Pattern-based Analysis of Windows Workflow

Marco Zapletal* †, Wil M. P. van der Aalst†, Nick Russell†, Philipp Liegl*, Hannes Werthner*
*Vienna University of Technology, Austria
Email: marco@ec.tuwien.ac.at, liegl@big.tuwien.ac.at, werthner@ec.tuwien.ac.at
† Eindhoven University of Technology, The Netherlands
Email: w.m.p.v.d.aalst@tue.nl, n.c.russell@tue.nl

Abstract

The Windows Workflow Foundation (WF) has been introduced as part of the .NET framework as a means of creating workflow-centric applications. Its intended field of application is broad, ranging from fat-client applications and web applications to enterprise application integration solutions. Unlike other approaches, Windows Workflow supports two distinct approaches to workflow specification - sequential workflows and state machine workflows - which deal with fundamentally different types of business scenarios. To date there has been minimal investigation into its capabilities and limitations, especially with respect to the two different control-flow styles it offers. To remedy this, in this paper we present a rigorous analysis of Windows Workflows's ability to deal with common control-flow scenarios. As a framework for this evaluation we use the Workflow Patterns. Our analysis outlines the strength and shortcomings of Windows Workflow's control-flow expressiveness and compares it to BPEL and jBPM - two other popular approaches for the design and implementation of business processes in a service-oriented context.

1. Motivation

The Windows Workflow Foundation is an emerging approach for creating workflow-centric applications in .NET. It enables developers to capture common business scenarios by providing a graphical and declarative approach for modeling business processes. It supports the realization of highly automated processes as well as workflows requiring human interaction. The Windows Workflow Foundation was released in 2006 and since then it has achieved significant popularity in both the business and academic communities. Several books and publications have been released about Windows Workflow, however, they mainly focus on fundamental aspects of its operation and configuration. This paper goes beyond the basic application of Windows Workflow and takes a look behind the scenes, analyzing its soundness and completeness as a language for workflow support. For our analysis we utilize the Workflow Patterns, a well established basis for evaluating workflow languages.

This research is conducted in the context of the Patterns for Process-Aware Information Systems (P4PAIS) project which is supported by the Netherlands Organisation for Scientific Research (NWO)

Workflow control-flow patterns [1] were established by Van der Aalst et. al as part of the Workflow Patterns Initiative. The goal of this initiative is to provide a conceptual basis for process technology. This research aims at analyzing the various perspectives of process technologies i.e. control-flow, data, resource, and exception handling. The original set of 20 control-flow patterns was later extended to a total of 43 patterns [2]. Since their inception, workflow patterns have gained considerable attention from the academic community, businesses, and tool vendors. The expressiveness of a wide variety of business process modeling languages and workflow systems [3]–[8] have been examined by analyzing their support for the workflow patterns.

Due to the recent emergence of Windows Workflow, no research has yet been conducted to assess its support for the workflow control-flow patterns. In this paper we analyze Windows Workflow and examine its support for the workflow patterns. In our research we specifically focus on control-flow patterns and concentrate on implementation details rather than conducting a high level process language analysis [9], [10]. We show, that although Windows Workflow provides a well-engineered framework, many of the proposed patterns are not supported.

The remainder of this paper is structured as follows: Section 2 gives an overview of Windows Workflow and introduces the main concepts necessary for understanding the pattern analysis. Section 3 analyzes the different workflow control-flow patterns in detail and gives an assessment of the level of support Windows Workflow provides for the patterns. Section 4 concludes the paper by comparing our research results with those for the comparable workflow languages BPEL and jBPM.

2. The Windows Workflow Foundation

The Windows Workflow Foundation is a technology introduced by Microsoft for the definition and execution of workflows. Since 2006 it has been part of the .NET Framework. The key concepts of the architecture of the

Windows Workflow Foundation are depicted in Figure 1. Note, that we use the term *Windows Workflow Foundation* to describe the overall technology and by *Windows Workflow* we refer to the workflow language itself. A thorough description of the technology can be found in [11] and [12]. Workflows run within a host process which can be

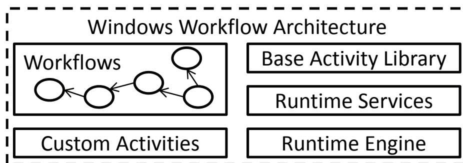

Figure 1: Overview of the Windows Workflow Foundation

any type of .NET application. The runtime engine provides intrinsic behavior for supporting activities e.g. workflow execution, state management etc. Runtime services are optional components, which realize functionality such as workflow persistence or workflow tracking/logging out-of-the-box. The basic building blocks of a workflow are activities. Windows Workflow distinguishes between the *base activity library* and *custom activities*. The former represent an out-of-the-box library of activities for common purposes such as control flow activities (*sequence*, *parallel*, *if/else* etc.), transaction activities (*transaction scope*, *compensable transaction scope*), error handling (*throw*, *fault handler*), lifetime activities (*suspend*, *terminate*), event waiting activities (*listen*, *delay*) and web service activities. From .NET 3.5 onwards, the Windows Communication Foundation (WCF) can be utilized in Windows Workflow for performing distributed communication (e.g., using Web Services, .NET Remoting, Message Queues, etc.) by means of the *send activity* and the *receive activity*. The simplest activity is a *code activity*, where a user can easily add any .NET code interacting with the workflow. In order to foster reuse of workflows users can specify *custom activities*, which implement specific business logic in a self-contained and reusable manner (e.g. persist order). Windows Workflow supports two workflow paradigms: *sequential* and *state-machine* workflows. A sequential workflow describes a workflow in a prescriptive manner using a series of consecutive execution directives. In general sequential workflows are more rigid than state-machine workflows. Although loops and branches with conditions can be used, the execution path through the workflow is basically deterministic. Typically sequential workflows are used in scenarios where a rigid protocol has to be applied and little or no human or any other external interaction with the workflow takes place. The sequential approach is predominant in contemporary process languages and workflow systems and for example is used in BPEL [13] or jPDL [14]. In addition, Windows Workflow supports the concept of state machine workflows, which are driven by external events, controlling the processing order of the workflow. A state machine workflow consists of states and transitions between the different states. The execution of the workflow is governed by events that are either released by the workflow itself or by an external source such as human interaction, which fires the transitions between states. State machine workflows are a hybrid approach between a state-based approach and the sequential workflow style. The latter one is used within states in order to describe the execution logic after events have been triggered. The usage scenarios for Windows Workflow are diverse ranging from regular application development to building middleware solutions - e.g. based on Microsoft's BizTalk server [15], [16]. In the upcoming version of BizTalk (2009), business processes will be modeled using Windows Workflow. This involves scenarios such as business-to-business communication, message broking or enterprise application integration, depending on the scenario the BizTalk server is used in.

