A Critical Analysis of Workflow Patterns

Egon Börger

Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy boerger@di.unipi.it

Abstract. We report work in progress about a critical analysis of the 43 workflow patterns recently presented by the Business Process Modeling Center in [14]. We disambiguate the natural language pattern descriptions given there, complete them by explicitly addressing relevant features that were left open in [14], and replace the Petri net formalizations, provided there to further illustrate the patterns, by truly abstract yet rigorous models, avoiding to introduce details that pertain only to the description framework and not to the problems addressed by the patterns. We identify parameters which turn numerous of the analyzed workflow patterns into instances of one main pattern, thus streamlining the pattern classification proposed in [14].

1 Introduction

In [18], revised and extended in [14], 43 workflow pattern descriptions are formulated "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". The descriptions, which are used for an analysis of how and to which degree the patterns are supported in a dozen of commercial workflow and business process systems, are given in natural language and further illustrated by Coloured Petri Net schemes. However, the informal descriptions are partly ambiguous or incomplete, in so far as some important questions that seem to be semantically relevant for the investigated patterns are left open. Many of their Petri net formalizations, presented to further specify the natural language descriptions, involve specific solution details that appear to be dictated more by the needs of the Petri net framework than by the problem addressed by the pattern.

In this paper we analyse the pattern descriptions presented in [18,14], trying to make the underlying relevant questions and implicit parameters explicit and to turn them into a precise truly abstract form, without introducing specific details that do not pertain to the investigated problem. We use for this the ASM method, which allows one to tailor descriptions to exactly fit the level of detailing required by the discussion of the problem at hand. In fact the accurate high-level models we provide for the patterns are *ASM ground models*, in the sense defined in [2]. These ASM ground models express the patterns directly, starting from scratch, avoiding any encoding, as required by the authors of the YAWL language [17], which has been specifically defined to support an encoding-free description of such patterns. We invite the reader to compare the ASM ground models with other pattern formalizations in the literature, e.g. using UML activity diagrams [12,10] or YAWL or Petri nets [16,11], the π-calculus [13], the process-calculus-based orchestration language Orc [9]. For reasons of space we cannot perform a detailed comparative evaluation here. The ASM definitions we provide are focussed on simple models that result from a direct analyis of the natural-language pattern descriptions and can be objectively checked to faithfully reflect the intended meaning. To easen such a ground model validation we esssentially follow the order of presentation adopted in [14], althoug this results in some repetitions given that in op.cit. some patterns are discussed in more than one place.

We identify some parameters, by which numerous patterns are turned into instances of one generic pattern. This streamlines the classification proposed in [18,14], from where all the descriptions quoted below are taken, turning the 43 patterns into a set of 13 basic patterns with simple refinements. 7 of these 13 basic patterns turn out to be versions of standard programming constructs, namely sequentialization, iteration (cycles), termination, cancellation, choice (selection), parallel split, interleaving.

We view the mathematically precise abstract definitions we propose here, although they are intendedly not formalized within any particular workflow specification or logic language, as an arguably safe basis for a further detailing of the patterns to executable versions, using the stepwise refinement technique described in [3] that allows one to justify and document on-the-fly that the code conforms to the specification.

This work extends the pattern description scheme outlined in [5] and adopted in [1]. The use of ASMs allows one to express the dynamics of abstract state changes by a suitable combination of operational and declarative language elements. This provides a high-level, both state-based and process-oriented view of workflow patterns, where the behavioral interface is defined through pattern actions performed by submachines that remain largely abstract, due to the intention to leave the design space open for further refinements to concrete pattern instantiations. Most of what we use below to model workflow patterns by ASMs is self-explanatory, given the semantically well-founded pseudo-code character of ASMs, an extension of Finite State Machines (FSMs) by a general notion of state. For a recent tutorial introduction into the ASM method for high-level system design and analysis see [4], for a textbook presentation see the AsmBook [8]. Note that due to the distributed nature of many of the patterns, we use distributed ASMs.

We make no attempt here to provide a detailed analysis of the basic concepts of *activity, thread, process*, of their being *active, enabled, complete* etc., which are used in [14] without further explanations. It seems to suffice for the present purpose to consider an activity or process as some form of high-level executable program, which we represent here as ASMs. Threads are considered as agents that execute activities. Thus an *active activity*, for example, is one whose executing agent is active, etc.

2 Basic Control Flow Patterns

These patterns capture elementary aspects of process control.

2.1 Sequence Pattern

"An activity in a workflow is enabled after the completion of another activity in the same process".

One among many ways to formalize this is to use control-state ASMs, which offer through final and initial states a natural way to reflect the completion and the beginning of an activity. Control state ASMs are generalized FSMs where all the instructions have the form $\textsc{Fsm}(i, \textbf{if } cond \textbf{ then } rule, j)$, standing for the following ASM rule (where *rule* is supposed to also be an ASM rule):

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

To display such rules we use the standard graphical notation for FSMs, as for example in Fig. 1. If one wants to hide those initial and final control states, one can use the **seq**-operator defined in [7] for composing an ASM A_1 **seq** A_2 out of component ASMs A_i $(i = 1, 2)$.

$\textsc{Sequence}(A_1, A_2) \;=\; A_1$ **seq** A_2

2.2 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."

