Abstract State Machines Method

for High-Level System Design and Analysis

A Software EngineeringTechnique

Egon Börger

Dipartimento di Informatica, Universita di Pisa
http://www.di.unipi.it/~boerger

For details see Chapter 2 (ASM Design and Analysis Method) of:

E. Börger, R. Stärk

Abstract State Machines

A Method for High-Level System Design and Analysis

Springer-Verlag 2003

For update info see AsmBook web page:

http://www.di.unipi.it/AsmBook

- Overall Goal and Key Concepts
- Hierarchical System Design
  - Ground Models and Refinements
- Working Definition and Simple Examples
  - Illustrating function classification
    - clock
  - Illustrating synchronous parallelism
    - Conway's Game of Life , Daemon Game
  - Illustrating choose
    - Sorting, Non-Deterministic Language Generation, new (Java, Occam, Double Linked List), Sampling increasing real-time moments to fire a rule
  - Illustrating Ground Model Construction & Refinement
    - Constrol State ASMs: Lift
- Practical Benefits & Real-Life Applications

Overall ASM Research & Technology Transfer Goal

Improve industrial system design by rigorous high-level (hw/sw co-) modeling which is linked seamlessly, in a way the practitioner can verify and validate, to executable code

– Develop succinct ground models with precise, unambiguous, yet understandable meaning

to provide the possibility for implementation independent, application oriented, system analysis and validation

- Refine models into a hierarchy of intermediate models, modularizing orthogonal design decisions ("for change") and justifying them as correct,
  - linking the ground model to the implementation
  - documenting the entire design for reuse and maintenance

ASM Method as solid methodological foundation for systems engineering (hw/sw co-design)

modeling mathematical features separated from (but relatable to) taking into account technological, managerial, financial features

in a way which enables to cope with characteristics of software engineering:

- complexity
- abstract mathematical nature
  - ease of producing and modifying sw (intellectual freedom and evolutionary character of development process)
  - difficult to find helpful intuitive “paradigms” and “natural” analogies or visualizations
- lack of generally accepted rigorous definitions for
  - quality criteria with quantitative measurements
  - production and certification standards

ASM Method guided by key attributes for well-engineered software systems

- reliable:
  - correct: "does what it is supposed to do"
    - ground model problem (inspection of adequacy by system user)
    - verifiability problem (correctness "proof" for system user & designer)

    NB. Requires comprehensibility for humans of spec/code

    Helpfulness of production standards (size/complexity restrictions)
  - robust: acceptable behaviour also in exceptional situations
  - secure: impossibility of unauthorized access
  - safe: unreachability of "dangerous" states
- maintainable: "written & documented (reflecting design decisions) so that changes can be made without undue costs" for corrections of errors, adaptation to changing env, optimization
- effficient: low cost for production and use (space & time)
- usable easily & reliably (appropriate user interface) in a problem oriented and for humans "natural" way (preventing confusion during system use)

J.- R. Abrial (The B Book)

… the task of programming … can be accomplished by returning to mathematics

- …the precise math definition of what a program does must be present at the origin of its construction
- … in the very process of program construction … the task is to assign a program to a well-defined meaning.
  - The idea is to accompany the technical process of program construction by a similar process of proof construction which guarantees that the proposed program agrees with its intended meaning

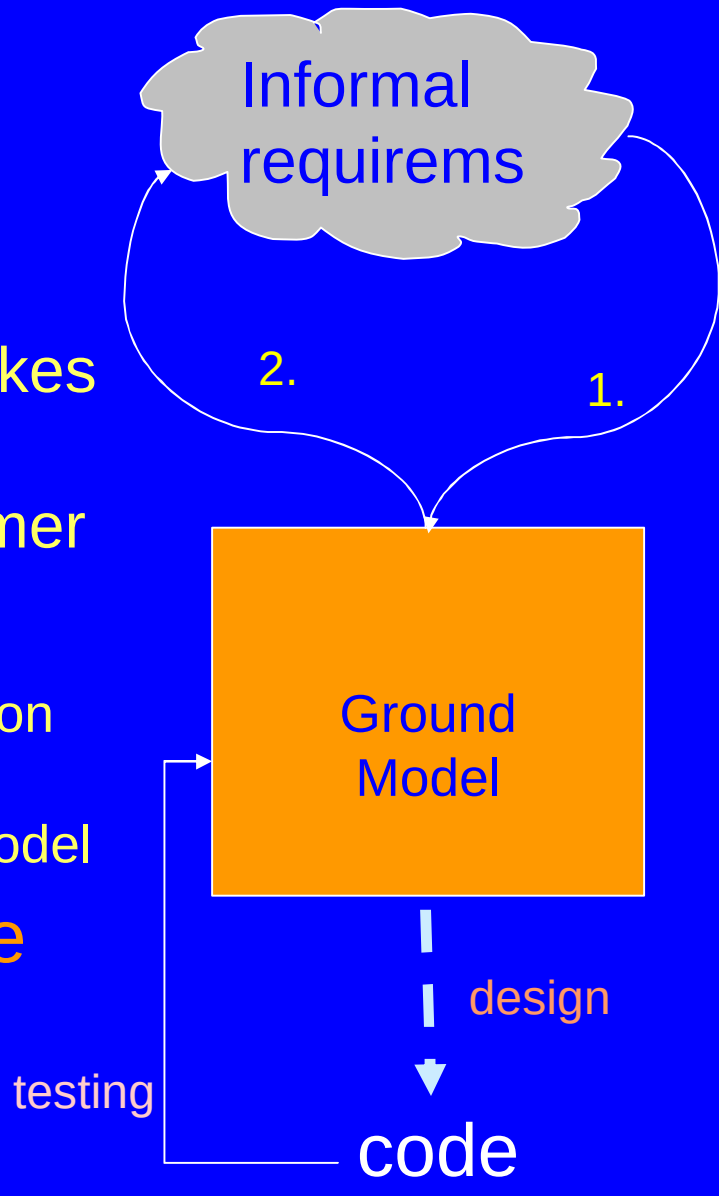
- Overall Goal and Key Concepts
- Hierarchical System Design
  - Ground Models and Refinements
- Working Definition and Simple Examples
  - Illustrating function classification
    - clock
  - Illustrating synchronous parallelism
    - Conway's Game of Life , Daemon Game
  - Illustrating choose
    - Sorting, Non-Deterministic Language Generation, new (Java, Occam, Double Linked List), Sampling increasing real-time moments to fire a rule
  - Illustrating Ground Model Construction & Refinement
    - Constrol State ASMs: Lift
- Practical Benefits & Real-Life Applications

Key Strategy for Hierarchy of Models: Divide et Impera

- Separation of Different Concerns
  - Separating orthogonal design decisions
    - to keep design space open (specify for change: avoiding premature design decisions & documenting design decisions to enhance maintenance)
    - to structure design space (rigorous interfaces for system (de)composition, laying the ground for the system architecture)
  - Separating design from analysis
    - Separating validation (by simulation) from verification
    - Separating verification levels (degrees of detail)
      - reasoning for human inspection (design justification)
      - rule based reasoning systems
        - » interactive systems
        - » automatic tools: model checkers, automatic theorem provers
- Linking system levels by abstraction and refinement

What is provided by ground models for requirements

1. **Requirements capture**
   documenting relevant application domain knowledge for designer
2. **Requirements inspection** makes correctness & completeness checkable for system user & customer
   - Verification of properties
   - Validation: mental/machine simulation (of user scenarios or components) supported by operational nature of model
3. Makes requirements traceable by relating them to design
4. Provides **test plan** basis



How to justify a ground model to satisfy the informally given requirements

- by checking (via inspection and reasoning)
- by validating (via simulation)
  - precision
  - consistency (of different views/agents)
  - completeness: providing every semantically relevant feature and basic architectural structure
  - minimality: abstracting from details that are relevant only for
    - further design (for the designer)
    - the application domain but not for the intended system (for the application domain expert or customer)

Constructing Ground Models “a major problem” of software development

Software developers in the IT, production and service sectors consistently ranked “requirements specification” and “managing customer requirements” as the most important problem they faced … more than 50% of respondants rated it as a “major problem” …

European User Survey Analysis, 30.1.1996

(M.Ibanez & H. Rempp, European Software Institute Report TR95104)

based upon 4000 questionnaire responses from 17 countries

Properties of Ground Models

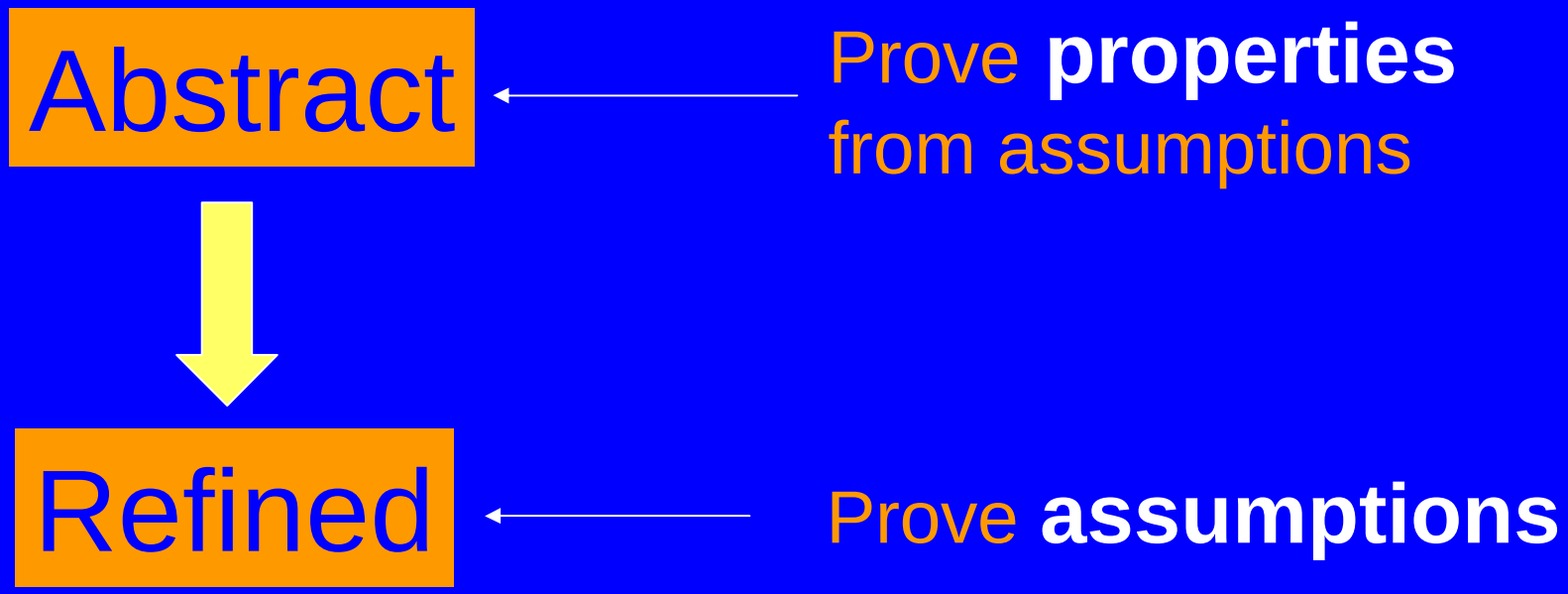
- precise at the right level of detailing, yet:
  - flexible:
    - adaptable to different application domains
    - easily modifiable/extendable to serve as prototype & for reuse
  - simple & concise:
    - to be understandable by domain expert & system user
    - for mathematical analyzability of consistency, completeness,minimality
  - resembling the structure of the real-world problem (oo credo!)
    - to be falsifiable (by experiment) and thus validatable (NB. No infinite purely mathematical justification chain is possible)
- abstract, yet:
  - complete: containing all semantically relevant params as interface
  - operational: to support process oriented understanding & simulation
- rigorous foundation for reliable tool development & prototyping

