Università degli Studi dell'Insubria
Dipartimento di Informatica e Comunicazione

Modelling requirements with UML: a rigorous approach
(Doing requirements well with UML)

Prof. Luigi Lavazza and Dr. Vieri del Bianco PhD

Contents

- The problem of requirements modelling
- Limits of the use cases
- Problem frames
- Problem frames with UML
- Dealing with time: extending OCL
- Possible evolutions of the proposed method
- Conclusions

Contents

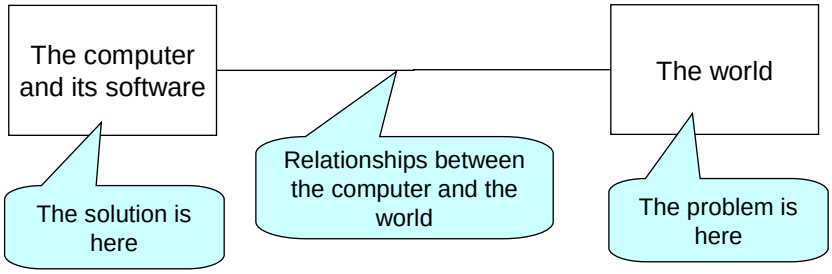
- The problem of requirements modelling
- Limits of the use cases
- Problem frames
- Problem frames with UML
- Dealing with time: extending OCL
- Possible evolutions of the proposed method
- Conclusions

The problem of requirements modelling

- This section addresses the problem of establishing the precise meaning of requirements modelling.
  - The concepts of Environment and Problem Domain are introduced.
    - The problem domain is the portion of the Environment that is visible by the machine.
  - The relationship between the Environment and the machine (which is seen as a black box in this phase) is described in terms of “phenomena” that are shared between the Problem Domain and the Machine.
  - The requirements are defined as properties expressed over the Environment and the Machine, while specifications are defined as properties expressed over the interface only.
  - An example application is used to illustrate the concepts.

Focusing on the problem

- In order to develop a software solution it is necessary to study and understand the problem
  - ⇨ understand where the problem is and where the solution is.



Problem solving

- How can we define and describe a problem in order to support the development of a software-based solution?
  - Developers have to be given all the relevant information concerning the desired behaviour of the system
  - This information constitutes the requirements of the software to be developed
  - Requirement = criterion, condition or constraint that a software artifact has to satisfy in order to be acceptable

Requirements engineering: what is it?

- Some misbelieve that requirements simply represent what the users asks.
- Actually, the user knows the problem well, but often he/she hasn't the slightest cue of how to solve it.
  - Or, even worse, has completely wrong ideas of how to solve the problem…
- Requirements engineering is a creative activity, that leads to defining the effects that the HW/SW machine will have on the application domain.
  - The behavior of the system that combines the domain and the (HW/SW) machine is expected to solve (or at least to reduce) the problems that induced the user to ask for a computer-based solution.

Role of the analyst

- The analyst has a double role in describing requirements:
  - He/she must understand the needs of the user
  - He/she must cooperate (pro)actively with the user in defining requirements that are
    - Effective (i.e., the machine actually contributes to solve the problem)
    - Technically feasible
    - Reasonable with respect to economic constraints

Requirements specification

- The goals of the requirements specifications are:
  - To describe what are the responsibilities of the system in satisfying the user's needs
  - To define what are the responsibilities of the environment in supporting the HW/SW system
    - For instance, the environment has to provide input data
- In this phase the analyst is active and propositive
- In fact, often the responsibilities of the system and the interaction between the machine and environment are not always determined univocally by the user needs.

Example

- The user needs that information provided by subsidiary agencies via email are permanently stored at the central agency, in order to be available for subsequent elaboration, when required.
- We can specify different machines that solve this problem:
  a) Email messages are read by a (human) operator that feeds manually the information to the system via suitable forms.
  b) Email messages are read directly by the system, which extracts the relevant information and stores it in the system. Of course in this case the messages have to be suitably formatted (this is environment responsibility). The system checks for format errors and an operator fixes errors "manually" (as in point a).
  c) Email messages –suitably formatted– are read by the system, which extracts information, etc. In case of error, the system sends automatically to the sender an email message containing a diagnostic (which is also an invitation to resend a correct message).

Solutions pros and cons

- Solution a)
  - It minimizes the software development cost, since it does not involve the elaboration of the email messages.
  - The operating cost could grow excessively high if the number of messages is big. Several people could be necessary just to read the email and perform data entry.

  ⇨The cost of the solution is not equal to the cost of the software!
- Solution b)
  - It requires less operators, since all the correct messages are managed without manual processing;
  - It requires that the system is equipped with both automatic and manual input management; thus, it is more expensive to develop than solution a.
  - It is viable only if it is possible to assure that the subsidiary agencies actually adopt the required email format.
    - If emails are generated automatically by the agencies, updating their systems would be required …

Solutions pros and cons

- Solution c)
  - No manual input management. Automatic error management, which was not present in solutions a and b.
  - Messages not correctly formatted can cause delays. For instance if the subsidiary is several hours away, or if the diagnostics are not clearly understood, etc.

⇒ The analyst and the user have to evaluate together the different possibilities, in order to identify the best solution.

Specifying requirements is a *design* activity

- As mush as the development of the machine …

  ... but in a different context!
- The development of the machine (e.g., the “design” phase) concerns the internals of the machine
- The definition of the requirements concerns the responsibility of the system and its interaction with the environment, so that the sum of the system, the environment and the interaction results in a situation that suits the user.

Notice

- In several books you find written that
  - Requirements specifications indicate ***what*** has to be done
  - Design documents specify ***how*** to do it.
- This is true only for the HW/SW machine
- On the contrary, considering the machine+environment, also the requirements specifications contain elements of the solution:
  - How the environment has to behave
  - What the machine does
  - How the machine and the environment interact

A reference model

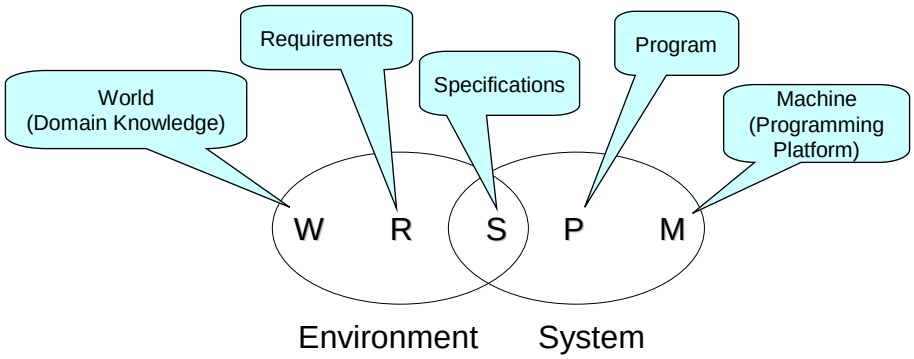
- The concepts informallyexpressed so far are described in a precise way in:

  Gunter C., Gunter E., Jackson M., Zave P. A reference model for requirements and specifications. IEEE Software 3(17), 2000

Context diagrams

- Context diagrams contain elements from the environment as well as to the software.
- There are several types of elements:
  - **Domain Knowledge** provides presumed facts about the environment,
  - **Requirements** indicate what the customers need from the system, described in terms of its effect in the environment,
  - **Specifications** provide enough information for a programmer to build a system to satisfy the requirements,
  - **Program** implements the specification on the programming platform,
  - **Programming Platform** provides the basis for programming a system to satisfy the requirements and specifications

5 element types



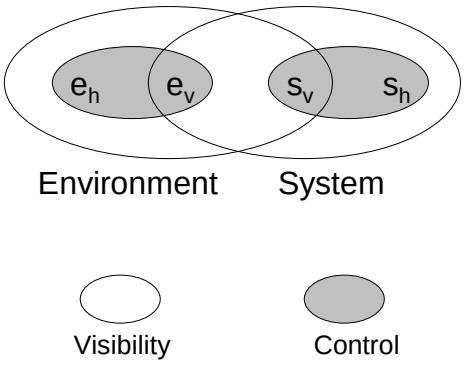
Elements

- Application domain
  - Description of the facts and relationships between facts coherent with the real world
- The machine: facts and properties of the SW
  - Represents aspects of the HW/SW solution
- Phenomena shared between domains
  - In the requirements phase we are interested in the phenomena shared between the machine and the environment
  - The part of the machine not shared with the environment is studied in the design and implementation phases
- (User) requirements
  - Express properties that the solution must own
- Specifications
  - Indicate the impact of requirements on the machine

