# Workflow Patterns

implemented in

# BindFlow™

Jason Kleban

BindFlow LLC

## Abstract

We present an operating environment, called BindFlow™, and a companion program structure well suited for workflow programs which automate portions of real-world business processes. The flowchart logic of most business processes can be authored concisely using traditional programming techniques; however, the extended delays, overlapped execution, and hardware constraints in practical workflow scenarios severely complicate programs. We examine workflow programs as a mix of nondeterministic operations and deterministic transformations. Isolating a process's deterministic transformations from its nondeterministic operations allows us to record every input into the process. Instances can be safely unloaded from memory, destroying state, because the record of the inputs is all that is required to rebuild the logical state of a process instance on demand.

We introduce the BindFlow model and explore its ability to express the workflow patterns that have been recognized in the research by van der Aalst, Russell, ter Hofstede, et al., documented at http://www.workflowpatterns.com.

# Pattern Implementation Quick Reference

# Introduction

We present an operating environment, called BindFlow™, and a companion program structure well suited for workflow programs which automate portions of real-world business processes. Such programs are relatively light-weight computationally and may be considered to be "in-process" for weeks, months, or even years. We demonstrate a much improved articulacy of this operating environment over alternatives that employ state-persistence.

As an example of workflow, the business process for a company employee to officially request vacation time may instruct that the employee complete a provided form, obtain management-level written approval, and deliver the approved request form to the Payroll department. Finally, the Payroll department adjusts the records accordingly and notifies all interested parties. Some parts of this procedure such as the choice of the vacation days, the decision of the manager, and the adjustment of the payroll data may not be automatable, while other parts such as the routing of a completed form to a manager, the forwarding of the approved request to payroll, and the notification of the interested parties upon completion may be fully automatable.

The flowchart logic of this and most other business processes can be authored concisely using common imperative programming style; however, the extended delays, overlapped execution, and hardware constraints in practical workflow scenarios severely complicate programs. As one would with other programming assignments, one could construct complex structures in memory to store the state of business processes using only the data and control-flow features found in popular programming languages. Unfortunately for that approach, some business processes take weeks or months to enact from start to finish. Furthermore, the worth of automating a business process suggests the expectation of multiple simultaneous enactments, or *instances*[1], overlapping at various stages of completion. Even ignoring the gluttonous resource consumption, interruptions of in-memory processes by a hosting computer's reboot would result in an unacceptable loss of state of each *instance*.

*Instance* state is not limited to easily storable data such as digital form entries – it also includes branching and multi-iterative functionality, code as data, and related data scoping concerns. Contemporary computer architectures encourage in-memory-only state management, therefore custom persistence routines are the burden of the developer.

The commercial workflow software that we have been able to examine all use what we will label the "state-persistence" approach to workflow. That approach is to record the entirety of the workflow state to disk prior to unloading the instance and to logically restore it from the persisted state when needed. In our experience, the state-persistence model requires frustrating departures from standard software development practices and is under-expressive.

---

[1] To improve readability, we have italicized special-use terms and have underlined each near its first contextual description. See also the Terminology section of this document.

We first examine workflow programs as a mix of *nondeterministic* operations and *deterministic* transformations. We contrast this model against the state-persistence model. We then illustrate the usage of this model through examples written in C# for our implementation of BindFlow Server. Finally, we explore this model's ability to express the workflow patterns that have been catalogued in the research by van der Aalst, Russell, ter Hofstede, et al.[1].

## The Model

Workflow programs are a mix of *nondeterministic* operations and *deterministic* transformations. To explain, we return to our vacation request example for some definitions: The non-automatable human decision steps are not dictated from the perspective of the business procedure or the system that is partially automating it: The employee is making the choices of which days to request and of when to submit the request to the manager. The manager may have biases, but the final decision of approval or disapproval is not predictable. The Payroll system is not governed or tracked by this particular policy and so the results of the records adjustments are unknown. An operation is considered *nondeterministic* from the perspective of the system performing it if the results of the operation cannot be predicted from the data input for the operation. Two identical vacation request form submissions might be reviewed with different results because of some criteria that the manager uses to make the decision that is outside the scope of what the business procedure or the implementing system tracks.

The automatable steps in this example are *deterministic* by definition. That is, their behavior is consistent for any particular set of input data and is not influenced by circumstances that are not declared to be part of the input data. An employee submission of a vacation request always gets sent to the management team. A manager's approval always gets forwarded to Payroll and a rejection always gets returned to the employee. The payroll analyst's reported actions always get sent to the interested parties. The interested parties are always calculated as the manager, the employee, and an email distribution list of HR analysts. An operation is considered *deterministic* from the perspective of the system performing it if the results of the operation over data inputs are always the same. In the example, the calculations or transformations performed to discover an employee's management team depend only on the identity of the employee and the dataset representing the current organizational hierarchy. Given the same employee and org chart, the result of repeated program executions will be consistent. If the system asks for the management team for a different employee, or if the organizational structure changes in an applicable way, the answer will be different from the previous example, but it will be consistent for that dataset.

The results of *nondeterministic* operations, such as the manager's decision to approve an employee's vacation, provide the input to *deterministic* functions which transform that input into a request for a next *nondeterministic* operation. Isolating a process's *deterministic* transformations from its *nondeterministic* operations allows us to record every input to a log in non-volatile storage. Persisting input is much simpler than persisting the resulting state. *Instances* can be safely unloaded from memory, destroying state, because the record of the inputs is all that is required to rebuild the logical state of a process *instance* when needed. Each time the *instance* is loaded in memory and finally unloaded is called a *Session*.

Borrowing a term from pure functional programming, *nondeterministic* operations are called *IO*, for input/output as they observe or manipulate information external to the system. Examples of *IO* are: sending an email, reading/writing the contents of a file, getting the current time, generating a random number, and asking for a user's answer to a question. Note that even though random number generation is usually done through *deterministic* pseudorandom number generating algorithms, these algorithms are external to the business process logic and are intended to provide *nondeterministic*-like behavior and so we may consider them to be *nondeterministic*.

Readers familiar with functional programming may recognize what we describe as a feature-rich version of the Replay monad, a construct in which pure functions bind the results of *IO* to a supplied pure function continuation. We have in a way adapted this concept for multi-threaded workflows and multi-user environments[2].

A log and replay approach has been applied in lower-level systems to virtualization, unit-test generation, and debugging[3].

In BindFlow, *deterministic* transformations are program code, quite naturally chained into control-flow logic called *sequences*. While *sequences* cannot perform *IO* themselves due to the *determinism* requirement, a *sequence* may emit requests for *IO* to be performed. Note that constructing and emitting a message describing a request for *IO* is not the same as performing the *IO*. We call these request descriptions *favors*. The *instance* requests something of its host through a *favor* and the host responds with the results of the operation.

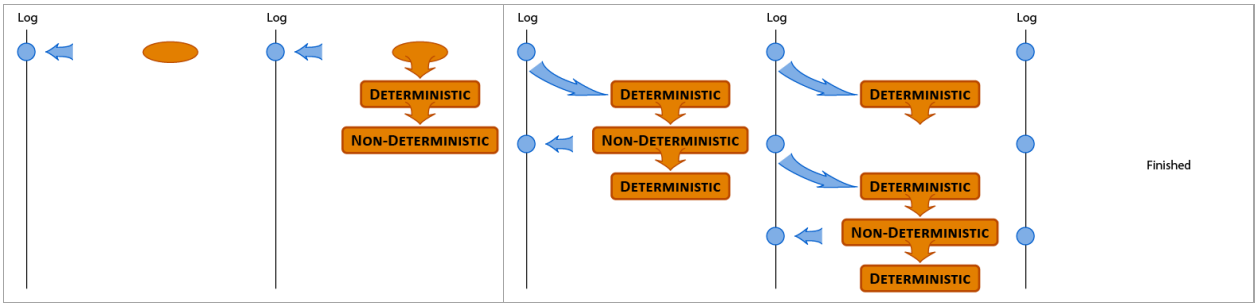
Besides generic *IO favors*, which request arbitrary real-world interaction, there are built-in system *favors*. Like fork() and other API calls used in traditional multitasking programming, these predefined *favors* are used to manipulate the host or the state of a running *instance* in ways that the *sequences* are unable to express or directly perform themselves. These include operations such as assigning an external *task* to a user or external system, *subscribing* to (or registering interest in) occurrences of an external type of event, jumping to another *sequence*, or forking to another *sequence* or process *instance* for parallel processing.

**Figure 1**

We may consider the *instance* isolation as a bubble membrane. Inside the bubble are the *deterministic* transformations and private data structures. Outside the bubble is the real world – the people, the external events, and external hardware and software components. Only the hosting environment can reach into the bubble and provide new input. Only the hosting environment can directly receive *favor* messages originating from inside the bubble. These *favor* messages describe either predefined actions OR arbitrary functions to be run by the host.

*Sequences* take the input from the last *favor* (or the input data of the *sequence*) and either terminates or transforms that data into a new *favor* to be requested of the system. Within a *sequence*, standard computation and control-flow mechanisms are used to implement the transformations. Figure 1 illustrates the alternating nature of *deterministic* transformations and *nondeterministic* operations.

By definition, a *process* will have a default state, possibly empty. An *instance* of a *process* is initiated when a *nondeterministic* external event, the request for a new *instance* along with its custom data payload (externally supplied, such as submitted form data), prompts the BindFlow system to allocate the new *instance* and submit the request's payload to the *process*'s known entry point, which we call the *Start sequence*. *Start* uses the default state and the supplied payload to determine what external action must be taken next, which then it communicates back to the BindFlow system by responding to the host's call into its entry point with a *favor*. The result of a *favor*, having been procured by the host, is passed back into the remainder of an unfinished *sequence* as additional input for the next transformation. This concept extends to internally multi-threaded *processes* as explained in the section explaining the implementation of the **Parallel Split** pattern.




| | | | | |
|---|---|---|---|---|
| An instance's initial data is written to the log. The input is deterministically transformed into a Favor and emitted to the host by the instance. | The host engine performs the requested operation and logs the result before supplying it to the next deterministic transformation. | The engine unloads the instance's state to conserve memory and CPU when receiving Favors, such as AssignTask, which may block an instance for extended durations. | Unloaded instances are restored by rapidly rerunning the same deterministic transformations using the original data read from the log. | This instance terminates once a final deterministic transformation returns without requesting a Favor. |

**Figure 2 – Deterministic Transformations and Nondeterministic operations**

By recording each input, BindFlow can track the interaction between *nondeterministic* operations and *deterministic* transformations. This guarantees that the logical state of a partially executed *instance* can be rebuilt from the *process*'s default state. When needed, BindFlow restores the state of the *instance* by feeding the recorded inputs back to the *sequences* in the original order, responding to each emitted *favor* with the stored result rather than a newly generated one. Once the log of results has been depleted, the *instance* will be in the same logical state as it was before being unloaded from memory.

Failures during *IO* can be handled by any appropriate corrective action whereas interrupted *sequence* transformations have no effect on the system or data due to their required functional purity and can be retried.

## Comparison to State-Persistence

The commercially available workflow software that we have been able to examine all use what we will label the "state-persistence" approach. That approach is to record the entirety of the workflow state to non-volatile storage prior to unloading the instance and to logically restore it from the persisted state when needed. In our experience, the state-persistence model requires frustrating departures from standard software development practices and is under-expressive. We will now summarize the

succession of compromises that the state-persistence approach dictates which lead, rather unavoidably, to those drawbacks:

A conceptually naïve implementation of state-persistence performs a core dump before unloading an instance and with significant effort, restriction, and storage cost, can resurrect the instance at a later time[4].

More practical state-persistence workflow engines avoid the costs of core dumps by requiring that a workflow be segmented into a collection of independent programs (or perhaps a library of equivalently-independent functions) such that idle-points in the process occur between segments. And just as a flowchart's steps are granulated to support looping and other conditional redirection, so too is a workflow's logic divided into its segments. Often, each one of the many segments is very short, existing merely to update a persisted variable, evaluate a condition, or perform a basic integration operation. Since the segments cannot communicate directly, each segment must load the relevant state, perform its small part of the total process, and update the relevant state accordingly.

Each segment is executed by the workflow engine if the conditional paths leading to it are met. The condition and its inputs, or at least the condition's outcome, must be accessible to the workflow engine and so constraints are imposed on the manner in which a state is persisted to conform to the engine's supported interface. Native control-flow mechanisms are unusable for the purposes of routing among segments since the engine performs the routing based on state-data; difficult-to-follow jump instructions are instead encoded into the state for the enactment by the engine. An otherwise simple program definition will become overrun by an inelegant mix of structural and data-based control-flow and persistence mechanisms.

Rather than communicating through in-memory data structures, the state-persistence model requires segments to communicate only through the persisted state – even for intermediate data that has no external use. It may seem to work well for simple workflows with a single, global data-scope but it severely complicates the state-management mechanisms for the workflow engines which are expected to support more complex data scoping patterns such as the **Multiple Instance Data** pattern discussed in the Workflow Patterns section. More detailed descriptions of workflow engine designs are made available by Georgakopoulos[5].

In contrast to state-persistence engines, BindFlow does not attempt to persist the state of a workflow, opting instead to merely record any new data as it enters the *instance* along its execution. The *instance* is free to create and maintain any arbitrary, strongly typed, and granularly scoped data structures in its memory-space. If an interruption occurs, the recorded inputs are replayed into the *deterministic* code and the logical state is restored. Language-native control-flow mechanisms are sufficient for most routing situations and the workflow engine imposes no restrictions on the state data that is accumulated during execution.

## Workflow Patterns

Van der Aalst et al.[1] have studied, named, and sorted more than 120 universal workflow requirements into the four pattern categories of Control-flow, Data, Resource, and Exception Handling. Control-flow is

perhaps the most familiar category, dealing with the rules that govern the progression of workflow state. Control-flow is the distinguished feature of graphical flow charts commonly represented as the lines and diamonds among the rectangular nodes. In an arbitrary process with multiple participants and some data payload routed among them, the control-flow patterns would describe the order or stage in which the participants are involved. It may be that each participant gets the payload in turn, sequentially, or they may receive the payload simultaneously, in parallel. Maybe they all do the same work, or maybe they each have different assignments. Often, multiple patterns are at work within a single process definition.

Data patterns describe the way data is scoped or shared among various parts of the workflow system, including getting data in or out of the *instance* state. Maybe all participants share a single copy of the data related to an assignment or maybe they each get an isolated copy.

Resource patterns include the selection of the participants, people or machines, and the access rights of the participants to *instance* status and data. The work might be assigned to an individual or an entire group. Perhaps the work assigned to a group should be locked to the first person to open the item. Perhaps future steps in the workflow should be directly assigned to the individuals who are now familiar with the case. Depending on the pattern, the choice of the assignees may be from a *deterministic* source (hard-coded) or a *nondeterministic* source (external).

The Exception Handling category discusses a system's ability to handle exceptional cases, such as invalid input, without having to explicitly code for each possibility.

The patterns, generally, are a taxonomy for the fundamental design challenges facing workflow developers. The aim of cataloging these patterns is that regardless of the industry, participants, or data, every workflow problem can be understood as an assembly of the patterns and that the workflow solutions can be realized by a composition of the implementation techniques. The patterns might describe routing for an online retailer's product fulfillment, a corporate budget approval, or a nuclear facility's safety regulations - but as illustrated by van der Aalst et al.[1], our introductory descriptions have unspecified nuance that must be understood before the actual process can be accurately automated. We encourage the reader to review the detailed technical descriptions and the interactive animations of these patterns at workflowpatterns.com.

## An Implementation

BindFlow™ is an implementation of this described model written in C# and .Net 3.5. To be clear, BindFlow is not based on Microsoft's Workflow Foundations (WF) which is a set of types useful for implementing a state-persisting workflow engine. This paper is based on the 2012 version of the BindFlow software.

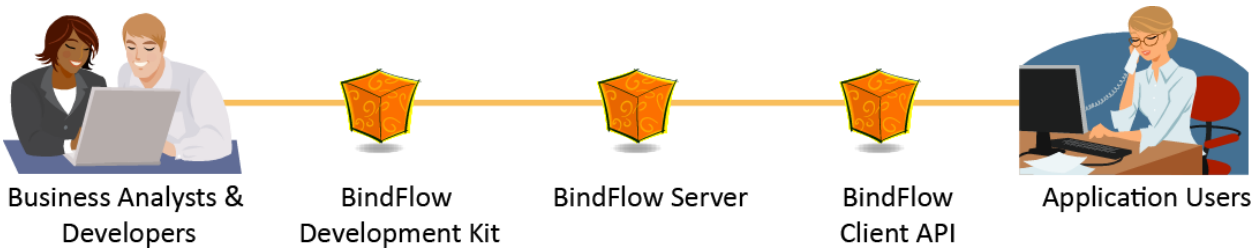



**Figure 3**

BindFlow has three main components: BindFlow Server™, the BindFlow Development Kit™, and the BindFlow Client API™. The BindFlow Server is an installable window service. It is responsible for providing the runtime environment for BindFlow *process instances* and for maintaining lists of assignments pending completion.

The BindFlow Development Kit is a code library defining the base types for BindFlow *processes* and custom *IO* and contains pre-built *IO* for common external systems integration. Business analysts and developers work together to extend these base classes to automate workflow logic as BindFlow *processes*. *Processes* are defined as classes extending the `ProcessBase` base class. .Net assemblies containing one or more process definitions are called *Process Sets*.

The BindFlow Client API is a code library to facilitate interaction with the BindFlow Server. User interface applications surfacing the forms or other data use this API to communicate with BindFlow server to provide such common application needs as displaying and completing assignments.

In this document, we focus on the use of the BindFlow Development Kit and its dependence on the BindFlow Server. We will ignore aspects of the BindFlow system that do not pertain to the implementation of the workflow patterns, such as safety mechanisms and security.

```
0 public class TrivialProcess : ProcessBase
1 {
2     public override IEnumerable<IFavor> Start(object data)
3     {
4         yield break;
5     }
6 }
```

**Code Listing 1 – The trivial process**

Referring now to Code Listing 1, this `TrivialProcess` is the simplest valid extension of `ProcessBase`. The *process* performs no computation and terminates immediately. The `data` parameter of the `Start` *sequence* would be ignored, if any were to be provided. This BindFlow implementation makes heavy use of the iterator concept available in many languages including C# and Java. All BindFlow *sequences* are enumerable code-blocks which conform to the `Sequence(object data):IEnumerable<IFavor>` signature, or its overloads. C# requires that each iterator block have at least one `yield` statement, either `yield break`, which terminates the iteration of the *sequence*, or `yield return favor`, where `favor`

is an object of a type that implements `IFavor`. Each `yield return` statement will be simply referred to as a yield or an emission.