Calibrating degree of formality
wrt application (system user or customer)

- Language must be appropriate (natural) for application domain
  - easy to understand/use for practitioner, supporting
    - concentration on problem instead of notation
    - manipulations for execution & analysis of terms, consistency, completeness, etc.
  - tunable to (and apt to integrate) any
    - data oriented application (e.g. using entity relationship model)
    - function oriented application (e.g. using flow diagrams)
    - control oriented application (automata: sequential, multiple agents, time sensitive, …)
- Spec must resemble structure of the real-world problem & provide
  - data model (conceptual, application oriented)
  - function model (defining dynamics by rule executing agents)
  - interface to
    - user for communication with data/fct model by dialogue or batch operation
    - environment (neighboring systems/applications)

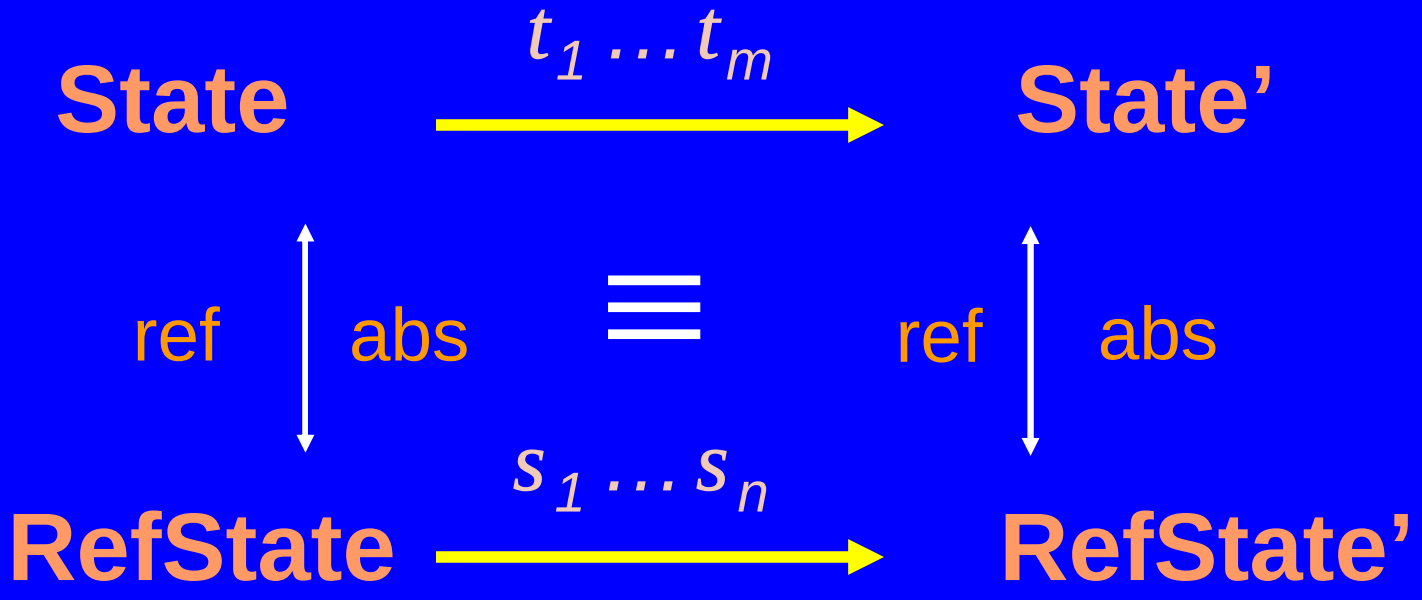
Verify implementation to meet design properties

Method: divide & conquer (ancient math paradigm)



Use correctness of refinement

The Scheme for a Correct Refinement/Abstraction Step



≡ defined

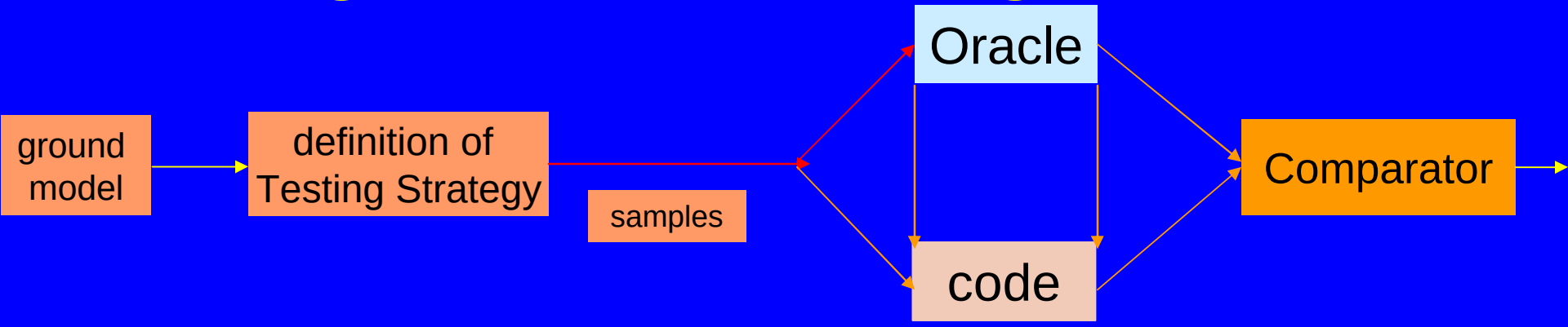
- relating the locations of interest
- in states of interest
- reached by comp segments of interest

The refinement task

- Find/formulate the right *refinement /abstraction* that
  - faithfully reflects the intended design decision (or reengineering idea)
  - can be justified to implement the given model correctly (or abstract from the given code), namely through
    - **Verification**
    - **Validation** testing model-based runtime assertions to show that design assumptions hold in the implementation
- Effect: enhancement of
  - communication of designs and system documentation (report of analysis)
  - effective reuse (exploiting orthogonalities, hierarchical levels)
  - system maintenance based upon accurate, precise, richly indexed & easily searchable documentation

See E.B.: High Level System Design and Analysis using ASMs LNCS 1012 (1999), 1-43 (Survey)

Using ASMs for test case generation



Creative, application domain driven selection/definition of test cases: guided by ground model
support user to specify relevant env parts & props to be checked, to discover req gaps

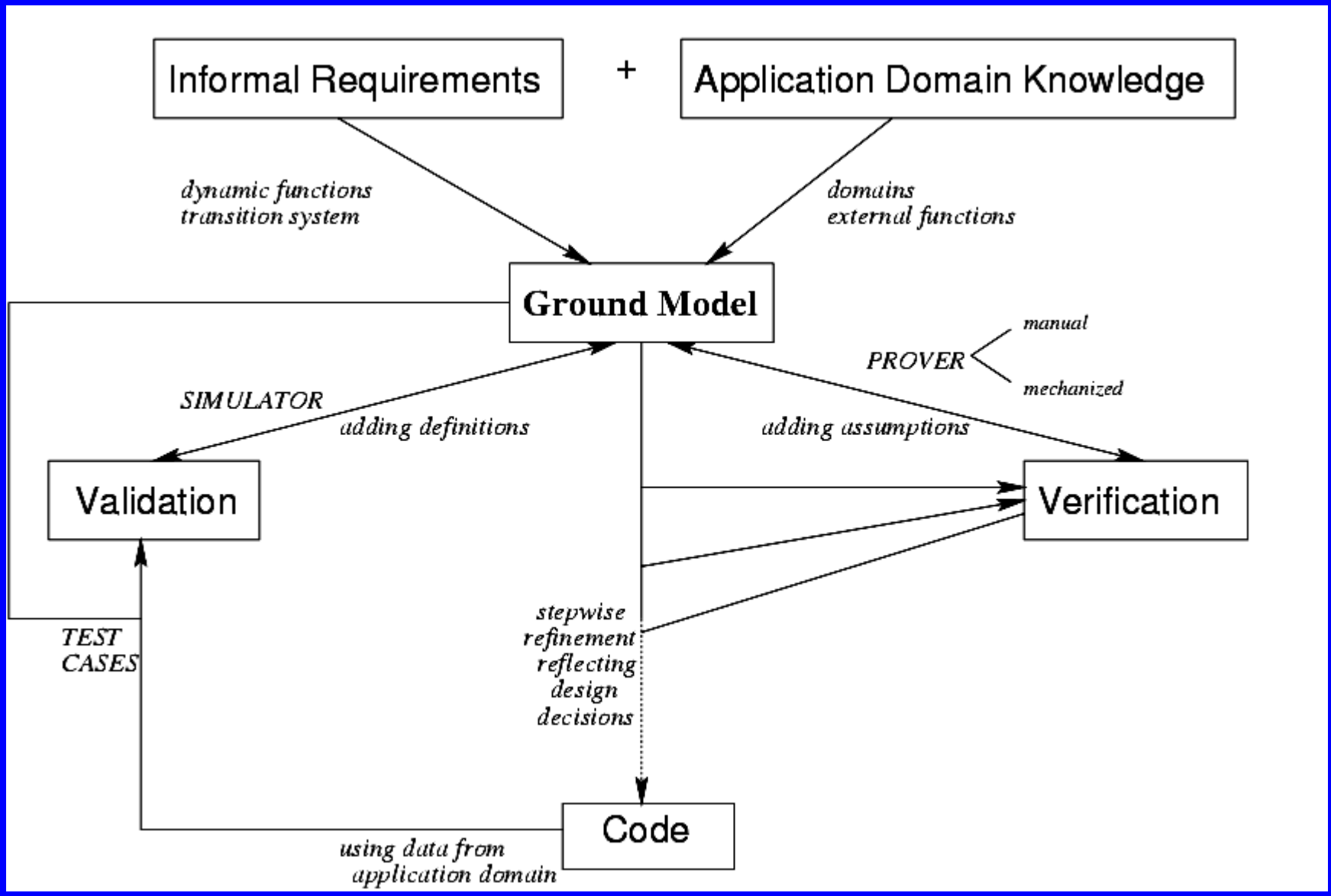
Definition of oracle to determine expected results for samples : by running the ground model

Definition of comparator using the refinement of oracle to code: determine states of interest to be related (spied) & locations of interest to be watched & when their comparison is considered successful (equivalence relation)

