Modeling Workflow Patterns from First Principles

Egon Börger

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

Abstract. We propose a small set of parameterized abstract models for workflow patterns, starting from first principles for sequential and distributed control. Appropriate instantiations yield the 43 workflow patterns that have been listed recently by the Business Process Modeling Center. The resulting structural classification of those patterns into eight basic categories, four for sequential and four for parallel workflows, provides a semantical foundation for a rational evaluation of workflow patterns.

1 Introduction

In [3] we have provided Abstract State Machine (ASM) models for the 43 workflow pattern descriptions that have been presented recently in [8] by the Business Process Modeling Center. Our goal there was to make the underlying relevant questions and implicit parameters explicit and to turn the patterns into a precise and truly abstract form. To easen the validation of these ASM ground models, in the sense defined in [1], we esssentially followed the order of presentation adopted in [8] and only hinted at the most obvious streamlining the ASM models offer for the classification presented in [8].

In this paper we revisit those workflow pattern ASMs and define eight basic workflow patterns, four for sequential and four for distributed control, from which all the other patterns can be derived by parameter instantiation.[1] This provides a conceptual basis for a rational workflow pattern classification that can replace the partly repetitive listing presented in [8].

We use again the ASM method to provide a high-level, both state-based and process-oriented view of workflow patterns. This provides a solid semantic foundation for reasoning about workflow functionality. In the ASM models the behavioral interface is defined through actions performed with the help of submachines that remain largely abstract. The parameterization exploits the possibility the ASM method offers the specifier to build 'models with holes', that is to leave

[1] We omit here the four so-called State-Based Patterns in [10], which concern "business scenarios where an explicit notion of state is required" and are only loosely connected to workFLOW. Exploiting the most general character of the ASM notion of state, these four state-based patterns can be expressed by rather simple ASMs.

parts of the specification either as completely abstract parameters or to accompany them by assumptions or informal explanations, which are named, but verified respectively detailed only at later refinement stages. The parameterization allows one in particular 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 [2], for a textbook presentation, which includes a formalized version of the definition of the semantics of ASMs, see the Asm-Book [6]. For sequential patterns we use mono-agent (so-called sequential) ASMs, for patterns of distributed nature multiple-agent asynchronous (also called distributed) ASMs.

We make no attempt here to provide a detailed analysis of the basic concepts of *activity, process, thread*, of their being *active, enabled, completed* etc., which are used in [8] 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. An *active activity*, for example, is one whose executing agent is active, etc. The quotes below are taken from [8] or its predecessor [10].

We start with the more interesting case of parallel control flow patterns, followed by the more traditional patterns for sequential control flow known from programmming.

2 Parallel Control Flow Patterns

The patterns related to parallel control flow can be conceptually categorized into four types: *splitting* one flow into multiple flows, *merging* multiple flows into one flow, forms of *interleaving* and *trigger* variations. As Andreas Prinz has pointed out, there are good reasons to use instead a classification into splitting and merging only. Interleaving appears then as parameter for different instances of splitting, whereas triggering is considered as belonging to start patterns in the context of distributed (not mono-agent sequential) computations. We do not claim to have a unique answer to the classification problem. What we believe is important is to start with a small number of basic patterns out of which more complex patterns can be obtained by composition and refinement.

2.1 Parallel Split Patterns

We quote the description of the parallel split pattern:

> 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.

This description contains two not furthermore specified parameters, which we represent by two sets *Activity* and *Thread* capturing the underlying activities and the threads executing them. It is left open whether *Activity* is declared as static or as dynamic, thus providing for static instantiations and for dynamic ones, whether as known at design time or as known only at the moment of executing the parallel split. In contrast the set *Thread* has to be declared as dynamic, since multiple threads of control have to be created without committing to the precise nature of the underlying parallelism, which is left unspecified in the above pattern description.

