*new*YAWL: Achieving Comprehensive Patterns Support in Workflow for the Control-Flow, Data and Resource Perspectives*

Nick Russell[1], Arthur H.M. ter Hofstede[1]
Wil M.P. van der Aalst[1,2], David Edmond[1]

[1] *BPM Group, Faculty of Information Technology, Queensland University of Technology*
GPO Box 2434, Brisbane QLD 4001, Australia
{n.russell,a.terhofstede,d.edmond}@qut.edu.au

[2] *Department of Technology Management, Eindhoven University of Technology*
GPO Box 513, NL-5600 MB Eindhoven, The Netherlands
w.m.p.v.d.aalst@tm.tue.nl

Abstract

The *Workflow Patterns* provide a conceptual foundation for the control-flow, data and resource perspectives of process-aware information systems (PAIS). In this paper we present *new*YAWL, a reference language for PAIS based on the workflow patterns. *new*YAWL radically extends previous work undertaken on the *YAWL* language and provides a comprehensive formal description of how the complete set of workflow patterns can be realized and integrated in the context of an operational PAIS.

1 Introduction

Business process management (BPM) has achieved marked prominence over the past five years as companies strive to further optimize the manner in which they undertake their core business activities. Recent surveys [WH06, Kas06] suggest that it is now a major focus for most medium and large companies and is seen as a key means of improving efficiency and reducing cost, the mantra of the modern business environment.

However the increased focus that BPM has received from companies and the media is at odds with the relative maturity of tools and techniques that are actually available in the area. Despite the attention that it is receiving, there is a lack of commonly agreed fundamental concepts that are applicable to the domain and whilst there has been explosive growth in the variety of enactment tools that are available, there is a palpable lack of broadly adopted modelling and enactment standards for business

*This work was partially supported by the Dutch research school BETA as part of the *PATINT* program and the Australian Research Council under the Discovery Grant *Expressiveness Comparison and Interchange Facilitation between Business Process Execution Languages.*

processes. Moreover those notations that show signs of achieving widespread adoptance (e.g. BPMN, BPEL, UML) are not based on a rigorous formal foundation but rather adopt a "committee-based" approach to standards development opportunistically absorbing concepts from a wide variety of domains without any consideration of the manner in which they might be integrated with existing language constructs or how they might actually be enacted in practical terms. The fact that these notations are specified in an informal manner (typically in natural language) leads to further ambiguities as concepts are interpreted and implemented in differing ways.

The need for a common conceptual foundation for business processes was one of the core motivations for the *Workflow Patterns Initiative*. Over the past seven years, this research project has identified a series of core business process constructs which it has presented in the form of *patterns*. In line with the classic definition of patterns, these concepts are generic in form and provide solutions to recurring requirements that arise when modelling and enacting business processes. The patterns encompass the *control-flow*, *data* and *resource* perspectives of business processes and provide a common vocabulary with which to describe them. Since their identification, the applicability of the *workflow patterns* has been demonstrated in a number of ways: they have been used for evaluating and comparing the capabilities of a wide variety of workflow and case handling systems, business process modelling and enactment languages, for workflow tool selection, for business process education and training. They have also influenced the development of a variety of commercial and open-source workflow systems[1].

Most recently, the workflow patterns triggered the development of *YAWL* – an acronym for *Yet Another Workflow Language*. Unlike other efforts in the BPM area, YAWL sought to provide a comprehensive modelling language for business processes based on a formal foundation. The content of the YAWL language was informed by the workflow patterns and one of its the major aims was to demonstrate how the workflow patterns should be both modelled and enacted in a deterministic way. It also sought to illustrate that they could coexist within a common framework. In order to validate that the language was capable of direct enactment, the *YAWL System*[2] was developed, which serves as a reference implementation of the language. Over time, the YAWL language and the YAWL System have increasingly become synonymous and have garnered widespread interest from both practitioners and the academic community alike.

Initial versions of the YAWL System focussed on the control-flow perspective and provided a complete implementation of 19 of the original 20 patterns in this perspective. Subsequent releases have incorporated limited support for selected data and resource patterns, however this effort has been hampered by the lack of a complete formal description of the patterns in these perspectives. Moreover, a recent review [RHAM06] of the control-flow perspective has identified 25 additional patterns. Hereafter in this paper, we refer to the collective group of YAWL offerings developed to date – both the YAWL language as defined in [AH05] and also more recent *YAWL System* implementations of the language based on the original definition (up to and including release Beta 8.1) – as *YAWL*.

In this paper, we propose a new version of *YAWL* – which we subsequently refer

[1] See `http://www.workflowpatterns.com/impact.htm` for further details.

[2] See `http://www.yawl-system.com` for further details of the YAWL System and to download the latest version of the software.

to as *new*YAWL – which aims to support the broadest range of the workflow patterns in the control-flow, data and resource perspectives. The major contributions of this paper are as follows: (1) it provides a comprehensive formal description of the workflow patterns, which to date have only partially been formalized, (2) it provides a complete abstract syntax and graphical notation for *new*YAWL which identifies the characteristics of each of the language elements, and (3) it provides a complete executable model for *new*YAWL which defines the runtime semantics of each of the language constructs.

CPN Tools [Jen97] was selected as the presentation format for the semantic model of *new*YAWL for a variety of reasons. It is a well-established modelling and simulation tool that is widely used for large, process-oriented models. It provides highly effective tool support for the capture, structuring and analysis of such models and has the added attraction that it shares a common foundation (Petri nets) with YAWL thus minimising the potential for conceptual inconsistencies to be introduced during the development of the semantic model. One of the major attractions arising from grounding the semantic model in CPN Tools is that it is executable, thus opening up a variety of future opportunities for analysis and simulation of *new*YAWL. The complete CPN model is included in this report. It is also available for download from the YAWL website[3].

The remainder of this paper is organized as follows. Section 2 considers the issue of language design in a business process context, identifying shortcomings of current efforts (including *YAWL*) and motivating the requirements for *new*YAWL. Section 3 introduces *new*YAWL and discusses the new language extensions. Section 4 presents a syntactic model for all of the constructs in *new*YAWL, together with a series of transformations that allow the control-flow perspective of a *new*YAWL model to be simplified and a set of mappings that allow it to be presented in the form of an initial marking of the semantic model. Section 5 presents a formalization of the *new*YAWL runtime environment in the form of a Coloured Petri-Net (CPN) model which provides a complete semantic description of *new*YAWL. Section 6 presents a worked example showing how a design-time *new*YAWL model can be transformed to an initial marking of the semantic model and illustrates the operation of several *new*YAWL constructs in the context of the semantic model. Section 7 evaluates the patterns support provided by *new*YAWL and finally Section 8 concludes the paper.

2 Language Design Principles

In this section we discuss the requirements for a business process modelling language. We subsequently use these requirements in conjunction with a patterns-based analysis to examine the language design for *YAWL* and on the basis of these insights we determine where shortcomings exist with the current language design. This leads to a proposal for a comprehensive revision – *new*YAWL – which will provide modelling and enactment support for the broadest possible range of the workflow patterns.

[3] See `http://www.yawl-system.com/newYAWL` for details.

2.1 Establishing Language Foundations

In order to effectively support the capture and enactment of business processes, there are some key considerations for the design of a business process modelling language. We consider each of these areas below.

2.1.1 Formality

One of the characterising aspects of the current state of business process modelling is the absence of a formal basis for defining the requirements of a business process and the manner in which it should be enacted. A particularly significant shortcoming is the lack of support for capturing the *semantics* of a business process (i.e. details associated with its fundamental intention, content and operation). There are initial attempts at providing a foundation for specific aspects of business process modelling, in particular process modelling and organizational modelling, however the field as a whole lacks a rigorous foundation. This paucity increases the overall potential for ambiguity when enacting a business process model. For this reason, any approach to language design must be underpinned by a complete and unambiguous description of both its syntax and semantics.

2.1.2 Suitability

In order for a modelling language to have the broadest applicability to a problem domain, it must provide support for the capture and enactment of as wide a range as possible of the requirements encountered in its field of usage. Key to effective capture is the ability of the language to record all of the aspects relevant to a given requirement in a form that accords with its actual occurrence in the problem domain i.e. there should not be a need for significant conceptual reorganization in order for a requirement to be recorded and in general terms, the constructs available in a modelling language should correlate relatively closely with the concepts in associated application domains in which it will be employed. The notion of suitability in the context of workflow languages is considered at length in [Kie03].

2.1.3 Conceptual independence

The modelling language should be complete, self-consistent and independent. Resultant models should be portable across a wide range of potential enactment technologies and the manner in which a model is interpreted must be independent of the environment in which it is subsequently implemented. To this end, it is vital that no aspect of the modelling language relies on specific characteristics of underlying enactment technologies.

2.1.4 Enactability

Ultimately, there is no benefit in proposing a business process modelling language that is not capable of being enacted. This may seem an obvious statement, but several recent initiatives in this area have proposed language elements that are not able to be enacted without some degree of ambiguity [OADH06, RAHW06]. Of particular interest in this area is the ability of the process language to facilitate verification and

validation activities on process models both to establish their consistency and correctness but also for more detailed investigations in regard to their likely performance in actual usage and potential areas for optimization.

2.2 Analysis of YAWL

The original version of *YAWL* was first proposed in 2002 [AH02]. It satisfies all four of the criteria for effective language design identified in Section 2.1. Initially it focussed primarily on the control-flow perspective. Subsequent versions have added support for aspects of the data and resource perspectives to this, but these perspectives are not comprehensively catered for and are not formalized. Most recently in 2006 a comprehensive review of the control-flow perspective was undertaken identifying an additional 25 patterns and providing a complete formalization of all of the control-flow patterns based on Coloured Petri nets [RHAM06].

In order to better understand the capabilities of both the *YAWL* language and the current implementation (Beta 8.1) across the various perspectives, in this section we have undertaken a patterns-based analysis of both of them using the control-flow, data and resource patterns. Table 1 identifies the extent of support for each of the control-flow patterns. It is immediately evident that there is comprehensive support for the original control-flow patterns (WCP1–WCP20) with only the *Implicit Termination* pattern not fully supported. However there are some gaps in its overall support for aspects of control-flow. *YAWL* incorporates notions of cancellation and state, hence patterns such as the *Cancel Region*, *Cancel MI Activity*, *Cancelling Discriminator*, *Static Cancelling Partial Join for MIs*, *Acyclic* and *General Synchronizing Merge*, *Critical Section* and *Interleaved Routing* patterns are directly implemented however other patterns identify areas that lack coverage, in particular there is no direct support for any form of structured iteration, for any form of external triggers, for partial AND-joins (i.e. the N-out-of-M join), for thread-based merges and splits. The overall termination semantics adopted for process instances is also unclear.

No data semantics have been formally defined for the YAWL language, hence all of these patterns evaluations are "–" (i.e. no support). The YAWL implementation adopts a net-based approach to data management with variables bound to workflow nets. Data is passed to and from tasks via input and output parameters. Mappings to and from these parameters are specified as XQuery expressions which are based on net variables. Similarly data passing between block tasks and their associated subprocess decompositions and also to and from multiple instance tasks are also based on XQuery expressions although in these situations, they are somewhat more complex in form than those used for atomic tasks. As a consequence of its XQuery foundations, all data passing is value-based. Table 2 illustrates the data patterns support provided by both the *YAWL* language and implementation. Whilst the support that is provided in the current implementation is effective, it is minimalistic in form and this limited scope is reflected by the small number of data patterns that are directly supported.

Finally, in Table 3 summarises the extent of resource patterns support by the YAWL language and implementation. There is no formal definition of the resource perspective in the YAWL language hence these ratings are negative. For the current implementation, there is limited support for the resource perspective. It provides the ability to distribute work items directly to specific users or indirectly via roles, their lifecycle is relatively simplistic: work items are distributed as soon as they are created,

Nr	Pattern	Language	Implementation	Nr	Pattern	Language	Implementation
	Basic Control				New Control-Flow Patterns		
1	Sequence	+	+	21	Structured Loop	–	–
2	Parallel Split	+	+	22	Recursion	–	+
3	Synchronization	+	+	23	Transient Trigger	–	–
4	Exclusive Choice	+	+	23	Persistent Trigger	–	–
5	Simple Merge	+	+	25	Cancel Region	+	+
	Adv. Branching & Synch.			26	Cancel MI Activity	+	+
6	Multiple Choice	+	+	27	Complete MI Activity	–	–
7	Structured Synch. Merge	+	+	28	Blocking Discriminator	–	–
8	Multiple Merge	+	+	29	Cancelling Discriminator	+	+
9	Structured Discriminator	+	+	30	Structured Partial Join	–	–
	Structural			31	Blocking Partial Join	–	–
10	Arbitrary Cycles	+	+	32	Cancelling Partial Join	–	–
11	Implicit Termination	+/–	+/–	33	Generalized AND-Join	+	+
	Multiple Instance			34	Static Partial Join for MIs	+	+
12	MI without Synchronization	+	+	35	Stat. Canc. Part. Join MIs	+	+
13	MI with a priori D/T Knowl.	+	+	36	Dynamic Partial Join for MIs	–	–
14	MI with a priori R/T Knowl.	+	+	37	Acyclic Synchronizing Merge	+	+
15	MI without a priori R/T Knowl.	+	+	38	General Synchronizing Merge	+	+
	State-based			39	Critical Section	+	+
16	Deferred Choice	+	+	40	Interleaved Routing	+	+
17	Interleaved Parallel Routing	+	+	41	Thread Merge	–	–
18	Milestone	+	+	42	Thread Split	–	–
	Cancellation			43	Explicit Termination	–	–
19	Cancel Activity	+	+				
20	Cancel Case	+	+				

Table 1: Support for Control-Flow Patterns in the YAWL Language and Implementation

by offering them to one or more users. Allocation of work items to users occurs at the instigation of specific users and the time of commencement is also chosen by the user. Users are able to execute multiple work items simultaneously and there is also support for work item execution without user intervention.

