Where are we?

- **SDL and MSC**
  - Notation
  - Behavioral Properties
  - Analysis methods
- **Petri Nets**
  - Notation
  - Behavioral Properties
- **Symbolic Analysis methods of finite models**
- **Timed automata**
  - Notation
  - Semantics
  - Analysis
- **Introduction to model checking ?**

Specification and Description Language

- SDL provides a graphical *Graphic Representation* (SDL/GR) as well as a textual *Phrase Representation* (SDL/PR) (same semantics!).
- ITU-T (Int. Telecom. Union - Telecom. Standardization Sector) Recommendation Z.100, released 1992 (corrections introduced 1996).
- Found acceptance in industry, now revisions are mainly initiated by tool builders

  => Lost a little of its clean formal basis,
  i.e. part of the semantics is tool dependent these days.
- Area of application:
  - Focus on telecommunication systems, but nowadays
  - process control systems, automotive applications, etc.
  - in general highly suited for (distributed real-time systems)

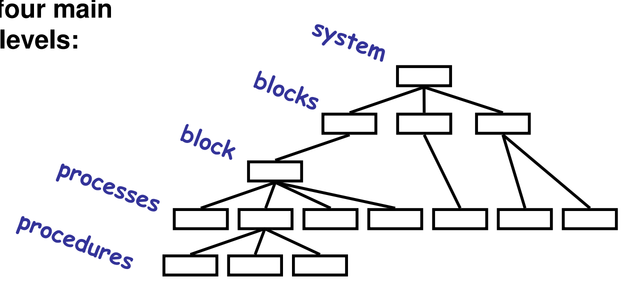
SDL history (brief sketch)

- First release in 1976 using graphical syntax (orange book). The textual form was introduced later for machine processing.
- The yellow book defined the process semantics
- The red book as released in 1984, it defined a SDL system's structure, added data, etc.
- The blue book released 1988 (SDL-88) gave a formal definition.
- SDL-92 contained concepts of OO: inheritance, abstract generic types for blocks processes, services parameterization of instances.
- SDL-2000 is the latest released version completely based on object-orientation. This version is accompanied by an SDL-UML-Profile.

Overview (1): Structure of an SDL specification

SDL comprises four main hierarchical levels:

1. **system**
2. **blocks**
3. **processes**
4. **procedures**



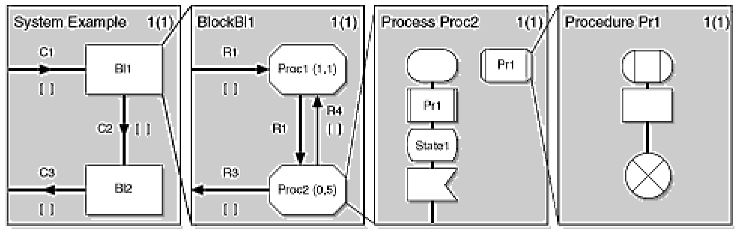
Informally: See it as tree structure

✓ the root is the system,

✓ the leaves are processes,

✓ everything in the middle is a block.

Overview (2): Structure of an SDL specification

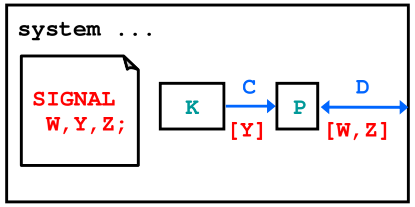


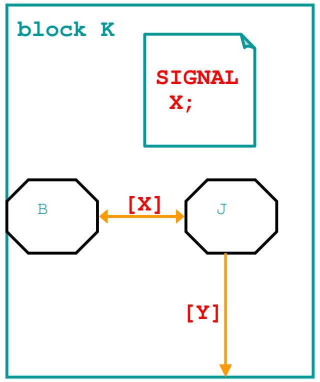
IEC Web tutorial (http://www.iec.org/online/tutorials/sdl/topic02.html)

- Processes are the entities which define actual behavior
- Procedures can be viewed as some mechanism for encapsulating (sub-)behavior to be used multiple times (~function/method).
- Some tools allow also to call external functions, implemented in C++ or Java

Overview (3): Signaling within SDL (a glance)

