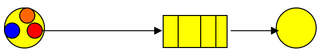
Embedded Systems 8

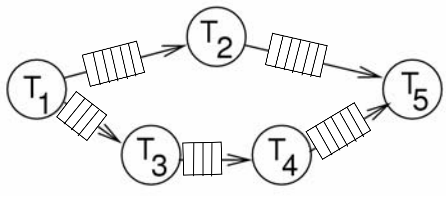
Reference model for data flow:
Kahn process networks (1974)

REVIEW

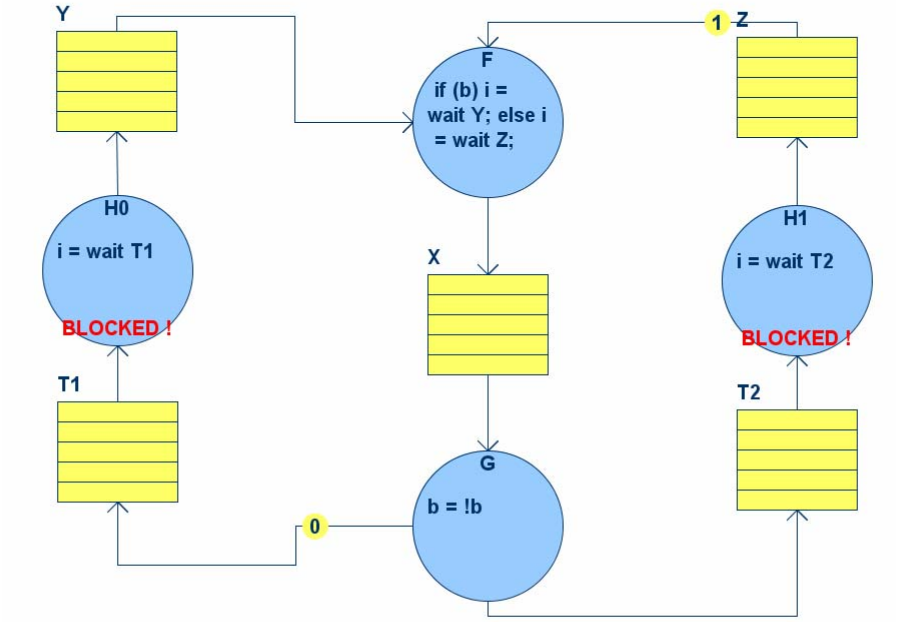
For asynchronous message passing:
communication between tasks is buffered



Special case: Kahn process networks:
executable task graphs;
Communication via infinitely large FIFOs

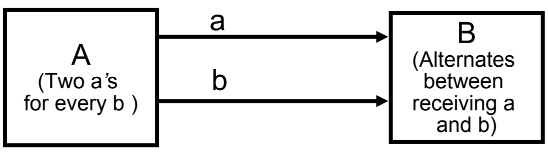


REVIEW



Scheduling may be impossible

REVIEW

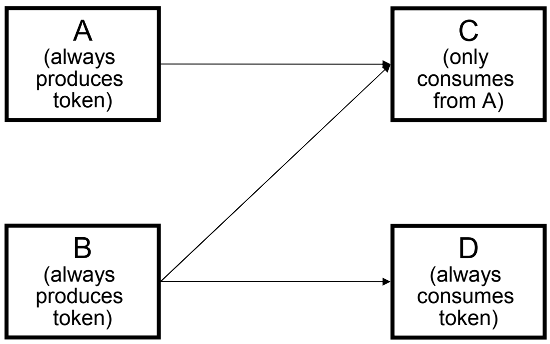


Parks' Scheduling Algorithm (1995)

REVIEW

- Set a capacity on each channel
- Block a write if the channel is full
- Repeat
  - Run until deadlock occurs
  - If there are no blocking writes → terminate
  - Among the channels that block writes,
    select the channel with least capacity
    and increase capacity until producer can fire.