2.3 Shortcomings

As with other patterns-based evaluations, the limitations of *YAWL* are immediately evident from a quick glance at Tables 1, 2 and 3.

The control-flow perspective offers a relatively wide range of features, however there is an obvious lack of support the majority of the new patterns both from a language and implementation perspective, in particular there is not for any form of integrated iteration or recursion[4]. It is not possible for the course of execution to

[4]Note that this pattern does function in the current implementation although its operation is not clearly defined and it is not a currently defined language feature

Nr	Pattern	Language	Implementation	Nr	Pattern	Language	Implementation
	Data Visibility				Data Interaction (cont.)		
1	Task Data	–	–	21	Env. to Case – Push	–	–
2	Block Data	–	+	22	Case to Env. – Pull	–	–
3	Scope Data	–	–	23	Subproc. to Env. – Push	–	–
4	Multiple Instance Data	–	+	24	Env. to Subproc. – Pull	–	–
5	Case Data	–	–	25	Env. to Subproc. – Push	–	–
6	Folder Data	–	–	26	Subproc. to Env. – Pull	–	–
7	Global Data	–	–		Data Transfer		
8	Environment Data	–	–	27	by Value Incoming	–	+
	Data Interaction (Internal)			28	by Value Outgoing	–	+
9	between Tasks	–	+	29	Copy In/Copy Out	–	–
10	Block Task to Subprocess	–	+	30	by Reference – Unlocked	–	–
11	Subprocess to Block Task	–	+	31	by Reference – Locked	–	–
12	to Multiple Instance Task	–	+	32	Data Transformation – Inp.	–	+
13	from Multiple Instance Task	–	+	33	Data Transformation – Outp.	–	+
14	Case to Case	–	–		Data-based Routing		
	Data Interaction (External)			34	Task Precond. – Data Exist.	–	–
15	Task to Env. – Push-Oriented	–	–	35	Task Precond. – Data Val.	–	–
16	Env. to Task – Pull-Oriented	–	–	36	Task Postcond. – Data Exist.	–	–
17	Env. to Task – Push-Oriented	–	–	37	Task Postcond. – Data Val.	–	–
18	Task to Env. – Pull-Oriented	–	–	38	Event-based Task Trigger	–	–
19	Case to Env. – Push-Oriented	–	–	39	Data-based Task Trigger	–	–
20	Env. to Case – Pull-Oriented	–	–	40	Data-based Routing	–	+

Table 2: Support for Data Routing in the YAWL Language and Implementation

be influenced by the external environment, i.e. an explicit *trigger* concept is missing. There are also a wide range of join semantics that are not supported, in particular any forms of *Partial Join*, as well as the *Blocking Discriminator*, the *Thread Merge* and *Dynamic Partial Join for MIs*. There is also not a clearly defined basis for the termination of process instances.

The shortcomings of *YAWL* from a data perspective are more significant. The YAWL language does not provide any support for the data perspective. From an implementation standpoint, the situation is a little better although there are a wide variety of data representation paradigms that are not supported (task, scope, case, folder and global) nor is there any recognition of data residing in the external environment. Consequently there is no support for any form of data interaction with the external environment. Data transfer facilities are limited to data passing by value and there is no support for data passing by reference, for managing concurrent data usage (e.g. locking) or for manipulating data being passed between tasks. Similarly, there is minimal support for forms of data-based routing where the sequence of control-flow is influenced by data values and data-based expressions.

The capabilities of *YAWL* from the resource perspective are also limited. There is no language support for the resource perspective. Whilst there is minimalistic re-

Nr	Pattern	Language	Implementation	Nr	Pattern	Language	Implementation
	Creation Patterns				Pull Patterns (cont.)		
1	Direct Allocation	–	+	24	Sys.-Determ. Wk Queue Cont.	–	–
2	Role-Based Allocation	–	+	25	Res.-Determ. Wk Queue Cont.	–	–
3	Deferred Allocation	–	–	26	Selection Autonomy	–	–
4	Authorization	–	–		Detour Patterns		
5	Separation of Duties	–	–	27	Delegation	–	–
6	Case Handling	–	–	28	Escalation	–	–
7	Retain Familiar	–	–	29	Deallocation	–	–
8	Capability-Based Allocation	–	–	30	Stateful Reallocation	–	–
9	History-Based Allocation	–	–	31	Stateless Reallocation	–	–
10	Organizational Allocation	–	–	32	Suspension/Resumption	–	–
11	Automatic Execution	–	+	33	Skip	–	–
	Push Patterns			34	Redo	–	–
12	Distrib. by Offer - Single Res.	–	+	35	Pre-Do	–	–
13	Distrib. by Offer - Mult. Res.	–	+		Auto-Start Patterns		
14	Distrib. by Alloc. - Single Res.	–	–	36	Commencement on Creation	–	–
15	Random Allocation	–	–	37	Creation on Allocation	–	–
16	Round Robin Allocation	–	–	38	Piled Execution	–	–
17	Shortest Queue	–	–	39	Chained Execution	–	–
18	Early Distribution	–	–		Visibility Patterns		
19	Distribution on Enablement	–	+	40	Conf. Unalloc. Wk Item Visib.	–	–
20	Late Distribution	–	–	41	Conf. Alloc. Wk Item Visib.	–	–
	Pull Patterns				Multiple Resource Patterns		
21	Resource-Init. Allocation	–	+	42	Simultaneous Execution	–	+
22	Res.-Init. Exec. - Alloc. WIs	–	+	43	Additional Resource	–	–
23	Res.-Init. Exec. - Offer. WIs	–	–				

Table 3: Support for Resource Patterns in the YAWL Language and Implementation

source support in the current implementation, concepts such as deferred distribution, authorization, distribution constraints based on previous execution history, organizational model and resource capability are missing. There is no opportunity to vary the distribution lifecycle for work items to enable resources to have more autonomy in regard to the manner in which work items are progressed or alternatively to give the system more control over their distribution. Resources are not able to change the basis on which work items are distributed to them or to assign them to other users. The timing of a work item cannot be varied for efficiency reasons either by the system or by individual resources. Finally, there is no ability to restrict the visibility of unassigned and allocated work items on any form of selectable basis.

2.4 Proposed Strategy for Resolution

The results described in the two preceding sections illustrate that whilst *YAWL* was based on the workflow patterns as described in [AH05], the maturing of these patterns across multiple perspectives [RAHE05, RHEA05] and further clarification of

the operation of these patterns [MAHR06, RHAM06] have led to a relative mismatch between the range of patterns now in existence and those supported by the YAWL offering (both formally from a langguage perspective and also practically in terms of those features currently implemented in the YAWL system).

As a remedy to this mismatch, in the remainder of this paper we propose an augmented version of *YAWL – new*YAWL – which provides direct support for a significantly wider range of patterns. **The aim of this new version is twofold: (1) to illustrate how a business process language can support the broadest possible range of the workflow patterns and (2) to demonstrate that the patterns can function together on an integrated basis.**

In order to achieve this, it is first necessary to identify the additional conceptual extensions that *new*YAWL will require. These can be determined based on an analysis of the patterns coverage in Tables 1 to 3 and are as follows:

- Constructs for representing activity iteration (while, repeat loops and recursion) (WCP-21, WCP-22);
- Trigger inclusion to enable execution of a work item to be initiated from outside of the process instance (WCP-23, WCP-24);
- A construct to allow a MI activity to be forcibly completed and subsequent activities to be triggered (WCP-25);
- Introduction of a partial AND-join construct which fires when N of the incoming M branches have received triggers (WCP-30, WCP-31, WCP-32);
- Extensions to the partial AND-join construct to allow (1) a *blocking region* to be specified, comprising activities that cannot be enabled once the join has fired but not yet reset for subsequent firing and (2) subsequent triggers received on an incoming branch that has already been enabled to be retained between partial AND-join resets (WCP-33);
- Inclusion of dynamic partial join support for MI activities (WCP-36);
- Introduction of thread merge and split constructs to allow multiple execution threads to be coalesced and initiated (WCP-40, WCP-41);
- Introduction of explicit termination semantics for process instances and net instances within process instances (WCP-43);
- Support for task, scope, case, folder, global and external data elements (WDP-1, WDP-3, WDP-5, WDP-6, WDP-7, WDP-8);
- The ability to directly pass data elements between task instances (WDP-9);
- The ability to pass data elements between task, case and global constructs and the external environment (WDP-15 to WDP-26);
- The ability to lock data elements during usage to prevent concurrency issues from arising (WDP-31);

- Support for pre and post-conditions on task instances based on existence of data elements and values of expressions containing data elements (WDP-34 to WDP-37);
- The ability to distribute work items based on resource capabilities and authorizations, preceding execution history (both within the process instance and across all process instances) and details of the organization in which the resources belong (WRP-5, WRP-7 to WRP-10);
- The ability to defer the determination of resource and role identity to runtime (WRP-3);
- The ability to directly allocate a work item to a resource and also to start the work item at allocation time (WRP-14, WRP-18);
- The ability to allocate a work item to one of a group of resources based on the size of the resource's work queue, on a round-robin or random basis (WRP-15, WRP-16, WRP-17);
- The ability to limit a resource's work queue to the item on which they are currently working (WRP-4, WRP-42);
- Support for a broader range of resource-initiated interventions in the work distribution process including direct start of offered work items, delegation, deallocation, reallocation on a stateful and stateless basis, and suspension/resumption of work items (WRP-4, WRP-23, WRP-26, WRP-27, WRP-29 to WRP-33);
- Support for system-initiated escalation of work items including state and resource changes (WRP-28);
- Support for various approaches to expediting work item throughput including automatic commencement at creation and allocation time and also for advanced distribution paradigms such as piled and chained execution (WRP-36 to WRP-39);
- Support for work list content management at both resource level and on a system-wide basis (WRP-24, WRP-25);
- Support for selective visibility of allocated and unallocated work items (WRP-40, WRP-41).

Subsequent sections will present the graphical form, an abstract syntax and the underlying semantics for a business process modelling language that achieves these aims.

3 An Introduction to *new*YAWL

*new*YAWL is a reference language for PAIS. It represents a synthesis of the contemporary range of control-flow, data and resource patterns. This section provides an introduction to each of the main constructs in the control-flow, data and resource perspectives of *new*YAWL.

3.1 Control-flow perspective

Figure 1 identifies the complete set of language elements which comprise the control-flow perspective of *new*YAWL. All of the language elements in *YAWL* have been retained[5] and perform the same functions. There is a brief recap of the operation of each of these constructs below. A more detailed discussion of YAWL can be found elsewhere [AH05]. A multitude of new constructs have also been added to address the full range of patterns that have been identified. Each of these constructs is subsequently discussed in more detail.

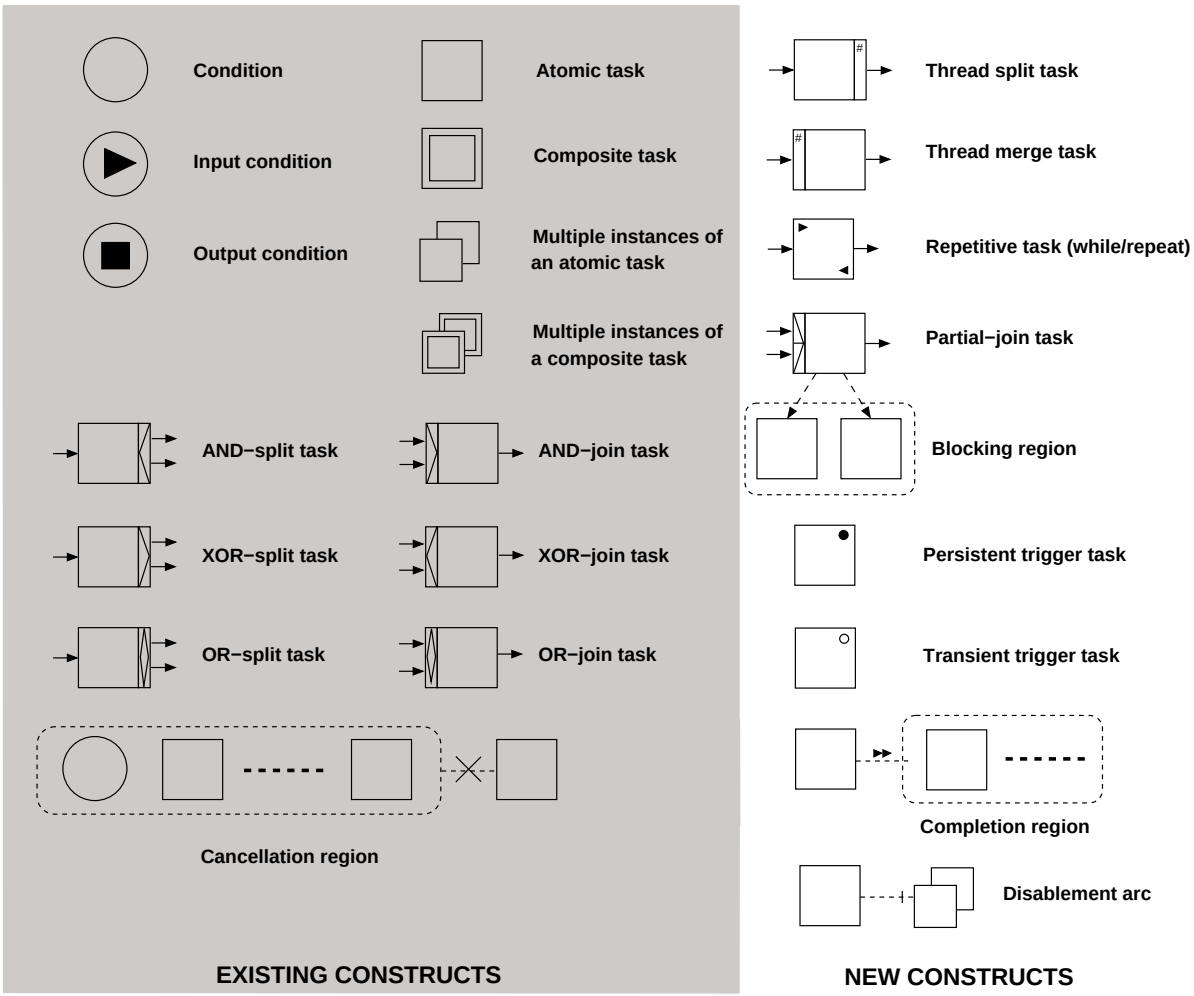


Figure 1: *new*YAWL symbology

3.1.1 Existing YAWL constructs

*new*YAWL inherits all of the existing constructs from YAWL together with its representation of a process model. As in YAWL, a *new*YAWL *specification* is composed of a set of *new*YAWL-nets in the form of a rooted graph structure. Each *new*YAWL-net

[5]The visual format of the cancellation region is slightly different although its operation is unchanged.

is composed of a series of tasks and conditions. Tasks and conditions in *new*YAWL nets play a similar role to transitions and places in Petri nets. Atomic tasks have a corresponding implementation that underpins them. Composite tasks refer to a unique *new*YAWL-net at a lower level in the hierarchy which describes the way in which the composite task is implemented. One *new*YAWL-net, referred to as the *top level process* or *top level net*, does not have a composite task referring to it and it forms the root of the graph.