We capture the parallel activities by a set parameter *Activity*. It may be declared as static or as dynamic, thus providing for both static instantiations and for dynamic ones, whether as known at design time or as known only at the moment of executing the parallel split. The above description leaves also the exact nature of the underlying parallelism unspecified. It may come

as an interleaving execution or as a simultaneous synchronous or asynchronous execution. This is captured by creating for each $a \in \mathit{Activity}$ a *new* thread to execute a. We assume each application of the function *new* to a set to provide a fresh element in this set (here the dynamic set of currently runnable threads). The parameterization of the machine to execute a by t in $\textsc{TriggerExec}_t(a)$ expresses the possible independence of the execution mechanisms for different threads.

$\textsc{ParallelSplit}(\mathit{Activity}, \mathit{Thread}, \textsc{TriggerExec}) =$
  forall $a \in \mathit{Activity}$ **let** $t = \mathit{new}(\mathit{Thread})$ **in** $\textsc{TriggerExec}_t(a)$

Remark. The well-known Occam instruction to spawn finitely many parallel subprocesses of a given process p matches this pattern exactly, see the OccamParSpawn-rule in [8, p.43], where $\textsc{TriggerExec}_t(a)$ describes the initialization of a by linking it to the triggering process p as its parent process, copying from there the current environment, setting a to run and p to wait (for all the spawned subprocesses to terminate).

In case one does not care about or does not want to have an explicit association of threads to activities, one can instantiate the above general scheme as follows, leaving all the subactivities of the pattern to be executed by one agent:

$\textsc{ParallelSplit}(\mathit{Activity}, \textsc{TriggerExec}) =$ **forall** $a \in \mathit{Activity}$ $\textsc{TriggerExec}(a)$

2.3 Synchronization

"A point in the workflow process where multiple parallel subprocesses/activities converge into one single thread of control, thus synchronizing multiple threads. It is an assumption of this pattern that each incoming branch of a synchronizer is executed only once..."

Besides the parameters for the set *Activity* of multiple parallel subprocesses a and their executing agents $\mathit{exec}(a)$, there is a second parameter to express the condition for the synchronization step to be enabled. It can be modeled by an abstract predicate $\mathit{SyncEnabled}(a)$ with arguments $a \in \mathit{Activity}$ to guard the call of the (here not furthermore specified abstract) process into which the elements of *Activity* Converge once they have all been executed. If one expects applications where *Activity* is a dynamic (read: run-time determined) set, it is reasonable to parameterize Converge by *Activity*. This leads to the following ASM description of the pattern.

$\textsc{Synchronizer}(\mathit{Activity}, \mathit{exec}, \mathit{SyncEnabled}, \textsc{Converge}) =$
  if forall $a \in \mathit{Activity}$ $\mathit{SyncEnabled}(\mathit{exec}(a))$ **then** $\textsc{Converge}(\mathit{Activity})$

The assumption "that each incoming branch of a synchronizer is executed only once" relates processes $a \in \mathit{Activity}$ to their unique executing thread $\mathit{exec}(a)$. It is natural to assume that during the execution of a, $\mathit{SyncEnabled}(\mathit{exec}(a))$ is false. One would probably also assume that Converge, besides choosing or creating one agent ("one single thread of control") to continue the workflow process, resets $\mathit{SyncEnabled}(a)$ to false for each a and $\mathit{exec}(a)$ to *undefined*, thus resetting Synchronizer for the next synchronization step.

2.4 Exclusive Choice

"A point in the workflow process where, based on a decision or workflow control data, one of several branches is chosen."

In this pattern besides the parameter for the set *Activity* of subprocesses among which to choose we have as second parameter a *DecisionCriterion* for the choice, which may take workflow control data as arguments. The underlying assumption seems to be that each time an exclusive choice point is reached (read: ExclChoice is executed), the *DecisionCriterion* yields exactly one $a \in \mathit{Activity}$ that fulfills it. This is expressed by the following ASM, where we use Hilbert's ι-operator notation $\iota x(P(x))$ to denote the unique element satisfying a property P.

ExclChoice($Activity, DecisionCriterion$) =
  let $act = \iota a(a \in Activity$ **and** $DecisionCriterion(a))$ **in**
    act

It is a question of how the *DecisionCriterion* is declared—as static or dynamic predicate, in the second case as monitored or controlled or shared—whether the decision for the choice is static (design time definable) or depends on runtime data. In case the uniqueness assumption is not guaranteed, it suffices to replace the ι-operator by Hilbert's choice operator ϵ. One may ask why instead of simply using this more liberal non-deterministic choose operator a decision criterion has been introduced for this pattern.

2.5 Simple Merge and Thread Merge

The *Simple Merge* pattern is described in [18] as follows: "A point in the workflow process where 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...". In [14] the description is weakend as follows, withdrawing the uniqueness condition: "The convergence of two or more branches into a single subsequent branch. Each enablement of an incoming branch results in the thread of control being passed to the subsequent branch." This formulation does not exclude multiple merge-enabled branches to proceed simultaneously.

In fact in [14] two variations are discussed, called *Thread Merge with Design/Run-Time Knowledge*, where a merge number *MergeNo* appears explicitly: "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", where this number can be "nominated" or "not known until run-time".

As in the Synchronizer pattern we see an implicit termination parameter *MergeEnabled* for threads executing elements of *Activity* that are to be merged. We foresee the frequent case that Proceed is parameterized by the merged activities.[1]

Merge($Activity, exec, MergeEnabled,$ Proceed$, MergeNo$) =
  let $A = Activity \cap \{a \mid MergeEnabled(exec(a))\}$ **if**$\mid A \mid = MergeNo$ **then** Proceed(A)

Instantiating Merge by $MergeNo = 1$ yields SimpleMerge under the mutual-exclusion hypothesis.

