Requirements Specification Using Executable Models

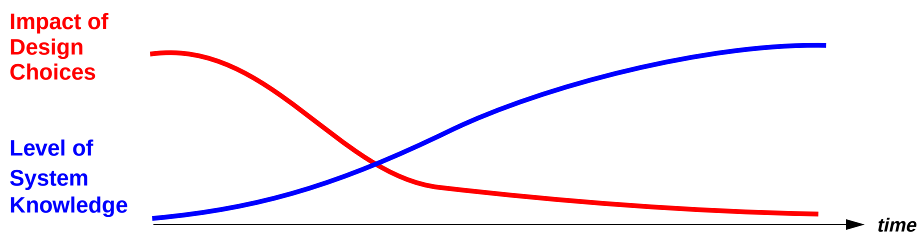
Bran Selic
ObjecTime Limited
bran@objectime.com

Overview

- **Requirements Modeling and Executable Models**
- **Basic Concepts of ROOM**
- **Requirements Capture Process**

Context...

- The most expensive errors in software development are those made early in the development process (foundation for subsequent work)



- Misunderstandings about requirements are among the principal sources of up-front mistakes

=> An explicit requirements specification is required

The Uses of Requirements Specs

- Basis for communication among the users, operators, and developers of a system
- Used to systematically verify the soundness of a requirements set
- Reference for final system certification and acceptance
- Controlling system evolution
  - fitting new requirements over existing set

An Extreme View...

"I maintain that there is only one way to determine the specification for a new piece of software—write the code and see what it looks like."

P.J. Plauger
C/C++ software guru

Problems of Requirements Specs

- **However, even for moderately complex systems, generating requirements specs is *hard***
  - **incomprehensibility**
  - **incorrectness**
  - **ambiguity**
  - **inconsistency (redundancy)**
  - **incompleteness**
  - **instability**
  - **implementability hurdles**
  - **design bias**

Implementability Hurdles

- In the early days of software development implementation often overlapped with design and, sometimes, even with requirements specification
- Hence: "what (requirements) before how (design)"
- However, this is often taken to an extreme:
  - the two are sometimes very difficult to decouple cleanly (e.g., is distribution just an implementation issue or is it a fundamental user-level requirement?)
  - detracts from comprehensive problem understanding
  - can lead to major implementation problems

An Approach: Formal Requirements Specs

- Use of formal specification techniques can mitigate and even eliminate many of the cited problems
- However, no guarantees...
  - e.g., comprehensibility

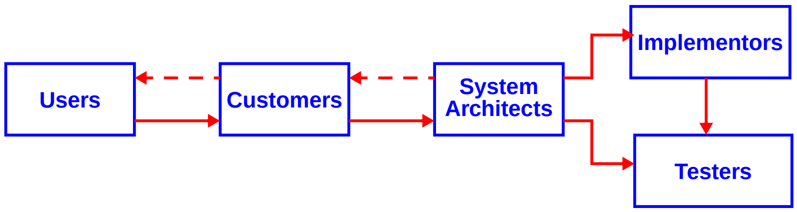
$$\frac{n \Rightarrow E_1 \xrightarrow{((a,n) \otimes \omega)} s_1}{n \Rightarrow (E_1 + E_2) \xrightarrow{((a,n) \otimes \omega)} s_1}$$

$$\frac{(s_1 \xrightarrow{((a,n) \otimes \omega)} s'_1)(s_2 \xrightarrow{((a,n) \otimes \omega)} s'_2)}{(s_1 \parallel_A s_2 \xrightarrow{((a,n) \otimes (\omega_1 \parallel \omega_2))} ([k_1,\omega_1]s'_1 \parallel_A [k_2,\omega_2]s'_2))}$$

  - e.g. implementability: "...we assume a loss-free broadcast communication medium with zero delay..."

Traditional Requirements Process

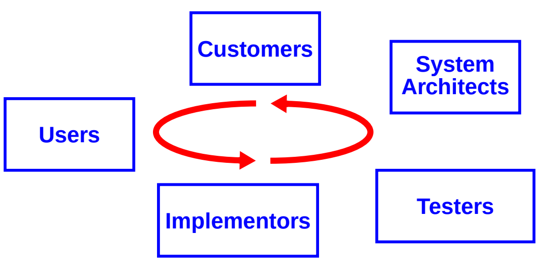
- Cascade process



  - mostly unidirectional
  - source of many of the problems cited earlier

A Different Process Model

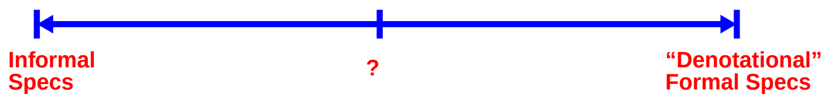
- Roundtable elicitation model (design-build teams)



  - synergy: ensures all stakeholders’ interests are taken into account
  - facilitates agreement
  - protects against downstream corruption of requirements

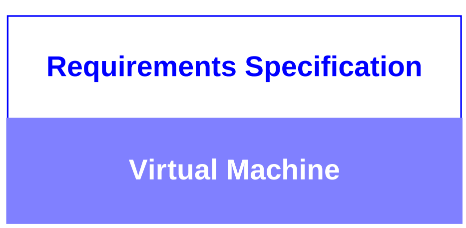
We Need...