By taking advantage of the syntactic sugaring provided by the C# compiler for iterator blocks, BindFlow can simulate interruption of function calls while in fact, each section of an enumerator is rearranged by the compiler as a stateful object conforming to the iterator interface. The reader is invited to learn more about this powerful language feature, as it is implemented in C#, on Microsoft's MSDN documentation[6] or elsewhere.

BindFlow is a cooperative multi-tasking system and as such, it is the responsibility of the *process* to return control to the host (BindFlow Server) either by terminating itself or by emitting a *favor*. Termination can be explicit with `yield break` or can be implicit by reaching the end of the code branch. As *sequences* must remain *deterministic*, *sequence* code is not allowed to directly interact with anything outside of the scope of the related *instance* except by passing *favor* messages back to the calling host. This restriction includes any sort of delay, which would require interaction with the system's timer and callback routines. Rather, a correctly built *sequence* performs any arbitrary *deterministic* transformation on the latest input as quickly as possible and does not perform any synchronization operations. When *nondeterministic* operations such as I/O or delays are required, the intention is emitted to the host in the form of a *favor*.

| | |
|---|---|
| `Jump` | An unconditional jump. Moves the current *sequence* iteration to the beginning of a *sequence*. The jumping *sequence* is not resumed at the completion of the `Jump`-targeted *sequence*. |
| `Call` | Suspends the current *sequence* iteration and executes a new *sequence*. Once finished, the calling iteration continues. |
| `Sequence` | A convenience. Wraps a single *favor* or an array of *favors* in a new *sequence*, to be emitted serially if iterated (with `Call`, `Jump`, or `AsyncCall`). |
| `AssignTask` | Blocks a *sequence*, pending a *task* result |
| `Subscribe` | Registers an interest in external events with the host. Asks the host to execute a particular *sequence* upon each occurrence of such event. |
| `Unsubscribe` | Revokes a *subscription*. Asks the host to no longer notify for the particular *subscription*. |
| `AsyncCall` | For multi-threading/parallel execution of *sequences*. The host will execute a new iteration of the given *sequence* until the target *sequence* iteration blocks and then will immediately resume execution of the calling *sequence*. Target *sequences*, the calling *sequence*, and all other unrelated *sequences* can be further executed in an order not specified at design-time. |
| `Wait` | Waits on a single *sequence* or sub-*instance* to complete. |
| `Cancel` | Cancels a sub-*instance* or an entire *branch* of execution of a root *sequence*. That is, the canceled *sequence* and all of its active sub-*branches* and *instances* are also canceled. |
| `Spawn` | Creates and executes a sub-*instance*, an *instance* of a named *process*. |
| `AbandonSession` | Abandons the current *session* without committing its progress. A convenience for special applications. Any sub-*instances* are not abandoned. |
| `ForceTerminate` | Terminates a *process instance* immediately. Sub-*instances* are also terminated. |
| `Return` | Immediately ends a *sequence* and returns some value (as a replacement for `yield break` which syntactically cannot return a value to the calling *sequence*) |

**Table 1 – System Favors**

*Favors* come in two varieties, system requests and *IO*. There are 13 primitive system *favors* that perform interaction with the host's managed data and/or perform *instance* state manipulations that must be honored or tracked by the host. They are as listed in Table 1.

| | |
|---|---|
| `Delay` | Blocks the current *sequence* for no less than a minimum duration |
| `AsyncDelay` | Executes a *sequence* after a minimum duration has elapsed |
| `WaitAll` | Waits on the completion of all of the listed *sequences* and/or sub-*instances*. |
| `WaitOnPrimary` | Terminates (prematurely abandons) the secondary *sequence* iteration upon the completion of the primary *sequence* iteration |

**Table 2 – Composite System Favors**

In addition, there are several built-in *favors* that are included for convenience, but which are merely common, user-definable composites of the 13 primitive system *favors*. They are as listed in Table 2. System *favors* are created within *processes* by calling the `ProcessBase`-defined protected methods named in Table 1 and Table 2. These methods each return an `IFavor` which must be emitted to the host. That is, the protected methods such as `AssignTask` which does not perform the assignment of the *task* directly create the *favor* (message object) that should be used to communicate the request for the related operation back to the host. System request *favors* are sealed and cannot be extended.

```
0 public class IOMyOperation : IOBase
1 {
2     protected override object Perform()
3     {
4         return null;
5     }
6 }
```

**Code Listing 2 – The trivial IO**

The second type of favor, *IO*, is based on the `IOBase` abstract base class. The trivial *IO* is illustrated in Code Listing 2. This *IO* takes no input, does no computation and returns `null`. Similar to the *deterministic* restrictions of *sequences*, correctly written *IO* should do whatever single-threaded computation is necessary and return immediately without explicit delay. *IO* is only for synchronous, immediate system integration. Any desired delays should be implemented by creating `Delay` *favors* and emitting them from within an executing *sequence*. Contrary to system *favors*, which are created by the factory methods, *IO favor* messages may be instantiated directly using the `new` keyword; however for consistency, we recommend providing static factory methods within *IO* definitions to mimic the calling convention of the system *favors*. We do not follow this advice here.

```
00 public class MyProcess : ProcessBase
01 {
02     public override IEnumerable<IFavor> Start(object data)
03     {
04         var getNameById = new IOGetNameByID(4);
05
06         yield return getNameById;
```

```
07
08          // use getNameById.Name …
09      }
10  }
11
12  class IOGetNameByID : IOBase
13  {
14      int id;
15
16      public string Name { get { return RawResult as string; } }
17
18      public IOGetNameByID(int id)
19      {
20          this.id = id;
21      }
22
23      protected override object Perform()
24      {
25          return SQL.GetNameByID(id);
26      }
27  }
```

**Code Listing 3 – Implementation and use of a less trivial custom IO.**

Code Listing 3 demonstrates a completed *IO*. The constructor sets the parameters to be used by `Perform` when called by the engine. `Perform` returns a serializable string object. The engine records the value and populates `IOBase.RawResult`, which is exposed by the custom `Name` property of the `getNameById` *favor* on Line 06.

```
00  public class Example : ProcessBase
01  {
02      public override IEnumerable<IFavor> Start(object data)
03      {
04          var firstTask = AssignTask(
05                              "Do you like peanut butter?",
06                              "",
07                              this.Initiator,
08                              null);
09
10          yield return firstTask;
11
12          if ((bool)firstTask.Result)
13          {
14              var secondTask = AssignTask(
15                                  "I have some.  Do you want a sandwich?",
16                                  "",
17                                  this.Initiator,
18                                  null);
19
20              yield return secondTask;
21          }
22      }
23  }
```

**Code Listing 4 – Example Process**

Code Listing 4 is an example of a simple workflow. In it, the `data` parameter is ignored – there is no initial payload required to start an *instance* of this *process*.

Starting on line 04, `firstTask` is assigned as a *favor* returned from the `AssignTask` factory method. Line 05 is the readable summary of this *task* which would be presented to the assignee as one assignment in a list of all assignments due by the assignee, called a ***task-list***. Line 06 would represent to URI of the resource that the assignee should use to complete the request, such as the URL of a web application. In this scenario, we are ignoring these practical concerns. Line 07 is the assignee, in this case, the initiator of the *instance*. Line 08 is the data associated with this request – in this case `null` since we have enough information for this example being the assignment's summary, "Do you like peanut butter?"

On line 10, we emit the `firstTask` *favor*. This has the effect of blocking the *sequence* (and since there are no other active *sequence* iterations, the entire *instance*) until the assignee completes the *task*. The *task* will appear in the *task-list* and the assignee responds through an end-user-application-validated choice of either `true` or `false`.

By line 12, the result of the `firstTask` *favor* is available and is cast as the expected `bool` type for evaluation. If the assignee responds with a value of `true` then a second assignment similar to the first is constructed and emitted, otherwise the *sequence* and, ultimately, the *instance* is terminated.

Though it is not important to understand to be productive in this model, you may note that at each *favor* emission, the host may decide to completely release the state of the *instance* from memory. If the host implementation should so choose to unload the *instance* from memory during line 20, for example, the host will resume the *instance* by recreating the *instance* state from scratch, starting with the default (empty) state and executing `Start` with the same input originally provided (probably `null`, in this example), running the `Start` *sequence* until line 10, receiving the `firstTask` *favor*, dequeueing the original result of that first assignment (which we must here assume to be `true`, since we reached line 20 at all), and immediately resuming at line 12. The *sequence* immediately continues creating `secondTask` just as it did in the first *session*. After the emission on line 20, the original state has been restored. At this point, if we assume that the completion of the assignment by the assignee was what prompted the host to restore the state of the *instance*, we are now ready to immediately inject the result of `secondTask` back into the remainder of the *sequence*. The remainder (or *continuation*) happens to be empty in this short example and the `secondTask's` result is not used, but very well could be as was the `firstTask's` result.

## Terminology

Note that a BindFlow *instance* is the runtime version of a BindFlow *process*. A *process* is the definition of all possible paths that an *instance* might take and contains no state (though it may describe a default state). *Instances* are the runtime embodiment of a specific path through a *process*. In some discussions, the terms are roughly interchangeable. Many independent *instances* may follow the rules of a single *process*, each with their own state and isolated from each other.

Note that a BindFlow ***branch*** is a runtime version of a BindFlow *sequence*. As a *process* is to an *instance*, a *sequence* is to a *branch*. A *sequence* is the definition of all possible paths that a *branch* might take and

contains no state. *Branches* are iterations over their *sequence*, the runtime embodiment of a specific path through a *sequence*. In some discussions, these terms are also roughly interchangeable. Many independent *branches* in a single *instance* may follow the rules of a single *sequence*, each with their own state and isolated from each other. Data may be shared in a variety of ways depending on scoping – for example, data fields scoped at the *instance* level may be shared by all *branches*.

Note that while methods such as `AssignTask()` and `AsyncCall()` are not exactly *favors*, but are instead factory methods that return *favors* of type `IAssignTask` and `IAsyncCall` respectively, it is productive to loosely refer to the use of such methods as *favors* in discussion for the sake of overall clarity - i.e. "The `AssignTask` favor".

Note that BindFlow terminology and the terminology chosen by van der Aalst et al.[1] sometimes collide. Care has been taken to translate the underlying concepts put forth by van der Aalst et al. into BindFlow's model. One example of a terminology difference is "Task". Van der Aalst et al. consider a "task" as a discrete step, a logical section of a workflow. In BindFlow, a "*task*" is an assignment to be completed by a user or external system. While both concepts are present in both models, the mapping of named concept to named concept is indistinct. Other terms to read carefully are "instance" (of a task), and "branch".

### BindFlow Stack & Threading Model

In BindFlow, the standard CIL call stack is supported (as used in any C# program) for normal method calls. Then there are the BindFlow stack and wait-list managed by the engine for multithreading support. It may be useful to understand how the stack and wait-list operate and how they are honored by various system *favors*. We refer to BindFlow engine's threads unless otherwise noted.

BindFlow's threading model is cooperative. While an *instance* may have many threads in play, only one is actively processed at any one time. Thread execution always continues uninterrupted until it emits a *favor*. Non-blocking *favors* that do not affect the *instance's* callstack are continued again after the *favor* is satisfied. Newly initiated *branches* (`Call`, `AsyncCall`) or newly spawned *instances* (`Spawn`) are placed on the top of the stack, pausing the execution of the initiating *branch*. `AsyncCall` runs a newly initiated *branch* until it first blocks (or terminates), then returns the control to the calling *branch*. `Spawn` behaves the similarly for spawned *instances* – *instances* are run from their `Start` *sequence* until the *instance* terminates or all of the *branches* are blocked, then the spawning *instance* is resumed at the spawning *branch*. `Call` runs a called *sequence* until *branch* termination and moves the calling *branch* from the stack to the wait-list. `Wait` similarly moves the emitting *branch* from the stack to the wait-list. Normal *branch* termination or `Cancel`, which terminates a *branch*, removes it from the wait-list, and pushes any waiting *branches* back on the stack.

Wait-list items are returned to the stack LIFO so that the most recently waited item will be at the top of the stack and resumed first.

`AssignTask` may be considered a `Call` to an external person or system – where the emitting *branch* is blocked until the assigned *task* is completed. `AssignTask` moves the calling *branch* to the wait-list.

While an `AssignTask` cannot be canceled directly, its containing *branch* may be canceled which has the effect of revoking the *task*.

`Jump` terminates the current *branch* and replaces it with a *branch* of the target *sequence*.

`Return` provides a way to terminate the emitting *branch* and populate the `Result` property of the `AsyncCall` *favor* that initiated it. The `ProcessBase.Result` property may be set to provide a value to the `Spawn` *favor* that spawned it. Setting `ProcessBase.Result` does not immediately terminate the *instance* as `Return` does for a *branch*.

*IO favors* may not block, and so do not affect the stack or wait-lists.

*Instances* in the same spawn tree – that is, *instances* sharing a common ancestral *instance* – may not be executed simultaneously since one might intend to terminate the other.

## The BindFlow Framework

There are two halves of the BindFlow APIs, as depicted in Figure 3: the BindFlow Client API and the BindFlow Development Kit. The Client API is used to communicate and configure the BindFlow Server, start new *instances*, retrieve work items, complete *tasks*, and notify on *subscriptions*. The BindFlow Development Kit contains the base classes for custom *processes* to be executed in BindFlow Server.

An *instance* should be considered to be within a protective bubble membrane isolated from the outside world. Outside of the bubble, the universe, anything can happen. Inside the bubble, anything can be calculated about the outside universe, but to support the replay, input must be strictly controlled through the portals in the bubble managed by the BindFlow Server. These portals are the *instance* creation, and completing of *IO*, including *tasks* and *subscriptions*.

```
ProcessBase:
Public Members -
  Start(object data) : Sequence
  Summarize() : string (must implement)

Protected Members -
  Initiated : DateTimeOffset
  Initiator : Account
  InstanceId : long
  Milestones : IMilestoneCollection
  Result : object
  SessionResult : object

  AssignTask(priority : TaskPriority, summary : string, assignment : string,
             assignee : Account, data : object) : IAssignTask (overloaded)
  AsyncCall(target : Sequence) : IAsyncCall (overloaded)
  Call(target : Sequence) : ICall (overloaded)
  Cancel(waitable : IWaitable) : ICancel
  Jump(target : Sequence) : IJump (overloaded)
  Return(result : object) : IReturn
  Sequence(favors : IFavor, …) : Sequence
  Spawn(processName : string, data : object, largeData : object) : ISpawn (overloaded)
  Subscribe(tag : string, data : object, callback : Sequence<object>) : ISubscribe
  Unsubscribe(subscription : ISubscribe) : IUnsubscribe
  Wait(waitable : IWaitable, …) : IWait
  AbandonSession() : IAbandonSession
  ForceTerminate() : IForceTerminate

IOBase:
Public Members -
  IsUsed : bool
  Succeeded : bool
  TreatErrorsAsData : bool
  UnhandledException : Exception

Protected Members -
  CurrentInstanceInfo : InstanceInfo
  RawResult : object

  Perform() : object (must implement)

  GetInstanceLargeData(preserve : bool) : object
  ClearInstanceLargeData() : void
  GetCurrentNotificationLargeData(preserve : bool) : object
  ClearCurrentNotificationLargeData() : void
  GetNotificationLargeData(subscriptionId : long, occurrenceId : long,
                           preserve : bool) : object
  ClearNotificationLargeData(subscriptionId : long, occurrenceId : long) : void
  GetTaskLargeData(taskId : long, preserve : bool) : object
  ClearTaskLargeData(taskId : long) : void
  GetConfigValue_(key : string) : string
  GetInstanceInfo(instanceId : long) : InstanceInfo

  CompleteTask(taskHandle : string, data : byte[],
               largeData : byte[]) : InternalSessionResponse
  NotifySubscriber(subscriptionHandle : string, data : byte[],
                   largeData : byte[]) : InternalSessionResponse
```

**Figure 4 BDK abbreviated framework types**

The BindFlow Development Kit's two major classes are `BindFlow.BDK.ProcessBase` and `BindFlow.BDK.IOBase`. Abbreviated members are listed in Figure 4. Subclasses of `ProcessBase` describe all *deterministic* transformations. Subclasses of `IOBase` describe custom *nondeterministic* operations, or integration with the real world. The Development Kit also contains a Visual Studio-integrated BindFlow implementation, called the "Workbench", that facilitates testing and break-point debugging.

```
ServerProxy:
  NewInstance(processName : string, startData : object,
              largeData : object, wait : bool) : SessionResponse

  GetTask(taskHandle : string) : GetTaskResult
  SaveTaskData(taskHandle : string, data : object, largeData : object) : void
  CompleteTask(taskHandle : string, data : object, largeData : object, wait : bool,
               faultBehavior : OnFaultBehavior) : SessionResponse

  GetSubscription(subscriptionHandle : string) : GetSubscriptionResult
  NotifySubscriber(subscriptionHandle : string, data : object, largeData : object,
                   wait : bool, faultBehavior : OnFaultBehavior) : SessionResponse
```

**Figure 5 Client API ServerProxy abbreviated listing**

The BindFlow Client API has one main class, `BindFlow.Client.ServerProxy`. Its abbreviated member list is in Figure 5.

These code samples illustrate the patterns minimally, but none of the implementations cheat such that an implementation could not be extended to practical scenarios. We do not use the full features of the *favors* when not required. An overload of the `AssignTask` *favor* factory method is listed in Figure 4 where `priority` is a `low`, `normal`, `high` enumeration; `summary` is a friendly short string describing the *task*; `assignment` is a URL to a web application or some other redirect where the *task* can be completed; `assignee` is an `Account` (A reference to a user); `data` is any serializable data or `null`, such as form data, to be available for completing the *task*; and `visibleInTaskList` is a Boolean value indicating a *task's* visibility in the built-in BindFlow Viewer. We test our implementations in the Workbench to avoid having to write a custom User Interface for each example, and so options such as `assignment` are omitted. For these examples, we often populate the data parameter with the default value of the expected result type. There is no requirement that this data value and the `IAssignTask.Result` are related in any way, however, providing such a template value facilitates testing within the Workbench.

