Egon Börger (Pisa)

The Abstract State Machines Method

for High-Level System Design and Analysis

An Introduction with an Application to Modeling Workflow Patterns

Università di Pisa, Dipartimento di Informatica, I-56127 Pisa, Italy
boerger@di.unipi.it

The question addressed in this talk

What characterizes the ASM method, a latecomer among the practical and scientifically well-founded systems engineering methods?

- We do not speak about special-purpose techniques, like static analysis, bytecode verification, model checking, run-time verification, etc., which draw their success from being tailored to particular types of problems.
- We focus on *wide-spectrum methods*, which assist system engineers in every aspect of an effectively controllable construction of reliable computer-based systems and thus bridge the gap between the two ends of system development:
  - *human understanding* and formulation of real-world problems
  - deployment of their solutions by *code-executing machines* on changing platforms

A coherent divide-and-conquer approach

The ASM method comes with a *simple practical framework, where in a coherent and uniform way the system engineer can*

- *systematically separate multiple concerns, concepts and techniques*, which are inherent in the large variety of system development activities,
- *freely choose for each task an appropriate combination of concepts and techniques* from the stock of engineering (including formal) methods, at the given level of abstraction and precision where the task occurs.

This allows one to coherently integrate into best sw engg practice:

- combination of design and analysis in a consistent way, integrating
  - *mathematical verification* by a variety of reasoning techniques, applicable to system models at different levels of precision
  - *experimental validation* of system behaviour through simulation (model-checking, run-time verification, testing) of rigorous models, at various levels of abstraction
- mathematical rigour to the degree needed for certifiable reliability

Ingredients: nothing new under the sun

- *abstract (richly structured) states*: abstract data types, algebraic specs
- *abstract instructions* for changing states: pseudo-code, VMs
- synchronous *parallel execution* model, including conditional multiple assignments: UNITY, COLD, B, etc.
- *locality principle*: programming languages
- *functional definitions*: mathematics and functional programming
- *declarative (axiomatic) definitions*: logic, declarative programming and specification languages
- *refinement* concept: generalizing method of Wirth/Dijkstra and numerous formal specification methods: Z, B, etc.
- *decomposition & hierarchy* concepts: automata theory, algebraic specifications, layered architectures,
- *function classification*: programming, Parnas' SCR method
- *verification* of model properties by proofs at various levels of precision
- *simulation* (experimental validation) by model execution

The three fundamental constituents of the ASM method

- notion of ASM: extension of FSM by Tarski structures as states
- ASM ground models: accurate descriptions of requirements at application-domain-determined abstraction level, providing authoritative reference for further system development activities:
  - *detailed design* via a series of design and coding decisions
  - *design evaluation* via analysis, including testing/inspection/review process, to certify *consistency, correctness, completeness* properties needed to establish and document desired degree of reliability
  - *system maintenance*, including requirements change management
- ASM refinements linking more detailed descriptions at successive development stages in an *organic and effectively maintainable chain of rigorous and coherent system models*. Refinement links must guarantee that ground model system properties are preserved via series of design decisions leading to code—and document this for maintenance (reuse and change management)

Notion of ASM: extend FSM states to Tarski structures

FSMs come with three characteristic restrictions:

```
$\textsc{MealyFsm}(in, out, \delta, \lambda) = \textbf{if } Defined(in) \textbf{ then}$
    $ctl\_state := \delta(ctl\_state, in)$
    $out := \lambda(ctl\_state, in)$
```

- *only three locations* read or updated: in, ctl_state, out,
- *only three special data types*: finite sets of
  - input/output symbols (letters of an alphabet)
  - abstract control states (labels/integers) representing bounded memory
- strict separation of input and output

ASMs withdraw those restrictions, permitting a machine in each step

- to read and update *arbitrarily many, possibly parameterized, locations whose values can be of arbitrary type*
- to have *arbitrary* conditions as *rule guard* (not only input definedness)

Structuring abstract state memory yields Tarski structures

Group subsets of data into *tables* (in logic: interpretation of functions)

- associating a value to each table entry $(l, (a_1, \ldots, a_n))$, called *location*
- $l(a_1, \ldots, a_n)$ denotes the value currently contained in the table entry $(l, (a_1, \ldots, a_n))$ (read: array variable)

Tarski structure $=$ a set of tables

NB. Tarski structures represent a most general notion of structure

- The structures of mathematics are Tarski structures
- The models of abstract data types are Tarski structures
- The object-oriented understanding of classes and class instances corresponds to Tarski structures

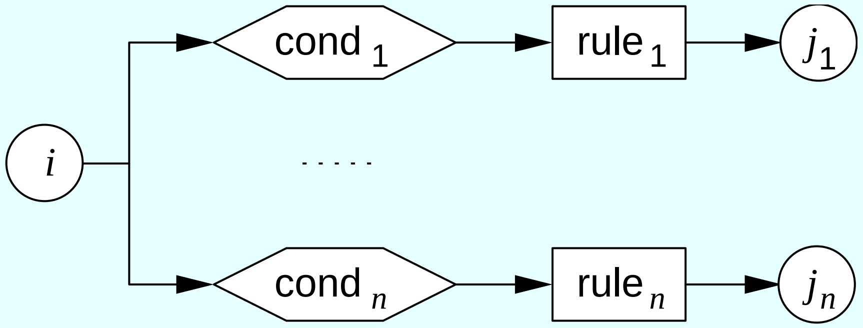
Thus FSMs operating over Tarski structures update tables:

- ASM $=$ finite set of rules **if** *Cond* **then** *Updates* where $Updates$ is a set of simultaneous assignments $f(t_1, \ldots, t_n) := t$
- in each step updates (consequently also rules) are *executed in parallel*

Control State ASMs

Control State ASM = ASM all of whose rules have the form

```
**if** $ctl\_state = i$ **and** $cond$ **then**
    $rule$
    $ctl\_state := j$
```



control-states $i, j, \ldots$ represent an overall system status (mode, phase), which allows the designer to

- *structure* the set of *states* into subsets
  – *visualize* this overall structure
- *refine* control-state transitions by control-state submachines (modules)

ExI: Sluice Gate Control (M. Jackson: Problem Frames)

Requirements Description:

