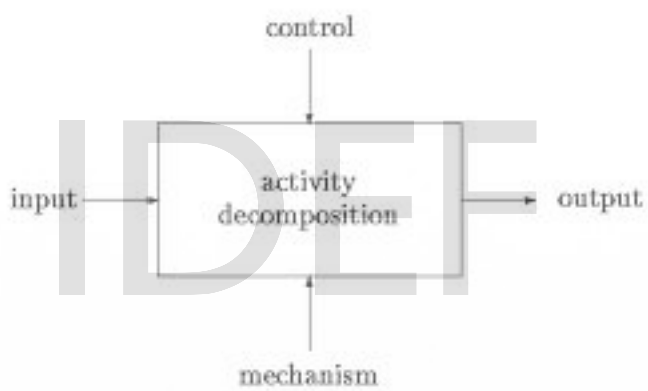
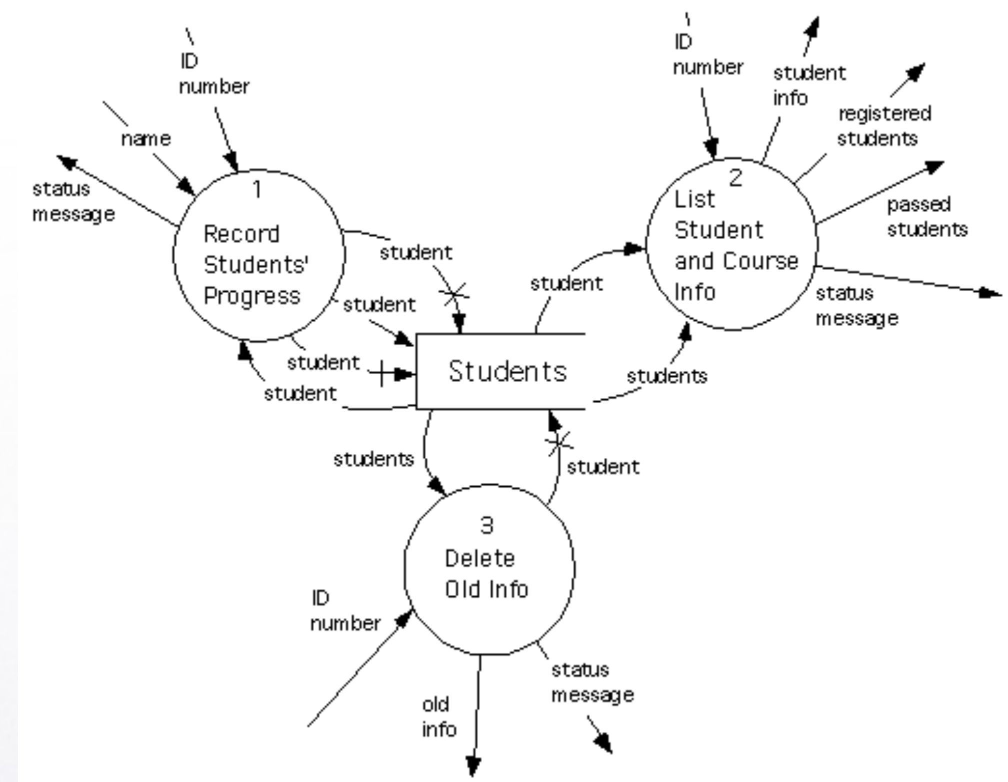
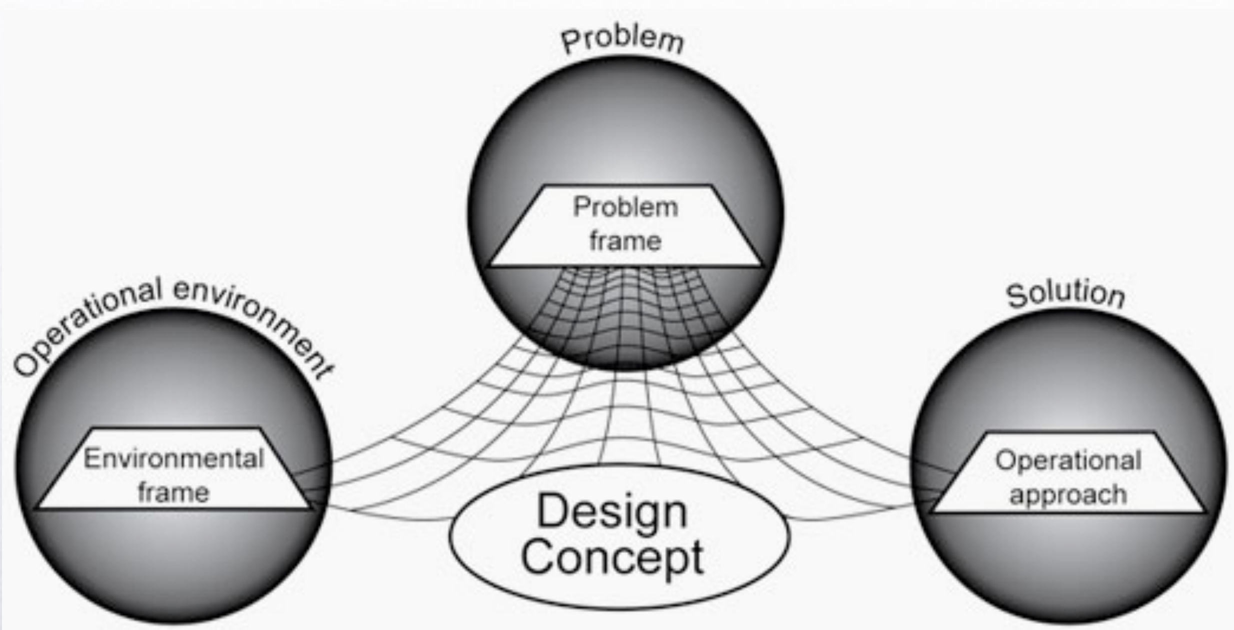
PMR 5020
Modelagem do Projeto de Sistemas

Aula 8: Paradigmas e métodos formais especificação de projeto

O primeiro paradigma

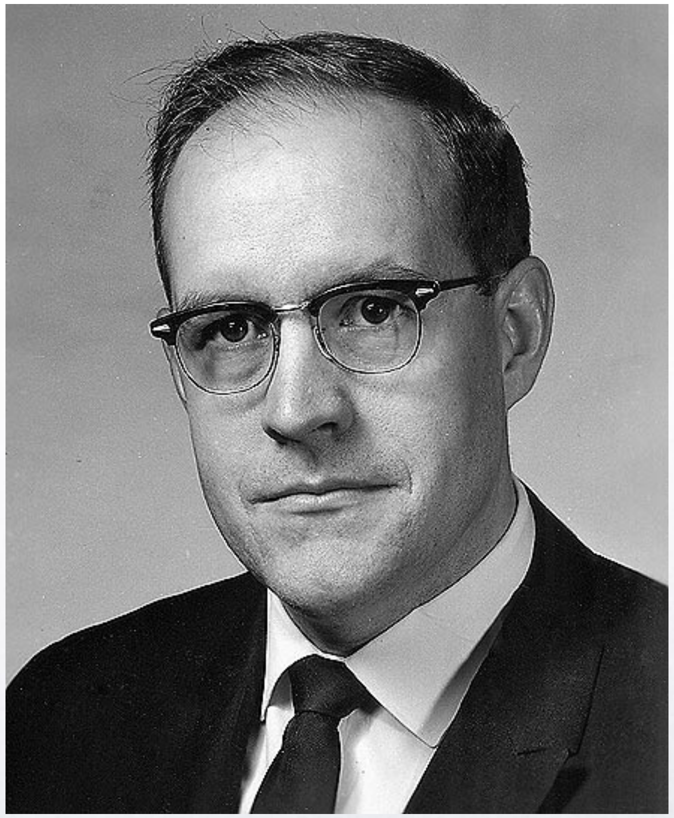






On February 19, 1985, Fred Brooks was one of three former IBM employees to receive the first National Technology Medal from U.S. President Ronald Reagan. Brooks, Erich Bloch and Bob O. Evans were recognized for their contributions to the development of the IBM System/360, which helped to revolutionize the data processing industry.

"The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements*:* No other part of the work so cripples the resulting system if done wrong. No other part is as difficult to rectify later. "



state-transition methods

No.	Attribute	Values
1	paradigm	state machine, algebra, process algebra, trace
2	formality	informal, semi-formal, formal
3	graphical representation	yes, no
4	object-oriented	yes, no
5	concurrency	yes, no
6	executability	yes, no
7	usage of variables	yes, no
8	non-determinism	yes, no
9	logic	yes, no
10	provability	yes, no
11	model checking	yes, no
12	event inhibition	yes, no

Modelagem Estado-Transição

- Estados e transições são noções distintas e intercaladas (no sentido de que estados são adjacentes a transições e vice-versa);
- Ambos, estados e transições são entidades distribuídas;
- A extensão das mudanças de causadas por uma transição é restrita ao escopo da mudança de estado (afeta portanto somente aos estados que antecedem e sucedem a transição);
- Uma transição (distribuída) t está habilitada em um estado (distribuído) s sse todas as componentes de t estiverem habilitadas e puderem ocorrer;

Sistema de Estados Finitos → **Modelagem Estado-Transição**

(State)-Transition Systems : A Logic definition

DEFINITION 13.1 (Transition System) A *transition system* is a tuple $\mathbb{S} = (S, In, T, \mathcal{X}, dom, L)$, where

(1) S is a finite non-empty set, called the set of *states* of $\mathbb{S}$.

(2) $In \subseteq S$ is a non-empty set of states, called the set of *initial states* of $\mathbb{S}$.

(3) $T \subseteq S \times S$ is a set of pairs of states, called the *transition relation* of $\mathbb{S}$.

(4) $\mathcal{X}$ is a finite set of *state variables*.

(5) dom is a mapping from $\mathcal{X}$ such that for each state variable $x \in \mathcal{X}$, $dom(x)$ is a non-empty set, called the *domain for* x.

(6) L is a function, called the *labeling function* of $\mathbb{S}$. It will be explained later.

The transition system is said to be *finite-state* if for every state variable x, the domain $dom(x)$ for this variable is finite. ❑

Automation is allays defined for a partially ordered sequence of events (or actions) we call “process”. Therefore if we are dealing with automated service design we should be aware of the processes this service generates to verify it.

Properties of systems

- safety: “the system never reaches a bad state”; in each state holds P
  - deadlock freedom
  - mutual exclusion etc.
- liveness: “there is progress in the system”; X occurs infinitely often
- fairness; once X has occurred, Y will occur in n steps
  - sent messages are eventually received
  - each request is served
- self-stabilisation: “the system recovers from a failure in a finite number of steps”

Representing time

Most properties on the previous slide can be formulated by combining two operations:

- *finally* in the future
- *globally* in the future

It must be chosen whether the present belongs to the future. Time can be described in several ways:

- global time or local time for each party
- linear or branching
- discrete or continuous

The time may be associated with the occurrences of events or with the state of the system at each moment.

Representing time: fixing the choices

- We use a discrete global time that is tied to the occurrences of events.
- The present belongs to the future.
- We mostly observe the states as a function of time, not the events.