The parallelism may be further specified as an interleaving execution, using one of the interleaving patterns of Sect. 2.3, or as a simultaneous synchronous or as asynchronous execution. The latter two cases can be expressed using synchronous respectively asynchronous (also called distributed) ASMs. The particular choice can be left open if we create for each $a \in \mathit{Activity}$ a new thread to execute a. For this purpose we use a function *new* that is assumed to provide a fresh element each time it is applied to a set. To provide a handle for expressing the possible independence of the execution mechanisms for different threads, we explicitly name a third parameter, namely a machine that triggers the execution of an activity by a thread. For the representation of such a mechanism we avoid committing to a particular framework, e.g. Petri nets where triggering is traditionally represented by placing tokens that enable a transition. This is the reason why we introduce an abstract machine $\textsc{TriggerExec}(t, a)$. It is not furthermore specified except for requiring that its call triggers enables the execution of activity a by thread t.

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

This pattern is widely used in various forms. A well-known one is represented by the Occam instruction [7] to spawn finitely many parallel subprocesses of a given process p, which matches this pattern exactly. See the $\textsc{OccamParSpawn}$-rule in [6, 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). This instance of $\textsc{ParallelSplit}$ uses as underlying parallelism the concept of asynchronous ASMs.

An instance with synchronous parallelism takes the following form, where all the activities in question are executed simultaneously, e.g. as action of one agent. This is already captured by the default parallelism of basic non-distributed ASMs so that it suffices to instantiate $\textsc{TriggerExec}$ as not depending on the thread parameter (whereby the creation of new threads can simply be deleted):

```
$\textsc{SyncParSplit}(\mathit{Activity}, \textsc{TriggerExec}) =\ \textbf{forall}\ a \in \mathit{Activity}\ \ \textsc{TriggerExec}(a)$
```

In [8] other parallel split patterns are discussed for multiple instances of one activity. One of the descriptions runs as follows.

> 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.

This so-called *Multiple Instances Without Synchronization* pattern, which apparently comes with an asynchronous understanding of the underlying parallelism, is an instance of ParallelSplit where *Activity* is further specified to be a multiset of multiple instances of one activity *act*. Formally $Activity = MultiSet(act, Mult)$ where *Mult* denotes the number of occurrences of *act* in the multiset and determines the multitude with which new threads for the execution of instances of *act* are to be created and triggered to execute *act*.

$$\textsc{MultInstWithoutSync}(act, Mult, Thread, \textsc{TriggerExec}) = \\ \quad \textsc{ParallelSplit}(MultiSet(act, Mult), Thread, \textsc{TriggerExec})$$

In [10] some variations of this pattern appear, which essentially differ by their interpretations on the static or dynamic character of the *Mult*itude parameter. In the ASM framework this is merely a matter of how the parameter is declared. Since in the formulation of these pattern variants some additional conditions appear that have to do with synchronization features, we postpone their discussion to Sect. 2.2 where combinations of split and join patterns are discussed.

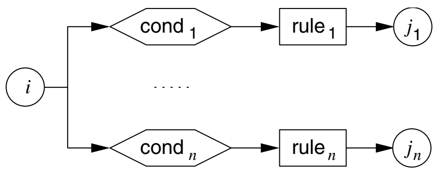
2.2 Merge Patterns

The following characterization seems to capture what is common to all the synchronization and merge patterns in [8]:

> 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.

The general scheme appears to be that one has to perform a specific convergence action that characterizes the start of the merge phase, namely when a *MergeEv*ent occurs, and then to complete the merge by some further actions. To represent these two successive and possibly durative aspects of a merge we use separate abstract machines StartMerge and CompleteMerge. To capture that multiple actions may be involved to complete a merge cycle, we formalize the above description by the following control state ASM Merge, i.e. an ASM all of whose rules have the form pictorially depicted in Fig. 1. Here $i, j_1, \ldots, j_n$ denote the control states corresponding to the internal states of an FSM (Finite State Machine), $cond_\nu$ (for $1 \leq \nu \leq n$) the guards and $rule_\nu$ the rule actions.

The control state ASM Merge switches between two modes *mergeStart*, *MergeCompl* and takes the merge event predicate and the two submachines for starting and completing the merge as not furthermore specified abstract parameters.