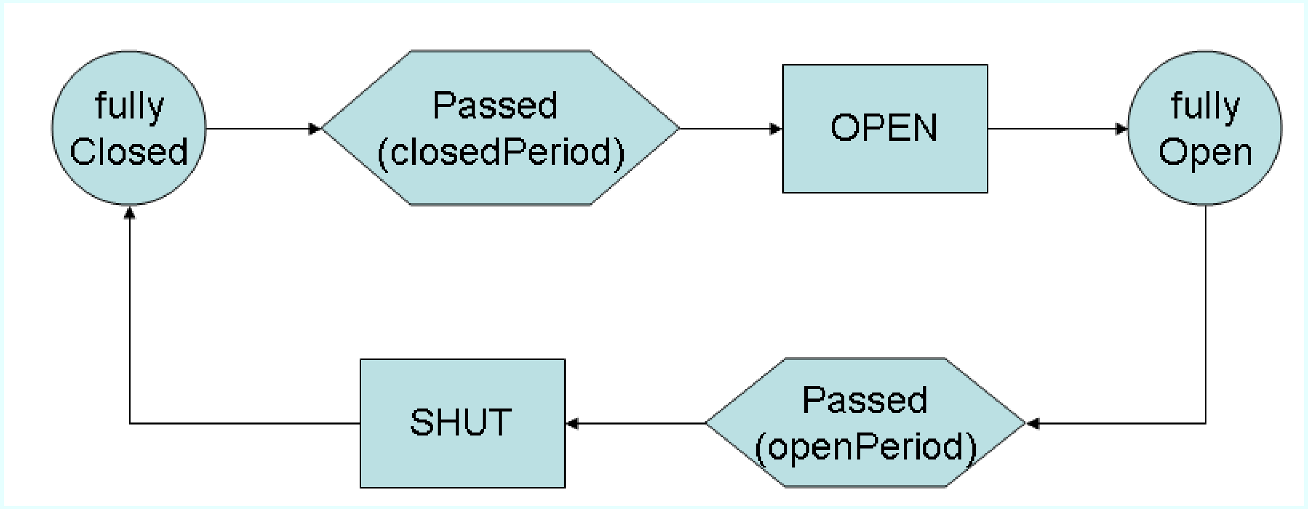
A small sluice, with a rising and falling gate, is used in a simple irrigation system. A computer system is needed to control the sluice gate: the *requirement* is that the gate should be held in the fully open position for ten minutes in every three hours and otherwise kept in the fully closed position.

The gate is opened and closed by rotating vertical screws. The screws are driven by a small *motor*, which can be controlled by clockwise, anticlockwise, on and off pulses.

There are *sensors* at the top and bottom of the gate travel; at the top it's fully open, at the bottom it's fully shut.

The *connection to the computer* consists of four pulse lines for motor control and two status lines for the gate sensors.

Ground Model to capture user requirements



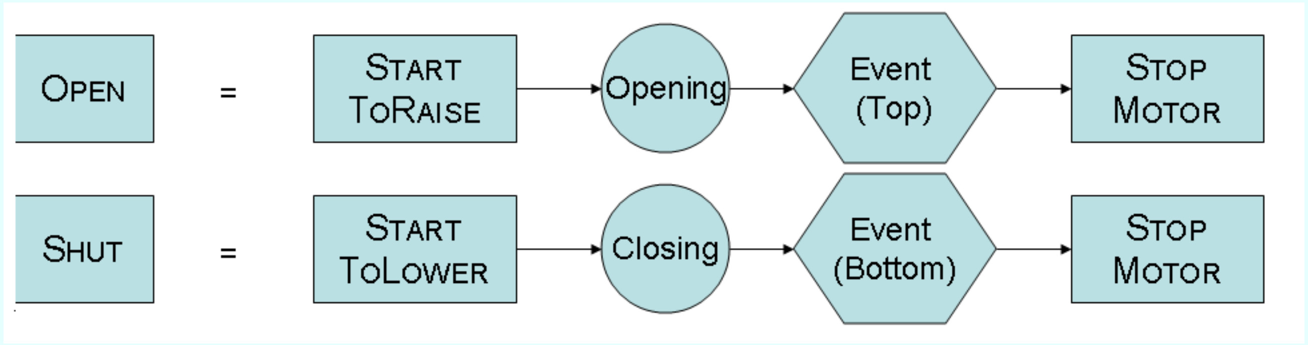
Ground model, displayed in FSM-style graphical notation, abstracts from

- *screws, motor, sensors, pulses*, reducing the device to switching from phase $fullyClosed$ to $fullyOpen$ when time $closedPeriod$ has passed, and back when $openPeriod$ has elapsed
- *timing model*, using monitored boolean-valued locations $Passed(openPeriod)$, $Passed(closedPeriod)$ with appropriate interpretation
- *motor actions* $\textsc{Open} = \textsc{Shut} = \mathbf{skip}$ (placeholder for refinement)

Copyright © 2007 Egon Börger (ER'07 Keynote)

Refinement SluiceGateMotorCtl defining motor actions

- controlled locs $motor : \{on, off\}$, $dir : \{clockwise, anticlockwise\}$
- Boolean-valued monitored locations $Event(Top)$, $Event(Bottom)$ with appropriate Input Locations Assumption (for correctness proof)



$$\text{StartToRaise} = (dir := clockwise,\ motor := on)$$

$$\text{StartToLower} = (dir := anticlockwise,\ motor := on)$$

$$\text{StopMotor} = (motor := off)$$

$$closedPeriod = period$$
$$-(StartToRaiseTime + OpeningTime + StopMotorTime)$$
$$-(StartToLowerTime + ClosingTime + StopMotorTime)$$

Refinement $\textsc{SluiceGate}$ defining status lines and pulses

Environment machine describing equipment actions when pulses appear

$$\begin{aligned}
\textsc{Pulses} = {} & \textbf{upon } \mathit{Event}(\mathit{Clockwise}) \textbf{ do } \mathit{dir} := \mathit{clockwise} \\
& \textbf{upon } \mathit{Event}(\mathit{AntiClockwise}) \textbf{ do } \mathit{dir} := \mathit{anticlockwise} \\
& \textbf{upon } \mathit{Event}(\mathit{MotorOn}) \textbf{ do } \mathit{motor} := \mathit{on} \\
& \textbf{upon } \mathit{Event}(\mathit{MotorOff}) \textbf{ do } \mathit{motor} := \mathit{off}
\end{aligned}$$

$\textsc{SluiceGateMotorCtl}$ with $\textsc{Emit}(\mathit{Pulse}(\ldots))$ submachines

$$\begin{aligned}
\textsc{StartToRaise} = {} & \textsc{Emit}(\mathit{Pulse}(\mathit{Clockwise})) \\
& \textsc{Emit}(\mathit{Pulse}(\mathit{MotorOn}))
\end{aligned}$$

$$\begin{aligned}
\textsc{StartToLower} = {} & \textsc{Emit}(\mathit{Pulse}(\mathit{AntiClockwise})) \\
& \textsc{Emit}(\mathit{Pulse}(\mathit{MotorOn}))
\end{aligned}$$