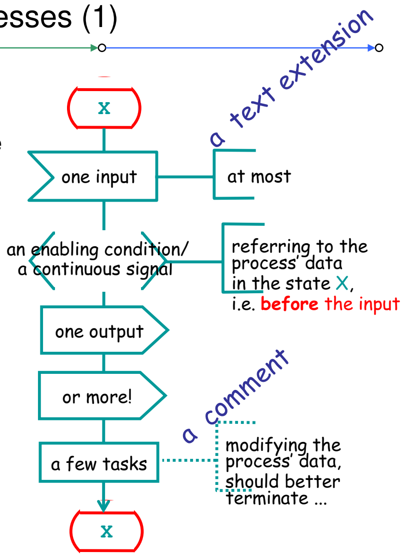
- Processes interact via signals, the signals can be associated with messages (send output signals and consume input signals)
- Outputs are sent away (non-blocking).
- Each signal is buffered at the receiving process side (FIFO)!
- Signal transmission is buffered (not immediate!)
- Routing is defined by
  - channels (on system level),
  - signal routes (on block level)





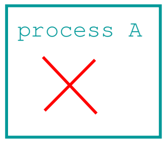
Let's start bottom-up: SDL-Processes (1)

- Processes are kind of automata (Extended Finite State Machines)
- Each Process has its own input queue (infinite storage capacity, in principle)
- A state of a process is determined by
  - current state of process
  - value of current variable
  - contents of input queue
- state changes can be triggered by input messages, where a transition may execute the following actions:
  - send output
  - call procedure
  - execute loop
  - manipulate local variables
  - create new instance of process
  - ....

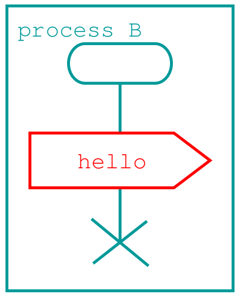


x
one input
at most
a text extension
an enabling condition/ a continuous signal
referring to the process' data in the state X, i.e. before the input
one output
or more!
a comment
a few tasks
modifying the process' data, should better terminate ...
x

The simplest SDL process (2)



stop symbol: process instance and its associated input queues are destroyed



textual' SDL:

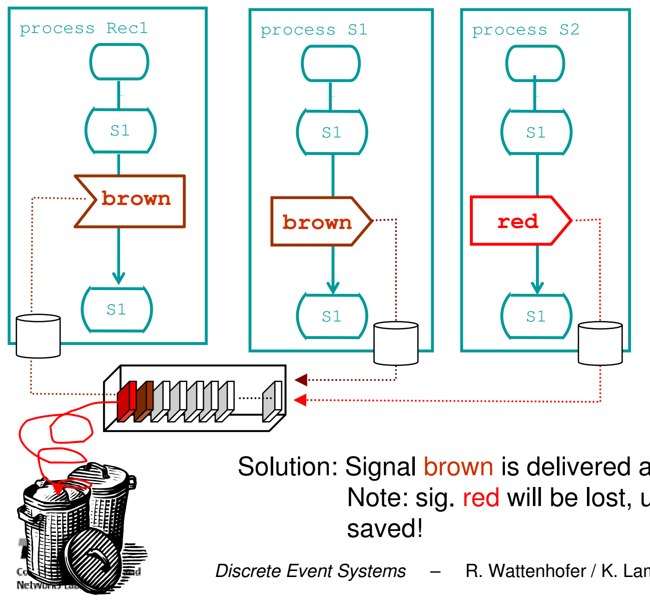
PROCESS B;
START;
OUTPUT hello;
STOP;
ENDPROCESS

sends signal **hello** and terminates.

start symbol (not a state), cannot wait for input

state, waits for inputs, or 'waits' for NONE, or is enabled by a continuous signal.

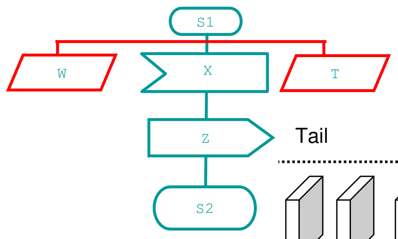
Process-interaction (1)



- S1 and S2 send signal **brown**, red resp.
- Order of arrival at input queue depends on scheduling.
- Let red be first.
- But, Rec1 wants to consume signal **brown** not red!

