speed of cars and humans who might be crossing tracks? Do they have enough time to clear? Will the crossing coming down interfere with their leaving?

D, S ⊢ R

The ZJVF encapsulates all the thinking the requirements engineers need to be doing, with input from the clients and users of the future SUD to make sure that it will work in the real world, that never behaves as we assume it to behave.

It gets the requirements engineers to ask the right questions of the clients and users.

Uncertainty in D, S ⊢ R

- The formula D, S ⊢ R tries to be formal in the sense of describing what happens completely.
- One would expect computers and software and their combination to be formal in this sense.
- But, the real world intervenes to make this formula only a guideline and not an accurate, precise model.

Uncertainty in D, S ⊢ R

- First, the real world *never* behaves as *any* model.
- Any model D is only an approximation.
- Generally, the simpler the model, the more of an approximation the model is, but the easier it is to prove things about the model.
  - There is a distinction between proving things about a model and proving that a model is correct!

Uncertainty in D, S ⊢ R

There is a distinction between

proving things about a model

[Math proof]

and

proving that a model is correct

[Empirical proof].

- …

Uncertainty in D, S ⊢ R

- …
- Modeling the real world accurately requires complexity to deal with all the weird exceptions.
- A mechanistic description generally has to be replaced by or tempered with a probabilistic model, e.g., 99.99% of drivers stop at a red light.

Uncertainty in D, S ⊢ R

- At the lowest level, a CBS is mechanistic, e.g., a traffic light, the sqrt function, and can be modeled with a consistent S that is mechanistic, that always gives for any input the same answer that the CBS does.
- But floating point arithmetic is not the same as real numbers, and integer arithmetic suffers over- & underflow.
- …

Uncertainty in D, S ⊢ R

- …
- At higher levels, e.g., MS Word, an operating system, process control, etc., the CBS is so large that we cannot understand all of its code and all of its behavior. So, we begin to give probabilistic models of what the CBS does.

Uncertainty in D, S ⊢ R

- All that applies to D, applies to R, because both are models of the real world, one as is, and the other as it is to be.
- R is always an approximation of what we want, because if we overlook something in the real world and it turns out to be relevant to the CBS's behavior, e.g., a gaggle of Canadian geese that fly near a jet engine, then R may not be correct.

Uncertainty in D, S ⊢ R

- The formula D, S ⊢ R tries to be formal in the sense of describing what happens completely.
- But, as we have seen, it cannot be completely formal because at least D and R have to describe the real world, which is not formal

What does this do to the hope of formally modeling computer systems?

More uncertainty in D, S ⊢ R

We are developing more and more systems with stochastic behavior:

- Molecular SW, e.g., DNA, RNA, Proteins, Catalysts
  - Molecules designed specifically to achieve a desired effect
  - Molecule is shown empirically to behave as specified in S, with 99.95% certainty
- …

More uncertainty in D, S ⊢ R

We are developing more and more systems with stochastic behavior:

- …
- AIs
- Learned Machines (LMs) = ML + teaching data

In these case, in D, S ⊢ R, also S is informal!

CS445 / SE463 / ECE 451 / CS645

Software requirements specification & analysis

A reference model for requirements engineering

Spring/Summer 2023

Mike Godfrey & Daniel Berry & Richard Trefler