RelaxSimpleMerge is the variant of Merge with cardinality check $\mid A \mid \geq 1$ and Proceed(A) refined to **forall** $a \in A$ Proceed(a).[2] At a later point in [18] this pattern is called Multi-Merge and described as follows: "A point in the workflow process where 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.*"[3]

To capture the two Thread Merge variants it suffices to instantiate *Activity* to the set of execution threads in the considered single branch of a process and to declare *MergeNo* as static respectively dynamic. It is unclear whether there is any difference between the Synchronizer and the Merge pattern besides considering in the latter only the "execution threads in a single branch of the same process instance".

---

[1] $\mid A \mid$ denotes the cardinality of set A.

[2] It comes natural to assume here that when Proceed(a) is called, $MergeEnabled(exec(a))$ changes to false and $exec(a)$ to *undefined*. This guarantees that each completed activity triggers "the subsequent branch" once per activity completion. One way to realize this assumption is to require such an update to be part of Proceed(a); another possibility would be to add it as update to go in parallel with Proceed(a).

[3] It is possible that the relaxed form of Simple Merge was intended not to allow multiple merge-enabled branches to proceed simultaneously, in which case it either implies a further selection of one $a \in A$ to Proceed(a) as proxy for the others or a sequentialization of Proceed(a) for all $a \in A$.

The structural similary between ExclChoice and SimpleMerge (under the mutual exclusion hypothesis) reflects that in both patterns, a unique *act*ivity determines how to proceed.[4] The two patterns differ mainly by the underlying assumptions on the criterion through which this unique *act*ivity is determined: in the first case by a (whether static or dynamic) *DecisionCriterion* that is applied to all potential elements of *Activity* when the pattern is executed, in the second case by checking which activity is (executed by a thread that is) currently merge-enabled.

3 Advanced Branching and Synchronization Patterns

3.1 Multi-Choice

"A point in the workflow process where, based on a decision or workflow control data, a number of branches are chosen."

This pattern generalizes ExclChoice by permitting to select not exactly one alternative activity, but to "choose multiple alternatives from a given set of alternatives" that are executed together. Formally one therefore chooses a subset A of activities satisfying the underlying *ChoiceCriterion*,[5] all of whose members are called together for execution.

MultiChoice(*Activity*, *ChoiceCriterion*) =
  choose $A \subseteq Activity \cap ChoiceCriterion$
    forall $act \in A$
      act

Remark. The difference between ExclChoice and MultiChoice can be viewed as the result of two instantiations of a choice function, say *select*, which applied to $Activity \cap ChoiceCriterion$ yields a subset of activities chosen for execution. For ExclChoice this set is further specified by the constraint to be a singleton set.

Choice(*Activity*, *ChoiceCriterion*) =
  forall $act \in select(Activity \cap ChoiceCriterion)$
    act

3.2 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. It is an assumption of this pattern that a branch that has already been activated, cannot be activated again while the merge is still waiting for other branches to complete."

We understand this pattern, in accordance with the revised description in [14, pg.17] that "the thread of control is passed to the subsequent branch when each active incoming branch has been enabled", as a generalization of the Synchronizer pattern, restricting the to be synchronized processes to those elements of *Activity* that are *Active*. This predicate denotes the crucial pattern parameter "to decide when to synchronize and when to merge" and to determine the branches "the merge is still waiting for ... to complete". The additional requirement that once the moment to merge did arrive, the other branches should reconverge without synchronization, can be expressed by an additional (here not furthermore specified abstract) submachine Reconverge. In particular, as already mentioned for Synchronizer, both machines Converge and Reconverge are reasonably viewed as parameterized by the *Active* resp. non active processes.

[4] This difference can be viewed as the result of different instantiations of one abstract machine Proceed(*act*), where Proceed(a) = a holds in case of ExclChoice.

[5] The revised version of the multi-choice pattern in [14, pg.15] describes the selection as "based on the outcome of distinct logical expressions associated with each of the branches". This can be reflected by the parameterization of *ChoiceCriterion* with the set *Activity*, e.g. to represent a disjunction over the "distinct logical expressions associated with each of the (activity) branches".

SynchronizingMerge
(*Activity*, *exec*, *SyncEnabled*, *Active*, Converge, Reconverge) =
if *SynchrEvent* **then**
Converge(*Active*)
Reconverge(*Activity* \ *Active*)
where *SynchrEvent* = **forall** $a \in Activity \cap Active$ *SyncEnabled*(*exec*(*a*))

Remark. The generalization of Synchronizer by SynchronizingMerge can be turned the other way round so that Synchronizer appears as an instance of SynchronizingMerge, namely by stipulating the two constraints *Active* = *Activity* **and** Reconverge = **skip**.

The Acyclic Synchronizing Merge pattern presented in [14] is a variation described by the additional requirement that "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". This variation is easily captured by refining the *SynchrEvent* predicate to check whether the necessary *synchNumber* of to be synchronized enabled and active branches has been reached:

AcyclSynchrMerge = SynchronizingMerge **where**
$SynchrEvent =| \{a \in Activity \mid Active(a)$ **and** $SyncEnabled(exec(a))\} | \geq synchNumber$

Another variation called *General Synchronizing Merge* is described in [14] by relaxing the firing condition from "when each active incoming branch has been enabled" through the alternative "or it is not possible that the branch will be enabled at any future time". Such a restriction is easily formulated, relaxing *SyncEnabled*(*exec*(*a*)) in *SynchrEvent* by the disjunct "**or** *NeverMoreEnabled*(*exec*(*a*))", but obviously to compute such a predicate "requires a (computationally expensive) evaluation of possible future states for the current process instance" [14, pg.71].