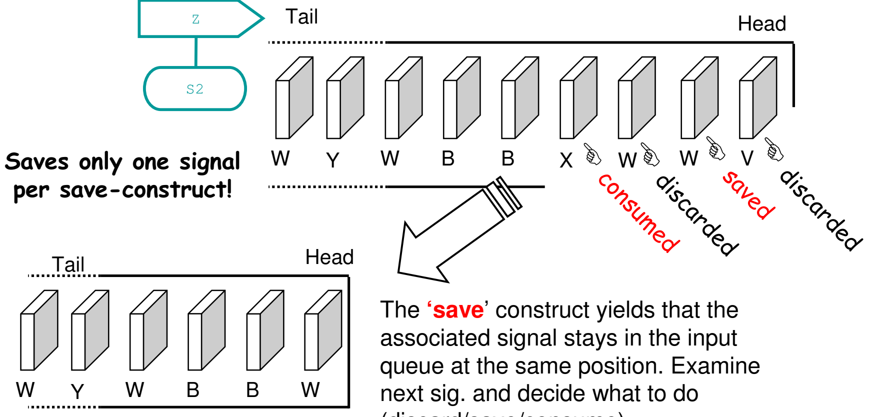
Solution: Signal brown is delivered and red is dumped!
Note: sig. red will be lost, unless it is explicitly saved!

Process-interaction (2) : Saving and consuming inputs



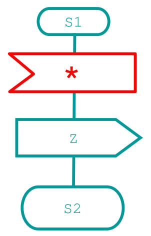
Saves only one signal per save-construct!

Inputs are non-persistent: top elements are discarded if they are unexpected
=> make use of save

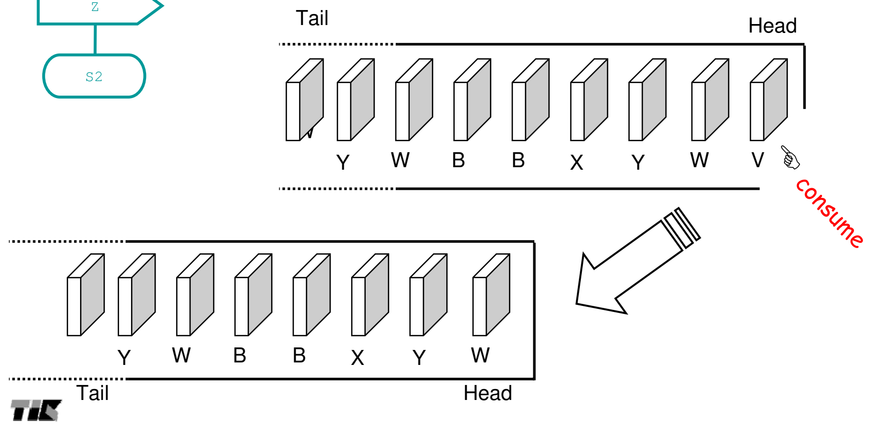


The '**save**' construct yields that the associated signal stays in the input queue at the same position. Examine next sig. and decide what to do (discard/save/consume)

Process-interaction (3) : Asterisk

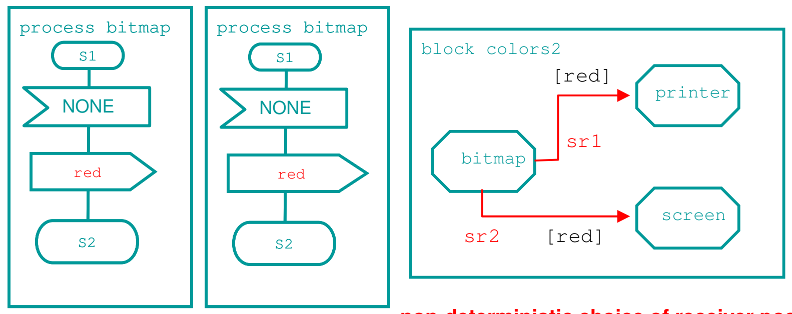


Asterisk in inputs behaves like a wild card
=> consume any signal from queue



Process-interaction (4) : Output

- output-symbol triggers sending of a signal to a connected process
- format: name_of_signal [list of parameters (optional)]
- With VIA or TO route or receiver can be addressed explicitly (why?)

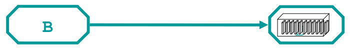


non-deterministic choice of receiver possible!

Process-interaction (5): Signaling on routes and channels

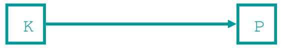
Block level: signal routes are non-delaying

- 'implemented' by synchronization
- but the inputs are still buffered at the receiver side.

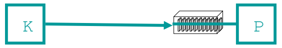


System level: channels can be

- non-delaying
  - 'implemented' by synchronization
  - but eventually the inputs are buffered at the receiver side!