Each *new*YAWL-net has one unique input and output condition. The input and output conditions of the top level net serve to signify the start and endpoint for a process instance. Similar to Petri nets, conditions and tasks are connected in the form of a directed graph, however there is one distinction in that *new*YAWL allows for tasks to be directly connected to each other. In this situation, it is assumed that an implicit condition exists between them from a semantic perspective.

It is possible for tasks (both atomic and composite) to be specified as having multiple instances (as indicated in Figure 1). Multiple instance tasks (abbreviated hereafter as MI tasks) can have both lower and upper bounds on the number of instances created after initiating the task. It is also possible to specify that the task completes once a certain threshold of instances have completed. If no threshold is specified, the task completes once all instances have completed. If a threshold is specified, the behaviour of the task depends on whether the task is identified as being *cancelling* or *non-cancelling*. If it is *cancelling*, all remaining instances are terminated when the threshold is reached and the task completes. If it is *non-cancelling*, the task completes when the threshold is reached, but any remaining instances continue to execute until they complete normally. However their completion is inconsequential and does not result in any other side-effects. Should the task commence with the required number of minimum instances and all of these complete but the required threshold of instance completions is not reached, the multiple instance task is deemed complete once there are no further instances being executed (either from the original set of instances when the task was triggered or additional instances that were started subsequently). Finally, it is possible to specify whether the number of task instances is fixed after creating the initial instances (i.e. the task is *static*) or whether further instances can be added while there are still other instances being processed (i.e. the task is *dynamic*). Through various combinations of these settings, it is possible to implement all of the MI patterns that have been identified.

Tasks in a *new*YAWL-net can have specific join and split behaviours associated with them. The traditional join and split constructs (i.e. AND-join, OR-join, XOR-join, AND-split, OR-split and XOR-split) are included in *new*YAWL together with three new constructs: *thread split*, *thread merge* and *partial join* which are discussed in detail in Sections 3.1.2 and 3.1.3. The operation of each of the joins and splits in *new*YAWL is as follows:

- *AND-join* – the branch following the AND-join receives the thread of control when all of the incoming branches to the AND-join in a given case have been enabled.

- *OR-join* – the branch following the OR-join receives the thread of control when either (1) each active incoming branch has been enabled in a given case or (2) it is not possible that any branch that has not yet been enabled in a given case will be enabled at any future time.

– *XOR-join* – the branch following the XOR-join receives the thread of control when one of the incoming branches to the XOR-join in a given case has been enabled.

– *AND-split* – when the incoming branch to the AND-split is enabled, the thread of control is passed to all of the branches following the AND-split.

– *OR-split* – when the incoming branch to the OR-split is enabled, the thread of control is passed to one or more of the branches following the OR-split, based on the evaluation of conditions associated with each of the outgoing branches.

– *XOR-split* – when the incoming branch to the XOR-split is enabled, the thread of control is passed to precisely one of the branches following the XOR-split, based on the evaluation of conditions associated with each of the outgoing branches.

Finally, *new*YAWL also inherits the notion of a *cancellation region* from YAWL. A cancellation region encompasses a group of conditions and tasks in a *new*YAWL-net. It is linked to a specific task in the same *new*YAWL-net. At runtime, when an instance of the task to which the cancellation region is connected completes executing, all of the tasks in the associated cancellation region that are currently executing for the same case are withdrawn. Similarly any tokens that reside in conditions in the cancellation region that correspond to the same case are also withdrawn.

This concludes the discussion of constructs in *new*YAWL that are inherited from YAWL. We now introduce new constructs for the control-flow perspective.

3.1.2 Thread split and merge

The *thread split* and *thread merge* constructs provide ways of initiating and coalescing multiple independent threads of control within a given process instance thus providing an alternate mechanism to the multiple instance activity for introducing concurrency into a process.

The *thread split* is a split construct which initiates a specified number of outgoing threads along the outgoing branch when a task completes. It is identified by a # symbol on the righthand side of the task. Only one outgoing branch can emanate from a *thread split* and all initiated execution threads flow along this branch. The number of required threads is identified in the design-time process model.

The *thread merge* is a join construct which coalesces multiple independent threads of control prior to the commencement of an activity. Similar to the *thread merge*, it is denoted by a # symbol however in this case it appears on the lefthand side of the activity. A *thread merge* can only have a single incoming branch.

Figure 2 provides an illustration of the use of these constructs in the context of a *car inspection* process. A car inspection consists of four distinct activities executed sequentially. First the lights are inspected, then the tyres, then the registration documents before finally a record is made of the inspection results. It is expected that a car has five tyres hence five distinct instances of the inspect tyre activity are initiated. Only when all five of these have completed does the following activity commence.

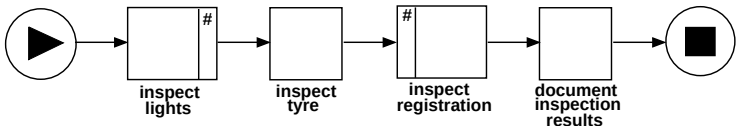


Figure 2: Example of thread split and merge usage

3.1.3 Partial join

The *partial join* (or the *n-out-of-m join*) is a variant of the AND-join which fires when input has been received on `n` of the incoming branches (i.e. it fires earlier than would ordinarily be the case with an AND-join which would wait for input to be received on all `m` incoming branches). The construct resets (and can be re-enabled) when input has been received on all (`m`) of the incoming branches. A *blocking region* can be associated with the join, in which tasks are blocked (i.e. they cannot be enabled or, if already enabled, cannot proceed any further in their execution) after the join has fired until the time that it resets.

Figure 3 illustrates the use of the partial join in the context of a more comprehensive *vehicle inspection* process. After the *initiate inspection* activity has completed, the *mechanical inspection*, *records inspection* and *licence check* activities run concurrently. If all of them complete without finding a problem, then no action is taken and the *file report* activity concludes the process instance. However if any of the inspection activities finds a problem, the *defect notice* activity is initiated and once complete any remaining inspection activities are cancelled, followed by a *full inspection* activity before the final *file report*. In this example, the partial join corresponds to a 1-out-of-3 join without a blocking region.

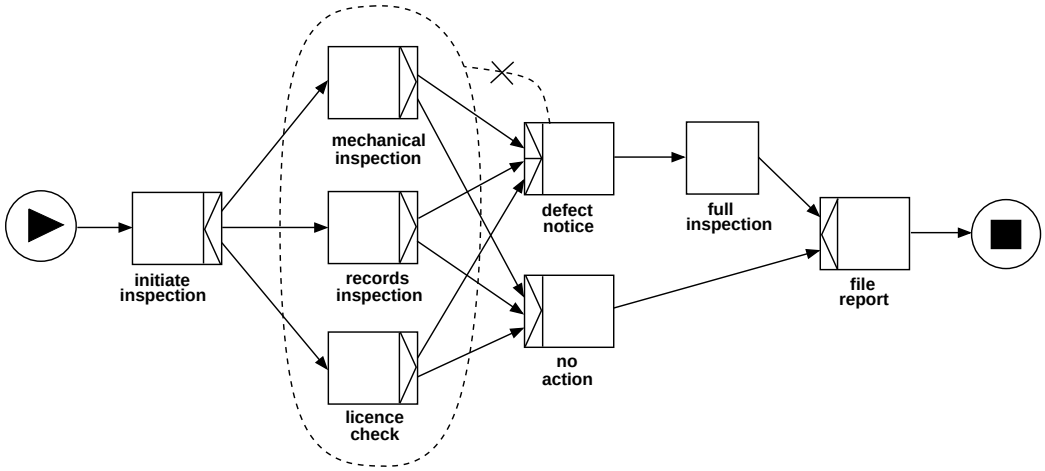


Figure 3: Example of the partial join: *defect notice* is a 1-out-of-3 join

3.1.4 Task repetition

Task repetition is essentially an augmentation of the task construct which has pre-test and/or post-test conditions associated with it. If the pre-test condition is not satisfied,

the task is skipped. If the post-test condition is not satisfied, the task repeats. These conditions allows while, repeat and combination loops to be constructed for individual tasks. Where a specific condition is not associated with a pre-test or post-test, they are assumed to correspond to true resulting in the task being executed or completed respectively

Figure 4 illustrates the implementation of a repeat loop for a given task in the context of a *renew drivers licence* process. An instance of the process is started for each person applying to renew their existing licence. Depending on the results of the *eyesight test* activity, a decision is made as to whether to issue a person with a new licence. For those people that pass the test, they have their photo taken and if they are not satisfied with the result, have it taken again until they are (i.e. *take photo* is executed one or more times). The licence is then issued. Finally the paperwork is filed for all applicants who apply for a licence.

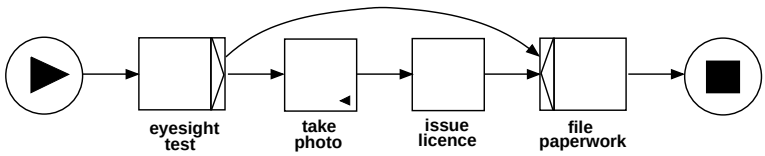


Figure 4: Example of a repeat loop

Another example of the use of this construct is illustrated in Figure 5. This shows how a while loop is implemented in the context of a composite task. The *motor safety campaign* process essentially involves a running a series of vehicle inspections at a given motor vehicle dealer to ensure that the vehicles for sale are roadworthy. It is a simple process, first the vehicles to be inspected are chosen. Then the list of vehicles is passed to the *conduct inspections* task. This is composite in form and repeatedly executes the associated subprocess while there are vehicles remaining on the list to inspect.

There is also support for combination loops in *new*YAWL. This construct is illustrated by the *conduct inspection* task in Figure 6. It has both a pre-test and a post-test condition associated with it. These are evaluated at the beginning and end of each task instance respectively and are distinct conditions. In this example, the pre-test condition is that there are vehicles to inspect. If this condition is true then an instance of the *conduct inspection* task is commenced otherwise the task is skipped. The post-test condition has two parts: (1) there are no more vehicles remaining to inspect or (2) there is not enough time for another inspection. If this condition is false then an instance of the *conduct inspection* task is commenced (providing the pre-test evaluates to true) otherwise the thread of control is passed to the following task.

3.1.5 Persistent and transient triggers

One of the major deficits of YAWL was the inability for processes to be influenced by or respond to stimuli from the external environment unless parts of the process or the entire process was subcontracted to an external service. The inclusion of triggers provides a means for the initiation of work items to be directly controlled from outside of the context of the process instance. Two distinct types of triggers are recognized

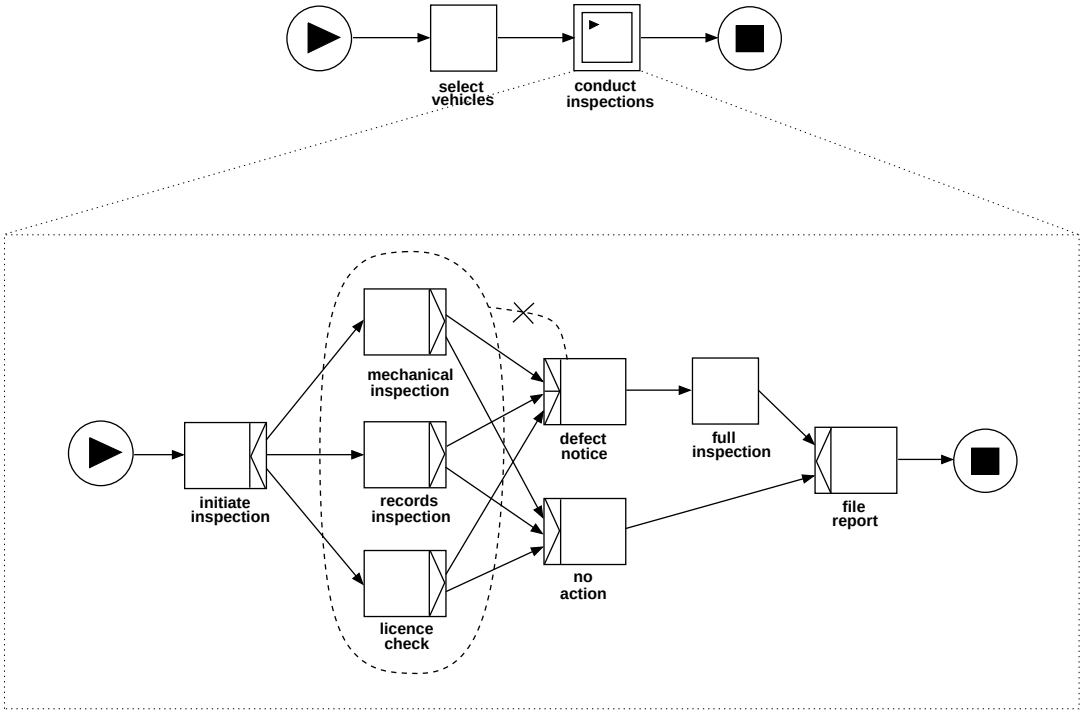


Figure 5: Example of a while loop

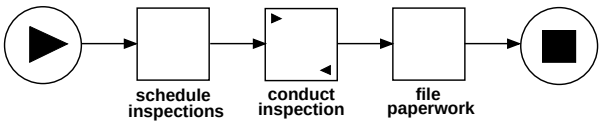


Figure 6: Example of a combination loop

in *new*YAWL: *persistent triggers* and *transient triggers*. These distinguish between the situations where the triggers are durable in form and remain pending for a task in a given process instance until consumed by the corresponding work item and those where the triggers are discarded if they do not immediately cause the initiation of the associated work item. In both cases, triggers have a unique name and must be directed at a specific task instance (i.e. a work item) in a specific process instance. To assist in identifying this information so that triggers can be correlated with the relevant work item, the process environment may provide facilities for querying the identities of currently active process and task instances.