3. The Workflow Patterns in Windows Workflow

In this section, we analyze the support of the Windows Workflow Foundation to implement the 43 workflow control-flow patterns. For each pattern we discuss if it is directly supported, partially supported or not supported by Windows Workflow. For most of the solutions, we provide examples in XAML in addition to the textual descriptions of the solutions. XAML (pronounced [za:ml]) is an XML-based language to describe the structure of Windows Workflow models - an alternative approach to regular .NET code. For the sake of readability, we will simplify the XAML code in order to emphasize the most important aspects.

Although Windows Workflow is a declarative and graphical workflow language, it is essentially an abstraction of the underlying .NET code. As outlined in Section 2, developers may execute custom .NET code by creating their own custom activities targeting specific business needs. Consequently, each of the identified workflow patterns could be implemented in Windows Workflow by developing a corresponding custom activity. For this analysis, however, we only consider the *base activity library* of the Windows Workflow Foundation.

In our evaluation we focus on solutions using the sequential workflow paradigm of Windows Workflow. However, some patterns are not realizable using the sequential style and require a state-based approach. In such cases, if a pattern is implemented using the state machine workflow style of Windows Workflow, we consider it to be fully supported. Due to space limitations, we do not discuss state machine-based solutions for patterns where sequential workflows already provide an appropriate solution.

3.1. Basic Workflow Patterns

WCP1 Sequence. An activity in a process is enabled after the completion of the preceding activity in the same process.

Solution: WCP1. This pattern is directly supported by the *sequence* activity. Listing 1 shows the XAML representation of a *sequence* activity containing two child activities. The construct is often used as an auxiliary construct. Since most of the composite activities allow only one child activity, a *sequence* activity may be used for nesting a series of activities.

Listing 1: Sequence (WCP1)

```
1 <SequentialWorkflowActivity>
2   <CodeActivity Name="activityA1" />
3   <CodeActivity Name="activityA2" />
4 </SequentialWorkflowActivity>
```

WCP2 Parallel Split. A branch in a process forks into two or more branches, which are executed concurrently.

WCP3 Synchronization. Two or more branches are merged into a single subsequent branch. The thread of control is passed to the subsequent branch as soon as all incoming branches have been enabled. Consequently, this pattern synchronizes the threads of the incoming branches into a single thread of control. The synchronized threads must belong to the same process instance.

Solutions: WCP2 & WCP3. Arbitrary activities are executed concurrently by including them in branches of a *parallel* activity (lines 1 to 8 in Listing 2). There is no limitation to the number of branches within the parallel activity. Each branch of a *parallel* activity is modeled as a *sequence* activity. The *parallel* activity waits for all branches to finish execution (line 8) before passing the thread of control to the subsequent activity (line 9), thereby realizing the synchronization pattern.

Listing 2: Parallel Split (WCP2) and Synchronization (WCP3)

```
1 <ParallelActivity Name="P">
2   <SequenceActivity Name="S1">
3     <CodeActivity Name="A1" />
4   </SequenceActivity>
5   <SequenceActivity Name="S2">
6     <CodeActivity Name="A2" />
7   </SequenceActivity>
8 </ParallelActivity>
9 <CodeActivity Name="A3" />
```

It is important to note that Windows Workflow uses a single thread model for executing workflow instances - i.e., only a single thread executes within a workflow instance at any time. Consequently, there is no real parallel execution of activities across multiple branches. Parallel behavior is in fact realized by the workflow runtime by scheduling activities for execution in an interleaved manner. In other words, the branches of a *parallel* activity are executed by interleaving their execution. The order of execution is non-deterministic. If there is a blocking activity in a certain branch (e.g., a *delay* activity), the next activity of another branch is scheduled for execution.

WCP4 Exclusive Choice. A branch is split into two or more subsequent branches. Using a decision mechanism exactly one of the subsequent branches is enabled (i.e., an XOR-split).

WCP5 Simple Merge. The fusion of two or more branches into a single subsequent branch without synchronizing them. The simple merge is generally used to merge branches that resulted from an exclusive choice. Therefore, the simple merge assumes that only one of the alternative branches is enabled.

Solutions: WCP4 & WCP5. The exclusive choice and simple merge patterns are supported by the *if/else* activity (cf. Listing 3). Within an *if/else* activity two or more branches may be nested. Each branch except the last one must have an associated condition. Conditions are evaluated sequentially starting from the first modeled branch to the last one. The first branch, whose condition evaluates to true, is executed. All remaining branches are canceled. The condition of the last branch is optional, which corresponds to the semantics of *else*, i.e., it is the default branch.

Listing 3: Exclusive Choice (WCP4) and Simple Merge (WCP5)

```
1  <IfElseActivity>
2    <IfElseBranchActivity Name="B1">
3      <IfElseBranchActivity.Condition>
4        <RuleConditionReference ConditionName="C1"/>
5      </IfElseBranchActivity.Condition>
6      <CodeActivity Name="A1" />
7    </IfElseBranchActivity>
8    <IfElseBranchActivity Name="B2">
9      <CodeActivity Name="A2" />
10   </IfElseBranchActivity>
11 </IfElseActivity>
12 <CodeActivity Name="A3"/>
```

3.2. Advanced Branching and Synchronization Patterns

WCP6 Multi-Choice. The divergence of one branch in a process into two or more subsequent branches. Based on a certain decision mechanism one or more of the alternative branches are enabled, which corresponds to the behavior of an OR-split.

WCP7 Structured Synchronizing Merge. The merger of two or more branches that originated from a previous *multi-choice* into a single subsequent branch. Depending on the *multi-choice* one or more incoming branches are executed. This pattern waits for all active incoming branches to complete before passing control to the subsequent activity.

Solutions: WCP6 & WCP7. In Windows Workflow support for these patterns is achieved via the *conditioned activity group activity* (CAG). A CAG may comprise one or many child activities. Child activities may have a *when* condition defined. Where a child activity has no when condition, it is executed exactly once. Otherwise, it is executed for long as the *when* conditions remains true. In addition, a CAG may have an *until* condition. If the *until* condition is present, the execution stops as soon as it evaluates to true. Upon initialization of a CAG, first of all the *until* condition is evaluated. If false, the *when* condition for each child is evaluated. When a child activity completes execution, the *until* condition as well as the *when* conditions of all other children that are currently pending are re-evaluated. As a result the completion of one child may influence the execution of another child. If no *until* condition is specified, the CAG completes when all child activities are completed. The different combinations of *when* and *until* conditions allow for different styles of execution of the CAG - e.g., parallel execution and looping behavior. For implementing the *multi-choice* and the *structured synchronizing merge* patterns, each diverging branch is nested as a child activity within the CAG. In Listing 4, the CAG comprises three branches (lines 2 to 6, 7 to 11, and 12 to 16, respectively). The *when* conditions are used to reflect the conditional activation of a branch. To ensure that each child is executed exactly once, each *when* condition has an additional Boolean variable assigned that indicates if a child has already been executed. For example, condition *C1* (line 4) may be defined as *condA && !aExecuted*, where *condA* corresponds to the logic for determining upon the execution of *activityA* and *aExecuted* is a Boolean variable. The Boolean is set to false at the initialization of the CAG and has to be set to true during the execution of *activityA*. Since no *until* condition is specified for the CAG, it waits until each child activity that has been activated has completed execution. Only then does it pass the thread of control to the next activity.