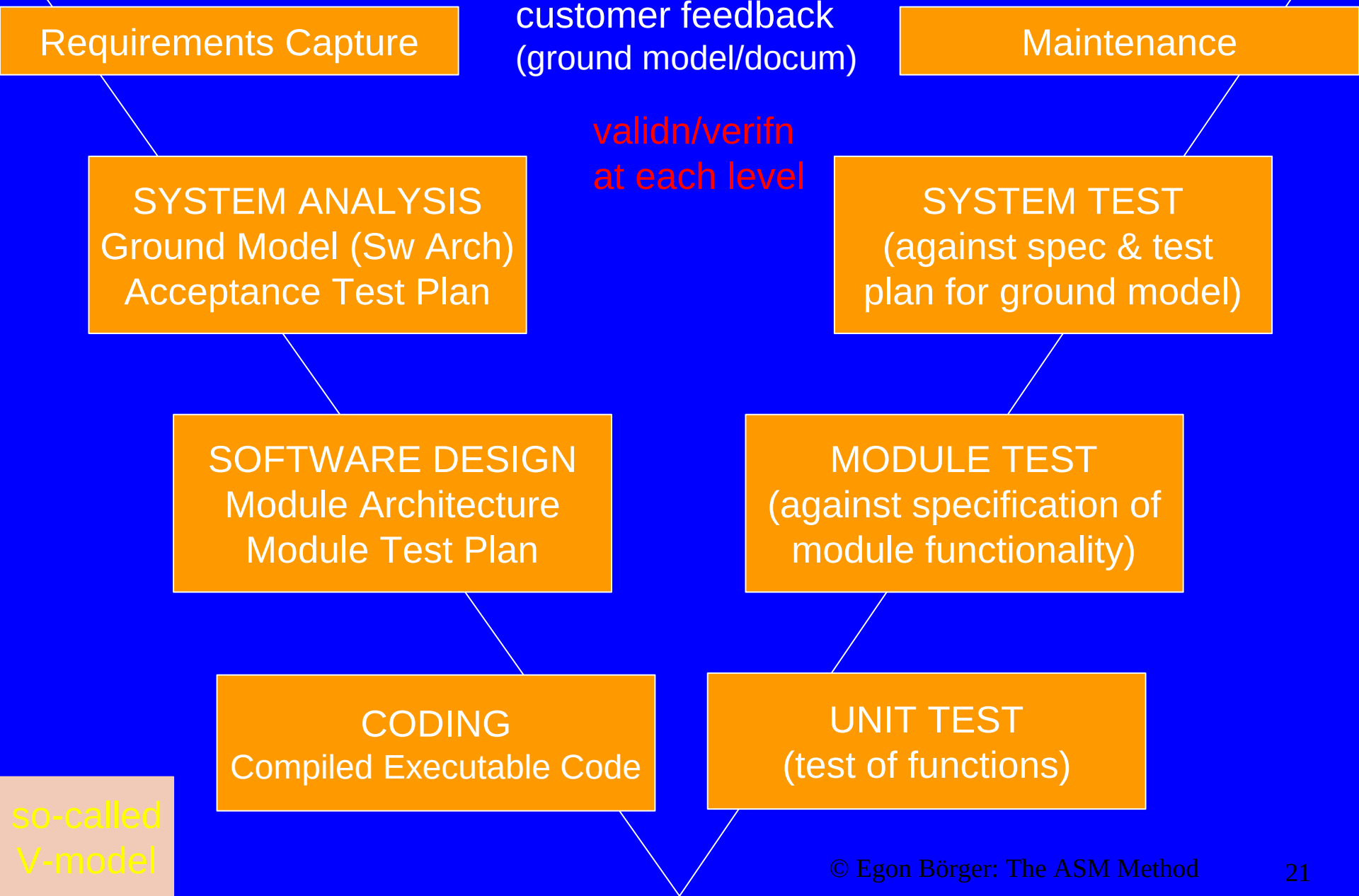
Using ASMs for maintenance

- Documentation: accurate, precise, richly indexed & easily searchable
  - reading off the relevant design features from the
    - ground model description (independent from the language chosen for the implementation)
    - refinement step descriptions
    - model-based runtime assertions appearing in the test reports
- Support
  - examine the model for fault analysis
  - use the model to recognize where to correct bugs which have been reported
- Versioning
  - reuse the model (exploiting orthogonalities, hierarchical levels)

Role of Abstract Models in Iterative System Design Process



Integration of ASMs: at different levels, to be kept coherent



so-called V-model

- Overall Goal and Key Concepts
- Hierarchical System Design
  - Ground Models and Refinements
- Working Definition and Simple Examples
  - Illustrating function classification
    - clock
  - Illustrating synchronous parallelism
    - Conway's Game of Life , Daemon Game
  - Illustrating choose
    - Sorting, Non-Deterministic Language Generation, new (Java, Occam, Double Linked List), Sampling increasing real-time moments to fire a rule
  - Illustrating Ground Model Construction & Refinement
    - Lift
- Practical Benefits & Real-Life Applications

All one needs to know about basic ASMs

- Def. A basic ASM is a finite set of rules of the form

  If Condition Then Updates

  with Updates a finite set of $f(t_1,\ldots,t_n):=t$ with arbitrary functions,

  Condition a Boolean valued expression (1st order formula).

  See separation of basic events (guarded assignments) from scheduling in event-B

- In the current state (structure) S:
  - determine all the fireable rules in S (s.t. Cond is true in S)
  - compute all exprs t_i,t occuring in updates $f(t_1,\ldots,t_n):=t$
  - execute simultaneously all these function updates
- The updating yields the next state S'
- NB. The parallelism of basic ASMs can be extended to synchronous or asynchronous multi-agent ASMs (see below).

Standard Notation choose/forall in Basic ASMs

- Supporting non-determinism by non-controlled selection functions:

choose x satisfying Cond

R

where Cond is a Boolean valued expression and R a rule. This is an abbreviation for R(x/select) where select is a selection function which applied to states with non-empty Cond yields an element satisfying Cond

- Supporting synchronous parallelism by simultaneous execution of function updates:

forall x satisfying Cond

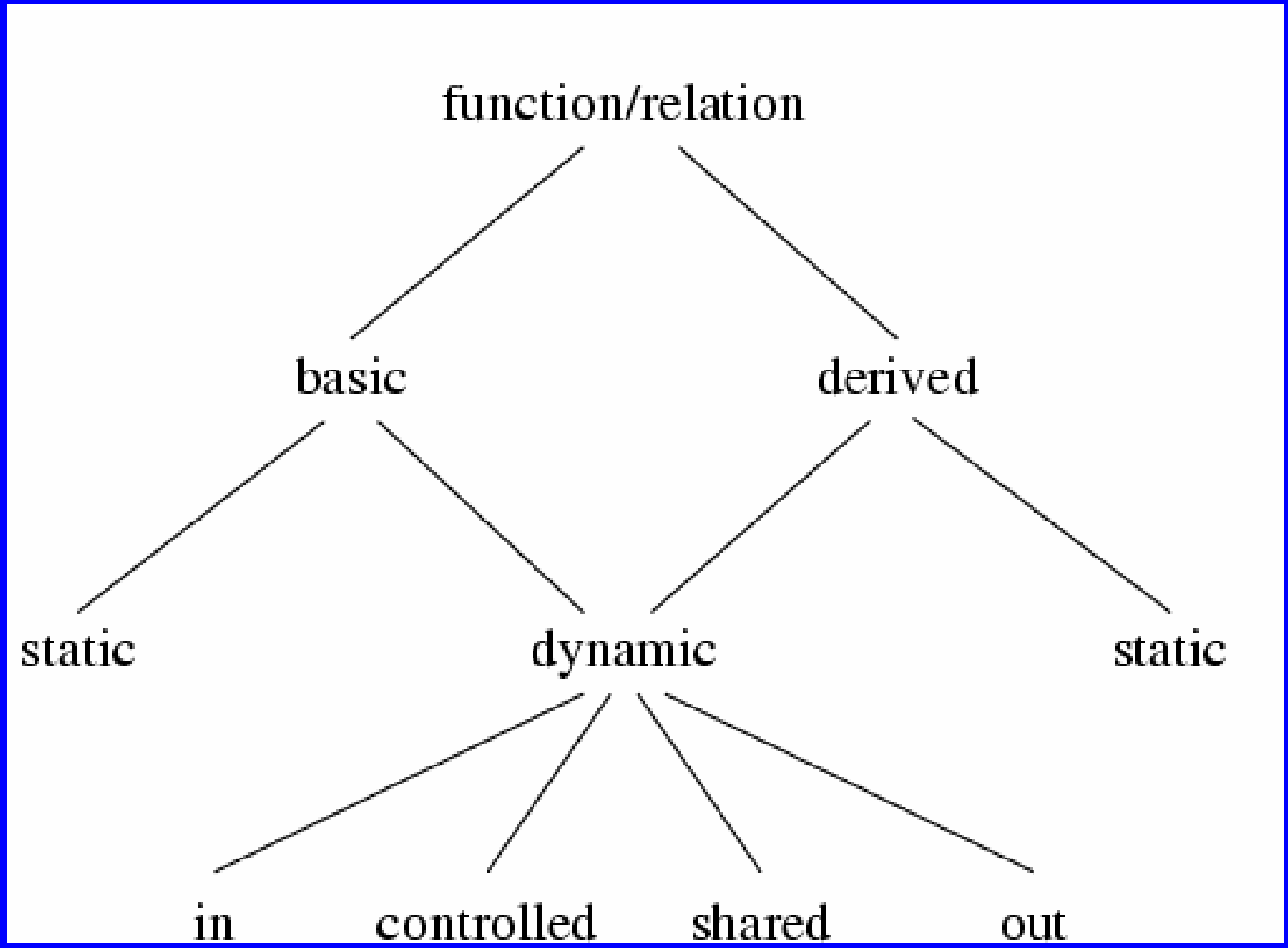
R

to simultaneously execute R(x), for all x satisfying Cond in the given state S, to form the next state S'

Classification of Locations/Functions for given ASM M

- Dynamic: values depend on states of M
  - in (monitored): only read (not updated) by M, written only by env of M
  - out: only written (not read) by M, only read by env
  - controlled: read and written by M
  - shared: read and written by M and by the env of M (so that a protocol is needed to guarantee consistency of updates)
- Derived: values computed from monitored and static functions by some fixed scheme
- Special notations for selection functions:
  - choose x satisfying Cond in R ≡ R(x/select) where select is assumed to provide some value satisfying Cond (if there is some)
  - let x=new (D) in R where new is supposed to provide fresh elements

ASM Function Classification (Interface and Modularization Mechanism)



helps system designer to control info given to programmer (e.g. only signature, an axiomatic spec, constraints, a module, …): information hiding

- Overall Goal and Key Concepts
- Hierarchical System Design
  - Ground Models and Refinements
- Working Definition and Simple Examples
  - Illustrating function classification
    - clock
  - Illustrating synchronous parallelism
    - Conway's Game of Life , Daemon Game
  - Illustrating choose
    - Sorting, Non-Deterministic Language Generation, new (Java, Occam, Double Linked List), Sampling increasing real-time moments to fire a rule
  - Illustrating Ground Model Construction & Refinement
    - Lift
- Practical Benefits & Real-Life Applications