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

Fig. 1. Control state (FSM like) ASM rules

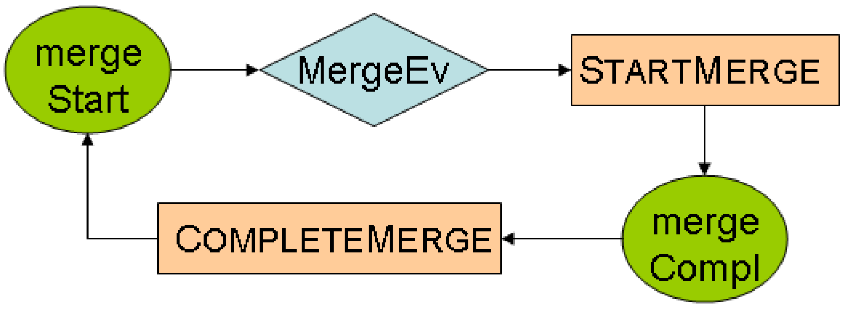


Fig. 2. General Merge Pattern ASM Merge

In accordance with the understanding of activities as executed by agents representing threads, we name explicitly also these underlying agents since they are the ones to be merged (not really the activities). We use for this a parameter *exec* denoting for every $a \in Activity$ the agent $exec(a)$ executing a, if there is one. It typically serves as parameter for defining the merge event *MergeEv*.

```
Merge($Activity, exec, MergeEv$, StartMerge, CompleteMerge) =
  if $ctl\_state = mergeStart$ and $MergeEv(exec)$ then
    StartMerge
    $ctl\_state := mergeCompl$
  if $ctl\_state = mergeCompl$ then
    CompleteMerge
    $ctl\_state := mergeStart$
```

Various forms of synchronizing merge patterns, whether with or without synchronization, can be described as instances of the general merge pattern ASM Merge. We illustrate this here by deriving the various merge patterns that appear in [8].

Discriminators. One class of patterns in [8] that represent instances of the two-phase merge pattern MERGE are the so-called Discriminator patterns. They present the durative character of a merging phase together with two additional basic merge features, namely merging with or merging without synchronization. The so-called *Structured Discriminator* pattern is described as follows:

> 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...

To view this pattern as an instance of MERGE, essentially we have to instantiate *MergeEv* to the check whether there is "one of the incoming branches to complete". Really this is a shorthand for expressing that a thread executing the activity a associated to a branch has reached a completion point for that activity, formally whether $Completed(a, exec(a))$.

As a cosmetic adaptation one may rename the control states *mergeStart* and *mergeCompl* to reflect the basic intention of the discriminator pattern as alternation between a *waitingToProceed* mode, namely until a first incoming branch completes, and a *reset* mode, during which all remaining branches will complete "without synchronization". Similarly one may rename STARTMERGE and COMPLETEMERGE to PROCEED respectivelyRESET.

Speaking about waiting "for one of the incoming branches to complete" leaves the case open where more activities complete simultaneously. We formalize the pattern so that this latter more general case is contemplated, where multiple activities that complete together may be synchronized. 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. Note that this formalization allows one to refine the 'synchronization' to choosing one among the simultaneously completed activities. This leads to the following instantiation of MERGE by Fig. 3.

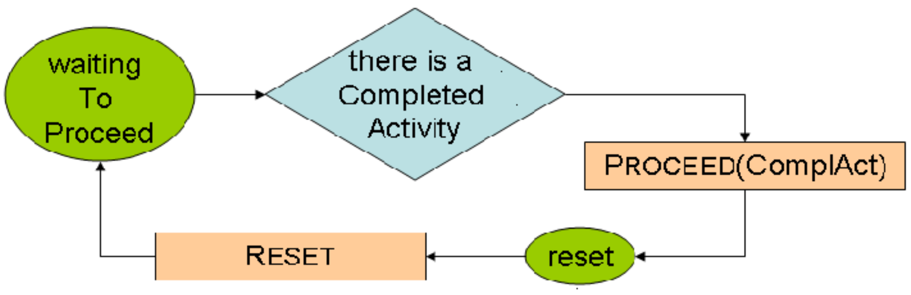


Fig. 3. Discriminator control-state ASM

$\textsc{Discriminator}(\mathit{Activity}, \mathit{exec}, \mathit{Completed}, \textsc{Proceed}, \textsc{Reset}) =$
$\quad \textsc{Merge}(\mathit{Activity}, \mathit{exec}, \mathit{MergeEv}, \textsc{Proceed}(\mathit{ComplAct}), \textsc{Reset})$
where
$\quad \mathit{MergeEv} = \mid \mathit{ComplAct} \mid \geq 1$
$\quad \mathit{ComplAct} = \{a \in \mathit{Activity} \mid \mathit{Completed}(a, \mathit{exec}(a))\}$