One thing is difficult to solve: should the time be linear or branching?

LTL (linear temporal logic): the behaviour is a collection of *infinite* transition sequences

CTL (computational tree logic): the behaviour is an *infinite* transition tree

Each logic can express properties that cannot be represented in the other logic. The union of **LTL** and **CTL**, **CTL*** is even more *expressive*: it can express some properties that are beyond the power of both **CTL** and **LTL**.

Linear temporal logic LTL

- The state propositions or formulae (Φ: $p \in \Phi$ if p) map system states to truth values.
- The formulae $Fma(\Phi) \supset \Phi$ include state propositions and
  - the false proposition $\perp \in Fma(\Phi)$
  - implication: if $a \in Fma(\Phi)$ and $b \in Fma(\Phi)$ then $a \rightarrow b \in Fma(\Phi)$, and
  - the connective "globally": if $a \in Fma(\Phi)$ then $\Box a \in Fma(\Phi)$.

Other connectives can be expressed using these:

$$\neg a \Leftrightarrow a \rightarrow \perp \qquad \Diamond a \Leftrightarrow \neg \Box \neg a$$
$$a \vee b \Leftrightarrow (\neg a) \rightarrow b \qquad \top \Leftrightarrow \neg \perp$$
$$a \wedge b \Leftrightarrow \neg(a \rightarrow \neg b)$$

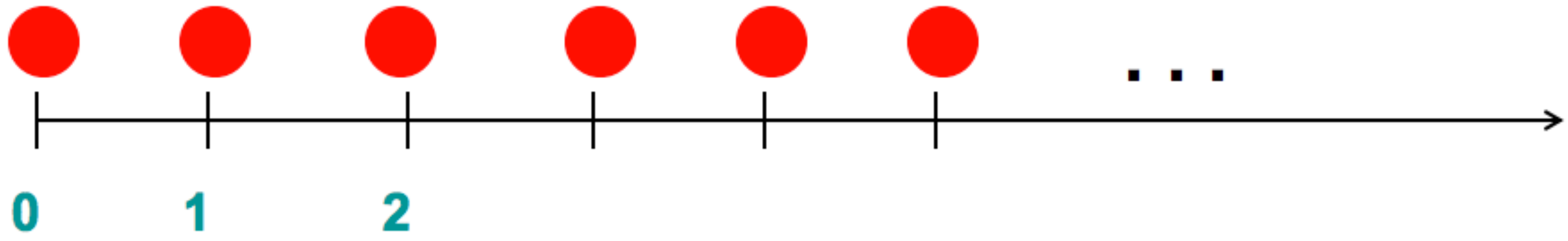
This is just one way of defining LTL and its basic connectives.

Globally (Always) p: G p

G p is true for a computation path if p holds at all states (points of time) along the path

p = 

Suppose G p holds along the path below starting at s_0

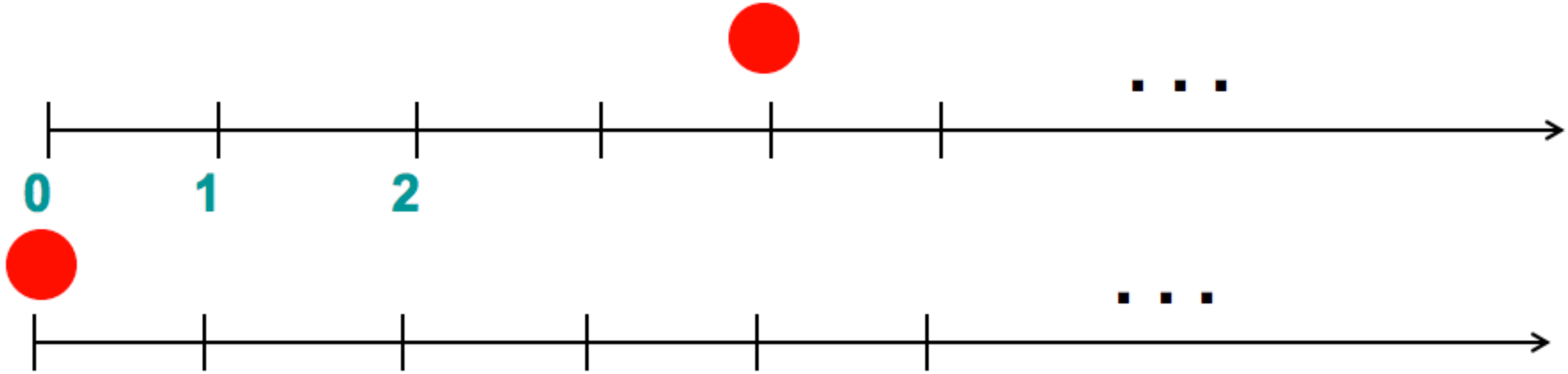


Eventually p: F p

- F p is true for a path if p holds at some state along that path

p = ●

Does F p holds for the following examples?

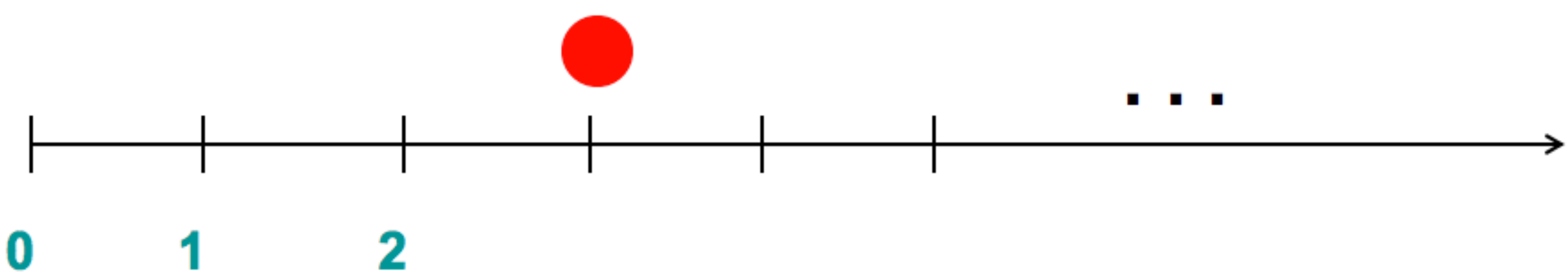


Next p: X p

- X p is true along a path starting in state s_i (suffix of the main path) if p holds in the next state s_{i+1}

p = 

Suppose X p holds along the path starting at state s_2



Notation

- Sometimes you'll see alternative notation in the literature:

  G □

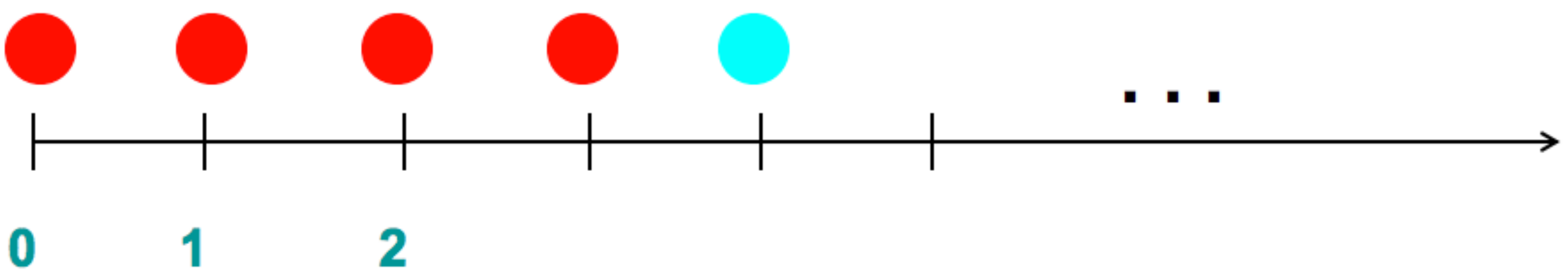
  F ◊

  X ∘

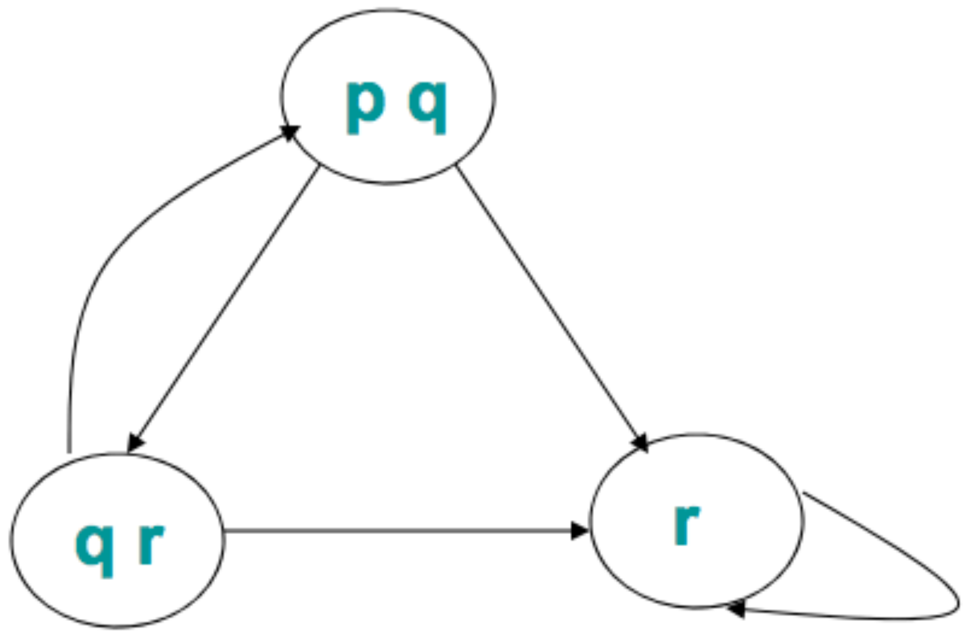
p Until q: p U q

- p U q is true along a path starting at s if
  - q is true in some state reachable from s
  - p is true in all states from s until q holds