Example



Parks' Scheduling Algorithm

- Whether a Kahn network can execute in bounded memory is undecidable
- Parks' algorithm does not violate this
- It will run in bounded memory if possible, and use unbounded memory if necessary

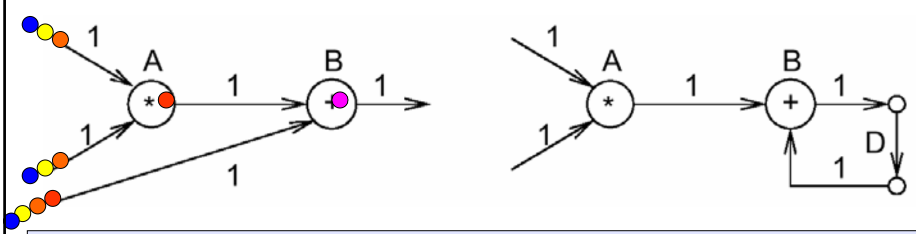
Disadvantages:

- Requires dynamic memory allocation
- Does not guarantee minimum memory usage
- Scheduling choices may affect memory usage
- Data-dependent decisions may affect memory usage
- Relatively costly scheduling technique
- Detecting deadlock may be difficult

Synchronous data flow (SDF)

REVIEW

- Asynchronous message passing=
  tasks do not have to wait until output is accepted.
- Synchronous data flow =
  all tokens are consumed at the same time.



SDF model allows static scheduling of token production and consumption.
In the general case, buffers may be needed at edges.

SDF: restriction of Kahn networks

REVIEW

An **SDF graph** is a tuple (V, E, cons, prod, d) where

- V is a set of nodes (activities)
- E is a set of edges (buffers)
- cons: E $\rightarrow$ N number of tokens consumed
- prod: E $\rightarrow$ N number of tokens produced
- d: E $\rightarrow$ N number of initial tokens

  d: „delay“ (sample offset between input and output)

SDF Scheduling Algorithm Lee/Messerschmitt 1987

REVIEW

1. Establish **relative execution rates**
   - Generate balance equations
   - Solve for smallest positive integer vector **c**
2. Determine **periodic schedule**
   - Form an arbitrarily ordered list of all nodes in the system
   - Repeat:
     - For each node in the list, schedule it if it is runnable, trying each node once
     - If each node has been scheduled $\mathbf{c}_n$ times, stop.
     - If no node can be scheduled, indicate deadlock.

Source: Lee/Messerschmitt, Synchronous Data Flow (1987)

Observations

REVIEW

- Consistent, connected systems have one-dimensional solution
- Disconnected systems have higher-dimensional solution
- Inconsistent systems have 0-schedule; otherwise infinite accumulation of tokens
- Systems may have multiple schedules

Summary dataflow

REVIEW

- Communication exclusively through FIFOs
- Kahn process networks
  - blocking read, nonblocking write
  - deterministic
  - schedulability undecidable
  - Parks‘ scheduling algorithm
- SDF
  - fixed token consumption/production
  - compile-time scheduling: balance equations

Message Sequence Charts

Communication/ local computations	Shared memory	Message passing	
		Synchronous	**Asynchronous**
Communicating finite state machines	StateCharts, StateFlow		SDL, MSCs
Data flow model $\subset$ **Computational graphs**			Kahn process networks, SDF Petri nets
Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA	
Discrete event (DE) model	VHDL, Simulink	Only experimental systems, e.g. distributed DE in Ptolemy	

Motivation: Scenario-based Specification

- "Well, the controller of my ATM can be in waiting-for-user-input mode or in connecting-to-bank-computer mode or in delivering-money-mode; in the first case, here are the possible inputs and the ATM's reactions, . . .; in the second case, here is what happens, . . ., etc.".

vs.

- "If I insert my card, and then press this button and type in my PIN, then the following shows up on the display, and by pressing this other button my account balance will show".

Motivation: Scenario-based Specification

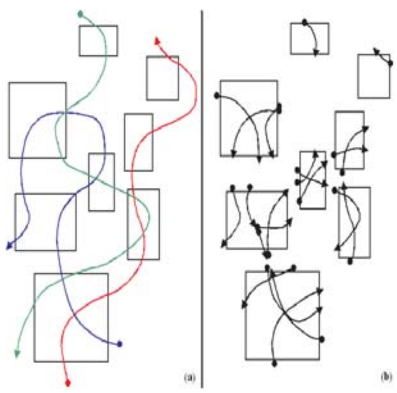
- **Claim:** it is more natural to *describe* and discuss the reactive behavior of a system by the **scenarios** it enables rather than by the state-based reactivity of each of its components.