## An Optimization for Large Data Input

Another valuable feature not demonstrated in the examples, but one which is relevant to practical applications built on BindFlow is "Large Data". Very large data sets, such as document attachments, to be submitted as part of an *instance's* initial payload, a *task*'s completion data, or a *subscription* notification are often immediately moved to some document repository as a process step. In these cases, loading this very large dataset for all subsequent *sessions*, just to satisfy the replay requirements is inefficient. Other workflow products face a similar obstacle in keeping state small and may work around it by advising the

application developer to upload the document to the target repository and only submit a reference to the document for processing. This puts unnecessary burden on the developer to host such a repository and have two methods for interacting with the repository (one prior to submission and one in the workflow), even if the need for the data is only short-term. To solve this problem in BindFlow, we have a mechanism called "Large Data" which can be used to pass any serializable data to a new *instance*, *task* completion, or *subscription* notification but which is only directly accessible from *IO*. When submitting a large document as an attachment for processing, the application developer can include it as Large Data. Any custom *IO* can access it, process it, and preferably discard it from the Large Data store. In some cases, the document is merely to be moved to some document repository and, optionally, the original submission can be cleared from the built-in temporary store. In other cases, only a subset of the document's data is necessary for process decision points and an *IO* can be used to arbitrarily summarize the contents of the Large Data down to those key decision factors. The entire Large Data dataset can be explicitly returned to the *Instance's* internal state from an *IO* in cases if deferred conditional access to the full Large Data is required.

# Pattern Support

Van der Aalst et al.[1] categorize their identified workflow patterns into four major groups: Control-Flow, Data, Resource, and Exception Handling. All quoted definitions are referenced for convenience from van der Aalst et al. though referenced pattern-variation definitions are not quoted here. Redundant lines of the code listings including namespace references are omitted after Code Listing 5 to save space.

Note that the example implementations do not necessarily represent best practices for code, but are crafted as short illustrations of the essence of the solution for each pattern to avoid the need to cover too many topics in a single pattern discussion.

Note also that pattern implementation complexity will vary by model. The order of presentation of the patterns here is copied from van der Aalst et al. for consistency rather than attempting to reorder them by complexity relative to the BindFlow model.

Not every technique or mechanism is discussed for every pattern, but we attempt to cover all techniques as necessary.

## Control-Flow

Control-flow patterns deal with the mechanics of decision points in a workflow. Van der Aalst et al.[1] organize control-flow patterns into groups of conceptually similar patterns. All 43 of these patterns are supported in BindFlow.

### Basic

#### *Sequence*

> *A task in a process in enabled after the completion of a preceding task in the same process.*

```
00 using System;
```

```
01 using System.Collections.Generic;
02 using BindFlow.BDK;
03 using System.Collections;
04 using System.Linq;
05
06 namespace WorkflowPatterns.ControlFlow
07 {
08     public class Sequence : ProcessBase
09     {
10         public override IEnumerable<IFavor> Start(object data)
11         {
12             var task = AssignTask("Advance", Initiator);
13             // First Task
14             yield return task;
15
16             // Second Task
17             yield return AssignTask("Advance Again", Initiator);
18         }
19     }
20 }
```

**Code Listing 5 – Sequence Implementation Example**

The **Sequence** pattern is implemented as an unconditional iteration over an ordered listing of instructions. These instructions can be either purely *instance*-state manipulations or can be *deterministic* operations interleaved with *IO*. In Code Listing 5, the `Start` *sequence* consists of two serial *tasks*. The first *task* is assigned and the *instance* blocks until the *task* is completed by a user. Then the second *task* is assigned and the *instance* blocks until the *task* is completed by a user. Then the *process* implicitly terminates. This *process* makes no considerations for the data passed in to `Start` nor data submitted when completing either *task*.

### *Parallel Split*

> *The divergence of a branch into two or more parallel branches each of which execute concurrently.*

```
00 public class ParallelSplit : ProcessBase
01 {
02     public override IEnumerable<IFavor> Start(object data)
03     {
04         // Split
05         yield return AsyncCall(Other);
06
07         // First Task
08         yield return AssignTask("Advance Main branch", Initiator);
09     }
10
11     IEnumerable<IFavor> Other()
12     {
13         // Second Task
14         yield return AssignTask("Advance Other branch", Initiator);
15     }
16 }
```

**Code Listing 6 – Parallel Split Implementation Example**

The **Parallel Split** pattern is implemented using the `AsyncCall` *favor* to initiate a new *sequence*. In Code Listing 6, Line 05 emits an `AsyncCall` requesting the execution of a new *branch* of the `Other` *sequence*. The BindFlow engine pushes a new `Other` *branch* onto the stack which immediately blocks with the "Advance Other Branch" *task*. The *branch* is moved to the wait-list and control is returned to `Start`, line 08, which assigns a second *task*. The *session* concludes as all of its *branches* are blocked and the server adds both *tasks* to the *instance* initiator's work list. Either *task* can be completed first and the other second. In this example, the completion of either *task* leads to an immediate implicit termination of the respective *branch*. Once both *branches* have completed, there is no work left to do and the *instance* is complete.

A note about multithreading: A *process* can define one or more *sequences*. These *sequences* are sections of code (implemented as iterator blocks which yield *favors* between *deterministic* steps. Parallelism means not that two steps are being actively executed simultaneously, but that the execution of the steps is only *partially ordered*. In `Process X` of Figure 6, step `A1` precedes `A2` which precedes `A3`; `B1` precedes `B2` which precedes `B3`; and `C1` precedes `C2` which precedes `C3`. The ordering between steps in each lettered *sequence* is not defined. Note that though the total order is not defined, only one of the many permutations occurs during each execution. As an *instance* progresses through its *process*, this order is recorded for consistent state restoration. This preservation of the total order in an *instance* is handled transparently to support the illusion of programming as though for a single *session*.

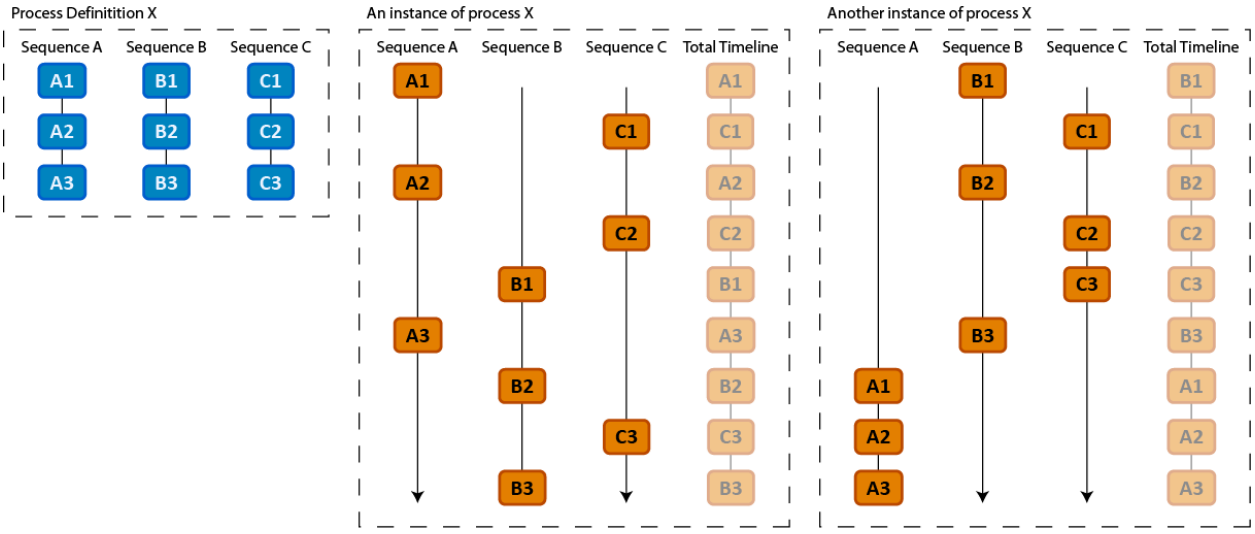



**Figure 6 - Two legitimate runtime ordering of steps in a Process with Sequences executed in parallel**

### *Synchronization*

> *The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled.*

```
00 public class Synchronization : ProcessBase
```

```
01 {
02     public override IEnumerable<IFavor> Start(object data)
03     {
04         var other =  AsyncCall(Other);
05 
06         // Split
07         yield return other;
08 
09         // First Task
10         yield return AssignTask("Advance Main branch", Initiator);
11 
12         // Wait for other branch
13         yield return Wait(other);
14 
15         // Final Task
16         yield return AssignTask("Finish", Initiator);
17     }
18 
19     IEnumerable<IFavor> Other()
20     {
21         // Second Task
22         yield return AssignTask("Advance Other branch", Initiator);
23     }
24 }
```

**Code Listing 7 – Synchronization Implementation Example**

The **Synchronization** pattern can be implemented by using the `Wait` *favor*. In Code Listing 7, the **Parallel Split** example is extended. A reference to the `AsyncCall` *favor* for the `Other` *sequence* is retained on Line 04 in order that it may be used on Line 13. By Line 13, both *tasks* have been assigned. If the labeled `"Second Task"` and, therefore, the `Other` *branch* is completed before the labeled `"First Task"`, then Line 13 does not block. If `"First Task"` is completed first, then Line 13's `Wait` causes the `Start` *branch* to be moved to the wait-list pending the completion of the `Other` *branch*. Once the `Other` *branch* completes as a consequence of the `"Second Task"` being completed, or if `Other` was completed first, `Start` continues at Line 16 with the assignment of a `"Final Task"`. Critically, this `"Final Task"` will not be assigned until both of the *branches* have synchronized following the completion of both *tasks*.

### *Exclusive Choice*

> *The divergence of a branch into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to precisely one of the outgoing branches based on a mechanism that can select one of the outgoing branches.*

```
00 public class ExclusiveChoice : ProcessBase
01 {
02     // Expects a boolean value as data
03     public override IEnumerable<IFavor> Start(object data)
04     {
05         if ((bool)data)
06         {
07             // True branch
08             yield return AssignTask("Advance 'true' branch", Initiator);
09         }
10         else
```

```
11          {
12              // False branch
13              yield return AssignTask("Advance 'false' branch", Initiator);
14          }
15      }
16  }
```

Code Listing 8 – Exclusive Choice Implementation Example

An **Exclusive Choice** pattern can be implemented using a traditional `if`-statement control-flow mechanism. In Code Listing 8, the payload provided to the initial *sequence* is cast as a `bool` to determine which `if-else` block should be followed. As expected, only one of the two `if-else` blocks will be executed.

### *Simple Merge*

> *The convergence of two or more branches into a single subsequent branch such that each enablement of an incoming branch results in the thread of control being passed to the subsequent branch.*

```
00 public class SimpleMerge : ProcessBase
01 {
02      // Expects a boolean value as data
03      public override IEnumerable<IFavor> Start(object data)
04      {
05          if ((bool)data)
06          {
07              // True branch
08              yield return AssignTask("Advance 'true' branch", Initiator);
09          }
10          else
11          {
12              // False branch
13              yield return AssignTask("Advance 'false' branch", Initiator);
14          }
15
16          // Common continuation
17          yield return AssignTask("Finish", Initiator);
18      }
19  }
```

Code Listing 9 – Simple Merge Implementation Example

A **Simple Merge** pattern can be implemented as a non-conditional code listing following a conditional code listing. Code Listing 9 expands upon Code Listing 8 with such a continuation.

## Advanced Branching and Synchronization

### *Multi-Choice*

> *The divergence of a branch into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to one or more of the outgoing branches based on a mechanism that selects one or more outgoing branches.*

```
00 public class MultiChoice : ProcessBase
```

```
{
    // Expects a string as data
    public override IEnumerable<IFavor> Start(object data)
    {
        var one = AsyncCall(One);
        var two = AsyncCall(Two);

        switch ((string)data)
        {
            case "Just One":
                yield return one;
                break;
            case "Just Two":
                yield return two;
                break;
            default:
                yield return one;
                yield return two;
                break;
        }
    }

    IEnumerable<IFavor> One()
    {
        yield return AssignTask("Advance One", Initiator);
    }

    IEnumerable<IFavor> Two()
    {
        yield return AssignTask("Advance Two", Initiator);
    }
}
```

**Code Listing 10 – Multi-Choice Implementation Example**

The **Multi-Choice** pattern is easily achieved with C# control-flow mechanisms. In Code Listing 10, a `switch` statement is used to the *instance's* input data to determine whether `One`, `Two`, or both `One` and `Two` should be executed. For effect, both *favors* are created (a benign calculation) even though within some *instances*, only one will be emitted.

### *Structured Synchronizing Merge*

> *The convergence of two or more branches (which diverged earlier in the process at a uniquely identifiable point) into a single subsequent branch such that the thread of control is passed to the subsequent branch when each active incoming branch has been enabled. The* Structured Synchronizing Merge *occurs in a structured context, i.e. there must be a single* Multi-Choice *construct earlier in the process model with which the* Structured Synchronizing Merge *is associated and it must merge all of the branches emanating from the* Multi-Choice. *These branches must either flow from the* Structured Synchronizing Merge *without any splits or joins or they must be structured in form (i.e. balanced splits and joins).*

```
public class StructuredSynchronizingMerge : ProcessBase
{
```

```
02      // Expects a string as data
03      public override IEnumerable<IFavor> Start(object data)
04      {
05          var one = AsyncCall(One);
06          var two = AsyncCall(Two);
07
08          // Split
09          switch ((string)data)
10          {
11              case "Just One":
12                  yield return one;
13                  break;
14              case "Just Two":
15                  yield return two;
16                  break;
17              default:
18                  yield return one;
19                  yield return two;
20                  break;
21          }
22
23          // Merge
24          if (one.IsUsed) yield return Wait(one);
25          if (two.IsUsed) yield return Wait(two);
26      }
27
28      IEnumerable<IFavor> One()
29      {
30          yield return AssignTask("Advance One", Initiator);
31      }
32
33      IEnumerable<IFavor> Two()
34      {
35          yield return AssignTask("Advance Two", Initiator);
36      }
37  }
```

**Code Listing 11 – Structured Synchronizing Merge Implementation Example**

The **Structured Synchronizing Merge** pattern can be implemented by waiting on the various created *branches*. Code Listing 11 extends Code Listing 10. A runtime error would occur if a `Wait` were emitted for an `AsyncCall` that had not already been emitted, hence the `IsUsed` check.

### *Multi-Merge*

> *The convergence of two or more branches into a single subsequent branch such that each enablement of an incoming branch results in the thread of control being passed to the subsequent branch.*

```
00  public class MultiMerge : ProcessBase
01  {
02      // Expects a string as data
03      public override IEnumerable<IFavor> Start(object data)
04      {
05          var one = AsyncCall(One);
06          var two = AsyncCall(Two);
```

```
07
08          switch ((string)data)
09          {
10              case "Just One":
11                  yield return one;
12                  break;
13              case "Just Two":
14                  yield return two;
15                  break;
16              default:
17                  yield return one;
18                  yield return two;
19                  break;
20          }
21      }
22
23      IEnumerable<IFavor> One()
24      {
25          yield return AssignTask("Advance One", Initiator);
26
27          yield return Jump(Final);
28      }
29
30      IEnumerable<IFavor> Two()
31      {
32          yield return AssignTask("Advance Two", Initiator);
33
34          yield return Jump(Final);
35      }
36
37      IEnumerable<IFavor> Final()
38      {
39          yield return AssignTask("Finish", Initiator);
40      }
41  }
```

**Code Listing 12 – Multi-Merge Implementation Example**

The **Multi-Merge** pattern can be implemented with `Jump`. Code Listing 12 extends Code Listing 10 such that upon the completion of each of `One` and `Two`, execution is transferred to two independent *branches* of the shared `Final` *sequence*.

### *Structured Discriminator*

> *The convergence of two or more branches into a single subsequent branch following a corresponding divergence earlier in the process model such that the thread of control is passed to the subsequent branch when the first incoming branch has been enabled. Subsequent enablements of incoming branches do not result in the thread of control being passed on. The* Structured Discriminator *construct resets when all incoming branches have been enabled. The* Structured Discriminator *occurs in a structured context, i.e. there must be a single* Parallel Split *construct earlier in the process model with which the* Structured Discriminator *is associated and it must merge all of the branches emanating from the* Structured Discriminator. *These branches must either flow from the* Parallel Split *to the*

> Structured Discriminator *without any splits or joins or they must be structured in form (i.e. balanced splits and joins).*

```
00 public class StructuredDiscriminator : ProcessBase
01 {
02     public override IEnumerable<IFavor> Start(object data)
03     {
04         var one = AsyncCall(One);
05         var two = AsyncCall(Two);
06 
07         // Split
08         switch ((string)data)
09         {
10             case "Just One":
11                 yield return one;
12                 break;
13             case "Just Two":
14                 yield return two;
15                 break;
16             default:
17                 yield return one;
18                 yield return two;
19                 break;
20         }
21 
22         yield return Wait(one, two);
23     }
24 
25     IEnumerable<IFavor> One()
26     {
27         yield return AssignTask("Advance One", Initiator);
28     }
29 
30     IEnumerable<IFavor> Two()
31     {
32         yield return AssignTask("Advance Two", Initiator);
33     }
34 }
```

**Code Listing 13 – Structured Discriminator Implementation Example**

The **Structured Discriminator** pattern can be implemented using the `Wait` *favor* with multiple wait-targets. In Code Listing 13, Line 22 emits such a *favor*. Once either of the supplemental *branches* completes, `Start` is resumed (to terminate). The remaining *branch* is left to continue on its own as the sole remaining *branch* for the *instance*. Having passed multiple wait-targets to `Wait` asks the engine to unblock once ANY of the targets has been completed or canceled.