$$\textsc{StopMotor} = \textsc{Emit}(\mathit{Pulse}(\mathit{MotorOff}))$$

Proving correctness of the refinement

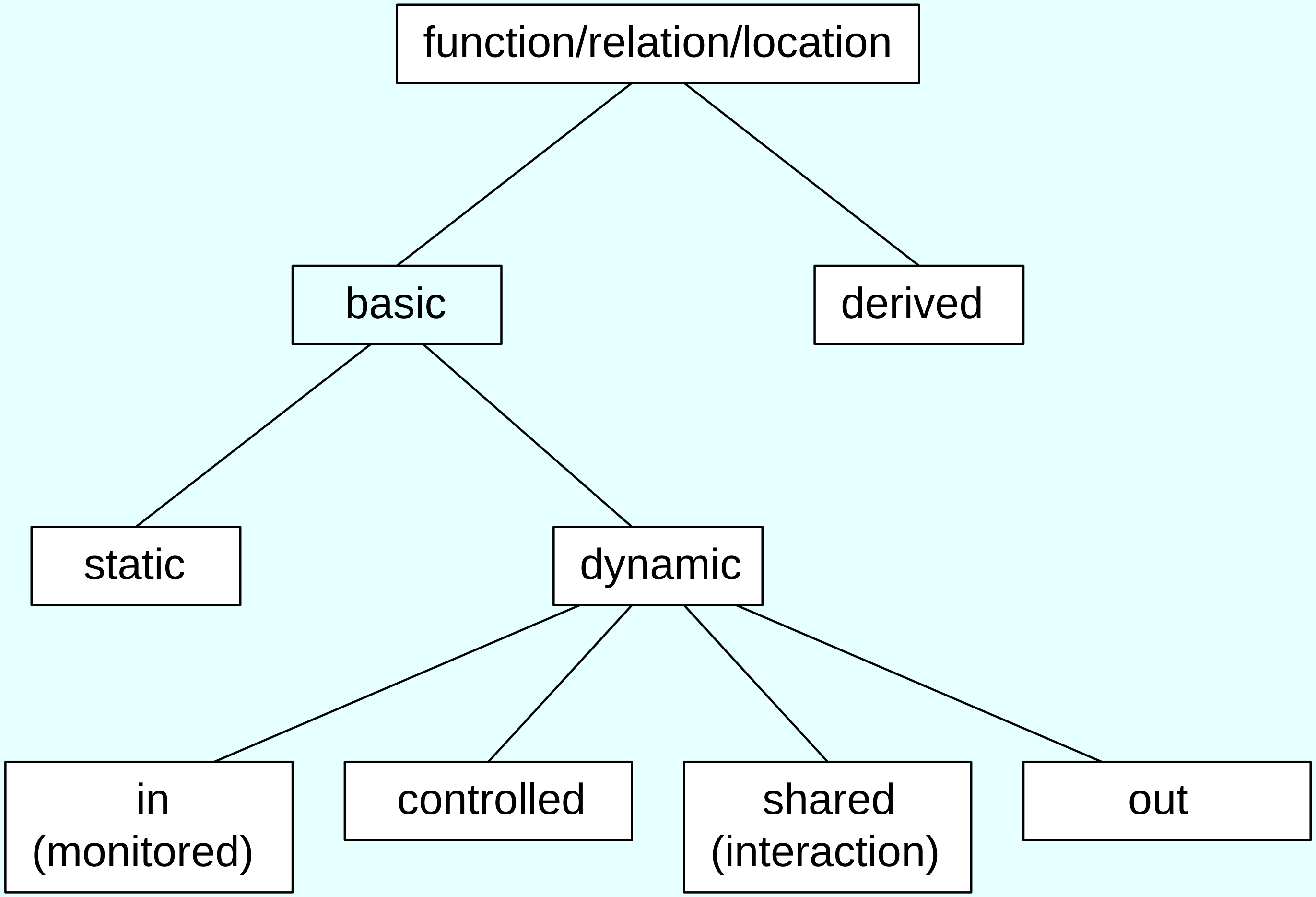
Correctness proof, which relates runs, relies upon:

- *Pulse Output Assumption*: each $\textsc{Emit}(Pulse(p))$ in $\textsc{SluiceGateCtl}$ yields $Event(p)$ to happen in the env machine
- *Input Locations Assumption* reinterpreted, reflecting that the information detected by the sensors arrives at $\textsc{SluiceGateCtl}$ as input $Event(Top), Event(Bottom)$ via two status lines
- Usual convention that events are consumed by triggering the rule guarded by them

NB. *Refinement type is (1,2)*, meaning that:

- every segment consisting of one step of $\textsc{SluiceGateMotorCtl}$ is refined by
- a segment of two corresponding steps of $\textsc{SluiceGate}$ consisting of
  - one step of software machine $\textsc{SluiceGateCtl}$
  - followed by one step of environment machine $\textsc{Pulses}$

Classification of locations/functions



supporting the separation of concerns: information hiding, data abstraction, modularization and stepwise refinement

Notation for non-determinism and parallelism

- selection functions (describing non-determinism) supported by dedicated notation for $rule(select\ \{x : \phi(x)\})$:

  choose x **with** ϕ **in** $rule$

  to execute $rule$ for one element x, which is arbitrarily chosen among those satisfying the selection criterion ϕ

- symmetric notation to enhance synchronous parallelism:

  forall x **with** ϕ **do** $rule$

  to execute $rule$ simultaneously for every element x satisfying the property ϕ

For *asynchronous multi-agent ASMs* it suffices to generalize the notion of run from sequences of moves (execution of rules) of just one basic ASM to partial orders of moves of multiple agents, each executing a basic ASM, subject to a natural coherence condition.

What are ground models?

Accurate blueprints of the to-be-implemented piece of real world

- *defining* 'the conceptual construct/the essence' of the software system (Brooks) *abstractly and rigorously*
  - at an application-problem-determined level of detailing (minimality)
  - formulated in application domain terms (precision)
  - authoritatively for the further development activities: design contract/process/evaluation and maintenance (simplicity)
- *grounding the design in reality* by justifying the definition as
  - *correct*: model elements reliably convey original intentions
  - *complete*: every semantically relevant feature is present (env,arch, domain knowledge), no gap in understanding of 'what to build'
  - *consistent*: conflicting objectives in requirements resolved
- epistemological problem: 'matching some significant part and aspect of an activity in the real world to the formal symbol manipulation that can be done by a program running on a computer' (Naur 1985)

Epistemological problem: ground model justification

- *Communication* problem: mediate between
  - sw designers & domain experts/customers for common understanding prior to coding of ‘precisely what to build’
  - problem domain & world of models, requiring
    - capability to calibrate degree of model precision to the problem
    - general data & operation framework and general interface concept (to represent system environments)
- *Verification method* problem: no infinite regress
  - no math. transition from informal to precise descriptions, BUT
  - inspection can provide evidence of direct correspondence bw ground model and reality the model has to capture (completeness, correctness, empirical interpretation of extra-logical terms)
  - domain-specific reasoning can check consistency issues
- *Validation* problem: need for repeatable experiments to validate (falsify) model behaviour (runtime verification and analysis, testing)

Variety of real-life ASM ground models (1)

- **industrial standards**: ground models for the standards of
  - OASIS for Business Process Execution Language for Web Services
  - ECMA for C#
  - ITU-T for SDL-2000
  - IEEE for VHDL93
  - ISO for Prolog
- **design, reengineering, testing of industrial systems**:
  - railway and mobile telephony network component software (at Siemens)
  - fire detection system in German coal mines
  - implementation of behavioral interface specifications on the .NET platform and conformence test of COM components (at Microsoft)
  - business systems interacting with intelligent devices (at SAP)
  - compiler testing and test case generation tools