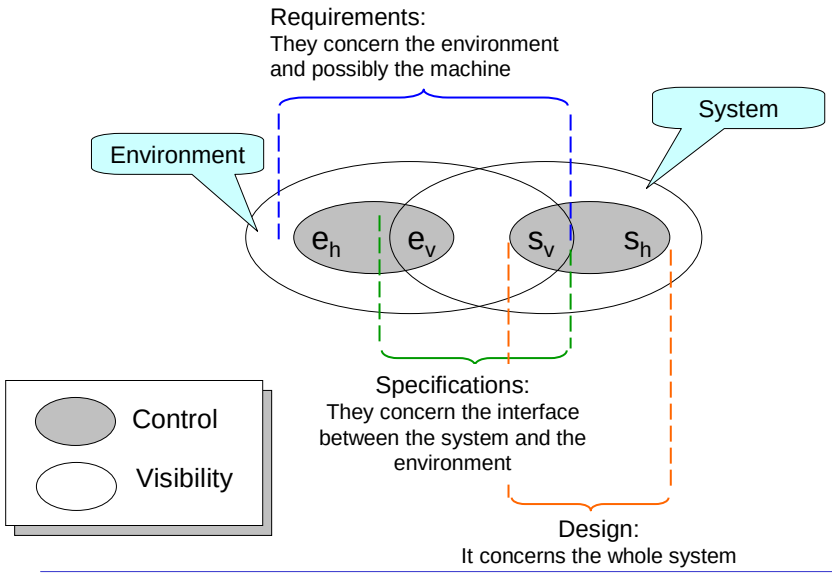
Phenomena: visibility and control

- Environment and system can be decomposed into phenomena
  - *e* belonging to the environment
  - *s* belonging to the system
- Then, phenomena can be classified according to visibility:
  - phenomena e_v are visible by the system, while the other phenomena e_h of the environment are hidden from the system.
  - phenomena s_v are visible by the environment, while the other phenomena s_h of the system are hidden from the environment.

Phenomena: visibility and control



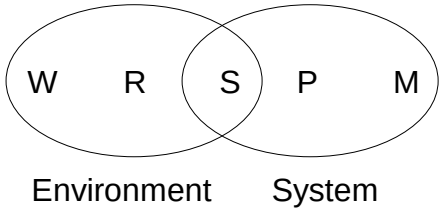
Requirements, specification, design



Descriptions

- Indicative descriptions: they say what is true (and we cannot modify)
  - Typically W and M
- Optative descriptions: they describe what the user wants to be (become) true
  - Typically R
- In the requirements specification M and P do not appear
  - P is the program to be built
  - M is the hardware, usually given, or chosen according to different considerations (cost, availability, maintenance, …)

Objective of Software development



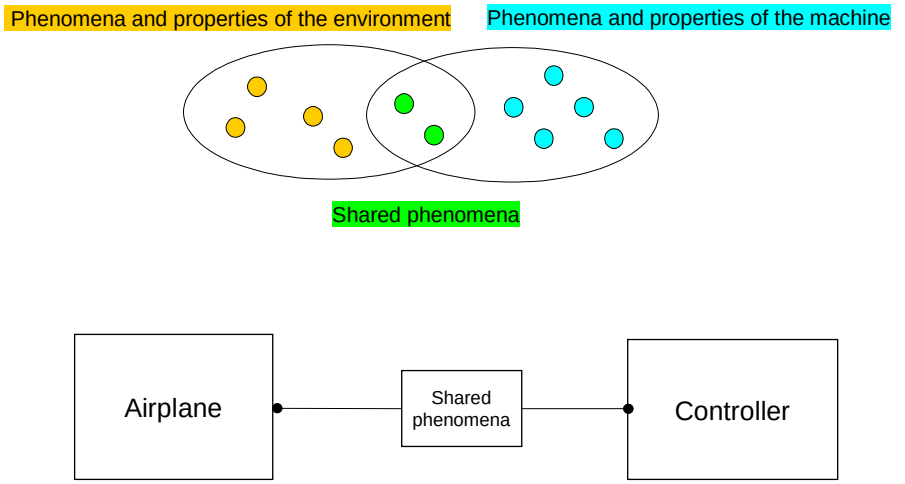
$\forall\ e\ s.\ W \wedge M \wedge P \Rightarrow R$

To configure (through P) the machine M so as to produce the effects R in the application domain W

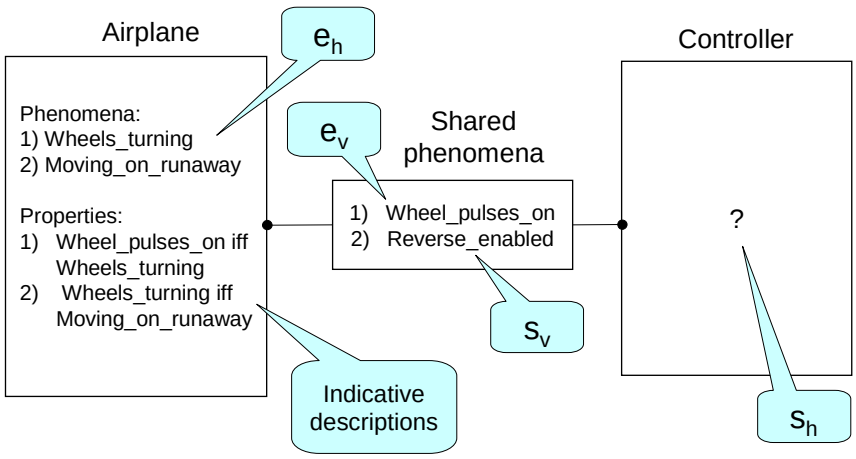
Example – part of an airplane controller

- Application domain
  - The set of phenomena that concern avionics, especially those related to the specific problem
- Shared phenomena
  - Phenomena of the airplane that are made visible to the machine (typically via sensors)
  - Phenomena of the machine that are made visible to the airplane (typically via actuators)
- Requirements (properties that the solution must own)
  - Example: the controller must reverse the motors' push when the plane has landed
- Specifications
  - Translate the requirements in terms of shared phenomena
  - Of course, we cannot build a machine that deals with something that is not visible…

Domains and relations among domains



Detailed context diagram



Example: requirements and specifications

- Requirement: engine push has to be reversed if and only if the plane is moving on runaway
  - REVERSE_ENABLED if and only if MOVING_ON_RUNWAY
- This is a rather straightforward description of the problem to be solved.
- From this requirement and the knowledge of the application domain we can derive the specifications
  - REVERSE_ENABLED if and only if WHEEL_PULSES_ON
- This is the specification of the machine interface, since it is expressed only in terms of shared phenomena

Observation

- The specification is wrong, since it does not consider the possibility of aquaplaning: in such conditions the plane is on the runaway, but the wheels do not turn, and therefore the engine reversing is not enabled.
- The defect derives from an ill (incomplete) understanding of the application domain and consequently from an incorrect description
- According to the specifications we build P so that

  $\forall\ e\ s.\ W \wedge M \wedge P \Rightarrow R$
- But if W does not describe correctly the environment (the correct description being W'), it may happen that

  $\forall\ e\ s.\ W' \wedge M \wedge P \Rightarrow R' \neq R$

Università degli Studi dell'Insubria
Dipartimento di Informatica e Comunicazione

Case study: the Intensive Care Unit

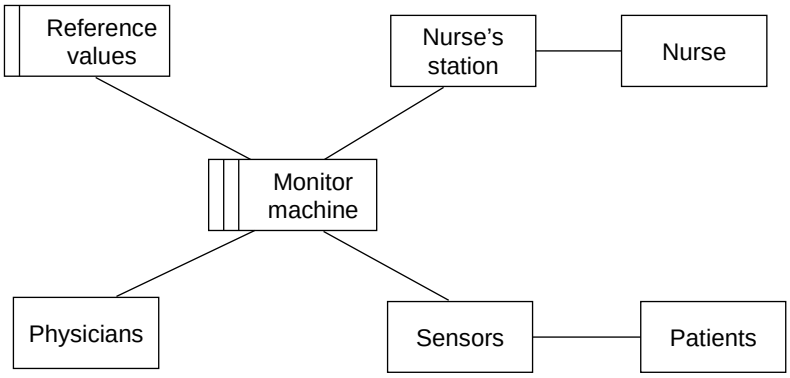
ICU requirements

- A warning system notifies a nurse that a patient's vital signs indicate a critical situation.
  - a computer-based system equipped with sensors that capture the patient's vital signs is connected with an actuator capable of sounding a buzzer.
  - The system can be programmed to sound the buzzer on the basis of data received from the sensors.
- The system has also to record the data received from sensors, so that a physician can later examine the evolution of the patient's situation over a period of time.

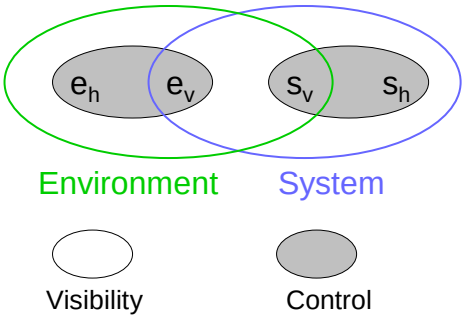
ICU requirements

- There is also some knowledge of the world:
  - There is always a nurse close enough to the nurse's station to hear a buzzer sounded there;
  - It is known how the vital signs from the patient must be interpreted in order to determine when the intervention of the nurse is needed.