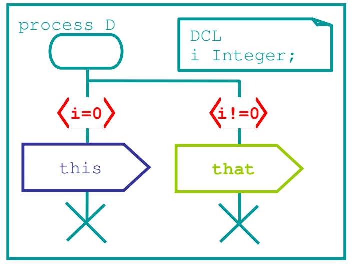
- delaying
  - 'implemented' by synchronization with another unbounded FIFO buffer in the middle (makes the duration of delay unpredictable).
  - additionally, the inputs are buffered at the receiver side.



Keep in mind that channels and signal routes may be uni-directional or bidirectional.

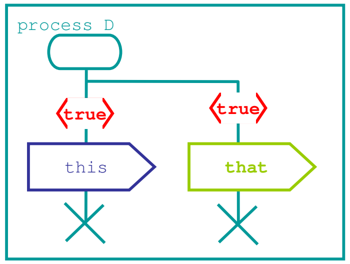
A data dependent process: Continuous signals (1)

Guarded choice



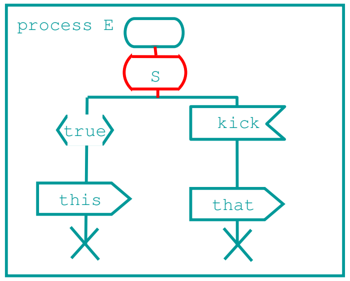
sends this if i equals 0, otherwise that.

non-deterministic choice



may do this, may do that

Continuous signals (2)



Idea: Trigger process behavior without having a new input into the process' queue

Continuous signal

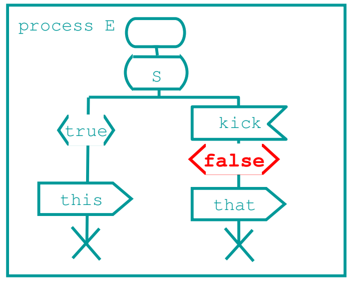
1. guards the branches of input-less alternatives,
2. guards are Boolean conditions on process data,
3. If available input signals are always processed first.
4. non-determinism among various enabled continuous signals can be resolved by assigning priorities to the conditions.

Scheduling:

1. check for input signal
2. check continuous signal with Prio 1, ….

Enabling condition

Idea: Block consumption of a signal when not ready



Enabling condition

- guards an input command, i.e.
- signal can only be consumed if guard evaluates to true
- guard is a Boolean expression,
- guard can not contain parameter of signal to be consumed !

Time in SDL

- Some snippets from the standard:
  - *" Time only progresses if all input buffers are empty. "*
  - *" Taking a transition takes a positive, but negligible amount of time. "*

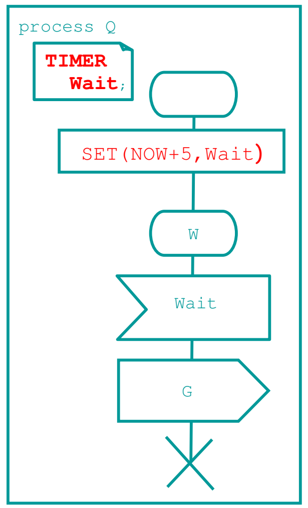
    => time consumption of transitions can not be modeled explicitly, alternatives?
- Why is time interesting to discuss?

  because SDL allows one to set timers and to react on timeouts,

  and allows to refer to the current time (NOW).

  this construct appears handy, and is often used.

Timers



1. setting a timer Wait to NOW + 5 time units
2. Stay in state W until receiving timer signal
3. Reacting on the expiration of timer Wait.

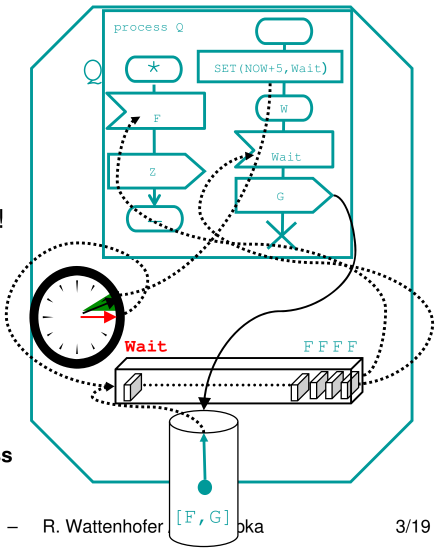
When a timer expires, a timer signal is inserted into the input queue of the process

Is this sufficient for specifying real-time systems with hard time bounds?