Variety of ASM ground models and their refinements (2)

- **programmming languages**: definition and analysis of the semantics and the implementation for the major real-life programmming languages, among many others for example
  - SystemC
  - Java/JVM (including bytecode verifier)
  - domain-specific languages used at the Union Bank of Switzerland

  including the verification of numerous compilation schemes and compiler back-ends
- architectural design: verification (e.g. of pipelining schemes or of VHDL-based hardware design at Siemens), architecture/compiler co-exploration
- protocols: for authentication, cryptography, cache-coherence, routing-layers for distributed mobile ad hoc networks, group-membership etc.
- modeling e-commerce and web services (at SAP)

ASM Refinements (Reflecting Design Decisions)

- practice-oriented method to systematically separate, structure and document orthogonal design decisions, relating different system aspects and (system architect's to programmer's) views
- supports cost-effective system maintenance and management of system changes
- supports piecemeal system validation and verification techniques

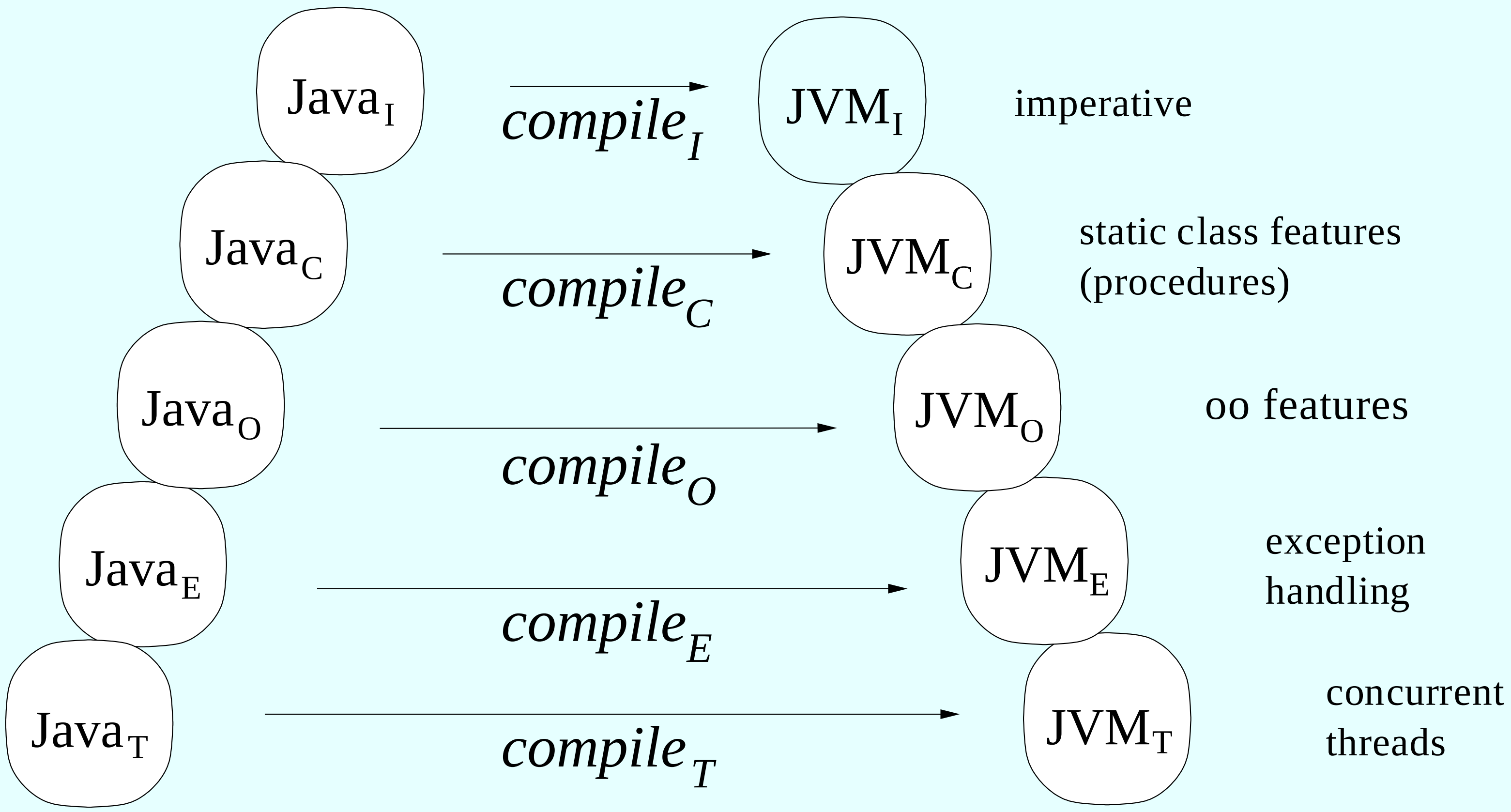
$$\begin{tikzcd}
\text{State } S \arrow[r, "\overbrace{\tau_1 \cdots \tau_m}^{m \text{ steps of } M}"] \arrow[d, leftrightarrow, "\equiv"'] & S' \arrow[d, leftrightarrow, "\equiv"] \\
\text{State } S^* \arrow[r, "\underbrace{\sigma_1 \cdots \sigma_n}_{n \text{ steps of } M^*}"'] & S^{*\prime}
\end{tikzcd}$$

With an equivalence notion $\equiv$ between data in locations of interest in corresponding states.