- In order to *implement* the system, as opposed to stating its required behavior or preparing test suites, **state-based modeling** is needed, whereby we must specify for each component the complete array of possibilities for incoming events and changes and the component's reactions to them.

Inter-Object vs Intra-Object

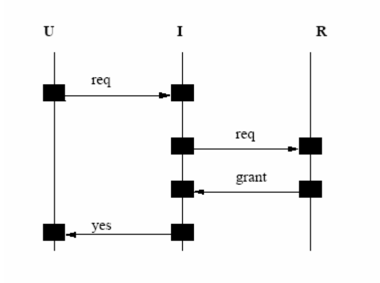
- **inter-object approach**
  'one story for all relevant objects'
  scenario-based behavioral descriptions, which cut across the boundaries of the components (or objects) of the system, in order to provide coherent and comprehensive descriptions of scenarios of behavior

- **intra-object approach**
  'all pieces of stories for one object'
  statebased behavioral descriptions, which remain within the component, or object, and are based on providing a complete description of the reactivity of each one



MSC: Example

- user (U) sends a request to an interface (I) to gain access to a resource R
- interface in turn sends a request to the resource, receives "grant" as a response
- Sends "yes" to U.



MSCs

- Graphical specification language
- Visual formalism widely used to capture system requirements in early design stages
- Describes „scenarios", patterns of interactions between processes or objects
- ITU-T Standard Z.120
- Integrated as *sequence diagrams* in UML

System Development with MSCs

- Use cases
- Specific instantiations of each use case – good scenarios
- State diagram to describe behavior for each instance
- Implementation

Live Sequence Charts (Damm/Harel 2001)

- Rather weak partial order semantics makes it impossible to capture interesting behavioral requirements.
  - “if *P* sends the message *M* to Q then *Q must* pass on this message to *R*” not possible
- LSCs are a „multimodal“ version of MSCs
  - Universal / existential
  - „Hot“ / „cold“
  - Prechart / main chart
- Executable specification

Basic MSCs

- *P* – a finite set of **processes** (agents, objects)
- *M* – a finite **message alphabet**
- *Act* – a finite set of **internal actions**

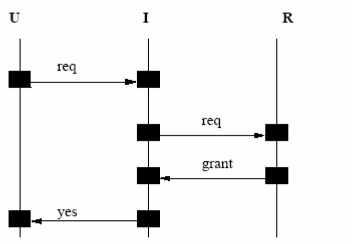
Processes in *P* communicate with each other by sending and receiving messages taken from *M* via point-to-point, reliable FIFOs.

Processes can also perform internal actions taken from *Act*, representing computational steps performed by the agents

- Set of actions Σ_p performed by a process *p*
  - *p!q(m)* – *p* sends a message *m* to *q*
  - *p?q(m)* – *p* receives a message *m* from *q*
  - *p(a)* – an internal action $a \in Act$ of *p*
- Set of actions $\Sigma := \{p!q(m), p?q(m), p(a) \mid p \neq q \; p,q \in P, m \in M, a \in Act\}$

Visual representation

- **Processes** (instances) – vertical lines or „life lines“
- time flows downwards along each life-line
- **Messages** – horizontal arrows across the life lines representing „causal links“ from a send event (the source of the arrow) to the corresponding receive event (the target of the arrow)
- label on the arrow denotes the message being **transmitted**



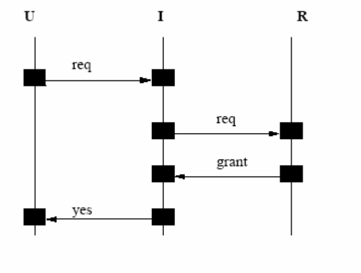
Basic MSCs

Consider the Σ-labeled poset $Ch = (E, \leq, \lambda)$ where

- E : set of **events**
- $\leq \subseteq E \times E$: **causality relation**
- $(E, \leq)$ is a **partially ordered set** (poset)
- $\lambda : E \rightarrow \Sigma$ is a **labeling function** with set of actions Σ