Illustrating the ASM Function Classification

- A real time CLOCK:
  - Monitored: CurrTime: Real (supposed to be increasing)
  - Controlled: DisplayTime: Nat x Nat
  - Static: Delta: Real (system dependent time granularity), +, conversion to convert Real values into elements of Nat

If DisplayTime + Delta = CurrTime
Then DisplayTime := conversion(CurrTime)

- With the following derived fct
  - ClockTick = 1 iff (CurrTime = DisplayTime + Delta)

expressing a standard computing procedure, the rule becomes

If ClockTick = 1 Then DisplayTime := CurrTime

Examples for modularization of ASM models by static and monitored functions

- f(self,x) allows agent self to interprete f his way
- PROLOG: procdef (derived) to determine alternatives for a goal in a program, unify (static), find_catcher (derived) to determine exception handler
- VHDL'93: reject (static) for inertial delay, signal propagation with driving_val (derived from signal sources) and effective_val (derived from port-signal association)
- PVM, UML: event for task/daemon/diagram interaction via asynchronous mssg passing
- OCCAM: flowchart for pgm layout (rec compile) & exp_eval
- Java/JVM: context(pos) for switching context when walking through abstract syntax tree for exp eval and stm execution

- Overall Goal and Key Concepts
- Hierarchical System Design
  - Ground Models and Refinements
- Working Definition and Simple Examples
  - Illustrating function classification
    - clock
  - Illustrating synchronous parallelism
    - Cycling, Conway's Game of Life , Daemon Game
  - Illustrating choose
    - Sorting, Non-Deterministic Language Generation, new (Java, Occam, Double Linked List), Sampling increasing real-time moments to fire a rule
  - Illustrating Ground Model Construction & Refinement
    - Lift
- Practical Benefits & Real-Life Applications

Bounded Synchronous Parallelism

- CycleThru(R_0,..,R_n) =

  forall i=0,…,n

    If cycle = i

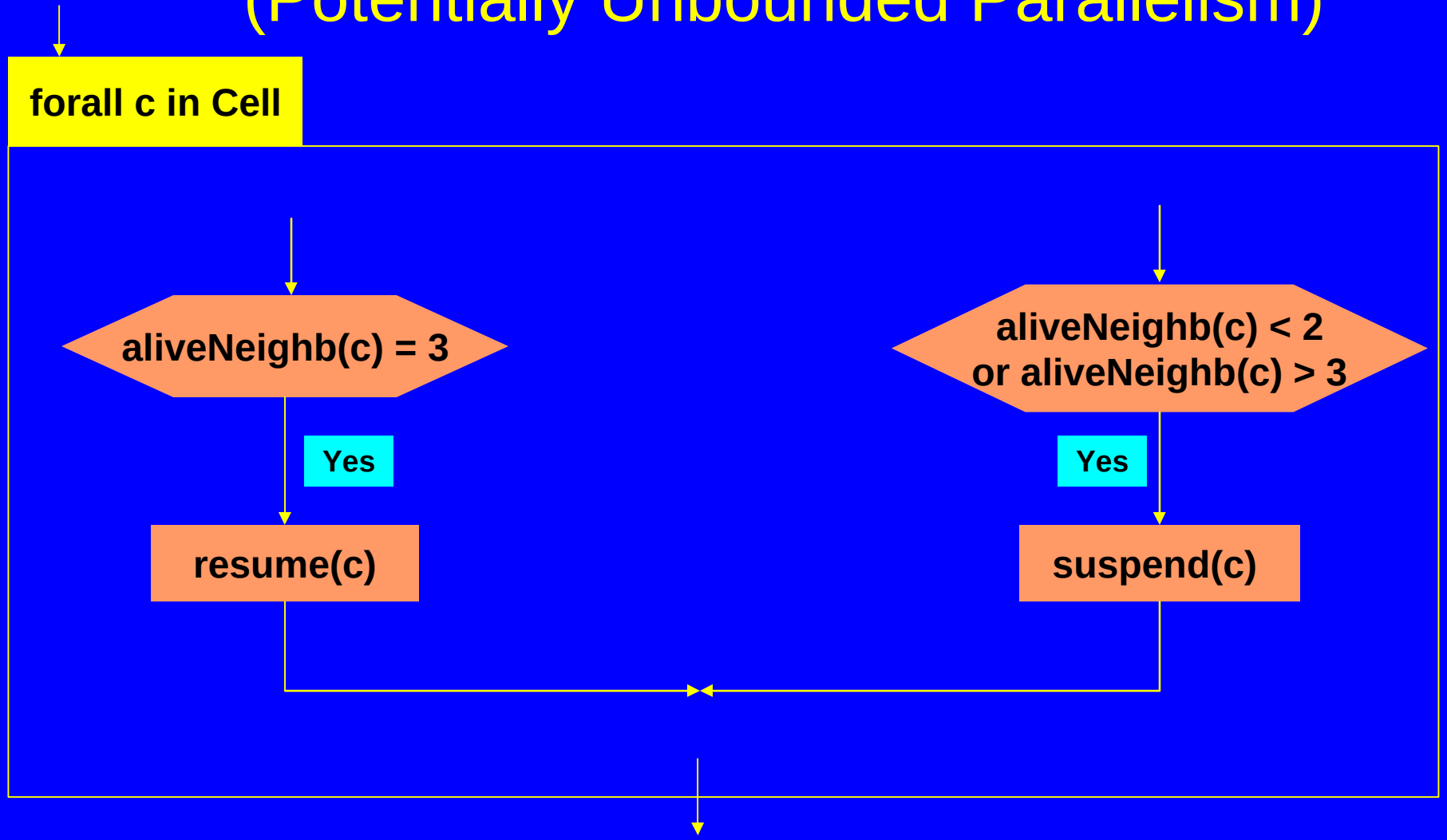
    Then

      R_i

      cycle := cycle+1 (mod n+1)

- Special case: Alternate(R,S) (n=1)

Conway's game of life: cell evolution rule (Potentially Unbounded Parallelism)



resume(c) = alive(c) := true

suspend(c) = alive(c) := false

Daemon Game (Turner 1993) Problem Description (1)

- Design the system for the following multi-player game:
  - A Daemon generates Bump signals at random. Players in the env of the system have to guess whether the number of generated Bump signals is odd or even, by sending a Probe signal. The system replies by sending the signal Win resp. Lose if the number of the generated Bump signals is odd resp. even.
  - The system keeps track of the score of each player. The score is initially 0. It is increased (decreased) by 1 for each (un) successful guess. A player can ask for the current value of the score by the signal Result , which is answered by the system with the signal Score.

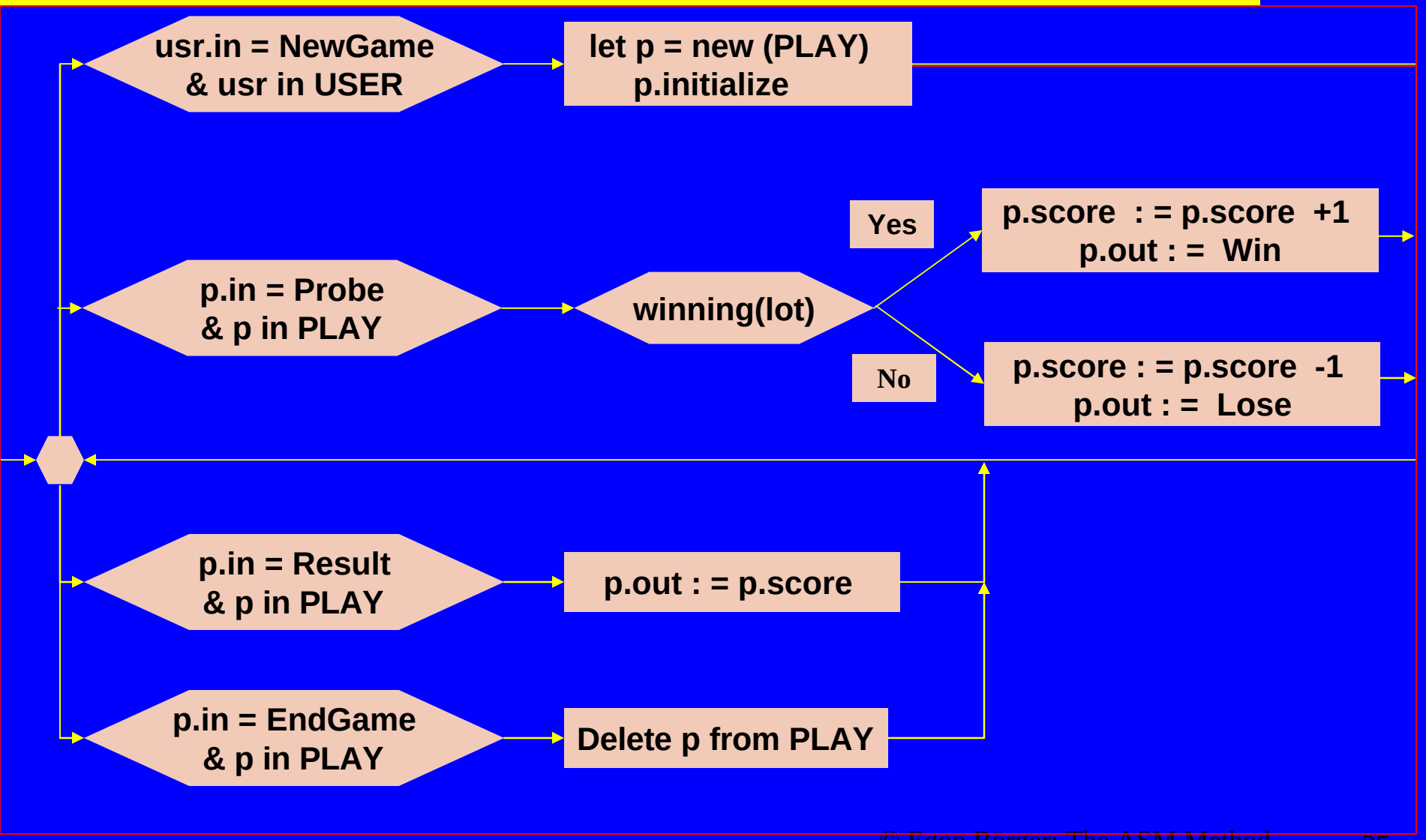
Daemon Game (Turner 1993) Problem Description (2)

– Before a player can start playing , the player must log in, by the signal NewGame. A player logs out by the signal EndGame. The system allocates a player a unique ID on logging in, and de-allocates it on logging out. The system cannot tell whether different IDs are being used by the same player.

Multi-agent ASM for Daemon Game

with global dynamic external lot issuing daemon

forall usr in USER s. t. usr. in =NewGame, forall p in PLAY s.t. p. in is defined



Daemon Game : Further Explanations (1)

- p.initialize = {p.score:=0, usr.out:=p}
- Is the (lot issuing) Daemon
  - 'integral part of the description' or 'artefact of the informal explanation'?
  - bump count global ('since the system started') or per game?