The parameters for defining an ASM refinement step

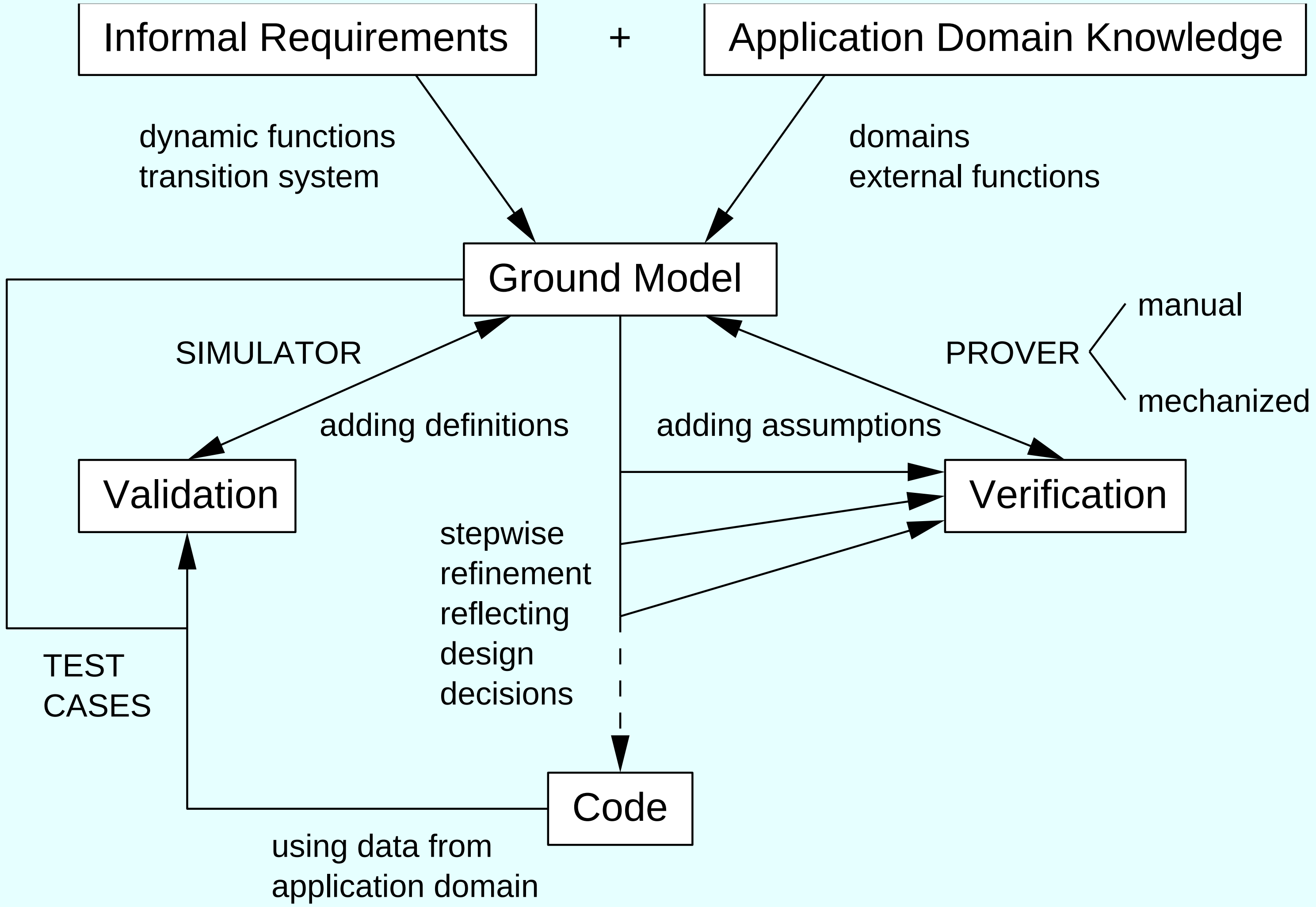
- a notion of *refined state*
- a notion of *states of interest* and of *correspondence* between M-states S and M^*-states S^* of interest, including usually initial/final states (if there are any)
- a notion of abstract *computation segments* $\tau_1, \ldots, \tau_m$, where each τ_i represents a single M-step, and of corresponding refined computation segments $\sigma_1, \ldots, \sigma_n$, of single M^*-steps σ_j, which in given runs lead from corresponding states of interest to (usually the next) corresponding states of interest (the resulting diagrams are called (m, n)-diagrams and the refinements (m, n)-refinements)
- a notion of *locations of interest* and of *corresponding locations*, i.e. pairs of (possibly sets of) locations one wants to relate in corresponding states
- a notion of *equivalence* $\equiv$ of the data in the locations of interest

Language-Oriented Refinement of Java/JVM into Layers



Layers are *conservative extensions* of each other and thus support componentwise design and analysis (validation & verification). Combination with an appropriate parameterization provides an *orthogonal treatment of language constructs* ("instructionwise").

Models and methods in an ASM-based development process



ASM Analysis Techniques (Validation and Verification)

Practitioner supported to analyze ASM models by reasoning and experimentation at the appropriate degree of detail, separating

- orthogonal design decisions and complementary methods: abstract operational vs declarative/functional/axiomatic, state- vs event-based
- design from analysis (definition from proof)
- validation (by simulation) from verification (by reasoning)
  - e.g. ASM Workbench (ML-based, DelCastillo 2000), AsmGofer (Gofer-based, Schmid 1999), XASM (C-based, Anlauff 2001), AsmL (.NET-based, MSR 2001), CoreASM (Glässer et al. 2005)
- verification levels (degrees of detail)
  - reasoning for human inspection (design justification)
  - rule based reasoning systems (e.g. Stärk's Logic for ASMs)
  - interactive proof systems, e.g. KIV, PVS, Isabelle, AsmPTP
  - automatic tools: model checkers, automatic theorem provers

References

E. Börger and R. F. Stärk: Abstract State Machines Springer 2003. pp.X+438. See *http://www.di.unipi.it/AsmBook/*

R. Stärk, J. Schmid, E. Börger: Java and the Java Virtual Machine. Definition, Verification, Validation. Springer-Verlag, 2001. See *http://www.inf.ethz.ch/personal/staerk/jbook*

E. Börger: Construction and Analysis of Ground Models and their Refinements as a Foundation for Validating Computer Based Systems. Formal Aspects of Computing J. 2006

E. Börger: *The ASM* Refinement *Method* Formal Aspects of Computing 15 (2003), 237-257

- G. Schellhorn: *Verification of ASM Refinements Using Generalized Forward Simulation* JUCS 7.11 (2001) 952–979
- G. Schellhorn: *ASM Refinement and Generalizations of Forward Simulation in Data Refinement: A Comparison* TCS 336/2-3 (2005) 403-436

Egon Börger (Pisa)

Modeling Workflow Patterns

Università di Pisa, Dipartimento di Informatica, I-56127 Pisa, Italy
boerger@di.unipi.it

The sources

- Wil M.P. van der Aalst and A.H.M. ter Hofstede and B. Kiepuszewski and A.P. Barros: Workflow Patterns
  Distributed and Parallel Databases 13 (4) (2003) 5-51
  – Definition of 20 individually named patterns
- N. Russel and A.H.M. ter Hofstede and Wil M.P. van der Aalst and N. Mulyar: Workflow Control-Flow Patterns. A Revised View
  BPM-06-22, BPMCenter.org, July 2006
  – Definition of 23 additional individually named patterns

Used "to identify comprehensive workflow functionality ... for an indepth comparison of a number of commercially available workflow management systems" and "*to systematically address workflow requirements, from basic to complex*".

But: descriptions are partly ambiguous or incomplete or overspecified by implementation features (belonging to coloured Petri nets)

The pattern classification proposed in op.cit.

- Control Flow (Basic and Additional)
- Advanced Branching and Synchronization
- Structural
- Multiple Instances
- Cancellation
- State-Based

Reasons for choice of patterns unclear

- 20 in 2003, 43 in 2006, doubling every 3 years?

We identify parameters turning numerous patterns into instances of one generic pattern. This provides a better understanding and classification:

- 4 standard sequential programming concepts applied to control flows
  – *sequentialization, iteration, begin/termination, choice*
- 4 parallel control flow patterns
  – *splitting, merging, interleaving, trigger*

Underlying basic concepts

What is the meaning of:

- *activity*
- *process* (often used as synonym for activity?!)
- *thread*
  - being active, enabled, completed
- *points in the workflow process* where some action is performed

Apparently it suffices to consider

- activities/processes as high-level executable programs (ASMs)
- threads as agents that execute ASMs and may be active, enabled, etc. or have completed their run
- pattern actions, at points in the workflow process, as described by pseudo-code (ASMs)

Parallel Split

A point in the workflow process where a single thread of control splits into *multiple threads* of control which can be executed in parallel, thus allowing activities to be *executed simultaneously or in any order*.