Case study: the Intensive Care Unit



Visibility and control for designated terms



e_h : the nurse, the physician and the patient.

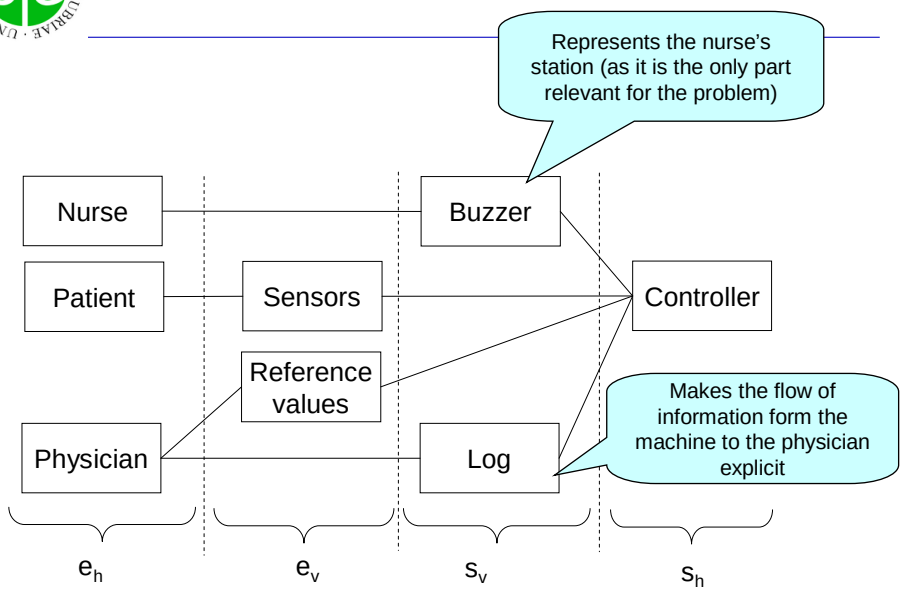
e_v : patient's vital signs detected by sensors, the set of reference values for vital signs.

s_v : the buzzer at the nurse's station, the recording of data from sensors.

s_h : internal representation of data from the sensors.

- Requirements involve e_h e_v s_v
- Specifications involve e_v and s_v, i.e., the interface between the system and the environment

Domain classification



Contents

- The problem of requirements modelling
- Limits of the use cases
- Problem frames
- Problem frames with UML
- Dealing with time: extending OCL
- Possible evolutions of the proposed method
- Conclusions

Use Cases: overview

Use Case: well known technique for capturing functional requirements

- Actor
  - specifies a role played by a person or thing when interacting with the system
  - representation of a person or system/device which exists outside the system under study
  - performs a sequence of activities in a dialogue with the system
- Use Case
  - defines the interactions between external actors and the system under consideration to accomplish a goal
  - single interaction between a primary actor (who initiates the interaction) and other (secondary) actors, and the system itself
  - interaction modeled as a sequence of simple steps

Use Cases: overview

- Use case focuses on describing how to achieve a goal or task
  - multiple use cases are needed to embrace the scope of a new system
  - degree of formality required and maturity (stage, lifecycle) influence the level of detail in each use case
- Use cases should not be confused with the features of the system
- Use cases treat the system as a black box, and the interactions with the system are perceived as from outside the system
- A use case should:
  - describe how the system shall be used by an actor to achieve a particular goal
  - have no implementation-specific language
  - be at the *appropriate* level of detail
  - not include details regarding user interfaces and screens

Use Cases: scope levels

- Use cases may be described at the abstract level (business use case, sometimes called essential use case), or at the system level (system use case)
  - Business use case
    - described in technology free terminology which treats the business process as a black box and describes the business process that is used by its business actors (people or systems) to achieve their goals
    - describe a process that provides value to the business actor, and it describes what the process does
  - System use cases
    - described at the sub process level and specify the data input and the expected data response
    - describe how the actor and the system interact
- Note: Alistair Cockburn in Writing Effective Use Cases introduces more scope levels

Use Cases: details level

- Details level
  - Brief
    - few sentences summarizing the use case
  - Casual
    - few paragraphs of text, elaborating the use case in the form of a summary or story, sketches the steps
  - Fully Dressed
    - formal document with fields for various sections
    - steps detailed in full details
    - preconditions, postconditions, alternative stories, … may be added
- Note: Alistair Cockburn in Writing Effective Use Cases introduces more details levels, mainly describing the Fully Dressed level in more details and levels

Use Cases in UML

- UML defines a graphical notation for diagramming a set of use cases: Use Case Diagrams
  - provide an efficient way to model and describe Use Cases relations
  - provide no format or template for describing single Use Cases
  - a single Use Case can be detailed using UML Sequence Diagrams
    - if and how to describe detail level, scope, preconditions, etc. is left to be decided to the methodology used

Use Case limits: general limits

- ***Non-interaction based requirements***. Use Cases are not well suited to capturing non-interaction based requirements of a system: algorithms, mathematical conditions, etc.
- ***Formal requirements***. Use Cases are not capable to capture formal requirements expressed with a chosen formal language
- ***Non-functional requirements***. Use Cases are not capable to capture non-functional requirements: platform to be used, performance, safety-critical aspects, etc.
- ***User Interface requirements***. UI requirements should not be captured by Use Cases; but it is often difficult to separate a Use Case from the associated UI
- ***Undefined***. No fully standard definitions of use cases; Use Cases relations in UML use case diagrams are not intuitive, whether not ambiguous
- ***Learning curve***. Learning curve involved in interpreting Use Cases correctly, for both end users and programmers

Use Case limits: general limits

- ***Non-interaction based requirements***.
  - Every requirement that is not directly involved in an interaction with the system cannot be easily captured by a Use Case
  - Use cases work well when modelling discrete services used in clearly delimited episodes. Unfortunately, in many real cases, problems do not consist of use cases, for instance when it is required that the machine continuously interact with the problem domain.
- ***Formal requirements***.
  - Use Cases are not capable to capture formal requirements expressed with a chosen formal language
  - Formal languages are rarely combined with Use Case; formal languages usually describe features, whether Use Cases address interaction stories and scenarios

Use Case limits: general limits

- ***Non-functional requirements***.
  - Non functional requirements cannot be captured by Use Cases since they do not imply any interaction at all
  - Non functional requirements have to be captured outside Use Cases possibly using an appropriate methodology
- ***User Interface requirements***.
  - UI requirements cannot be captured by Use Cases since they mainly address graphical details
  - UI requirements have to be captured outside Use Cases possibly using an appropriate methodology
  - UI requirements are often tightly connected with and dependent on the stories/scenarios/Use Cases that use such a UI. This makes the separation of aspects (Use Case, UI details) sometimes difficult to obtain and counterintuitive

Use Case limits: general limits

- ***Undefined***.
  - Use cases, in very different formats, are used in many methodologies and software processes: Unified Process, Rational Unified Process, Catalysis, Crystal Methodology, Extreme Programming…
  - Every methodology and software process capture the general concept of a Use Case, but then it is used with different levels of details, with different terminology, etc.
  - Development Teams usually further customize Use Cases semantics and terminology
- ***Learning curve***.
  - Even if Use Cases are highly intuitive, a learning curve is necessary to fully grasp the whole meaning of a Use Case
  - Learning curve problem is very accentuated by the lack of a Standardized definition of Use Cases.

Use Case limits: specific limits

- There are Use Case limits that are generally not perceived as such
  - Levels undefined
  - Interactions complexity undefined
- Relevant when dealing with Software Development Cost and Effort Estimation

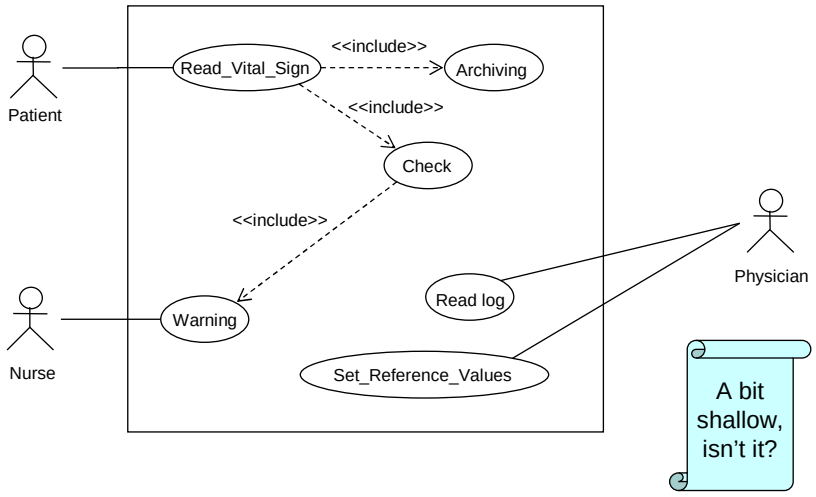
Use Case limits: specific limits

- ***Levels undefined***
  - In Use Cases we can have various types of Use Cases, the scope and the details levels can vary greatly. This leads to a form of ambiguity at the granularity level
  - In other words, a user story could be captured with just a few sentences, or with more Use Cases fully dressed: there is no standard way to precisely understand at what level of granularity a Use Case is described

Use Case limits: specific limits

- ***Interactions complexity undefined***
  - Every single interaction is described by means of one or more textual sentences, and possibly of a precondition to be verified; nothing is said about the complexity of the interaction.
    - Example: if the system has to write a large amount of highly structured data to an external storage (an Actor) the interaction can be described in a very simple way, but it actually hides the real complexities
  - This is not a problem when just capturing requisites, but it is a problem if there is the need to understand the effort required to implement a Use Case

The use case diagram for the ICU



Complementing use case diagrams

- As seen, use case diagrams give only a very schematic and superficial description of the functionality of the system.
- They are usually complemented by
  - Text, or
  - Sequence diagrams
- Problem: in order to write meaningful sequence diagrams you need to have performed domain analysis, to have identified shared phenomena, etc.