3.3 Discriminator Variants

"The discriminator is 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 resets itself so that it can be triggered again..."

This description of what is called *Structured Discriminator* is about an alternation between two modes, say *waitingToProceed*, namely until a first incoming branch completes, and *reset*, namely after all remaining branches have completed. It is a clear case of a control-state ASM, see Fig. 1. Apparently *Completed* $\subseteq$ *Activity* is assumed.

The requirements speak about waiting "for one of the incoming branches to complete" before Proceeding, leaving the case open where more activities may complete simultaneously. We formalize this latter more general case. In doing this we foresee that the way to Proceed may be parameterized by the set of incoming branches whose activities have been the first to be simultaneously completed.

Discriminator(*Activity*, *Completed*, Proceed, Reset) =
if *mode* = *waitingToProceed* **and** $| Activity \cap Completed | \geq 1$ **then**
Proceed(*Activity* $\cap$ *Completed*)
mode := *reset*
if *mode* = *reset* **then** Reset

The variant *Structured N-out-of-M Join* discussed in [14] is the very same Discriminator machine, replacing the cardinality threshold 1 by N and letting $M =| Activity |$. The pattern discussed in [14] under the name *Generalized AND-Join* is the same as Structured N-out-of-M Join with additionally $N = M$.

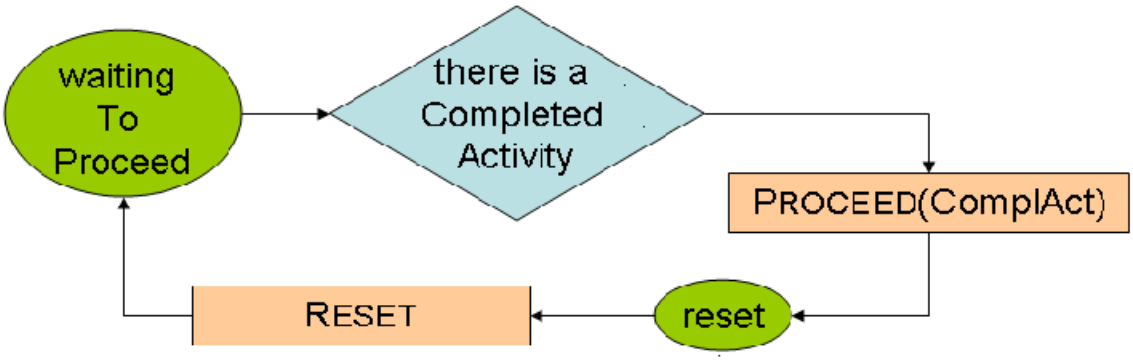

Fig. 1. Discriminator control-state ASM

RESET appears in the description of the structured discriminator as a durative action of waiting for other activities to complete. To check whether "all incoming branches have been triggered", one has to distinguish the activities that have not yet been detected as *Completed*. One option is to include such a *NotYetDetected* attribute into the predicate *Completed*; another option is to treat *NotYetDetected* as a separate predicate, assumed to be initially the entire set *Activity* and updated during each pattern round until it becomes empty. We choose the second option. In the description below *init*, *exit* denote the initial respectively final control state; as Fig. 2 shows, here we identify *init* with the *reset* mode, in which it is called by DISCRIMINATOR, and *exit* with the initial mode *waitingToProceed*.

StructuredDiscriminatorReset =
  if *mode* = *init* **then**
    MarkAsUnDetected(*Activity*)
    mode := *waitingForOtherActToComplete*
  if *mode* = *waitingForOtherActToComplete* **then**
    if *NotYetDetected* ≠ ∅ **then let** *A* = *Activity* ∩ *Completed* ∩ *NotYetDetected*
      if *A* ≠ ∅ **then** MarkAsDetected(*A*)
    else *mode* := *exit*
where
  MarkAsDetected(*A*) = (**forall** *a* ∈ *A* *NotYetDetected*(*a*) := *false*)
  MarkAsUnDetected(*A*) = (**forall** *a* ∈ *A* *NotYetDetected*(*a*) := *true*)

The variations called *Cancelling Discriminator* and *Cancelling N-out-of-M Join* are described in [14] by the additional requirement that "Triggering the discriminator (join) also cancels the execution of all of the other incoming branches and resets the construct". This comes up to define CancellingDiscriminatorReset as **forall** *a* ∈ *Activity* Cancel(*a*) (assuming for the sake of simplicity but without loss of generality that also the completed activities are 'cancelled', since their execution has terminated already).