The parameters:

- $Activity$: static ('known at design-time') or dynamic ('known at run-time') set
- $Thread$: dynamic set, extendable using new function
- parallelism: left open whether synchronous, interleaved, asynchronous. Therefore we use an abstract machine $\textsc{TriggerExec}(t, a)$, refinable in various ways, to trigger thread t to execute activity a

```
$\textsc{ParSplit}(Activity, Thread, \textsc{TriggerExec}) =$
  $\mathbf{forall}\ a \in Activity$
    $\mathbf{let}\ t = new(Thread)\ \mathbf{in}\ \textsc{TriggerExec}(t, a)$
```

Instances of Parallel Split: Synchronous Parallelism

Synchronous parallelism is the default parallelism of ASMs

- no multiple agents need to be mentioned and *Threads* can be skipped

```
$\textsc{SyncParSplit}(Activity, \textsc{TriggerExec})$
    $= \textsc{ParSplit}(Activity, Thread, \textsc{TriggerExec})$
    $= \textbf{forall } a \in Activity$
        $\textbf{let } t = new(Thread) \textbf{ in } \textsc{TriggerExec}(t, a)$
    $= \textbf{forall } a \in Activity \ \ \textsc{TriggerExec}(a)$
```

Instances of Parallel Split: multiple Instances of act

- 'multiple instances (of one activity) without synchronization':

  Within the context of a single case (i.e., workflow instance) multiple instances of an activity can be created, i.e. there is a facility to *spawn off new threads* of control. Each of these threads of control is *independent* of other threads. Moreover, there is *no need to synchronize* these threads.

Instantiation: $Activity = multiSet(act, mult)$

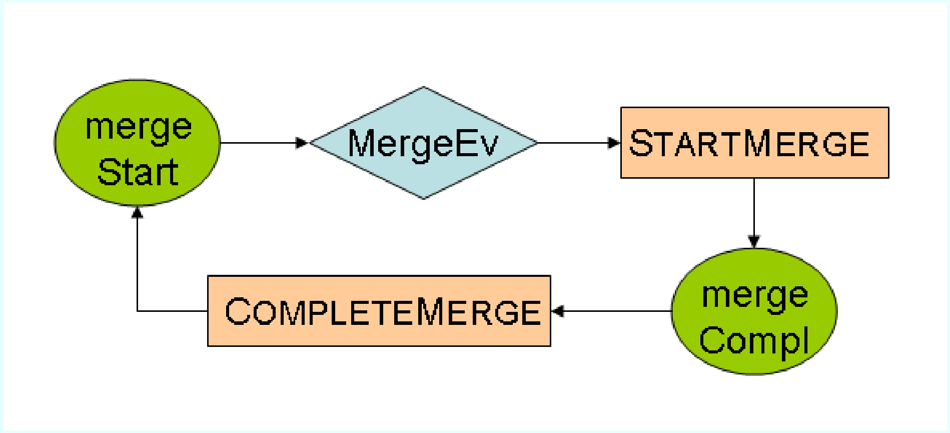
$\textsc{MultInstWithoutSync}(act, mult, Thread, \textsc{TriggerExec}) =$
$\textsc{ParSplit}(multiSet(act, mult), Thread, \textsc{TriggerExec})$

- Variations by design-time/run-time knowledge of *how many instances* are to be created
  – matter of declaring $mult$itude as static or dynamic

Basic Merge Pattern MERGE

A point in the workflow process where multiple parallel subprocesses/activities *converge* into one single thread of control ... *once ... completed some other activity* needs to be *started*.

- specific convergence action upon $MergeEv$ent (at merge phase start)
- completion (later) by some further actions



NB. Make explicit also the set $Exec(a)$ of agents that execute a: their runs are merged, not the activities.

Control State ASM code for $\textsc{Merge}$

```
$\textsc{Merge}$
    $(Activity, Exec, MergeEv, \textsc{StartMerge}, \textsc{CompleteMerge}) =$

        **if** $ctl\_state = mergeStart$ **and** $MergeEv(Exec)$ **then**
            $\textsc{StartMerge}$
            $ctl\_state := mergeCompl$
        **if** $ctl\_state = mergeCompl$ **then**
            $\textsc{CompleteMerge}$
            $ctl\_state := mergeStart$
```

NB. *Control state ASMs* extend FSMs by simultaneous elaboration (reading/writing) of *arbitrarily many locations storing values of arbitrary type*

‘Discriminator’ instantiation of $\textsc{Merge}$

... a point in a workflow process that waits for one of the incoming branches to complete before activating the subsequent activity. From that moment on it waits for all remaining branches to complete and “ignores” them. Once all incoming branches have been triggered, it $\textsc{Reset}$s itself so that it can be triggered again...

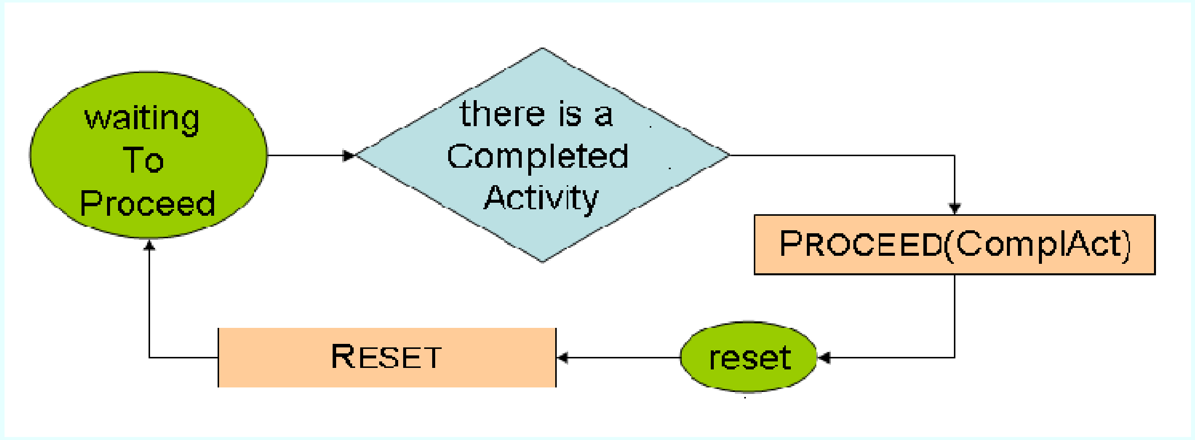
Qu: what happens when multiple branches complete simultaneously?