What happens when a timer expires

- Recall the process-interaction via buffered FIFO queues
- Expiring of a local timer just *appends* a signal to the local input queue
- This implies that reaction to timeouts may not be immediate!

(widely considered unsatisfactory by practitioners)



We will come back to this in the ex. class

SDL-Wrap-up

- SDL provides a graphical *Graphic Representation* (SDL/GR) and a textual *Phrase Representation* (SDL/PR) (same semantics!) for the specification and description of the behavior of reactive and distributed systems.
- A system is specified as a set of interconnected abstract machines (processes) which are extensions of finite state machines (FSM).
- The FSM are communicating via FIFO-queues
- SDL is a formal method, i.e. a SDL system description has a unique interpretation (semantics). Thus it is a basis for determining the correctness of a specification:
  - consistency of specification and designed system
  - verification of a specification, does it fulfill some de
- To some extend SDL guarantee implementation, due to :
  - automatic generation of code.
  - automatic generation of test case

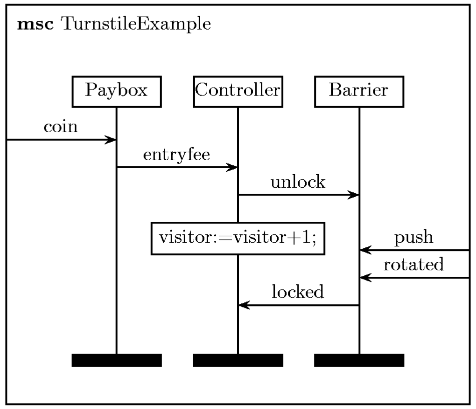
We will come back to this latter, when validating SDL-specs

MSC: Message Sequence Chart

- Standardized language for describing interaction among concurrent processes
- complements the description of a system,
- Most high-level modeling formalisms are state-based, and correspond to executable specifications (not only the ones we discuss here like SDL, PNs and TAs)

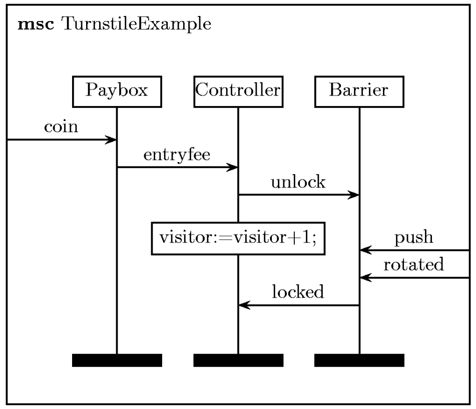
  => Often difficult to capture "informal" initial sketches.
- Therefore MSCs are often used for defining
  - Use- or Test-cases
  - Scenario-based requirements modeling which seems in particular of value for early design stages of systems. Furthermore MSC is often used for
  - documentation purpose
- UML-Sequence charts are based on MSCs, which is one of its heavier used parts.

MSC: Message Sequence Charts



- MSCs are precedence graphs with locality information
- Each process instance has its own time line
- each vertical line represents a process (or the environment)
- the arrows represent signals/messages
  the blocks represent (internal) process activities

MSC: Message Sequence Charts

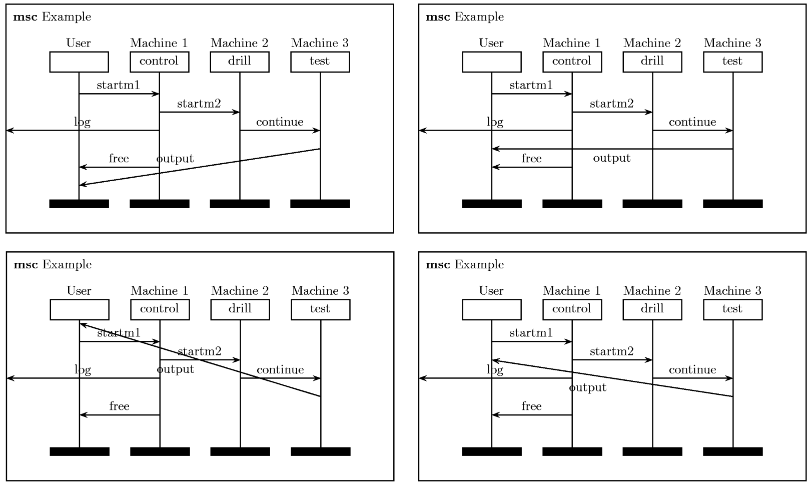


- time evolves along time lines of processes from top to bottom
  => events on the same time-line are totally ordered
- Model of communication: asynchronous FIFO channels, messages can overtake each other
- Sending and Receiving of a message is ordered
  => partial order on set of events

Recall definition of partial order?