Another variation, coming under two names *Blocking Discriminator* and *Blocking N-out-of-M Join*, is described in [14] by the additional requirement that "Subsequent enablements of incoming branches are blocked until the discriminator (join) has reset." It comes up to declare *Completed* as a set of queues *Completed*(*a*) of completion events for *a*, so that in each discriminator round only the first element *fstout* to leave a queue is considered and blocks the others. Correspondingly we refine a) the abstract completion predicate to **not** *Empty*(*Completed*(*a*)) and b) the updates of *NotYetDetected*(*a*) in MarkAs(Un)Detected by replacing *a* by *fstout*(*Completed*(*a*)) under the additional guard that *fstout*(*Completed*(*a*)) is defined. Upon exiting, i.e. in the last **else** branch

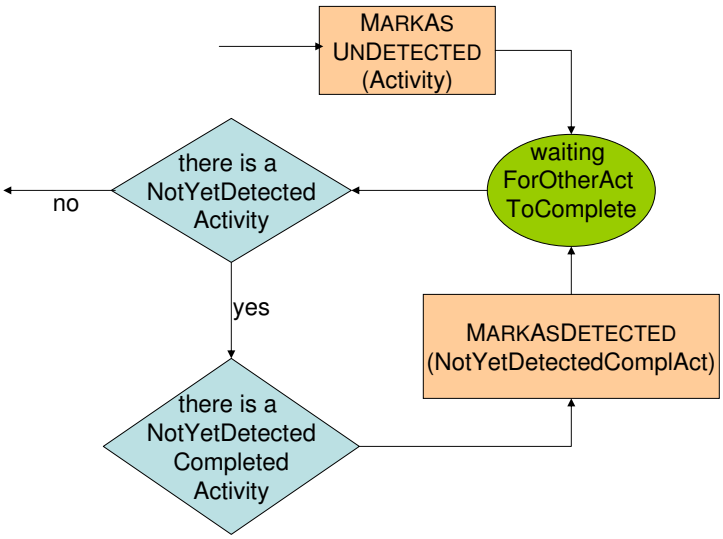

Fig. 2. STRUCTURED DISCRIMINATOR RESET

of STRUCTUREDDISCRIMINATORRESET, one has to add the deletion of the completion events that have been considered in this round:

forall $a \in Activity$ $Delete(fstout(Completed(a)), Completed(a))$

In [14] also variations of the preceding discriminator pattern versions are presented that work in concurrent environments. This is captured in our model by the fact that we have parameterized it among others by *Activity* and *Completed*, so that it can execute in an asynchronous manner simultaneously for different instances of these parameters.

3.4 Streamlining the Pattern Classification

As a result of the preceding analysis we would streamline the classification of basic control flow and advanced branching and synchronization patterns as follows:

SEQUENCE PARALLELSPLIT MERGE SYNCHRONIZINGMERGE CHOICE DISCRIMINATOR

SIMPLEMERGE, RELAXSIMPLEMERGE, MULTIMERGE refine MERGE. The two variants EXCLCHOICE and MULTICHOICE refine CHOICE. SYNCHRONIZER refines SYNCHRONIZINGMERGE by the two constraints $Active = Activity$ and RECONVERGE = **skip**. The formulation using a synchronization event invites to incorporate into the pattern classification also types of synchronization with more sophisticated synchronization criteria than mere completion of activities. This classification is not the only one possible. One referee suggests for example fo define DISCRIMINATOR in terms of the other schemes. Also MERGE and SYNCHRONIZING MERGE could be unified. It is unclear what classification to choose best.

4 Structural Patterns

4.1 Arbitrary Cycles, Structured Loop, Recursion

For arbitrary cycles the following rather loose description is given: "A point in a workflow process where one or more activities can be done repeatedly."

For the elements of *Activity* to be repeatedly executed, it seems that a *StopCriterion* is needed to express the point where the execution of one instance terminates and the next one starts. The additional stipulation in the revised description in [14] that the cycles may "have more than one entry or exit point" is a matter of further specifying this *StopCriterion* and starting points for activities, e.g. exploiting initial and final control states of control-state ASMs. The ITERATE construct defined for ASMs in [7] yields a direct formalization of this pattern that hides the explicit mentioning of entry and exit points.

ARBITRARYCYCLES(*Activity*, *StopCriterion*) =
  forall $a \in$ *Activity* ITERATE(a) **until** *StopCriterion*(a)

In [14] two further 'special constructs for structured loops' are introduced, called Structured Loop and Recursion. The formalization of the former comes up to the ASM constructs **while** *Cond* **do** M respectively **do** M **until** *Cond* defined in [7], for an ASM formalization of the latter we refer to [6] and skip further discussion of these well known programming constructs.

4.2 Termination

In [18] the following *Implicit Termination* pattern is described. "A given subprocess should be terminated when there is nothing else to be done. In other words, there are no active activities in the workflow and no other activity can be made active (and at the same time the workflow is not in deadlock)."

The point of this patterns seems to be to make it explicit that a subprocess has to be TERMINATEd depending on a typically dynamic *StopCriterion*. This varies from case to case. It may depend upon the subprocess structure. It may also include global features like that "there are no active activities in the workflow and no other activity can be made active"; another example is the projection of the run up-to-now into the future, namely by stipulating that the process should terminate "when there are no remaining work items that are able to be done either now or at any time in the future" [14, pg.25]. Such an abstract scheme is easily formulated as an ASM. It is harder to define reasonable instances of such a general scheme, which have to refine the *StopCriterion* in terms of (im)possible future extensions of given runs.

TERMINATION(P, *StopCriterion*, TERMINATE) =
  if *StopCriterion*(P, *Activity*) **then** TERMINATE(P)

In [14] the following variation called *Explicit Termination* is discussed. " A given process (or sub-process) instance should terminate when it reaches a nominated state. Typically this is denoted by a specific end node. When this end node is reached, any remaining work in the process instances is cancelled and the overall process instance is recorded as having completed successfully." It is nothing else than the instantiation of TERMINATION by refining a) the *StopCriterion* to *currstate* = *exit*, expressing that the current state has reached the end state, and b) TERMINATE(P) to include CANCEL(P) and marking the overall process *parent*(P) as *CompletedSuccessfully*.