The CAG is a structured concept - metaphorically speaking the beginning of the CAG corresponds to the *multi-choice* pattern and its end to the *structured synchronizing merge*. A branch of a CAG has exactly one entry and one exit point. No additional entry/exit points to/from the branches of a CAG are possible.

Listing 4: Multi-Choice (WCP6) and Structured Synchronizing Merge (WCP7)

```
1  <ConditionedActivityGroup>
2    <CodeActivity Name="activityA">
3      <ConditionedActivityGroup.WhenCondition>
4        <RuleConditionReference ConditionName="C1" />
5      </ConditionedActivityGroup.WhenCondition>
6    </CodeActivity>
7    <CodeActivity x:Name="activityB">
8      <ConditionedActivityGroup.WhenCondition>
9        <RuleConditionReference ConditionName="C2" />
10     </ConditionedActivityGroup.WhenCondition>
11   </CodeActivity>
12   <CodeActivity x:Name="activityC">
13     <ConditionedActivityGroup.WhenCondition>
14       <RuleConditionReference ConditionName="C3" />
15     </ConditionedActivityGroup.WhenCondition>
16   </CodeActivity>
17 </ConditionedActivityGroup>
18 <CodeActivity Name="activityD" />
```

WCP8 Multi-Merge. Two or more branches are merged into a single subsequent branch, whereby each enabled incoming branch passes a thread of control to the subsequent branch. As a result, more than one thread is active within the subsequent branch.

Solution: WCP8. There is no support for this pattern in Windows Workflow due to its single-threaded nature (cf. WCP2). A state machine workflow instance can only reside in a single state at a given point of time. The block-oriented structure of sequential workflows prevents two threads of control from being active along the same path in a single process instance.

WCP9, WCP28 & WCP29 Discriminator Patterns. The discriminator patterns merge two or more branches, which result from a *parallel split* into a single subsequent branch. When the first incoming branch is enabled the thread of control is passed to the subsequent branch. All further incoming branches have no effect and are ignored. The *structured discriminator pattern* (WCP9) lets the remaining branches complete. The *blocking discriminator* (WCP28) is intended for environments that allow multiple execution threads within the same process instance. It blocks further enabled incoming branches from entering a specified region after it has activated. A reset is performed when all incoming branches to the blocking discriminator have been enabled once. The *canceling discriminator* (WCP29) differs from the *structured discriminator* in that it cancels the remaining branches instead of allowing them to complete.

Solutions: WCP9, WCP28 & WCP29. Discriminator behavior is to some extent realizable by a *conditioned activity group* (CAG) together with an optional *until* condition. Furthermore, a Boolean variable is required to indicate if one of the branches has already finished execution. The first child activity to finish manipulates the Boolean variable (e.g., named *finished*), so that the *until* condition of the CAG evaluates to true. The CAG immediately cancels all remaining branches and passes the thread of control to the subsequent activity. The solution provides full support for the *canceling discriminator pattern* (WCP29). The evaluation criteria for the *structured discriminator* (WCP9) rates canceling the activities instead of letting them complete as partial support. The *blocking discriminator* (WCP28) requires that multiple concurrent threads are executed in the same process instance. Due to the single-threaded nature of Windows Workflow, there is no concept for blocking one of multiple threads.

WCP30 – WCP32 Partial Join Patterns. The partial join patterns correspond to a generalization of the semantics of the discriminator patterns (WCP9, WCP28 & WCP29), where the join fires after two or more incoming branches are enabled depending on a given condition. In fact, the join is

activated when n out of the total number (m) of incoming branches are enabled (where $2 \leq n < m$).

The *structured partial join* (WCP30) allows the remaining branches, which are not required to trigger the join, to complete whereas the *canceling partial join* (WCP32) pattern terminates the remaining branches. Finally, the *blocking partial join* (WCP31) behaves like the *blocking discriminator* (WCP28) except that it joins two or more active branches.

Solutions: WCP30 – WCP32. The implementation of the partial join patterns in Windows Workflow is similar to the discriminator patterns. In fact, they differ only in the specification of the *until* condition and the need for Boolean or counter variables: If the join behavior requires specific branches to complete for the join to fire (e.g.,$(A\&\&B)||C$), then each branch requires a Boolean variable to indicate whether it has finished execution or not. Otherwise, if a specific number of enabled incoming branches trigger the join, a single counter variable is needed. Evidently, the corresponding join semantics have to be reflected in the *until* condition of the CAG.

The support for this pattern is similar to that for the discriminator patterns: There is partial support for the *structured partial join* (WCP30) and full support for the *canceling partial join* (WCP32). The *blocking partial join* is not supported (WCP31).

WCP33 Generalized AND-Join. The generalized AND-join fires after all incoming branches have been enabled. In contrast to the synchronization pattern (WCP3) further triggers on incoming branches between two firings are retained for a later activation of the join. This conserves triggers on incoming branches of an AND-join in situations where multiple threads operate on the same process instance.

Solution: WCP33. Since it is not possible in Windows Workflow for multiple threads to execute in a single process instance this pattern is not supported.

WCP37 Local Synchronizing Merge. The merger of two or more branches that have split earlier into a single subsequent branch. The subsequent branch is enabled after each *active* incoming branch has been enabled. The *local synchronizing merge* must determine how many branches are active and, hence, require synchronization. In general, the pattern does not assume a structured process model. Furthermore, the synchronization decision is based on information that is locally available to the merge construct.

Solution: WCP37. We consider this pattern to be partially supported by the join capabilities of the conditioned activity group (CAG). Full support is not possible as the branches of a CAG can only contain structured process models.

WCP38 General Synchronizing Merge. Two or more alternative branches that have diverged earlier are synchronized into a single subsequent branch such that the outgoing branch can be enabled when each active incoming branch has completed execution. The pattern provides a general approach to OR-join implementation [2], thereby relying on non-local semantics [17]. The pattern assumes that the process model is non-structured and may have arbitrary looping structures. In addition, there may be multiple entry- and/or exit points to/from the branches that are subject to synchronization. The merge is triggered when all active incoming branches are enabled and it is determined that none of the remaining branches will be enabled at any future time. Determination of when to activate the merge involves the calculation of possible future states of the whole process instance, which is a complex and costly computation. For a detailed elaboration of WCP37 and WCP38 (as well as differences between them) we refer the interested reader to the workflow pattern descriptions in [2].

Solution: WCP38. This pattern can neither be realized using the sequential workflow style nor via the state machine-based approach. The latter is not able to support the pattern due to the single-threaded model of Windows Workflow and the fact that only one state can be executed at a given time. Furthermore, the *general synchronizing merge* cannot be captured by the CAG, since this construct requires its child activities to form a structured process fragment.