### *Blocking Discriminator*

> *The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when the first active incoming branch has been enabled. The* Blocking Discriminator *construct resets when all active incoming branches have been*

> *enabled once for the same process instance. Subsequent enablements of incoming branches are blocked until the* Blocking Discriminator *has reset.*

```
00 public class BlockingDiscriminator : ProcessBase
01 {
02     public override IEnumerable<IFavor> Start(object data)
03     {
04         yield return Subscribe("QUEUE", "", Enqueue);
05     }
06 
07     IAsyncCall previousEnablement = null;
08 
09     IEnumerable<IFavor> Enqueue(object newEntry)
10     {
11         var currentEnablement = AsyncCall(ProcessEntry,
12             (string)newEntry, previousEnablement);
13 
14         previousEnablement = currentEnablement;
15 
16         yield return currentEnablement;
17     }
18 
19     IEnumerable<IFavor> ProcessEntry(string newEntry,
20         IAsyncCall myPreviousEnablement)
21     {
22         if (myPreviousEnablement != null)
23             yield return Wait(myPreviousEnablement);
24 
25         var splitOne = AsyncCall(SplitOne, newEntry);
26         var splitTwo = AsyncCall(SplitTwo, newEntry);
27 
28         yield return splitOne;
29         yield return splitTwo;
30 
31         yield return Wait(splitOne);
32         yield return Wait(splitTwo);
33     }
34 
35     IEnumerable<IFavor> SplitOne(string newEntry)
36     {
37         yield return AssignTask(newEntry + "A: One at a time ", Initiator);
38     }
39 
40     IEnumerable<IFavor> SplitTwo(string newEntry)
41     {
42         yield return AssignTask(newEntry + "B: One at a time ", Initiator);
43     }
44 }
```

**Code Listing 14 – Blocking Discriminator Implementation Example**

The **Blocking Discriminator** pattern can be implemented as a *subscription* that processes incoming work through a queue, a **Parallel Split**, and subsequent **Merge**. In Code Listing 14, the `Start` *sequence* merely opens the *subscription* for the `Enqueue` *sequence*. After this initial *session*, the *instance's* `Start` *branch* completes and the *instance* is held unfinished only by its `Enqueue` *subscription* and *branch*. With each new entry, as input through the *subscription*, a new *branch* of `Enqueue` is run. `previousEnablement`

serves as a references to the back of a queue that forms within BindFlow's wait-list as additional enablements arrive. `currentEnablement` is assigned as a new `AsyncCall` *favor* to `ProcessEntry`, additionally storing the parameters for `ProcessEntry`. The `currentEnablement`, newly created is set to be pushed to the back of the queue by reassigning `previousEnablement` and is then emitted. This starts a new *branch* of `ProcessEntry` which immediately `Waits` on the stored `myPreviousEnablement` which was passed in to the `ProcessEntry` *sequence* as a parameter. If this `previousEnablement` was already completed, then the `Wait` does not block. Each enablement continues on to the **Split** and **Merge** starting on Line 25. Only once the **Merge** is completed and the `ProcessEntry` *branch* terminates implicitly on line 33 is any waiting following entry allowed to take its own turn through the remainder of `ProcessEntry` from its own line 25. Note that in this example, the *subscription* is never removed with an `Unsubscribe` and thus the *instance* never terminates. Note that because BindFlow *instances* store and replay every input for each *session*, a single *instance* would get slower and require more storage, linearly, as additional entries are accepted by `Enqueue`. Extending this example to spawn new *instances* at strategic times, such as when the queue is momentarily empty, could keep performance optimal.

### *Cancelling Discriminator*

> *The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when the first active incoming branch has been enabled. Triggering the* Cancelling Discriminator *also cancels the execution of all of the other incoming branches and resets the construct.*

```
00 public class CancellingDiscriminator : ProcessBase
01 {
02     public override IEnumerable<IFavor> Start(object data)
03     {
04         var splitOne = AsyncCall(SplitOne);
05         var splitTwo = AsyncCall(SplitTwo);
06 
07         yield return splitOne;
08         yield return splitTwo;
09 
10         yield return Wait(splitOne, splitTwo);
11 
12         if (splitOne.IsComplete) yield return Cancel(splitTwo);
13         if (splitTwo.IsComplete) yield return Cancel(splitOne);
14     }
15 
16     IEnumerable<IFavor> SplitOne()
17     {
18         yield return AssignTask("One", Initiator);
19     }
20 
21     IEnumerable<IFavor> SplitTwo()
22     {
23         yield return AssignTask("Two", Initiator);
24     }
25 }
```

Code Listing 15 – Cancelling Discriminator Implementation Example

The **Cancelling Discriminator** pattern can be implemented by holding references to each of multiple *branches*, waiting on any of the *branches*, and cancelling the remaining *branches* after the `Wait` is unblocked. Code Listing 15 is such an implementation.

### *Structured Partial Join*

> *The convergence of two or more branches (say m) into a single subsequent branch following a corresponding divergence earlier in the process model such that the thread of control is passed to the subsequent branch when n of the incoming branches have been enabled where n is less than m. Subsequent enablements of incoming branches do not result in the thread of control being passed on. The join construct resets when all active incoming branches have been enabled. The join occurs in a structured context, i.e. there must be a single* Parallel Split *construct earlier in the process model with which the join is associated and it must merge all of the branches emanating from the* Parallel Split. *These branches must either flow from the* Parallel Split *to the join without any splits or joins or be structured in form (i.e. balanced splits and joins).*

```
00 public class StructuredPartialJoin : ProcessBase
01 {
02     // Expects an int as data
03     public override IEnumerable<IFavor> Start(object data)
04     {
05         IAsyncCall[] assignments = new IAsyncCall[(int)data]; // m
06 
07         // Create the assignments
08         for (int counter = 0; counter < assignments.Length; counter++)
09             assignments[counter] = AsyncCall(Assignment, counter);
10 
11         // Emit the assignments
12         for (int counter = 0; counter < assignments.Length; counter++)
13             yield return assignments[counter];
14 
15         // Wait for m-1 assignments to be completed
16         IWait wait = Wait(assignments);
17         int remaining;
18         do
19         {
20             yield return wait;
21             remaining = wait.NotCompleted.Count();
22             wait = Wait(wait.NotCompleted.ToArray());
23         } while (remaining > 1);
24 
25         yield return AssignTask("Finish", Initiator);
26     }
27 
28     // Expects an int as data
29     IEnumerable<IFavor> Assignment(int assignmentNumber)
30     {
31         yield return AssignTask("Advance #" + assignmentNumber, Initiator);
32     }
33 }
```

Code Listing 16 – Structured Partial Join Implementation Example

The **Structured Partial Join** pattern can be implemented as in Code Listing 16. Some integer `m >= 2` is passed in to the new *instance* as `object data`. An array is allocated to hold m `AsyncCall` *favors* on Line 05, and is then filled by the loop on Line 08. The Assignment *sequence* takes an `assignmentNumber` parameter, an identity to help differentiate each of its *branches* for the user. This identity is passed to the *sequence* through the `AsyncCall`. The second parameter of the `AsyncCall` *favor* factory method on Line 09 provides this value as the loop's incremented counter. Once the `AsyncCalls` are created and referenced in the assignments array, they are emitted for parallel execution by the loop on Line 12. To fulfill the requirements of the **Partial Join**, we choose to `Wait` on m-1 of the Assignment *branches* to complete before continuing to the final `"Finish"` *task*. Line 16 holds a reference to the `Wait` *favor* so that Lines 21 and 22 may refer to it. Including multiple wait-targets, `AsyncCalls` or `Spawns`, in a `Wait` blocks until any one of the targets is terminated or canceled. Line 21 stores the count of the remaining uncompleted targets for the loop condition on Line 25. Line 22 reassigns a new `Wait` *favor* with only the remaining targets. The loop is repeated until only `1` unfinished target remains indicating that `m-1` targets did finish. The last remaining *branch* of Assignment is not canceled, but no longer blocks the rest of the *instance*. The *process* terminates after both the last remaining assignment and the `"Finish"` *task* of Line 25 are completed in either order.

### *Blocking Partial Join*

> *The convergence of two or more branches (say m) into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when n of the incoming branches has been enabled (where 2 = n < m). The join construct resets when all active incoming branches have been enabled once for the same process instance. Subsequent enablements of incoming branches are blocked until the join has reset.*

```
00 public class BlockingPartialJoin : ProcessBase
01 {
02     public override IEnumerable<IFavor> Start(object data)
03     {
04         yield return Subscribe("QUEUE", "", Enqueue);
05     }
06
07     IAsyncCall previousEnablement = null;
08
09     IEnumerable<IFavor> Enqueue(object newEntry)
10     {
11         var currentEnablement = AsyncCall(ProcessEntry,
12             (string)newEntry, previousEnablement);
13
14         previousEnablement = currentEnablement;
15
16         yield return currentEnablement;
17     }
18
19     IEnumerable<IFavor> ProcessEntry(string newEntry,
20         IAsyncCall myPreviousEnablement)
21     {
22         if (myPreviousEnablement != null)
```

```
23              yield return Wait(myPreviousEnablement);
24
25          IAsyncCall[] assignments = new IAsyncCall[3]; // m
26
27          // Create the assignments
28          for (int counter = 0; counter < assignments.Length; counter++)
29              assignments[counter] = AsyncCall(Assignment, counter);
30
31          // Emit the assignments
32          for (int counter = 0; counter < assignments.Length; counter++)
33              yield return assignments[counter];
34
35          // Wait for m-1 assignments to be completed
36          IWait wait = Wait(assignments);
37          int remaining;
38          do
39          {
40              yield return wait;
41              remaining = wait.NotCompleted.Count();
42              wait = Wait(wait.NotCompleted.ToArray());
43          } while (remaining > 1);
44
45          yield return AsyncCall(PostJoin);
46      }
47
48      IEnumerable<IFavor> Assignment(int assignmentNumber)
49      {
50          yield return AssignTask("Advance #" + assignmentNumber, Initiator);
51      }
52
53      IEnumerable<IFavor> PostJoin()
54      {
55          yield return AssignTask("Finish", Initiator);
56      }
57  }
```

**Code Listing 17 – Blocking Partial Join Implementation Example**

The **Blocking Partial Join** pattern can be implemented as a combination of the **Blocking Discriminator** and the **Structured Partial Join**. Code Listing 17 combines Code Listing 14 and Code Listing 16. The `AsyncCall` on Line 45 followed by an immediate implicit termination of the *sequence* provides the reset mechanism for an extended continuation after the **Partial Join** without blocking further enablements.

### *Cancelling Partial Join*

> *The convergence of two or more branches (say m) into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when n of the incoming branches have been enabled where n is less than m. Triggering the join also cancels the execution of all of the other incoming branches and resets the construct.*

```
00 public class CancellingPartialJoin : ProcessBase
01 {
02      // Expects an int as data
03      public override IEnumerable<IFavor> Start(object data)
```

```
04      {
05          IAsyncCall[] assignments = new IAsyncCall[(int)data]; // m
06
07          // Create the assignments
08          for (int counter = 0; counter < assignments.Length; counter++)
09              assignments[counter] = AsyncCall(Assignment, counter);
10
11          // Emit the assignments
12          for (int counter = 0; counter < assignments.Length; counter++)
13              yield return assignments[counter];
14
15          // Wait for m-1 assignments to be completed
16          IWait wait = Wait(assignments);
17          int remaining;
18          do
19          {
20              yield return wait;
21              remaining = wait.NotCompleted.Count();
22              wait = Wait(wait.NotCompleted.ToArray());
23          } while (remaining > 1);
24
25          foreach (var remainingAssignment in wait.NotCompleted)
26          {
27              yield return Cancel(remainingAssignment);
28          }
29
30          yield return AssignTask("Finish", Initiator);
31      }
32
33      // Expects an int as data
34      IEnumerable<IFavor> Assignment(int assignmentNumber)
35      {
36          yield return AssignTask("Advance #" + assignmentNumber, Initiator);
37      }
38  }
```

**Code Listing 18 – Canceling Partial Join Implementation Example**

The **Canceling Partial Join** pattern can be implemented as a variation of the **Structured Partial Join** of Code Listing 16. Code Listing 18 adds the loop of Line 25 to cancel any remaining Assignment *branches* following the **Partial Join**. In this example, `n=m-1` is hard-coded, exactly one remaining *branch* is canceled.

### *Generalized AND-Join*

> *The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled. Additional triggers received on one or more branches between firings of the join persist and are retained for future firings. Over time, each of the incoming branches should deliver the same number of triggers to the AND-join construct (although obviously, the timing of these triggers may vary).*

```
00  public class GeneralizedAndJoin : ProcessBase
01  {
02      public override IEnumerable<IFavor> Start(object data)
03      {
```

```
04          yield return Subscribe("QUEUE", "", Enqueue);
05      }
06
07      IAsyncCall previousEnablement = null;
08
09      IEnumerable<IFavor> Enqueue(object newEntry)
10      {
11          var currentEnablement = AsyncCall(ProcessEntry,
12              (string)newEntry, previousEnablement);
13
14          previousEnablement = currentEnablement;
15
16          yield return currentEnablement;
17      }
18
19      IEnumerable<IFavor> ProcessEntry(string newEntry,
20          IAsyncCall myPreviousEnablement)
21      {
22          var splitOne = AsyncCall(SplitOne, newEntry);
23          var splitTwo = AsyncCall(SplitTwo, newEntry);
24
25          yield return splitOne;
26          yield return splitTwo;
27
28          yield return Wait(splitOne);
29          yield return Wait(splitTwo);
30
31          if (myPreviousEnablement != null)
32              yield return Wait(myPreviousEnablement);
33
34          yield return AsyncCall(PostJoin);
35      }
36
37      IEnumerable<IFavor> SplitOne(string newEntry)
38      {
39          yield return AssignTask(newEntry + "A: One at a time ", Initiator);
40      }
41
42      IEnumerable<IFavor> SplitTwo(string newEntry)
43      {
44          yield return AssignTask(newEntry + "B: One at a time ", Initiator);
45      }
46
47      IEnumerable<IFavor> PostJoin()
48      {
49          yield return AssignTask("Finish", Initiator);
50      }
51  }
```

**Code Listing 19 – Generalized AND-Join Implementation Example**

The **Generalized AND-Join** pattern can be implemented as a variation of the **Blocking Discriminator**. Code Listing 19 is based on Code Listing 14 such that the `Wait` emission is delayed until Line 32 and a `PostJoin` *sequence* is added as an extended continuation after the join has reset.

### *Local Synchronizing Merge*

> *The convergence of two or more branches which diverged earlier in the process into a single subsequent branch such that the thread of control is passed to the subsequent branch when each active incoming branch has been enabled. Determination of how many branches require synchronization is made on the basis on 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.*

```
00 public class LocalSynchronizingMerge : ProcessBase
01 {
02     public override IEnumerable<IFavor> Start(object data)
03     {
04         LocalContext context = new LocalContext();
05 
06         var selection = AssignTask("Provide some selection of A, B, and C",
07             Initiator, "ABC");
08 
09         yield return selection;
10 
11         if (((string)selection.Result).Contains("A"))
12             context.threads.Add(AsyncCall(GenericWork, "A"));
13 
14         if (((string)selection.Result).Contains("B"))
15             context.threads.Add(AsyncCall(GenericWork, "B"));
16 
17         if (((string)selection.Result).Contains("C"))
18         {
19             context.waitOnC = true;
20             context.c = AsyncCall(C, context);
21             context.threads.Add(context.c);
22         }
23 
24         foreach (var thread in context.threads)
25             yield return thread;
26 
27         yield return AsyncCall(Finish, context);
28     }
29 
30     IEnumerable<IFavor> GenericWork(string data)
31     {
32         yield return AssignTask(data, Initiator);
33     }
34 
35     IEnumerable<IFavor> C(LocalContext context)
36     {
37         var choice = AssignTask("C", Initiator, true);
38         yield return choice;
39         context.waitOnC = (bool)choice.Result;
40 
41         // replace the c that we're looking for,
42         //allowing the finish to trigger.
43         context.c = AsyncCall(C2);
44         context.threads.Add(context.c);
```

```
45
46          yield return context.c;
47      }
48
49      IEnumerable<IFavor> C2()
50      {
51          yield return AssignTask("C2", Initiator);
52      }
53
54      IEnumerable<IFavor> Finish(LocalContext context)
55      {
56          while ((!context.waitOnC && context.threads.Any(t =>
57              t != context.c)) ||
58              (context.waitOnC && context.threads.Any()))
59          {
60              var waitAny = Wait(context.threads.ToArray());
61              yield return waitAny;
62
63              foreach (var thread in waitAny.Completed)
64              {
65                  context.threads.Remove(thread);
66              }
67          }
68
69          yield return AssignTask("Merge Complete", Initiator);
70      }
71  }
72
73  class LocalContext
74  {
75      public List<IWaitable> threads = new List<IWaitable>();
76      public IAsyncCall c = null;
77      public bool waitOnC = false;
78  }
```

**Code Listing 20 - Local Synchronizing Merge Implementation Example**

The **Local Synchronizing Merge** pattern by passing an object representing local context through to the different actors of the construct. Code Listing 20 mimics the interactive demonstration by van der Aalst et al.[1] for this pattern. Any or all of `A`, `B`, and `C` are chosen to execute. As *branch* `C` has extended functionality to control the **Synchronizing Merge**, `LocalContext` is shared between the `A`, `B` section and the `C` section. After the **Split**, the **Merge** of `C` is conditional. If `C` should be **Merged**, the local context is populated with `C`'s continuation; otherwise, `C`'s continuation is run without blocking the **Merge**.

### *General Synchronizing Merge*

> *The convergence of two or more branches which diverged earlier in the process into a single subsequent branch such that the thread of control is passed to the subsequent branch when either (1) each active incoming branch has been enabled or (2) it is not possible that any branch that has not yet been enabled will be enabled at any future time.*

```
00  public class GeneralSynchronizingMerge : ProcessBase
01  {
02      public override IEnumerable<IFavor> Start(object data)
03      {
```

```
        var selection = AssignTask("Provide some selection of A, B, and C",
            Initiator, "ABC");

        yield return selection;

        var a = AsyncCall(GenericWork, "A");
        var b = AsyncCall(GenericWork, "B");
        var c = AsyncCall(C);

        if (((string)selection.Result).Contains("A"))
            yield return a;

        if (((string)selection.Result).Contains("B"))
            yield return b;

        if (((string)selection.Result).Contains("C"))
            yield return c;

        if (a.IsUsed) yield return Wait(a);
        if (b.IsUsed) yield return Wait(b);
        if (c.IsUsed) yield return Wait(c);

        yield return AssignTask("Merge Complete", Initiator);
    }

    IEnumerable<IFavor> GenericWork(string taskSummary)
    {
        yield return AssignTask(taskSummary, Initiator);
    }

    IEnumerable<IFavor> C()
    {
        IAssignTask choice = null;

        while (choice == null || choice.Result == "C")
        {
            choice = AssignTask("C, D, or E?", Initiator, "C");

            yield return choice;
        }

        if ((string)choice.Result == "D")
        {
            var d = AsyncCall(D);
            yield return d;
            yield return Wait(d);
        }
        else if ((string)choice.Result == "E")
        {
            yield return AsyncCall(E);
        }
    }

    IEnumerable<IFavor> D()
    {
        yield return AssignTask("D", Initiator);
    }
```

```
61
62      IEnumerable<IFavor> E()
63      {
64          yield return AssignTask("E", Initiator);
65      }
66  }
```

**Code Listing 21 - General Synchronizing Merge Implementation Example**

The **General Synchronizing Merge** pattern can be implemented using techniques used in previous patterns. Code Listing 21 implements the demonstration of this same pattern by van der Aalst et al.[1] and is similar to Code Listing 20 but without the need for a local context and with the addition of the `while` loop on Line 37.