- A relation is a set of pairs drawn from some set, say E.
- A reflexive relation is a relation that contains the pair (e,e) for each element e of E.
- A transitive relation is a relation which contains the pair (e,g) whenever it contains both (e,f) and (f,g).
- A partial order is a reflexive and transitive relation.
- A total order is a partial order which for each pair (e,f) of E (with e≠f) does either contain (e,f) or (f,e) - but not both.

MSC – Did we specify the same scenario?



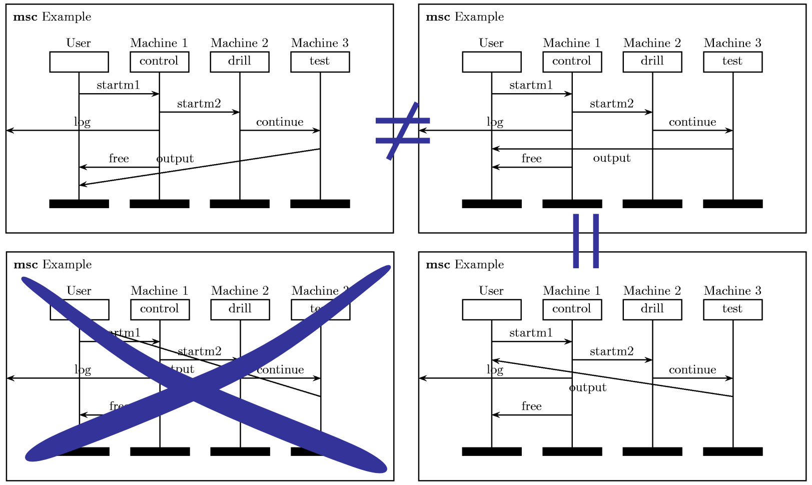
Definition: Basic MSC

A (basic) MSC M is a tuple $(P, E, L, c, <)$, with

- a set P of process labels (labelling the instance axis),
- a finite set E events $E = S \cup R \cup A$, consisting of send events S, receive events R and action events A (task executions etc.)
- a labeling function $L : E \leftarrow P$ (putting events on the instance axis),
- a bijection $c : S \rightarrow R$ (for send-receive edges)
- a precedence relation $< \subseteq E \times E$.

Send of a message occurs before its receipt. Events on the same instance are totally ordered **Must be well-formed: no cycles in precedence graph.**

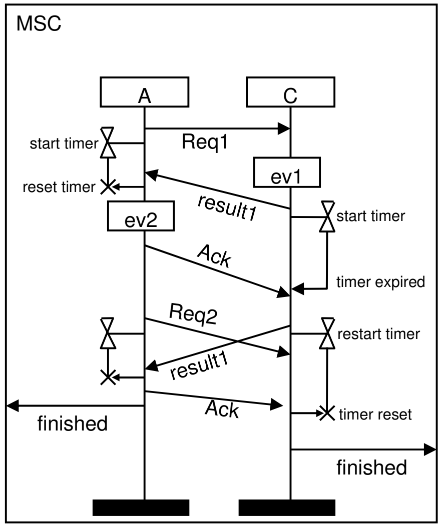
MSC - How simple!



Semantics of Basic MSC

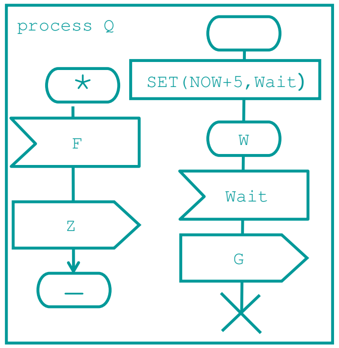
- $<^*$, the transitive closure of <, defines a partial order on E
- A trace of MSC M is a linearization of the partial order $<^*$.
  - every trace is a finite sequence of events that “obeys” the precedence.
  - each event occurs exactly once in a trace and only after all its preceding events have already occurred in the trace so far.
  - always finite.
- Semantics of MSC M
  - is the set of all possible traces.
  - can be represented as a finite LTS (Labeled Transition System).