WCP41 Thread Merge. At a certain point within a process, a designated number of concurrent execution threads in a single branch in a process instance are merged into a single execution thread.

WCP42 Thread Split. At a given point in a branch, the execution thread is split into a specified number of concurrent execution threads.

Solutions: WCP41 & WCP42. These patterns are not supported in Windows Workflow, since it is not possible to spawn or merge multiple threads in a single workflow instance. A minimalistic workaround may be achieved by triggering the number of required execution instances outside of the workflow as shown in Listing 5.

Listing 5: Thread Split (WCP41) and Thread Merge (WCP42)

```
1  <CallExternalMethodActivity Name="C1" MethodName="m1" />
2  <CodeActivity Name="A1" />
3  <CodeActivity x:Name="A2" />
4  <HandleExternalEventActivity Name="H1" EventName="e1" />
```

The *call external method* activity in line 1 of Listing 5 interacts with the host application. The call corresponds to a synchronous method invocation that creates a thread of execution. The workflow continues immediately with the execution of the activities *A1* and *A2*. The *handle external*

event activity in line 4 then waits for an event raised by the host, which "merges" the externally created thread into the execution thread of the workflow instance. Where multiple workflow instances are waiting for the same type of event, correlation with the appropriate instance is assured since the event carries the unique ID of the workflow instance. A similar workaround is achievable by using the *send activity* / *receive activity* for creating threads of execution on distributed components.

3.3. Multiple Instance Patterns

WCP12 Multiple Instances without Synchronization. At a given point in the execution of a process multiple instances of a task are initiated. The instances run concurrently and are independent of each other. Upon completion of the task instances no synchronization behavior is required. The number of task instances that are initiated is known at design time.

WCP13 – WCP15 Multiple Instances with Synchronization. The multiple instances with synchronization patterns describe the creation of multiple instances of a task at a given point within a single workflow case. All three patterns require synchronization after the completion of all instances before commencing any subsequent activities. In WCP13 the number of task instances is known at design-time. WCP14 requires that number is known at run-time before the instances are started. In WCP15 the number of instances to be started is not known, but as long as instances are running new ones can be initiated.

Solutions: WCP12, WCP13 & WCP14. The realization of multiple instance behavior in Windows Workflow requires the *replicator* activity. The *replicator* creates multiple instances of its child activity at run-time. The number of instances to be created is determined through an input collection (i.e., an object that implements the .NET interface *System.Collections.IList*). By default a *replicator* stops when all instances of its child activity have finished. Using the optional *until* condition a *replicator* can even stop before all of the child activity's instances have finished. *Replicators* can be used in sequential or parallel mode. In the former case the different child activity instances are executed one after the other. In the latter replicated child activities are executed in parallel, causing all instances of the child activity to be created once the replicator commences execution. Regardless of the execution mode used, the *replicator* waits for each instance to complete before passing on control to subsequent activities in the workflow, which realizes the required synchronization behavior.

Listing 6 shows a simple example using the replicator. The number of instances that are created of the *code* activity *A1* is determined by the size of the collection *aList*. The collection can be manipulated (i.e., the complete set of items or a specific item) until the *replicator* activity is enabled. This behavior provides support for *multiple instances without synchronization* (WCP12), *multiple instances with a priori design-time knowledge* (WCP13) and *multiple instances with a priori run-time knowledge* (WCP14) patterns. In regard to WCP12, the instances are synchronized although this is not explicitly required by the pattern description.

Listing 6: Multiple Instances with a Priori Design-Time(WCP13)/Run-Time Knowledge (WCP14)

```
1 <ReplicatorActivity InitialChildData="aList" ExecutionType="Parallel">
2   <CodeActivity Name="A1" />
3 </ReplicatorActivity>
```

Solution: WCP15. This pattern is not realizable in Windows Workflow. As soon as the *replicator* is enabled, its input collection is processed in order to determine the required instances. Although the collection itself can be manipulated during the execution of the *replicator* (e.g., *aList* in the example in Listing 6) this has no effect on the instances once they have been created.

WCP34 – WCP36: Partial Joins for Multiple Instances. The partial join patterns describe different alternatives for passing the thread of control to the subsequent activity after multiple instances of a task have been created. In WCP34 (*static partial join for MI*), the number of instances (m) is known when the first instance is started. The join condition specifies how many instances (n, where $n < m$) have to completed in order to pass the thread of control to the subsequent task. The remaining instances complete, but do not have any effect on the control flow. WCP35 (*canceling partial join for MI*) is similar to WCP34, except that any remaining instances are canceled. In WCP36 (*dynamic partial join for MI*) the number of instances to be initiated is not known until the last instance completes. New instances may be added as long as at least one instance is still executing and creation of further instances has not been disabled. Upon completion of each instance, the join condition is re-evaluated. If it evaluates to true, control is passed to the subsequent activity. Completion of any remaining instances has no effect on the process instance.

Solutions: WCP34 & WCP35. Partial join behavior is to some extent supported by the synchronization capabilities of the *replicator*. The join semantics are specified using the optional *until* condition of the *replicator*. The *until* condition is evaluated upon activation of the *replicator* as well as after completion of an instance. As soon as the *until* condition evaluates to true, the instances that are still executing are canceled. This constitutes full support for the *canceling partial join for multiple instances* (WCP35). Similar to the evaluation of the *structured partial join* (WCP30), we also

consider this to be partial support for the *static partial join for multiple instances* (WCP34).

Solution: WCP36. This pattern is a variant of *multiple instances without a priori run-time knowledge* (WCP15) and is, thus, also not supported by the offering.

3.4. State-based Patterns

WCP16 Deferred Choice. At a given point in a process one of several subsequent branches is activated based on a certain event, which occurs in the operating environment of the process. The event results in the activation of the first activity in one the alternative branches. Once this occurs, any alternate activities in other branches are withdrawn. Thus, there is a race between the different branches rather than a choice based on some condition.

Solution: WCP16. This pattern is directly supported by the *listen* activity (see listing 7). The *listen* activity may contain two or more branches represented by *event driven* activities. The first activity within an *event driven* activity is the receiver of the event (note: it must implement the *System.Workflow.Activities.IEventActivity* interface). The *event driven* activity is simply a wrapper that is required by the listen activity (and some other composite activities).

In Listing 7, the process waits at the *listen* activity (line 1) either to receive a message (line 3) or until a delay of one minute in line 6 passed representing a timeout. Whichever event occurs first is taken.

Listing 7: Deferred Choice (WCP16)

```
1 <ListenActivity Name="L1">
2   <EventDrivenActivity Name="ED1">
3     <ReceiveActivity Name="R1" ServiceOperationInfo="S1.OP1" />
4   </EventDrivenActivity>
5   <EventDrivenActivity Name="ED2">
6     <DelayActivity TimeoutDuration="00:01:00" Name="D1" />
7   </EventDrivenActivity>
8 </ListenActivity>
```

WCP17 Interleaved Parallel Routing. A set of activities in a process must be executed in some partial order. Some of the activities may be ordered, others may be executed in an arbitrary order. Each activity must be executed exactly once. Furthermore, only one of these activities must be active at the same time.