- A ***formal*** **requirements specification technique that represents a** ***balance*** **between the extremes of:**
  - **highly-idealized denotational formalisms (that are often difficult to understand and potentially infeasible) and**
  - **generally understandable but seriously flawed informal specifications**



Operational Specifications

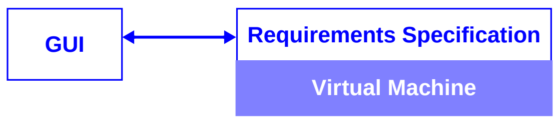
- **Specifications in the form of programs written for a formally specified "virtual machine"**
  - **requirements specifications = executable models**



- **These specifications define elements of** ***structure*** **and** ***behavior***

Advantages of Operational Specifications

- Formality a basis for ensuring:
  - consistency, completeness, precision, correctness
- Facilitates system understanding through observing the executing model "in action"
  - through suitable GUI interfaces, can be presented in forms directly comprehended by users and operators



  - usually accelerates the requirements specification process and cuts down on requirements instability

The Virtual Machine

- The abstraction level of the virtual machine can have a significant impact
- A highly abstract virtual machine
  - reduces bias towards particular designs
  - increases expressive power (ability to directly model complex phenomena)
  - may have complex and very subtle semantics
  - may have inappropriate semantics for a given problem domain
  - may result in unimplementable specifications

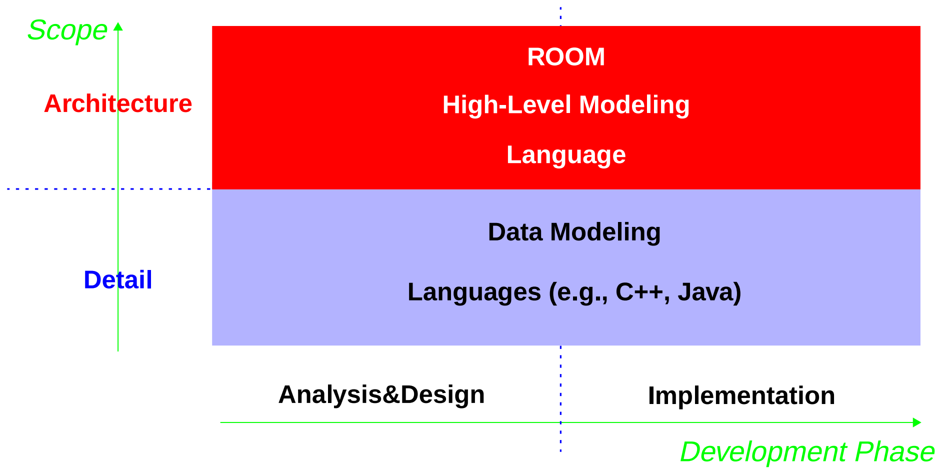
ROOM Approach

- A middle ground: virtual machine specialized for distributed reactive system domain
- Object-oriented approach: takes advantage of the features of the object paradigm (classification, compositionality, encapsulation)
- A full cycle language: modeling concepts can be applied to:
  - requirements specification
  - analysis and design
  - (implementation step can be automated)

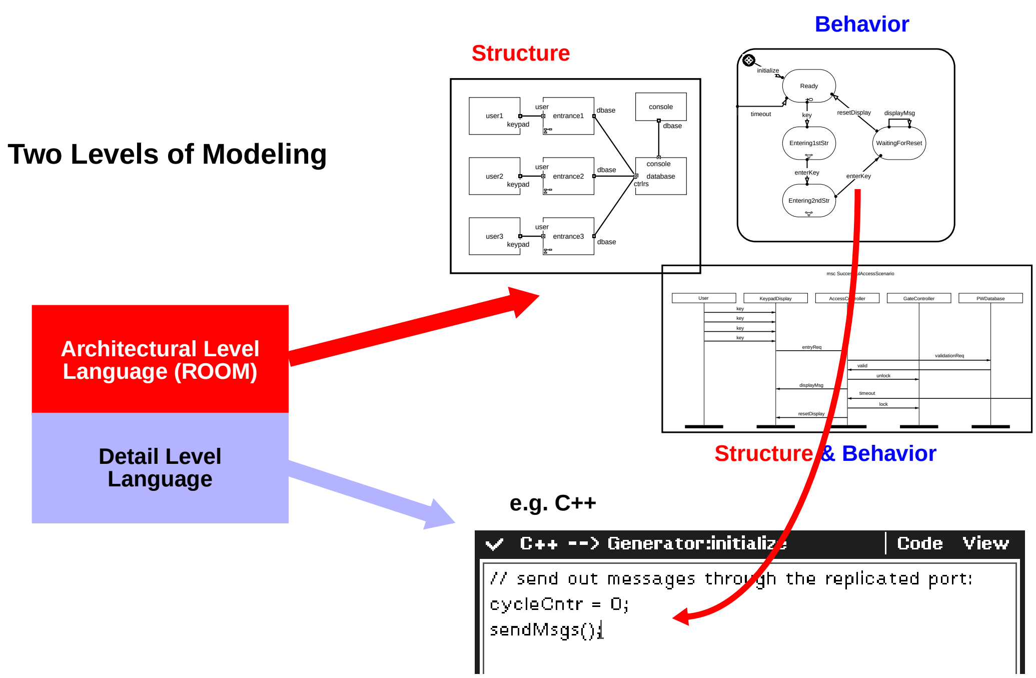
- Requirements Modeling and Executable Models
- **Basic Concepts of ROOM**
- Requirements Capture Process