### *Thread Merge*

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

```
00 public class ThreadMerge : ProcessBase
01 {
02      public override IEnumerable<IFavor> Start(object data)
03      {
04          List<IAsyncCall> threads = new List<IAsyncCall>();
05
06          for (int x = 0; x < (int)data; x++)
07          {
08              threads.Add(AsyncCall(ThreadWork, x));
09          }
10
11          foreach (var thread in threads) yield return thread;
12
13          yield return AsyncCall(Merge, threads);
14      }
15
16      IEnumerable<IFavor> ThreadWork(int threadNumber)
17      {
18          yield return AssignTask("Move forward on " +
19              threadNumber.ToString(), Initiator);
20      }
21
22      IEnumerable<IFavor> Merge(List<IAsyncCall> threads)
23      {
24          foreach (var thread in threads)
25          {
26              yield return Wait(thread);
27          }
28
29          yield return AssignTask("All threads have been merged", Initiator);
30      }
31  }
```

**Code Listing 22 – Thread Merge Implementation Example**

The **Thread Merge** pattern can be implemented using the `Wait` *favor*. Code Listing 22 stores a List of `AsyncCall` *favors* of some length determined at runtime, runs multiple parallel threads. The list of threads to be merged is passed to the `Merge` *sequence* which waits on all of the threads.

### *Thread Split*

> *At a given point in a process, a nominated number of execution threads can be initiated in a single branch of the same process instance.*

```
00 public class ThreadSplit : ProcessBase
01 {
02     public override IEnumerable<IFavor> Start(object data)
03     {
04         for (int x = 0; x < (int)data; x++)
05         {
06             yield return AsyncCall(ThreadWork, x);
07         }
08     }
09 
10     IEnumerable<IFavor> ThreadWork(int threadNumber)
11     {
12         yield return AssignTask("Move forward on " +
13             threadNumber.ToString(), Initiator);
14     }
15 }
```

**Code Listing 23 – Thread Split Implementation Example**

The **Thread Split** pattern can be implemented as the first part of the **Thread Merge** example. Code Listing 23 excerpts Code Listing 22 to only include the **Parallel Splits**. The references to the split threads are not maintained in `Start`. `Start` could continue with other independent sections of execution, possibly with additional and isolated **Thread Split** implementations or other patterns.

## Multiple Instance

As noted in the terminology section, van der Aalst et al.[1] use the term “instance” to refer to a particular execution of a section of work in a workflow rather than a particular execution of a *process*. In BindFlow, any code can be executed multiple times within a single *instance* of a *process*. Code is internally referenceable at the *sequence* level (and as its *branches*, at runtime). A *sequence*, as the starting point for any complex code path, can therefore represent any section of work of the *process*.

### *Multiple Instances without Synchronization*

> *Within a given process instance, multiple instances of a task can be created. These instances are independent of each other and run concurrently. There is no requirement to synchronize them upon completion. Each of the instances of the multiple instance task that are created must execute within the context of the process instance from which they were started (i.e. they must share the same case identifier and have access to the same data elements) and each of them must execute independently from and without reference to the task that started them.*

```
00 public class MultipleInstancesWithoutSynchronization : ProcessBase
```

```
{
    string InstanceData = "Some Value";

    public override IEnumerable<IFavor> Start(object data)
    {
        yield return AsyncCall(One);
        yield return AsyncCall(Two);
    }

    IEnumerable<IFavor> One()
    {
        yield return AssignTask("Advance One, " + InstanceData, Initiator);
    }

    IEnumerable<IFavor> Two()
    {
        yield return AssignTask("Advance Two, " + InstanceData, Initiator);
    }
}
```

**Code Listing 24 – Multiple Instances Without Synchronization Implementation Example**

The **Multiple Instances Without Synchronization** pattern can be implemented the same as a **Parallel Split**. The unit of work can be multiple *branches* of the same *sequence* or multiple *branches* of multiple *sequences*. For clarity, Code Listing 24 uses two unique *sequences* which both have access to the same *instance*-wide data.

### *Multiple Instances with a Priori Design-Time Knowledge*

> *Within a given process instance, multiple instances of a task can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently. It is necessary to synchronize the task instances at completion before any subsequent tasks can be triggered.*

```
public class MultipleInstancesWithAPrioriDesignTimeKnowledge : ProcessBase
{
    public override IEnumerable<IFavor> Start(object data)
    {
        // Create the assignments
        var assignment1 = AsyncCall(Assignment, 1);
        var assignment2 = AsyncCall(Assignment, 2);
        var assignment3 = AsyncCall(Assignment, 3);

        // Emit the assignments
        yield return assignment1;
        yield return assignment2;
        yield return assignment3;

        // Wait for all of the assignments to be completed (in any order)
        yield return Wait(assignment1);
        yield return Wait(assignment2);
        yield return Wait(assignment3);

        yield return AssignTask("Finish", Initiator);
    }
```

```
21
22      // Expects an int as data
23      IEnumerable<IFavor> Assignment(int assignmentNumber)
24      {
25          yield return AssignTask("Advance #" + assignmentNumber, Initiator);
26      }
27  }
```

**Code Listing 25 – Multiple Instances with a Priori Design-Time Knowledge Implementation Example**

The **Multiple Instances with a Priori Design-Time Knowledge** pattern can be implemented with a series of **Thread Splits** and **Thread Merges** where the number of *branches* or threads is hard-coded. **Code Listing 25** generates a fixed three *branches* or threads of the `Assignment` *sequence*. All three threads are synchronized before `Start` may continue.

### *Multiple Instances with a Priori Run-Time Knowledge*

> *Within a given process instance, multiple instances of a task can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications, but is known before the task instances must be created. Once initiated, these instances are independent of each other and run concurrently. It is necessary to synchronize the instances at completion before any subsequent tasks can be triggered.*

```
00 public class MultipleInstancesWithAPrioriRunTimeKnowledge : ProcessBase
01 {
02      // Expects an int as data
03      public override IEnumerable<IFavor> Start(object data)
04      {
05          IAsyncCall[] assignments = new IAsyncCall[(int)data];
06
07          // Create the assignments
08          for (int counter = 0; counter < assignments.Length; counter++)
09              assignments[counter] = AsyncCall(Assignment, counter);
10
11          // Emit the assignments
12          for (int counter = 0; counter < assignments.Length; counter++)
13              yield return assignments[counter];
14
15          // Wait for the assignments to be completed
16          for (int counter = 0; counter < assignments.Length; counter++)
17              yield return Wait(assignments[counter]);
18
19          yield return AssignTask("Finish", Initiator);
20      }
21
22      // Expects an int as data
23      IEnumerable<IFavor> Assignment(int assignmentNumber)
24      {
25          yield return AssignTask("Advance #" + assignmentNumber, Initiator);
26      }
27  }
```

**Code Listing 26 – Multiple Instances with a Priori Run-Time Knowledge Implementation Example**

The **Multiple Instances with a Priori Run-Time Knowledge** pattern can be implemented with a dynamic loop. Code Listing 26 provides such an implementation.

### *Multiple Instances without a Priori Run-Time Knowledge*

> *Within a given process instance, multiple instances of a task can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications and is not known until the final instance has completed. Once initiated, these instances are independent of each other and run concurrently. At any time, whilst instances are running, it is possible for additional instances to be initiated. It is necessary to synchronize the instances at completion before any subsequent tasks can be triggered.*

```
00 public class MultipleInstancesWithoutAPrioriRunTimeKnowledge : ProcessBase
01 {
02     // Expects an int as data
03     public override IEnumerable<IFavor> Start(object data)
04     {
05         var assignments = new List<IWaitable>();
06 
07         var primer = Call(Assignment);
08         yield return primer;
09 
10         int toCreate = (int)primer.Result;
11 
12         while (assignments.Any() || 0 < toCreate)
13         {
14             // Add toCreate new assignments, possibly 0
15             for (int x = 0; x < toCreate; x++)
16             {
17                 var assignment = AsyncCall(Assignment);
18                 assignments.Add(assignment);
19                 yield return assignment;
20             }
21 
22             // Wait on any of the remaining assignments
23             var wait = Wait(assignments.ToArray());
24             yield return wait;
25 
26             // Single is safe as we are assured that one and
27             // only one assignment can complete per session
28             toCreate = (int)wait.Completed.Single().Result;
29 
30             // Reduce the number of assignments remaining by
31             // the one that we just processed
32             assignments = wait.NotCompleted.ToList();
33         }
34 
35         // A final task
36         yield return AssignTask("Finish", Initiator);
37     }
38 
39     IEnumerable<IFavor> Assignment()
40     {
```

```
41          var task = AssignTask("Create how many more?", Initiator, 0);
42          yield return task;
43          yield return Return((int)task.Result);
44      }
45  }
```

**Code Listing 27 - Multiple Instances without a Priori Run-Time Knowledge Implementation Example**

The **Multiple Instances without a Priori Run-Time Knowledge** pattern can be implemented with a dynamically managed list of unfinished *branches*. Code Listing 27 begins with a blocking call to `Assignment` to get positive integer result of how many additional *tasks* to create. Each requested *branch* is added to the dynamic list. Each assignment asks the user for a number of additional *branches* to create. The *instance* continues to the `"Finish"` *task* on Line 36 after the count of completed additional *branches* reaches the total count of those requested.

### *Static Partial Join for Multiple Instances*

> *Within a given process instance, multiple concurrent instances of a task (say m) can be created. The required number of instances is known when the first task instance commences. Once n of the task instances have completed (where n is less than m), the next task in the process is triggered. Subsequent completions of the remaining m-n instances are inconsequential, however all instances must have completed in order for the join construct to reset and be subsequently re-enabled.*

```
00 public class StaticPartialJoinForMultipleInstances : ProcessBase
01 {
02      public override IEnumerable<IFavor> Start(object data)
03      {
04          yield return Subscribe("QUEUE", "", Enqueue);
05      }
06
07      IAsyncCall previousEnablement = null;
08
09      IEnumerable<IFavor> Enqueue(object newEntry)
10      {
11          var currentEnablement = AsyncCall(ProcessEntry,
12              (string)newEntry, previousEnablement);
13
14          previousEnablement = currentEnablement;
15
16          yield return currentEnablement;
17      }
18
19      IEnumerable<IFavor> ProcessEntry(string newEntry,
20          IAsyncCall myPreviousEnablement)
21      {
22          if (myPreviousEnablement != null)
23              yield return Wait(myPreviousEnablement);
24
25          IAsyncCall[] assignments = new IAsyncCall[3]; // m
26
27          // Create the assignments
28          for (int counter = 0; counter < assignments.Length; counter++)
29              assignments[counter] = AsyncCall(Assignment, counter);
```

```
30
31          // Emit the assignments
32          for (int counter = 0; counter < assignments.Length; counter++)
33              yield return assignments[counter];
34
35          // Wait for m-1 assignments to be completed
36          IWait wait = Wait(assignments);
37          int remaining;
38          do
39          {
40              yield return wait;
41              remaining = wait.NotCompleted.Count();
42              wait = Wait(wait.NotCompleted.ToArray());
43          } while (remaining > 1);
44
45          yield return AsyncCall(PostJoin);
46
47          yield return Wait(wait.NotCompleted.ToArray());
48      }
49
50      IEnumerable<IFavor> Assignment(int assignmentNumber)
51      {
52          yield return AssignTask("Advance #" + assignmentNumber, Initiator);
53      }
54
55      IEnumerable<IFavor> PostJoin()
56      {
57          yield return AssignTask("Finish", Initiator);
58      }
59  }
```

**Code Listing 28 - Static Partial Join for Multiple Instances Implementation Example**

The **Static Partial Join for Multiple Instances** pattern is very similar to the **Blocking Partial Join** in that only `n < m` *branches* must complete before the **Partial Join** runs a post-join step except that all `m` *branches* must be finished before the join resets allowing subsequent enablements. Code Listing 28 adds Line 47 to Code Listing 17 to delay the reset.

### *Cancelling Partial Join for Multiple Instances*

> *Within a given process instance, multiple concurrent instances of a task (say m) can be created. The required number of instances is known when the first task instance commences. Once n of the task instances have completed (where n is less than m), the next task in the process is triggered and the remaining m-n instances are cancelled.*

```
00 public class CancellingPartialJoinForMultipleInstances : ProcessBase
01 {
02      public override IEnumerable<IFavor> Start(object data)
03      {
04          yield return Subscribe("QUEUE", "", Enqueue);
05      }
06
07      IAsyncCall previousEnablement = null;
08
09      IEnumerable<IFavor> Enqueue(object newEntry)
```

```
    {
        var currentEnablement = AsyncCall(ProcessEntry,
            (string)newEntry, previousEnablement);

        previousEnablement = currentEnablement;

        yield return currentEnablement;
    }

    IEnumerable<IFavor> ProcessEntry(string newEntry,
        IAsyncCall myPreviousEnablement)
    {
        if (myPreviousEnablement != null)
            yield return Wait(myPreviousEnablement);

        IAsyncCall[] assignments = new IAsyncCall[3]; // m

        // Create the assignments
        for (int counter = 0; counter < assignments.Length; counter++)
            assignments[counter] = AsyncCall(Assignment, counter);

        // Emit the assignments
        for (int counter = 0; counter < assignments.Length; counter++)
            yield return assignments[counter];

        // Wait for m-1 assignments to be completed
        IWait wait = Wait(assignments);
        int remaining;
        do
        {
            yield return wait;
            remaining = wait.NotCompleted.Count();
            wait = Wait(wait.NotCompleted.ToArray());
        } while (remaining > 1);

        // ToArray copies the enumeration to avoid
        // any lazy evaluation issues
        foreach (var w in wait.NotCompleted.ToArray())
        {
            yield return Cancel(w);
        }

        yield return AsyncCall(PostJoin);
    }

    IEnumerable<IFavor> Assignment(int assignmentNumber)
    {
        yield return AssignTask("Advance #" + assignmentNumber, Initiator);
    }

    IEnumerable<IFavor> PostJoin()
    {
        yield return AssignTask("Finish", Initiator);
    }
}
```

Code Listing 29 - Cancelling Partial Join for Multiple Instances Implementation Example

The **Cancelling Partial Join for Multiple Instances** pattern is very similar to the **Blocking Partial Join** in that only `n < m` *branches* must complete before the **Partial Join** runs a post-join step except that any `m-n` *branches* must be canceled before the join resets allowing subsequent enablements. Code Listing 29 adds the cancelation loop on Line 47 to Code Listing 17 to cancel the remaining *branches*. In this example, the loop will execute exactly once since `n` is effectively hard-coded as `m - 1`.

### *Dynamic Partial Join for Multiple Instances*

> *Within a given process instance, multiple concurrent instances of a task can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications and is not known until the final instance has completed. At any time, whilst instances are running, it is possible for additional instances to be initiated providing the ability to do so had not been disabled. A completion condition is specified which is evaluated each time an instance of the task completes. Once the completion condition evaluates to true, the next task in the process is triggered. Subsequent completions of the remaining task instances are inconsequential and no new instances can be created.*

```
00 public class DynamicPartialJoinForMultipleInstances : ProcessBase
01 {
02     List<IWaitable> allThreads = new List<IWaitable>();
03     IAsyncCall currentWaitingAny = null;
04
05     public override IEnumerable<IFavor> Start(object data)
06     {
07         var newThread = AsyncCall(ProcessEntry);
08         allThreads.Add(newThread);
09         yield return newThread;
10
11         var subscription = Subscribe("QUEUE", null, Enqueue);
12         yield return subscription;
13
14         bool partialJoinSatisfied = false;
15
16         while (!partialJoinSatisfied &&
17             allThreads.Any(t => !t.IsComplete))
18         {
19             currentWaitingAny = AsyncCall(WaitForAny, allThreads);
20             yield return currentWaitingAny;
21             yield return Wait(currentWaitingAny);
22
23             if (currentWaitingAny.Result is bool &&
24                 (bool)currentWaitingAny.Result)
25                 partialJoinSatisfied = true;
26         }
27
28         yield return Unsubscribe(subscription);
29
30         yield return AssignTask("Post-join work", Initiator);
31     }
32
33     IEnumerable<IFavor> WaitForAny(List<IWaitable> threads)
34     {
```

```
35          IWait waitAny = Wait(threads.Where(t => !t.IsComplete).ToArray());
36
37          yield return waitAny;
38
39          yield return Return((bool)waitAny.Completed.Single().Result);
40      }
41
42      IEnumerable<IFavor> Enqueue(object other)
43      {
44          var newThread = AsyncCall(ProcessEntry);
45          allThreads.Add(newThread);
46          yield return newThread;
47
48          yield return Cancel(currentWaitingAny);
49      }
50
51      IEnumerable<IFavor> ProcessEntry()
52      {
53          var task = AssignTask("Finished?", Initiator, false);
54          yield return task;
55
56          yield return Return((bool)task.Result);
57      }
58  }
```

**Code Listing 30 - Dynamic Partial Join for Multiple Instances Implementation Example**

The **Dynamic Partial Join for Multiple Instances** pattern can be implemented by combining several of the techniques demonstrated earlier. Code Listing 30 begins with an initial `ProcessEntry` *branch* and an opening of the `Enqueue` *subscription* introduced in the **Blocking Discriminator** implementation. A reference to a single active `WaitForAny` *branch* is held in `currentWaitingAny` and is accessible from both the `Start` *branch* and any `Enqueue` *branches*. In a loop, `Start` waits on its current `WaitForAny` *branch* which waits on any of the `ProcessEntry` *branches* in `allThreads`. If any `ProcessEntry` *branch* completes, returning a `true` or `false`, the result is passed back to `Start` where the **Partial Join** criteria can be reevaluated. If `Enqueue` receives a new item and adds it to the `allThreads` list, it cancels the `WaitForAny` *branch* help in `currentWaitingAny`, triggering `Start` to unblock and renew its `WaitForAny` *branch* including the newly added `ProcessEntry` *branch*. Once the **Partial Join** criteria is satisfied the *subscription* is closed and some post-join work is initiated. Remaining `ProcessEntries` are allowed to complete normally.