Solution: WCP17. This pattern is not realizable using sequential workflows without serious restrictions. The only possibility is modeling all possible execution sequences in the process model, so that one of the execution sequence may be chosen at runtime.

An effective solution for this pattern is provided by Windows Workflow's state machine workflow style. Listing 8 shows a state machine workflow composed of the five *state* activities A to E. Based on a set of possible events in each state, changes to other states occur (realized by *set state* activities). In the example, we indicate the event handling exemplarily by *event driven* activities and *handle external event* activities. In consideration of the fact that each state must be executed once, the possible execution sequences are ABCDE, ABDCE, and ACBDE. In order ensure that each state is not executed more than once, some programmatic extensions are required in order to mark already executed states (e.g., by a Boolean variable). As a certain event occurs, the state machine must check if the designated state has not already been executed (e.g, by an *if/else* activity before the *set state* activity), which is not reflected in this example. Despite the programmatic extensions, the pattern is considered to be fully supported based on the evaluation criteria for this pattern.

Listing 8: Interleaved Parallel Routing (WCP17)

```
1 <StateMachineWorkflowActivity>
2   <StateActivity Name="A">
3     <EventDrivenActivity Name="fromAtoB">
4       <HandleExternalEventActivity Name="H1" EventName="AtoB" />
5       <SetStateActivity TargetStateName="B" />
6     </EventDrivenActivity>
7     <EventDrivenActivity Name="fromAtoC">
8       <HandleExternalEventActivity Name="H2" EventName="AtoC" />
9       <SetStateActivity TargetStateName="C" />
10    </EventDrivenActivity>
11  </StateActivity>
12  <StateActivity Name="B">
13    <EventDrivenActivity Name="fromBtoC">
14      <HandleExternalEventActivity Name="H3" EventName="BtoC" />
15      <SetStateActivity TargetStateName="C" />
16    </EventDrivenActivity>
17    <EventDrivenActivity Name="fromBtoD">
18      <HandleExternalEventActivity Name="H4" EventName="BtoD" />
19      <SetStateActivity TargetStateName="D" />
20    </EventDrivenActivity>
21  </StateActivity>
22  <StateActivity x:Name="C">
23    <EventDrivenActivity Name="fromCtoD">
24      <HandleExternalEventActivity Name="H5" EventName="CtoD" />
25      <SetStateActivity TargetStateName="D" />
26    </EventDrivenActivity>
27    <EventDrivenActivity Name="fromCtoE">
28      <HandleExternalEventActivity Name="H6" EventName="CtoE" />
29      <SetStateActivity TargetStateName="E" />
30    </EventDrivenActivity>
31    <EventDrivenActivity Name="fromCtoB">
32      <HandleExternalEventActivity Name="H7" EventName="CtoB" />
33      <SetStateActivity TargetStateName="B" />
34    </EventDrivenActivity>
35  </StateActivity>
36  <StateActivity x:Name="D">
37    <EventDrivenActivity Name="fromDtoC">
38      <HandleExternalEventActivity Name="H8" EventName="DtoC" />
39      <SetStateActivity TargetStateName="C" />
40    </EventDrivenActivity>
41    <EventDrivenActivity Name="fromDtoE">
42      <HandleExternalEventActivity Name="H9" EventName="DtoE" />
43      <SetStateActivity TargetStateName="E" />
44    </EventDrivenActivity>
45  </StateActivity>
46  <StateActivity Name="E" />
47 </StateMachineWorkflowActivity>
```

WCP18 Milestone. A task in a process can only be activated when a process instance is in a specific state. Prior or subsequent to this specific state, this task can not be enabled. Such a state represents a specific execution point in a process known as milestone.

Solution: WCP18. Similar to WCP17, the realization of the milestone pattern requires the state machine workflow style of Windows Workflow (see Listing 9 for an example and Figure 2 for a graphical representation of the state machine).

Listing 9: Milestone (WCP18)

```
1  <StateMachineWorkflowActivity>
2    <StateActivity Name="A">
3      <EventDrivenActivity Name="fromAtoB">
4        <HandleExternalEventActivity Name="H1" EventName="AtoB" />
5        <SetStateActivity TargetStateName="B" />
6      </EventDrivenActivity>
7    </StateActivity>
8    <StateActivity Name="Milestone">
9      <EventDrivenActivity Name="fromMilestoneToD">
10       <HandleExternalEventActivity Name="H4" EventName="MilestoneToD" />
11       <SetStateActivity TargetStateName="D" />
12     </EventDrivenActivity>
13     <StateActivity Name="B">
14       <EventDrivenActivity Name="fromBtoC">
15         <HandleExternalEventActivity Name="H2" EventName="BtoC" />
16         <SetStateActivity TargetStateName="C" />
17       </EventDrivenActivity>
18     </StateActivity>
19     <StateActivity Name="C">
20       <EventDrivenActivity Name="fromCtoE">
21         <HandleExternalEventActivity Name="H3" EventName="CtoE" />
22         <SetStateActivity TargetStateName="E" />
23       </EventDrivenActivity>
24     </StateActivity>
25   </StateActivity>
26   <StateActivity x:Name="stateD">
27     <EventDrivenActivity Name="fromDtoB">
28       <HandleExternalEventActivity Name="H5" EventName="DtoB" />
29       <SetStateActivity TargetStateName="B" />
30     </EventDrivenActivity>
31   </StateActivity>
32   <StateActivity Name="E" />
33 </StateMachineWorkflowActivity>
```

The example consists of five *state* activities (A to E) plus a composite state activity representing the milestone (lines 8 to 25). States B and C are sub-states of the milestone state. Moreover, the milestone state comprises an *event driven* activity (lines 9 to 12), whose first activity (*H4*) is one that listens to an event *MilestoneToD* (line 10). If the process is in either sub-state of the milestone state, the transition to state D can be triggered by receiving the event *MilestoneToD*. If the process is in any other state (A and E in the example), state D can not be reached.

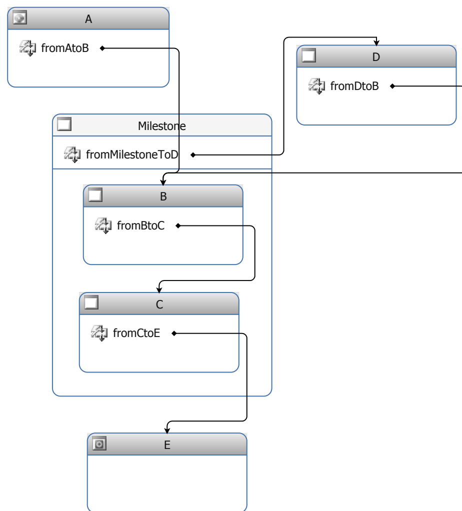

Figure 2: Graphical representation of the WCP18 example

WCP39 Critical Section. Two or more branches of a process form *critical sections* if tasks in only one of those sections can be active at the same time. One *critical section* must complete before another can become active.