- $E_p = \{e \mid \lambda(e) \in \Sigma_p\}$ „events in which p takes part"
- $E_{p!q} = \{e \mid e \in E_p$ and $\lambda(e) = p!q(m)$ for some $m \in M\}$
- $E_{p?q} = \{e \mid e \in E_p$ and $\lambda(e) = p?q(m)$ for some $m \in M\}$

MSC: Example



Basic MSCs

set $\downarrow X = \{ e' \mid e' \leq e$ for some $e \in X \}$

- For a **channel** $c = (p, q)$
  we define the **communication relation** R_c such that

  $(e, e') \in R_c$ iff

  $| \downarrow(e) \cap E_{p!q} | = | \downarrow(e') \cap E_{q?p} |$
  and $\lambda(e) = p!q(m)$, $\lambda(e') = q?p(m)$ for some message m

Definition basic MSCs

An **MSC over** (P, M, Act) is a Σ-labeled poset
$Ch = (E, \leq, \lambda)$ that satisfies:

1. *All events that a process takes part in are linearly ordered; each process is a sequential agent:*

   $\leq_p$ is a linear order for each p, where $\leq_p$ is $\leq$ restricted to $E_p \times E_p$.

2. *Messages must be sent before they can be received:*

   Let $\lambda(e) = p?q(m)$, then $| \downarrow(e) \cap E_{p?q} | = | \downarrow(e) \cap E_{q!p} |$ and there exists $e' \in \downarrow(e)$ such that $\lambda(e') = q!p(m)$ and
   $| \downarrow(e) \cap E_{q!p} | = | \downarrow(e') \cap E_{q!p} |$

Definition basic MSCs

3. *There are no dangling communication edges in an MSC; all sent messages have also been received:*

   For every p,q with $p \neq q$, $| E_{p?q} | = | E_{q!p} |$

4. *Causality relation between the events in an MSC is completely determined by the order in which the events occur within each process and communication relation relating send-receive pairs:*

   $\leq \; = (\leq_P \cup R_P)^*$, where $\leq_P = \cup_{p \in P} \leq_p$ and
   $R_P = \cup_{p,q \in P, p \neq q} R_{(p,q)}$

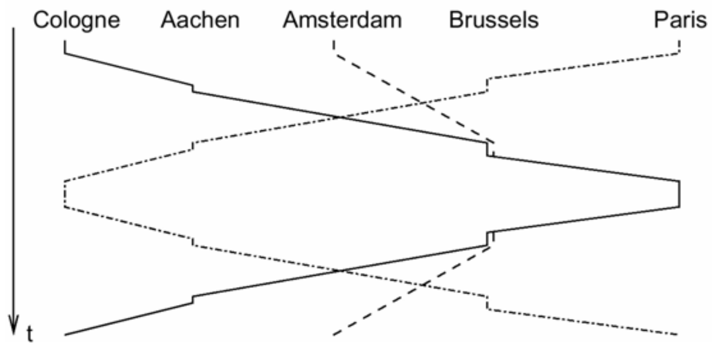
Graphical Representation

$Ch = (E, \leq, \lambda)$

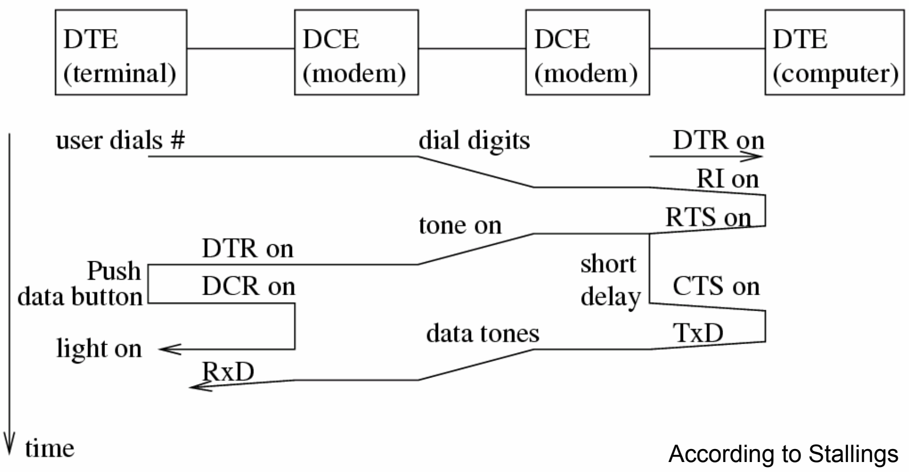
- the elements of E_p are arranged along a life-line with the earlier elements appearing above the later elements.
- A directed arrow labeled with m is drawn from $e \in E_p$ to $e' \in E_q$ provided $\lambda(e) = p!q(m)$ and $\lambda(e') = q?p(m)$ and $| \downarrow(e) \cap E_{p!q} | = | \downarrow(e') \cap E_{q?p} |$

Message sequence charts (MSC)

- Graphical means for representing schedules; time used vertically, geographical distribution horizontally.