Both questions are about the nature of a lot (global or local):

- Monitored fct (s)
  - winning/losing lot, Bump counter increasing by 1, alternating 0-1-fct
- Agent with rule forall p in PLAY p.bump := 1 – p.bump
- Derived fct p.bump = bump–$p.bump_{init}$, with global bump, adding to p.initialize the update $p.bump_{init}$:= bump

Daemon Game : Further Explanations (2)

- Identification of players and games: 'Presumably some identifiers are needed, but how should they be allocated and what should they distinguish?' 'The players should be an anonymous part of the env of the system.'
  - **each invokation of new(PLAY) adds a fresh element to PLAY**
  - linking user and play by adding p.user:= sender to p.initialize
- Interruption of Probe or Result: 'Should it be allowable for another signal to be processed by the system bw Probe and Win/Lose, or bw Result and Score?' 'Probe and Result should be followed by their respective responses before any other signal is processed.

  Guaranteed by the atomicity of ASM rule execution

Daemon Game : Further Explanations (3)

- Robustness Conditions:
  - 'What should happen if a player who is already logged in tries to issue NewGame again?' 'NewGame should be allowed to happen in a current game, but should be ignored.'
  - 'What should happen if a player issues any signal other than NewGame before logging into a game?' 'The intention was to allow Probe, Result, or EndGame when a game is not current, but to ignore these signals.'

  Guaranteed by no ASM rule being applicable
- Excluding behaviour: 'What should happen if the player issues Win, Lose, or Score signals?' 'The intention was to disallow such behaviour:it simply must not happen, as opposed to happening but be ignored.'

  Excluded by the signature cond

  x.in =NewGame,Probe,Result,EndGame

- Overall Goal and Key Concepts
- Hierarchical System Design
  - Ground Models and Refinements
- Working Definition and Simple Examples
  - Illustrating function classification
    - clock
  - Illustrating synchronous parallelism
    - Conway's Game of Life , Daemon Game
  - Illustrating choose
    - Sorting, Non-Deterministic Language Generation, new (Java, Occam, Double Linked List), Sampling increasing real-time moments to fire a rule
  - Illustrating Ground Model Construction & Refinement
    - Lift
- Practical Benefits & Real-Life Applications

Non-deterministic Sorting or Variable Assignment

- Non-deterministic sorting by iterating local swap:

choose i,j in dom(a) s.t.i < j & a(i) > a(j)

a(i) := a(j)

a(j) := a(i)

- Non-deterministic choice of variable assignments in COLD:

ColdModify(Var) =

choose n Î N

choose x(1),…,x(n) Î Var , choose v(1),…,v(n) Î Value

forall i=1,…,n

val(x(i)) := v(i)

Non-deterministic language generation (1)

- generating all and only the pairs $vw \in A^*$ of different words v,w of same length (i.e. $v \neq w$ and $|v| = |w|$)

choose n, i with $i < n$

choose $a,b \in A$ with $a \neq b$

$v(i) := a$

$w(i) := b$

forall $j < n$, $j \neq i$

choose $a,b \in A$

$v(j) := a$

$w(j) := b$

When all possible choices are realized, the set of reachable states vw of this ASM, started with (say) a b for some $a \neq b$, is the set of all vw with $v \neq w$ and $|v| = |w|$.

The power of non-determinism

- Let $L_n = \{ v \# w \mid v \neq w \text{ and } |v| = |w| = n \}$.
- Exercise. Show that for each n, L_n can be accepted by a non-deterministic finite automaton with $O(n^2)$ states.
- Every unambiguous automaton that accepts L_n needs at least 2^n states.
  - See C. M. R. Kintala and K-Y Pun and D. Wotschke: Concise representations of regular languages by degree and probabilistic finite automata. In: Math. Systems Theory 26 (4) 1993, 379—395.

Non-deterministic language generation (2)

- generating the words over alphabet {0,1} of length at least n with a 1 in the n-th place, i.e. the words of form $v1w \in \{0,1\}^{n-1}\ 1\ \{0,1\}^*$.
- Let n be arbitrary, but fixed.

choose $v \in \{0,1\}^{n-1}$

choose $w \in \{0,1\}^*$

out := v1w

NB. When all possible choices are realized, the set of words appearing as values of out is the desired set.

For each n, there is a non-deterministic FSM with O(n) states which accepts the set $\{0,1\}^{n-1}\ 1\ \{0,1\}^*$, but every deterministic FSM accepting this set has at least 2^n states.

Creating new class instances in Java

if pgm(pos)=new c pos denotes the current "pc" value in the program

then if initialized(c)

then let ref = new(REFERENCE) in

heap(ref):= Object(c, InstFieldsDefaultVal(c))

pgm:= pgm[ref/pgm(pos)]

"return" ref substituting it in pgm at pos for new c

else initialize(c)

Java classes have to be initialized before being accessed

Idea: When the current instruction requires to create a new class instance (of an initialized class), a new object reference, which is returned as result of executing the new-instruction, is placed on the heap with its instance fields initialized to their default values

Creating new parallel processes in Occam

if instr(pos(a))=par S_1... S_n — pos(a) = current "pc" value of agent a

and running(a)

then let x_1,..., x_n = new(AGENT) in — create n new agents

forall $1 \leq i \leq n$

running(x_i):= true — start each new agent to run

pos(x_i):= next(pos(a),i) — place each agent on its code

env(x_i):= env(a) — equip each agent with its env, inherited from a

parent(x_i):= a — record for each agent the parent process to whom to report upon termination

count(a):=n — parent process records how many subagents have to report to him before he will proceed

running(a):= false — parent process remains idle until all created subprocesses have terminated and reported to him

Double Linked Lists : Desired Operations

- Define an ASM which offers the following operations, predicates and functions on double linked lists, whose elements have values in a given set VALUE:
  - CreateList (VALUE) : create a new double linked list with elements taking values in Value
  - Append (L, Val) : append at the end a new element with given value
  - Insert (L, Val, Elem) : insert after Elem in L a new element with Val
  - Delete (L, Elem) : delete Elem from L
  - AccessByValue (L, Val) : return the first element in L with Val
  - AccessByIndex (L, i) : return the i-th element in L
  - empty (L), length (L), occurs (L, Elem), position (L, Elem)
  - Update (L, Elem, Val) : update the the value of Elem in L to Val
  - Cat (L1,L2) : concatenate two given lists in the given order
  - Split (L, Elem, L1, L2) : split L into L1, containing L up to including Elem, and L2 containing the rest list of L

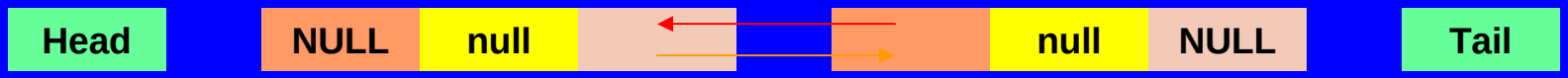
Double Linked Lists : Desired Properties

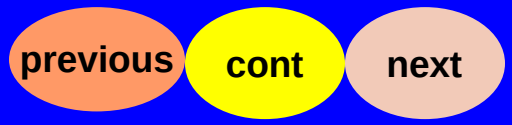
- Prove that the Linked List ASM has the following properties:
  - If the next-link of a list element Elem points to Elem', then the previous-link of Elem' points to Elem.
  - L is empty iff the next-link of its head points to its tail.
  - The set ELEM (L) of elements occurring in a list is the set of all E which can be reached, starting from the list head, by applying next-links until the list tail is encountered.
  - After applying Append (L, Val), the list is not empty.
  - A newly created linked list is empty and its length is 0.
  - By Append/Delete the list length in/de-creases by 1.
  - For non empty L and arbitrary elements E the following holds:

    Append (Delete (L,E),E) = Delete (Append (L,E),E)

Double Linked Lists : Signature

- LINKED-LIST (VALUE) : dynamic set, with fcts "pointing" to structures of the following form (often VALUE suppressed) :
  - dynamic set ELEM (L) of "objects" currly listed in L
  - distinguished elems Head (L), Tail (L) Î ELEM (L)
  - previous (L), next (L): ELEM (L) → ELEM (L) dyn link fcts
  - cont (L) : ELEM (L) → VALUE yields curr value of list elems
- initialize(L) for L Î LINKED-LIST (as usual, L is suppressed) as empty linked list with values in VALUE, defined as follows:
  - ELEM := { Head, Tail } next (Head) := Tail previous (Tail) := Head
  - previous (Head) := next (Tail) := null (ELEM) Head/Tail start/end the list
  - cont (Head) := cont (Tail) := null (VALUE) Head/Tail have no content





Double Linked Lists : Definition of Operations (1)

- CreateList (VALUE) ≡

  let L = new (LINKED-LIST (VALUE)) in initialize (L)

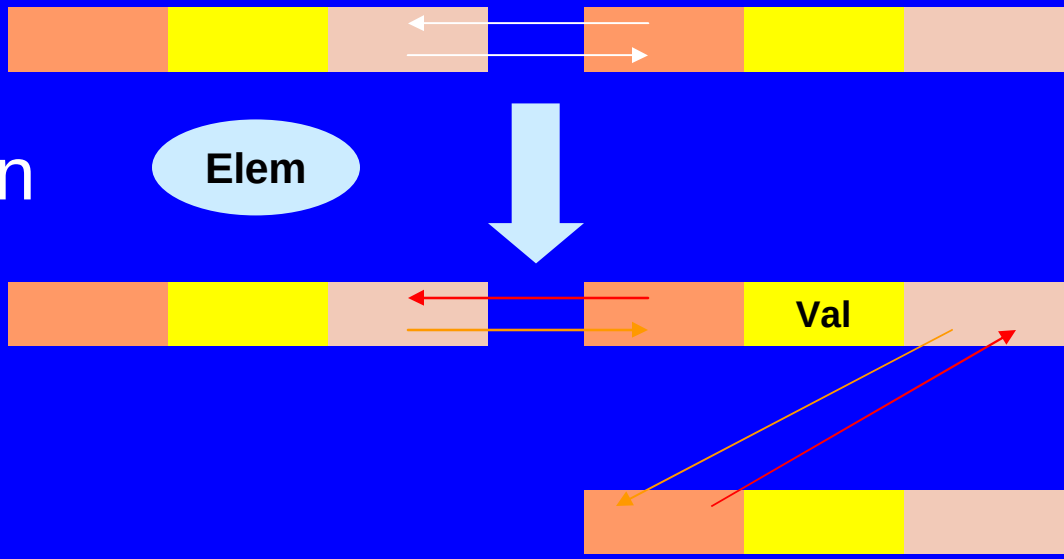
- Append (L, Val) ≡

  let e = new (ELEM (L)) in

  Link previous (Tail) & e

  Link e &Tail

  cont (e) := Val



- Insert (L, Val, Elem) ≡ let e = new **(ELEM (L))** in

  cont (e) := Val

  Link Elem & e

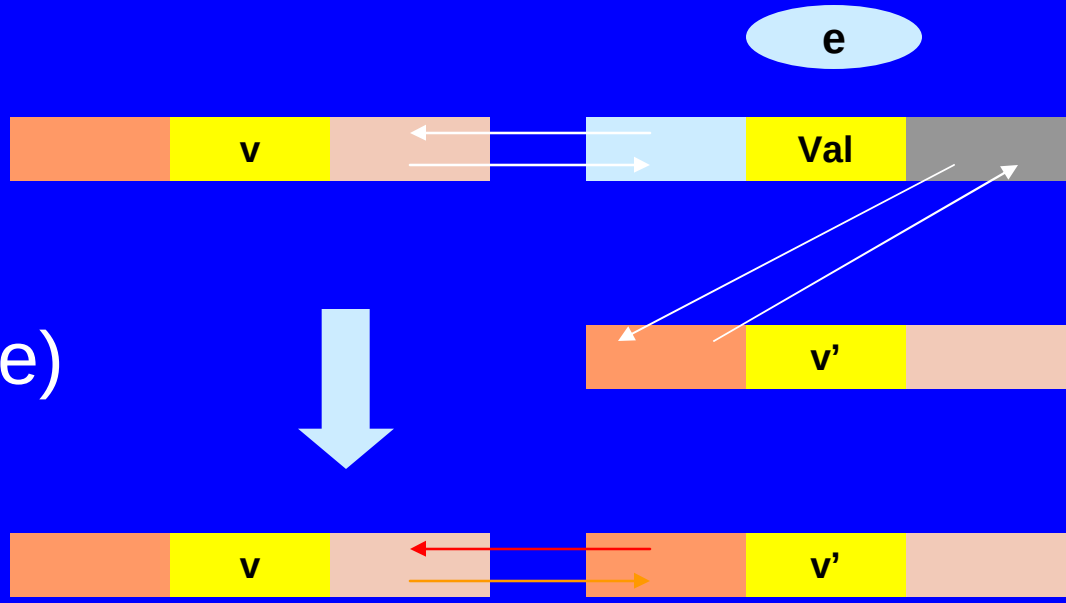
  Link e & next (Elem)