## State-based

### *Deferred Choice*

> *A point in a process where one of several branches is chosen based on interaction with the operating environment. Prior to the decision, all branches represent possible future courses of execution. The decision is made by initiating the first task in one of the branches i.e. there is no explicit choice but rather a race between different branches. After the decision is made, execution alternatives in branches other than the one selected are withdrawn.*

```
00 public class DeferredChoice : ProcessBase
```

```
{
    IAsyncCall one, two;

    public override IEnumerable<IFavor> Start(object data)
    {
        one = AsyncCall(One);
        two = AsyncCall(Two);

        yield return one;

        yield return two;
    }

    IEnumerable<IFavor> One()
    {
        yield return AssignTask("Advance One and cancel Two", Initiator);

        yield return Cancel(two);

        yield return AssignTask("Advance One and finish", Initiator);
    }

    IEnumerable<IFavor> Two()
    {
        yield return AssignTask("Advance Two and cancel One", Initiator);

        yield return Cancel(one);

        yield return AssignTask("Advance Two and finish", Initiator);
    }
}
```

**Code Listing 31 - Deferred Choice Implementation Example**

The **Deferred Choice** pattern can be implemented with a **Parallel Split** in which each *branch* has access to cancel the other *branches*. Code Listing 31 splits to each `One` and `Two`. A *task* is assigned in each *branch* and the user can choose between them. The first *branch* to advance cancels the other and assigns a follow-up *task*. `IAsyncCall one` and `two` are defined at the *instance* level; however, a variation on this implementation might pass the other *branches* as parameters to `One` and `Two`.

### *Interleaved Parallel Routing*

> *A set of tasks has a partial ordering defining the requirements with respect to the order in which they must be executed. Each task in the set must be executed once and they can be completed in any order that accords with the partial order. However, as an additional requirement, no two tasks can be executed at the same time (i.e. no two tasks can be active for the same process instance at the same time).*

```
public class InterleavedParallelRouting : ProcessBase
{
    public override IEnumerable<IFavor> Start(object data)
    {
        List<IAssignTask> orderOne = new List<IAssignTask>();
        List<IAssignTask> orderTwo = new List<IAssignTask>();

```

```
07          for (int x = 0; x < 3; x++)
08          {
09              orderOne.Add(AssignTask("The " + x +
10                  "th created task of order ONE", Initiator));
11              orderTwo.Add(AssignTask("The " + x +
12                  "th created task of order TWO", Initiator));
13          }
14
15          while (orderOne.Any() || orderTwo.Any())
16          {
17              List<IAssignTask> current;
18
19              if (orderOne.Any() && orderTwo.Any())
20              {
21                  var randomTask = IO.IORandom(new[] {0, 1});
22                  yield return randomTask;
23
24                  current = randomTask.Result == 1 ? orderOne : orderTwo;
25              }
26              else if (orderOne.Any())
27              {
28                  current = orderOne;
29              }
30              else
31              {
32                  current = orderTwo;
33              }
34
35              var task = current[0];
36
37              current.RemoveAt(0);
38
39              yield return task;
40          }
41      }
42  }
```

**Code Listing 32 – Interleaved Parallel Routing Implementation Example**

The **Interleaved Parallel Routing** pattern can be implemented by creating multiple lists representing each part of a partial order, and then using some selector to make progress along one of the partial orderings at a time. Code Listing 32 uses a random number generator as the selector input, choosing between one of the two partial orderings at a time.

Note that because BindFlow *sequences* must be *deterministic*, the random number generator is invoked indirectly through the `IO.IORandom` *favor*. Emitting this *favor* allows the `IO.IORandom`'s (hidden) `Perform` method to be called by the engine and generate the value randomly. The engine records the value for future *sessions*. That is, after three iterations of this loop, the random number generator has been invoked only three times, as one would hope. This is in spite of the fact that each new *session* following the completion of the blocking *task* (Line 33) requires the engine to replay all previous inputs and *deterministic* transformations to restore the state of the *instance*. With each new *session*, triggered by the completion of some *task* or other input, the recorded inputs, including the previously numbers

which were chosen at random by a random number generator, are fed into the *process* just as they were their first time through.

### *Milestone*

> *A task is only enabled when the process instance (of which it is part) is in a specific state (typically a parallel branch). The state is assumed to be a specific execution point (also known as a* milestone*) in the process model. When this execution point is reached the nominated task can be enabled. If the process instance has progressed beyond this state, then the task cannot be enabled now or at any future time (i.e. the deadline has expired). Note that the execution does not influence the state itself, i.e. unlike normal control-flow dependencies it is a test rather than a trigger.*

```
00 public class Milestone : ProcessBase
01 {
02     bool open = false;
03 
04     public override string Summarize()
05     {
06         return string.Format("The window is {0}.",
07             open ? "open" : "closed");
08     }
09 
10     // Expects a string as data
11     public override IEnumerable<IFavor> Start(object data)
12     {
13         var listen = Subscribe("WINDOW", null, Handle);
14 
15         yield return listen;
16 
17         yield return AssignTask("Open the window", Initiator);
18 
19         open = true;
20 
21         yield return AssignTask("Close the window", Initiator);
22 
23         open = false;
24 
25         yield return AssignTask("Stop", Initiator);
26 
27         yield return Unsubscribe(listen);
28     }
29 
30     IEnumerable<IFavor> Handle(object data)
31     {
32         if (open)
33         {
34             yield return AssignTask("Process Submission", Initiator);
35         }
36     }
37 }
```

**Code Listing 33 – Milestone Implementation Example**

The **Milestone** pattern can be implemented with a Boolean flag. Code Listing 33 opens a *subscription* to handle data submissions, but only processes such submissions once `Start` has passed Line 18 and until it passed Line 22. As the `"WINDOW"` *subscription* is invoked (perhaps by user action), Line 33 is reached only if the `open` flag is set and is discarded otherwise.

### *Critical Section*

> *Two or more connected subgraphs of a process model are identified as "critical sections". At runtime for a given process instance, only tasks in one of these "critical sections" can be active at any given time. Once execution of the tasks in one "critical section" commences, it must complete before another "critical section" can commence.*

```
00 public class CriticalSection : ProcessBase
01 {
02     // BindFlow is cooperatively multitasking, so this is safe
03     bool mutex = false;
04
05     // Mandatory main entry point
06     public override IEnumerable<IFavor> Start(object data)
07     {
08         yield return AsyncCall(Other);
09
10         yield return AssignTask("Advance into Main critical section",
11             Initiator);
12
13         if (mutex) throw new Exception("Cannot enter Main critical " +
14             "section at this time");
15
16         mutex = true;
17
18         yield return AssignTask("Complete Main critical section",
19             Initiator);
20
21         mutex = false;
22     }
23
24     IEnumerable<IFavor> Other()
25     {
26         yield return AssignTask("Advance into Other critical section",
27             Initiator);
28
29         if (mutex) throw new Exception("Cannot enter Other critical " +
30             "section at this time");
31
32         mutex = true;
33
34         yield return AssignTask("Complete Other critical section",
35             Initiator);
36
37         mutex = false;
38     }
39 }
```

**Code Listing 34 – Critical Section Implementation Example**

The **Critical Section** pattern can be implemented with a mutex flag that blocks executing continuations into a locked critical section. Code Listing 34 throws an `System.Exception` back to the *session* initiator if the critical section is locked. The engine dumps a *session* upon catching an exception. So long as no side-effects were generated prior to the exception, there are no consequences to the failure and entering the critical section can be attempted again at a later time, perhaps after the critical section has been released.

### *Interleaved Routing*

> *Each member of a set of tasks must be executed once. They can be executed in any order but no two tasks can be executed at the same time (i.e. no two tasks can be active for the same process instance at the same time). Once all of the tasks have completed, the next task in the process can be initiated.*

```
00 public class InterleavedRouting : ProcessBase
01 {
02     public override IEnumerable<IFavor> Start(object data)
03     {
04         List<IAssignTask> tasks = new List<IAssignTask>();
05 
06         for (int x = 0; x < 4; x++)
07         {
08             tasks.Add(AssignTask("The " + x + "th created task",
09                 Initiator));
10         }
11 
12         while (tasks.Any())
13         {
14             var randomTask = IO.IORandom(tasks.Count - 1);
15             yield return randomTask;
16 
17             var task = tasks[randomTask.Result];
18 
19             tasks.Remove(task);
20 
21             yield return task;
22         }
23     }
24 }
```

**Code Listing 35 - Interleaved Routing Implementation Example**

The **Interleaved Routing** pattern can be implemented with a set of *tasks* and some selector. Code Listing 35 uses an `IO.IORandom` *favor* to request a randomly generated number. The selected *task* is performed and removed from the set of pending *tasks*.

## Cancellation and Force Completion

### *Cancel Task*

> *An enabled task is withdrawn prior to it commencing execution. If the task has started, it is disabled and, where possible, the currently running instance is halted and removed.*

```
00 public class CancelTask : ProcessBase
```

```
01 {
02     public override IEnumerable<IFavor> Start(object data)
03     {
04         var other = AsyncCall(Other);
05 
06         yield return other;
07 
08         yield return AssignTask("Retract Other Assignment", Initiator);
09 
10         yield return Cancel(other);
11     }
12 
13     IEnumerable<IFavor> Other()
14     {
15         yield return AssignTask("Finish Other", Initiator);
16     }
17 }
```

**Code Listing 36 – Cancel Task Implementation Example**

The **Cancel Task** pattern can be implemented with a cancellation of a particular *branch* of execution. The meaning of "Task" in "Cancel Task" represents a terminology conflict with van der Aalst et al., and should be read as "branch of execution". Code Listing 36 performs a **Parallel Split** to `Other` and then, after completing the `"Retract Other Assignment" AssignTask` of Line 08, prematurely terminates the `Other` *branch*, if it wasn't completed first.

It is also possible to cancel an `AssignTask` indirectly by canceling its containing *branch*. If only one assignment should be canceled without canceling the entire *branch*, consider refactoring the `AssignTask` to its own *sequence*.

### *Cancel Case*

> *A complete process instance is removed. This includes currently executing tasks, those which may execute at some future time and all sub-processes. The process instance is recorded as having completed unsuccessfully.*

```
00 public class CancelCase : ProcessBase
01 {
02     public override IEnumerable<IFavor> Start(object data)
03     {
04         var task = AssignTask("Spawn a child instance?",
05             Initiator);
06 
07         yield return task;
08 
09         if ((bool)task.Result)
10         {
11             var spawn = Spawn("WorkflowPatterns.ControlFlow.CancelCase",
12                 false);
13 
14             yield return spawn;
15 
16             var cancelation = AssignTask("Cancel all child instances?",
17                 Initiator);
```

```
18
19              yield return cancelation;
20
21              if ((bool)cancelation.Result)
22                  yield return Cancel(spawn);
23
24              yield return Wait(spawn);
25          }
26      }
27  }
```

**Code Listing 37 – Cancel Case Implementation Example**

The **Cancel Case** pattern can be implemented with a cancellation of a new *instance* spawned by the parent *instance*. Code Listing 37 optionally spawns a new *instance* of itself recursively. The resulting spawned *instances* may grow to be many levels deep, depending on the user's choices. If any parent is told to cancel all child *instances* through the `AssignTask` created on Line 16, descendant *instances* are canceled recursively.

### *Cancel Region*

> *The ability to disable a set of tasks in a process instance. If any of the tasks are already executing (or are currently enabled), then they are withdrawn. The tasks need not be a connected subset of the overall process model.*

```
00 public class CancelRegion : ProcessBase
01 {
02      public override IEnumerable<IFavor> Start(object data)
03      {
04          var conditional = AssignTask("Split to a new branch?",
05              Initiator);
06
07          yield return conditional;
08
09          if ((bool)conditional.Result)
10          {
11              var split = AsyncCall(Start, (object)null);
12
13              yield return split;
14
15              var cancelation = AssignTask("Cancel all child branches?",
16                  Initiator);
17
18              yield return cancelation;
19
20              if ((bool)cancelation.Result)
21                  yield return Cancel(split);
22          }
23      }
24  }
```

**Code Listing 38 – Cancel Region Implementation Example**

The **Cancel Region** pattern can be implemented with a cancelation of some *branch* of an *instance*. Code Listing 38 optionally creates new *branches* of the `Start` *sequence* recursively. The resulting split *branches*

may grow to be many levels deep, depending on the user's choices. If any parent *branch* is told to cancel all child *branches* through the `AssignTask` created on Line 15, descendant *branches* are canceled recursively.

### *Cancel Multiple Instance Activity*

> *Within a given process instance, multiple instances of a task 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 task can be cancelled and any instances which have not completed are withdrawn. Task instances that have already completed are unaffected.*

```
00 public class CancelMultipleInstanceActivity : ProcessBase
01 {
02     public override IEnumerable<IFavor> Start(object data)
03     {
04         var multiTask = AsyncCall(MultiTask);
05
06         yield return multiTask;
07
08         yield return AssignTask("Advance to cancelation", Initiator);
09
10         var cancel = Cancel(multiTask);
11
12         yield return cancel;
13     }
14
15     IEnumerable<IFavor> MultiTask()
16     {
17         // Create the assignments
18         var assignment1 = AsyncCall(Assignment, 1);
19         var assignment2 = AsyncCall(Assignment, 2);
20         var assignment3 = AsyncCall(Assignment, 3);
21
22         // Emit the assignments
23         yield return assignment1;
24         yield return assignment2;
25         yield return assignment3;
26
27         // Wait for all of the assignments to be completed (in any order)
28         yield return Wait(assignment1);
29         yield return Wait(assignment2);
30         yield return Wait(assignment3);
31     }
32
33     // Expects an int as data
34     IEnumerable<IFavor> Assignment(int assignmentNumber)
35     {
36         yield return AssignTask("Advance #" + assignmentNumber, Initiator);
37     }
38 }
```

**Code Listing 39 - Cancel Multiple Instance Activity Implementation Example**

The **Cancel Multiple Instance Activity** pattern can be implemented by canceling multiple *branches* of the same *sequence*. Code Listing 39 **Splits** to multiple concurrent and independent *branches* of `Assignment` within `MultiTask`. When the `MultiTask` *branch* is canceled, any unfinished `Assignments` are also canceled but the results of finished `Assignments` are unaffected.

### *Complete Multiple Instance Activity*

> *Within a given process instance, multiple instances of a task can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently. It is necessary to synchronize the instances at completion before any subsequent tasks can be triggered. During the course of execution, it is possible that the task needs to be forcibly completed such that any remaining instances are withdrawn and the thread of control is passed to subsequent tasks.*

```
00 public class CompleteMultipleInstanceActivity : ProcessBase
01 {
02     public override IEnumerable<IFavor> Start(object data)
03     {
04         var multiTask = AsyncCall(MultiTask);
05 
06         yield return multiTask;
07 
08         yield return AssignTask("Force completion", Initiator);
09 
10         var cancel = Cancel(multiTask);
11 
12         yield return cancel;
13 
14         yield return AssignTask("Finish", Initiator);
15     }
16 
17     IEnumerable<IFavor> MultiTask()
18     {
19         // Create the assignments
20         var assignment1 = AsyncCall(Assignment, 1);
21         var assignment2 = AsyncCall(Assignment, 2);
22         var assignment3 = AsyncCall(Assignment, 3);
23 
24         // Emit the assignments
25         yield return assignment1;
26         yield return assignment2;
27         yield return assignment3;
28 
29         // Wait for all of the assignments to be completed (in any order)
30         yield return Wait(assignment1);
31         yield return Wait(assignment2);
32         yield return Wait(assignment3);
33     }
34 
35     // Expects an int as data
36     IEnumerable<IFavor> Assignment(int assignmentNumber)
37     {
38         yield return AssignTask("Advance #" + assignmentNumber, Initiator);
39     }
```

```
40 }
```
Code Listing 40 - Complete Multiple Instance Activity Implementation Example

The **Complete Multiple Instance Activity** pattern can be implemented as a trivial extension of the **Cancel Multiple Instance Activity** pattern. Code Listing 40 extends Code Listing 39 with an explicit continuation on Line 14 after the cancellation of `MultiTask`.

## Iteration

### *Arbitrary Cycles*

> *The ability to represent cycles in a process model that have more than one entry or exit point. It must be possible for individual entry and exit points to be associated with distinct branches.*

```
00 public class ArbitraryCycles : ProcessBase
01 {
02     public override IEnumerable<IFavor> Start(object data)
03     {
04     Step1:
05         // First Task
06         var task1 = AssignTask("Step 1: Jump to Step3?", Initiator);
07         yield return task1;
08         if ((bool)task1.Result) goto Step3;
09
10     Step2:
11         // Second Task
12         var task2 = AssignTask("Step 2: Jump to Step1?", Initiator);
13         yield return task2;
14         if ((bool)task2.Result) goto Step1;
15
16     Step3:
17         // Third Task
18         var task3 = AssignTask("Step 3: Jump to Step2?", Initiator);
19         yield return task3;
20         if ((bool)task3.Result) goto Step2;
21     }
22 }
```
Code Listing 41 - Arbitrary Cycles Implementation Example

The **Arbitrary Cycles** pattern can be implemented using labels and C# `goto` statements. Code Listing 41 allows each section of code to conditionally `Jump` to another step in an unstructured fashion as illustrated by the demonstration for this pattern by van der Aalst et al.[1].

### *Structured Loop*

> *The ability to execute a task or sub-process repeatedly. The loop has either a pre-test or post-test condition associated with it that is either evaluated at the beginning or end of the loop to determine whether it should continue. The looping structure has a single entry and exit point.*

```
00 public class StructuredLoop : ProcessBase
01 {
```

```
02      public override IEnumerable<IFavor> Start(object data)
03      {
04          IAssignTask task;
05
06          do
07          {
08              task = AssignTask("Repeat?", Initiator);
09
10              yield return task;
11          } while ((bool)task.Result);
12      }
13  }
```

**Code Listing 42 – Structured Loop Implementation Example**

The **Structured Loop** pattern can be implemented with any standard C# looping construct such as `while` or `for` loops. Code Listing 42 uses a `do-while` loop conditional on the result of an assigned *task*.