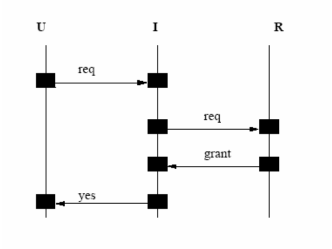
Further example: Establishing a modem connection in computer communications



According to Stallings

Language

- A **linearization** of a basic MSC is a sequence of actions $\lambda(e_0), \lambda(e_1), \ldots, \lambda(e_n)$ such that $E=\{e_0,e_1, \ldots,e_n\}$ and $e_0 \leq e_1 \leq \ldots \leq e_n$.
- Each basic MSC $Ch = (E, \leq, \lambda)$ defines a set of linearizations: $lin(Ch) \subseteq \Sigma^*$



Regular Collections of MSCs

- collection of charts constitutes the requirements that an implementation should meet.
- chart collections should be considered as requirement sets,
- e.g. to be suitable for analysis and possible detection of design errors at an early stage.

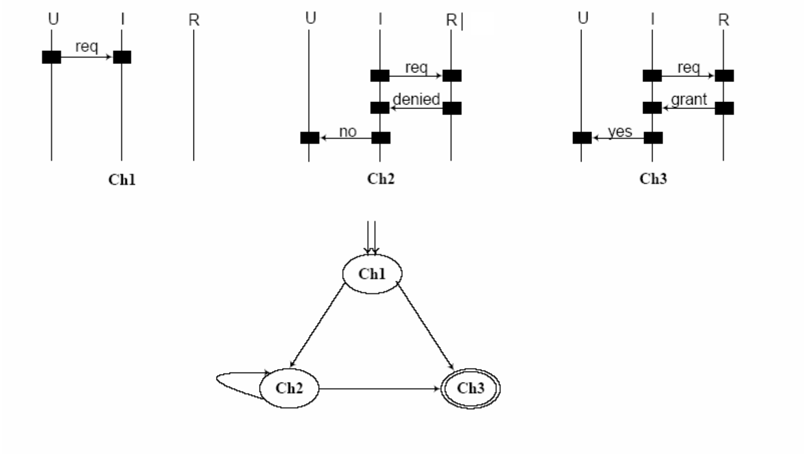
Let L be a set of MSCs.
L is a **regular collection** or **language** if $lin(L)$ is a regular subset of Σ^* where

$$lin(L) := \cup \{ lin(Ch) \mid Ch \in L \}$$

HMSCs

- HMSC is a finite state automaton whose states are labeled with MSCs over (*P, M, Act*).
  - Results in finite specifications involving choice, concatenation and iteration operations over a finite set of seed MSCs.
- [in general, specification can be hierarchical, i.e. a state of the automaton can be labeled by an HMSC instead of an MSC.
  Here:
  flattened HMSCs, *message sequence graphs* (MSGs).]

Example (1)



Synchronous Concatenation

- **edges** in an MSG represent chart **concatenation:** collection of charts represented by an MSG consists of all those charts obtained by tracing a path in the MSG from an initial control state to a terminal control state and concatenating the MSCs that are encountered along the path.
- **Synchronous concatenation** *Ch:Ch´* means that *all* the events in *Ch* must finish before *any* event in *Ch´* can occur.
- Synchronous composition requires a **protocol** for all life-lines to synchronize.

Asynchronous Concatenation

Asynchronous concatenation $Ch1 \circ Ch2$ is carried out at the level of life-lines.

Let $Ch_1 = (E_1, \leq_1, \lambda_1)$ and $Ch_2 = (E_2, \leq_2, \lambda_2)$ be a pair of MSCs. Assume that E_1 and E_2 are disjoint sets.