5 Pattern Variations

5.1 Variations of Multiple Instances 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."

The two crucial parameters of this pattern are an *act*ivity and the *Mult*itude with which new agents $a(act)$ for the execution of instances of act are to be run. We use again the *new* function to provide the sufficiently fresh elements, here for the dynamic set of *Agent*s empowered to RUN(act).

MULTINSTWITHOUTSYNC($act, Mult, Agent$) =
  forall $i \in Mult$ **let** $a_i = new(Agent)$ **in** RUN(a_i, act)

Three variations of this pattern appear in [18]. They have the same structure, but different interpretations on the static or dynamic nature of the *Mult*itude parameter.

For the *Multiple Instances With a Priori Design Time Knowledge* pattern the set *Mult* is declared to be known a priori at design time. For this pattern it is required in addition that "once all instances are completed some other activity needs to be started." We capture this requirement by a successively to be executed rule to PROCEED when all newly created agents have *Completed* their run. Since in [14] also a variation is considered under the name *Static N-out-of-M Join for Multiple Instances*, where to PROCEED only N out of $Mult = M$ activity instances need to have completed, we make here the cardinality parameter explicit and specialize it then to $N = \mid Agent(act) \mid$. The variation *Static Cancelling N-out-of-M Join for Multiple Instances* in [14] can be obtained by adding a cancelling submachine, adopting the scheme explained above for the discriminator pattern.

MULTINSTNMJOIN($act, Mult, Agent, Completed$, PROCEED, N) =
  MULTINSTWITHOUTSYNC($act, Mult, Agent$) **seq**
    if $\mid Agent(act) \cap Completed \mid \geq N$ **then** PROCEED

MULTINSTAPRIORIDESIGNKNOWL($act, Mult, Agent, Completed$, PROCEED) =
  MULTINSTNMJOIN($act, Mult, Agent, Completed$, PROCEED, $\mid Agent(act) \mid$)

The pattern *Multiple Instances With a Priori Run Time Knowledge* is the same except that the *Mult*itude "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." This can be expressed by declaring *Mult* for MULTINSTAPRIORIRUNKNOWL as a dynamic set.

The *Multiple Instances Without a Priori Run Time Knowledge* pattern is the same as *Multiple Instances With a Priori Run Time Knowledge* except that for *Mult*itude it is declared that "the number of instances of a given activity for a given case is not known during desing time, nor is it known at any stage during runtime, before the instances of that activity have to be created", so that "at any time, whilst instances are running, it is possible for additional instances to be initiated" [14, pg.31]. This means that as part of the execution of a RUN(a, act), it is allowed that the set $Agent(act)$ may grow by new agents a' to RUN(a', act), all of which however will be synchronized when *Completed*. Analogously the pattern *Dynamic N-out-of-M Join for Multiple Instances* discussed in [14] is a variation of Static N-out-of-M Join for Multiple Instances.

The *Complete Multiple Instance Activity* pattern in [14] is yet another variation: "... 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."

To reflect this additional requirement it suffices to add the following machine to the second submachine of MULTINSTAPRIORIDESIGNKNOWL:

if $Event(ForcedCompletion)$ **then**
  forall $a \in (Agent(act) \setminus Completed)$ **do** CANCEL(a)
  PROCEED

5.2 State-Based Patterns

The patterns discussed here concern "business scenarios where an explicit notion of state is required". We know of no other computational framework whose notion of state comes up to what ASMs provide in terms of generality, simplicity and abstract (representation independent) character of the underlying explicit notion of state. This is also technically reflected in the fact that the four state-based patterns considered in [18] can be expressed by rather simple ASMs.

5.3 Deferred Choice

"A point in the workflow process where one of several branches is chosen. In contrast to the XOR-split, the choice is not made explicitly (e.g. based on data or a decision) but several alternatives are offered to the environment. However, in contrast to the AND-split, only one of the alternatives is executed ... It is important to note that the choice is delayed until the processing in one of the alternative branches is actually started, i.e. the moment of choice is as late as possible."

All this is simply expressed by the ExclChoice ASM defined above, where the *DecisionCriterion* is declared to be a monitored predicate.

5.4 Interleaved Parallel Routing

"A set of activities is executed in an arbitrary order: Each activity in the set is executed, the order is decided at run-time, and no two activities are executed at the same moment (i.e. no two activities are active for the same workflow at the same time)."

We illustrate some among the numerous ways to make this description rigorous, depending on the degree of detail with which one wants to describe the interleaving scheme. A rather liberal way is to execute the underlying activities one after another until *Activity* has become empty, in an arbitrary order, left completely unspecified:

InterleavedPar(*Activity*) = **choose** *act* ∈ *Activity*
  act
  Delete(*act*, *Activity*)

A more detailed scheme forsees the possibility to impose a certain scheduling algorithm for updating the currently executed activity *curract*. The function *schedule* used for the selection of the next not-yet-completed activity comes with a name and thus may be specified explicitly elsewhere. For example, to capture the generalization of this pattern in [14, pg.34], where the activities are partially ordered and the interleaving is required to respect this order, *schedule* can simply be specified as choosing a minimal element among the not-yet-completed activities.