$$\textsc{Discriminator}(\mathit{Activity}, \mathit{Exec}, \mathit{Completed}, \textsc{Proceed}, \textsc{Reset})$$
$$=$$
$$\textsc{Merge}(\mathit{Activity}, \mathit{Exec}, \mathit{MergeEv}, \textsc{Proceed}(\mathit{ComplAct}), \textsc{Reset})$$

$$\textbf{where } \mathit{MergeEv} = \mid \mathit{ComplAct} \mid \geq \mathit{threshold}$$
$$\mathit{ComplAct} =$$
$$\{a \in \mathit{Activity} \mid \mathit{Completed}(a,t) \ \textbf{forsome } t \in \mathit{Exec}(a)\}$$

- Pattern presents the three aspects of $\textsc{Merge}$:
  – duration, threshold synchronization, othw no synchronization

Discriminator with Threshold Variations



- *Structured N-out-of-M Join*
  instantiate DISCRIMINATOR by $\mathit{threshold} = N$ and $M = \mid \mathit{Activity} \mid$

- *Generalized AND-Join*
  instantiate STRUCTURED N-OUT-OF-M JOIN by $N = M$

Discriminator Variations with Cancelling

Triggering the discriminator (join) also cancels *the execution* (Sic) of all of the other incoming branches and resets the construct

Two versions are considered in op.cit., which turn out to be variations of two discriminator patterns defined above:

- *Cancelling Discriminator*: a refinement of DISCRIMINATOR
- *Cancelling N-out-of-M Join*: a refinement of STRUCTURED N-OUT-OF-M JOIN

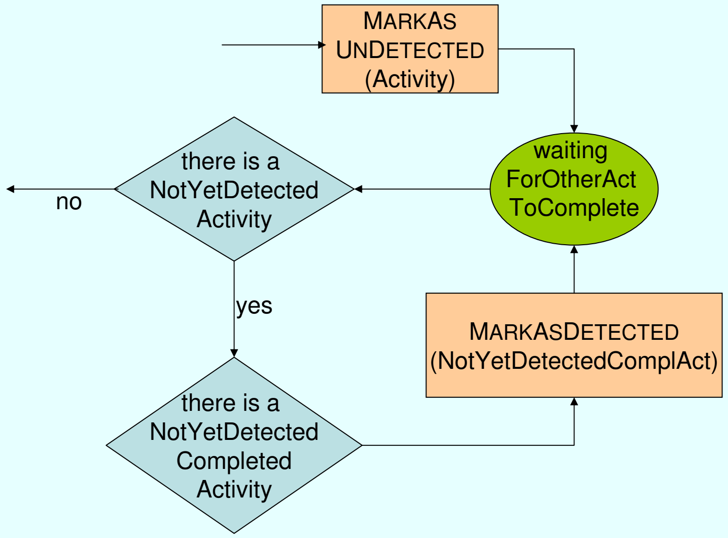
It suffices to refine RESET by a CANCEL concept:

$$\text{RESET} = \textbf{forall } a \in \mathit{Activity} \setminus \mathit{ComplAct} \;\; \text{CANCEL}(\mathit{Exec}(a))$$

NB. The CANCEL machine deliberately remains abstract

'Structured Discriminator' Variation by Durative RESET

- RESET is refined as durative action of waiting for other activities to complete: check which activities have not yet been detected as completed until every activity has been detected


MARKAS UNDETECTED (Activity)
waiting ForOtherAct ToComplete
there is a NotYetDetected Activity
no
yes
there is a NotYetDetected Completed Activity
MARKASDETECTED (NotYetDetectedComplAct)

$$\text{MARKASDETECTED}(A) = \textbf{forall } a \in A \ \mathit{NotYetDetected}(a) := \mathit{false}$$

StructuredDiscriminatorReset =

```
if $mode = init$ then
    MarkAsUnDetected($Activity$)
    $mode := waitingForOtherActToComplete$
if $mode = waitingForOtherActToComplete$ then
    if $NotYetDetected \neq \emptyset$
    then let $A = ComplAct \cap NotYetDetected$
        if $A \neq \emptyset$ then MarkAsDetected($A$)
    else $mode := exit$
where
  MarkAsDetected($A$) =
    forall $a \in A$ $NotYetDetected(a) := false$
  MarkAsUnDetected($A$) =
    forall $a \in A$ $NotYetDetected(a) := true$
```

Blocking variations of Discriminator

Blocking Discriminator, *Blocking N-out-of-M Join* are defined in op.cit. by adding the following requirement:

> Subsequent enablements of incoming branches are blocked until the discriminator (join) has reset.

It suffices to *instantiate each discriminator round for* fstCompleted, the first $t \in Exec(a)$ that $Completed(a, t)$, to block the subsequent executions $t' \in Exec(a)$ of a.

- delete $fstCompleted$ from $Exec(a)$ at the end of each round: a refinement of $\textsc{StructuredDiscriminatorReset}$ to $\textsc{BlockStructDiscrReset}$ by adding the following update in the last **else** branch:

```
$\mathbf{forall}\ a \in Activity$
  $\mathbf{let}\ f = fstCompleted(\{t \in Exec(a) \mid Completed(a, t)\})$
    $\textsc{Delete}(f, Exec(a))$
```

Instantiating MERGE to Synchronizing Merge

A point in the workflow process where multiple paths converge into one single thread. If more than one path is taken, *synchronization of the active threads* needs to take place. If only one path is taken, the alternative branches should reconverge without synchronization ... assumption ... that a branch that has already been activated, cannot be activated again while the merge is still waiting for other branches to complete ... the thread of control is passed to the subsequent branch when each active incoming branch has been enabled.

- the merge event is instantiated to a concept of all active threads being SyncEnabled, generalizing the threshold property in the discriminator pattern instances
- the dynamic set $Exec(a)$ is assumed to contain for each merge round at most one thread (which may be $Active$)
- what happens to non-active branches when multiple active threads synchronize via CONVERGE? Should they RECONVERGE?

SynchronizingMerge

$$\textsc{SynchronizingMerge}(Activity, Exec, \mathsf{Active}, SyncEnabled, \textsc{Converge}, \textsc{Reconverge}) =$$

$$\textsc{Merge}(Activity, Exec, MergeEv, \textsc{Converge}(Active), \textsc{Reconverge}(Activity \setminus Active))$$

where

$$MergeEv = \textbf{forsome } a \in Activity\ Active(Exec(a)) \textbf{ and } \textbf{forall } a \in Activity \textbf{ if } Active(Exec(a)) \textbf{ then } SyncEnabled(Exec(a))$$

- assuming at most one thread per activity per round
  - $\textbf{forall } a \in Activity \mid Exec(a) \mid \leq 1$

Variations of syncNumber in SynchronizingMerge

Determination of how many branches require synchronization is made on the basis of information locally available to the merge construct. This may be communicated directly to the merge by the preceding diverging construct or alternatively it can be determined on the basis of local data such as the threads of control arriving at the merge.

$$\textsc{AcyclSynchrMerge} = \textsc{SynchronizingMerge}\ \mathbf{where}$$
$$MergeEv = \\ \mid \{a \mid Active(Exec(a))\ \mathbf{and}\ SyncEnabled(Exec(a))\} \mid \\ \geq syncNumber$$

Variations depend on how *syncNumber* is defined

$$\textsc{Synchronizer} = \textsc{SynchronizingMerge}\ \mathbf{where}$$
$$syncNumber = \mid Activity \mid$$
$$\textsc{Reconverge} = \mathbf{skip}$$ all branches are synchronized!