### *Recursion*

> *The ability of a task to invoke itself during its execution or an ancestor in terms of the overall decomposition structure with which it is associated.*

```
00 public class Recursion : ProcessBase
01 {
02      public override IEnumerable<IFavor> Start(object data)
03      {
04          var conditional = AssignTask("Call to a new branch?", Initiator);
05
06          yield return conditional;
07
08          if ((bool)conditional.Result)
09          {
10              yield return Call(Start, (object)null);
11          }
12      }
13  }
```

**Code Listing 43 – Recursion Implementation Example**

The **Recursion** pattern can be implemented with a `Call` to a new *branch* of the calling *sequence*. Code Listing 43 calls back onto `Start` optionally. Emitting the `Call` pops the calling *sequence* from the stack and moves it to the wait-list and then pushes a new *branch* of the called *sequence* on to the stack for immediate execution. Parent *branches* remain in the wait-list until the called *branch* completes.

## Termination

### *Implicit Termination*

> *A given process (or sub-process) instance should terminate when there are no remaining work items that are able to be done either now or at any time in the future and the process instance is not in deadlock. There is an objective means of determining that the process instance has successfully completed.*

```
0 public class ImplicitTermination : ProcessBase
```

```
1 {
2     public override IEnumerable<IFavor> Start(object data)
3     {
4         yield return AssignTask("Finish", Initiator);
5     }
6 }
```

**Code Listing 44 – Implicit Termination Implementation Example**

The **Implicit Termination** pattern is implemented by abstaining from providing more instructions for a *process* to follow. When no work remains, the *instance* is considered terminated. In Code Listing 44, following Line 4, there are no remaining instructions to follow and nothing left to do.

### *Explicit Termination*

> *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 instance is cancelled and the overall process instance is recorded as having completed successfully, regardless of whether there are any tasks in progress or remaining to be executed.*

```
0 public class ExplicitTermination : ProcessBase
1 {
2     public override IEnumerable<IFavor> Start(object data)
3     {
4         yield return ForceTerminate();
5     }
6 }
```

**Code Listing 45 – Explicit Termination Implementation Example**

The **Explicit Termination** pattern can be implemented with `ForceTerminate`. `ForceTerminate` instructs the engine to immediately dump all state, including the execution of other *branches*, and mark the *instance* as finished. Code Listing 45 is a simple such implementation.

## Trigger

### *Transient Trigger*

> *The ability for a task instance 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 task. A trigger can only be utilized if there is a task instance waiting for it at the time it is received.*

```
00 public class TransientTrigger : ProcessBase
01 {
02     ISubscribe externalTrigger;
03
04     public override IEnumerable<IFavor> Start(object data)
05     {
06         externalTrigger = Subscribe("TRIGGER", null, EventHandler);
07
08         yield return externalTrigger;
09
```

```
10          IAssignTask task;
11
12          do
13          {
14              task = AssignTask("Internally trigger", Initiator, true);
15
16              yield return task;
17
18              if ((bool)task.Result)
19                  yield return AsyncCall(EventHandler, (object)null);
20          } while ((bool)task.Result);
21
22          yield return Unsubscribe(externalTrigger);
23          externalTrigger = null;
24      }
25
26      IEnumerable<IFavor> EventHandler(object data)
27      {
28          if (externalTrigger != null)
29              yield return AssignTask("Triggered!", Initiator);
30      }
31  }
```

**Code Listing 46 – Transient Trigger Implementation Example**

The **Transient Trigger** pattern can be implemented with a *subscription* and `AsyncCalls` for internal calls. **Code Listing 46** opens a *subscription* and an event loop for both external and internal calls to `EventHandler`. Calls are only processed while the *subscription* is open (and for the benefit of synchronizing the internal calls, when `externalTrigger` is not `null`).

### *Persistent Trigger*

> *The ability for a task to be triggered by a signal from another part of the process or from the external environment. These triggers are persistent in form and are retained by the process until they can be acted on by the receiving task.*

```
00  public class PersistentTrigger : ProcessBase
01  {
02      int count;
03      ISubscribe externalTrigger;
04
05      public override IEnumerable<IFavor> Start(object data)
06      {
07          externalTrigger = Subscribe("TRIGGER", null, EventHandler);
08
09          yield return externalTrigger;
10
11          IAssignTask task = null;
12
13          do
14          {
15              task = AssignTask("Internally trigger", Initiator, true);
16
17              yield return task;
18
19              if ((bool)task.Result)
```

```
20                  yield return AsyncCall(EventHandler, (object)null);
21          } while ((bool)task.Result);
22
23          yield return Unsubscribe(externalTrigger);
24          externalTrigger = null;
25
26          yield return AssignTask("Advance", Initiator);
27
28          for (int i = 0; i < count; i++)
29          {
30              yield return AssignTask("Handle deferred event", Initiator);
31          }
32      }
33
34      IEnumerable<IFavor> EventHandler(object data)
35      {
36          if (externalTrigger != null)
37              count++;
38
39          yield break; // Required by C# compiler
40      }
41  }
```

**Code Listing 47 – Persistent Trigger Implementation Example**

The **Persistent Trigger** pattern can be implemented with a *subscription* and a trigger queuing mechanism. Code Listing 47 opens a *subscription* to `EventHandler` as in Code Listing 46 but queues up triggers with a counter, `count`. The internal call loop also serves to keep the *subscription* open. Once the *subscription* is closed, the deferred queue of work (`count` occurrences, in this example) can be processed in *sequence*.

## Data Patterns

BindFlow supports 34 of the 40 Data Patterns.

### Data Visibility

#### *Task Data*

> *Data elements can be defined by tasks which are accessible only within the context of individual execution instances of that task.*

The **Task Data** pattern can be implemented with local variables which are scoped only to a particular *task*, such as at the C# method level or code block level. Code Listing 10 uses this pattern to track references to child *branches* of `One` and `Two` – data which is not available by other areas of the workflow. Ever narrower scoping can be achieved with C# block scoping.

#### *Block Data*

> *Block tasks (i.e. tasks which can be described in terms of a corresponding subprocess) are able to define data elements which are accessible by each of the components of the corresponding subprocess.*

The **Block Data** pattern can be implemented by passing the data by value first to a *branch* of a *sequence*. The Value variation, of the variations described by van der Aalst et al.[1], is achieved by passing values to *sequences* as parameters as in Code Listing 16. The Global Reference and Reference variations are achieved by passing context objects to *sequences* as parameters as in Code Listing 20.

### *Scope Data*

> *Data elements can be defined which are accessible by a subset of the tasks in a case.*

The **Scope Data** pattern can be implemented by passing (push) or referencing (pull) data selectively within *tasks*.

### *Multiple Instance Data*

> *Tasks which are able to execute multiple times within a single case can define data elements which are specific to an individual execution instance.*

The **Multiple Instance Data** pattern can be implemented by *branching* to a single *sequence* multiple times at runtime, as in Code Listing 16. Each *branch* maintains a private state.

### *Case Data*

> *Data elements can be defined by tasks which are accessible only within the context of individual execution instances of that task.*

The **Case Data** pattern can be implemented with data fields at the class-level (*process*-global) as in Code Listing 24.

### *Folder Data*

> *Data elements can be defined which are accessible by multiple cases on a selective basis. They are accessible to all components of the cases to which they are bound.*

The **Folder Data** pattern is not natively supported.

### *Workflow Data*

> *Data elements are supported which are accessible to all components in each and every case of the process and are within the context of the process itself.*

**Workflow Data** is provided by the `ConfigVariableAttribute`, configuration variables, and the `IO.IOGetConfigValue` *favor*. Configured data is read-only from within a *process*.

### *Environment Data*

> *Data elements which exist in the external operating environment are able to be accessed by components of processes during execution.*

**Environment data** from any source is accessible through custom *IO*. Within the `IO.Perform` method, any serializable data can be written to or read from the real world. Written data could be from or influenced by any data stored in the `IO` object during construction within the *instance*. Read data is passed back to the *instance* through the return value of the `Perform` method. The engine, which called the `Perform` method, writes any result to the *instance's* log before setting the `IO's Result` property and resuming the *instance*. In subsequent *sessions*, the engine places the stored result in the `IO's Result` property, bypassing the `Perform` method.

### Internal Data Interaction

#### *Task to Task*

> *The ability to communicate data elements between one task instance and another within the same case. The communication of data elements between two tasks is specified in a form that is independent of the task definitions themselves.*

The **Task to Task** data pattern can be implemented as the Integrated Control and Data Channel (passing data to *sequences* through parameters) or the Global Data Store (*instance*-level data) variations of van der Aalst et al.[1].

#### *Block Task to Sub-Workflow Decomposition*

> *The ability to pass data elements from a block task instance to the corresponding subprocess that defines its implementation. Any data elements that are available to a block task are able to be passed to (or be accessed) in the associated subprocess although only a specifically nominated subset of those data elements are actually passed to the subprocess.*

The **Block Task to Sub-Workflow Decomposition** pattern is supported by the Explicit data passing via parameters by van der Aalst et al.[1].

#### *Sub-Workflow Decomposition to Block Task*

> *The ability to pass data elements from the underlying subprocess back to the corresponding block task. Only nominated data elements defined as part of the subprocess are made available to the (parent) block task.*

The **Sub-Workflow Decomposition to Block Task** pattern is supported by the `Return` *favor*. The value passed to the host as part of the `Return` *favor* is copied to any `Call`, `AsyncCall`, or `Waits` relevant to the terminated *branch*.

#### *To Multiple Instance Task*

> *The ability to pass data elements from a preceding task instance to a subsequent task which is able to support multiple execution instances. This may involve passing the data elements to all instances of the multiple instance task or distributing them on a selective basis. The data passing occurs when the multiple instance task is enabled.*

The **To Multiple Instance Task** pattern is supported by the Shared Data Passed by Reference (passing data by reference as *sequence* parameters), Instance Specific Data Passed by Value (passing data by value as *sequence* parameters), and Instance Specific Data Passed by Reference (passing *instance* specific objects by reference as *sequence* parameters) variations by van der Aalst et al.[1].

### *From Multiple Instance Task*

> *The ability to pass data elements from a task which supports multiple execution instances to a subsequent task. The data passing occurs at the conclusion of the multiple instance task. It involves aggregating data elements from all instances of the task and passing them to a subsequent task.*

The **From Multiple Instance Task** pattern is supported by through the `Return` *favor* or by modifying some commonly referenced context object. The **Join** construct may perform the aggregation or the context object may on-write or on-read aggregation functionality.

### *Case to Case*

> *The passing of data elements from one case of a process during its execution to another case that is executing concurrently.*

The **Case to Case** pattern is supported through the `IO.IOCompleteTask` and the `IO.IONotifySubscriber` *favors* which, as the names suggest, complete the *tasks* or notify on the *subscriptions* of other *instances*.

### External Data Interaction

> *The ability of a task to initiate the passing of data elements to a resource or service in the operating environment.*

Environment data from any source is accessible through custom *IO*. Within the `IO.Perform` method, any serializable data can be written to or read from the real world. Written data could be from or influenced by any data stored in the `IO` object during construction within the *instance*. Read data is passed back to the *instance* through the return value of the `Perform` method. The engine, which called the `Perform` method, writes any result to the *instance's* log before setting the `IO's Result` property and resuming the *instance*. In subsequent *sessions*, the engine places the stored result in the `IO's Result` property, bypassing the `Perform` method.

### *Task to Environment - Push-Oriented*

> *The ability of a task to initiate the passing of data elements to a resource or service in the operating environment.*

The **Task to Environment** pattern is supported through `AssignTask`, `Subscribe`, and any custom *IO*, as previously described.

### *Environment to Task - Pull-Oriented*

> *The ability of a task to request data elements from resources or services in the operational environment.*

The **Environment to Task - Pull-Oriented** pattern is supported through any custom *IO*, as previously described.

### *Environment to Task - Push-Oriented*

> *The ability for a task to receive and utilise data elements passed to it from services and resources in the operating environment on an unscheduled basis.*

The **Environment to Task – Push-Oriented** pattern is supported through `AssignTask`.

### *Task to Environment - Pull-Oriented*

> *The ability of a task to receive and respond to requests for data elements from services and resources in the operational environment.*

The **Task to Environment - Pull-Oriented** pattern is not supported.

### *Case to Environment - Push-Oriented*

> *The ability of a case to initiate the passing of data elements to a resource or service in the operational environment.*

The **Case to Environment** pattern is supported through `AssignTask`, `Subscribe`, and any custom *IO*, as previously described. Here, the *instance* (*case*) is understood to necessarily include its component parts – a *process* can be designed with *branches* of execution that act on behalf of the *instance* in a singleton fashion.

### *Environment to Case - Pull-Oriented*

> *The ability of a case to request data from services or resources in the operational environment.*

The **Environment to Case - Pull-Oriented** pattern is supported through any custom *IO*, as previously described.

### *Environment to Case - Push-Oriented*

> *The ability of a case to accept data elements passed to it from services or resources in the operating environment.*

The **Environment to Case - Push-Oriented** pattern is supported through `AssignTask` and `Subscribe`.

### *Case to Environment - Pull-Oriented*

> *The ability of a case to respond to requests for data elements from a service or resource in the operating environment.*

The **Case to Environment - Pull-Oriented** pattern is supported through the selected data published to the environment through an *instance's* `Summarize` method and it's collection of **Milestones**.

### *Workflow to Environment - Push-Oriented*

> *The ability of a process environment to pass data elements to resources or services in the operational environment.*

The **Workflow to Environment – Push-Oriented** pattern is not supported.

### *Environment to Workflow - Pull-Oriented*

> *The ability of a process environment to request global data elements from external applications.*

The **Environment to Workflow - Pull-Oriented** pattern is not supported.

### *Environment to Workflow - Push-Oriented*

> *The ability of services or resources in the operating environment to pass global data to a process.*

The **Environment to Workflow - Push-Oriented** pattern is supported through the configuration value mechanism, either through a management console or through the APIs programmatically.

### *Workflow to Environment - Pull-Oriented*

> *The ability of the process environment to handle requests for global data from external applications.*

The **Workflow to Environment - Pull-Oriented** pattern is supported through the configuration value mechanism, either through a management console or through the APIs programmatically.

## Data Transfer Patterns

### *By Value - Incoming*

> *The ability of a process component to receive incoming data elements by value avoiding the need to have shared names or common address space with the component(s) from which it receives them.*

The **By Value - Incoming** pattern is supported through the serialization of data objects inbound through new *instances*, completed *tasks*, notified *subscriptions*, or *IO*.

### *By Value - Outgoing*

> *The ability of a process component to pass data elements to subsequent components as values avoiding the need to have shared names or common address space with the component(s) to which it is passing them.*

The **By Value - Outgoing** pattern is supported through the serialization of data objects outbound through `AssignTasks`, `Subscribes`, *IO*, the optional `ProcessBase.Summarize` overload or a set of `ProcessBase.Result`.

### *Copy In/Copy Out*

> *The ability of a process component to copy the values of a set of data elements from an external source (either within or outside the process environment) into its address space at the commencement of execution and to copy their final values back at completion.*

The **Copy In/Copy Out** pattern is supported through the serialization of data objects inbound or outbound, as previously discussed.

### *By Reference - Unlocked*

> *The ability to communicate data elements between process components by utilizing a reference to the location of the data element in some mutually accessible location. No concurrency restrictions apply to the shared data element.*

The **By Reference - Unlocked** pattern is supported by passing a pointer, such as a database key value, as data by value. *IO* must be employed to access the referenced data.

### *By Reference - With Lock*

> *The ability to communicate data elements between process components by passing a reference to the location of the data element in some mutually accessible location. Concurrency restrictions are implied with the receiving component receiving the privilege of read-only or dedicated access to the data element. The required lock is declaratively specified as part of the data passing request.*

The **By Reference - With Lock** is not supported natively, however, a locking mechanism at the application level is practical.

### *Data Transformation - Input*

> *The ability to apply a transformation function to a data element prior to it being passed to a process component. The transformation function has access to the same data elements as the receiving process component.*

The **Data Transformation - Input** pattern is supported by leveraging the standard C# data transformation capabilities within a *process*.

### *Data Transformation - Output*

> *The ability to apply a transformation function to a data element immediately prior to it being passed out of a process component. The transformation function has access to the same data elements as the process component that initiates it.*

The **Data Transformation - Output** pattern is supported by leveraging the standard C# data transformation capabilities within a *process*.

## Data-based Routing

### *Task Precondition - Data Existence*

> *Data-based preconditions can be specified for tasks based on the presence of data elements at the time of execution. The preconditions can utilize any data elements available to the task with which they are associated. A task can only proceed if the associated precondition evaluates positively.*

The **Task Precondition – Data Existence** pattern is supported through typical programming data existence test such as testing for `null`.

### *Task Precondition - Data Value*

> *Data-based preconditions can be specified for tasks based on the value of specific parameters at the time of execution. The preconditions can utilize any data elements available to the task with which they are associated. A task can only proceed if the associated precondition evaluates positively.*

The **Task Precondition – Data Value** pattern is supported through arbitrary data validations written in C# within a *process*.

### *Task Postcondition - Data Existence*

> *Data-based postconditions can be specified for tasks based on the existence of specific parameters at the time of task completion. The postconditions can utilize any data elements available to the task with which they are associated. A task can only proceed if the associated postcondition evaluates positively.*

The **Task Postcondition – Data Existence** pattern is supported through typical programming data existence test such as testing for `null`. Postcondition failure may involve throwing a .Net exception or some alternative code path.

### *Task Postcondition - Data Value*

> *Data-based postconditions can be specified for tasks based on the value of specific parameters at the time of execution. The postconditions can utilize any data elements available to the task with which they are associated. A task can only proceed if the associated postcondition evaluates positively.*

The **Task Postcondition – Data Value** pattern is supported through arbitrary data validations written in C# within a *process*. Postcondition failure may involve throwing a .Net exception or some alternative code path.

### *Event-based Task Trigger*

> *The ability for an external event to initiate a task and to pass data elements to it.*

The **Event-based Task Trigger** pattern is supported through *tasks* and *subscriptions*.

### *Data-based Task Trigger*

> *Data-based task triggers provide the ability to trigger a specific task when an expression based on data elements in the process instance evaluates to true. Any data element accessible within a process instance can be used as part of a data-based trigger expression.*

The **Data-based Task Trigger** pattern is not supported.