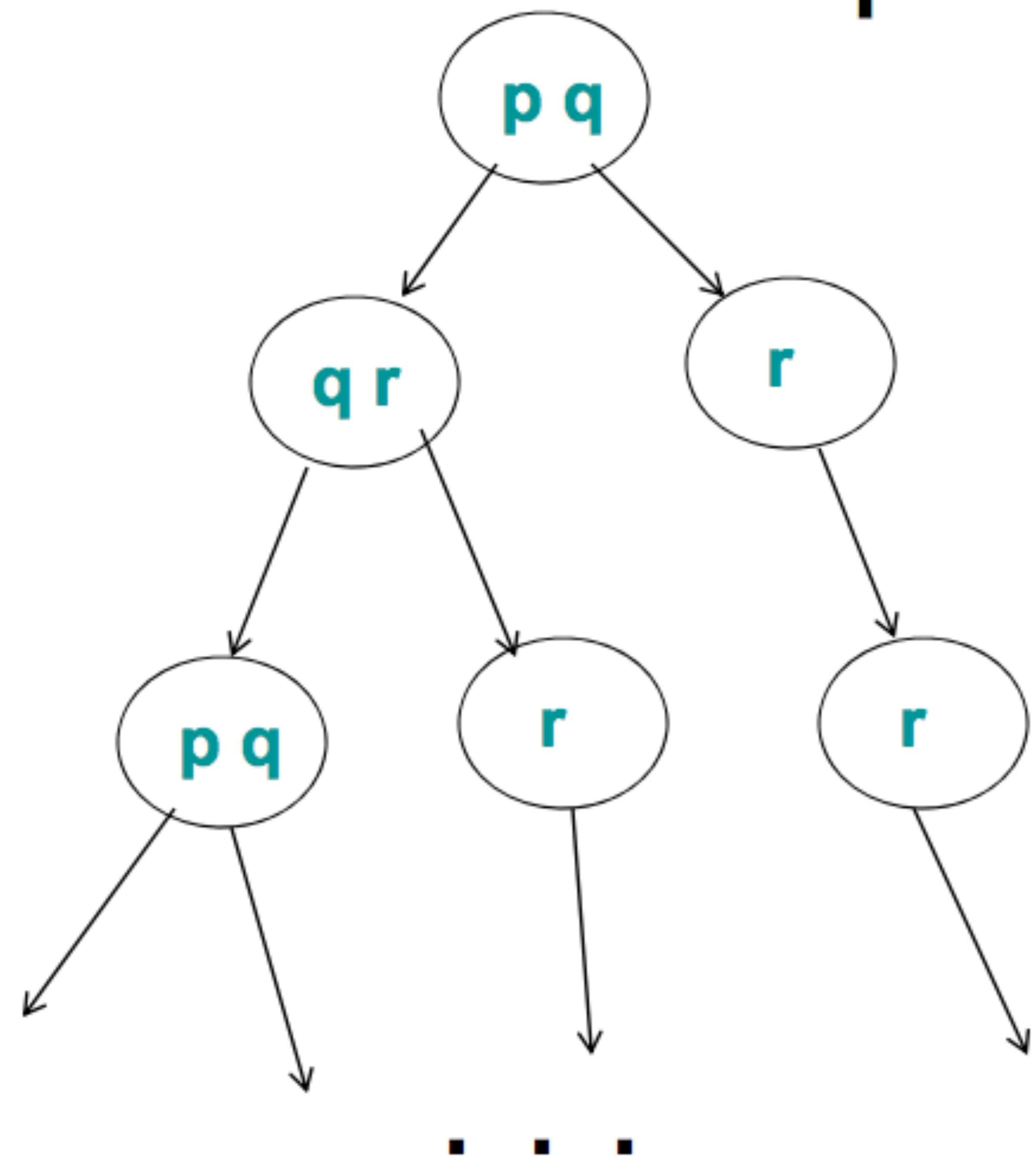
Suppose p U q holds for the path below



Labelled State Transition Graph



"Kripke structure"



Infinite Computation Tree

As a restriction let us now take a finite-state transition system where $|S|=1$, that is, where

there is one and only one initial state and each transition does not depend on the previous state-transition history. These *dynamic systems* constitutes a special class of systems very important to engineering and theoretical computer science which we can call *machines* or *automaton*.

Modelagem baseada em autômatos

Def.] Um autômato finito é definido pela n-upla (Q, Σ, Q_0, R, T) onde,

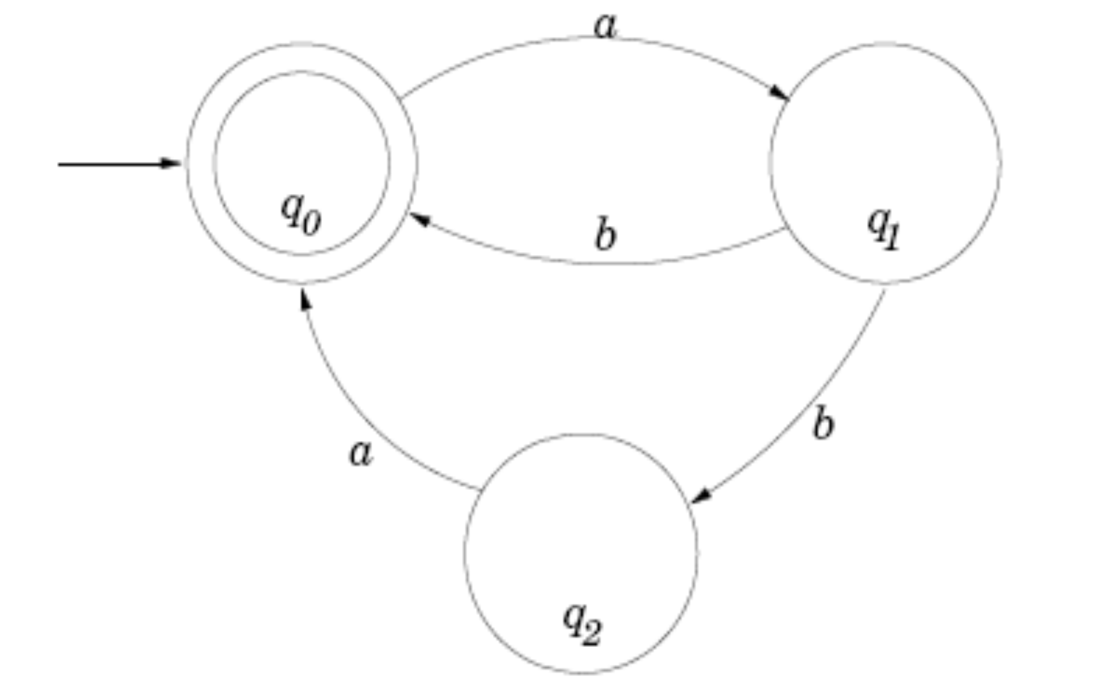
- Q é um conjunto de estados, $Q \neq \varnothing$
- Σ é um conjunto finto de letras (eventos)
- Q_0 é o estado inicial
- R é um conjunto de estados definidos como estados finais (ou de saída)
- $T : Q \times \Sigma \rightarrow Q$, é um mapeamento de $Q \times \Sigma$ em Q que denota as condições e o efeito para a ocorrência de uma transição.

O conjunto de eventos característicos de um sistema (autômato) está associado a uma linguagem formal determinada por seus processos aceitáveis.

Um exemplo simples

Example: Consider the finite automaton M in Figure 1. In this case,

– Q = {q0, q1, q2}
– § = {a, b}
– q1 = R(q0, a), q2 = R(q1, b), q0 = R(q2, a), q0 = R(q1, b) (note that R is not a function)
– Init = q0 (indicated by the straight arrow in Figure 1)
– F = q0 (indicated by a double circle in Figure 1)
– L(M) = {², ab, aba, abab, abaaba, . . .} = ((ab)¤(aba)¤)¤ µ §¤
– non-deterministic



Redes de Petri

Tese de doutorado de Carl Adam Petri sobre comunicação entre autômatos, Kommunication mit Automaten, apresentada em 1962 no Schriften des Institutes Instrumentelle Matematik, Bonn.

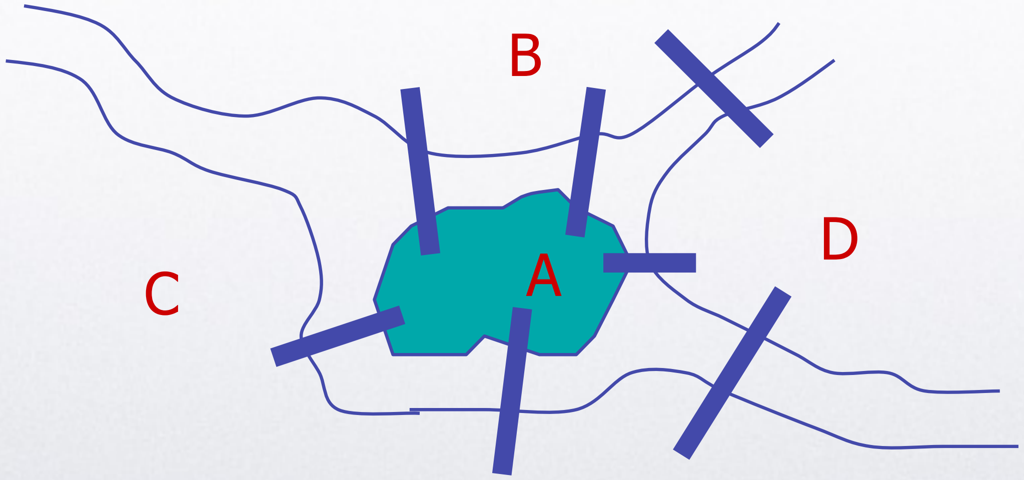
Modelagem distribuída estado-transição

Teoria de Grafos: a base formal

O primeiro artigo sobre teoria de Grafos foi apresentado por Euler, onde o problema das pontes de Konisberg foi proposto.

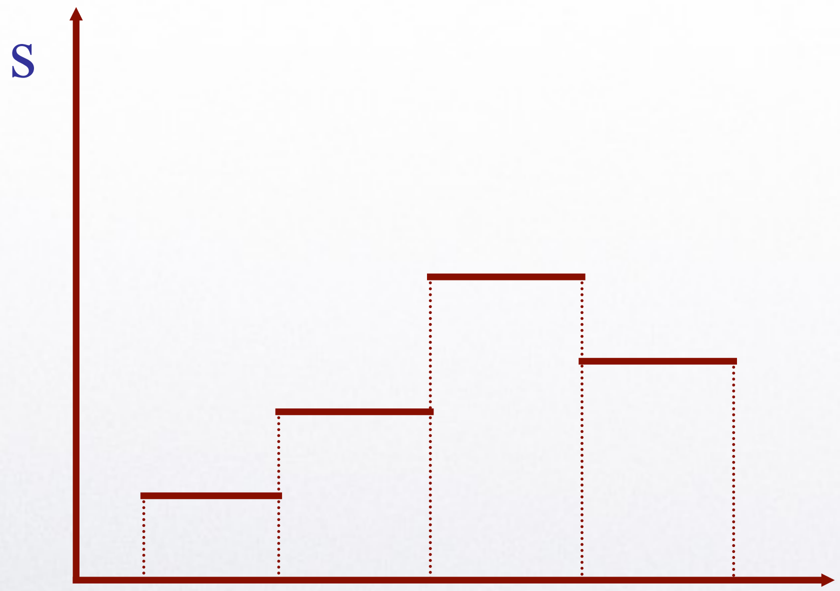
L. Euler, ´Solutio Problematis Ad geometriam Situs Pertinentis`, Commenrarii Academiae Sciencitiarum Imperialis Petropolitanae 8 (1736), pp. 128-140.

O teor do artigo consistia em mostrar que existe uma classe de problemas que pode ser formalizado por relações de adjacência e de forma independente dos aspectos geométricos.