The Languages of ROOM

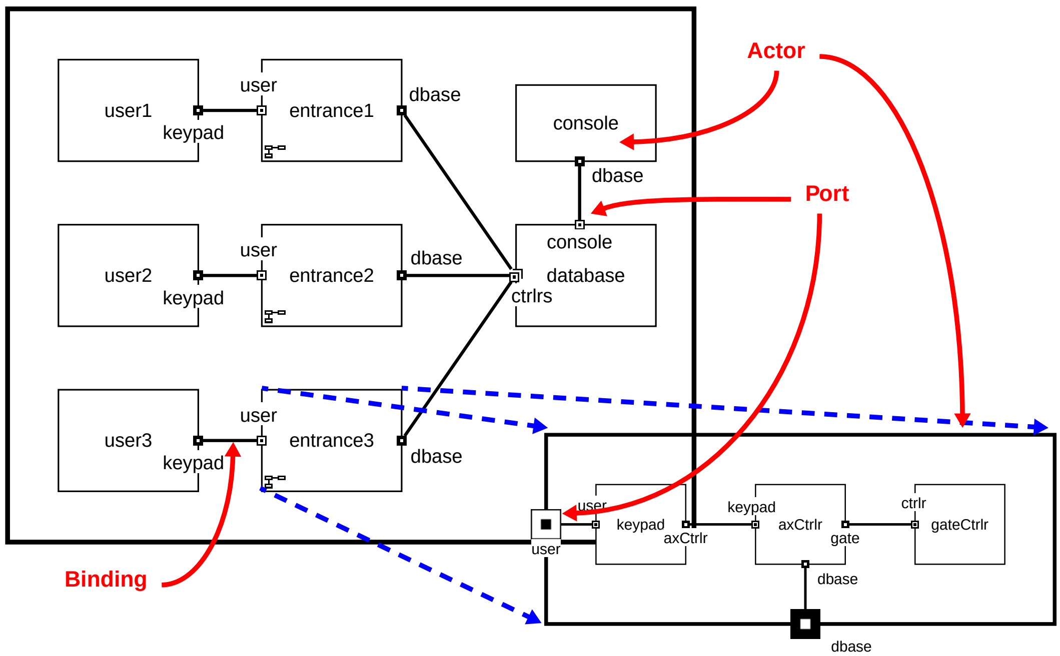
- **Phase-independent modeling concepts split across two formally correlated levels**



Modeling in ROOM



Basic Structure Modeling



Interfaces and Protocols

- **Each actor interface is defined by its protocol attribute**
  - **an extension of the classical interface concept to cover information exchange sequences**

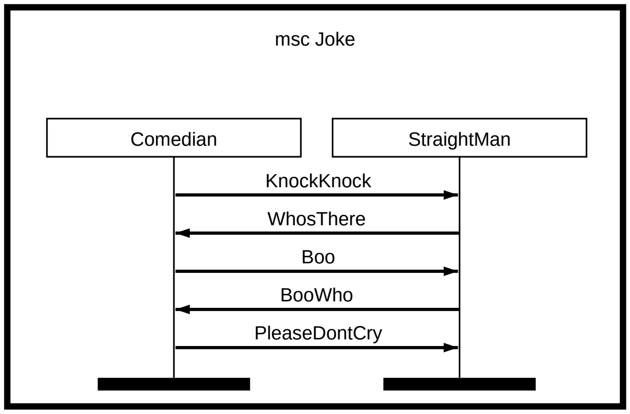
JokeProtocol — Protocol View

In Signals	Data Class	Out Signals	Data Class
KnockKnock	Null	WhosThere	Null
Boo	Null	BooWho	Null
PleaseDontCry	Null		

  - **set of incoming and outgoing message types**
- **Only compatible protocol-based interfaces can be bound to each other**

Message Sequence Charts

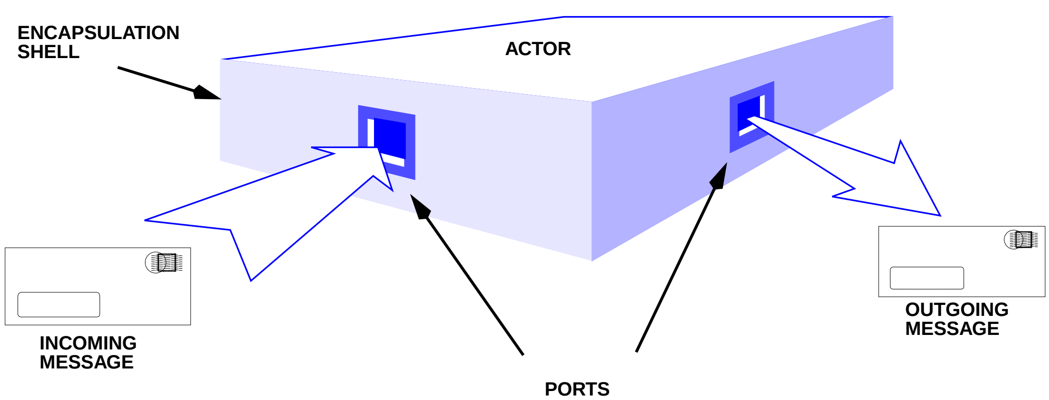
- Message sequences are expressed by ***Message Sequence Charts*** (MSCs)



- Defined by ITU standard Z.120

Actors

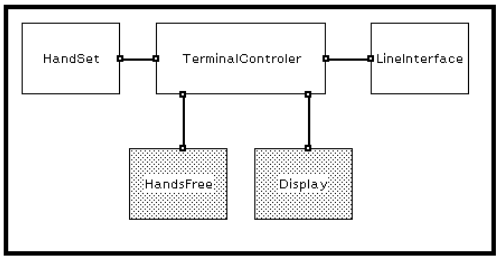
- The active objects of ROOM are called ***actors***