Then $Ch_1 \circ Ch_2$ is the MSC $Ch = (E, \leq, \lambda)$, where:

- $E = E_1 \cup E_2$
- $\lambda(e) = \lambda_1(e)$ ($\lambda_2(e)$) if e is in E_1 (E_2).
- $\leq$ is the least partial ordering relation over E that contains $\leq_1$ and $\leq_2$ and satisfies: If $e \in E_{1_p}$ and $e' \in E_{2_p}$ for some p, then $e \leq e'$

Properties

- asynchronous concatenation of two charts is also a chart.
- synchronous concatenation of two charts
  may not result in a chart.

Properties

- Asynchronous concatenation may lead to non-regular languages

Undecidability

Theorem: The intersection of two MSGs (with asynchronous concatenation) is undecidable.

Communication-boundedness

Communication-boundedness is a sufficient condition for regularity.

- The **communication graph** of a basic MSC
is a directed graph,
where the nodes are the processes,
edge $p \rightarrow q$ if $p!q(m)$ for some m in chart.
- MSC is **communication-bounded** iff communication graph consists of a single strongly-connected component (+ isolated nodes)
- MSG is **communication-bounded** iff communication graph of all loops is communication-bounded.

Life Sequence Charts* (LSCs)

Duality of possible and necessary, existential and universal

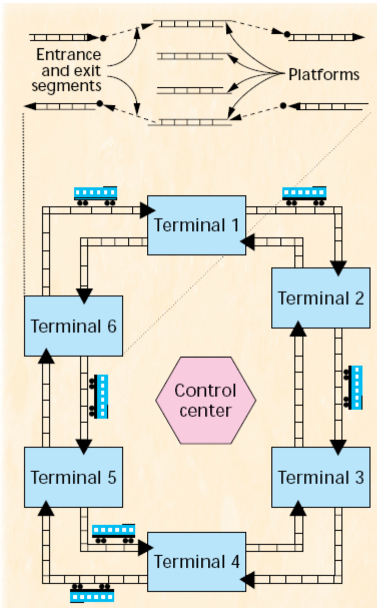
- done both on the level of an entire chart and on the level of its elements.

Charts - two types of charts, *universal* and *existential*.

- existential charts specify sample interactions — typically between the system components and the environment that at least one system run must satisfy
- universal chart typically contains a *prechart* followed by a main chart, to capture the requirement that if along any run the scenario depicted in the prechart occurs, the system *must* also execute the main chart. .

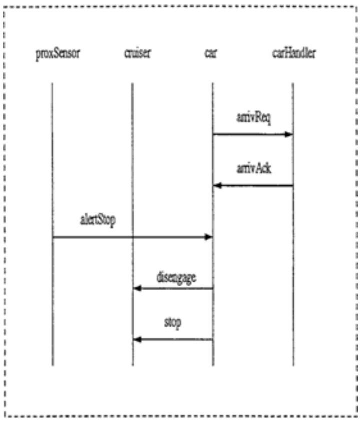
* [DH] W. Damm, D. Harel: LSCs: Breathing Life into Message Sequence Charts, *Formal Methods in System Design*, 19, 45–80, 2001

Automated Railcar System

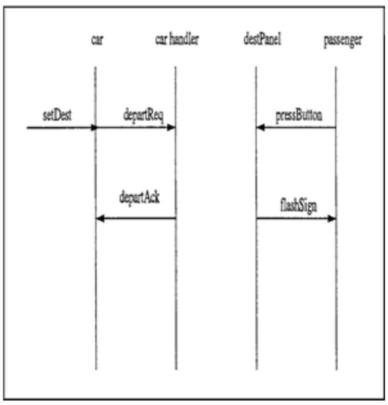


Example in
„Executable Object Modeling with Statecharts“ by Harel/Gery, 1997

Existential Chart



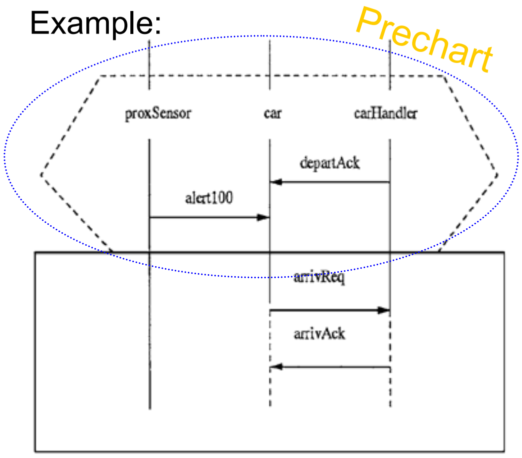
Universal Chart



Universal LSC with Prechart

Precharts describe conditions that must hold for the main chart to apply.