### *Data-based Routing*

> *Data-based routing provides the ability to alter the control-flow within a case based on the evaluation of data-based expressions. A data-based routing expression is associated with each outgoing arc of an OR-split or XOR-split. It can be composed of any data-values, expressions and functions available in the process environment providing it can be evaluated at the time the split construct with which it is associated completes. Depending on whether the construct is an XOR-split or OR-split, a mechanism is available to select one or several outgoing arcs to which the thread of control should be passed based on the evaluation of the expressions associated with the arcs.*

The **Data-based Routing** pattern is supported by selecting one or more routes through standard C# control-flow mechanisms.

## Resource Patterns

`AssignTask` assignments in the code examples above use the convenient `ProcessBase.Initiator` to assign the task back to the initiator of the *process*. Assigning to other accounts requires only knowing the name of the account to assign with the `Account` constructor `new Account("TAG", "ACCOUNT_NAME")` `where "TAG"` is the provider prefix for a configured user manager, such as Active Directory, and `"ACCOUNT_NAME"` is the unique identifier of a user or group within that user manager's scope. In BindFlow, the provider prefix can be omitted, defaulting to Active Directory. `new Account("[TAG]ACCOUNT_NAME")` formatting is also supported.

Furthermore, role-membership, permissions, and organizational hierarchy queries against Active Directory or other user managers is available through *IO*. The built-in `IOActiveDirectory` *IO* factory implements several user and group queries.

While most of the Control-Flow and Data patterns are supported from within *process* code, the BindFlow Development Kit, many of the Resource patterns are realized through the external Client API.

BindFlow fully-supports 39 of the 43 Resource Patterns and at least partially supports all 43 Resource Patterns.

### Creation

#### *Direct Distribution*

> *The ability to specify at design time the identity of the resource(s) to which instances of this task will be distributed at runtime.*

The **Direct Distribution** pattern is supported by assigning a task to a particular user within a configured user manager's scope.

#### *Role-Based Distribution*

> *The ability to specify at design-time one or more roles to which instances of this task will be distributed at runtime. Roles serve as a means of grouping resources with similar characteristics. Where an instance of a task is distributed in this way, it is distributed to all resources that are members of the role(s) associated with the task.*

The **Role-Based Distribution** pattern is supported to a particular group within a configured user manager's scope.

#### *Deferred Distribution*

> *The ability to specify at design-time that the identification of the resource(s) to which instances of this task will be distributed will be deferred until runtime.*

The **Deferred Distribution** pattern is supported by using variable data to create the `Account` object at runtime.

#### *Authorization*

> *The ability to specify the range of privileges that a resource possesses in regard to the execution of a process. In the main, these privileges define the range of actions that a resource can initiate when undertaking work items associated with tasks in a process.*

The **Authorization** pattern is supported by querying user managers through *IO* or by embedding some scheme of authorization tables within a *process*.

#### *Separation of Duties*

> *The ability to specify that two tasks must be executed by different resources in a given case.*

The **Separation of Duties** pattern is supported by enforcing such rules in *process* code.

### *Case Handling*

> *The ability to allocate the work items within a given case to the same resource at the time that the case is commenced.*

The **Case Handling** pattern is supported by enforcing such rules in *process* code.

### *Retain Familiar*

> *Where several resources are available to undertake a work item, the ability to allocate a work item within a given case to the same resource that undertook a preceding work item.*

The **Retain Familiar** pattern is supported by enforcing such rules in *process* code. Familiarity among cases is not natively supported, but can be implemented with an application-level external data store.

### *Capability-Based Distribution*

> *The ability to distribute work items to resources based on specific capabilities that they possess. Capabilities (and their associated values) are recorded for individual resources as part of the organizational model.*

The **Capability-Based** pattern is supported by querying user managers through *IO* or by embedding some scheme of capability tables within a *process*.

### *History-Based Distribution*

> *The ability to distribute work items to resources on the basis of their previous execution history.*

The **History-Based** pattern is supported by enforcing such rules in *process* code. History among cases is not natively supported.

### *Organizational Distribution*

> *The ability to distribute work items to resources based their position within the organization and their relationship with other resources.*

The **Organizational Distribution** pattern is supported by querying some organizational hierarchy through *IO* or by embedding it within a *process*.

### *Automatic Execution*

> *The ability for an instance of a task to execute without needing to utilize the services of a resource.*

The **Automatic Execution** pattern is supported as many steps in *process* code and *IO* do not require human intervention. Furthermore, a built-in system timer can be used to delay automated execution until some statically or dynamically chosen future date.

### Push

#### *Single Distribution by Offer*

> *The ability to distribute a work item to a selected individual resource on a non-binding basis.*

The **Single Distribution by Offer** pattern is supported. An assigned *task* may be delegated or reassigned as allowed by system permissions.

#### *Multiple Distribution by Offer*

> *The ability to distribute a work item to a group of selected resources on a non-binding basis.*

The **Multiple Distribution by Offer** pattern is supported by assigning work to an `Account` representing a group.

#### *Single Distribution by Allocation*

> *The ability to distribute a work item to a specific resource for execution on a binding basis.*

The **Single Distribution by Allocation** pattern is supported. An assigned *task* may be delegated or reassigned as allowed by system permissions.

#### *Random Allocation*

> *The ability to allocate work items to a selected resource chosen from a group of eligible resources on a random basis.*

The **Random Allocation** pattern is supported through the use of `IO.IORandom` or similar *IO*.

#### *Round Robin Allocation*

> *The ability to allocate a work item to a selected resource chosen from a group of eligible resources on a cyclic basis.*

The **Round Robin Allocation** within a single *instance* is supported. **Round Robin Allocation** among *instances* is not natively supported.

#### *Shortest Queue*

> *The ability to allocate a work item to a selected resource chosen from a group of eligible resources on the basis of having the shortest work queue.*

The **Shortest Queue** pattern within a single *instance* is supported. The **Shortest Queue** pattern among *instances* is not natively supported.

### *Early Distribution*

> *The ability to advertise and potentially distribute a work items to resources ahead of the moment at which it is actually enabled.*

The **Early Distribution** pattern is supported. The allocation of a *task* prior to its enablement may be achieved in many ways such as sending an advisory email or assigning a *task* early with a low priority setting and later upgrading the priority at intended enablement.

### *Distribution on Enablement*

> *The ability to advertise and distribute a work items to resources at the moment that the task to which it corresponds is enabled for execution.*

The **Distribution on Enablement** pattern is supported as the default.

### *Late Distribution*

> *The ability to advertise and distribute work items to resources after the task to which the work item corresponds has been enabled for execution.*

The **Late Distribution** pattern is supported. An email reminder or upgrade of *task* priority are two possible approaches.

## Pull

### *Resource-Initiated Allocation*

> *The ability for a resource to commit to undertake a work item without needing to commence working on it immediately.*

The **Resource-Initiated Allocation** pattern is supported. Work assigned to one or more assignees may be Claimed by one of them.

### *Resource-Initiated Execution - Allocated Work Item*

> *The ability for a resource to commence work on a work item that is allocated to it.*

The **Resource-Initiated Execution – Allocated Work Item** pattern is supported.

### *Resource-Initiated Execution - Offered Work Item*

> *The ability for a resource to select a work item offered to it and commence work on it immediately.*

The **Resource-Initiated Execution - Offered Work Item** pattern is supported. A *task* opened by a user will be immediately claimed.

### *System-Determined Work Queue Content*

> *The ability of the system to order the content and sequence in which work items are presented to a resource for execution.*

The **System-Determined Work Queue Content** pattern is supported. *Tasks* are delivered to the user sorted by some default ordering, such as by *task* age.

### *Resource-Determined Work Queue Content*

> *The ability for resources to specify the format and content of work items listed in the work queue for execution.*

The **Resource-Determined Work Queue Content** pattern is supported. *Tasks* are delivered to the user sorted by some default ordering, such as by *task* age; however, the resource may choose other presentation formats, if allowed by configuration.

### *Selection Autonomy*

> *The ability for resources to select a work item for execution based on its characteristics and their own preferences.*

The **Selection Autonomy** pattern is supported. By default, resources are presented with a list of active *tasks* from which they are free to select.

## Detour

### *Delegation*

> *The ability for a resource to allocate an unstarted work item previously allocated to it (but not yet commenced) to another resource.*

The **Delegation** pattern is supported. In BindFlow, it is called “Reassign” to differentiate it from a similar feature called “Delegation” which allows the original assignee to maintain ownership of the item and recall it if desired.

### *Escalation*

> *The ability of a system to distribute a work item to a resource or group of resources other than those it has previously been distributed to in an attempt to expedite the completion of the work item.*

```
00 public class Escalation : ProcessBase
01 {
02     string summary;
03     public override string Summarize() { return summary; }
```

```
04
05      public override IEnumerable<IFavor> Start(object data)
06      {
07          // Do nothing for one day
08          yield return Delay(TimeSpan.FromHours(24));
09
10          var call = Call<Sequence<Account, TaskPriority>>(Escalate,
11              EscalatedStep);
12
13          yield return call;
14
15          summary = (string)call.Result;
16      }
17
18      IEnumerable<IFavor> Escalate(
19          Sequence<Account, TaskPriority> escalatedStep)
20      {
21          var escalatedStepBranch = AsyncCall(escalatedStep,
22              Initiator, TaskPriority.NORMAL);
23
24          var delay = AsyncDelay(TimeSpan.FromHours(4));
25          var wait = Wait(escalatedStepBranch, delay);
26
27          yield return escalatedStepBranch;
28          yield return delay;
29          yield return wait;
30
31          if (wait.Completed.Single() == delay)
32          {
33              yield return Cancel(escalatedStepBranch);
34
35              var call = Call(escalatedStep,
36                  new Account("Supervisors"), TaskPriority.HIGH);
37
38              yield return call;
39
40              yield return Return(call.Result);
41          }
42          else
43          {
44              yield return Cancel(delay);
45
46              yield return Return(escalatedStepBranch.Result);
47          }
48      }
49
50      IEnumerable<IFavor> EscalatedStep(
51          Account assignee, TaskPriority priority)
52      {
53          var task = AssignTask(priority, "Complete Me",
54              "http://intranet/run?task={TASK}", assignee, null);
55
56          yield return task;
57
58          yield return Return(task.Result);
59      }
60  }
```

Code Listing 48 – Escalation Implementation Example

The **Escalation** pattern is supported through a combination of `AsyncCalls`, delay timer *tasks*, and cancelations. Code Listing 48 implements the escalation of `EscalatedStep` to a Supervisors group.

Line 10 involves high-level programming features of C# Generics and C# Delegates. To demonstrate some of the advanced capabilities of the model, we generalize the `Escalate` method to accept any `Sequence<Account, TaskPriority>`. The type parameters of `Call` are often discovered by the C# type-inferencer, but in this case the C# compiler needs our assistance.

The `Escalate` *sequence* employs the **Deferred Choice** pattern between the passed in `escalatedStep` and a `Delay` timer step, triggered by the system timer. If the timer fires first, the still blocked `escalatedStep` is canceled and replaced with a new one escalated to "Supervisors" with high priority. Otherwise, the timer is canceled. In either case, the eventual completion of either of the assignments is returned back to the waiting `Start` *branch*.

### *Deallocation*

> *The ability of a resource (or group of resources) to relinquish a work item which is allocated to it (but not yet commenced) and make it available for distribution to another resource or group of resources.*

The **Deallocation** pattern is supported. The allocation or "Claim" of a work item can be revoked by "Releasing" the claim.

### *Stateful Reallocation*

> *The ability of a resource to allocate a work item that they are currently executing to another resource without loss of state data.*

The **Stateful Reallocation** pattern is supported. *Task* progress, such as filling out only some of an application's form, can be stored in BindFlow attached to the *task* without submitting it for completion. During reassignment, the reassigner has the option of retaining that saved progress.

### *Stateless Reallocation*

> *The ability for a resource to reallocate a work item that it is currently executing to another resource without retention of state.*

The **Stateless Reallocation** pattern is supported. *Task* progress, such as filling out only some of an application's form, can be stored in BindFlow attached to the *task* without submitting it for completion. During reassignment, the reassigner has the option of deleting that saved progress.

### *Suspension-Resumption*

> *The ability for a resource to suspend and resume execution of a work item.*

The **Suspension-Resumption** pattern is supported. *Task* progress, such as filling out only some of an application's form, can be stored in BindFlow attached to the *task* without submitting it for completion. It can be recalled later to continue the assignment.

### *Skip*

> *The ability for a resource to skip a work item allocated to it and mark the work item as complete.*

The **Skip** pattern behavior can be implemented by completing a *task* with some result data signaling the desire to skip the work, but is not a distinct mechanism.

### *Redo*

> *The ability for a resource to redo a work item that has previously been completed in a case. Any subsequent work items (i.e. work items that correspond to subsequent tasks in the process) must also be repeated.*

The **Redo** pattern can be implemented as a follow-up step giving a user the option of recalling any future work, but is not a distinct mechanism.

### *Pre-Do*

> *The ability for a resource to execute a work item ahead of the time that it has been offered or allocated to resources working on a given case. Only work items that do not depend on data elements from preceding work items can be "pre-done".*

The **Pre-do** pattern can be implemented by careful design of a *process* similar to the **Early Distribution Resource Allocation** pattern, but is not a distinct mechanism.

## Auto-Start

### *Commencement on Creation*

> *The ability for a resource to commence execution on a work item as soon as it is created.*

The **Commencement on Creation** pattern is supported. *Tasks* assigned to an individual rather than to a group do not need to be allocated.

### *Commencement on Allocation*

> *The ability to commence execution on a work item as soon as it is allocated to a resource.*

The **Commencement on Allocation** pattern is supported. *Tasks* assigned to a group are first allocated to an individual by claim, delegation, or reassignment before being commenced.

### *Piled Execution*

> *The ability to initiate the next instance of a task (perhaps in a different case) once the previous one has completed with all associated work items being allocated to the same resource. The transition to Piled Execution mode is at the instigation of an individual resource. Only one resource can be in Piled Execution mode for a given task at any time.*

The **Piled Execution** pattern is supported through the Client API calls within an application.

### *Chained Execution*

> *The ability to automatically start the next work item in a case once the previous one has completed. The transition to Chained Execution mode is at the instigation of the resource.*

The **Chained Execution** pattern is supported. Upon completing a *task* or notifying on a *subscription*, the engine tracks the applicable thread of execution and returns relevant follow-up *tasks* from the same thread, if available.

## Visibility

### *Configurable Unallocated Work Item Visibility*

> *The ability to configure the visibility of unallocated work items by process participants.*

The **Configurable Unallocated Work Item Visibility** pattern is supported through group membership of the allocation pool or supervisory and process-oversight permissions on the *task-list*.

### *Configurable Allocated Work Item Visibility*

> *The ability to configure the visibility of allocated work items by process participants.*

The **Configurable Allocated Work Item Visibility** pattern is supported through supervisory and process-oversight permissions on the *task-list*.

## Multiple Resource

### *Simultaneous Execution*

> *The ability for a resource to execute more than one work item simultaneously.*

The **Simultaneous Execution** pattern is supported. Multiple assignments can be outstanding simultaneously and can be completed any order, according to the control-flow pattern at work.

### *Additional Resources*

> *The ability for a given resource to request additional resources to assist in the execution of a work item that it is currently undertaking.*

The **Additional Resources** pattern is supported through "Delegation" which allows the original assignee to maintain ownership of the item and recall it if desired.

## Exception Handling

Exception handling in workflow is supported at the low level, step-by-step through traditional exception handling mechanisms in C#. Exceptions can be handled in *deterministic* transformation steps and in *nondeterministic IO* steps, but cannot span multiple steps. Code Listing 49 demonstrates several exception handling examples.

```
00 public class ExceptionHandling : ProcessBase
01 {
02     string summary;
03     public override string Summarize() { return summary; }
04 
05     public override IEnumerable<IFavor> Start(object data)
06     {
07         int i;
08 
09         try
10         {
11             i = int.Parse((string)data);
12         }
13         catch (InvalidCastException ex)
14         {
15             summary = "Data could not be cast to a string";
16             i = 0;
17         }
18         catch (ArgumentNullException ex)
19         {
20             summary = "String could not be null";
21             i = 0;
22         }
23         catch (FormatException ex)
24         {
25             summary = "String could not be interpreted as a Int32";
26             i = 0;
27         }
28         catch (OverflowException ex)
29         {
30             summary = "String represents a value out of range of a Int32";
31             i = 0;
32         }
33         catch
34         {
35             throw;
36         }
37         finally
38         {
39             summary = summary.ToUpper();
40         }
41 
42         var breakMe = new IOBroken(i) { TreatErrorsAsData = true };
43         yield return breakMe;
44         if (!breakMe.Succeeded)
```

```
45          {
46              throw breakMe.UnhandledException;
47          }
48
49          yield return AssignTask("i = " + i.ToString(), Initiator);
50      }
51  }
52
53  class IOBroken : IO
54  {
55      int i;
56
57      public IOBroken(int i)
58      {
59          this.i = i;
60      }
61
62      protected override object Perform()
63      {
64          int x = 0;
65
66          return DateTime.Now.Second * (i / x);
67      }
68  }
```

**Code Listing 49 – Exception Handling Implementation Examples**

## Future Work

Additional work can be done to further support the few remaining Workflow Patterns. We are not aware of any fundamental issues with the model that prevent these patterns from being supported by the engine or the hosting server environment.

# Code Listings

# References

[1] van der Aalst, Russell, ter Hofstede, et al. “Workflow Patterns,” April, 2009 <http://www.workflowpatterns.com>

[2] Peter Thiemann “WASH Server Pages,” Lecture Notes in Computer Science, Functional and Logic Programming, 2006, Volume 3945/2006, 277-293

[3] Alessandro Orso and Bryan Kennedy “Selective capture and replay of program executions,” SIGSOFT Softw. Eng. Notes 30, 4. May 1-7, 2005

[4] Steven Osman, Dinesh Subhraveti, Gong Su, Jason Nieh “The Design and Implementation of Zap: A System for Migrating Computing Environments,” Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002). December 9–11, 2002, Boston, MA.

[5] Diimitrios Georgakopoulos, Mark Hornick, Amit Sheth "An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure," Distributed and Parallel Databases, 1995

[6] Microsoft Corporation “Iterators (C# Programming Guide),” April, 2009 <http://msdn.microsoft.com/en-us/library/dscyy5s0.aspx>