  ⇒ Sequence diagrams have to *follow* a great deal of requirements analysis.

This bring us back to the question:
How should we do requirements analysis?

Contents

Problem frames

- Problem frames are briefly introduced.
- An example for each type of frame is given.
- The frame concern is illustrated.
  - Why ‘system models’ trying to combine the specification, domain properties and requirements are unsuitable to support complex developments, and can actually lead to bad results.

Introduction

- Problem Frames classify software development problems.
- They are a sort of patterns, but concerning the problem, not the solution
  - As opposed to well known “design pattern”
- The Problem Frames structure the world in which the problem exists – the problem domain– and describe what is there and what effect a system put there has to provoke.
- By drawing the attention on the problems, rather than on solutions, problem frames lead to exploit the identification and characterization of a class of problems, allowing the “problem owner” who knows the domain, to lead the requirements engineering process.
- The graphic notation is useful to make easier the communication between the problem owner (the analyst) and the developer.

Problem and solution

- Even though the problem frames were conceived to model requirements, it is possible to use them in the transition towards SW development
- Given a problem frame, it is possible to recognize the structure of the solution (e.g., a design or architectural pattern) that is suitable for the problem
- The understanding of the relations between problem and solution can be improved
- The PF can guide/inspire development

Problem context

- The problem context describes the application domain and the general context of the problem to be solved.
  - Analysis of the application domain.
  - Identification of the problem to be solved.
  - Specification of the features of the machine to be developed.
  - The requirements specification document
- Critical issue: what is the boundary between the application domain and the machine?
  - In other terms: what is part of the problem and what is part of the solution?
  - There is no general answer!

Example

- Development of an information system.
  - Domain: the organization that benefits from the system
  - Solution (machine): the information system to be developed.
- Development of a measurement tool that collects statistical data about the usage of an information system
  - Domain: the organization and its information system.
  - Machine: the measurement software to be developed.

Context diagram

- In order to analyse correctly the problem it is important to localise the problem and to understand what it involves, as precisely as possible.
- For this purpose, it is possible to use context diagrams.
- A context diagram contains:
  - The domains to be analysed
    - Machine domain
    - Problem domains
  - The interfaces between domains
    - Shared phenomena

Domain

- The portion of the real world in which the software to be developed will be used
- A domain is characterized by:
  - Entity: car, person, ...
  - Attributes: colour, price, ...
  - Relations among entities: a car is owned by a person.
  - Events: a car is sold.
  - Causal laws: the alarm can be deactivated only if the user has given his/her PIN correctly

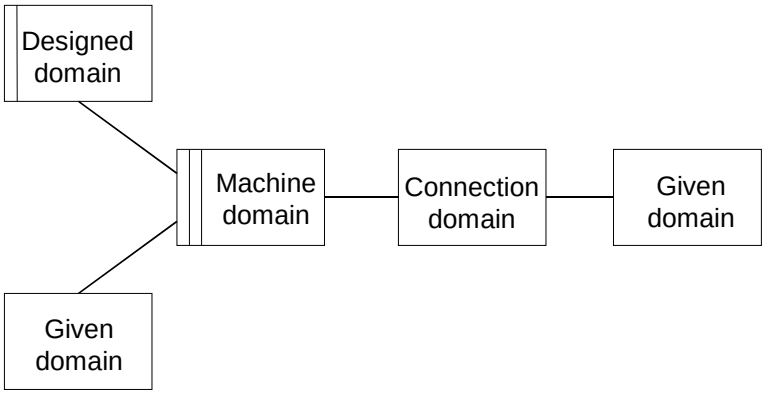
Domains to be analysed

- All the domains defined in the context diagram refer to the physical world.
  - Machine domain: it represents the computer and the software to be developed. Every context diagram contains a unique machine domain.
  - Problem domains: they represent all the parts of the world outside the machine that are relevant with respect to the problem
    - Designed domain
    - Given domain
    - Connection domain

Interfaces among domains

- Phenomena shared among two or more domains, such as events, states, values.
  - Examples:
    - Keystrokes sensed by the software.
    - Signal sent by the airplane wheel to the software controller.
    - ...
- The interfaces among domains allow the sharing of phenomena

Context diagram



Case study: the Intensive Care Unit

- We consider the patient monitoring system of an Intensive Care Unit (ICU), a well-known reference problem.
- The requirement is that a warning system notifies a nurse that a patient's vital signs indicate a critical situation. To do this, there is a computer-based system equipped with sensors that are able to capture the patient's vital signs and an actuator capable of sounding a buzzer. The system can be programmed to sound the buzzer on the basis of data received from the sensors.
- The system has also to record the data received from sensors, so that a physician can later see the evolution of the patient's situation over a period of time.

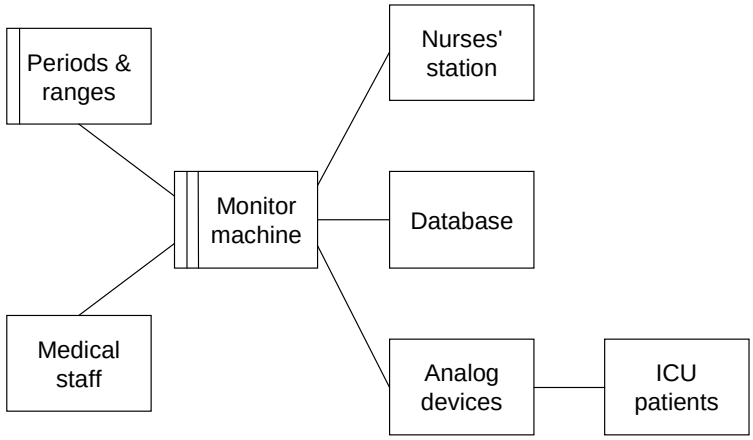
Case study: the Intensive Care Unit

- Some knowledge of the world is also incorporated in the requirements: there is always a nurse close enough to the nurse's station to hear a buzzer sounded there, and it is known how the vital signs from the patient must be interpreted in order to determine when the intervention of the nurse is needed.
- Every patient is monitored by means of an analogic device that measures vital signs. The program has to read these values in on a periodic base (the period is specified for each patient) and record these values in a database
- The medical staff defines the safe ranges for each vital factor for each patient. If a value falls out of the specified range, or if the system detects an error in the measurement devices, the nurses' station is notified by activating the buzzer.

Case study: the Intensive Care Unit

- How to correctly identify all the involved domains?
  - *Every patient is monitored by means of an analogic device that measures vital signs*
    - Involved domains: ICU patients, analog devices
  - *The medical staff defines the safe ranges for each vital factor for each patient*.
    - Involved domains : medical staff, periods & ranges
  - *The program has to read these values in on a periodic base (the period is specified for each patient)*
    - Involved domains : periods & ranges, analog devices
  - *The program has to record values in a database*
    - Involved domain: database
  - *If a value falls out of the specified range, or if the system detects an error in the measurement devices, the nurses' station is notified by activating the buzzer*
    - Involved domains : factors database, periods & ranges, nurses' station

Context diagram



Considerations on the context diagrams

- The context diagram shows all and only the domains and the interfaces that are relevant to the problem
  - Drawing correctly the context diagram is a critical activity
- In order to identify all the domains that have a role in the problem, the context diagram has to be refined. The requirements of the problem have to be iteratively considered, in order to
  - Add new aspects, not formerly considered, to a domain already identified.
  - Identify new domains
  - Delete from domains the aspects that are scarcely relevant
  - Delete domains that are scarcely relevant