Example:



Basic universal LSC with a prechart

A *basic universal LSC* (over (P,M,Act)) *with a prechart* is a Structure $S = (E, \leq, Pch, \lambda)$ where

1. $Pch = (E_{Pch}, \leq_{Pch}, \lambda_{Pch})$ is a chart with $E_{Pch} \cap E = \varnothing$.
2. $(E, \leq, \lambda)$ is a labeled partial order with $\lambda : E \rightarrow \Sigma \cup \{Pch\}$, where Σ is as defined for MSCs w.r.t. (P,M,Act).
3. There is a **unique event** e_0 which is the least under $\leq$ and $\lambda^{-1}(Pch) = \{e_0\}$
4. $Ch = (E', \leq', \lambda')$ is a chart called the **main chart**, where $E' = E - \{e_0\}$, $\leq'$ is $\leq$ restricted to $E' x E'$, and λ' is λ restricted to E'.

Basic universal LSC with a prechart

- *Pch* is the prechart serving as the guard of the main chart. Prechart is specified as a refinement of the least event e_0, to capture the idea that the prechart must execute before the main chart can begin.
- The semantics of this basic LSC is that in any execution of the system, whenever *Pch* is executed, it must be followed by an execution of the main chart.

Basic universal LSC with a precondition (1)

- Conditions are predicates concerning the local states of the processes.
- Here: predicates are propositional formulas constructed from boolean assertions about the local states.
- Consider a family of pairwise-disjoint sets of atomic propositions $\{AP_p\}_{p \in P}$, and set $AP = \cup_{p \in P} AP_p$
- Suppose φ is a propositional formula built out of *AP*. Then *loc* (φ) is the set of processes whose atomic propositions appear in φ.

Basic universal LSC with a precondition (2)

A *basic universal LSC* (over (*P,M,Act*)) *with a precondition* is a structure $S = (E, \leq, \varphi, \lambda)$ where

1. φ is a Boolean formula over AP
2. $(E, \leq, \lambda)$ is a labeled partial order with $\lambda : E \rightarrow \Sigma \cup \{\varphi\}$, where Σ is as defined for MSCs w.r.t. (*P,M,Act*).
3. There is a unique event e_0 which is the least under $\leq$ and $\lambda^{-1}(\varphi) = \{e_0\}$
4. $Ch = (E', \leq', \lambda')$ is a chart called the *main chart*, where $E' = E - \{e_0\}$ and $\leq'$ is $\leq$ restricted to $E'xE'$ and λ' is λ restricted E'.

Basic universal LSC with a precondition (3)

- idea is that all the processes in *loc*(φ) first synchronize and φ is evaluated. If true, the main chart is executed and if false it is skipped.
- The semantics of this basic LSC is that, along any execution, if φ holds it must be followed by an execution of the main chart, otherwise there is no constraint.

- Basic existential LSCs,
  basic LSCs with post-conditions …
  can be defined in a similar way

Life Sequence Charts (LSCs)

duality of possible and necessary, existential and universal

- done both on the level of an entire chart and on the level of its elements.

Charts - two types of charts, *universal* and *existential*.