ScheduledInterleaving(*Activity*, *Completed*, *schedule*) =
  if *Completed*(*curract*) **then** *curract* := *schedule*({*a* ∈ *Activity* | **not** *Completed*(*a*)})

A more sophisticated interleaving scheme could permit that the execution of activities can be suspended and resumed later. A characteristic example appears in [15, Fig.1.3] to describe the definition of the multiple-thread Java interpreter using a single-thread Java interpreter. It can be paraphrased for the workflow context as follows, assuming an appropriate specification of suspending and resuming activities and using a composed abstract predicate *ExecutableRunnable* that filters the currently executable and runnable activities from *Activity*.

InterleaveWithSuspension
  (*Activity*, *ExecutableRunnable*, Execute, Suspend, Resume) =
    choose *a* ∈ *ExecutableRunnable*(*Activity*) **if** *a* = *curract* **then** Execute(*curract*)
      else
        Suspend(*curract*)
        Resume(*a*)

The generalization from atomic activities to critical sections, proposed in [14] as separate pattern *Critical Section*, is a straightforward refinement of the elements of *Activity* to denote "whole sets of activities". Also the variation, called *Interleaved Routing*, where "once all of the activities have completed, the next activity in the process can be initiated" is simply a sequential composition of Interleaved Parallel Routing with NxtActivity.

There is a large variety of other realistic interpretations of Interleaved Parallel Routing, yielding pairwise different semantical effects. The informal requirement description in [18,14] does not suffice to discriminate between such differences.

5.5 Milestone

"The enabling of an activity depends on the case being in a specified state, i.e. the activity is only enabled if a certain milestone has been reached which did not expire yet."

This rather loose specification is easily translated as follows:

Milestone(*milestone*, *Reached*, *Expired*, *act*) =
  if *Reached*(*milestone*) **and not** *Expired*(*milestone*) **then** *act*

6 Cancellation Patterns

6.1 Cancel (Multiple Instance) Activity, Cancel Case, Cancel Region

The *Cancel Activity* pattern is described as follows: "An enabled activity is disabled, i.e. a thread waiting for the execution of an activity is removed."

Using an association *agent*(*act*) of threads to activities allows one to delete the executing agent, but not the activity, from the set *Agent* of currently active agents:

CancelAct(*act*, *Agent*, *exec*) =
  let $a = exec(act)$ **in if** *Enabled*(a) **then** Delete(a, *Agent*)

The *Cancel Case* pattern is described as follows: "A case, i.e. workflow instance, is removed completely (i.e., even if parts of the process are instantiated multiple times, all descendants are removed)."

If we interprete 'removing a workflow instance' as deleting its executing agent,[6] this pattern appears to be an application of CancelAct to all the *Descendant*s of an *act*ivity (which we assume to be executed by agents), where for simplicity of exposition we assume *Descendant* to include *act*.

CancelCase(*act*, *Agent*, *exec*, *Descendant*) =
  forall $d \in Descendant(act)$ CancelAct(d, *Agent*, *exec*)

For the *Cancel Region* pattern we find the following description in [14]: "The ability to disable a set of activities in a process instance. If any of the activities are already executing, then they are withdrawn. The activities need not be a connected subset of the overall process model."

CancelRegion is a straightforward variation of CancelCase where *Descendant*(p) is defined as the set of activities one wants to cancel in the process instance p. Whether this set includes p itself or not is a matter of how the set is declared. The additional requirement that already executing activities are to be withdrawn is easily satisfied by refining the predicate *Enabled*(a) to include executing activities a. The question discussed in [14] whether the deletion may involve a bypass or not is an implementation relevant issue, suggested by the Petri net representation of the pattern.

An analogous variation yields an ASM for the *Cancel Multiple Instance Activity* pattern, for which we find the following description in [14]: "Within a given process instance, multiple instances of an activity can be created. The required number of instances is known at design time. These

[6] To delete the activity and not only its executing agent would imply a slight variation in the ASM below.

instances are independent of each other and run concurrently. At any time, the multiple instance activity can be cancelled and any instances which have not completed are withdrawn. This does not affect activity instances that have already completed." Here it suffices to define *Descendant*(p) in CancelCase as the set of multiple instances of an activity one wants to cancel and to include 'activity instances which have not yet completed' into the *Enabled* predicate of CancelAct.

7 Additional Control-Flow Patterns

In this section we analyse the remaining patterns defined in the revised version [14] of [18].

7.1 Transient Trigger, Persistent Trigger

Transient Trigger: "The ability for an activity to be triggered by a signal from another part of the process or from the external environment. These triggers are transient in nature and are lost if not acted on immediately by the receiving activity."

Two variants are considered. In the so-called 'safe' variant, only one instance of an activity 'can wait on a trigger at any given time'. In the unsafe variant multiple instances of an activity 'can remain waiting for a trigger to be received'.[7]

Persistent Trigger: " ... These triggers are persistent in form and are retained by the workflow until they can be acted on by the receiving activity."

Two variants are considered. In the first one 'a trigger is buffered until control-flow passes to the activity to which the trigger is directed', in the second one 'the trigger can initiate an activity (or the beginning of a thread of execution) that is not contingent on the completion of any preceding activities'.