Figure 7 illustrates the operation of a persistent trigger in the context of a *registration plate production* process. Once initiated, the process instance receives triggers from an external system advising that another registration plate is required. It passes these on to the *produce plates* task. As this is a mechanical process which involves the use of a specific machine and bulk materials, it is more efficient to produce the plates in batches rather than individually. For this reason, the *produce plates* activity waits until 12 requests have been received and then executes, cancelling any further instances of the *receive registration request* task once it has completed. The trigger

associated with the *receive registration request* task is persistent in form as it is important that individual registration requests are not lost. A transient trigger does not retain pending triggers (i.e. triggers must be acted on straightaway or they are irrevocably lost) hence it is unsuitable for use in this situation.

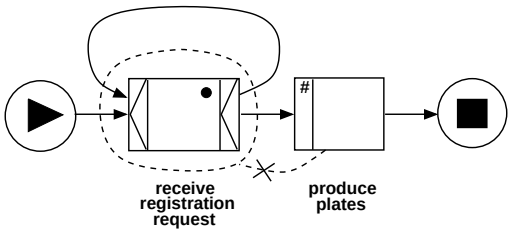


Figure 7: Example of persistent trigger usage

Figure 8 illustrates the operation of a transient trigger in the context of a *refinery plant initiation* process. The *request startup* task commences the firing up of the various plant and equipment associated with an oil refinery, however this task can only proceed when both the *refinery plant initiation* process has been initiated and a *safety check completed* trigger has been received. The *safety check completed* is an output from an internal safety check run by the plant machinery every 90 seconds. The trigger is transient in form and if not consumed immediately, it is discarded and the user must wait until the next trigger is received before the *request startup* task can commence.

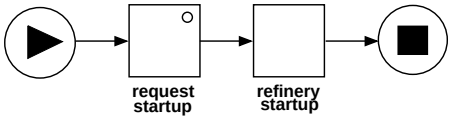


Figure 8: Example of transient trigger usage

3.1.6 Completion region

Cancellation regions have been retained from *YAWL* and a new construct – the *completion region* – has been introduced which operates in an analogous way to the cancellation region, except that when the task to which the region is linked completes, all of the tasks within the region are *force-completed*. This means that if they are enabled or executing, they are disabled and the task(s) subsequent to them are triggered. All such tasks are noted as being "completed" (this may involve inserting appropriate log entries in the execution log). Where the task is composite in form, it is marked as having completed and its associated subprocess(es) has all active tasks within it disabled and any pending tokens removed.

There are two scenarios associated with the completion region that deserve special merit. The first of these is where a multiple instance is included within the completion region. In this situation, any incomplete instances are automatically advanced to a *completed* state. Where data elements are passed from the multiple instance task to subsequent tasks (e.g. as part of the aggregator query) there may be the potential

for the inclusion of incomplete data (i.e. multiple instance data elements will be aggregated from all task instances, regardless of whether they have been defined or not).

The second situation involves data passing from tasks that have not yet completed. If these tasks pass data elements to subsequent tasks, there is the potential for ambiguity in the situation where a "force completion" occurs. In particular, the use of mandatory output data parameters or postconditions (see Section 3.2.4 for more details on the operation of postconditions) that assume the existence of final data values may cause the associated work item to hang. In order to resolve this problem, any tasks within a completion region should not have mandatory output data parameters or postconditions specified for them that assume the existence of final data values or care should be taken to ensure that appropriate values are set at task initiation.

Figure 9 provides an example of the operation of the cancellation region in the context of the *collate monthly statistics* process for the motor vehicle registry. At the end of each month the preceding month's activities are reviewed to gather statistics suitable for publication in the department's marketing brochure. Because this is a time-consuming activity, it is time-bounded to ensure timely publication of the statistics. This generally means that only a subset of the previous month's records are actually examined. After the review is initiated, this is achieved by setting a timeout. In parallel with this, multiple instances of the *review records* task are initiated (one for each motor vehicle record processed in the last month). The *review records* task is in a completion region which is triggered by the *timeout* activity being completed. This stops any further records processing and allows the *publish statistics* task to complete on a timely basis. Should the *review records* task complete ahead of schedule, then the *timeout* is cancelled.

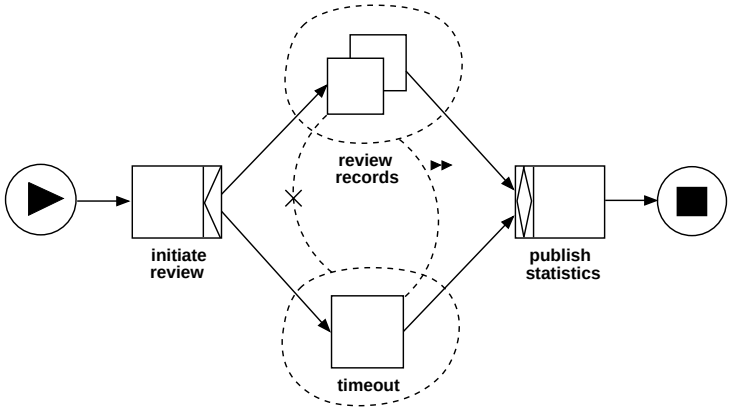


Figure 9: Example of completion region usage: *timeout* forces *review records* to complete

A variant of this process is illustrated in Figure 10. Unlike the previous model, it uses an external trigger to signal when the review process should complete. This trigger is transient in form as any triggers received before the *review records* task has commenced must be ignored. This variant is less efficient as the *publish statistics* task can only commence when the *publish decision* trigger has been received (i.e. there is no provision for the *review records* task to complete early).

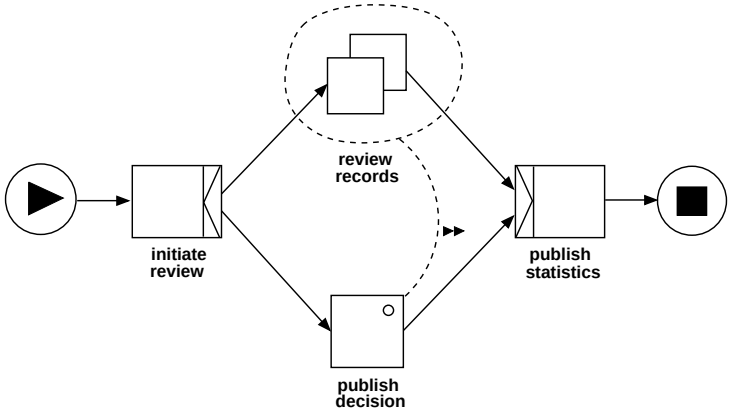


Figure 10: Example of completion region usage using transient triggers

3.1.7 Dynamic multiple instance task disablement

The *disablement link* provides a means of stopping a dynamic multiple instance task from having any further instances created. It is triggered when another task in the same process completes. The example in Figure 11 illustrates part of the paper review process for a conference. After a call for papers has been made, multiple instances of the *accept submission* activity can execute. One is triggered for each paper submission that is received. A manual activity exists for imposing the submission deadline. This is triggered by an external transient trigger. Once received, it disables the *accept submission* activity preventing any further papers from being accepted. However, all of the processing for already accepted papers completes normally and once done, the *organize reviews* activity is initiated.

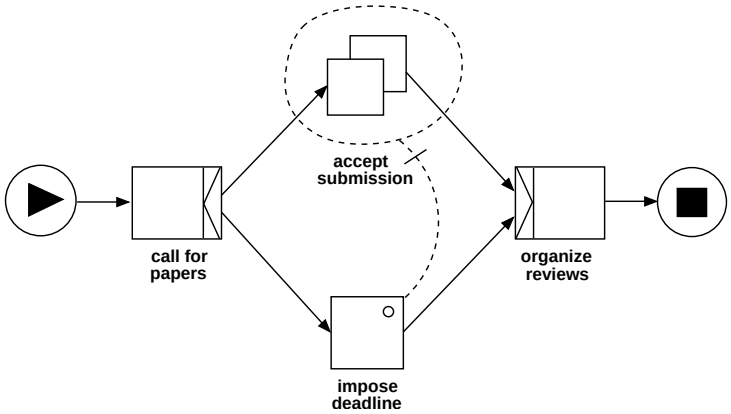


Figure 11: Example of dynamic multiple instance task disablement

3.2 Data perspective

The data perspective of *new*YAWL encompasses the definition of a range of data elements, each with a distinct scoping. These data elements are used for managing

data with a *new*YAWL process instance and are passed between process components using formal parameters. In order to integrate the data and control-flow perspectives, support is provided in *new*YAWL for specifying logical conditions based on data elements that define whether the thread of control can be passed to a given branch in a process (link condition) and also whether a specific process component can start or complete execution (pre or postcondition). Finally, there is also support for managing consistency of data elements in concurrent environments using locks. All of these *new*YAWL data constructs are discussed subsequently.

3.2.1 Data element support

*new*YAWL incorporates a wide range of facilities for data representation and handling. Variables of eight distinct scopings are recognized as follows:

- *External* variables are defined in the operational environment in which a *new*YAWL process executes (but are external to it). They can be accessed throughout all instances of all *new*YAWL process models;
- *Folder* variables are defined in the context of a (named) folder. A folder can be accessed by one or more process instances at runtime. They do not necessarily need to relate to the same process model. Individual process instances can specify the folders they require at the time they initiate execution;
- *Global* variables are bound to a specific *newYAWL* specification and are accessible throughout all *newYAWL-nets* associated with it at runtime;
- *Case* variables are bound to an instance of a specific *newYAWL* specification. At runtime a new instance of the case variable is created for every instance of the *newYAWL* specification that is initiated. This variable instance is accessible throughout all *newYAWL-nets* associated with the *newYAWL* process instance at runtime;
- *Block* variables are bound to a specific instance of a *newYAWL-net*. At runtime a new instance of the block variable is created for every instance of the *newYAWL-net* that is initiated. This variable instance is accessible throughout the *newYAWL-nets* at runtime;
- *Scope* variables are bound to a specific scope within a process instance of a *newYAWL-net*. At runtime a new instance of the scope variable is created for every instance of the *newYAWL-net* instance that is initiated. This variable instance is accessible only to the task instances belonging to the scope at runtime;
- *Task* variables are bound to a specific task. At runtime a new instance of the task variable is created for every instance of the task that is initiated. This variable instance is accessible only to the corresponding task instance at runtime; and
- *Multiple-Instance* variables are bound to a specific instance of a multiple-instance task. At runtime a new instance of the multiple instance variable is created for every instance of the multiple instance task that is initiated. This variable instance is accessible only to this instance of the task instance at runtime.

3.2.2 Data interaction support

There are a series of facilities for transferring data elements between internal and external locations. Data passing between process constructs (e.g. block to task, composite task to subprocess decomposition, block to multiple instance task) is specified using formal parameters and utilizes a function-based approach to data transfer, thus providing the ability to support inline formatting of data elements and setting of default values. Parameters can be associated with tasks, blocks and processes. They take the following form:

`parameter` *input-vars mapping-function output-vars direction participation*

where *direction* specifies whether the parameter is an input or output parameter and *participation* indicates whether it is mandatory or optional. Input parameters are responsible for passing data elements into the construct to which the parameter is bound and output parameters are responsible for passing data elements out. Mandatory parameters require the evaluation of the parameter to yield a defined result (i.e. not an undefined value) in order to proceed. Depending on whether the parameter is an input or output parameter, an undefined result for the parameter evaluation would prevent the associated construct from commencing or completing.

The action of evaluating the parameter for singular tasks, blocks and processes occurs in two steps: (1) the specified *mapping function* is invoked with the values of the nominated *input variables* then (2) the result is copied to the nominated *output variable*. For all constructs other than multiple instance tasks, the resultant value can only be copied to one output variable in one instance of the construct to which they are bound. For multiple instance parameters, the situation is a little more complicated as the parameter is responsible for interacting with multiple variables in multiple task instances. The situation for input multiple instance parameters is illustrated in Figure 12. When a multiple instance parameter is evaluated, it yields a tabular result (indicated by var saltab). The number of rows indicate how many instances of the multiple instance task are to be started. The list of output variables for the parameter correspond to variables that will be created in each task instance. Each row in the tabular result is allocated to a distinct task instance and each column in the row corresponds to the value allocated to a distinct variable in the task instance.

For output multiple instance parameters, the situation is reversed as illustrated in Figure 13 and the values of the input variables listed for the parameter are coalesced from multiple task instances into a single tabular result that is then assigned to the output variable.