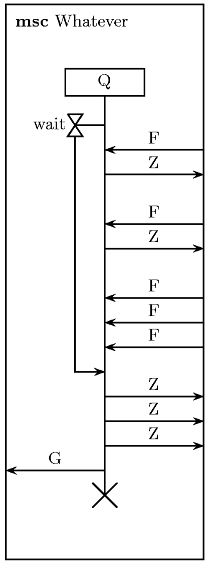
MSC: Timers



- Each timer is associated with a particular instance of a process
- Once it expires it triggers some action, i.e. sending of a signal.
- Like in SDL timer signals are queued
  => reaction might be delayed

MSC: Timers, an example





MSC and SDL

- SDL has from the very beginning been devised in combination with MSC (ITU standard Z.100/Z.120)
- Within SDL, MSC are used
  - for requirements engineering
  - for test case engineering
  - as example scenarios which the SDL system is supposed to comply to
  - as a logging means for traces generated from an SDL system
- MSC has spinned off as a requirements engineering formalism.

What next?

By now it should be clear that the the design of concurrent system is difficult and error-prone and often leads to inconsistent system behaviors with respect to the desired or required behavior.

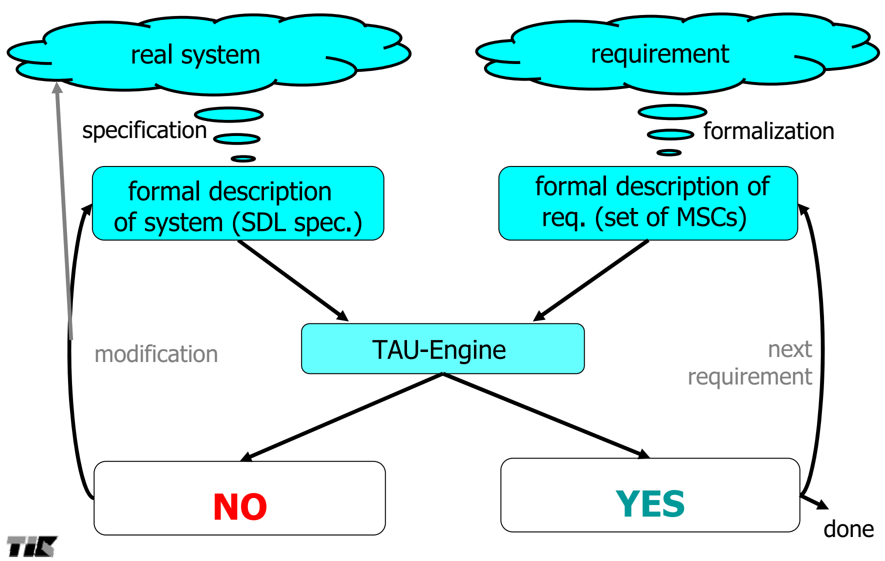
This is often a threat in real life and not acceptable

How can we check for inconsistency?

Look for
• safety violations like deadlocks,
• progress violations like livelocks

Visual inspection is infeasible!

Computer-assisted verification (The TAU-tool suite)



Tau-assisted verification

Does a design *D* satisfy a requirement ϕ?

- The Tau tool supports simulation, and some rudimentary form of state space exploration
- **Simulation**: generates traces through the state space
  - user-driven (that's not really verification, but useful)
- **State Space Exploration**:
  - explores
    - randomly (driven by a pseudo random number generator), or
    - exhaustively up to a given depth.
  - checks
    - whether the encountered states satisfy some (built in) sanity requirements, which are some simple safety properties.
    - whether a given MSC (requirement) can be generated.

Variable abstraction in Tau

- Assume that you are validating a property which is independent of the exact values of the process variables
- Further assume that your SDL specification contains masses of variables used for purposes like
  - counting messages,
  - computing results,
  - logging information,
  - etc .
- It seems a good idea not to care about the variables in the property specific verification

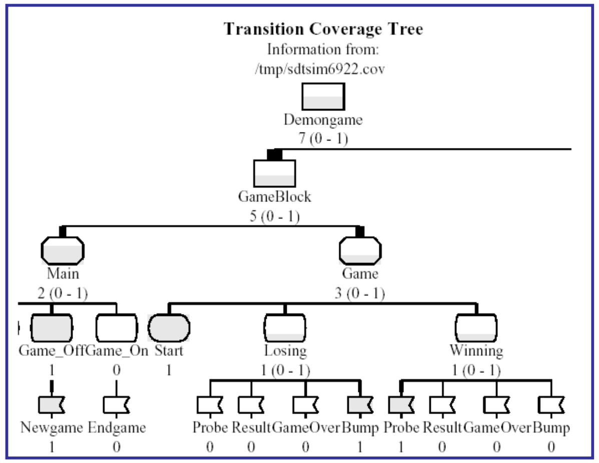
  => This is the essence of variable abstraction, and it is supported manually in Tau. The user has the opportunity to mask variables he guesses to be irrelevant for the verification.