We see these patterns and the proposed variants as particular instantiations of one Trigger pattern, dealing with monitored events to trigger a process and instantiated depending on whether at a given moment multiple processes wait for a trigger and on the time that may elapse between the trigger event and the reaction to it. We add to this the possibility that in a distributed environment, at a given moment multiple trigger events may yield a simultaneous reaction of multiple ready processes. We leave the submachines for Buffering and UnBuffering abstract and only require that as result of an execution of Buffer(a) the predicate *Buffered*(a) becomes true. For notational reasons we consider monitored events as consumed by the execution of a rule.[8]

Trigger =
  TriggerEvent
  TriggerReaction
where
  TriggerEvent = **if** *Event*(*Trigger*(a)) **then** Buffer(a)
  TriggerReaction =
    if not *Empty*(*Buffered* ∩ *Ready*) **then**
      choose $A \subseteq$ *Buffered* ∩ *Ready* **forall** $a \in A$ **do**
        a
        UnBuffer(a)

The two variants considered for the Persistent Trigger differ from each other only by the definition of *Ready*(a), meaning in the first case *WaitingFor*(*Trigger*(a)) and in the second case *curract* = a ('process has reached the point to execute a'), where *curract* is the activity counter pointing to the currently to be executed activity.

For the Transient Trigger it suffices to stipulate that there is no buffering, so that *Buffered* coincides with the happening of a triggering event and upon the arrival of an event, TriggerEvent and TriggerReaction are executed simultaneously if the event concerns a *Ready*(a), in which case (and only in this case) it triggers this activity.

[7] Note that this safety notion is motivated by the Petri net framework.
[8] This convention allows us to suppress the explicit deletion of an event from the set of active events.

TransientTrigger = Trigger **where**
Buffer = UnBuffer = **skip**
Buffered(*a*) = *Event*(*Trigger*(*a*))

The difference between the safe and unsafe version is in the assumption on how many activity (instances) may be ready for a trigger event at a given moment in time, at most one (the safe case) or many, in which case a singleton set A is required to be chosen in TriggerReaction.

8 Conclusion and Outlook

The rational for the choice of the 20 individually named patterns collected in [18], for their revision and extension by 23 new ones in [14] and the reasons for the particular classification that is proposed in those two papers remain unclear. We are not yet sure how a reasonable classification will look like and hope that the present ASM formulation will help to provide a simple, truly abstract basis for an accurate analysis and evaluation of relevant control-flow patterns, preventing the pattern variety to grow without rational guideline.

Acknowledgement Thanks to three referees who provided critical remarks.

References

1. A. Barros and E. Börger. A compositional framework for service interaction patterns and communication flows. In K.-K. Lau and R. Banach, editors, *Proc. 7th International Conference on Formal Engineering Methods (ICFEM 2005)*, volume 3785 of *LNCS*, pages 5–35. Springer, 2005.
2. E. Börger. The ASM ground model method as a foundation of requirements engineering. In N.Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 145–160. Springer-Verlag, 2003.
3. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.
4. E. Börger. The ASM method for system design and analysis. A tutorial introduction. In B. Gramlich, editor, *FroCoS 2005*, volume 3717 of *LNAI*, pages 264–283. Springer, 2005.
5. E. Börger. Design pattern abstractions and Abstract State Machines. In D. Beauquier, E. Börger, and A. Slissenko, editors, *Proc.ASM05*, pages 91–100. Université de Paris 12, 2005.
6. E. Börger and T. Bolognesi. Remarks on turbo ASMs for computing functional equations and recursion schemes. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003 – Advances in Theory and Applications*, volume 2589 of *Lecture Notes in Computer Science*, pages 218–228. Springer-Verlag, 2003.
7. E. Börger and J. Schmid. Composition and submachine concepts for sequential ASMs. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (Proceedings of CSL 2000)*, volume 1862 of *Lecture Notes in Computer Science*, pages 41–60. Springer-Verlag, 2000.
8. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.
9. W. R. Cook, S. Patwardhan, and J. Misra. Workflow patterns in Orc. In *Proc. of the 8th Intl. Conf. on Coordination Models and Languages (COORDINATION)*, Springer LNCS 4038, pages 82–96, 2006.
10. M. Dumas and A. H. ter Hofstede. UML activity diagrams as a workflow specification language. Technical report, Cooperative Information Systems Research Center, Queensland University of Technology, November 2003.
11. R. Eshuis and J. Dehnert. Reactive Petri nets for workflow modeling. In W. van der Aaalst and E. Best, editors, *Proc. 24th Int. Conf. on Applications and Theory of Petri Nets (ICATPN 2003)*, volume 2679 of *LNCS*, pages 296–315. Springer, 2003.
12. R. Eshuis and R. Wieringa. Comparing Petri net and activity diagram variants for workflow modelling– A quest for reactive Petri nets. In H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors, *Petri Net Technology for Communication-Based Systems*, volume 2472 of *LNCS*, pages 321–351. Springer, 2003.
13. F. Puhlmann and M. Weske. Using the π-calculus for formalizing workflow patterns. volume 3649 of *LNCS*. Springer, 2005.

14. N. Russel, A. ter Hofstede, W. M. P. van der Aalst, and N. Mulyar. Workflow control-flow patterns. A revised view. Draft Manuscript BPMCenter.org, July 2006.
15. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.
16. W. van der Aalst. The application of Petri nets to workflow management. *J. of Circuits, Systems and Computers*, 8(1):21–66, 1998.
17. W. van der Aalst and A. ter Hofstede. YAWL: Yet another workflow language. *Information Systems*, 30(4):245–275, 2005.
18. W. M. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51, July 2003.