3.2.3 Link conditions

As a consequence of a fully-fledged data perspective, *new*YAWL is able to support conditions on outgoing arcs from OR-splits and XOR-splits. These conditions take the following form:

`link condition` *link-function input-variables*

The value from each of the *input-variables* is passed to the *link function* which evaluates to a Boolean result indicating whether the thread of control can be passed to this link or not. Depending on the construct in question, the evaluation sequence of

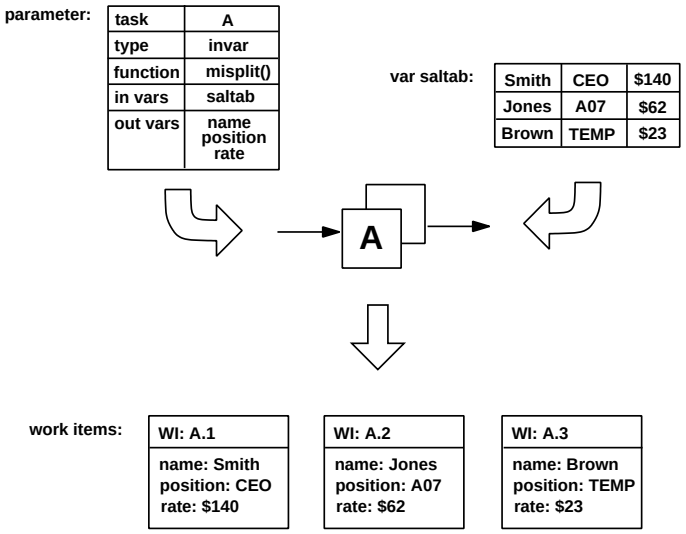


Figure 12: Multiple instance input parameter handling

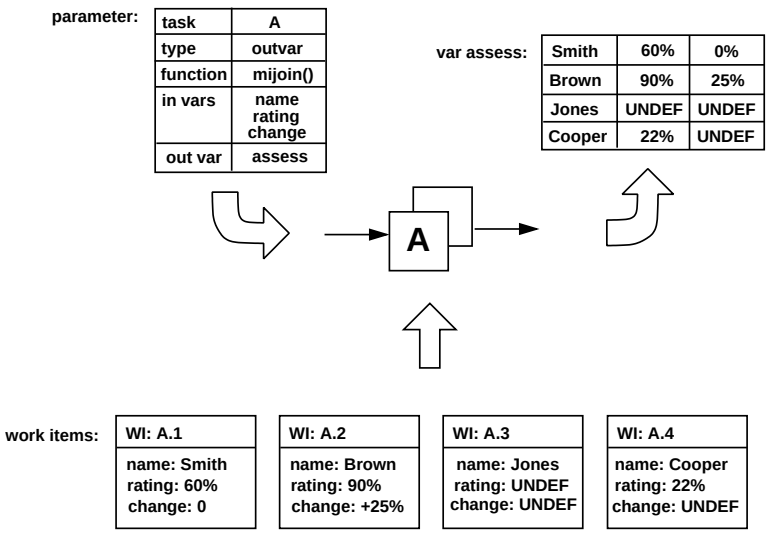


Figure 13: Multiple instance output parameter handling

link conditions varies. For OR-splits, all outgoing link conditions are evaluated and the thread of control is passed to all that evaluate to true. A default link is specified for each OR-split and if none of the link conditions evaluate to true, then the thread of control is passed to this link.

For XOR-splits, there is a sequence specifying the order in which the link conditions should be evaluated. Once the first link condition evaluates to true, then the thread of control is passed to that link and any further evaluation of link conditions ceases. Should none of them evaluate to true, then the thread of control is passed to the link that is specified as the default.

3.2.4 Preconditions and postconditions

Preconditions and postconditions can be specified for tasks and processes in *new*YAWL. They take the same form as link conditions and are evaluated at the enablement or completion of the task or process with which they are associated. Unless they evaluate to true, the task or process instance with which they are associated cannot commence or complete execution.

3.2.5 Locks

*new*YAWL allows tasks to specify data elements that they require exclusive access to (within a given process instance) in order to commence. Once these data elements are available, the associated task instance retains a lock on them until it has completed execution preventing any other task instances from using them concurrently. This lock is relinquished once the task instance completes.

3.3 Resource perspective

*new*YAWL provides support for a broad range of work distribution facilities, inspired by the *Resource Patterns*, that have not been previously embodied in other PAIS. Traditional approaches to work item routing based on itemization of specific users and roles are augmented with a sophisticated array of new features. There are a variety of differing ways in which work items may be distributed to users. Typically these requirements are specified on a task-by-task basis and have two main components:

1. The *interaction strategy* by which the work item will be communicated to the user, their commitment to executing it will be established and the time of its commencement will be determined; and

2. The *routing strategy* which determines the range of potential users that can undertake the work item.

*new*YAWL also provides additional routing constraints that operate with a given case to restrict the users a given work item is distributed to based on the routing decisions associated with previous work items in the case. There is also the ability to specify privileges for each user in order to define the range of operations that they can perform when undertaking work items and there are also two advanced operating modes that can be utilized in order to expedite work throughput for a given user. Each of these features is discussed in detail below.

3.3.1 Work item interaction strategies

The potential range of interaction strategies that can be specified for tasks in *new*YAWL are listed in Table 4. They are based on the specification at three main interaction points – offer, allocation and start – of the identity of the party that will be responsible for determining when the interaction will occur. This can be a resource (i.e. an actual user) or the system. Depending on the combination of parties specified for each interaction, a range of possible distributions are possible as detailed below. From the perspective of the resource, each interaction strategy results in a distinct experience in terms of the way in which the work item is distributed to them. The range

of strategies supported range from highly regimented schemes (e.g. SSS) where the work item is directly allocated to the resource and started for them and the resource has no involvement in the distribution process through to approaches that empower the resource with significant autonomy (e.g. RRR) where the act of committing to undertake a work item and deciding when to start it are completely at the resource's discretion.

As an aid to understanding the distinctions between the various interactions described in Table 4, it is possible to illustrate them quite effectively using UML *Sequence Diagrams* as depicted in Figure 14. These show the range of interactions between the system and resources that can occur when distributing work items. The work distribution, worklist handler and management intervention objects corresponds to the system, resource and process administrator (or manager) respectively. An arrow from one object to another indicates that the first party sends a request to the second, e.g. in the RRR interaction strategy, the first request is a *manual_offer* from the system to the process administrator. The implications of these requests are discussed further in Section 5.

3.3.2 Work item routing strategies

The second component of the work distribution process concerns the routing strategy employed for a given task. This specifies the potential user or a group of users from which the actual user will be selected who will ultimately execute a work item associated with the task. There are a variety of means by which the task routing may be specified as well a series of additional constraints that may be brought into use at runtime. These are summarized below. Combinations of these strategies and constraints are also permissible.

Task routing strategies

Direct user distribution
This approach involves routing to a specified user or group of users.

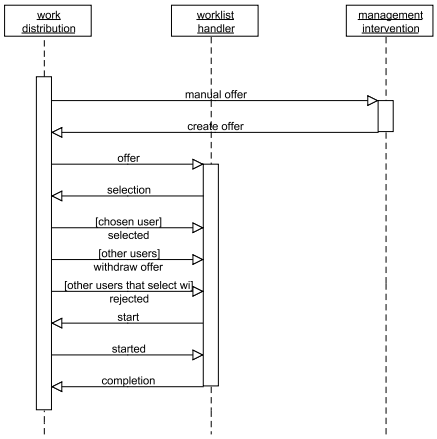
Role-based distribution
This approach involves routing to one or more roles. A role is a "handle" for a group of users that allows the group population to be changed without the necessity to change all of the task routing directives. The population of the role is determined at runtime at the time of the routing activity.

Deferred distribution
This approach allows a task variable to be specified which is accessed at runtime to determine the user or role that the work item associated with the task should be routed to.

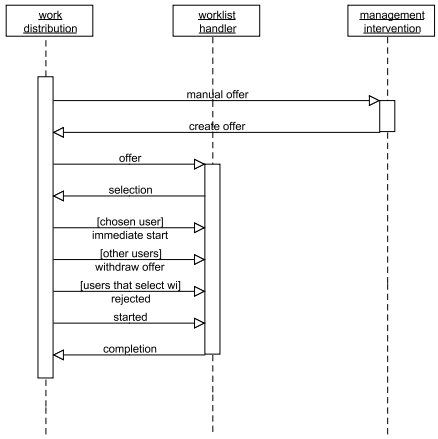
Organizational distribution
This approach allows an organizational distribution function to be specified for a task which utilizes organizational data to make a routing decision. As part of the *new*YAWL semantic model, a simple organizational structure is supported which identifies the concept of organizational groups, jobs and an organizational hierarchy and allows users to be mapped into this scheme. The function takes the following form.

	Offer	**Allocation**	**Start**	**Effect**
SSS	system	system	system	The system directly allocates work to a resource and it is automatically started.
SSR	system	system	resource	The system directly allocates work to a resource. It is started when the user selects the *start* option.
SRS	system	resource	system	The system offers work to one or more users. The first user to choose the *select* option for the work item has the work item allocated to them and it is automatically started. It is withdrawn from other user's work lists.
SRR	system	resource	resource	The system offers work to one or more users. The first user to choose the *select* option for the work item has the work item allocated to them. It is withdrawn from other user's work lists. The user can choose when to start the work item via the *start* option.
RSS	resource	system	system	The work item is passed to a manager who decides which resources the work item should be allocated to. The work item is then directly allocated to that user and is automatically started.
RSR	resource	system	resource	The work item is passed to a manager who decides which resources the work item should be allocated to. The work item is then directly allocated to that user. The user can choose when to start the work item via the *start* option.
RRS	resource	resource	system	The work item is passed to a manager who decides which resource(s) the work item should be offered to. The work item is then offered to those user(s). The first user to choose the *select* option for the work item has the work item allocated to them and it is automatically started. It is withdrawn from all other user's work lists.
RRR	resource	resource	resource	The work item is passed to a manager who decides which resource(s) the work item should be offered to. The work item is then offered to those user(s). The first user to choose the *select* option for the work item has the work item allocated to them. It is withdrawn from all other user's work lists. The user can choose when to start the work item via the *start* option.

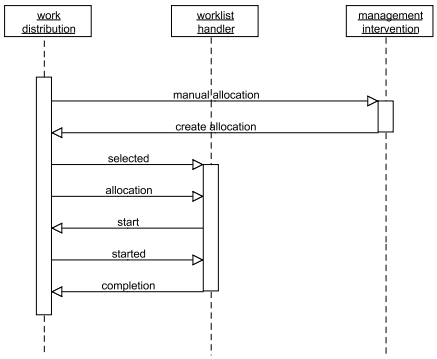
Table 4: Work item interaction strategies supported in *new*YAWL



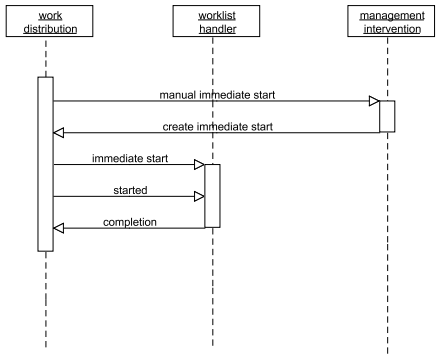
(a) RRR interaction strategy



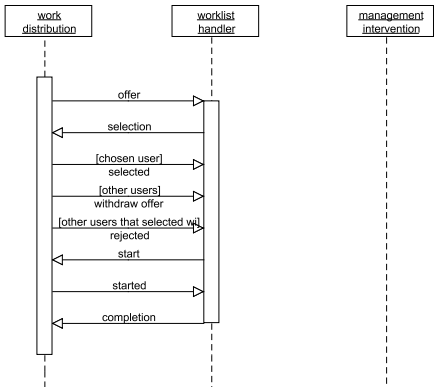
(b) RRS interaction strategy



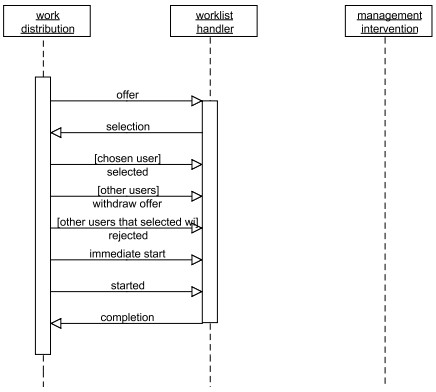
(c) RSR interaction strategy



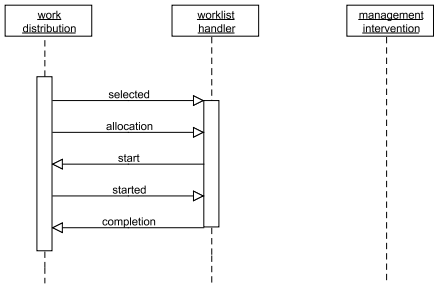
(d) RSS interaction strategy



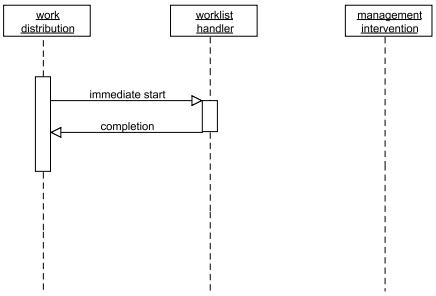
(e) SRR interaction strategy



(f) SRS interaction strategy



(g) SSR interaction strategy



(h) SSS interaction strategy

Figure 14: Work item interaction strategies in *new*YAWL

`function` *function-name users org-groups jobs user-jobs*

where *users* is the base population from whom the final set of users to whom the work item will be distributed will be chosen, *org-groups* is the hierarchy of organizational groups to which users can belong, *jobs* indicates the job roles within the organization and *user-jobs* maps individual users to the jobs that they perform. The function returns a set of users to whom the work item should be routed.