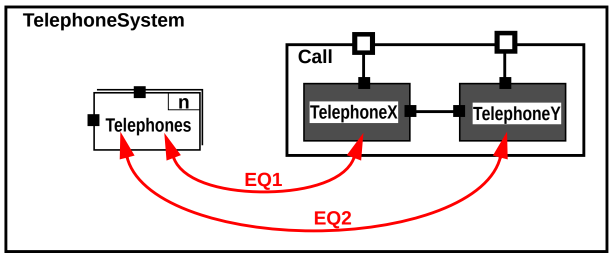
- Actors can send and receive messages through one or more ports

Modeling Dynamic Structures

- **Components created after their container**

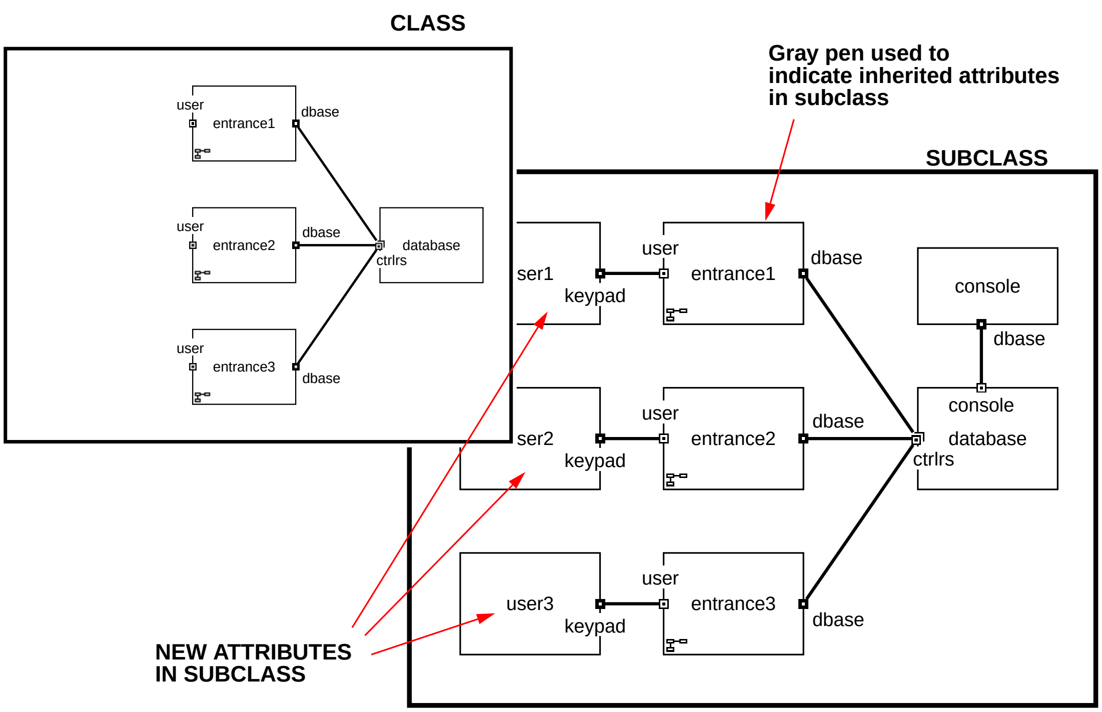


- **Multiple containment (support for roles and dynamic relationships)**



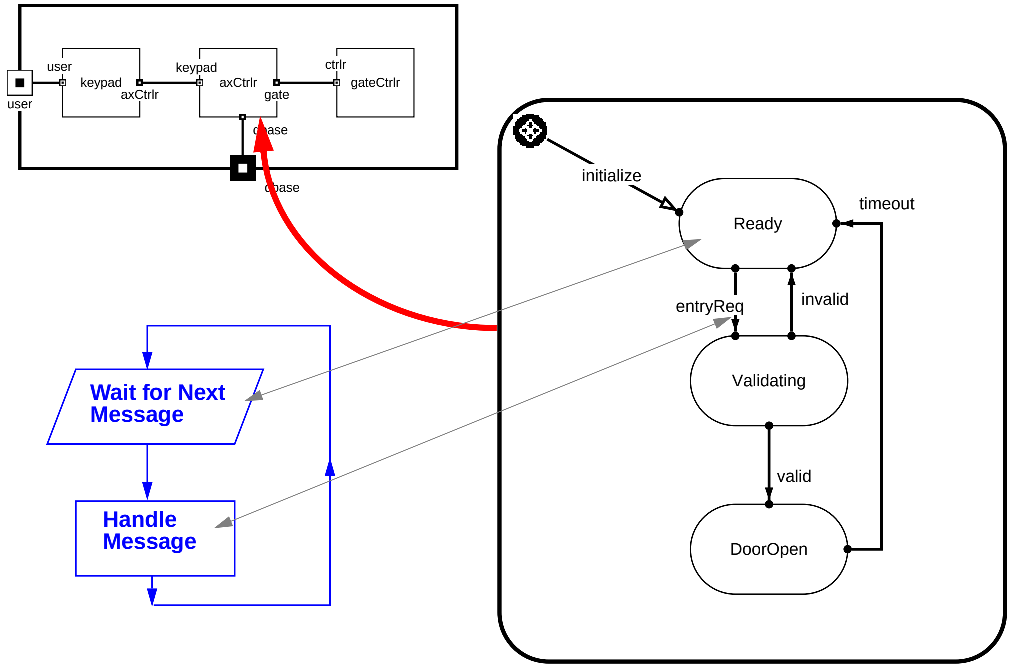
ObjecTime Right in Real-Time

Actor Class Inheritance — Structure

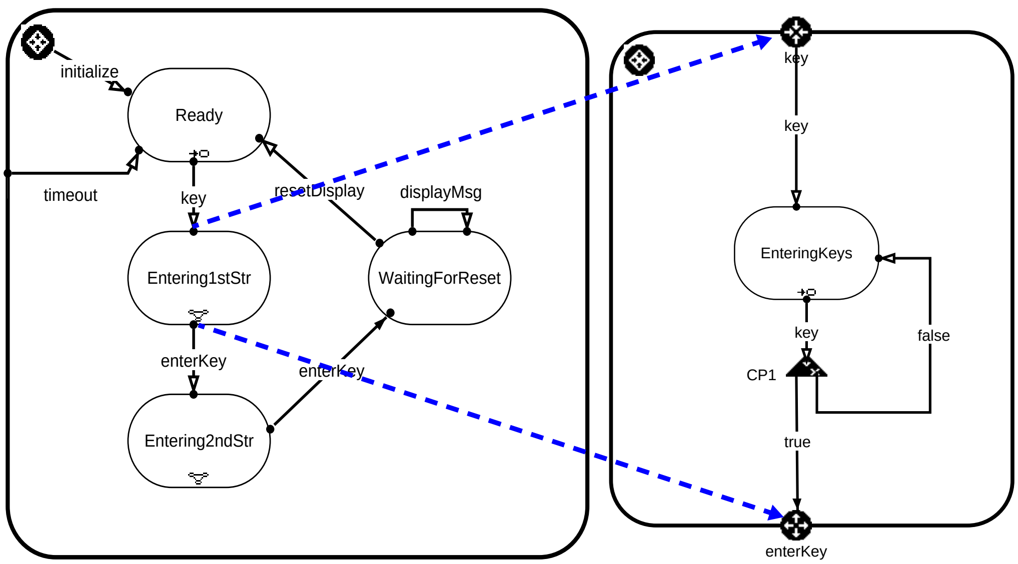


ObjecTime Right in Real-Time

Modeling Behavior — ROOMchart Basics



Modeling Behavior — Hierarchical States

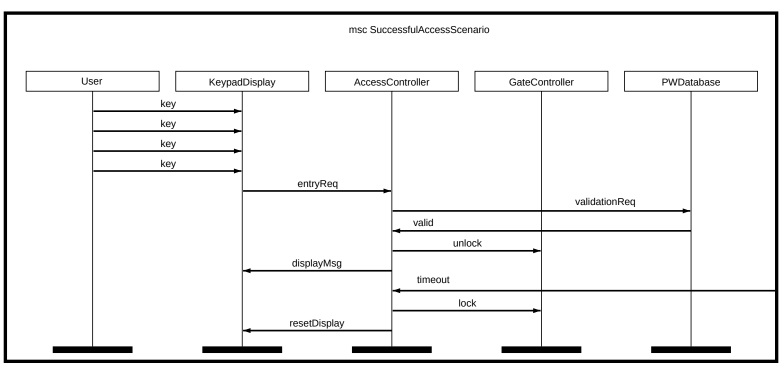


ROOMcharts vs. Statecharts