The variant *Structured N-out-of-M Join* discussed in [8] is the very same Discriminator machine, replacing the cardinality threshold 1 by N and letting $M = \mid \mathit{Activity} \mid$[2]. The pattern discussed in [8] under the name *Generalized AND-Join* is the same as Structured N-out-of-M Join with additionally $N = M$.

Reset appears in the above quoted description of the structured discriminator as a durative action of waiting for other activities to complete. It suffices to refine Reset to the following machine Structured Discriminator Reset. To check whether “all incoming branches have been triggered”, one has to distinguish the activities which have not yet been detected as completed. Thus one needs a *NotYetDetected* test predicate, which initially is satisfied by every element of the set *Activity* and is updated until it becomes empty. In the description below *init*, *exit* denote the initial respectively final control state of the refined machine. As Fig. 4 shows, for the replacement of Reset by Structured Discriminator Reset we identify *init* with the *reset* mode, in which it is called by Discriminator, and *exit* with the initial mode *waitingToProceed*.

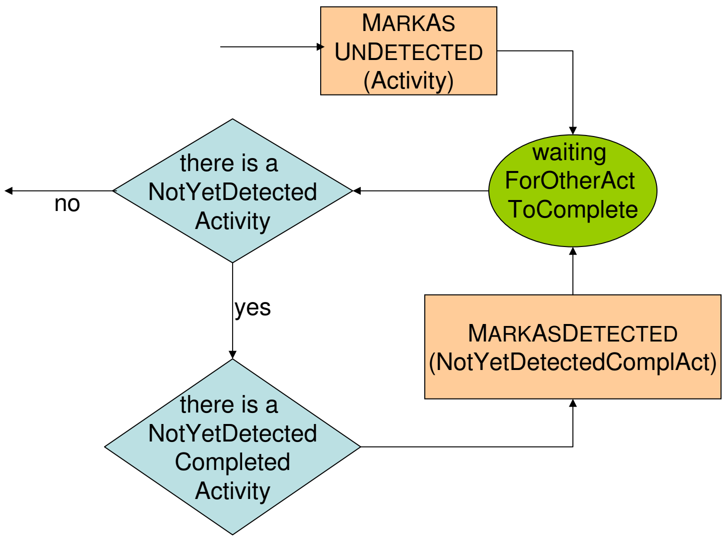

MarkAs UnDetected (Activity)
waiting ForOtherAct ToComplete
there is a NotYetDetected Activity
no
yes
MarkAsDetected (NotYetDetectedComplAct)
there is a NotYetDetected Completed Activity

Fig. 4. Structured Discriminator Reset

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

```
$\textsc{StructuredDiscriminatorReset} =$
  $\textbf{if } mode = init \textbf{ then}$
    $\textsc{MarkAsUnDetected}(Activity)$
    $mode := waitingForOtherActToComplete$
  $\textbf{if } mode = waitingForOtherActToComplete \textbf{ then}$
    $\textbf{if } NotYetDetected \neq \emptyset \textbf{ then let } A = ComplAct \cap NotYetDetected$
      $\textbf{if } A \neq \emptyset \textbf{ then } \textsc{MarkAsDetected}(A)$
    $\textbf{else } mode := exit$
$\textbf{where}$
  $\textsc{MarkAsDetected}(A) = (\textbf{forall } a \in A \; NotYetDetected(a) := false)$
  $\textsc{MarkAsUnDetected}(A) = (\textbf{forall } a \in A \; NotYetDetected(a) := true)$
  $ComplAct = \{a \in Activity \mid Completed(a, exec(a))\}$
```