Capability-based distribution
Capabilities can be specified for each user which describe the qualities that they possess that may be of relevance in making routing decisions. A capability-based distribution function can be specified for each task which allows user capabilities to be used in making work distribution decisions. The function takes the following form.

`function` *function-name users user-capabilities*

where *users* is the base population from whom the final set of users to whom the work item will be distributed will be chosen and *user-capabilities* is the list of capabilities that each user possesses. The function returns a set of users to whom the work item should be routed.

Historical distribution
A historical distribution function can be specified for each task which allows historical data – essentially the content of the execution log – to be used in making work distribution decisions. The function takes the following form.

`function` *function-name users events*

where *users* is the base population from whom the final set of users to whom the work item will be distributed will be chosen and *events* are the list of records making up the process log. The function returns a set of users to whom the work item should be routed.

3.3.3 Additional routing constraints

There are several additional constraints supported by *new*YAWL that can be used to further refine the manner in which work items are routed to users. They are used in conjunction with the routing and interaction strategies described above.

Retain familiar
This constraint on a task overrides any other routing strategies and allows a work item associated with it to be routed to the same user that undertook a work item associated with a specified preceding task in the same process instance. Where the preceding task has been executed several times within the same process instance (e.g. as part of a loop), it is routed to one of the users that undertook a preceding instance of the task.

Four eyes principle
This constraint on a task operates in essentially the reverse way to the *Retain familiar* constraint. It ensures that the potential users to whom a work item associated with a task is routed does *not* include the user that undertook a work item associated with a nominated preceding task in the same process instance. Where the preceding task has been executed several times within the same process instance, it cannot be routed to any of the users that undertook a preceding instance of the task.

Random allocation
This constraint on a task ensures that any work items associated with it are only ever routed to a single user where the user is selected from a group of potential users on a random basis.

Round robin allocation
This constraint on a task ensures that any work items associated with it are only ever routed to a single user where the user is selected from a group of potential users on a cyclic basis such that each of them execute work items associated with the task the same number of times (i.e. the distribution is intended to be equitable).

Shortest queue allocation
This constraint on a task ensures that any work items associated with it are only ever routed to a single user where the user is selected from a group of potential users on the basis of which of them has the shortest work queue.

3.3.4 Advanced user operating modes

*new*YAWL supports two advanced operating modes for user interaction with the system. These modes are intended to expedite the throughput of work by imposing a defined protocol on the way in which the user interacts with the system and work items are allocated to them. These modes are described below.

Chained execution
Chained execution is essentially an operating mode that a given user can choose to enable. Once they do this, upon the completion of a given work item in a process, should any of the immediately following tasks in the process instance have potential to be routed to the same user (or to a group of users that include the user), then these routing directives are overridden and the associated work items are placed in the user's work list with a *started* status.

Piled execution
Piled execution is another operating mode however it operates across multiple process instances. It is enabled for a specified user-task combination and once initiated, it overrides any routing directive for the nominated task and ensures that any work items associated with the task in any process instance are routed to the nominated user.

3.3.5 Runtime privileges

*new*YAWL provides support for a number of privileges that can be enabled on a per-user basis that affect the way in which work items are distributed and the various interactions that the user can initiate to otherwise change the normal manner in which the work item is handled. These are summarized in Table 5.

Additionally, there are also privileges that can be enabled for users on a per task basis. These are summarized in Table 6.

Privilege	Explanation
choose	The ability to select the next work item to start execution on
concurrent	The ability to execute more than one work item simultaneously
reorder	The ability to reorder items in the work list
viewoffers	The ability to view work items offered to other users
viewallocs	The ability to view work items allocated to other users
viewexecs	The ability to view work items started by other users
chainedexec	The ability to enter the *chained execution* operating mode

Table 5: User privileges supported in *new*YAWL

Privilege	Explanation
suspend	The ability for a user to suspend execution of work items corresponding to this task
reallocate	The ability for the user to reallocate work items corresponding to this task (which have been commenced) to other users without any implied retention of state
reallocate_state	The ability for the user to reallocate work items corresponding to this task (which have been commenced) to another user and retain the state of the work item
deallocate	The ability for the user to deallocate work items corresponding to this task (which have not yet been commenced) and cause them to be re-allocated
delegate	The ability for the user to delegate work items corresponding to this task (which have not yet been commenced) to another user
skip	The ability for the user to skip work items corresponding to this task
piledexec	The ability to enter the *piled execution* operating mode for work items corresponding to this task

Table 6: User task privileges supported in *new*YAWL

4 Syntax

This section presents an *abstract syntax* for the *new*YAWL reference language. The syntax facilitates the capture of all aspects of the control-flow, data and resource perspectives of a *new*YAWL business process model. The aim of the syntax is twofold. First, to provide complete coverage of the constructs that make up the *new*YAWL reference language. Secondly, to do so in sufficient detail, that it is possible to directly enact a *new*YAWL process on the basis of the information captured about it in the abstract syntax model. The manner in which a *new*YAWL syntactic model is prepared for enactment is illustrated in Figure 15.

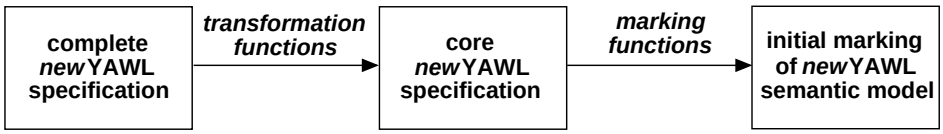


Figure 15: Preparing a *new*YAWL process model for enactment

A *complete newYAWL specification* corresponds to an instance of the abstract syntax. Details of the abstract syntax are presented in Section 4.1. However it is not necessary to describe the enactment of the *new*YAWL language in terms of all of the constructs that it contains. Most of the new control-flow constructs in *new*YAWL can be transformed into existing YAWL constructs without any loss of generality. This approach to enactment has two advantages: (1) the existing YAWL engine can be used as the basis for executing *new*YAWL processes and (2) the existing verification techniques established for the control-flow perspective of YAWL continue to be applicable for *new*YAWL. The transformation of the *new*YAWL control flow constructs into YAWL constructs occurs using a series of *transformation functions*. These are described both informally and formally in Section 4.2. The resultant process model after transformation is called a *core newYAWL specification*.

The manner in which a *new*YAWL specification is ultimately enacted is the subject of Section 5 which presents an operational semantics for *new*YAWL based on CP-nets. A core *new*YAWL specification can be prepared for enactment by mapping it to an *initial marking* of the *new*YAWL semantic model. The activities associated with transforming a core *new*YAWL specification to an initial marking of the semantic model are defined via a series of marking functions which are described in Section 4.3.

4.1 Abstract syntax for *new*YAWL

This section presents a complete abstract syntax for all language elements in *new*YAWL[6]. As described earlier, *new*YAWL assumes the same conceptual basis as *YAWL*, and includes all of its language constructs. A *new*YAWL *specification* is a set of *newYAWL-nets* which form a rooted graph structure. It also has an *Organizational model* associated with it that describes the various resources that are available to undertake work items and the relationships that exist between them in an organizational context.

[6]In doing so it utilizes a number of non-standard mathematical notations. These are explained in further detail in Appendix A

Each *newYAWL-net* has a *Data passing model* associated with it that describes how data is passed between constructs within the process specification. Each *newYAWL-net* is composed of a series of tasks. In order to specify how each task will actually be distributed to specific resources when it is enacted, a *Work distribution model* is associated with each *newYAWL-net*. All of these notions are now formalized, starting with the *new*YAWL *specification*.

Definition 1. (*new*YAWL Specification) A *newYAWL specification* is a tuple = *(NetID, ProcessID, FolderID, TaskID, MITaskID, ScopeID, VarID, TriggerID, TNmap, NYmap, STmap, VarName, DataType, VName, DType, VarType, VGmap, VFmap, VCmap, VBmap, VSmap, VTmap, VMmap, PushAllowed, PullAllowed)* such that:

(* global objects *)
- *NetID* is the set of net identifiers (i.e. the top-level process together with all subprocesses);
- $ProcessID \in NetID$ is the process identifier (i.e. the top-level net);
- *FolderID* is the set of identifiers of data folders that can be shared among a selected group of cases;
- *TaskID* is the set of task identifiers in nets;
- $MITaskID \subseteq TaskID$ is the set of identifiers of multiple instance tasks;
- *ScopeID* is the set of scope identifiers which group tasks within nets;
- *VarID* is the set of variable identifiers used in nets;
- *TriggerID* is the set of trigger identifiers used in nets;

(* decomposition *)
- $TNmap\colon TaskID \nrightarrow NetID$ defines the mapping between composite tasks and their corresponding subprocess decompositions which are specified in the form of a *new*YAWL-net, such that for all t, $TNmap(t)$ yields the *NetID* of the corresponding *newYAWL*-net, if it exists;
- $NYmap\colon NetID \rightarrow$ *new*YAWL-nets, i.e. each net has a complete description of its contents such that for all $n \in NetID$, *NYmap(n)* is governed by Definition 2 where the notation T_n denotes the set of tasks that appear in a net n. Tasks are not shared between nets hence $\forall_{m,n \in NetID}[T_m \cap T_n \neq \emptyset \Rightarrow m = n]$. *TaskID* is the set of tasks used in all nets and is defined as $TaskID = \bigcup_{n \in NetID} T_n$;
- In the directed graph defined by $G = (NetID, \{(x, y) \in NetID \times NetID \mid \exists_{t \in T_x}[t \in dom(TNmap) \wedge TNmap(t) = y]\})$ there is a path from *ProcessID* to any node $n \in NetID$;
- $STmap\colon ScopeID \rightarrow \mathbb{P}^+(TaskID)$ such that $\forall_{s \in ScopeID} \exists_{n \in NetID}[STmap(s) \subseteq T_n]$ i.e. a scope can only contain tasks within the same net;

(* variables *)
- *VarName* is the set of variable names used in all nets;
- *DataType* is the set of data types;
- $VName\colon VarID \rightarrow VarName$ identifies the name for a given variable;
- $DType\colon VarID \rightarrow DataType$ identifies the underlying data type for a variable;
- $VarType\colon VarID \rightarrow \{Global, Folder, Case, Block, Scope, Task, MI\}$ describes the various variable scopings that are supported. The notation $VarID^x = \{v \in VarID \mid VarType(v) = x\}$ identifies variables of a given type;

- $VGmap \subseteq VarID^{Global}$, identifies global variables that are associated with the entire process;
- $VFmap\colon VarID^{Folder} \rightarrow FolderID$ identifies the folder to which each folder variable corresponds, such that $dom(VFmap) = VarID^{Folder}$ and $\forall_{v_1,v_2 \in dom(VFmap)}[VName(v_1) = VName(v_2) \Rightarrow (v_1 = v_2 \vee VFmap(v1) \neq VFmap(v2))]$, i.e. folder variable names are unique within a given folder;
- $VCmap \subseteq VarID^{Case}$ identifies the case variables for the process;
- $VBmap\colon VarID^{Block} \rightarrow NetID$ identifies the specific net to which each block variable corresponds, such that $dom(VBmap) = VarID^{Net}$;
- $VSmap\colon VarID^{Scope} \rightarrow ScopeID$ identifies the specific scope to which each scope variable corresponds, such that $dom(VSmap) = VarID^{Scope}$;
- $VTmap\colon VarID^{Task} \rightarrow TaskID$ identifies the specific task to which a task variable corresponds, such that $dom(VTmap) = VarID^{Task}$;
- $VMmap\colon VarID^{MI} \rightarrow MITaskID$ identifies the specific task to which each multiple-instance variable corresponds, such that $dom(VMmap) = VarID^{MI}$;
- $PushAllowed \subseteq VarID$ identifies those variables that can have their values updated from external data locations;
- $PullAllowed \subseteq VarID$ identifies those variables that can have their values read from external data locations;

Having described the global characteristics of a *newYAWL specification*, we can now proceed to the definition of a *newYAWL-net*. Note that *newYAWL-nets* is the set of all instances governed by Definition 2.

Definition 2. (*newYAWL*-net) A *new*YAWL-net is a tuple *(nid, C, i, o, T, T_A, T_C, M, F, Split, Join, Default, $<_{XOR}$, Rem, Comp, Block, Nofi, Disable, Lock, Thresh, ThreadIn, ThreadOut, ArcCond, Pre, Post, PreTest, PostTest, WPre, WPost, Trig, Persist)* such that:

(* basic control-flow elements *)
- $nid \in NetID$ is the identity of the *newYAWL-net*;
- C is a set of conditions;
- $i \in C$ is the input condition;
- $o \in C$ is the output condition;
- T is the set of tasks;
- $T_A \subseteq T$ is the set of atomic tasks;
- $T_C \subseteq T$ is the set of composite tasks;
- T_A and T_C form a partition over T;
- $M \subseteq T$ is the set of multiple instance tasks;
- $F \subseteq (C \setminus \{o\} \times T) \cup (T \times C \setminus \{i\}) \cup (T \times T)$ is the flow relation, such that every node in the graph $(C \cup T, F)$ is on a directed path from i to o;
- $Split : T \nrightarrow \{AND, XOR, OR, THREAD\}$ specifies the split behaviour of each task, such that $\forall_{t \in dom(Split)}[Split(t) = THREAD \Rightarrow |\, t\bullet\,| = 1]$, i.e. thread splits can only have one output arc;
- $Join : T \nrightarrow \{AND, XOR, OR, PJOIN, THREAD\}$ specifies the join behaviour of each task such that $\forall_{t \in dom(Join)}[Join(t) = THREAD \Rightarrow |\, \bullet t\,| = 1]$, i.e. thread merges can only have one input arc;
- $Default \subseteq F$, $Default\colon dom(Split \rhd \{OR, XOR\}) \rightarrow T \cup C$ denotes the default arc for each OR-split and XOR-split.