Solution: WCP39. This pattern is directly supported by the *synchronization scope* activity. Listing 10 shows the parallel execution of two branches (S1 and S2). Each branch contains a *synchronization scope*. Child activities in a synchronization scope are provided with synchronized and mutually exclusive access to common variables (e.g., *mutex* in Listing 10) shared across multiple *synchronization scopes*. In our example, this ensures that activities A1 and A2 cannot be executed at the same time.

Listing 10: Critical Section (WCP39)

```
1  <ParallelActivity Name="P1">
2    <SequenceActivity Name="S1">
3      <SynchronizationScopeActivity Name="Sync1" SynchronizationHandles="mutex">
4        <CodeActivity Name="A1" />
5      </SynchronizationScopeActivity>
6    </SequenceActivity>
7    <SequenceActivity Name="S2">
8      <SynchronizationScopeActivity Name="Sync2" SynchronizationHandles="mutex">
9        <CodeActivity Name="A2" />
10     </SynchronizationScopeActivity>
11   </SequenceActivity>
12 </ParallelActivity>
```

WCP40 Parallel Routing. A set of tasks is executed in arbitrary order. Each task is executed exactly once and no two tasks in this set are allowed to execute at the same time. Essentially, this pattern relaxes the partial ordering requirement of WCP17.

Solution: WCP40. Although motivated by a different intention, this pattern may be realized by the same approach as introduced for WCP39 before (see Listing 10). The *parallel* activity comprises a branch for each of the interleaved activities. Since only one the interleaved activities is allowed to execute at a given time, each activity is enclosed by a *synchronization scope* activity (accessing the same synchronization handle). This construct allows for an arbitrary execution order, but guarantees at the same time isolated execution of each activity by means of the *synchronization scopes*.

3.5. Cancellation and Force Completion Patterns

WCP19 Cancel Task, WCP20 Cancel Case & WCP25 Cancel Region. *Cancel task* (WCP19) withdraws an activity prior to its execution. If it is already started, the activity is halted. *Cancel region* (WCP25) disables a selected set of activities in a single process instance. These activities need not be a connected sequence. *Cancel case* (WCP20) terminates the execution of a whole process instance.

Solutions: WCP19 & WCP25. In sequential workflows, canceling an activity (WCP19) is supported by using fault handlers. In Windows Workflow, one or more *fault handler* activities may be associated with each composite activity type. Each *fault handler* activity listens to a different type of fault/exception. This achieves the same fault handling approach as is utilized in object-oriented programming

languages (e.g., Java and .NET). However, this construct only partially satisfies the evaluation criteria for WCP25, because activities that are nested within a complex activity form essentially a connected sequence. Thus, only activities within the same scope (i.e., the same complex activity) can be canceled, but not arbitrary activities. State machine workflows provide support for both patterns as demonstrated in the code snippet in Listing 11. State A is a composite state having a sub-state B. Furthermore, state A is an event sink for events of type *e1* (lines 2 to 5), which implies that all sub-states of A react to this event. Where an event of type *e1* is received during the execution of state B (which just delays the process for 10 seconds (line 8)), the execution of state B (and hence the delay activity) is immediately canceled and the process proceeds to state D (line 4; note that state D is not shown in the Listing).

Within state A, arbitrary further states may be nested. The sub-states are not required to form a connected sequence. Each sub-state that executes, however, is canceled when event *e1* is received. Thus, state A corresponds to the concept of a *cancel region* (WCP25).

Listing 11: Cancel Task (WCP19) and Cancel Region (WCP25)

```
1  <StateActivity Name="A">
2    <EventDrivenActivity Name="ED1">
3      <HandleExternalEventActivity Name="H1" EventName="e1"/>
4      <SetStateActivity TargetStateName="D" />
5    </EventDrivenActivity>
6    <StateActivity Name="B">
7      <EventDrivenActivity Name="ED2">
8        <DelayActivity TimeoutDuration="00:00:10" Name="D1" />
9        <SetStateActivity TargetStateName="C" />
10     </EventDrivenActivity>
11   </StateActivity>
12 </StateActivity>
```

Solution: WCP20. *Cancel case* is directly supported by the *terminate* activity. A process instance that is ended by a *terminate* activity is not considered as having ended successfully.

WCP26 Cancel Multiple Instance Activity & WCP27 Complete Multiple Instance Activity. The two patterns build upon the multiple instance patterns that provide synchronization (WCP13–WCP15). At any time during the execution of a multiple instance task it may be necessary to either cancel (WCP26) or forcibly complete (WCP27) the multiple instance task. In the former case, currently running instances are canceled and the thread of execution is not passed on. In the latter case, currently executing instances can complete and the thread of execution is passed to the subsequent activity. In both cases, all instances that have not yet started are withdrawn and any completed instances remain unaffected.

Solutions: WCP26 & WCP27. WCP26 is supported by associating a *fault handler* activity with the *replicator*. When the fault handler is activated any remaining instances, which have not yet completed, are canceled. Since execution should not continue with the subsequent activity, the fault handler must rethrow the fault. There is no means of triggering the completion of multiple instances that are still executing. Thus, there is no support for WCP27.

3.6. Iteration Patterns

WCP10 Arbitrary Cycles. A process may contain cycles having more than one entry or exit point, which results in unstructured loops.

Solution: WCP10. *Arbitrary cycles* are not supported by sequential workflows due to their block orientation. State machine workflows support *arbitrary cycles* with the following restriction: Since concurrent execution of two or more states is not possible, there must not be a parallel branching from one state in a process to multiple subsequent states.

WCP21 Structured Loop. A task or a subprocess may be executed repeatedly. Each time before or after the task is executed a condition is evaluated to determine if the task should be executed again. The looping structure has a single entry and exit point.

Solution: WCP21. Windows Workflow provides a dedicated construct for structured loops in the form of the *while* activity.

WCP22 Recursion. A task may invoke itself at a certain of execution. The parent task instance waits until the child instance has completed.

Solution: WCP22. There is no dedicated construct in Windows Workflow to perform recursive invocations of tasks.

3.7. Termination Patterns

WCP11 Implicit Termination. A process instance completes when there is no remaining work to do. Instances that complete in accordance with the *implicit termination* pattern are deemed to have ended successfully.

Solution: WCP11. *Implicit termination* is supported by sequential workflows. The workflow runtime recognizes when there is no more work to do for a given workflow instance. The instance is then completed and a corresponding event is triggered by the workflow runtime for optional further processing.

With respect to state machine workflows, *implicit termination* is not supported as outlined in the solution for WCP43.

WCP43 Explicit Termination. A process instance completes by reaching a designated endpoint in the process. There may be remaining activities waiting for execution, which are canceled immediately. Regardless of that fact, the process is considered as having completed successfully.

Solution: WCP43. There is no dedicated construct for representing *explicit termination* in sequential workflows. State machine workflows, however, require the definition of exactly one *completed state* per workflow model in order to complete a workflow instance. It is interesting to note that the definition of a completed state is not required to compile and execute a workflow, but an instance with no completed state remains idle if there is no more work to do.