  with Link a&b ≡ next (a) := b

  previous (b) := a

Double Linked Lists : Operations & Derived Fcts (2)

Delete (L, e) ≡



Link previous (e) & next (e)

length (L) ≡ ι m (next $^{m+1}$ (Head) = Tail) well defined by initialization

occurs (L, e) ≡ ∃ i ≤ length (L) : next ^{i} (Head) = e (e ∈ ELEM(L))

position (L, Elem) ≡ ιm (next ^{m} (Head) = Elem) if occurs (L, Elem)

AccessByIndex (L, i) ≡ next ^{i} (Head) if i ≤ length (L)

AccessByValue (L, Val) ≡ next ^{m} (Head) fst occ of Val

where m= min { i | cont (next ^{i} (Head)) = Val } is defined

Double Linked Lists : Definition of Operations (3)

Update (L, Elem, Val) ≡ If occurs (L, Elem)

then cont (Elem) := Val

else error msg "Elem does not occur in L"

Cat (L_1, L_2) ≡ let L = new (LINKED-LIST) in

Head (L) := Head (L_1)

Tail (L) := Tail (L_2)

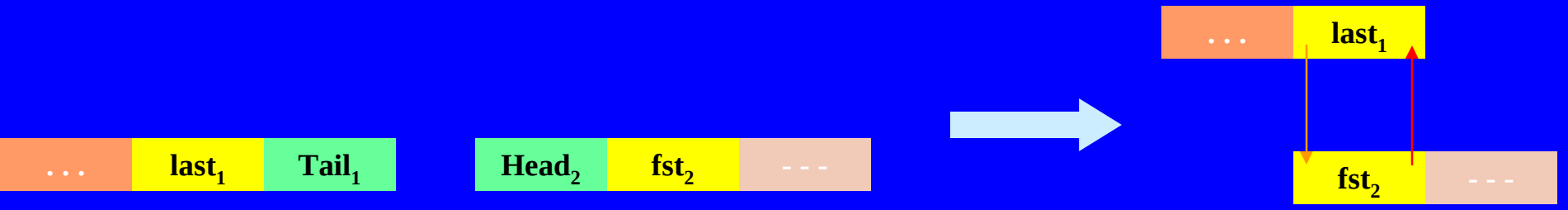
Link (L) previous (L_1) (Tail (L_1)) & next (L_2) (Head (L_2))

forall e∈ ELEM (L_1) - {previous (L_1) (Tail (L_1)),Tail (L_1) }

Link (L) e & next (L_1) (e)

forall e∈ ELEM (L_2) - { Head (L_2) ,Tail (L_2) }

Link (L) e & next (L_2) (e)



Double Linked Lists : Definition of Operations (4)

Split (L, e, L_1, L_2) ≡ let e_1 = new-tail, e_2 = new-head

Head (L_1) := Head (L)

Tail (L_1) := e_1

Link (L_1) e & e_1

where **e' = new-tail/head** ≡
cont (e') := null (VALUE)
next/previous (e') := null (ELEM)

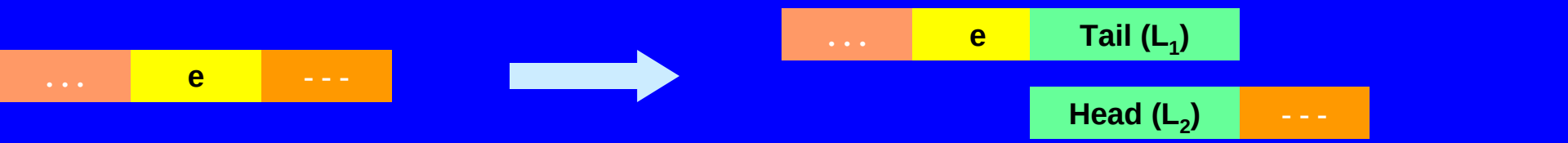
forall E∈ ELEM (L) if position (L, E) < position (L, e)
then Link (L_1) E & next (L) (E)

Head (L_2) := e_2

Link (L_2) e_2 & next (L) (e)

Tail (L_2) := Tail (L)

forall E∈ ELEM (L)-{Tail(L)}
if position (L, e) < position (L, E)
then Link (L_2) E & next (L) (E)



Double Linked Lists : Proving the Properties (1)

- If the next-link of a list element Elem points to Elem', then the previous-link of Elem' points to Elem.
  - Initially true by defn of initialize (L), preserved by each opn due to the defn of Link (L) and the fact that next/previous are modified only using this macro.
- L is empty iff
  the next-link of its head points to its tail.
- A newly created linked list is empty and
  its length is 0.
- After applying Append (L, Val), the list is not empty
- By Append/Delete the list length in/de-creases by 1.

Double Linked Lists : Proving the Properties (2)

- For L≠[ ]: Append (Delete (L,E),E) = Delete (Append (L,E),E)
  - Follow from the defn of initialize (L), length (L), Append, Delete & the fact that Append/Insert yield a non null cont.
- The set ELEM (L) of elements occurring in a list is the set of all E which can be reached, starting from the list head, by applying next-links until the list tail is encountered.
  - Follows from the defn of ELEM(L).

Sampling increasing real-time moments for firing a rule in a discrete time sequence

Goal: approximate a continuous set of rules rule(t,t') with t<t', which are to be fired in a discrete time sequence, by a finite number of samples with increasing sequence of real interval boundaries

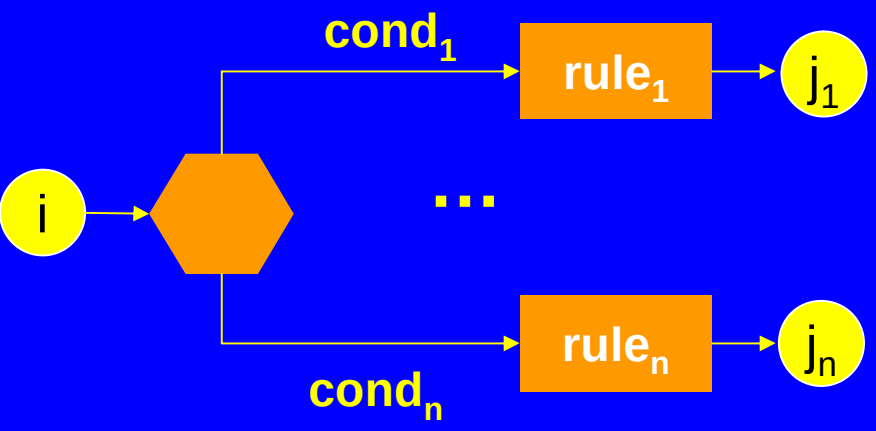
```
choose t > currtime in
        rule(currtime,t)
        samples := add(t,samples)
        currtime := t
```

Here currtime is a monitored function which is supposed to evolve continuously

- Overall Goal and Key Concepts
- Hierarchical System Design
  - Ground Models and Refinements
- Working Definition and Simple Examples
  - Illustrating function classification
    - clock
  - Illustrating synchronous parallelism
    - Conway's Game of Life , Daemon Game
  - Illustrating choose
    - Sorting, Non-Deterministic Language Generation, new (Java, Occam, Double Linked List), Sampling increasing real-time moments to fire a rule
  - Illustrating Ground Model Construction & Refinement
    - Control State ASMs: Lift
- Practical Benefits & Real-Life Applications

Extending FSMs to Control State ASMs

ASMs of rules of the form



rules are combined branching/action nodes with disjoint conds (UML parlance)

meaning

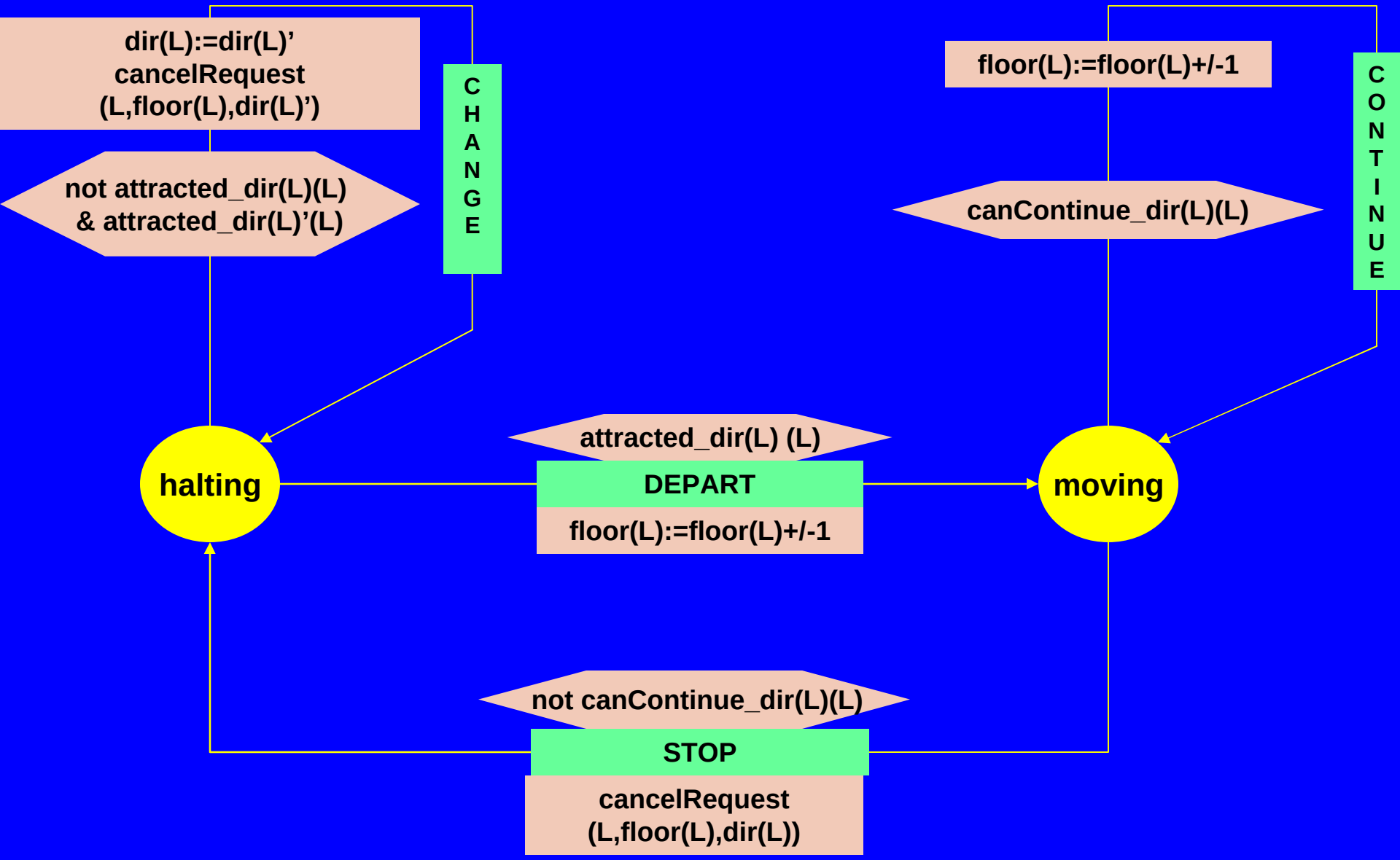
labeling of the arrows by "control" states
i, j_k often suppressed