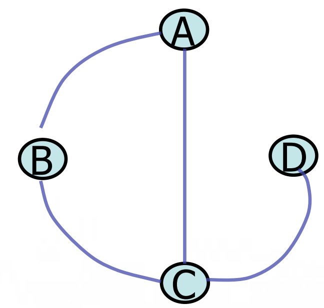
Sistemas Discretos: a Inspiração

Eventos causam uma mudança instantânea no estado



Representação de Grafos no computador

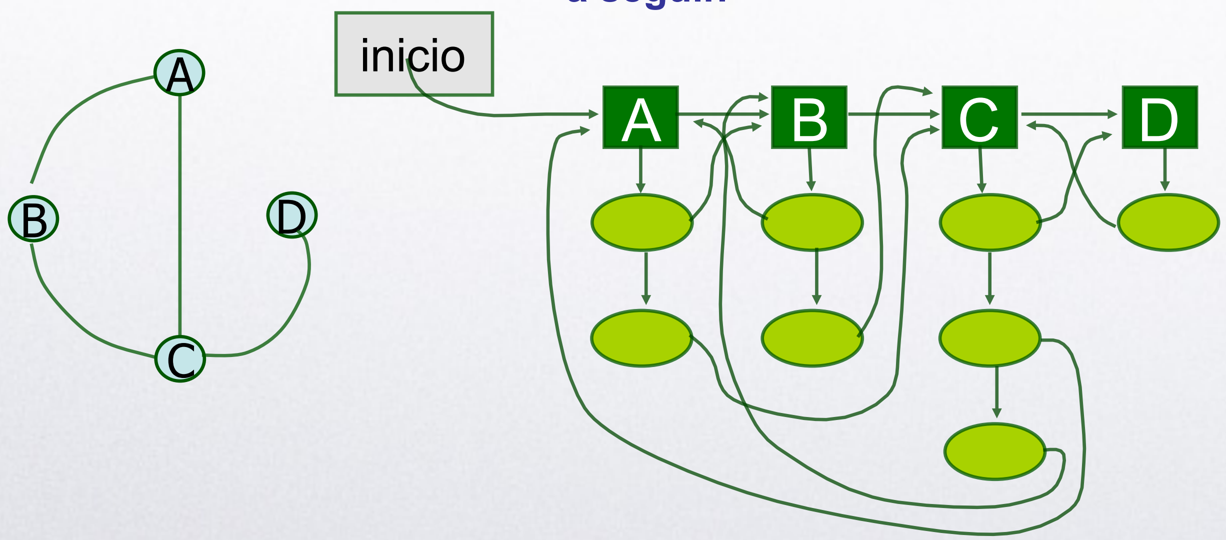
Grafos podem ser representados em duas estruturas de dados básicas: a matriz de incidência e o vetor adjacências

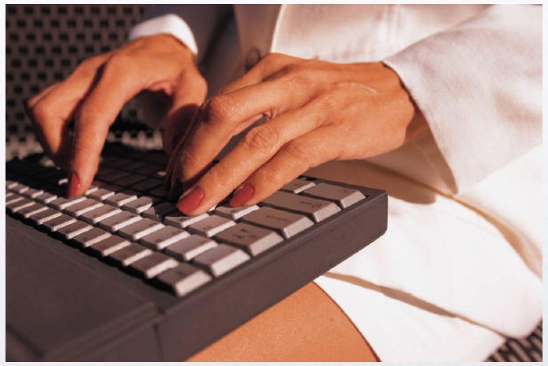


	A	B	C	D
A	0	1	1	0
B	1	0	1	0
C	1	1	0	1
D	0	0	1	0

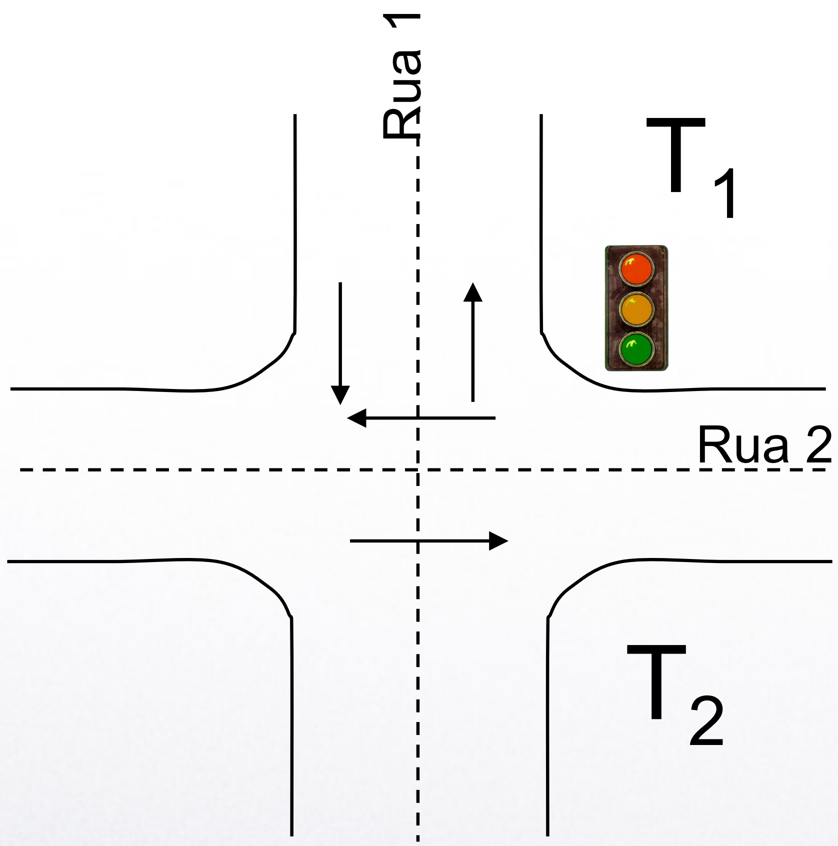
Vetor de adjacências

Uma representação pictórica desta estrutura de dados é mostrada a seguir.





Um exemplo realístico: semáforo em “*dois tempos*”



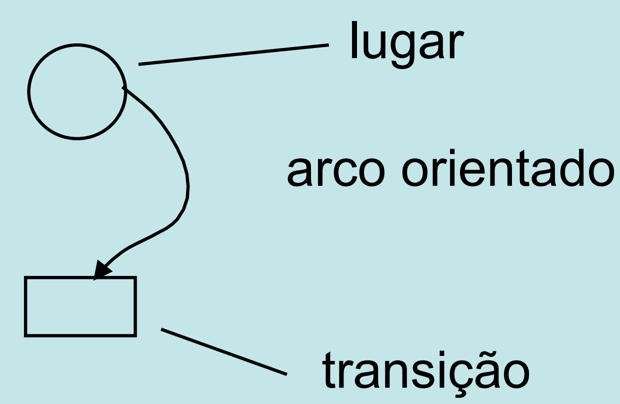
O Semáforo em dois tempos corresponde a um sistema simples com dois estados: aberto para a Rua 1 e aberto para a Rua 2. Tem ainda a restrição expressa pela seguinte expressão: ou o semáforo está aberto para a Rua 1 OU está aberto para a Rua 2

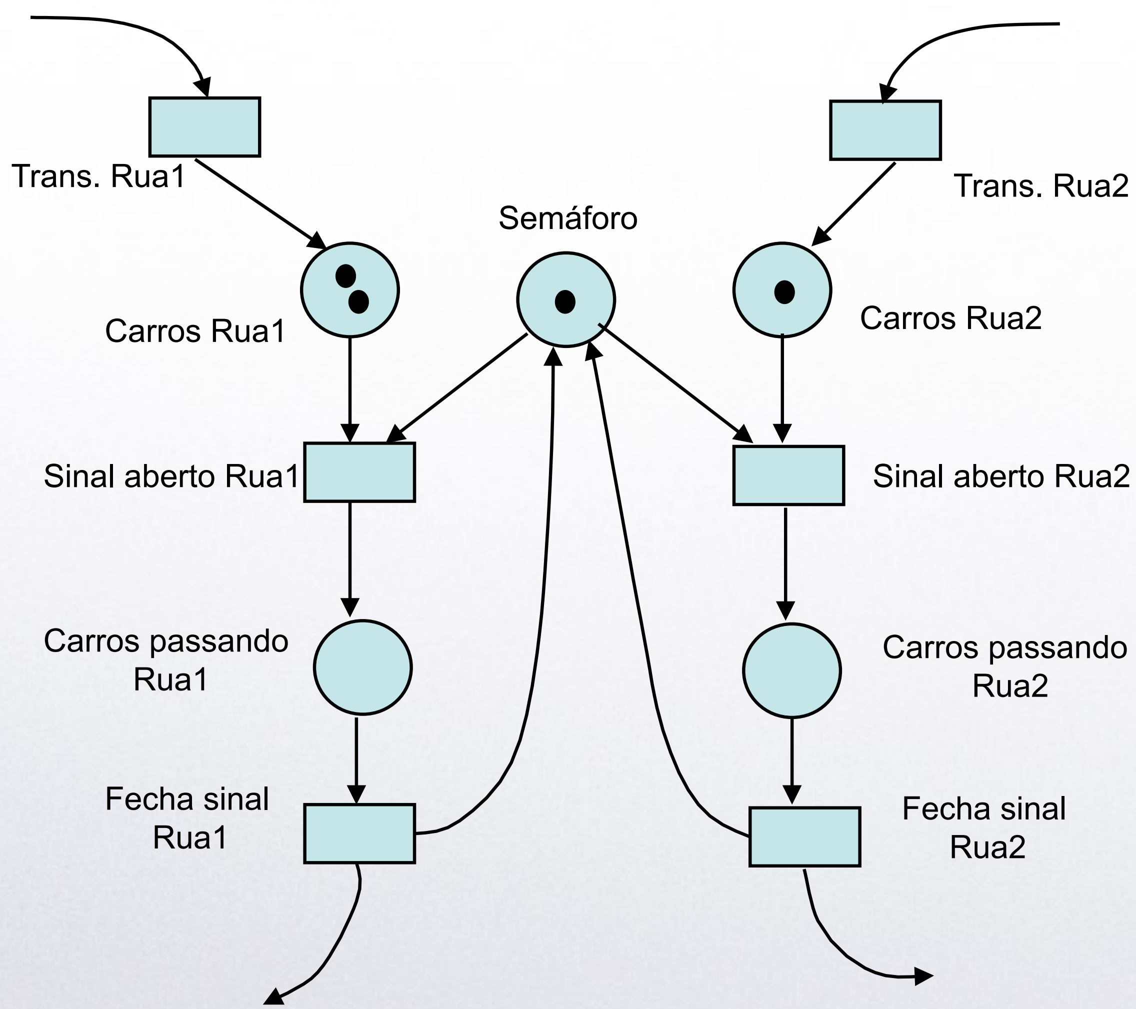
aberto R1	aberto R2	aberto R1 OU R2
V	V	F
V	F	V
F	V	V
F	F	F

Redes de Petri

As redes de Petri se tornaram uma representação formal poderosa, esquemática e genérica, com um apelo visual, para a representação de sistemas discretos em geral.