3.8. Trigger Patterns

WCP23 Transient Trigger & WCP24 Persistent Trigger. The execution of a task may be triggered by an internally or externally raised event. A *transient trigger* (WCP23) can only be consumed if there is a task instance waiting for it and it must be processed immediately. Otherwise, it is lost. In contrast, a persistent trigger (WCP24) is durable in nature and is kept until the receiving task acts upon it.

Solution: WCP23. The concept of *transient triggers* is supported by the *handle external event* activity, which receives events from the host. When a workflow instance enables a *handle external event* activity, it blocks until a corresponding event is received. When an event is sent to the workflow and the instance is not in the appropriate state to deal with the event (i.e., a corresponding *handle external event* activity has not yet been activated), the trigger is discarded.

Solution: WCP24. Full support for *persistent triggers* can be achieved through a workaround using the *event handling scope* activity. This activity allows the workflow to respond to events in parallel with the regular process flow. One or more *event handlers* may be added to the *event handling scope* in order to react to different types of event. The event handling scope in Listing 12 waits for an event of type *e1* (line 5). When *e1* is received, it is retained using the *code* activity in line 6 for later processing. Receiving events does not affect the execution of the *delay* activity in line 2.

Listing 12: Persistent Trigger (WCP24)

```
1 <EventHandlingScopeActivity Name="EHS1">
2   <DelayActivity TimeoutDuration="00:01:00" Name="D1" />
3   <EventHandlersActivity Name="EH1">
4       <EventDrivenActivity Name="ED1">
5         <HandleExternalEventActivity Name="H1" EventName="e1" />
6         <CodeActivity Name="A1" />
7       </EventDrivenActivity>
8   </EventHandlersActivity>
9 </EventHandlingScopeActivity>
```

4. Conclusion

It is the basic function of a workflow language to implement the flow of a business process. Consequently, a workflow language is required to support control-flow scenarios that occur in real-world business processes. In order to analyze the capabilities of a workflow language, it is necessary to investigate its expressiveness when capturing common control-flow structures [18]. In this paper, we scrutinized the Windows Workflow Foundation, an upcoming workflow framework currently gaining ground in the .NET community, through a pattern-based analysis. For our evaluation we used the Workflow Patterns [2], which is widely accepted in academia as well as in industry. In the past, other contemporary offerings in field of business process management (BPM) and workflow languages have been investigated using this framework.

Pattern	WF	WF*	BPEL	jBPM
Basic Control Flow				
1. Sequence	+	+	+	+
2. Parallel Split	+	+	+	+
3. Synchronization	+	+	+	+
4. Exclusive Choice	+	+	+	+
5. Simple Merge	+	+	+	+
Advanced Synchronization				
6. Multi Choice	+	+	+	-
7. Structured Synch. Merge	+	+	+	-
8. Multi Merge	-	-	-	+
9. Structured Discriminator	+/-	+/-	-	-
28. Blocking Discriminator	-	-	-	-
29. Canceling Discriminator	+	+	-	-
30. Structured Partial Join	+/-	+/-	-	-
31. Blocking Partial Join	-	-	-	-
32. Canceling Partial Join	+	+	-	-
33. Generalized AND-Join	-	-	-	+
37. Local Synchronizing Merge	+/-	+/-	+	-
38. General Synchronizing Merge	-	-	-	-
41. Thread Merge	-	-	+/-	+/-
42. Thread Split	-	-	+/-	+/-
Multiple Instances				
12. MI without Synchronization	+	+	+	+
13. MI w. a pr. Design-Time Kwlg	+	+	+	-
14. MI w. a pr. Run-Time Kwlg	+	+	-	-
15. MI wo. a pr. Run-Time Kwlg	-	-	-	-
34. Static Partial Join for MI	+/-	+/-	-	-
35. Canceling Partial Join for MI	+	+	-	-
36. Dynamic Partial Join for MI	-	-	-	-
State-Based				
16. Deferred Choice	+	+	+	+
17. Inter. Parallel Routing	-	+	+/-	-
18. Milestone	-	+	-	-
39. Critical Section	+	+	+	-
40. Interleaved Routing	+	+	+	-
Cancellation				
19. Cancel Activity	+	+	+	+
20. Cancel Case	+	+	+	-
25. Cancel Region	+	+	+/-	-
26. Cancel MI Activity	+	+	-	-
27. Complete MI Activity	-	-	-	-
Iteration				
10. Arbitrary Cycles	-	+	-	+
21. Structured Loop	+	+	+	-
22. Recursion	-	-	-	-
Termination				
11. Implicit Termination	+	-	+	+
43. Explicit Termination	-	+	-	-
Trigger				
23. Transient Trigger	+	+	-	+
24. Persistent Trigger	+	+	+	-

Table 1: Comparison matrix

Table 1 summarizes the analysis conducted in section 3. The results for Windows Workflow are shown in the first

two columns, where the first one reflects the support by sequential workflows only and the second one (WF*) allows for solutions using state machine workflows. In order to compare the results for Windows Workflow with other recent analysis, we included the findings for the Business Process Execution Language (BPEL) [3] and JBoss jBPM [4] in this table.

The basic control flow patterns are directly supported in all three offerings. With respect to the advanced branching and synchronization patterns, support varies between the different languages. In general, Windows Workflow lacks support for patterns dealing with multiple threads that are executed on the same workflow instance, since it has a strict single-threaded execution model. This applies also for state machine workflows, where a workflow instance can only reside in one single state at a given time. If Windows Workflow were to provide multi-threading for state machine workflow instances, support for most of the 43 patterns would be possible. Nevertheless, the state machine workflows can support most patterns in which a state-based notion is indispensable such as for example the *interleaved parallel routing* and *milestone* patterns. Due to it's support for multiple threads, jBPM stands out in the advanced branching category by supporting *multi-merge*, the *generalized AND-join* as well as partially supporting *thread split/merge*. Two constructs in Windows Workflow's *base activity library* are responsible for supporting a number of patterns. The *conditioned activity group* provides *multi-choice* as well as *discriminator* and *partial join* behavior. Multiple instances with synchronization are supported by the *replicator* activity. The synchronization capabilities of the *replicator* are also able to facilitate partial join behavior. BPEL as well as jBPM lack support for these patterns. The state-based patterns are completely supported by Windows Workflow. The state machine workflow style of modeling can be used where the sequential style is not expressive enough. Only limited support is provided by the other offerings in this category.

Except for the *complete multiple instance activity* pattern, cancellation behavior is fully supported by Windows Workflow. In terms of iteration patterns, none of the offerings provides direct support for *recursion*. Depending on their underlying flow model, BPEL supports *structured loops* and jBPM the *arbitrary cycles* pattern. Due to the two workflow styles in Windows Workflow, it supports both patterns. The same applies for the termination patterns, where state machines provide *explicit termination* and *implicit termination* is handled by sequential workflows. It is interesting to note, that the other offerings both provide *implicit termination* behavior (and lack explicit termination). Finally, both trigger patterns are supported in Windows Workflow, whereas the other offerings only support one.