- $<_{XOR} \subseteq \{t \in T \mid Split(T) = XOR\} \times \mathbb{P}(T \cup C) \times (T \cup C)$ describes the evaluation sequence of outgoing arcs from an XOR-split such that for any $(t, V) \in <_{XOR}$ we write $<^t_{XOR} = V$ and V is a strict total order over $t\bullet = \{x \in T \cup C \mid (t, x) \in F\}$. Link conditions associated with each arc are evaluated in this sequence until the first evaluates to true. If none evaluate to true, the default arc indicated by *Default(t)* is selected, thus ensuring that exactly one outgoing arc is enabled;
- $Rem : T \nrightarrow \mathbb{P}^+(T \cup C \backslash \{i, o\})$ specifies the additional tokens to be removed by emptying a part of the net and tasks that should be cancelled as a consequence of an instance of this task completing execution;
- $Comp : T \nrightarrow \mathbb{P}^+(T)$ specifies the tasks that are force completed as a consequence of the completion of an instance of this task completing execution;
- $Block : T \nrightarrow \mathbb{P}^+(T)$ where $t \in dom(Block) \Leftrightarrow Join(t) = PJOIN$, specifies the tasks that are blocked after the firing of a partial join task prior to its reset such that $\forall_{t \in dom(Block)}\ t \notin Block(t)$;
- $Nofi : M \rightarrow \mathbb{N} \times \mathbb{N}^{inf} \times \mathbb{N}^{inf} \times \{dynamic, static\} \times \{cancelling, non\text{-}cancelling\}$ specifies the multiplicity of each task – in particular the lower and upper bound of instances to be created at task initiation, the threshold for continuation indicating how many instances must complete for the thread of control to be passed to subsequent tasks, whether additional instances can be created "on the fly" once the task has commenced and whether partial synchronization results in remaining instances being cancelled or not;
- $Disable : T \rightarrow \mathbb{P}^+(M)$ specifies the multiple-instance tasks that are disabled from creating further instances as a consequence of an instance of this task completing execution.

(* locks *)

- $Lock : T \rightarrow \mathbb{P}(VarID)$ is a function mapping tasks to the data elements that they require locks on during execution;

(* partial joins *)

- $Thresh : T \nrightarrow \mathbb{N}$ where $t \in dom(Thresh) \Leftrightarrow Join = PJOIN$ and $\forall_{t \in dom(Thresh)} [1 \leq Thresh(t) < \ | \bullet t |]$ identifies the firing threshold for partial (i.e. n-out-of-m type) joins;

(* thread splits and joins *)

- $ThreadIn : T \nrightarrow NatExpr$ where $dom(ThreadIn) = dom(Join \rhd \{THREAD\})$ identifies the number of incoming threads that the task requires in order to fire;
- $ThreadOut : T \nrightarrow NatExpr$ where $dom(ThreadOut) = dom(Split \rhd \{THREAD\})$ identifies the number of outgoing threads that the task generates on firing;

(* conditions on arcs *)

- $ArcCond : F \cap (dom(Split \rhd \{XOR, OR\}) \times (T \cup C)) \rightarrow BoolExpr$ identifies the specific condition associated with each branch of an OR or XOR split.

(* pre/post conditions and pre/post tests for task iteration *)

- $Pre : T \nrightarrow BoolExpr$ is a function identifying tasks which have a precondition associated with them;

– $Post: T \nrightarrow BoolExpr$ is a function identifying tasks which have a postcondition associated with them;
– $PreTest: T \nrightarrow BoolExpr$ is a function identifying tasks which have an iteration pre-test condition associated with them. Where this condition is met, at enablement the task is executed otherwise it is skipped;
– $PostTest: T \nrightarrow BoolExpr$ is a function identifying tasks which have an iteration post-test condition associated with them. Where this condition is not met at task completion, the task is repeated otherwise it completes execution;
– $WPre \in BoolExpr$ indicates the precondition for commencement of a process instance;
– $WPost \in BoolExpr$ indicates the postcondition for completion of a process instance;

(* triggers *)

– $Trig: T \nrightarrow TriggerID$ identifies tasks which have a trigger associated with them;
– $Persist \subseteq dom(Trig)$ identifies the subset of triggers which are persistent;

Each *newYAWL-net* is identified by a unique *nid*. As for *YAWL*, the tuple (C,T,F) takes its form from classical Petri nets where C corresponds to the set of conditions, T to the set of tasks and F to the flow relation (i.e. the directed arcs between conditions and tasks). However there are two distinctions: (1) i and o describe specific conditions that denote the start and end condition for a net and (2) the flow relation allows for direct connections between tasks in additional to links from conditions to tasks and tasks to conditions.

Expressions are denoted informally via *Expr* which identifies the set of expressions relevant to a *newYAWL-net*. It may be divided into a number of disjoint subsets including *BoolExpr*, *IntExpr*, *NatExpr*, *StrExpr* and *RecExpr*, these being the sets of expressions that yield Boolean, integer, natural number, string and record-based results when evaluated. There is also recognition for work distribution purposes of capability-based, historical and organizational distribution functions that are denoted by the *CapExpr*, *HistExpr* and *OrgExpr* subsets of *Expr* respectively.

As already indicated, one of the major features of *newYAWL* is the inclusion of the data perspective. In order to facilitate the utilization of data elements during the operation of the process, it is necessary to define a model to describe the way in which data is passed between active process components.

Definition 3. (Data passing model) Within the context of a *newYAWL-net nid*, there is a data passing model *(InPar, OutPar, OptInPar, OptOutPar, MIInPar, MIOutPar, InNet, OutNet, OptInNet, OptOutNet, InProc, OutProc, OptInProc, OptOutProc)* with the following components:

(* data passing to/from atomic tasks *)

– $InPar: T_A \times VarID \nrightarrow Expr$ is a function identifying the input parameter mappings to a task at initiation, such that $\forall_{(t,v)\in dom(InPar)}[VTmap(v) = t]$;
– $OutPar: T_A \times VarID \nrightarrow Expr$ is a function identifying the output parameter mappings from a task at completion, such that $\forall_{(t,v)\in dom(OutPar)}[v \in VarID^{Global} \cup VarID^{Folder} \cup VarID^{Case} \vee t \in T_{VBmap(v)} \cup STmap(VSmap(v))]$, i.e. the output variable can be a global, case or folder variable, a block variable corresponding to this net or a scope variable in the same scope as the task;

– $OptInPar \subseteq dom(InPar)$ identifies those input mappings to a task which are optional;
– $OptOutPar \subseteq dom(OutPar)$ identifies those output mappings to a task which are optional;

(* data passing between composite tasks and subprocess decompositions *)
– $InNet\colon T_C \times VarID \nrightarrow Expr$ is a function identifying the input parameter mappings to the subprocess corresponding to a composite task at commencement, such that $\forall_{(t,v)\in dom(InNet)}\,[VBmap(v) = TNmap(t)]$;
– $OutNet\colon T_C \times VarID \nrightarrow Expr$ is a function identifying the output parameter mappings from a subprocess decomposition to its parent composite task at completion, such that $\forall_{(t,v)\in dom(OutPar)}\,[v \in VarID^{Global} \cup VarID^{Folder} \cup VarID^{Case} \vee t \in T_{VBmap(v)} \cup STmap(VSmap(v))]$, i.e. the output variable can be a global, case or folder variable, a block variable corresponding to this net or a scope variable in the same scope as the task;
– $OptInNet \subseteq dom(InNet)$ identifies those input mappings to a subprocess which are optional;
– $OptOutNet \subseteq dom(OutNet)$ identifies those output mappings from a subprocess which are optional;

(* data passing to/from multiple-instance tasks *)
– $MIInPar\colon TaskID \times \mathbb{P}(VarID) \nrightarrow RecExpr$ is a function identifying the input parameter mapping for each instance of a multiple instance task at commencement, such that $\forall_{(t,v)\in dom(MIInPar)}[t \in dom(Nofi) \wedge VMmap(v) = t]$;
– $MIOutPar\colon TaskID \times VarID \nrightarrow RecExpr$ is a function identifying output parameter mappings from the various multiple instance tasks at completion, such that $\forall_{(t,v)\in dom(OutPar)}\,[t \in T_{VBmap(v)} \cup STmap(VSmap(v)) \vee v \in VarID^{Global} \cup VarID^{Folder} \cup VarID^{Case}]$, i.e. the output variable can be a scope variable in the same scope as the task, a block variable corresponding to this net or any global, folder or case variable;

(* data passing to/from nets *)
– $InProc\colon VarID \nrightarrow Expr$ is a function identifying the input parameter mappings to a process instance at commencement, such that $\forall_{v\in dom(InProc)}[(VarType(v) = Scope \wedge STmap(VSmap(v)) \subseteq T_{ProcessID}) \vee (VarType(v) = Block \wedge VBmap(v) = ProcessID) \vee VarType(v) = Case]$, i.e. the parameter value can only be mapped to a scope or block variable in the top level net or a case variable;
– $OutProc\colon VarID \nrightarrow Expr$ is a function identifying the output parameter mappings from a process instance at completion, such that $\forall_{v\in dom(OutProc)}[VarType(v) \in \{Global, Folder\}]$, i.e. the parameter value can only be mapped to a folder or global variable;
– $OptInProc \subseteq dom(InProc)$ identifies optional input mappings to a process;
– $OptOutProc \subseteq dom(OutProc)$ identifies optional output mappings from a process;

*new*YAWL also incorporates a comprehensive characterization of the resource perspective. This characterization is composed of two main components: the *Organizational model* which provides a description of the overall structure of the organization in terms of organizational groups, users, jobs and reporting lines and the *Work distribution model* which defines the manner in which work items are distributed to users

at runtime for execution as well as identifying the interactions that individual users are able to invoke to influence the way in which this distribution occurs. These two models are specified in more detail below.

Definition 4. (Organizational model) Within the context of a *new*YAWL specification *ProcessID*, there is an organizational model described by the tuple *(UserID, RoleID, CapabilityID, OrgGroupID, JobID, CapVal, RoleUser, OrgGroupType, GroupType, JobGroup, OrgStruct, Superior, UserQual, UserJob*) as follows:

(* basic definitions *)
- *UserID* is the set of all individuals to whom work items can be distributed;
- *RoleID* is the set of designated groupings of those users;
- *CapabilityID* is the set of qualities that a user may possess that are useful when making work distribution decisions;
- *OrgGroupID* is the set of groups within the organization;
- *JobID* is the set of all jobs within the organization;
- *CapVal* is the set of values that a capability can have;

(* organizational definition *)
- $RoleUser: RoleID \rightarrow \mathbb{P}(UserID)$ indicates the set of users in a given role;
- $OrgGroupType = \{team, group, department, branch, division, organization\}$ identifies the type of a given organizational group;
- $GroupType: OrgGroupID \rightarrow OrgGroupType$;
- $JobGroup: JobID \rightarrow OrgGroupID$ indicates which group a job belongs to;
- $OrgStruct: OrgGroupID \nrightarrow OrgGroupID$ forms an acyclic intransitive graph with a unique root which identifies a composition hierarchy for groups;
- $Superior: JobID \nrightarrow JobID$ forms an acyclic intransitive graph which identifies the reporting lines between jobs;

(* user definition *)
- $UserQual: UserID \times CapabilityID \rightarrow CapVal$ identifies the capabilities that a user possesses;
- $UserJob: UserID \rightarrow \mathbb{P}(JobID)$ maps a user to the jobs that they hold;

The *new*YAWL organizational model takes the form of a tree, based on the reporting relationships between groups where the most senior group within the organization is the root node of the tree. This model is deliberately chosen to be simple and generic so that it applies to a relatively broad range of situations in which *new*YAWL may be used. Finally, the *Work distribution model* is presented that captures the various ways in which work items are distributed to users and any constraints that need to be taken into account when doing so.

Definition 5. (Work distribution model) Within the context of a *newYAWL-net nid*, it is possible to describe the manner in which work items are distributed to users for execution. A work distribution model is a tuple *(Auto,* T_M*, Initiator, DistUser, DistRole, DistVar, SameUser, FourEyes, HistDist, OrgDist, CapDist, UserSel, UserPriv, UserTaskPriv)* as follows:

(* work allocation *)

- $Auto \subseteq T_A$ is the set of tasks which execute automatically without user intervention, where T_A is the set of atomic tasks;
- $T_M \subseteq T_A \backslash Auto$ is the set of atomic tasks that must be allocated to users for execution;
- $Initiator : T_M \rightarrow \{system, resource\} \times \{system, resource\} \times \{system, resource\}$ indicates who initiates the offer, allocate and commence actions;
- $DistUser : T_M \nrightarrow \mathbb{P}(User)$ identifies the users to whom a task should potentially be distributed;
- $DistRole : T_M \nrightarrow \mathbb{P}(Role)$ identifies the roles to whom a task should potentially be distributed;
- $DistVar : T_M \nrightarrow \mathbb{P}(VarID)$ identifies a set of variables holding either user or roles to whom a task should potentially be distributed;
- dom($DistUser$), dom($DistRole$) and dom($DistVar$) form a partition over T_M;
- $SameUser : T_M \nrightarrow T_M$ is an irreflexive function that identifies that a task should be executed by one of the same users that undertook another specified task in the same case;
- $FourEyes : T_M \nrightarrow T_M$ is an irreflexive function that identifies a task that should be executed by a different user to the one(s) that executed another specified task in the same case;
- $HistDist : T_M \nrightarrow HistExpr$ identifies a set of historical criteria that users that execute the task must satisfy;
- $OrgDist : T_M \nrightarrow OrgExpr$ identifies a set of organizational criteria that users that execute the task must satisfy;
- $CapDist : T_M \nrightarrow CapExpr$ identifies a set of capabilities that users that execute the task must possess;
- $UserSel : T_M \nrightarrow \{random, round\text{-}robin, shortest\text{-}queue\}$ indicates how a specific user who will execute a task should be selected from a group of possible users;

(* user privilege definition *)

- $UserPriv : UserID \nrightarrow \mathbb{P}(UserAuthKind)$ indicates the privileges that an individual user possesses, where $UserAuthKind = \{$*choose, concurrent, reorder, viewoffers, viewallocs, viewexecs, chainedexec*$\}$;
- $UserTaskPriv : UserID \times TaskID \nrightarrow \mathbb{P}(UserTaskAuthKind)$ indicates the privileges that an individual user possesses in relation to a specific task, where $UserTaskAuthKind = \{$*suspend, start, reallocate, reallocate_state, deallocate, piledexec, delegate,skip*$\}$;

4.2 From complete to core *new*YAWL

The complete capabilities of *new*YAWL are captured by Definitions 1 to 5 of the abstract syntax. Several of the new constructs can be seen in terms of other constructs and thus can be eliminated through structural transformations to the *new*YAWL specification in which they occur, thus minimizing the need to extend the underlying execution environment. In this section, we present six distinct sets of transformations that simplify a *new*YAWL specification and allow these constructs to be directly embodied within a refinement of the model in which they were originally captured. The language elements that are addressed by these transformations are as follows:

– Persistent and transient triggers;
– While, repeat and combination loops;
– Thread merges;
– Thread splits;
– Partial joins; and
– Tasks directly linked to tasks.

