

Workflow Design Description

A. F. Griesser, Ph.D
Prometheus Computing LLC

Although many of the larger participants in the electronics industry have their own internal workflow systems, it would be desirable to have a shared workflow that glues together all the participants. High level requirements for such a system were considered in a paper that can be downloaded separately. A design of a workflow system was also produced. The design is available in the form a MagicDraw 14 project. This design was described verbally as part of a presentation (available as a PDF) delivered at a NIST meeting November 16. Regrettably no transcript of the verbal part is available. One of the goals of this design was support for the patterns elucidated by the P4PAIS project, and described at:

- Control patterns: <http://www.workflowpatterns.com/patterns/control/index.php>
- Data patterns: <http://www.workflowpatterns.com/patterns/data/index.php>
- Resource patterns: <http://www.workflowpatterns.com/patterns/resource/index.php>

This document describes how the design implements all 43 control patterns. Most of the data and resource patterns are also supported, but there is no corresponding description of how this is done.

Most of the information here is in the form of table, with one row per pattern. The Description column is split into two sub-rows: the top sub-row summarizes how the pattern is described using Objects from the design. The bottom sub-row describes how those Objects implement the pattern.

Like all designs, this is a sketch: it does not attempt to resolve every last detail, that's the job of the coders.

The workflow patterns are easier to understand and implement if they are only required to handle a single case. This has the advantage of allowing on-the-fly modification of the process on a case-by-case basis. The disadvantage is that it's harder to summarize and report on the collection of cases if they do not all described by the same flow graph.

At any time, the state of the process is represented by the last `W_state` in the `thread_history` of the `active_threads` of the `W_Process`. `W_States` collectively describe the entire history of the `W_Process`. `W_States` are needed for this purpose because a given `Process_Vertex` may be executed multiple times. `W_State` is abstract: if the definition is a `Task_Definition`, the `W_State` is a `Task`. All other `Process_Vertices` are represented in the `thread_history` by a `Control_Vertex`.

Some of the patterns are said to require a "safe" environment. When this is the case, the `W_State` with that `Process_Vertex` as a description checks to see if some other similar `W_State` is already active. If it is, the new `W_State` is suspended until one that's already executing has finished.

#	Name	pp	Description
1	Sequence	8	Process_Vertex#successor
			Task#finish sets status to :FINISHED, gets definition.next_vertices. If this collection is not empty, then schedule new tasks corresponding to those definitions. If the collection is empty, execute thread.implicit_termination. This method sets Thread@status to :FINISHED and then informs parent.child_implicit_termination (which recursively implicitly terminates that parent if all of it's child threads have terminated).
2	Parallel Split	9	Branch (semantics==:FORK_ALL, without branch_selector)
			A new W_Thread is created for each of the Branch#branches, and registered as a child of the Branch thread. If the Branch has a structured_join, set that Join's required_threads to the collection of W_Threads created for the branches. The Branch thread suspends until all the child threads have terminated: typically this never happens, but it is not invalid. The child threads all execute concurrently.
3	Synchronization	10	Join (semantics == :AND)
			When each predecessor has finished, it invokes Join#activate_input. Once all of the predecessors have finished, the Join's successor executes: it does this in a new W_Thread which is registered as a child of all predecessor threads.
4	Exclusive Choice	12	Branch (semantics==:CHOICE, with branch_selector, without structured_join)
			The brach_selector is required: select_one picks a single one of the branches to execute. A new W_Thread is created for this branch, and registered as a child of the Branch thread. The branch thread suspends until this child terminates (which may never happen).
5	Simple Merge	13	Join (with semantics == :XOR_SINGLE)
			This type of Join is treated like a no-op: the Join's successor is activated next, for every predecessor to the Join. The Join's successor remains part of each thread thread that reaches the Join, so that multiple threads have Control_Vertices that have definitions pointing to the same Join and successor.