Elementos constituintes





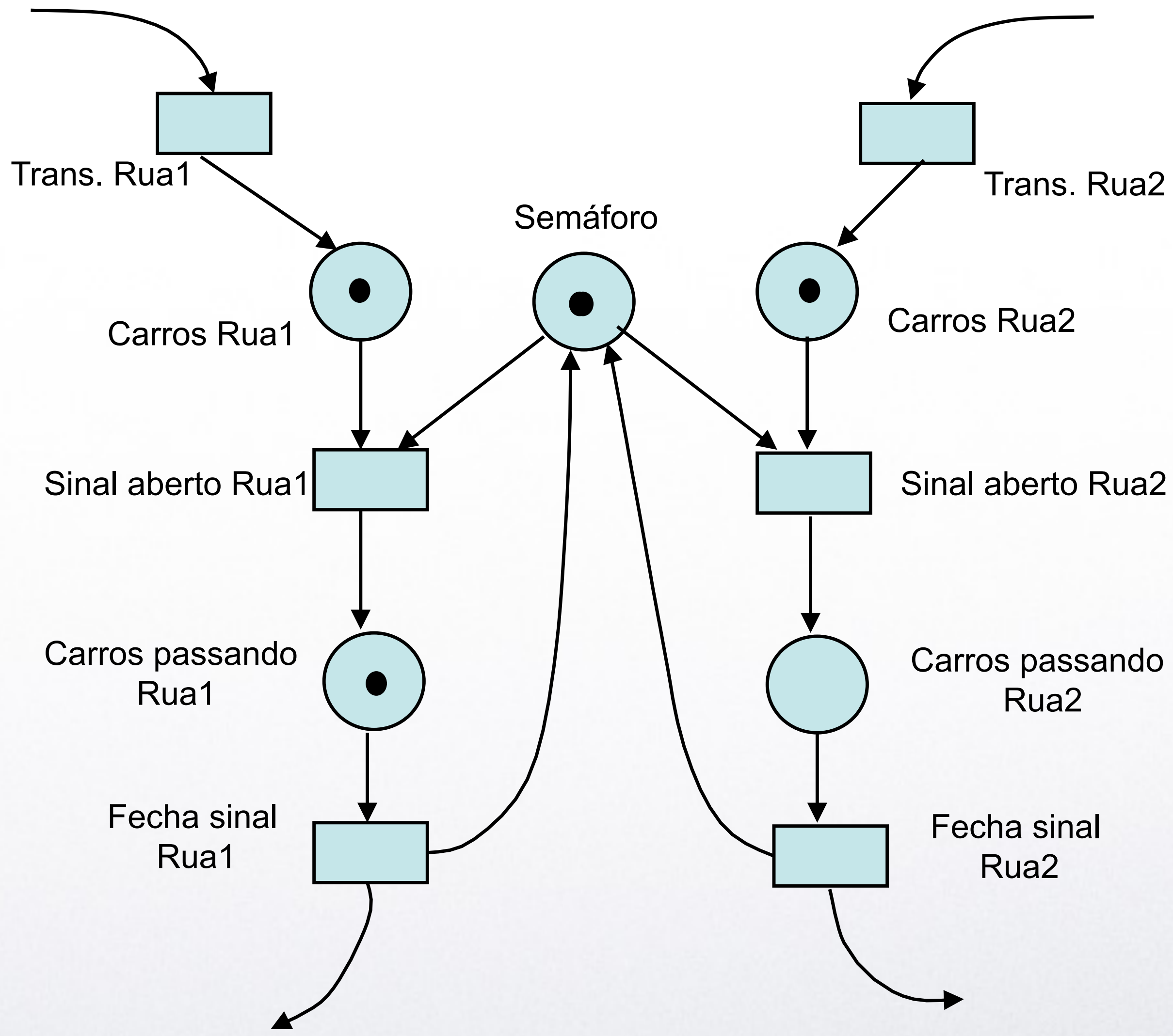
Representação de estado

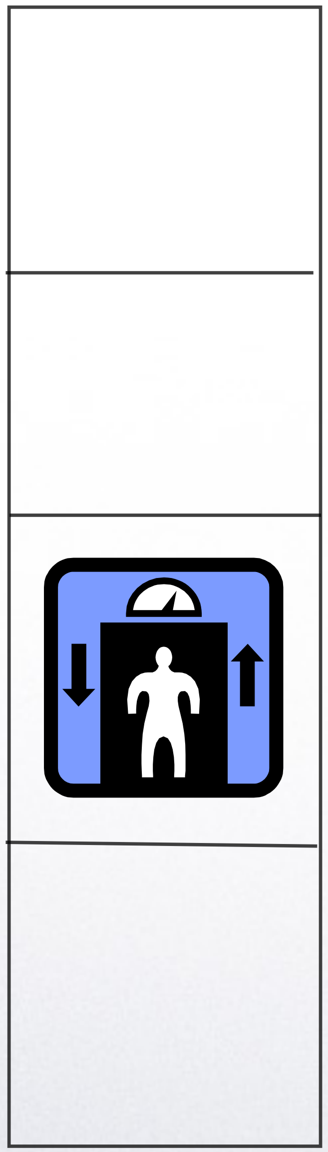
- conceito de marcação
- estado distribuído

Condição de disparo (estrita)

- pré-codições marcadas
- capacidade nas pós-condições

Um **jogador de marcas** é um sistema capaz de identificar as transições habilitadas e implementar as respectivas mudanças de estado corretamente. Isto pode ser feito de forma heurística mas, é recommendável que seja feito de maneira formal.



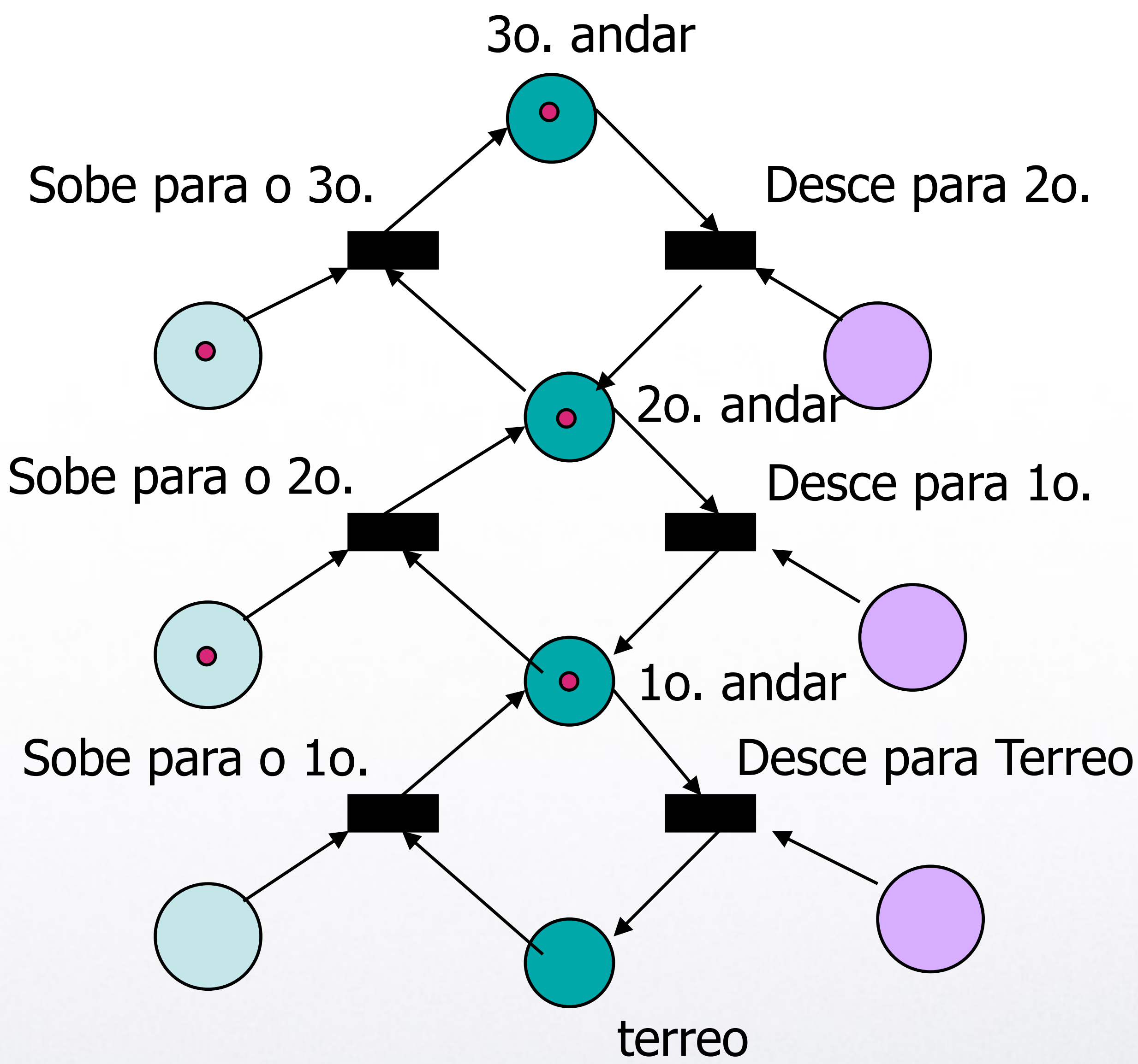


3o. andar

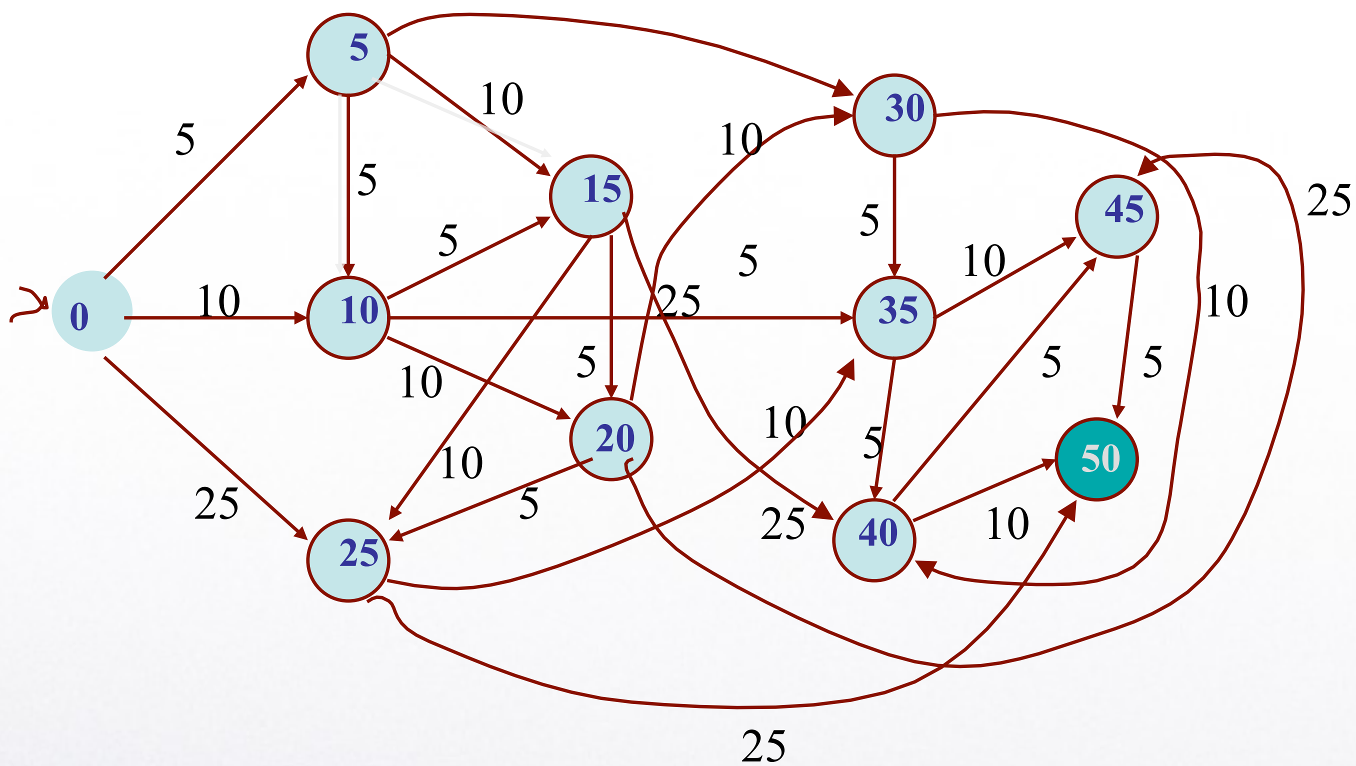
2o. andar

1o. andar

Terreo



Um outro exemplo: serving machine



Uma máquina de vender refrigerantes trabalha com moedas como estímulo (evento), onde estas podem ser de \$5, \$10 e \$25, e o refrigerante custa R\$0,50

Formalismo de Redes de Petri

Podemos então introduzir os conceitos elementares para se definir uma rede, ou melhor o que passaremos a chamar de rede elementar.