The variations called *Cancelling Discriminator* and *Cancelling N-out-of-M Join* are described in [8] 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 the following instances of RESET:

```
$\textsc{CancellingDiscriminatorReset} =$
  $\textbf{forall } a \in Activity \setminus ComplAct \; \textsc{Cancel}(exec(a))$
```

In [8] some more variations, coming under the names *Blocking Discriminator* and *Blocking N-out-of-M Join*, are described 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 (read: announcing the completion of some thread's execution) for each activity a, so that in each discriminator round only the first element *fstout* to leave a queue is considered and blocks the others. This leads to the following refinement step:

- refine the abstract completion predicate to **not** $Empty(Completed(a))$,
- refine the updates of $NotYetDetected(a)$ by replacing a by $fstout(Completed(a))$ (under the additional guard that $fstout(Completed(a))$ is defined),
- for exiting, i.e. in the last **else** branch of STRUCTUREDDISCRIMINATOR − RESET, add the deletion of the completion events that have been considered in this round:

  $\textbf{forall } a \in Activity \; \textsc{Delete}(fstout(Completed(a)), Completed(a))$

In [8] 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.

Synchronizing Merge. This pattern too presents two merge components, one with and one without synchronization. It is described in [8] as follows:

> 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. ... the thread of control is passed to the subsequent branch when each active incoming branch has been enabled.

This is a merge pattern instance where the threads, which execute the activities associated to branches and are described as *Active*, have to be synchronized, whereas the remaining threads have to "reconverge without synchronization". The synchronization event 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". Nevertheless no definition of threads being *Active*, *Activated* or *Enabled* is given, so that in particular it is unclear whether by interpreting enabledness as completion this pattern reduces to the structured discriminator pattern. Since a decision is needed, we choose to formalize the synchronization event in terms of enabledness of active branch activities by instantiating the *MergeEv*ent in MERGE correspondingly. As a cosmetic change we rename STARTMERGE and COMPLETEMERGE to CONVERGE respectively RECONVERGE. The iterative nature of the not furthermore specified RECONVERGE machine can be formalized by a structured version, as done for the discriminator RESET submachine. In this case the description of this pattern in fact is just another wording for the structured version of the discriminator.

```
$\text{SynchronizingMerge}(\textit{Activity}, \textit{exec}, \textit{Active}, \textit{SyncEnabled}, \text{Converge}, \text{Reconverge}) =$
  $\text{Merge}(\textit{Activity}, \textit{exec}, \textit{MergeEv}, \text{Converge}(\textit{Active}), \text{Reconverge}(\textit{Activity} \setminus \textit{Active}))$
where
  $\textit{MergeEv} =$ forall $a \in \textit{Activity}$ if $\textit{Active}(\textit{exec}(a))$ then $\textit{SynEnabled}(\textit{exec}(a))$
```

The assumption "that each incoming branch of a synchronizer is executed only once" relates each $a \in \textit{Activity}$ to a unique executing thread $\textit{exec}(a)$. It is natural to assume that at the beginning of the execution of a, $\textit{SyncEnabled}(\textit{exec}(a))$ is false and that after having become true during this execution, it is reset to false by CONVERGE respectively RECONVERGE, thus resetting SYNCHRONIZINGMERGE for the next synchronization round.

The machine SYNCHRONIZINGMERGE has been further simplified in [8] to a pattern called SYNCHRONIZER. This can be defined as an instantiation of SYNCHRONIZINGMERGE by declaring all activities to be active (i.e. $\textit{Active}(\textit{exec}(a))$ holds for each $a \in \textit{Activity}$ when the pattern is used) and reconverging to be empty (RECONVERGE = **skip**).