It is evident that Windows Workflow benefits from its two workflow styles in regard to pattern support. From the beginning, Windows Workflow has been designed to support both, highly automated processes and workflows involving human interactions. The former is commonly realized using the sequential approach, whereas the latter can be supported by a workflow model that has a notion of state and is driven by external events. Although there are recent proposals for fostering human interaction in BPEL (BPEL4People [19] and WS-HumanTask [20]) its underlying flow model is not particularly suited for dealing with too much human interaction. BPEL seems to have inspired the design of Windows Workflow: Aside from the flow construct, the semantic of a BPEL process may be realized using sequential workflows. However, there is no concept like the flow construct in Windows Workflow [16]. Although both, state machines in Windows Workflow and the flow activity in BPEL allow for unstructured processes, their behavior differs in the following way: BPEL's *flow* activity is limited to acyclic graphs, but multiple branches can execute concurrently. State machines allow cyclic graphs with the limitation that there is no parallel branching across multiple states. The incompatibility between the flow models of these two approaches when designing process-aware information systems highlights an interesting fact: On the one hand, Microsoft is part of BPEL's standardization committee. On the other hand, Microsoft is simultaneously Windows Workflow, which has some fundamental differences in terms of control-flow logic. But the reasons for these differences are unclear and not motivated by their target audience.

In regard to the development of workflow-centric applications, Windows Workflow has the advantage that it can be incorporated in any type of .NET application, e.g., console applications, GUI-based applications, web applications, Web Services, etc. The same applies for jBPM in regard to Java. Both offerings are executable in a lightweight execution runtime that can be included in a regular code just like a common class library. In addition, Windows Workflow is shipped with the .NET framework and thus is available to every .NET developer. BPEL processes in contrast have to be executed in a BPEL engine running on a dedicated middleware platform (e.g., IBM WebSphere Process Server). This should not be considered as a disadvantage, but is a result of BPEL's focus on realizing business processes in terms of web service compositions.In a nutshell, Windows Workflow is a promising approach for making workflows first-class citizens. Its distinct benefit is that developers have the choice between sequential workflows and state machine workflows. Whatever workflow type is chosen as an implementation approach depends on the business scenario to be realized. Together, both workflow styles support a considerable set of the common workflow patterns.

Disclaimer. We, the authors and the associated institutions, assume no legal liability or responsibility for the accuracy and completeness of any product-specific information con-

tained in this paper. However, we have made all possible efforts to make sure that the results presented are, to the best of our knowledge, up-to-date and correct.

References

[1] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. Barros, "Workflow patterns," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, July 2003.

[2] N. Russell, A. H. M. ter Hofstede, W. M. P. van der Aalst, and N. Mulyar, "Workflow control-flow patterns: A revised view," BPM Center Report BPM-06-22, Tech. Rep., 2006, http://www.BPMcenter. org.

[3] P. Wohed, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede, "Analysis of web services composition languages: The case of BPEL4WS," in *22nd International Conference on Conceptual Modeling (ER 2003)*, ser. Lecture Notes in Computer Science, vol. 2813, 2003, pp. 200–215. [Online]. Available: http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2003/BPM-03-04.pdf

[4] P. Wohed, B. Andersson, A. H. M. ter Hofstede, N. Russell, and W. M. P. van der Aalst, "Patterns-based Evaluation of Open Source BPM Systems: The Cases of jBPM, OpenWFE, and Enhydra Shark," Eindhoven University of Technology, Tech. Rep., 2007.

[5] N. Russell, W. M. P. van der Aalst, A. H. M. ter Hofstede, and P. Wohed, "On the suitability of uml 2.0 activity diagrams for business process modelling," in *APCCM '06: Proceedings of the 3rd Asia-Pacific conference on Conceptual modelling*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2006, pp. 95–104.

[6] P. Wohed, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede, *Business Process Management*. Springer, 2006, ch. On the Suitability of BPMN for Business Process Modeling, pp. 161–176.

[7] W. M. P. van der Aalst, "Patterns and XPDL: A Critical Evaluation of the XML Process Definition Language," Queensland University of Technology, Tech. Rep. FIT-TR-2003-06, 2003.

[8] P. Wohed, E. Perjons, M. Dumas, and A. H. M. ter Hofstede, "Pattern Based Analysis of EAI Languages - The Case of the Business Modeling Language," in *ICEIS (3)*, 2003, pp. 174–184.

[9] E. Söderström, B. Andersson, P. Johannesson, E. Perjons, and B. Wangler, "Towards a framework for comparing process modelling languages," in *14th International Conference on Advanced Information Systems Engineering*. London, UK: Springer-Verlag, 2002, pp. 600–611.

[10] P. Green and M. Rosemann, "An Ontological Analysis of Integrated Process Modelling," in *11th International Conference on Advanced Information Systems Engineering*. London, UK: Springer-Verlag, 1999, pp. 225–240.

[11] B. Bukovics, *Pro WF: Windows Workflow in .NET 3.0*. Apress, 2007.

[12] P. Andrew, J. Conard, S. Woodgate, J. Flanders, G. Hatoun, I. Hilerio, P. Indurkar, D. Pilarinos, and J. Willis, *Presenting Windows Workflow Foundation*, 1st ed. Sams, 9 2005.

[13] *Web Services Business Process Execution Language*, OASIS, 2007, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html [acc.: 2009-03-19].

[14] JBoss, *jBPM Process Definition Language (jPDL)*, 2009, http://www.jboss.com/products/jbpm/, [acc.: 2009-03-19].

[15] Microsoft, *BizTalk Server 2009*, http://www.microsoft.com/biztalk/, [acc.: 2009-03-19].

[16] D. Green, "Biztalk server, windows workflow foundation, and bpm," Keynote at the Fourth International Conference on Business Process Management (BPM), Vienna, Austria, 2006.

[17] W. M. P. van der Aalst, J. Desel, and E. Kindler, "On the semantics of EPCs: A vicious circle," in *Proceedings des GI-Workshops und Arbeitskreistreffens Geschaeftsprozessmanagement mit Ereignisgesteuerten Prozessketten (EPK 2002)*, M. Nuettgens and F. J. Rump, Eds., 2002, pp. 71–79.

[18] B. Kiepuszewski, A. H. M. ter Hofstede, and W. M. P. van der Aalst, "Fundamentals of control flow in workflows," *Acta Informatica*, vol. 39, no. 3, pp. 143–209, March 2003.

[19] *WS-BPEL Extension for People (BPEL4People)*, Active Endpoints, Adobe, BEA, IBM, Oracle, SAP, 2007, version 1.0.

[20] *Web Services Human Task (WS-HumanTask)*, Active Endpoints, Adobe, BEA, IBM, Oracle, SAP, 2007, version 1.0.