if ctl = i then
if $cond_1$ then $rule_1$
$ctl:=j_1$
...
if $cond_n$ then $rule_n$
$ctl:=j_n$

Lift Control (N.Davis, 1984) The Problem

- Design the logic to move n lifts bw m floors, and prove it to be well functioning, where
  - Each lift has for each floor a button which, if pressed, illuminates & causes the lift to visit (i.e. move to & stop at) the floor. Upon visiting the floor the illumination is cancelled.
  - Each floor (except ground and top) has two buttons to request an up-lift and a down-lift. They are cancelled when a lift visits the floor and is either travelling in the desired direction, or visits the floor with no requests outstanding. In the latter case,if both floor request buttons are illuminated, only one should be cancelled.
  - A lift without requests should remain in its final destination and await further requests.
  - Each lift has an emergency button which, if pressed, causes a warning to be sent to the site manager. The lift is then deemed ‘out of service’. Each lift has a mechanism to cancel its ‘out of service’ status.

Lift Control : control state ASM



Refining Parametrized Macros for Lift ASM

- attracted_up (L) iff for some Floor > floor(L)
  - hasToDeliverAt (L,Floor) or
  - existsCallFromTo (Floor,Dir) for some Dir
- attracted_down (L) same with <
    - NB. attracted_dir(L) (L) stands for attracted_Dir(L) & Dir=dir(L)
- canContinue_Dir (L) iff
  - attracted_Dir (L) &
  - neither hasToDeliverAt (L,floor(L))
  - nor existsCallFromTo (floor(L),Dir(L))
    - NB. Priority to keeping direction of travel
- cancelRequest (L,Floor,Dir) =
  - hasToDeliverAt (L,Floor) := false (cancel request in L for Floor)
  - existsCallFromTo (Floor,Dir) := false

(cancel request from Floor for Dir)

Signature/Constraints for Lift Control State ASM

- shared functions bw users (for input) and lift
  - existsCallFromTo (Floor,Dir) (initially everywhere false)

    formalizes that each floor has 2 buttons to call a lift for going up/down

    constraints: always false for (ground,down), (top,up),
    and for (floor(L),dir(L)) when ctl_state(L)=halting

    formalizes that where a lift is halting no further call can be made for going into its direction of travel
  - hasToDeliverAt (L,Floor) (initially everywhere false)

    formalizes that in every lift there is for every floor a button to request delivery there

    constraint: always false for (L,floor(L)) when ctl_state(L)=halting

    formalizes that no further delivery can be requested for a floor where the lift is halting
- controlled functions
  - ctl_state(L) = halting,moving (initially halting)
  - direction(L) = up, down (initially up)
  - floor(L) current lift position, bw ground and top (initially ground)

Analyzing Possible Runs of Lift Control State ASM

L1. Non-empty runs of any L (from the initial state) have form

(DEPART CONTINUE* STOP) $^{+}$ (CHANGE (DEPART CONTINUE* STOP) *)*

Proof by induction on walks thru the diagram

L2. Moving from any reachable state, every L

moves floor by floor to the farest point of attraction in its direction of travel where, after at most |FLOOR| steps, it STOPs & then either terminates - namely iff L is not attracted in any direction - or it CHANGEs direction & moves into the new direction.

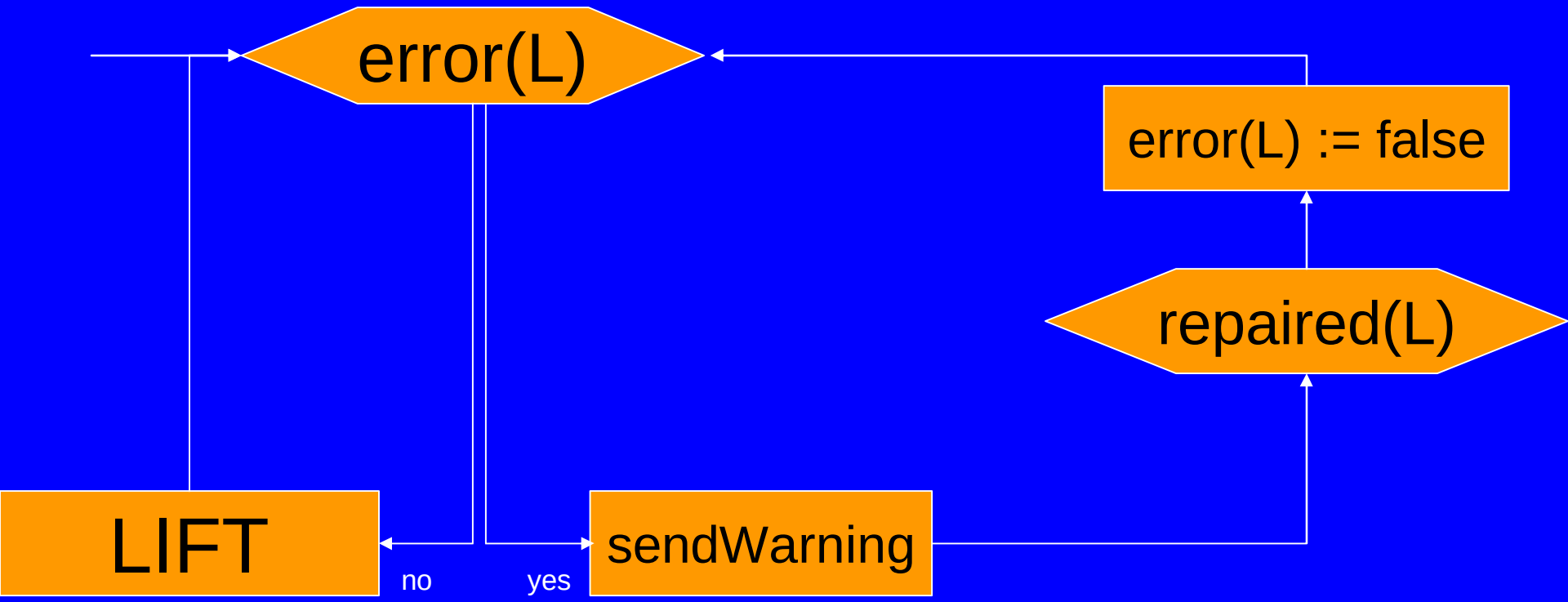
L3. When moving to the farest point of attraction in its direction of travel, every L

STOPs at each floor where it is attracted, wrt its direction of travel, and turns off the (internal) delivery request and the (external) call from that floor to go into the current direction of travel. When it CHANGEs, it turns off the (external) call from its current floor to go into the new direction of travel .

Justifying the Well Functioning of the LIFT ASM

- To show (Lift Correctness):
  - All requests for floors within lifts must be serviced eventually, with floors being serviced sequentially in the direction of travel
  - All requests for lifts from floors must be serviced eventually, with all floors given equal priority
- Proof follows from the three lemmas above, since every internal request from within L, and every external request, make L being eventually attracted in the requested direction
- NB. The proof does not exclude real-life situations with crowded lifts, where requests may be satisfied logically, but the lack of capacity prevents people from entering the lift. This problem has no logical solution, and should be solved providing more capacity ("larger bandwith")

Adding Error Handling to LIFT Machine



Add new error guard

Add ‘out-of-service’entry/exit rules

This stepwise development method is characteristic for ASMs. See for ex. the extension of ASM models for Java and the JVM by exception handling, defined and analysed in

R. Stärk, J. Schmid, E. Börger

Java and the Java Virtual Machine: Definition, Verification, Validation

Introducing Scheduling for LIFT ASM

Problem: all lifts attracted by all external calls i.e. where existsCallFromTo (Floor,_) is true

Solution: a scheduler hasBeenCalledTo assigns ONE Lift to each external call, from any Floor (for any Dir)

Modular implementation : refine the interface predicates

attracted_up (L) iff for some Floor > floor(L) (< for down)

- hasToDeliverAt (L,Floor) or
- existsCallFromTo (Floor,Dir)
  and L=hasBeenCalledTo (Floor,Dir) for some Dir

canContinue_Dir (L) iff attracted_Dir (L) & neither
hasToDeliverAt (L,floor(L)) nor (existsCallFromTo (floor(L),Dir(L))
and L=hasBeenCalledTo (floor(L),Dir(L)))

NB. cancelRequest (L,Floor,Dir) refined to
hasToDeliverAt (L,Floor) := false,
If L=hasBeenCalledTo(Floor,Dir) then existsCallFromTo (Floor,Dir) := false

Optimizing Scheduling for LIFT ASM

- Scheduling only non-crowded lifts: constraining scheduler hasBeenCalledTo to non-crowded(L).
  NB.This scheduling affects the correctness property in case of a continuously crowded lift.
- Reserving L for floor 1 and section [n,m]:
  - constraining the shared fct hasToDeliverAt(L,_) to those floors
  - restricting the shared fct existsCallFromTo (Floor,_) for those Floor to calls within that section
  - refining hasBeenCalledTo correspondingly to take into account also the requested target section.
- NB.The correctness property is relativized to the served section.

Exercise to Lift ASM

- Extend the LIFT ASM by introducing opening and closing doors
  - as atomic action
  - as durative action
  - with error handling (doors do not open/close)

References to Lift ASM

- For a solution in B which inspired the ASM developed here see Section 8.3. of
  - J.-R.Abrial: The B-Book . Cambridge University Press 1996
- For a Petri net solution see section 4, in particular Fig. 26 pg. 87, of
  - W. Reisig: Petri Nets in Software Engineering.
    - In: W. Brauer, W. Reisig, G. Rozenberg (Eds.): Petri Nets: Applications and Relationships to other Models of Concurrency. Springer LNCS 255 (1987) pp.63-96.