The Acyclic Synchronizing Merge pattern presented in [8] is another variation described by the following additional requirement:

> 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 *MergeEv*ent predicate to check whether the necessary *synchNumber* of to be synchronized enabled and active branches has been reached:

ACYCLSYNCHRMERGE = SYNCHRONIZINGMERGE **where**
  $MergeEv = \mid \{a \in Activity \mid Active(exec(a))$ **and** $SyncEnabled(exec(a))\} \mid \geq synchNumber$

Another variation called *General Synchronizing Merge* is described in [8] 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". To reflect this restriction it suffices to relax $SyncEnabled(exec(a))$ in *MergeEv* by the disjunct "**or** $NeverMoreEnabled(exec(a))$", but obviously the crux is to compute such a predicate. It "requires a (computationally expensive) evaluation of possible future states for the current process instance" [8, pg.71].

Simple and Thread Merge. The *Simple Merge* pattern described in [10] is an example of merging without synchronization. Its description runs 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.

This is an instance SIMPLE MERGE of the MERGE ASM where the two control states are identified and we set COMPLETEMERGE = **skip**. In [8] the description is weakened 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 weakening can be made explicit by incorporating into the STARTMERGE submachine of SIMPLEMERGE a choice among two or more branches that try to converge simultaneously, using one of the selection patterns discussed in Sect. 3.4. In [8] two more variations are discussed under the names *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

This number is furthermore specified to be either "nominated" or "not known until run-time", which is a question of how the number is declared. As to the definition of ThreadMerge, it can be obtained as follows, reusing the instantiation of Merge to Simple Merge and refining the *MergeEv* further by an analogous condition as the one used for AcyclSynchMerge above:

```
ThreadMerge($Activity, exec, MergeEnabled$, Proceed, $MergeNo$) =
  Merge($Activity, exec, MergeEv$, Proceed, **skip**)
where
  $MergeEv = (\mid \{a \in Activity$ **and** $MergeEnabled(exec(a))\} \mid = MergeNo)$
  $mergeStart = mergeCompl$
```

Thus SimpleMerge appears as ThreadMerge with $MergeNo = 1$ under the mutual-exclusion hypothesis.

RelaxSimpleMerge is the variant of ThreadMerge with cardinality check $\mid A \mid \geq 1$ and Proceed refined to **forall** $a \in A$ Proceed(a).[3] At a later point in [10] 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.*"[4]

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 a difference worth the two namings between the Synchronizer and the ThreadMerge pattern besides considering in the latter only the "execution threads in a single branch of the same process instance".

Coupled Split and Join Patterns. For the instantiation of ParallelSplit to the pattern MultInstWithoutSync for multiple instances without synchronization (see Sect. 2.1) three variations appear in [10]. They derive from different interpretations of the static or dynamic nature of the *Mult*itude parameter and from adding a join component to the split feature.

For the *Multiple Instances With a Priori Design Time Knowledge* pattern the set *Mult* is declared to be known a priori at design time. In addition the following is required:

[3] 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).

[4] 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 $a \in A$.

> ... once all instances are completed some other activity needs to be started.

These two requirements can be captured by using the two phases of the MERGE machine, one for the (unconditioned)[5] splitting action and one to PROCEED upon the completion event, when all newly created agents have *Completed* their run of the underlying activity. Since in [8] 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. It can then be specialized to $N = \mid Agent(act) \mid$. The variation *Static Cancelling N-out-of-M Join for Multiple Instances* in [8] can be obtained by adding a cancelling submachine.

```
MULTINSTNMJOIN($act, Mult, Thread, Completed$, TRIGGEREXEC, PROCEED, $N$) =
  MERGE($MultiSet(act, Mult), -, true$,
    MULTINSTWITHOUTSYNC($act, Mult, Thread$, TRIGGEREXEC),
    if $CompletionEv$ then PROCEED)
  where
    $CompletionEv = (\mid \{t \in Thread \mid Completed(t, act)\} \mid \geq N)$

MULTINSTAPRIORIDESIGNKNOWL
  ($act, Mult, Thread, Completed$, TRIGGEREXEC, PROCEED) =
    MULTINSTNMJOIN
      ($act, Mult, Thread, Completed$, TRIGGEREXEC, PROCEED, $\mid Thread(act) \mid$)
```

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" [8, 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 [8] is a variation of Static N-out-of-M Join for Multiple Instances.

The *Complete Multiple Instance Activity* pattern in [8] 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

[5] As a consequence the parameter *exec* plays no role here.