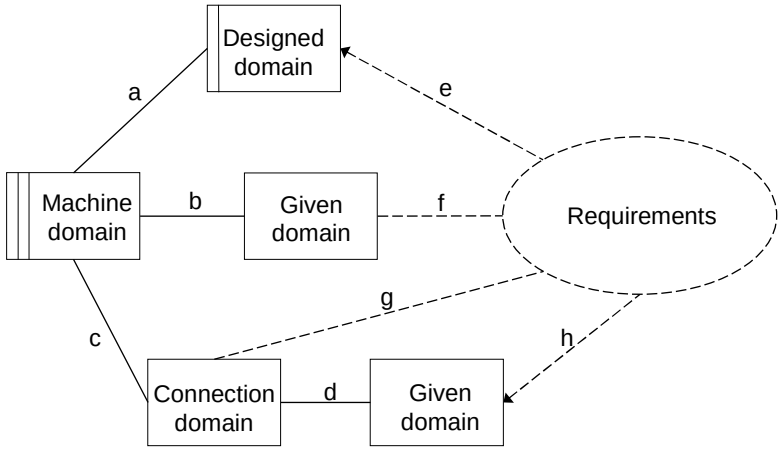
From the context diagram to the problem diagram

- The context diagram identifies the problem and solution spaces, but does not specify the requirements
- Requirements are related to problem domains
  - It is important to explicitly show these relationships
- The problem diagram extends the context diagram and provides a first base for problem analysis:
  - Requirements are introduced
  - Connections between requirements and domains are shown, thus highlighting the role played by every problem domain
- Requirements
  - They may constrain the behaviour of the domains to which they are connected
  - Graphically they are represented through dotted ovals

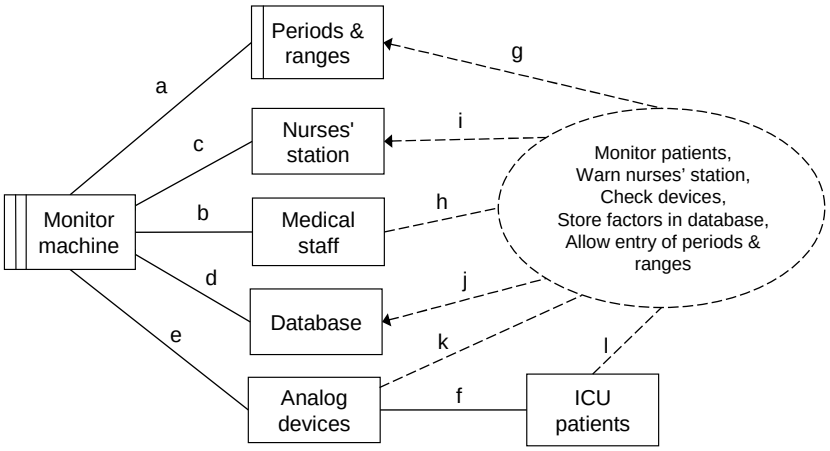
The problem diagram

- The problem diagram introduces:
  - Annotations on the interfaces: they provide information on the phenomena shared between domains, and show which domain controls each phenomenon
    - Represented by letters (that work as labels)
    - Letters are then expanded into domain!{phenomenon}
  - Requirement reference: indicate the relationships connecting requirements and domains
    - Represents as dotted lines or dotted arrows. The latter constrain the referenced domain, while the former just describe the connected domain
    - lines: indicative descriptions
    - arrows: optative descriptions

A generic problem diagram

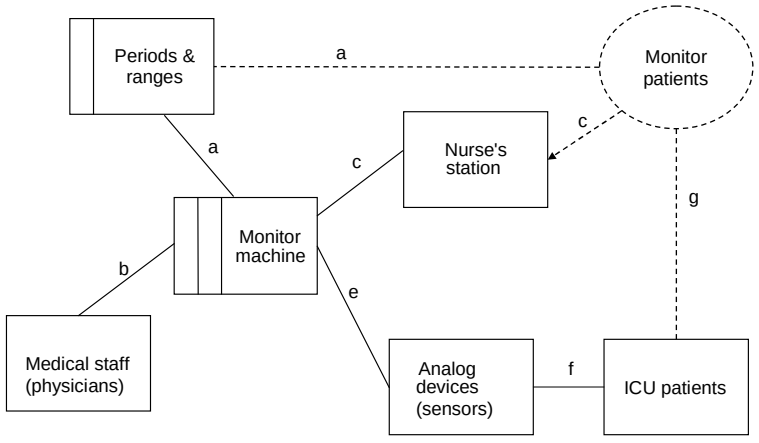


ICU: problem diagram



a: Period, Range, Patient name, Factor
b: EnterPeriod, EnterRange, EnterPatientName, EnterFactor
c: Notify
e: RegisterValue
f: FactorEvidence
d: Factor, Patient

ICU problem diagram



a: Period, Range, Patient name
b: EnterPeriod, EnterRange, EnterPatientName
c: Notify
e: RegisterValue
f: FactorEvidence
g: VitalFactor, Patient

Complexity and decomposition

- The complexity of the problem makes it difficult to identify all the relationships that exist among the domains.
- A possible way to tackle the problem is to decompose it into simpler and smaller subproblems.
  - Top-down functional decomposition
  - Use case decomposition
  - Problem frames

Problem decomposition

- According to the Problem Frames technique, the problem is decomposed into subproblems that belong to recognizable and familiar classes.
- In this way it is possible to exploit the knowledge of problems
  - of difficulties
  - of possible solutions,
  - …
- A sort of “library” of frequently recurring problems is build
  - A new problem can be modelled according to a well-known structure;
  - Experience can be reused.
- Problem frames are subproblem classes the analysts are familiar with
  - They do not describe the problem completely
  - They are a guide to identify (sub)problem patterns

Characteristics of subproblems

- Every subproblem is complete
  - It includes a machine, problem domains and requirements
  - Every subproblem has to be analysed as if the others were already solved.
- Subproblems exist in parallel
  - Every subproblem is a projection of the whole problem
- Subproblems are concurrent
  - We must pay attention to interactions!
- Some familiar classes of problems are composed

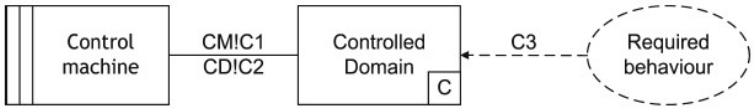
Problem frames

- Patterns that can be used to understand (problem frames) and represent (frame diagrams) the structure of a problem
- They refine context diagrams
- They describe the problem structure and domains according to known problem types
- They guide the analysts in the creation of the requirements specification document
- A complex problem can be described by means of a suitable combination of elementary problem frames (multi-frame problem)

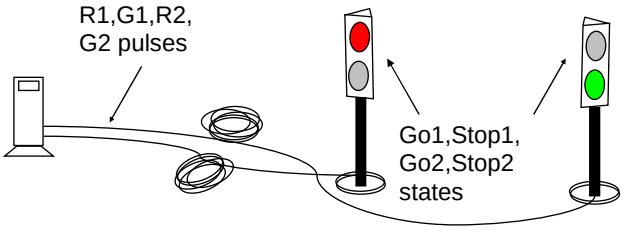
Required behaviour problem

'*There is some part of the physical world whose behaviour is to be controlled so that satisfies certain conditions. The problem is to build a machine that will impose that control.'*

Required behaviour frame



Example: control traffic lights on a one-way road

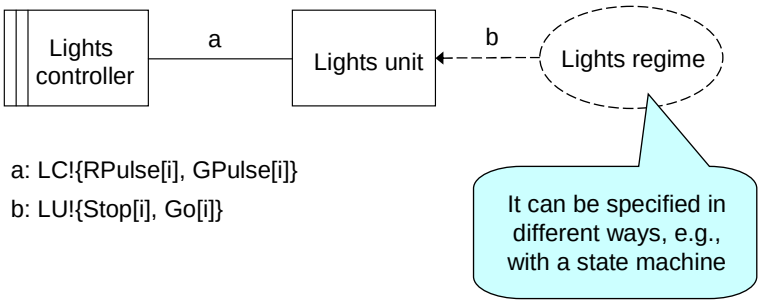


- Two portable traffic signal units connected to a small computer enforce one-way traffic over a section of road under repair. Each signal has a Stop and a Go light.
- The computer can send RPulses and GPulses to each unit. A certain regime of Stop and Go lights is required.

Example: traffic lights control

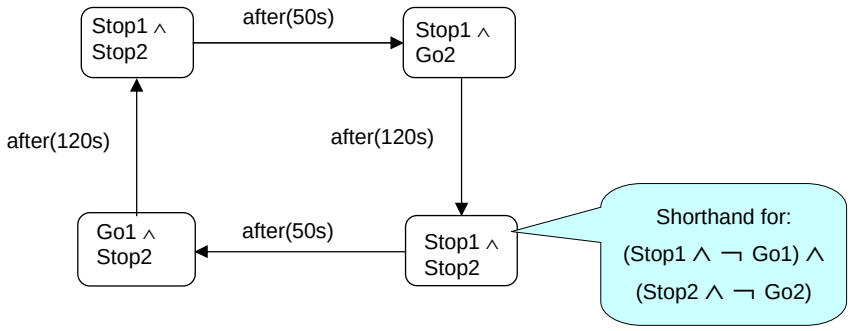
- The computer controls the lights by sending RPulses and GPulses, to which the traffic lights react by switching on and off the Red and Green light.
- The traffic lights have to repeat the following cycle:
  - For 50 seconds both TLs are red,
  - Then, for 120 seconds TL1 is red and TL2 is green,
  - Then ,for 50 seconds both TLs are red
  - Then, for 120 seconds TL2 is red and TL1 is green.