‘Simple’ (Selection) Instances of $\textsc{Merge}$

Case with unique activity, without need for synchronization:

SimpleMerge ... two or more alternative branches come together without synchronization. It is an assumption of this pattern that none of the alternative branches is ever executed in parallel.

Case with multiple activities, synchronization by selecting one:

MultiMerge (also called RelaxSimpleMerge)... two or more branches reconverge without synchronization. If more than one branch gets activated, possibly concurrently, the activity following the merge is started for every activation of every incoming branch.

$\textsc{SelectMerge} = \textsc{Merge}$ **where**

$mergeCompl = mergeStart \quad \textsc{CompleteMerge} = \textbf{skip}$

$\textsc{StartMerge} = \textsc{Converge}(select(Activity))$

$\textsc{SimpleMerge} = \textsc{SelectMerge}$ **where** $select = id$

An Instance of $\textsc{SelectMerge}$: Thread Merge

At a given point in a process, a ... number of execution threads in a single branch of the same process instance should be merged together into a single thread of execution

$\textsc{ThreadMerge}(\ldots, MergeEnabled, mergeNo) =$
$\quad \textsc{SelectMerge}(\ldots)$

where

$Activity = \{t \mid t \text{ executes the given branch activity}\}$

$MergeEv = (\mid \{t \in Activity \mid MergeEnabled(t)\} \mid = mergeNo)$

$\mid select(Activity) \mid = mergeNo$

- what if more than $mergeNo$ threads are $MergeEnabled$?
- *Design/Run-Time Knowledge* variations
  - a question of declaring $mergeNo$ as “nominated” (static? derived?) or “not known until run-time” (dynamic)

Patterns to Join Previously Split Multiple Instances

Static/Dynamic (Cancelling) N-out-of-M Join for Multiple Instances:
Link $\textsc{ParSplit}$ for 'multiple instances without sync' with $\textsc{Merge}$:

... once all (resp. N out of $mult = M$) instances are completed some other activity needs to be started.

$$
\begin{aligned}
&\textsc{MultInstNMJoin}\\
&\quad(act, mult, Thread, Completed, \textsc{TrigExec}, \textsc{Proceed}, N)\\
&= \textsc{Merge}(multiSet(act, mult), -, true, \textsc{Start}, \textsc{Complete})\\
&\quad\textbf{where}\\
&\qquad \textsc{Start} =\\
&\qquad\quad \textsc{MultInstWithoutSync}(act, mult, Thread, \textsc{TrigExec})\\
&\qquad \textsc{Complete} = \textbf{if } CompletionEv \textbf{ then } \textsc{Proceed}\\
&\qquad CompletionEv = (\mid \{t \in Thread \mid Completed(t, act)\} \mid \geq N)
\end{aligned}
$$

- variations depending on declaration/definition of $mult$ (static/dynamic) and on inclusion of $\textsc{Cancel}$ submachine

Instances of $\textsc{MultInstNMJoin}$ (1)

Multiple Instances With a Priori Design Time Knowledge:

$\textsc{MultInstNMJoin}(\ldots, \mid \mathit{Thread}(\mathit{act}) \mid)$

Multiple Instances With a Priori Run Time Knowledge: multitude of instances of a given activity for a given case varies and may depend on characteristics of the case or availability of resources, but is known at some stage during runtime, before the instances of that activity have to be created.

$\textsc{MultInstNMJoin}(\ldots, \mid \mathit{Thread}(\mathit{act}) \mid)$
where mult is defined as dynamic

Instances of $\textsc{MultInstNMJoin}$ (2)

Multiple Instances Without a Priori Run Time Knowledge: dto but

> the number of instances of a given activity for a given case is not known during design time, nor is it known at any stage during runtime, before the instances of that activity have to be created ... at any time, whilst instances are running, it is possible for additional instances to be initiated

This means that as part of the execution of a $\textsc{Run}(t, act)$, it is allowed that the set $Thread(act)$ may grow by new agents t' to $\textsc{Run}(t', act)$, all of which however will be synchronized when $Completed$.

Instances of MultInstNMJoin (3)

Complete Multiple Instance Activity

It is necessary to synchronize the instances at completion before any subsequent activities can be triggered. During the course of execution, it is possible that the activity needs to be forcibly completed such that any remaining instances are withdrawn and the thread of control is passed to subsequent activities.

This is captured by adding in MultInstAPrioriDesignKnowl to the Complete submachine the following machine:

```
if $Event(ForcedCompletion)$ then
    forall $a \in (Thread(act) \setminus Completed)$ do Cancel($a$)
    Proceed
```

Classification of MERGE Instantiations

- DISCRIMINATOR versions, refined by adding
  - threshold variations: one, N-out-of-M, all
  - CANCELling not synchronized activities upon RESETting
  - durative RESET, waiting for not synchronized activities to complete
  - blocking activity threads per round, deleting *fstCompleted* thread(s) (depending on threshold variations) from $Exec(a)$ upon RESETting
- SYNCHRONIZINGMERGE versions, refined by syncNumber variations
  - ACYCLSYNCHRMERGE
  - SYNCHRONIZER
- selection variations, named without synchronization or ‘simple’
  - SIMPLEMERGE selecting the unique one
  - SELECTMERGE selecting one out of many
  - THREADMERGE selecting *mergeNo* many (see SYNCHRONIZER)
- MULTINSTNMJOIN versions joining previously split *mult* instances
  - variations depending on definition of *mult* and CANCEL

Lesson: Use Standards to Express Pattern Variety

- Doable:
  - 23 patterns formalized in BPMN and BPEL (S. A. White 2007)
  - 43 patterns formalized in a slightly extended sublanguage of BPMN (Grosskopf Master Thesis HNI 2007)
- Problem: Widely accepted standards, like BPMN, BPEL, UML2.0, are without precise semantics (except some proposed implementation to which the standard constructs are transformed)
  - Recent Formalization Proposals are Incomplete and Biased by Imposing Implementation Details, e.g. Petri Nets, YAWL, CSP

Proposal: Use ASMs for semantical ground model of standards that is

- *uniform* capturing commonalities/differences of BPMN, UML2.0, ...
- *extendable* (to easily capture new enriched patterns) *by feature-based modular approach* (construct-wise definition)
- *integrates event/data and control* at needed level of abstraction
  - ongoing work with Bernhard Thalheim

References

- E. Börger, *Modeling Workflow Patterns from First Principles*.
  - ER 2007 Conference Proceedings, LNCS 4801, 2007
- E. Börger and R. Stärk, *Abstract State Machines*.
  A Method for High-Level System Design and Analysis.
  - Springer-Verlag 2003.
    See http://www.di.unipi.it/AsmBook for downloadable material including slides for lecturing