- Overall Goal and Key Concepts
- Hierarchical System Design
  - Ground Models and Refinements
- Working Definition and Simple Examples
  - Illustrating function classification
    - clock
  - Illustrating synchronous parallelism
    - Conway's Game of Life , Daemon Game
  - Illustrating choose
    - Sorting, Non-Deterministic Language Generation, new (Java, Occam, Double Linked List), Sampling increasing real-time moments to fire a rule
  - Illustrating Ground Model Construction & Refinement
    - Control ASMs: Lift
- Practical Benefits & Real-Life Applications

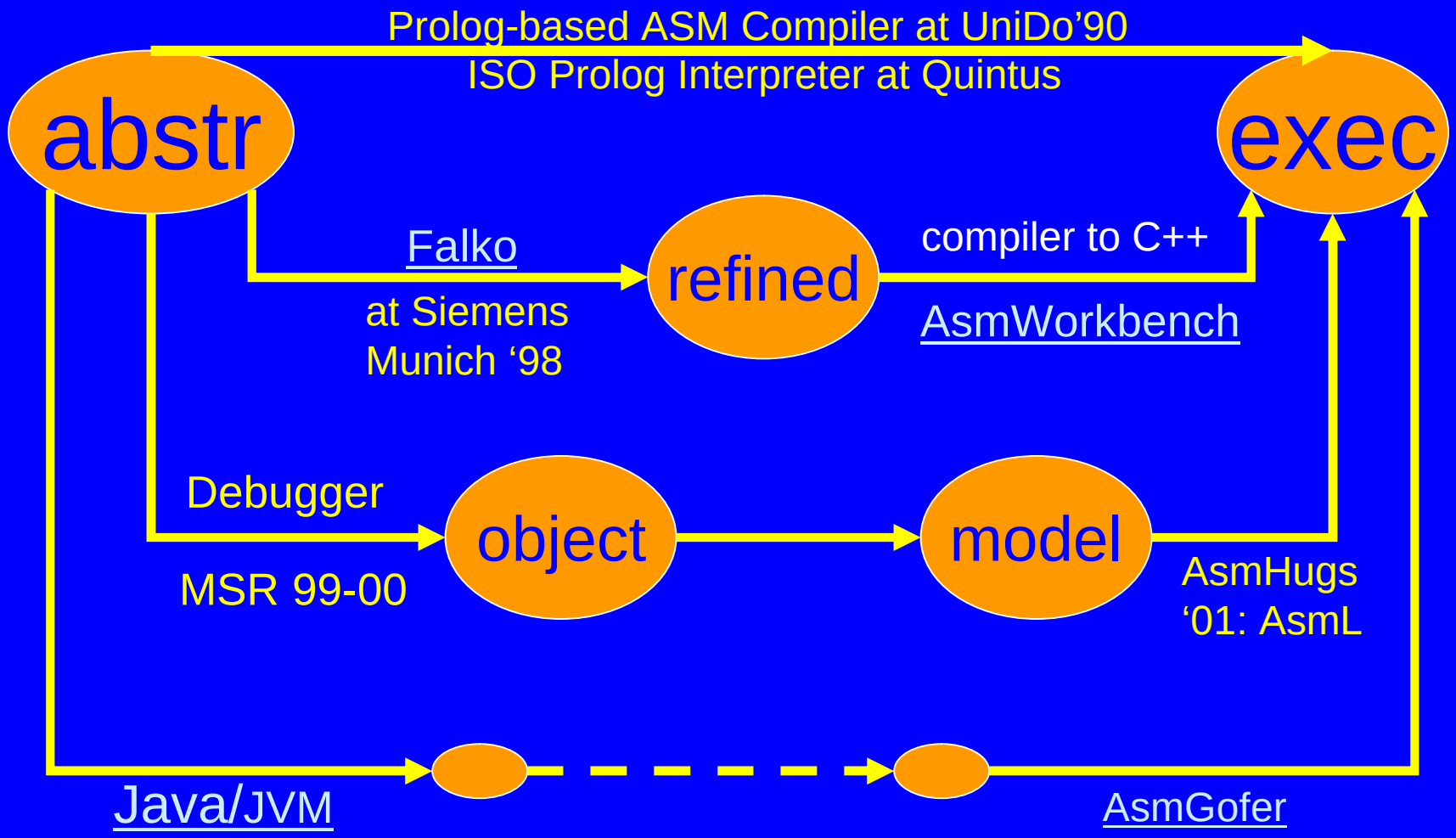
The practical benefits of ASMs

- The definition provides rigorous yet practical (process oriented & easy to use) semantics for pseudocode on arbitrary data structures, which
  - has clear notion of state & state transition,
  - can be made executable (in various natural ways, compatible with cross-language interoperable implementations, e.g. in .NET),
  - is tunable to desired level of abstraction with
    - well defined interfaces (rigor without formal overkill)
    - a most general refinement notion supporting a method of stepwise development by traceable links bw different abstraction levels

  The parallel ASM execution model
  - easens specification of “macro” steps (refinement & modularization)
  - avoids unnecessary sequentialization of independent actions
  - easens parallel/distributed implementations

Validation of ASM behavior

Make models executable by **implementing the abstractions**



Examples for Design & Verification of ASM Hierarchies

Architectures: Pipelining of RISC DLX: model checking, PVS verification

Control Systems: Production Cell (model checked), Steam Boiler (refinements to C++ code) Light Control (executable requirements model)

Compiler correctness

ISO Prolog to WAM: 12 refinement steps, KIV verified

backtracking, structure of predicates, structure of clauses, structure of terms & substitution, optimizations

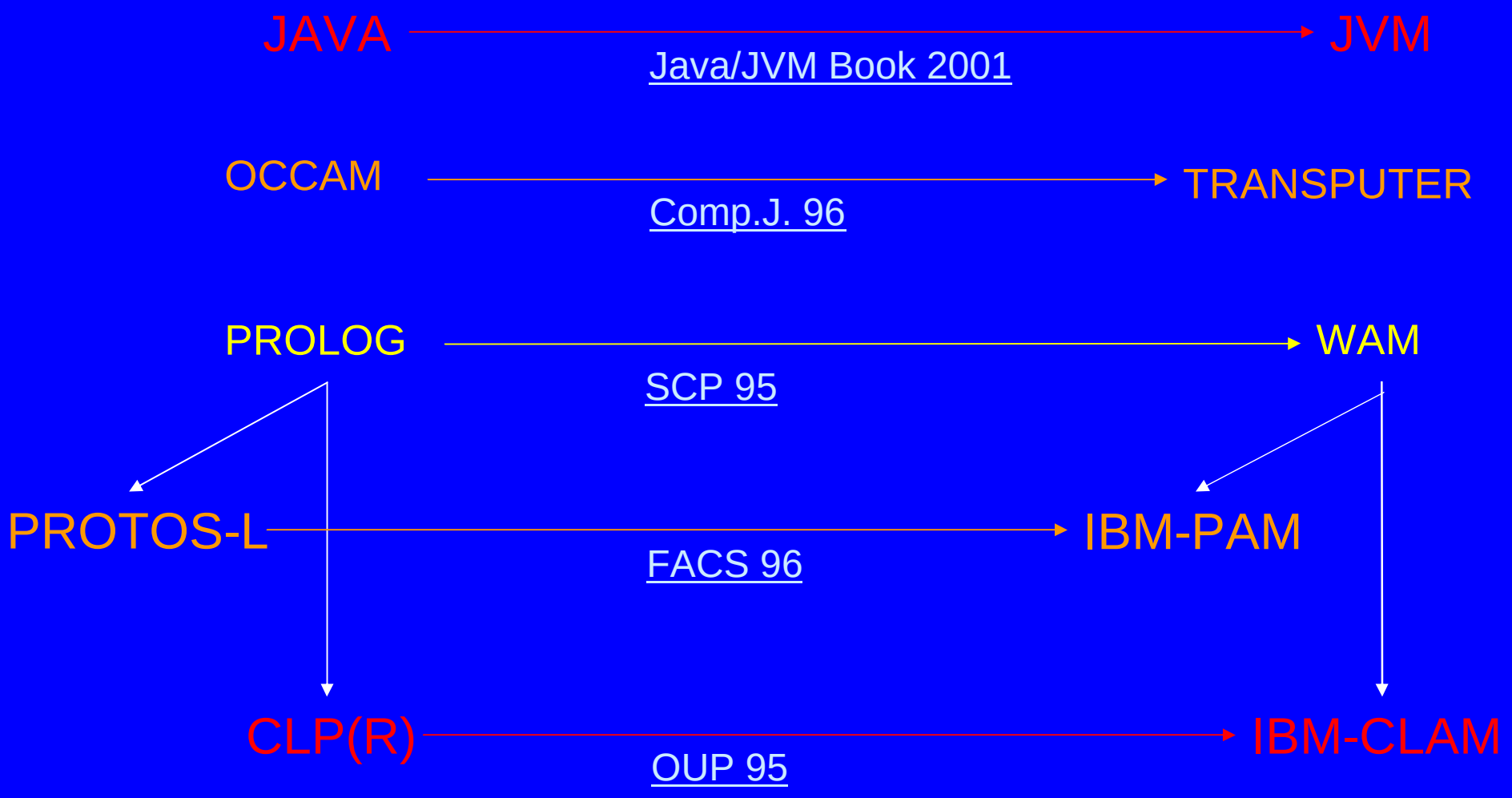
Occam to Transputer :15 models exhibiting channels, sequentialization of parallel procedures, pgm ctrl structure, env, transputer datapath and workspace, relocatable code (relative instr addresses & resolving labels)

Java to JVM: language and security driven decomposition into

5 horizontal sublanguage levels (imperative, modules, oo, exceptions, concurrency) and

4 vertical JVM levels for trustful execution, checking defensively at run time and diligently at link time, loading (modular compositional structuring)

Reusability of ASM Specifications and Verifications



Reuse of layered components (submachines) and of lemmas

References

ASM Book E. Börger, R. Stärk

Abstract State Machines

A Method for High-Level System Design and Analysis

Springer-Verlag 2003, see

http://www.di.unipi.it/AsmBook

ASM Case Study Book

R. Stärk, J. Schmid, E. Börger

Java and the Java Virtual Machine: Definition, Verification, Validation

Springer-Verlag 2001, see

http://www.inf.ethz.ch/~jbook

References

ASM Web Sites http://www.di.unipi.it/~boerger
http://www/eecs.umich.edu.gasm

ASM Survey E. Börger High Level System Design and Analysis using ASMs LNCS Vol. 1012 (1999), pp. 1-43

ASM History E. Börger The Origins and the Development of the ASM Method for High Level System Design and Analysis J. Universal Computer Science 8 (1) 2002

Original ASM Definition Y. Gurevich Evolving algebra 1993: Lipari guide Specification and Validation Methods (Ed.E. Börger) OUP 1995

B Method J.-R.Abrial The B-Book Cambridge University Press 1996