An example of required behaviour frame



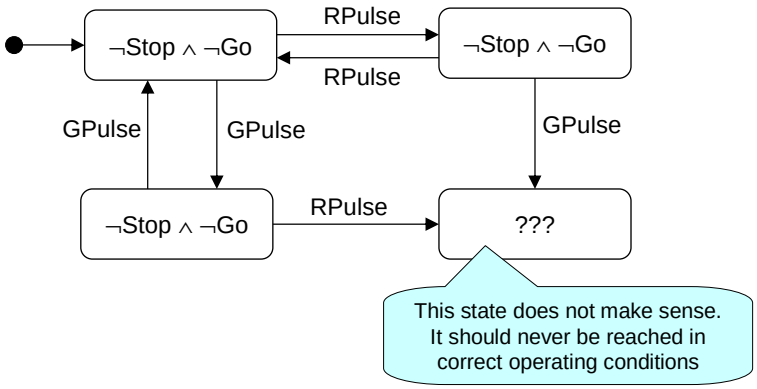
State machine describing requirements

- This is the behaviour we want to achieve
- I.e., this is the behaviour the controller must guarantee
- I.e., this is the behaviour of the environment when coupled with a properly developed machine



State machine describing the environment

- The environment is described by specifying the behaviour of a traffic light unit (i.e., specifying the reactions of the TLU to all the possible commands in every possible state)



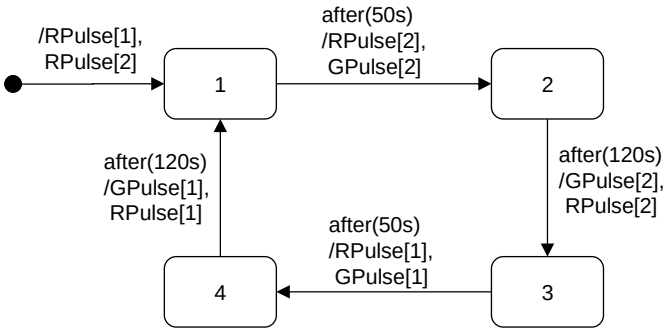
Specifications

- The specification describes the machine behaviour at the interface with the environment
  - i.e., in terms of phenomena shared with TL1 and TL2
- To change, we use a program-like notation, instead of the usual state machine

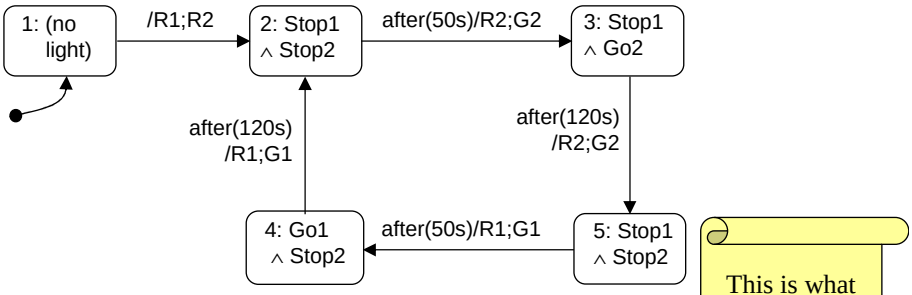
```
{RPulse[1], RPulse[2];
  forever {
    wait 50 seconds; RPulse[2], GPulse[2];
    wait 120 seconds; GPulse[2], RPulse[2];
    wait 50 seconds; RPulse[1], GPulse[1];
    wait 120 seconds; GPulse[1], RPulse[1];
 }}
```

Specifications

- Of course, specifications can be given through a state machine
- This is the statechart that describes the behaviour of the machine at the interface with the environment



A combined description

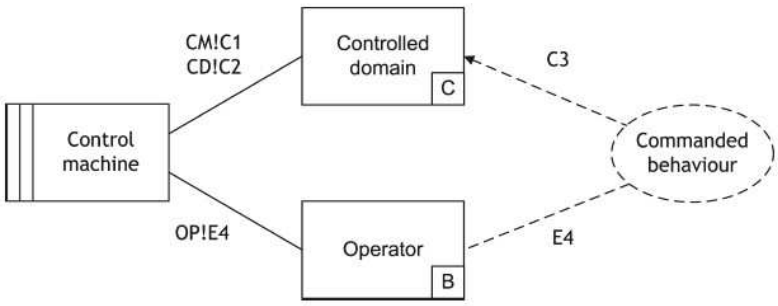


- How to use this description ...
  - to show the customer the requirement?
  - to show the programmer the specification?
  - to show the engineer the (assumed) properties W?
- What actually are the properties of the light units?

Commanded behaviour problem

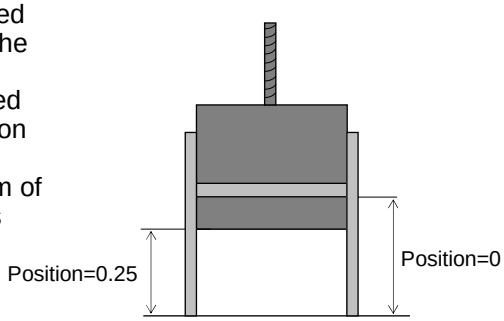
'*There is some part of the physical world whose behaviour is to be controlled in accordance with commands issued by an operator. The problem is to build a machine that will accept the operator's commands and impose the control accordingly.*'

Commanded behaviour frame

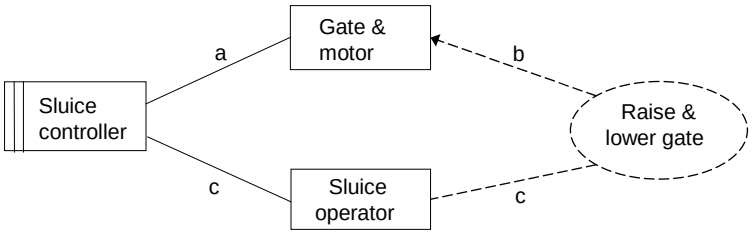


Example: sluice gate control

- A small sluice, with a rising and a falling gate, is used in a simple irrigation system. A computer system is needed to raise and lower the sluice gate in response to the commands of an operator.
- The gate is opened and closed by rotating vertical screws. The screws are driven by a small motor, which can be controlled by clockwise, anticlockwise, on and off pulses. There are sensors at the top and bottom of the gate travel; at the top it is fully open, at the bottom it is fully shut.
- The connection to the computer consists of four pulse lines for motor control, two status lines for gate sensors, and a status line for each class of operator commands.



An example of a commanded behaviour frame



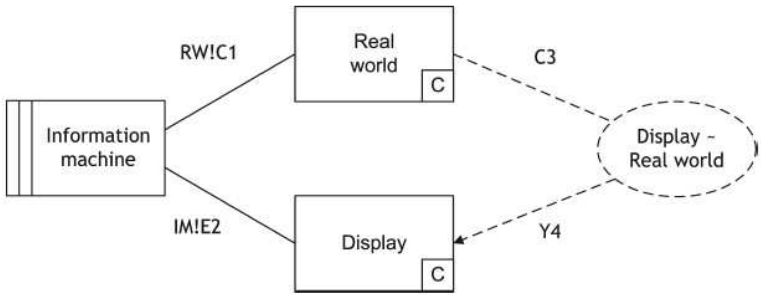
a: SC!{Clockw, Anti, On, Off}
GM!{Top, Bottom}

b: GM!{Open, Shut, Rising, Falling}
c: SO!{Raise, Lower, Stop}

Information display problem

'*There is some part of the physical world about whose states and behaviour certain information is continually needed. The problem is to build a machine that will obtain this information from the world and present it at the required place in the required form.*'

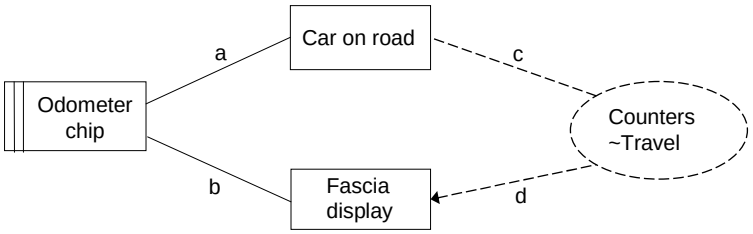
Information display frame



Odometer display problem

- A microchip computer is required to control a digital electronic speedometer and odometer in a car.
- One of the car wheels generates pulses as it rotates. The computer can detect these pulses and must use them to set the current speed and total number of miles travelled in the two visible counters on the car fascia. The underlying registers of the counters are shared by the computer and the visible display.