- ROOMcharts incorporate the major features of the object paradigm, notably, encapsulation and inheritance
- ROOMcharts do not allow "and" states and their accompanying idealizations due to concerns regarding:
  - virtual machine complexity (semantics of steps)
  - reliability of implicit communication
  - general implementability of broadcast semantics

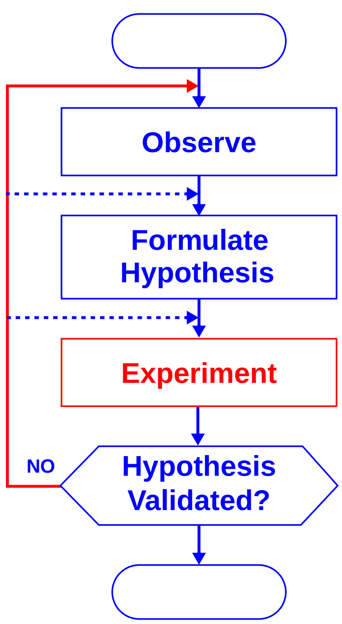
End-to-End Behavior Modeling

- Use of Message Sequence Charts (ITU Z.120)



- Requirements Modeling and Executable Models
- Basic Concepts of ROOM
- **Requirements Capture Process**

An Analogy — The Scientific Method

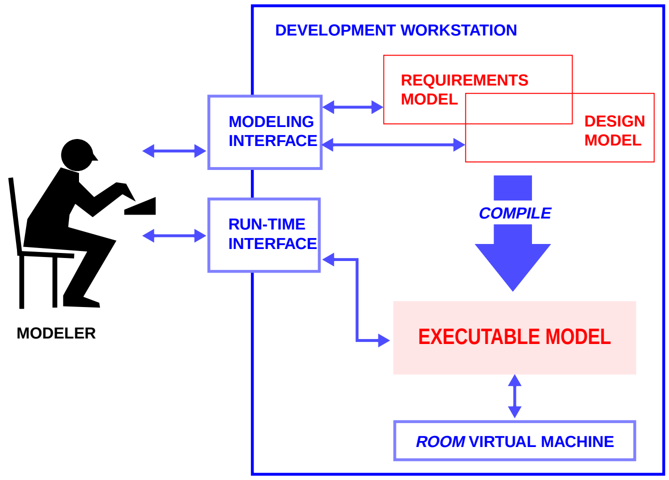


Hypothetico-deductive (HD) method:

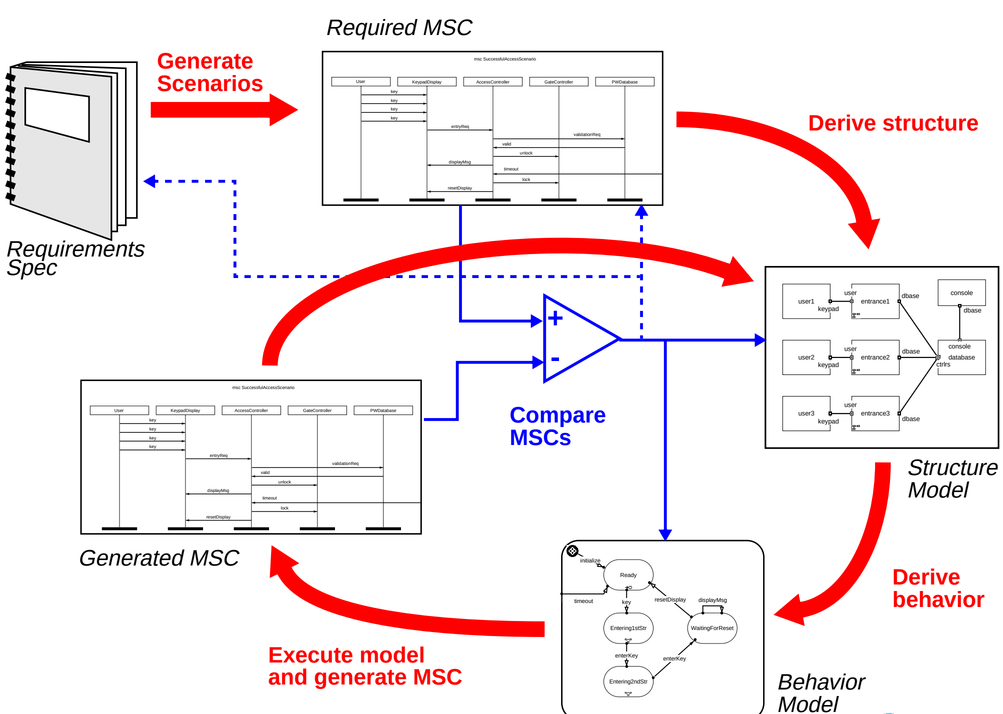
Developing specifications and software for complex systems has much in common with this process

- ***Iteration*** **and** ***experimentation*** **are the key elements of this process**

ROOM and Automation



Typical ROOM Microcycle

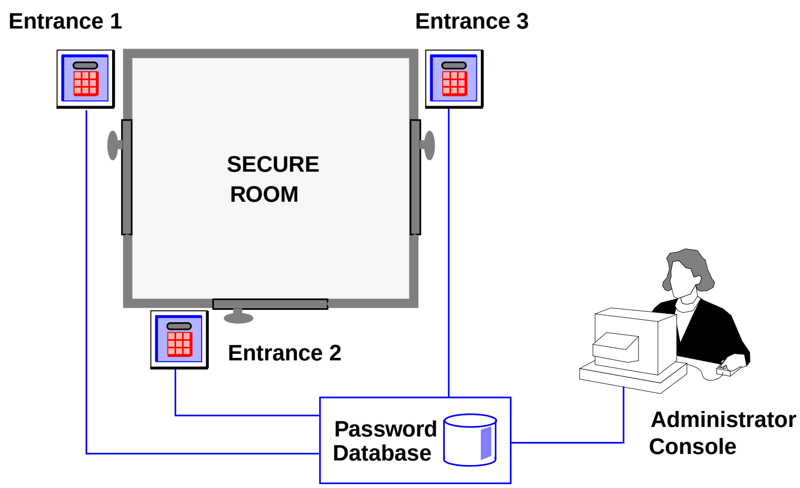


But Isn't This Just Design?

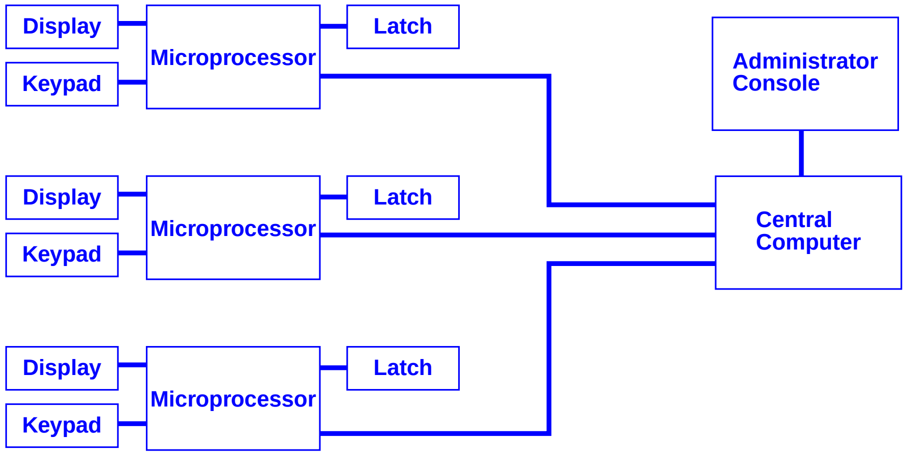
- No, since *what is being "designed/modeled" during requirements identification is not the software to be developed but the requirements*
- Example: Secure room problem
  - design the software for a system that will allow access to a "secure" room only to authorized personnel:
  - To gain access, it is necessary to key in a user id and a personalized password on the keypad situated next to each entrance