Def.] Uma rede N é um grafo bipartido, não-nulo, direcionado, representado pela n-upla (S, T; F), onde a relação de incidência F, aqui chamada de relação de fluxo é tal que $F \subseteq (S \times T) \cup (T \times S)$.

Se a rede N não possui laços, esta é dita pura. Se além disso a rede é simples, isto é, se não possui duas arestas distintas com os mesmos extremos – mesmo que estes não sejam coincidentes – então a rede é dita simples.

www.informatik.uni-gamburg.de/TGI/PetriNets/

object-orientation

No.	Attribute	Values
1	paradigm	state machine, algebra, process algebra, trace
2	formality	informal, semi-formal, formal
3	graphical representation	yes, no
4	object-oriented	yes, no
5	concurrency	yes, no
6	executability	yes, no
7	usage of variables	yes, no
8	non-determinism	yes, no
9	logic	yes, no
10	provability	yes, no
11	model checking	yes, no
12	event inhibition	yes, no

Objects

Uma coisa visível e tangível com forma relativamente estável; uma coisa que pode ser percebida intelectualmente; uma coisa para qual o pensamento ou ação pode ser direcionada.

Randon College Dictionary

Um objeto tem identidade, estado e comportamento

Grady Booch

Um objeto é uma unidade de modularidade estrutural e comportamental que tem propriedades

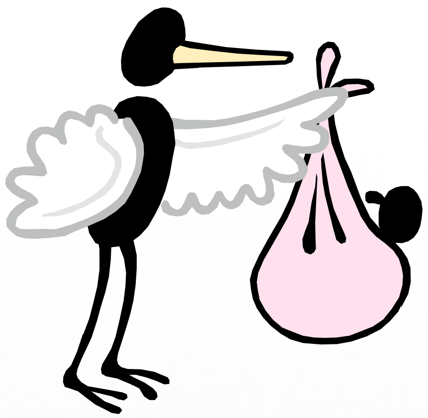
R. Buhr

Genesis dos objetos

Os objetos têm duas origens praticamente paralelas:

- estrutural : frames, Marvin Minsky, MIT, 1975
- programação : Simula 67

Objetos e programação



1966 Montagem da Simula 67: introdução do conceito de “information hiding” e encapsulamento.

1980 Aparecimento do Smalltalk 80 de Adele Goldberg

Features X Languages	Abstract Data Types	Inheritance Support	Dynamic Binding	Extensive Library
Simula	yes	yes	yes	no
CLU	yes	no	yes	no
Ada	yes	no	no	yes
Smalltalk	yes	yes	yes	yes
ObjectiveC	yes	yes	yes	yes
C++	yes	yes	yes	yes
CLOS	yes	yes	yes	no
Obj.Pascal	yes	yes	yes	no
Beta	yes	yes	yes	no
Eiffel	yes	yes	yes	yes
Actor	yes	yes	yes	no
Java	yes	yes	yes	yes

Conceitos Originais

Tipos abstratos de dados – David Parnas

Nome

Estrutura de dados da fila

Insert(x, Fila)

Retira(x, Fila)

Disciplina FIFO

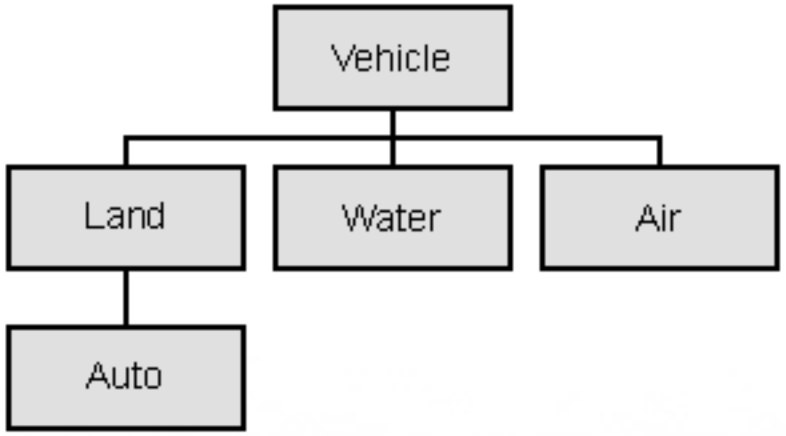
Classificação: o conceito principal

Reutilização de software

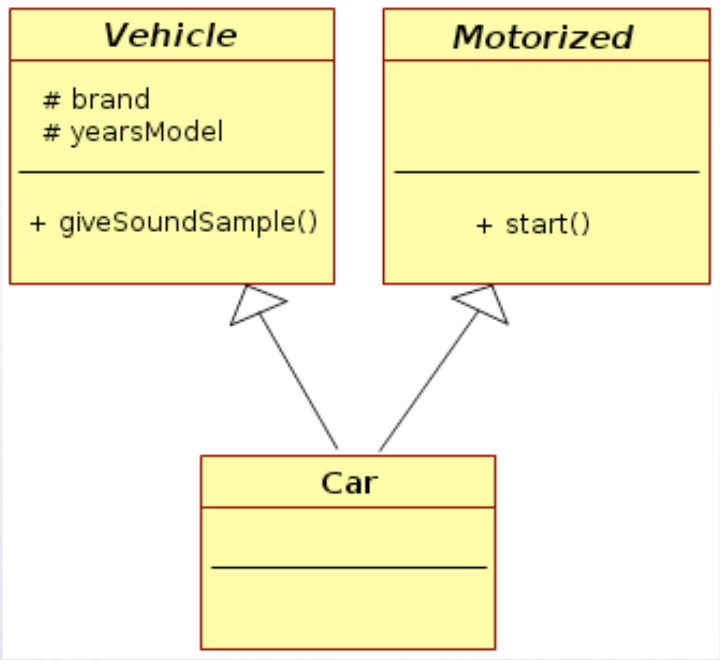
Reutilização de designs

Classificação : Charles Darwin

Espécie
Família
Grupo
Instância



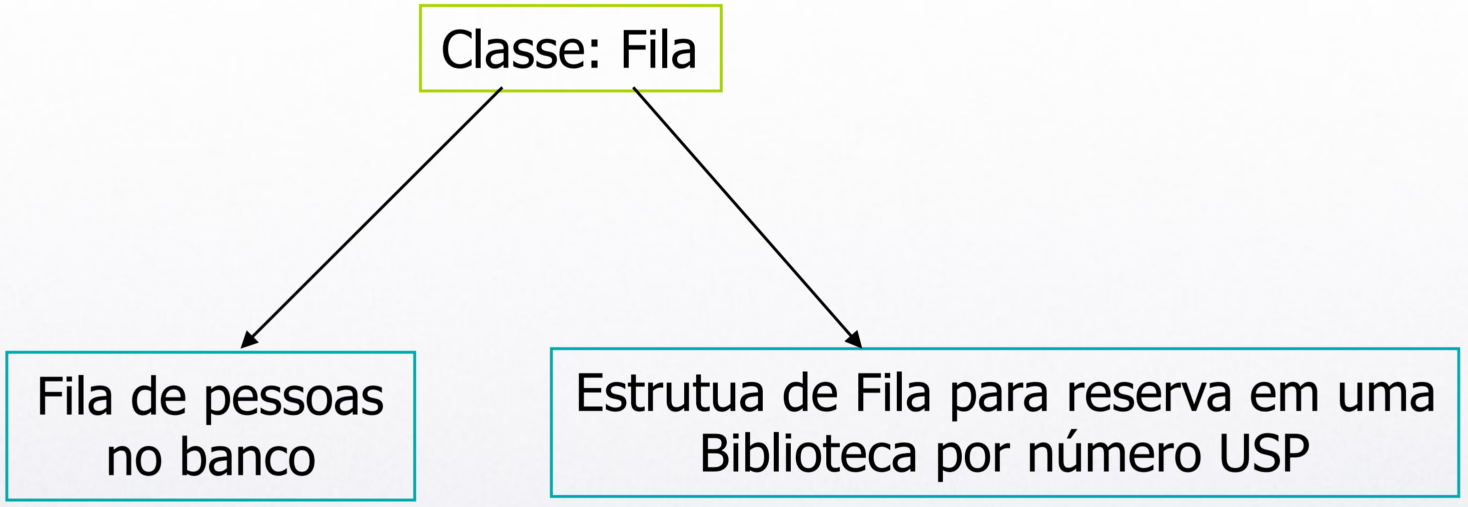
A herança simples deriva diretamente do conceito de classificação. Neste caso cada elemento ou instância de objeto tem um e somente um ancestral.