An example of information display frame



a: CR!{WheelPulse}

b: OM!{IncSpeed, IncDist, DecSpeed, DecDist}

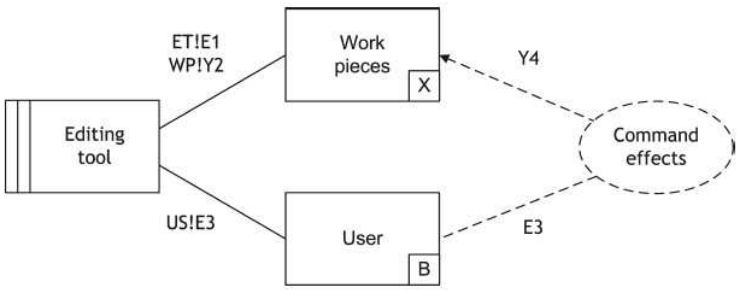
c: CR!{Speed, CumDist}

d: FD!{SpeedCount, DistCount}

Simple workpiece problem

'*A tool is needed to allow a user to create and edit a certain class of computer-processable text or graphic objects, or similar structures, so that they can be subsequently copied, printed, analysed or used in other ways. The problem is to build a machine that can acts as this tool.'*

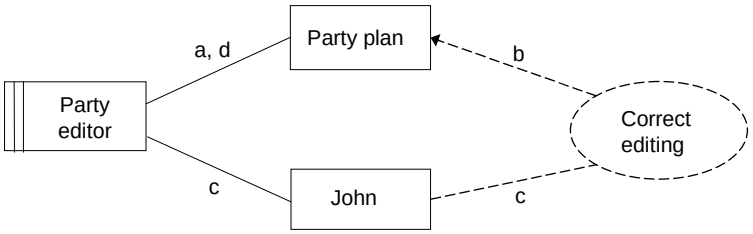
Simple workpiece frame



Example: party plan editing

- John needs a system to keep track of the many parties he gives and the many guests he invites to them. He wants a simple editor to maintain the “party plan” information.
  - Party plan: a list of parties, guests, and a note of who is invited to each party.

An example of simple workpiece frame



a: PE!{ PlanOperations}

b: PP!{ PlanStates}

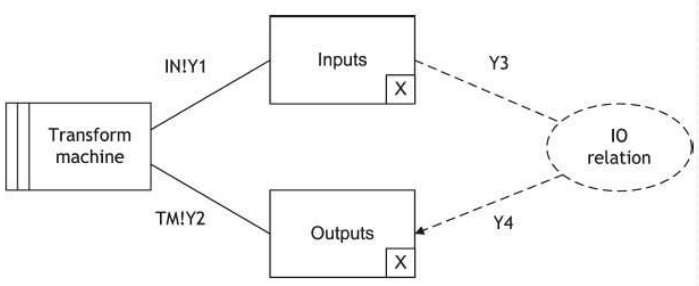
c: J!{ Commands}

d: PP!{ PlanEffects}

Transformation problem

'*There are some given computer-readable input files whose data must be transformed, according to certain rules, to give certain required output files. The output data must be in a particular format. The problem is to build a machine that will produce the required outputs from the inputs.'*

Transformation frame

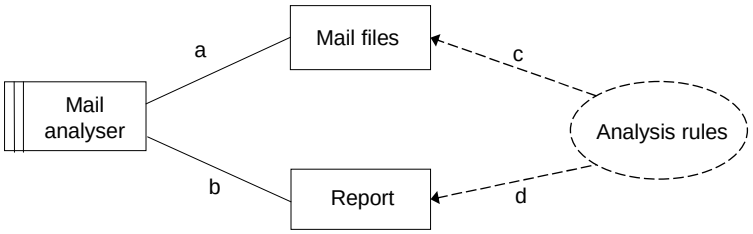


Example: mail files analysis

- Fred needs a program to analyse some patters in his email. He is interested in the average number of message received in a week, the average and maximum message length, etc.
- He would like a report like the following, with a line for each correspondent

Name	days	#In	Max	Avg	#Out	Max	Avg
Dilbert	124	19	52121	6027	17	21941	2123
Alice	92	31	13249	1736	37	34773	2918

An example of transformation frame



a: MF!{MsgDir, File, Line, Char}

b: MA!{ReportLine, Char}

c: MF!{Msg, From, To, Date Length}

d: RP!{LineData}

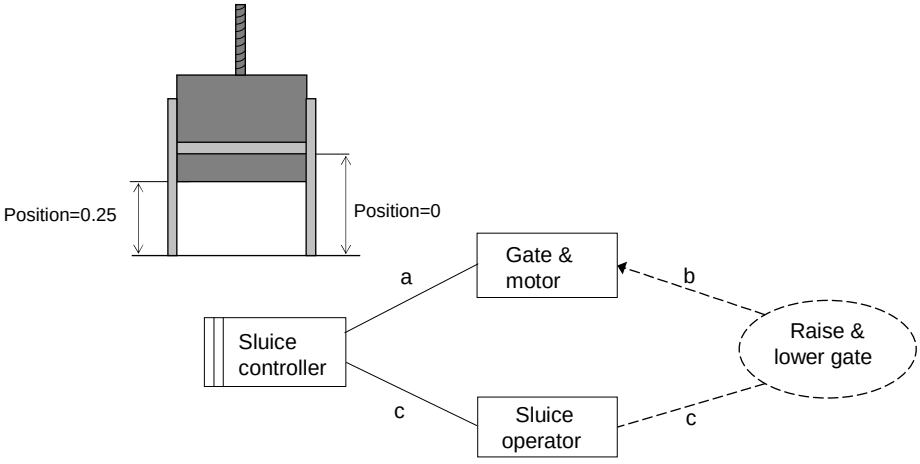
Warning

- In principle, new problem frames could be defined, but it would be necessary to demonstrate that they are actually useful and applicable to a set of problems.
- Problem frames generalize problems: it is not correct to create a new problem frame that nicely adapts to the problem at hand.

Frame concerns

- Frame concerns identify the descriptions that have to be provided with every problem frame.
- Addressing frame concerns adequately means making requirements, domain and specification descriptions that fit together properly.
  - The combination of these descriptions must result in a correctness argument, i.e., they must provide evidence that the proposed machine will ensure that the requirements are satisfied in the problem domain.
  - In the case of the commanded behaviour frame, we must assure that only sensible and viable commands are executed. For instance, the machine should not try to open an already open gate.

Il problem diagram



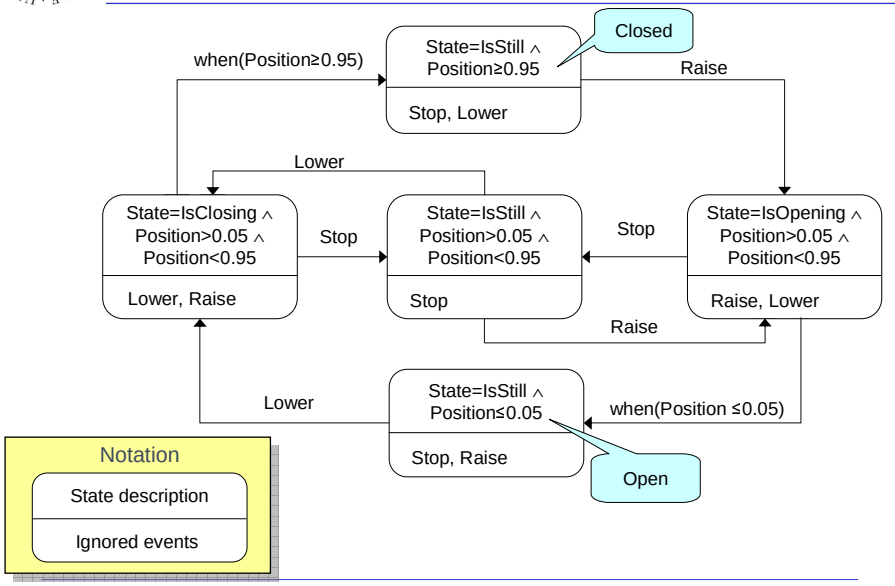
a: SC!{Clockw, Anti, On, Off}
GM!{Top, Bottom}

b: GM!{Open, Shut, Rising, Falling, Still}
c: SO!{Raise, Lower, Stop}

Requirements

- Requirements can be expressed as the desired effects on the problem domain caused by the user's commands (and by other events, such as the gate reaching the completely open or closed position).
- These effects are expressed by means of a state machine.