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 $a \in (Thread(act) \setminus Completed)$ do CANCEL($a$)
  PROCEED
```

2.3 Interleaving Patterns

As observed by Andreas Prinz and mentioned above, interleaving should perhaps be considered as parameter for different forms of parallelism and not as pattern. Interleaving is described in [8] as follows:

> 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 $act \in Activity$
  $act$
  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 [8, 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 $curract := schedule(\{a \in Activity \mid$ 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 [9, 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 what it means to Suspend and to Resume an activity and using an abstract predicate *ExecutableRunnable* that filters the currently executable and runnable activities from *Activity*.

```
InterleaveWithSuspension
  ($Activity$, $ExecutableRunnable$, Execute, Suspend, Resume) =
    **choose** $a \in ExecutableRunnable(Activity)$ **if** $a = curract$ **then** Execute($curract$)
      **else**
        Suspend($curract$)
        Resume($a$)
```

The generalization from atomic activities to critical sections, proposed in [8] 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 NextActivity.

There is a large variety of other realistic interpretations of Interleaved Parallel Routing, yielding pairwise different semantical effects. The informal requirement description in [10,8] does not suffice to discriminate between such differences.

2.4 Trigger Patterns

Two basic forms of trigger patterns are discussed in [8], called *Transient Trigger* and *Persistent Trigger*. The description of Transient Trigger reads as follows:

> 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 of this pattern 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'.[6]

The description of the Persistent Trigger goes as follows:

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

Again 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

[6] Note that this safety notion is motivated by the Petri net framework underlying the analysis in [8].

instantiated depending on a) whether at a given moment multiple processes wait for a trigger and on b) 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.[7]

```
Trigger =
  TriggerEvent
  TriggerReaction
where
  TriggerEvent = if $Event(Trigger(a))$ then Buffer($a$)
  TriggerReaction =
    if not $Empty(Buffered \cap Ready)$ then
      choose $A \subseteq Buffered \cap Ready$  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. 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.

```
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.

3 Sequential Control Flow Patterns

The patterns related to sequential control flow can be conceptually categorized into four types: *sequencing* of multiple flows, *iteration* of a flow,

[7] This convention allows us to suppress the explicit deletion of an event from the set of active events.

begin/termination of a flow and *choice* among (also called sequential split into) multiple flows. These patterns capture aspects of process control that are well known from sequential programming.

3.1 Sequence Patterns

We find the following description for this well-known control-flow feature:

"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. If one wants to hide those initial and final control states, one can use the **seq**-operator defined in [5] for composing an ASM A_1 **seq** A_2 out of component ASMs A_i $(i = 1, 2)$.

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

A related pattern is described as follows under the name *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 can be translated as follows:

```
$\textsc{Milestone}(\mathit{milestone}, \mathit{Reached}, \mathit{Expired}, \mathit{act}) =$
  $\textbf{if } \mathit{Reached}(\mathit{milestone}) \textbf{ and not } \mathit{Expired}(\mathit{milestone}) \textbf{ then } \mathit{act}$
```

3.2 Iteration Patterns

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 [8] that the cycles may "have more than one entry or exit point" is a matter of further specifying the starting points and the *StopCriterion* for activities, e.g. exploiting initial and final control states of control-state ASMs. The Iterate construct defined for ASMs in [5] yields a direct formalization of this pattern that hides the explicit mentioning of entry and exit points.

```
$\textsc{ArbitraryCycles}(\mathit{Activity}, \mathit{StopCriterion}) =$
  $\textbf{forall } a \in \mathit{Activity} \ \textsc{Iterate}(a) \textbf{ until } \mathit{StopCriterion}(a)$
```

In [8] two further 'special constructs for structured loops' are introduced, called Structured Loop and Recursion. The formalization of StructuredLoop comes up to the constructs **while** *Cond* **do** *M* respectively **do** *M* **until** *Cond*, defined for ASMs in [5]. For an ASM formalization of RECURSION we refer to [4] and skip further discussion of these well known programming constructs.

3.3 Begin/Termination Patterns

In [10] 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 should TERMINATE 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" [8, 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 [8] 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 *CompletedSucc*essfully.

Related to termination patterns are the so-called cancellation patterns.

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:

```
$\textsc{CancelAct}(act, Agent, exec) =$
  **let** $a = exec(act)$ **in if** $Enabled(a)$ **then** $\textsc{Delete}(a, Agent)$
```

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,[8] 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*.

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

For the *Cancel Region* pattern we find the following description in [8]: "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 [8] 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 [8]: "Within a given process instance, multiple instances of an activity can be created. The required number of instances is known at design time. These 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.

3.4 Selection Patterns

A general workflow selection pattern named *Multichoice* is described in [8] as follows:

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

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

Besides the parameter for the set *Activity* of subprocesses among which to choose, we see here as second parameter a *ChoiceCriterion*,[9] used to "choose multiple alternatives from a given set of alternatives" that have to be executed together. It may take workflow control data as arguments. Using the non deterministic **choose** construct for ASMs yields the following formalization:

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

An equivalent wording for this machine explicitly names a choice function, say *select*, which applied to *Activity* ∩ *ChoiceCriterion* yields a subset of activities chosen for execution:

```
$\textsc{Choice}(Activity, ChoiceCriterion, select) =$
  **forall** $act \in select(Activity \cap ChoiceCriterion)$
    $act$
```

The *Exclusive Choice* pattern is described in [8] as follows, where the additional assumption is that each time an exclusive choice point is reached (read: ExclChoice is executed), the decision criterion yields exactly one $a \in Activity$ that fulfills it:

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

This is a specialization of Choice where the range of the *select* function is requested to consist of singleton sets.

We also find the following description of a *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.

This is captured by an instance of ExclChoice ASM where the *ChoiceCriterion* is declared to be a monitored predicate because the decision for the choice may depend on runtime data.

[9] The revised version of the multi-choice pattern in [8, 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".

4 Conclusion and Outlook

We have identified a few elementary workflow patterns that help to structure the variety of individually named workflow patterns collected in [10,8]. We hope that this provides a basis for an accurate analysis and evaluation of practically relevant control-flow patterns, in particular in connection with business processes and web services, preventing the pattern variety to grow without rational guideline.

Acknowledgement. We thank Andreas Prinz and three anonymous referees for valuable criticism of previous versions of this paper.

References

1. Börger, E.: The ASM ground model method as a foundation of requirements engineering. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 145–160. Springer, Heidelberg (2003)
2. Börger, E.: The ASM method for system design and analysis. A tutorial introduction. In: Gramlich, B. (ed.) Frontiers of Combining Systems. LNCS (LNAI), vol. 3717, pp. 264–283. Springer, Heidelberg (2005)
3. Börger, E.: A critical analysis of workflow patterns. In: Prinz, A. (ed.) ASM 2007, Grimstadt (Norway) (June 2007), Agder University College (2007)
4. Börger, E., Bolognesi, T.: Remarks on turbo ASMs for computing functional equations and recursion schemes. In: Börger, E., Gargantini, A., Riccobene, E. (eds.) ASM 2003. LNCS, vol. 2589, pp. 218–228. Springer, Heidelberg (2003)
5. Börger, E., Schmid, J.: Composition and submachine concepts for sequential ASMs. In: Clote, P.G., Schwichtenberg, H. (eds.) CSL 2000. LNCS, vol. 1862, pp. 41–60. Springer, Heidelberg (2000)
6. Börger, E., Stärk, R.F.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
7. INMOS. Transputer Implementation of Occam – Communication Process Architecture. Prentice-Hall, Englewood Cliffs, NJ (1989)
8. Russel, N., ter Hofstede, A., van der Aalst, W.M.P., Mulyar, N.: Workflow control-flow patterns. A revised view. BPM-06-22 (July 2006), at `http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/`
9. Stärk, R.F., Schmid, J., Börger, E.: Java and the Java Virtual Machine: Definition, Verification, Validation. Springer, Heidelberg (2001)
10. van der Aalst, W.M., ter Hofstede, A., Kiepuszewski, B., Barros, A.: Workflow patterns. Distributed and Parallel Databases 14(3), 5–51 (2003)