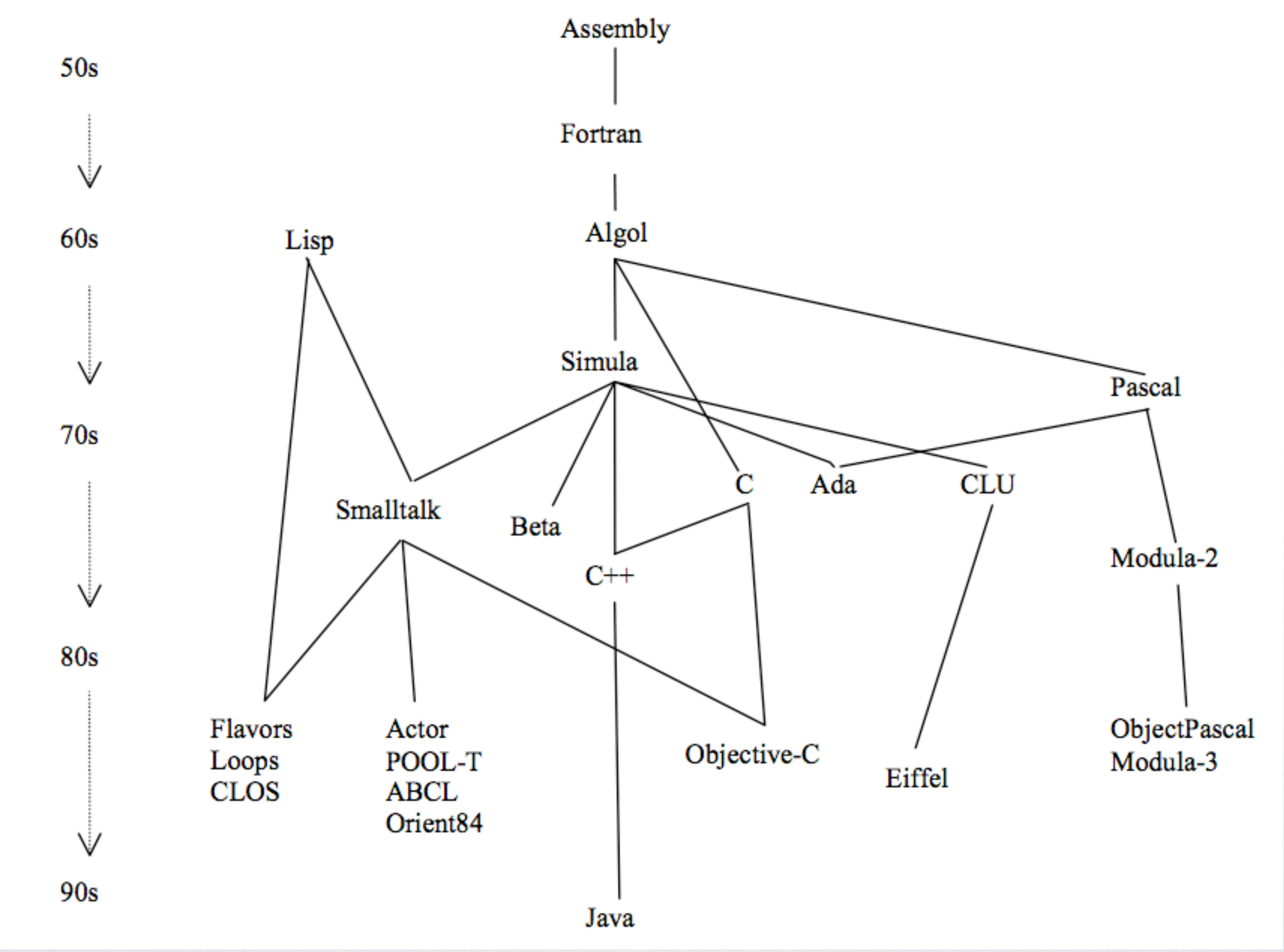
Na herança múltipla uma mesma instância pode herdar propriedades de "pais" distintos de forma composicional. Naturalmente esta implementação, mesmo em linguagens de programação é mais complexa.

Um exemplo simples

Objetos e suas propriedades : herança (simples e múltipla), polimorfismo e Vinculação dinâmica



dynamic binding - The property of object-oriented programming languages where the code executed to perform a given operation is determined at run time from the class of the operand(s) (the receiver of the message). There may be several different classes of objects which can receive a given message. An expression may denote an object which may have more than one possible class and that class can only be determined at run time. New classes may be created that can receive a particular message, without changing (or recompiling) the code which sends the message. A class may be created that can receive any set of existing messages.


Assembly
50s
Fortran
60s
Lisp
Algol
Simula
Pascal
70s
C
Ada
CLU
Smalltalk
Beta
C++
Modula-2
80s
Flavors
Loops
CLOS
Actor
POOL-T
ABCL
Orient84
Objective-C
Eiffel
ObjectPascal
Modula-3
90s
Java

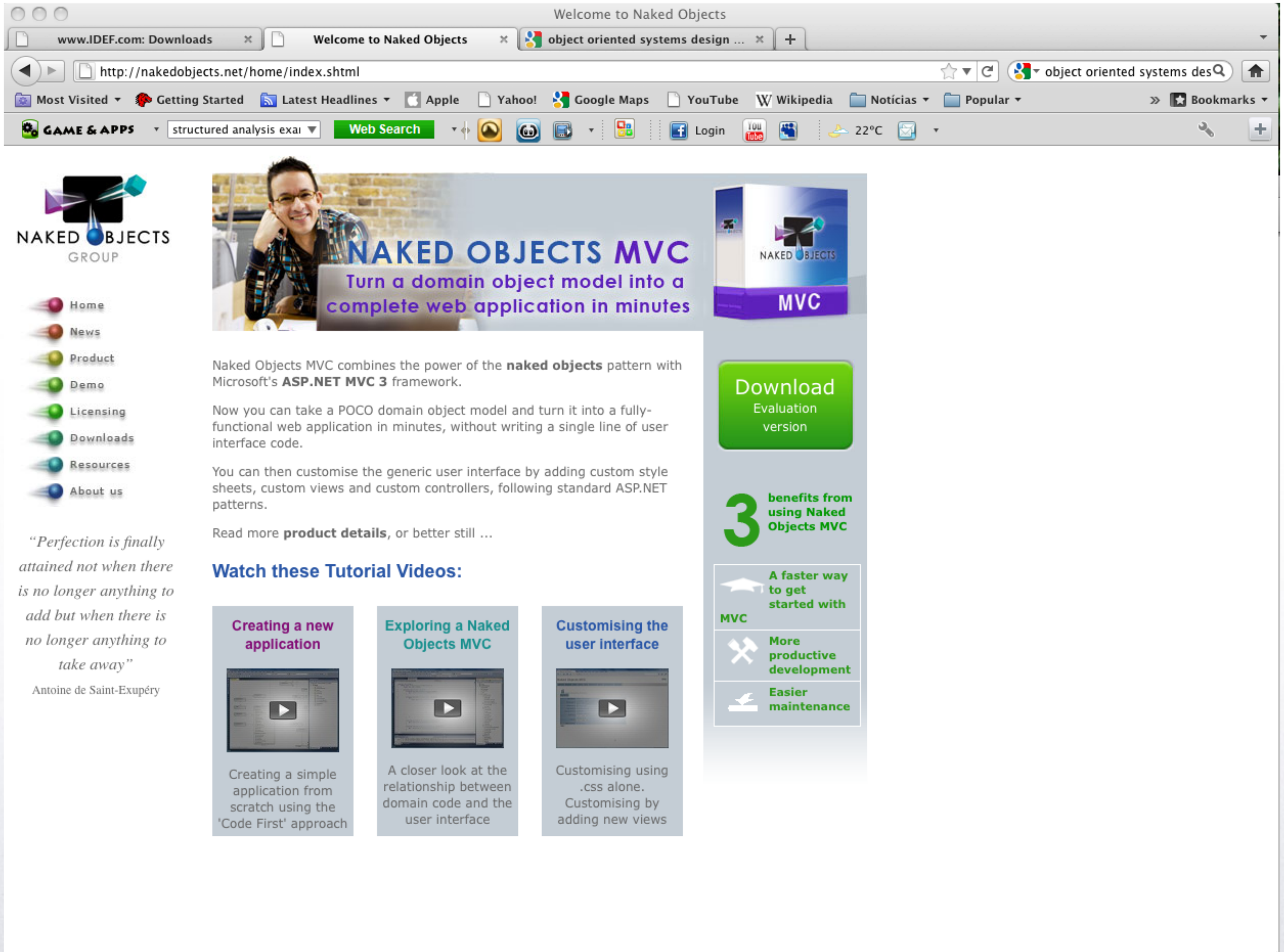
Object Oriented Design

- o sistema é composto por um conjunto de objetos
- o estado do sistema é dado pelos atributos de todas as instâncias de objeto
- uma transição no sistema se dá através de mensagens que por sua vez dispara um ou mais métodos.

A abordagem de objetos

Completeza comportamental

- separation of concerns
- encapsulation
- classification
- inheritance (single and multiple)
- polymorphism


Welcome to Naked Objects
www.IDEF.com: Downloads
Welcome to Naked Objects
object oriented systems design ...
http://nakedobjects.net/home/index.shtml
Most Visited
Getting Started
Latest Headlines
Apple
Yahoo!
Google Maps
YouTube
Wikipedia
Notícias
Popular
Bookmarks
GAME & APPS
structured analysis exa
Web Search
Login
22°C
NAKED OBJECTS
GROUP
Home
News
Product
Demo
Licensing
Downloads
Resources
About us
"Perfection is finally attained not when there is no longer anything to add but when there is no longer anything to take away"
Antoine de Saint-Exupéry
NAKED OBJECTS MVC
Turn a domain object model into a complete web application in minutes
MVC
Naked Objects MVC combines the power of the naked objects pattern with Microsoft's ASP.NET MVC 3 framework.
Now you can take a POCO domain object model and turn it into a fully-functional web application in minutes, without writing a single line of user interface code.
You can then customise the generic user interface by adding custom style sheets, custom views and custom controllers, following standard ASP.NET patterns.
Read more product details, or better still ...
Watch these Tutorial Videos:
Creating a new application
Creating a simple application from scratch using the 'Code First' approach
Exploring a Naked Objects MVC
A closer look at the relationship between domain code and the user interface
Customising the user interface
Customising using .css alone. Customising by adding new views
Download
Evaluation version
3 benefits from using Naked Objects MVC
A faster way to get started with MVC
More productive development
Easier maintenance

Behaviorally Complete Objects

– or – Back to the Roots

- An Object models the (complete) behavior of the thing it represents
- An Object
  - knows something
    - →Properties and associations
    - →Fields
  - does something
    - →Methods

Voltando ao design orientado a objetos

1. Understand and define the context and the modes of use of the system
2. Design the system architecture
3. Identify the principal objects in the system
4. Develop design models
5. Specify object interfaces

No.	Attribute	Values
1	paradigm	state machine, algebra, process algebra, trace
2	formality	informal, semi-formal, formal
3	graphical representation	yes, no
4	object-oriented	yes, no
5	concurrency	yes, no
6	executability	yes, no
7	usage of variables	yes, no
8	non-determinism	yes, no
9	logic	yes, no
10	provability	yes, no
11	model checking	yes, no
12	event inhibition	yes, no

http://www.embedded.com/design/prototyping-and-development/4024929/An-introduction-to-model-checking

An introduction to model checking

Girish Keshav Palshikar

FEBRUARY 12, 2004

Share +1 0 Tweet 0 Like 0

Model checking has proven to be a successful technology to verify requirements and design for a variety of real-time embedded and safety-critical systems. Here's how it works.

Before you even start writing code on a project, you face the chronic problem of software development: flawed design requirements. It makes sense to find flaws up front because flawed requirements breed bugs that you have to get rid of later, often at a high cost to the project and the bottom line.

In the last decade or so, the computer science research community has made tremendous progress in developing tools and techniques for verifying requirements and design. The most successful approach that's emerged is called *model checking*. When combined with strict use of a formal modeling language, you can automate your verification process fairly well. In this article, I'll introduce you to model checking and show you how it works.