Requirements



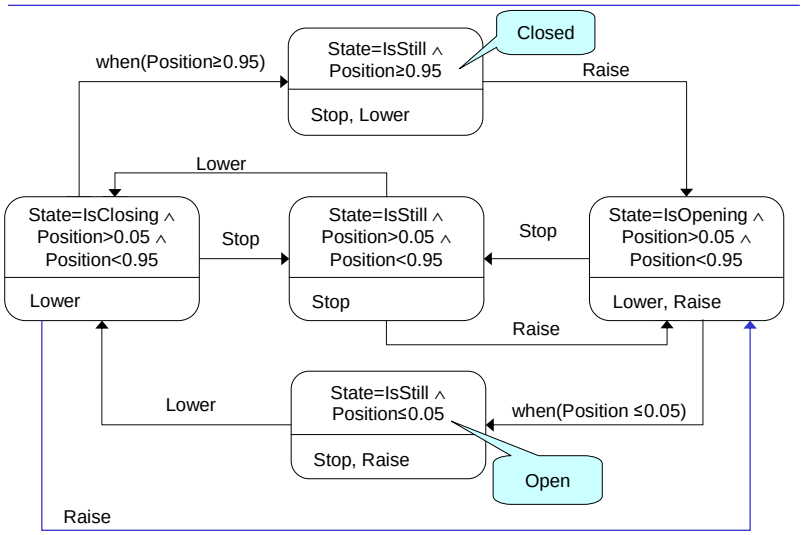
Observation

- In the statechart representing requirements some transition conditions are expressed on the actual position of the gate, rather than on the state of the sensors.
- In fact, requirements represent the user point of view, which generally concerns the real state of objects, rather than the representation of their state which is made available to the machine.

Requirements extension

- The requirements described by the statechart above (taken from M. Jackson's book) do not consider that a Raise command could be issued when the gate is closing.
- We adopt an extended version of these requirements, according to which the gate should switch to the opening state.
- This requires a "good" behaviour of the machine, since just switching the working direction would break the motor (or bring the domain in an unknown state as specified in the problem domain statechart).

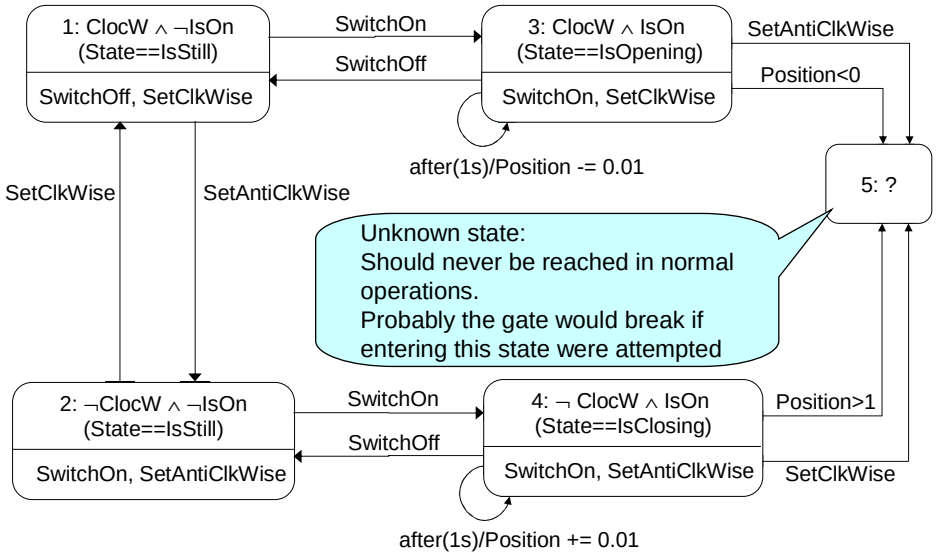
Requirements



Problem domain

- The problem domain is described in terms of the reaction of the gate to commands (i.e., to any possible command, including meaningless ones)
- The reaction to commands is specified in terms of state transitions and/or time spent in states or time taken to complete transitions.
- Again, the problem domain can be described by means of a state machine.

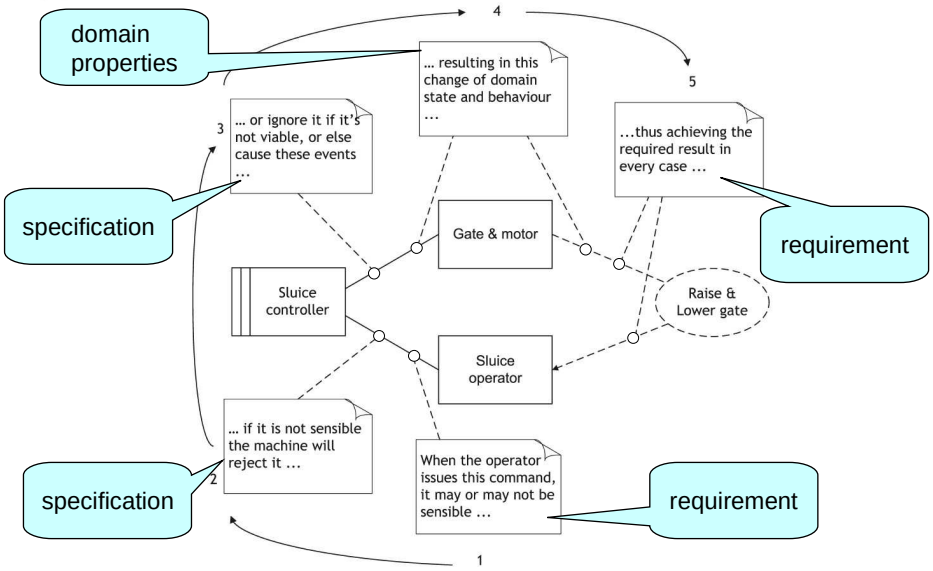
Problem domain



Specifications

- Now, in the PF methodology, we must describe a machine that, combined with the problem domain, satisfy the requirements. The machine will:
  - Detect commands issued by the operator
  - Reject commands that are not sensible according to the sensible commands requirements
  - Ignore those sensible commands that are not viable and cannot be obeyed in the current state of the gate & motor according to the obeyed commands requirement
  - Exploit the game & motor domain properties to achieve the required Rising and Falling (and Still) states in response to the sensible viable commands, according to the obeyed commands requirements.

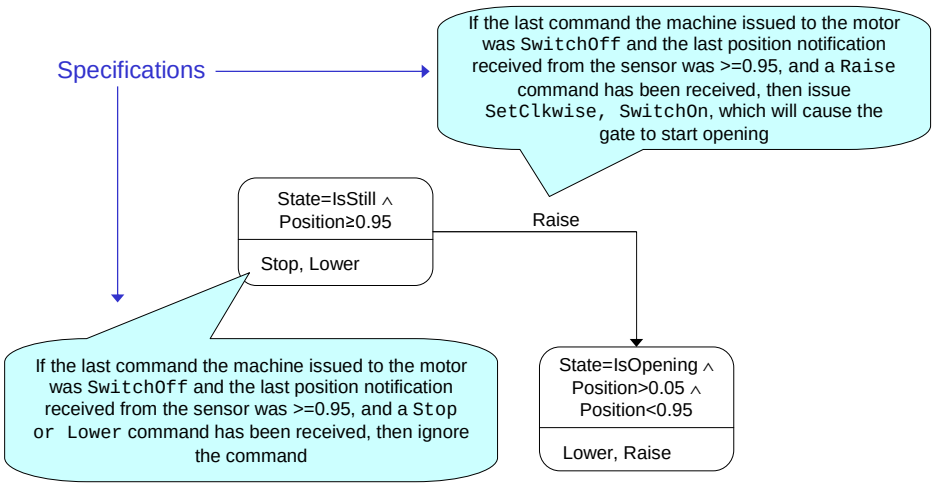
Frame concern in commanded behaviour frame



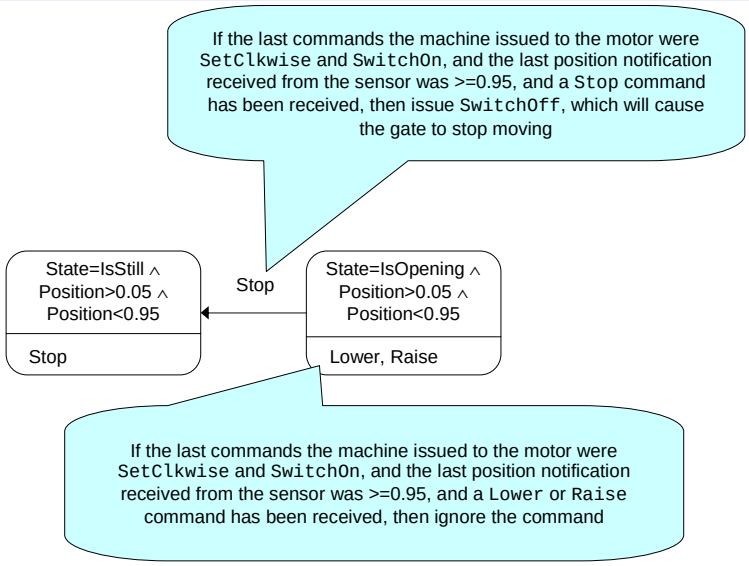
Machine specifications

- Machine specifications are derived by “rewriting” requirements in term of phenomena shared with the machine.

Machine specifications



Machine specifications



Machine specifications

- The rest of the requirements are similarly converted into rules of behaviour that involve only the phenomena shared between the machine and the environment.
- Such rules can be represented in various ways, e.g.,
  - By means of statecharts
  - In some logic notation
  - Informally
  - …