The Secure Room — Requirements

- A room with 3 secured entrances



Requirements — Hardware Configuration

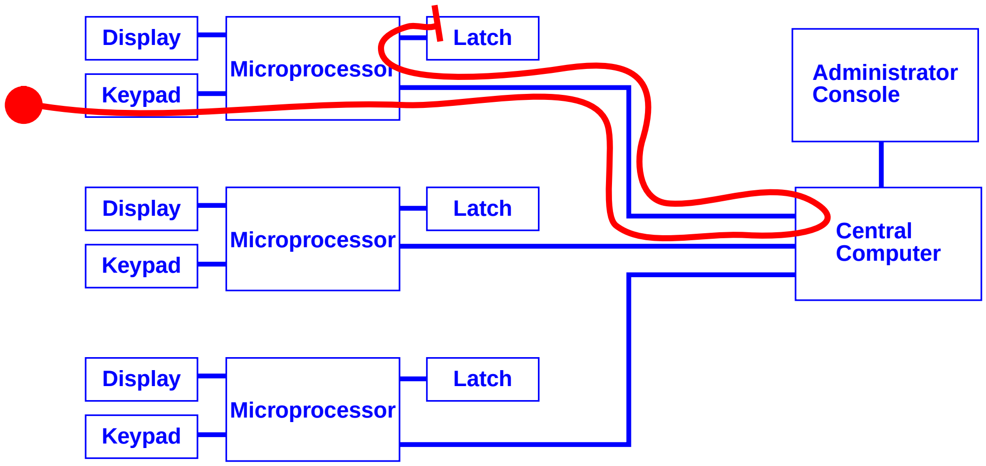


Requirements — Sample Usage Scenario

1. User enters user id on keypad
2. User enters personal password on keypad
3. System validates user id and password against central database
4. For valid access codes, door is unlocked for 3 seconds during which user can enter
5. After 3 seconds expire, door is locked again

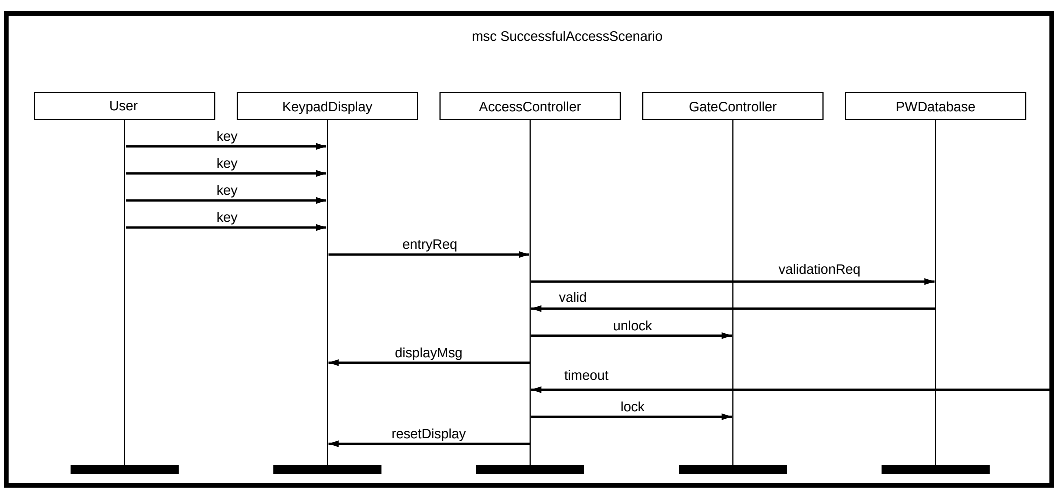
Usage Scenario — Graphical Rendering (1)

Use-case Map (UCM):

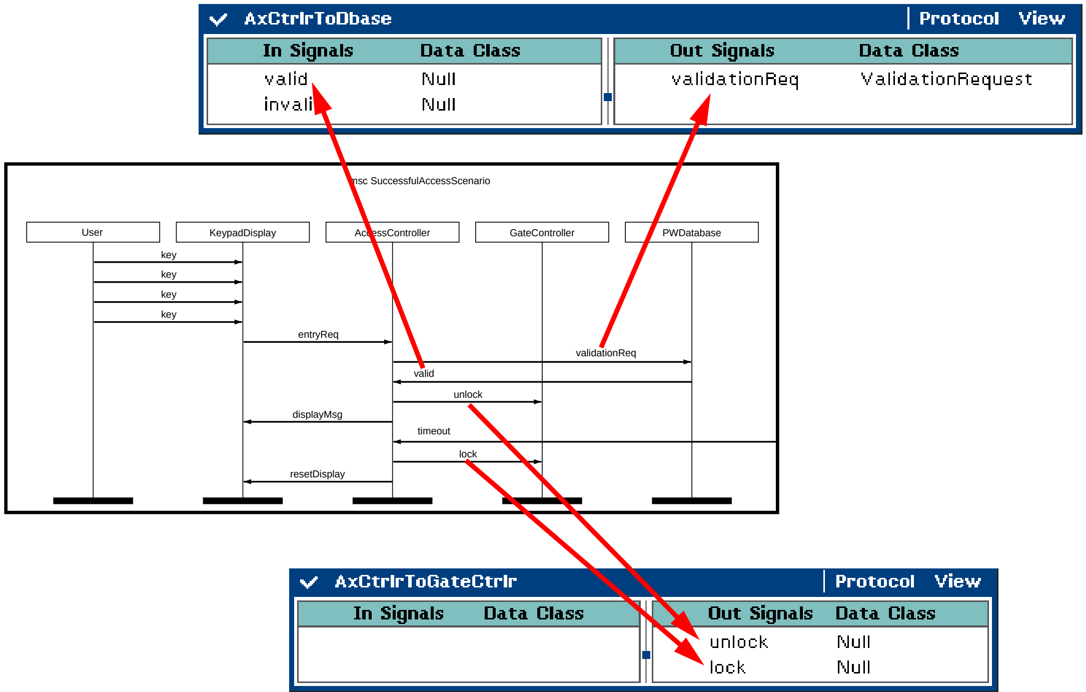


Usage Scenario — Graphical Rendering (2)