(Vice-versa it could also be the case that one simply assumes that variables are within some intervals => abstract interpretation)

Simulation and verification within Tau.

- The simulator and validator explore the state space and allow one to view how much of a specification has been visited in the current simulation or verification run.
- BUT, the presented data refers to the executed transitions and the visited states **of the specification**

  (=> coverage rate)
- *Is this sufficient?*



State - and transition coverage vs. State space coverage

State - and transition coverage refer to the specification level.
=> Simulator and **Validator** explore the state space (**in parts only**!).

- From the TAU tool documentation

> Symbol coverage : 100.00
> All SDL symbols in the system were executed during the exploration. If the symbol coverage is not 100% the validation cannot be considered finished…..
>
> *Remark: here validation has to be read as verification.*

- This is misleading! Due to concurrency, it is very well possible
  - to have a specification that is wrong (e.g. deadlocks), but a fragment of the state space with 100% of the transitions/states covered does not show this deadlock.
  - to have a specification that works 100% correct, but where less that 100% of the specification are covered.

What to do with a verified SDL spec?

Implement the system in software (and/or hardware, sometimes).

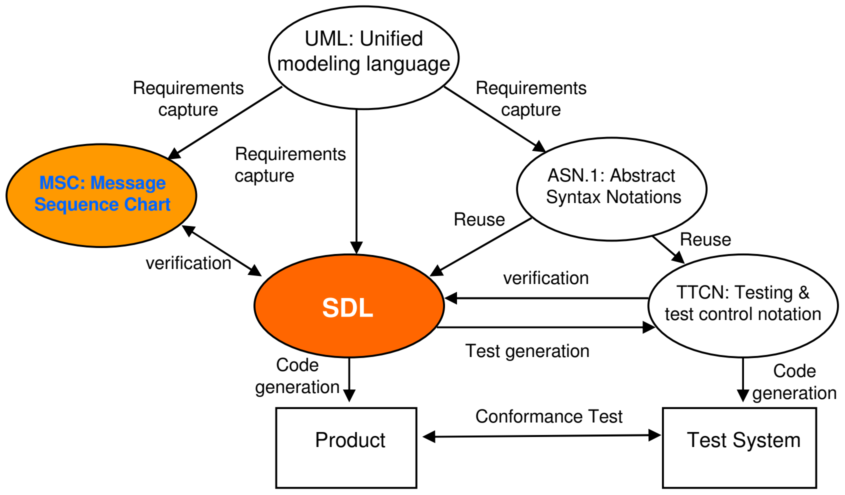
Do this

- automatically (code generation)
  - is supported by the Tau tool;
  - is often considered inefficient;
    One reason: Buffer-based communication disturbing in non-distributed implementations of SDL blocks;

or

- manually
  - specification is reference for the implementers,
  - specification can provide test-suites for the running code

The Relations between Different Languages and SDL/MSC



IEC Web tutorial (http://www.iec.org/online/tutorials/sdl/topic02.html)

Acknowledgment and Literature

- **Acknowledgements:**
  The presented slides contain material of Prof. H. Hermanns (Univ. des Saarlandes) and Prof. M. Siegle (Univ. der Bundeswehr Muenchen), who kindly approved this.

- **Literature**
  - TIMe Electronic Textbook version 4.0 September 1999
    - Chapter 13: Tutorial on SDL
      http://www.sintef.no/time/ELB40/ELB/SDL/SDL.pdf
    - Chapter 14: Tutorial on MSC-96
      http://www.sintef.no/time/ELB40/ELB/MSC96/MSC96.pdf
  - L. Doldi:
    Validation of communications systems with SDL :
    the art of SDL simulation and reachability analysis, Chichester Wiley, 2003
    Zugriff über: http://www3.interscience.wiley.com/cgi-bin/bookhome/109867833

- **More information can be found @**
  - http://www.sdl-forum.org
  - http://www.iec.org/online/tutorials/sdl/