#	Name	pp	Description
6	Multi-Choice	15	Branch (semantics==:MULTI_CHOICE with branch_selector)
			The brach_selector is required. select_multiple chooses any number of branches. A new W_Thread is created for each of the chosen branches, and registered as a child of the Branch thread. If the Branch has a structured_join, disable the Branch and set the Join's required_threads to the collection of W_Threads created for the branches (the Branch will stay disabled until the Join re-enables it). The Branch thread suspends until all the child threads have terminated: typically this never happens, but it is not invalid. The child threads all execute concurrently.
7	Structured Synchronizing Merge	17	Join (semantics == :STRUCTURED, merges all the branches from a specific Multi-Choice)
			When each predecessor has finished, it invokes Join#activate_input. Activated inputs are checked against required_threads, set by the Branch the Join is re-merging. When all the required_threads have been received, the Join's source_branch is re-enabled, and the successor executes: it does this in a new W_Thread which is registered as a child of all predecessor threads.
8	Multi-Merge	19	Join (with semantics == :XOR_MULTI)
			Effectively identical to 5.
9	Structured Discriminator	20	Join (with semantics == :DISC_STRUCT
			Like 7, but passes the thread of control on to the successor as soon as the first thread reaches the Join. The output is not triggered again when the remaining threads reach the Join, until all of them have gotten there and the Join resets.
10	Arbitrary Cycles	24	There are no looping constraints (other than those inherent in structured vertices)
11	Implicit Termination	25	Process.implicit_termination (if allowed by Process_Definition).
			The thread implicitly terminates when the last successor has completed.

#	Name	pp	Description
12	Multiple Instances without Synchronization	26	Spawn (num_instance_getter references process data, possible control_successor, possible successor, concurrent==true, merge_semantics==:UNMERGED)
			Each time the vertex is activated, it uses num_instance_getter to figure out how many instances of the activity to create. Each activity runs synchronously in it's own child thread. As soon as these activities are kicked off, the control_successor (if any) is activated in the control_thread. If a control successor is present, it must loop back to the same pattern #12 vertex to create additional instances. When the num_instance_getter returns 1, the control successor provides an alternative way to spawn multiple instances of the desired activity. Once the control_thread terminates, the successor to the pattern #12 vertex is activated: this can happen before any of the activity child threads finish. Alternatively the control thread can break out of the loop and continue onward (in which case the successor should not be present). When the control thread is used, child activities may be kicked off at different times. When the num_instance_getter returns something other than 1, multiple child activities are triggered simultaneously.
13	Multiple Instances with a priori Design-Time Knowledge	28	Spawn (num_instance_getter references literal, no control_successor, with successor, merge_semantics! =:UNMERGED)
			Identical to 14, except the num_instance_getter fetches the value of a constant specified at process design time.
14	Multiple Instances with a priori Run-Time Knowledge	29	Spawn (num_instance_getter references process data, no control_successor, with successor, merge_semantics! =:UNMERGED)
			When the vertex is activated, the num_instance_getter computes from process data how many child threads to create. The all execute in parallel. There is no control_successor. The pattern #13 successor does not begin until the last child thread has terminated.
15	Multiple instances without a priori run-time knowledge	31	Spawn (num_instance_getter references process data, possible control_successor, possible successor, merge_semantics! =:UNMERGED)
			Identical to 12, without the constraint the number of instances is known at design time.

#	Name	pp	Description
16	Deferred Choice	33	Branch (semantics=:DEFERRED_CHOICE with branch_selector that waits for external event)
			Identical to 4, but the branch_selector waits for an external event.
17	Interleaved Parallel Routing	34	Partial_Ordering.
			Each Vertex_Order specifies any number of prerequisites for a given Process_Vertex. Initially executed is empty: as each vertex is terminates, it is added to the executed collection. The choice method returns all the Process_Vertices that can run next, taking into account the prerequisites and those vertices which have already executed. The selector Rule selects from the available choices the next vertex to execute. This may or may not involve human interaction. Once all of the choices have executed, the executed collection is emptied and the thread proceeds to the successor of the Partial_Ordering.
18	Milestone	36	State_Test
			The waiting_for Vertex_Path identifies some other Process_Vertex in the process. When the State_Test predecessor attempts to activate its successor, the process checks to see if any of the active_thread thread_history last W_States are defined by that Process_Vertex. If so, the State_Test's successor is activated. Otherwise the State_Test successor will never be activated.
19	Cancel Activity	37	Send_Signal with signal == :CANCEL_THREAD, Token.cancel
			When the Send_Signal is activated, the process finds all the active_threads having a thread_history ending with a W_State defined by the Process_Vertex identified by the Send_Signal@recipient. These threads are then cancelled. Since Regions are required for other purposes anyway, we might as well make it possible for Vertex_Path identify a Region, so as to support pattern 25. If Send_Signal does not have a Vertex_Path, the current thread is cancelled.
20	Cancel Case	39	Cancel with semantics == :PROCESS, Process.cancel
			When Send_Signal is activated, the entire W_Process being executed is cancelled. The recipient is not used.