Conditions -

Conditions

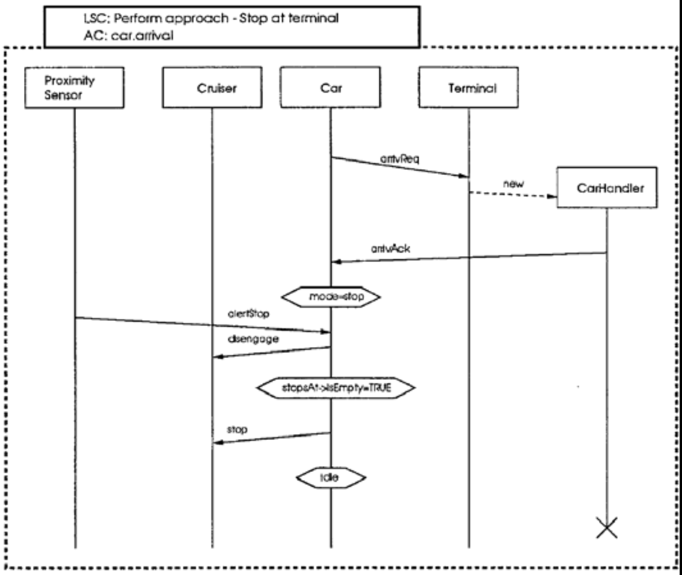
- *cold* and *hot* conditions, which are provisional and mandatory guards
  - If a cold condition holds during an execution, control passes to the location immediately after the cold condition
  - if it is false, the chart context in which this condition occurs is exited and execution may continue.
  - A hot condition must always be true. If an execution reaches a hot condition that evaluates to false this is a violation of the requirements, and the system should abort.
  - For example, if we form an LSC from a prechart *Ch* and a main chart consisting of a single *false* hot condition, the semantics is that *Ch* can never occur. In other words, it is forbidden, an *anti-scenario*.

Mandatory (hot) vs. provisional (cold) behavior

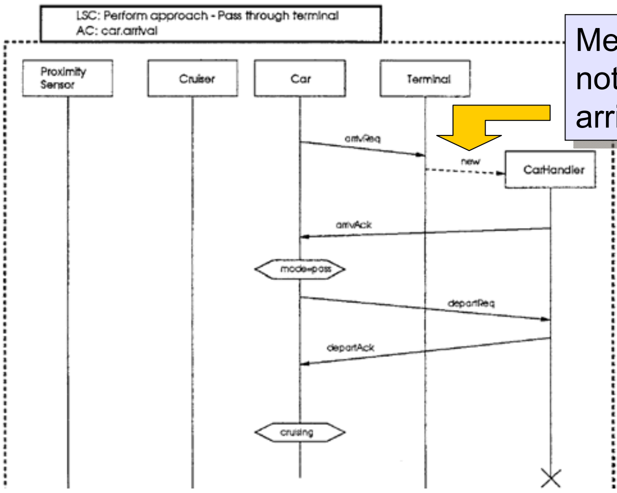
Level	Mandatory (solid lines)	Provisional (dashed lines)
Chart	All runs of the system satisfy the chart	At least one run of the system satisfies the chart
Location	Instance must move beyond location/time	Instance run need not move beyond loc/time
Message	If message is sent, it will be received	Receipt of message is not guaranteed
Condition	Condition must be met; otherwise abort	If condition is not met, exit subchart

Provisional charts: Behavior may be this one:

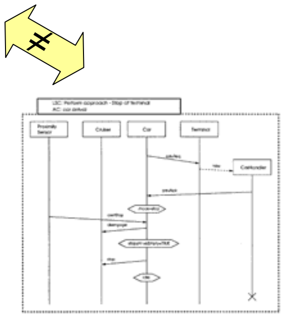
- Dashed charts



… or this one:

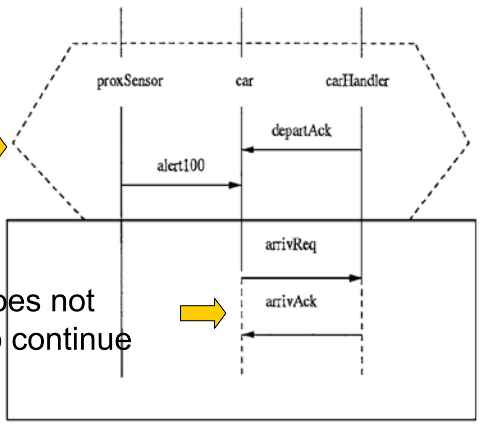


Message does not need to arrive



Locations: Mandatory/provisional

Don’t enter main chart, if condition is not met.



“run” does not need to continue

Composition of LSCs

- compose basic LSCs to construct more complex LSCs.
- asynchronously concatenate LSCs.

Composition of LSCs