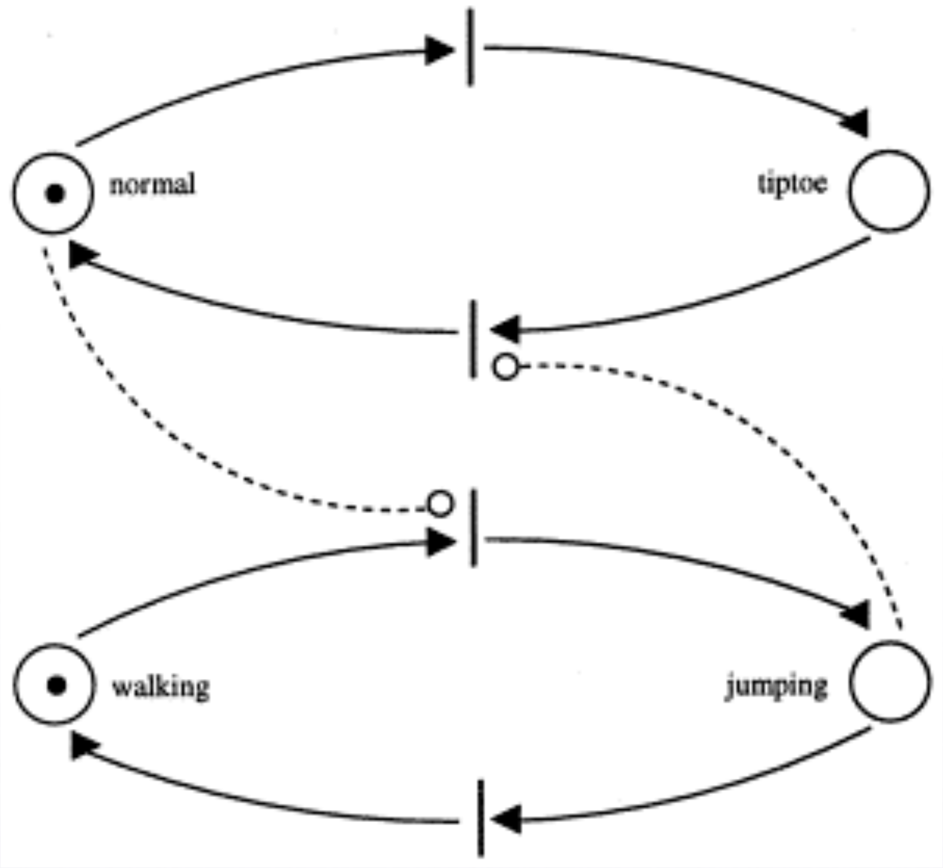
Hypothesis

- Model checking is an algorithmic approach to analysis of finite-state systems
- Model checking has been originally developed for analysis of hardware designs and communication protocols
- Model checking algorithms and tools have to be tuned to be applicable to analysis of software

Natasha Sharygina
Lecturer

No.	Attribute	Values
1	paradigm	state machine, algebra, process algebra, trace
2	formality	informal, semi-formal, formal
3	graphical representation	yes, no
4	object-oriented	yes, no
5	concurrency	yes, no
6	executability	yes, no
7	usage of variables	yes, no
8	non-determinism	yes, no
9	logic	yes, no
10	provability	yes, no
11	model checking	yes, no
12	event inhibition	yes, no

Event inhibition



No.	Attribute	Values
1	paradigm	state machine, algebra, process algebra, trace
2	formality	informal, semi-formal, formal
3	graphical representation	yes, no
4	object-oriented	yes, no
5	concurrency	yes, no
6	executability	yes, no
7	usage of variables	yes, no
8	non-determinism	yes, no
9	logic	yes, no
10	provability	yes, no
11	model checking	yes, no
12	event inhibition	yes, no

method name	paradigm	formality	graphical represen-tation	object-oriented
Action Systems	state transition	formal	no	no
B	state transition	formal	no	no
CASL	algebra	formal	no	yes
Cleanroom & JSD	traces & process algebra	formal	yes	no
COQ	state transition	formal	no	no
Estelle	state transition	formal	no	no
LOTOS	process algebra	formal	no	yes
OMT & B	state transition	formal	yes	yes
Petri Nets	state transition	formal	yes	no
Petri Nets with Objects	state transition	formal	yes	yes
SART	state transition	informal & semi-formal	yes	no
SAZ	state transition	semi-formal & formal	yes	no
SCCS	process alge-bra	formal	no	no
SDL	state transition	formal	yes	yes
UML	state transition	informal & semi-formal	yes	yes
VHDL	state transition	formal	no	no
Z	state transition	formal	no	no

method name	concurrency	executability	usage of variables	non-determinism
Action Systems	no	yes	yes	yes
B	no	yes	yes	yes
CASL	no	yes	yes	no
Cleanroom & JSD	no	yes	yes	yes
COQ	no	yes	yes	yes
Estelle	yes	yes	yes	no
LOTOS	yes	yes	yes	yes
OMT & B	no	yes	yes	yes
Petri Nets	yes	yes	no	yes
Petri Nets with Objects	yes	yes	yes	yes
SART	yes	no	no	yes
SAZ	no	yes	yes	yes
SCCS	yes	yes	yes	yes
SDL	yes	yes	no	yes
UML	yes	no	no	no
VHDL	yes	yes	yes	no
Z	no	yes	yes	yes

Leitura da Semana

IdeaGroup.pdf (page 1 of 10) — Locked

IDEA GROUP PUBLISHING # ITJ2302

701 E. Chocolate Avenue, Hershey PA 17033-1117, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

Formal Approaches to Systems Analysis Using UML: An Overview

JON WHITTLE, NASA Ames Research Center, USA

Formal methods, whereby a system is described and/or analyzed using precise mathematical techniques, is a well -established and yet, under-used approach for developing software systems. One of the reasons for this is that project deadlines often impose an unsatisfactory development strategy in which code is produced on an ad-hoc basis without proper thought about the requirements and design of the piece of software in mind. The result is a large, often poorly documented and un-modular monolith of code, which does not lend itself to formal analysis. Because of their complexity, formal methods work best when code is well structured, e.g., when they are applied at the modeling level. UML is a modeling language that is easily learned by system developers and, more importantly, an industry standard, which supports communication between the various project stakeholders. The increased popularity of UML provides a real opportunity for formal methods to be used on a daily basis within the software lifecycle. Unfortunately, the lack of preciseness of UML means that many formal techniques cannot be applied directly. If formal methods are to be given the place they deserve within UML, a more precise description of UML must be developed. This article surveys recent attempts to provide such a description, as well as techniques for analyzing UML models formally.

INTRODUCTION

The Unified Modeling Language (UML) (Object Management Group, 1999; Booch, Jacobson, and Rumbaugh, 1998) provides a collection of standard notations for modeling almost any kind of computer artifact. UML supports a highly iterative, distributed software development process, in which each stage of the software lifecycle (e.g., requirements capture/analysis, initial and detailed design) can be specified using a combination of particular UML notations. The fact that UML is an industry standard promotes communication and understanding between different project stakeholders. When used within a commercial tool (e.g., Rhapsody (I-Logix Inc, 1999), Rational Rose (Rational Software Corporation, 1999)) that supports stub code generation from models, UML can alleviate many of the traditional problems with organizing a complex software development project. Although a powerful and flexible approach, there currently exist a number of gaps in the support provided by UML and commercial tools. First and foremost, the consistency checks provided by current tools are limited to very simple syntactic checks, such deeper semantic analyses of UML models. Unfortunately, although many of these techniques already exist, having been developed under the banner of Formal Methods, they cannot be applied directly to UML. UML is, in fact, grossly imprecise. There is as yet no standard formal semantics for any part of UML, and this makes the development of semantic tool support an onerous task.

This article gives an overview of current attempts to provide an additional degree of formality to UML and also of attempts to apply existing Formal Methods analyses to UML models. Space prevents the presentation of too much detail, so the description is at a more introductory level. Our starting point is the UML definition document itself (Object Management Group, 1999) which actually includes a section on UML semantics. Unfortunately, this semantics is by no means formal but essentially provides merely a collection of rules or English text describing a subset of the necessary semantics.

To motivate the need for a formal semantics of UML, consider Figure 1, which gives a simple *sequence diagram* describing a trace in an automated teller machine (ATM). Sequence diagrams, derived in part from their close neighbor

Leitura da Semana

p6-capretz.pdf (page 1 of 10) — Locked

ACM SIGSOFT Software Engineering Notes vol 28 no 2 March 2003 Page 1

A Brief History of the Object-Oriented Approach

Luiz Fernando Capretz
University of Western Ontario
Department of Electrical & Computer Engineering
London, ON, CANADA, N6G 1H1
lcapretz@acm.org

ABSTRACT:

Unlike other fads, the object-oriented paradigm is here to stay. The road towards an object-oriented approach is described and several object-oriented programming languages are reviewed. Since the object-oriented paradigm promised to revolutionize software development, in the 1990s, demand for object-oriented software systems increased dramatically; consequently, several methodologies have been proposed to support software development based on that paradigm. Also presented are a survey and a classification scheme for object-oriented methodologies.

1. INTRODUCTION

Over the past three decades, several software development methodologies have appeared. Such methodologies address some or all phases of the software life cycle ranging from requirements to maintenance. These methodologies have often been developed in response to new ideas about how to cope with the inherent complexity of software systems. Due to the increasing popularity of object-oriented programming, in the last twenty years, research on object-oriented methodologies has become a growing field of interest.

There has also been an explosive growth in the number of software systems described as object-oriented. Object-orientation has already been applied to various areas such as programming languages, office information systems, system simulation and artificial intelligence. Some important features of present software systems include:

- Complexity: the internal architecture of current software systems is complex, often including concurrency and parallelism. Abstraction in terms of object-oriented concepts is a technique that helps to deal with complexity. Abstraction involves a selective examination of certain aspects of an application. It has the goal of isolating those aspects that are important for an understanding of the application, and also suppressing those aspects that are irrelevant. Forming abstractions of an application in terms of classes and objects is one of the fundamental tenets of the object-oriented paradigm.
- Friendliness: this is a paramount requirement for software systems in general. Iconic interfaces provide a user-friendly [...] quite naturally into the concepts of the object-oriented paradigm.
- Reusability: reusing software components already available facilitates rapid software development and promotes the production of additional components that may themselves be reused in future software developments. Taking components created by others is better than creating new ones. If a good library of reusable components exists, browsing components to identify opportunities for reuse should take precedence over writing new ones from scratch. Inheritance is an object-oriented mechanism that boosts software reusability.

The rapid development of this paradigm during the past ten years has important reasons, among which are: better modeling of real-world applications as well as the possibility of software reuse during the development of a software system. The idea of reusability within an object-oriented approach is attractive because it is not just a matter of reusing the code of a subroutine, but it also encompasses the reuse of any commonality expressed in class hierarchies. The inheritance mechanism encourages reusability within an object-oriented approach (rather than reinvention!) by permitting a class to be used in a modified form when a sub-class is derived from it [1, 2, 3, 4].

2. THE BACKGROUND OF THE OBJECT-ORIENTED APPROACH

The notion of "object" naturally plays a central role in object-oriented software systems, but this concept has not appeared in the object-oriented paradigm. In fact, it could be said that the object-oriented paradigm was not invented but actually evolved by improving already existing practices. The term "object" emerged almost independently in various branches of computer science. Some areas that influenced the object-oriented paradigm include: system simulation, operating systems, data abstraction and artificial intelligence. Appearing almost simultaneously in the early 1970s, these computer science branches cope with the complexity of software in such a way that objects represent abstract components of a software system. For instance, some notions of "object" that emerged from these research fields are:

- Classes of objects used to simulate real-world applications, in Simula [5]. In this language the execution of a computer pro-

Obrigado

Reinaldo