#	Name	pp	Description
21	Structured Loop	42	Loop
			If semantics is :WHILE, the if_true test is performed: if that computation returns true, the content is executed. Once the content terminates, repeat. If semantics is :UNTIL, the content is executed before the if_true, instead of after.
22	Recursion	44	Call
			Evaluates if_true: if that computation returns false, proceeds to the successor. If if_true does return true, creates a new instance of the process identified by proc_name. This is the general case. For recursion, proc_name is not provided so the name of the current W_Process is used.
23	Transient Trigger	45	Send_Signal (with signal == :PROCEED), Wait_Signal (with semantics == :TRANSIENT)
			One branch of the work flow encounters the Wait_Signal. Another branch encounters a Send_Signal with a recipient that identifies the Wait_Signal. All active Control_Vertices having that Wait_Signal as a definition proceed on to the successor. If there no such Control_Vertices are currently active, nothing happens.
24	Persistent Trigger	47	Send_Signal (with signal == :PROCEED), Wait_Signal (with semantics == :PERSISTENT)
			Operates like 23, except if no Control_Vertex is found, the Send_Signal is added to the Engine's enqueued_signals. When the Wait_Signal specified by the enqueued signal's recipient Vertex_Path is finally encountered, it proceeds to it's successor without waiting.
25	Cancel Region	49	Region, Send_Signal with signal == :CANCEL_REGION and path pointing to Region.
			Identical to Cancel_Activity with a recipient that identifies a Region instead of a Process_Vertex.
26	Cancel Multiple Instance Activity	50	13 together with a Send_Signal with signal == :CANCEL_SPAWN and path pointing to Spawn.
			Taken literally, this is just pattern 19 applied to cancel pattern 13. There seems to be little reason, however, to single out 13 for the this extra feature: patterns 12, 14, and 15 could be made cancelable well.

#	Name	pp	Description
27	Complete Multiple Instance Activity	51	Send_Signal with signal == :COMPLETE and path pointing to a Spawn instance implementing pattern 13, 14, or 15.
			This is similar to 26 in that a Send_Signal is applied to a Spawn, but in this case after the child threads and control thread are stopped, the Spawn's thread proceeds to execute the successor of the Spawn.
28	Blocking Discriminator	53	Join (semantics == :BLOCKING)
			Like 31, but activates the successor as soon as the first predecessor thread reaches the Join.
29	Cancelling Discriminator	55	Join (semantics == :CANCELLING)
			Like 32, but activates the successor as soon as the first predecessor thread reaches the Join.
30	Structured Partial Join	58	Join (semantics == :PARTIAL) with trigger_num specified
			Behaves like 7, but the Join_Task's W_Thread is allowed to proceed after trigger_num of the required_threads have been activated. Subsequent activation of additional predecessors is tracked, but does not trigger further activation of the Join_Task's W_Thread until the all of the inputs have been triggered (returning the Join to it's initial condition) and a the trigger_num criterion has been met all over again. Multiple activations of the same predecessor are ignored.
31	Blocking Partial Join	60	Join (semantics == :BLOCKING_PARTIAL) with trigger_num specified
			Operates the same way as 30, but enqueues multiple activations of the same predecessor, which are used once the Join has been reset. For example, if the Join receives three inputs from predecessor A before the trigger condition has been met, two of those activations are saved for future use. Once the join has triggered, it will act as though it has received an input from predecessor A. The same thing happens after the second deactivation, using the last enqueued activation.
32	Cancelling Partial Join	62	Join (semantics == :CANCELLING_PARTIAL) with trigger_num specified
			Operates the same way as 30, but but reset cancels the required predecessor threads that have not yet reached the Join.