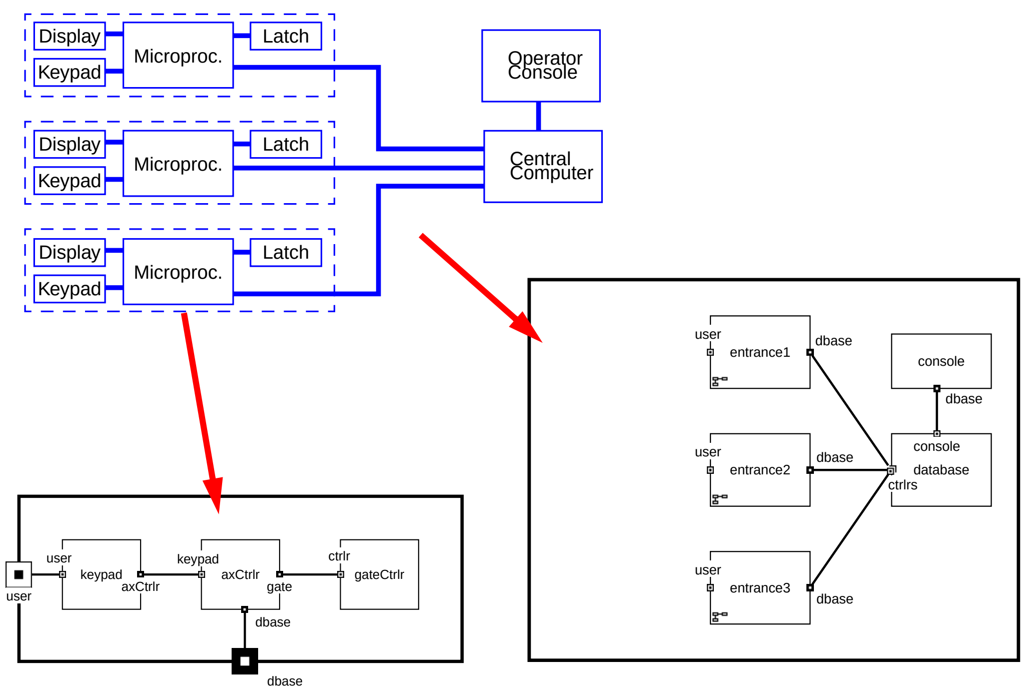
Message Sequence Chart (MSC):



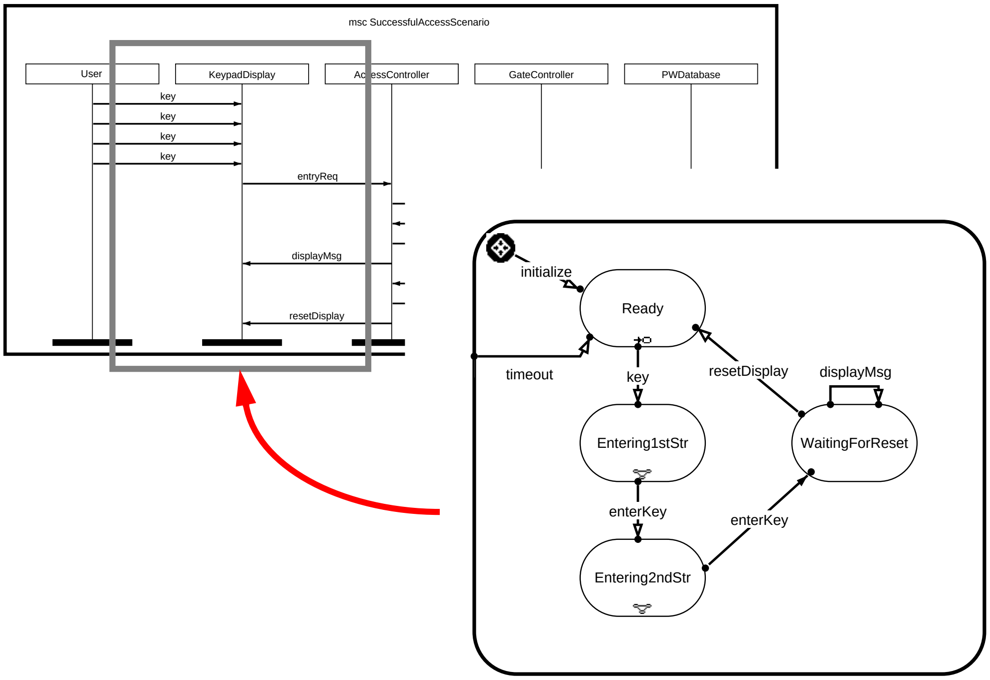
Deriving Protocols From Scenarios



Deriving Structure From Requirements



Deriving High-Level Behavior



Summary

- Specifying requirements for complex systems is a hard problem
- ROOM is a formal modeling language that allows the capture the structural and behavioral requirements of real-time systems
- The same modeling concepts are applicable to the design phase greatly facilitating the transition from requirements to design
- Extensive industrial experience has proven the viability of the approach

Appendix: More About ROOM

- **Real-Time Object-Oriented Modeling (ROOM)**
- **Developed at Bell-Northern Research**
  - **suitable for event-driven distributed systems**
  - **full-cycle method (A → D → I)**
  - **uses a formal graphical modeling language**
  - **Described in: B. Selic, G. Gullekson, and P. Ward, “Real-Time Object-Oriented Modeling,’ John Wiley & Sons, NY, 1994.**