#	Name	pp	Description
33	Generalized AND-Join	63	Join (semantics == :QUEUED)
			Operates like 3, but additional inputs are enqueued for subsequent activations, as in 31.
34	Static Partial Join for Multiple Instances	65	Spawn (merge_semantics==:PARTIAL, num_completions_required interpreted as absolute at start)
			Similar to either pattern 13 or 14, except that the successor is triggered once num_completions_required child vertices have finished (instead of waiting for all the child threads). num_completions_required is evaluated once, before the child activities begin. Triggering the successor has no effect on the remaining child threads: they are allowed to complete.
35	Cancelling Partial Join for Multiple Instances	67	Spawn (merge_semantics==:CANCELLING_PARTIAL, num_completions_required interpreted as absolute at start)
			Identical to 13, except that triggering the successor cancels any unfinished child threads.
36	Dynamic Partial Join for Multiple Instances	68	Spawn (merge_semantics==:PARTIAL, num_completions_required interpreted as relative after each instance completion)
			Similar to either pattern 13 or 14, with dynamic determination of when to proceed on to the successor. As each activity terminates, num_completions_required is computed (or re-computed). As long as the result is greater than zero, the successor does not trigger. The successor is triggered as soon as num_completions_required returns a number smaller than one. The unfinished activities are allowed to complete, but they do not trigger additional activations of the successor.
37	Acyclic Synchronizing Merge	69	Join (semantics==:ACYCLIC, the vertex is not inside a loop unless the source branch vertex is also)
			Effectively identical to 7, without the requirement to merge all the branches from a Multi-Choice. It does, however, need to merge branches (not necessarily all of them) that ultimately come from a Branch (of any variety). There may be intervening Branches as well. Notification of which threads to expect is a bit more complicated. Firstly the Branch must trace the chain of successors (even though intermediate Joins and Branches) to determine which threads can reach the Join in question: it must expect all of them until any intermediate Branches determine which threads to activate.

#	Name	pp	Description
38	General Synchronizing Merge	71	Join (semantics==:GENERAL)
			Similar to 37, but allows loops. See the algorithm cited in the control pattern paper.
39	Critical Section	72	Contention
			<p>Before any activity begins a check is performed to determine if the thread will be entering a Contention@critical_sections Region. Normally this will not be the case, but if the thread is entering a critical region, there are at least two cases:</p> <ul style="list-style-type: none"> • If there is no other thread currently executing a W_State with a definition corresponding to a Process_Vertex in some Region that's part of the same Contention, execution proceeds. • If another thread is in a different critical_section that's part of the same Contention, the entering thread is put to sleep until the other thread (that's already in the conflicting critical region) finishes. To put the activity to sleep, the W_Thread status is set to :WAITING_CRITICAL_SECTION, and the engine is given a new Watcher having for == :WAITING_CRITICAL_SECTION, and affecting == the critical section that's currently running. • If another thread of the same process is already in this Region, it's less clear what to do. Taken literally, it seems execution proceeds. But contention is intended to model contention for a resource, so you probably shouldn't have two threads operating in the the same Region. If this is the better interpretation, the entering thread sleeps until there are no other threads in the the same Contention.
40	Interleaved Routing	73	Branch (semantics==:ACTOR_SEQUENCE)
			The brach_selector selects one branch from those which have not already executed. It then creates a child thread to execute that branch. This proceeds until all the branches have executed. Once that's done, execution proceeds to the Branch successor.

#	Name	pp	Description
41	Thread Merge	74	Thread_Op (split==false)
			When activated, checks to see how many of the W_Thread's sister_threads are waiting at the same Thread_Op. Then computes num_instances. If the number of waiting threads is smaller, sleeps until there are enough threads. If the number is greater or equal, then a new W_Thread is created, starting with the Thread_Op successor, and registered as a child of the first num_instances sister threads.
42	Thread Split	75	Thread_Op (split==true)
			Upon activation, the num_instances rule determines how many threads to start. That many child threads are created, each starting from the Thread successor. The W_Thread that encountered the Thread_Op terminates when the children have all terminated.
43	Explicit Termination	76	Send_Signal (signal==:TERMINUS, vertex_path not used (the process is the recipient))

This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States License: <http://creativecommons.org/licenses/by-nc-sa/3.0/us/>