Each of these transformations is described in the following section, first via a high-level graphical illustration of its operation and then more completely using set theory. The order in which the transformations are applied is material as later transformations assume that some structural modifications enacted by earlier transformations have been completed. For this reason, the transformations should be applied in the order presented in this section. Once a complete *new*YAWL specification has been appropriately simplified through the application of these transformations, it is known as a core *new*YAWL specification. Additionally, in order to preserve the integrity of the specification being transformed, the transformations are not applied to all constructs in a specification simultaneously but rather on an incremental (i.e. item-by-item) basis. The transformations are applied iteratively for a given specification until all constructs have been appropriately dealt with. Note that in the interest of brevity, for all transformations we only describe changes to the elements in each specification. Elements that remain unchanged are omitted. The final core *new*YAWL specification is obtained by aggregating the latest version of each of the models for the *new*YAWL specification, *new*YAWL-nets, Data passing model, Work distribution model and Organizational model once all transformations have been applied.

The transformations presented are *semantics-defining* and give an operational meaning to the higher-level *new*YAWL constructs embodied in a complete *new*YAWL specification by defining their function in terms of core *new*YAWL constructs. As such, they cannot be seen as *equivalence-preserving* and it is important to note that whilst some of the transformations appear to change the moment of choice for some constructs in a complete *new*YAWL specification, in fact there can be no direct meaning ascribed to such a specification and it is only when it is appropriately transformed to a core *new*YAWL specification that the actual moment of choice for these constructs is revealed.

4.2.1 Persistent and transient triggers

Persistent and transient triggers that are defined for tasks in *new*YAWL specifications are operationalized in core *new*YAWL specifications as specific tasks that identify when the trigger has been received. Figure 16 illustrates that manner in which a trigger is incorporated into a *new*YAWL specification. In essence, the trigger *trig* associated with task A becomes the task A_{T1}. This task is only enabled when the process instance is running (signified by a token in place C_T) and triggers received for the task are collected in condition C_A. Any relevant join associated with task A is moved into a dedicated task which proceeds it. The enablement of task A then becomes an AND-join based on the normal thread of control and the condition that collects tokens (C_A). The actual transformation for persistent and transient triggers is identical except that transient triggers have the additional requirement that a means of deleting the trigger is required if it cannot be utilized immediately. This is provided

via reset task A_{T2} as illustrated in Figure 16 which is associated with the condition which collects tokens for triggers that are received. Once a trigger has been received, there is a race condition between the enablement of the task being (A) triggered and the reset task (A_{T2}) thus ensuring that any tokens that do not immediately trigger task A are discarded[7].

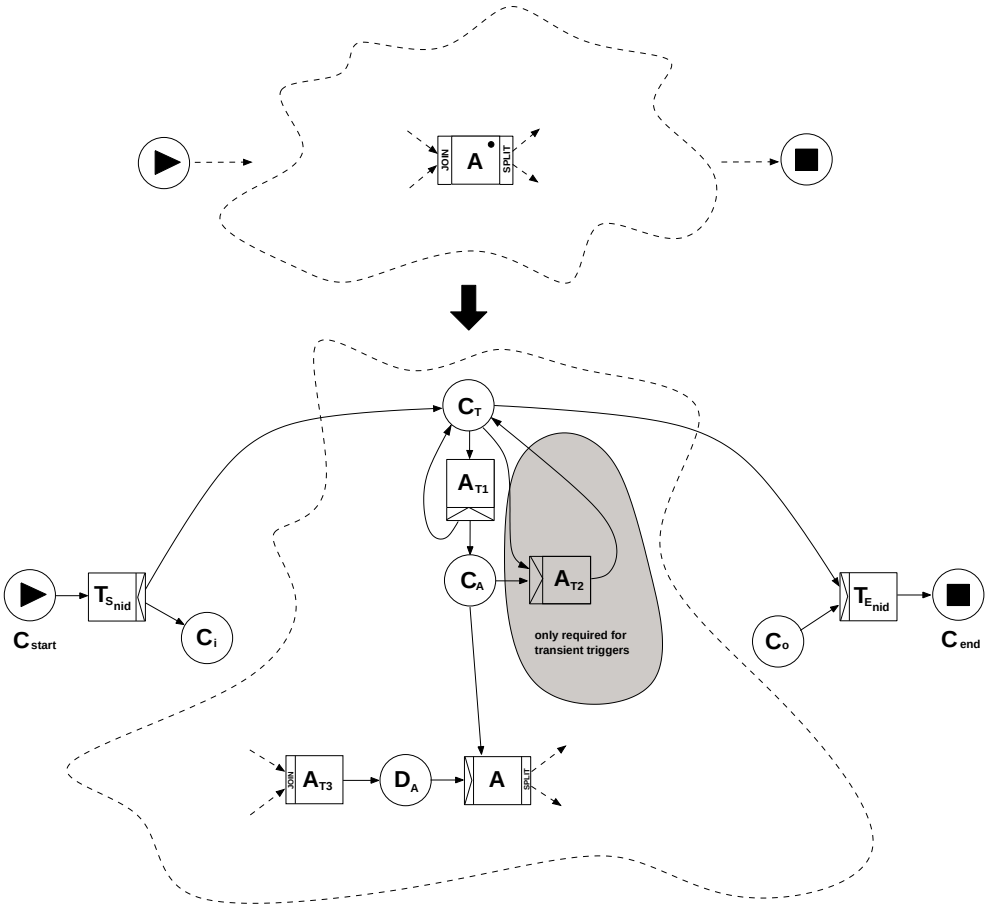


Figure 16: Persistent and transient trigger transformation

As the transformation involves the addition of both tasks and conditions as well as changes to the flow relation, it necessitates changes to the *new*YAWL specification and to each *new*YAWL-net which includes a trigger. There are also minor changes to the Data passing model to provide task A_{T3} with access to the data provided by the input parameters in order to allow the task precondition (if any) to be evaluated[8]. There are also changes to the Work distribution model to ensure that any tasks added by the transformation are automatic (i.e. they do not need to be allocated to a user for execution). The *new*YAWL specification is amended to include any new tasks

[7]In practice, this task would have a delay associated with its enablement to ensure that task A has first option to utilize any tokens that may be delivered to condition C_A before they are discarded.

[8]Note that where a task is split into several parts by a transformation, the precondition is evaluated both for the first task into which it is split and it is also retained for the original task. The postcondition is retained for the original task and replicated for the last task into which the task is split. Locking requirements and parameters are also replicated as required.

added during steps 1 and 2 of the transformation and also to add them to any scopes to which they might apply.

The transformation has five steps. First of all, the general extensions are made to each *new*YAWL-net that includes triggers. These extensions accommodate all trigger transformations in a given *new*YAWL-net and hence only need to be made once. The next step is to transform each trigger into core *new*YAWL constructs. The final three steps amend the Data passing model, Work distribution model and *new*YAWL specification. The Organizational model is unchanged.

Step 1: Initial transformations for *new*YAWL-net nid with triggers

precondition: $dom(Trig) \neq \emptyset$

$C' = C \cup \{C_T, C_{start}, C_{end}\}$
$i' = C_{start}$
$o' = C_{end}$
$T' = T \cup \{T_{S_{nid}}, T_{E_{nid}}\}$
$T'_A = T_A \cup \{T_{S_{nid}}, T_{E_{nid}}\}$
$F' = F \cup \{(C_{start}, T_{S_{nid}}), (T_{S_{nid}}, i), (T_{S_{nid}}, C_T), (C_T, T_{E_{nid}}), (o, T_{E_{nid}}), (T_{E_{nid}}, C_{end})\}$
$Join' = Join \cup \{(T_{E_{nid}}, AND)\}$
$Split' = Split \cup \{(T_{S_{nid}}, AND)\}$

Step 2: Transformations for *new*YAWL-net nid′ (resulting from step 1) to replace individual triggers with core *new*YAWL constructs

Let $t \in dom(Trig')$, transforming t in net nid' leads to the following changes:

$$
\begin{aligned}
C'' &= C' \cup \{C_t, D_t\} \\
T'' &= T' \cup \{t_{T1}, t_{T3}\} \cup \{x_{T2} \mid x \in dom(Trig') \backslash Persist' \wedge x = t\} \\
T''_A &= T'_A \cup \{t_{T1}, t_{T3}\} \cup \{x_{T2} \mid x \in dom(Trig') \backslash Persist' \wedge x = t\} \\
F'' &= (F' \backslash \{(x,t) \mid x \in {\bullet t}\}) \\
&\quad \cup \{(x, t_{T3}) \mid x \in {\bullet t}\} \\
&\quad \cup \{(C_T, t_{T1}), (t_{T1}, C_T), (t_{T1}, C_t), (C_t, t), (t_{T3}, D_t), (D_t, t)\} \\
&\quad \cup \{(C_x, x_{T2}) \mid x \in dom(Trig') \backslash Persist' \wedge x = t\} \\
&\quad \cup \{(C_T, x_{T2}) \mid x \in dom(Trig') \backslash Persist' \wedge x = t\} \\
&\quad \cup \{(x_{T2}, C_T) \mid x \in dom(Trig') \backslash Persist' \wedge x = t\} \\
Join'' &= (Join' \backslash \{(x, Join'(x)) \mid x \in dom(Join') \wedge x = t\}) \\
&\quad \cup \{(x_{T3}, Join'(x)) \mid x \in dom(Join') \wedge x = t\} \\
&\quad \cup \{(t, AND)\} \\
&\quad \cup \{(x_{T2}, AND) \mid x \in dom(Trig') \backslash Persist' \wedge x = t\} \\
Split'' &= Split' \cup \{(t_{T1}, AND)\} \\
Rem'' &= (Rem' \backslash \{(x, Rem'(x)) \mid x \in dom(Rem') \wedge t \in Rem'(x)\}) \\
&\quad \cup \{(x, Rem'(x) \cup \{t_{T3}, D_t\}) \mid x \in dom(Rem') \wedge t \in Rem'(x)\})
\end{aligned}
$$

$$Block'' = ((Block' \setminus \{(x, Block'(x)) \mid x \in dom(Block') \wedge x = t\}) \setminus \{(x, Block'(x)) \mid x \in dom(Block') \wedge t \in Block'(x)\}) \cup \{(x_{T3}, Block'(x)) \mid x \in dom(Block') \wedge x = t\} \cup \{(x, ((Block'(x) \setminus \{t\}) \cup \{t_{T3}\})) \mid x \in dom(Block') \wedge t \in Block'(x)\}$$

$$Lock'' = Lock' \cup \{(x_{T3}, Lock'(x)) \mid x \in dom(Lock') \wedge x = t\}$$

$$Thresh'' = (Thresh' \setminus \{(x, Thresh'(x)) \mid x \in dom(Thresh') \wedge x = t\}) \cup \{(x_{T3}, Thresh'(x)) \mid x \in dom(Thresh') \wedge x = t\}$$

$$ThreadIn'' = (ThreadIn' \setminus \{(x, ThreadIn'(x)) \mid x \in dom(ThreadIn') \wedge x = t\}) \cup \{(x_{T3}, ThreadIn'(x)) \mid x \in dom(ThreadIn') \wedge x = t\}$$

$$Pre'' = Pre' \cup \{(x_{T3}, Pre'(x)) \mid x \in dom(Pre') \wedge x = t\}$$

$$Trig'' = \emptyset$$

$$Persist'' = \emptyset$$

Step 3: Transformations for Data passing model for *new*YAWL-net nid

Let $t \in dom(Trig)$, transforming t in net nid leads to the following changes:

$$InPar' = InPar \cup \{((x_{T3}, v), e) \mid x \in dom(Trig) \wedge ((x, v), e) \in InPar \wedge x = t\}$$

$$OptInPar' = OptInPar \cup \{(x_{T3}, v) \mid x \in dom(Trig) \wedge (x, v) \in OptInPar \wedge x = t\}$$

Step 4: Transformations for Work distribution model for *new*YAWL-net nid

$$Auto' = Auto \cup (T''_{nid} \setminus T_{nid});$$

Step 5: Transformations for *new*YAWL specification

$$TaskID' = \bigcup_{n \in NetID} T''_n$$

$$STmap' = (STmap \setminus \{(s, STmap(s)) \mid s \in dom(STmap) \wedge STmap(s) \cap dom(Trig) \neq \emptyset\}) \cup \{(s, STmap(s) \cup \{t_{T3}\}) \mid s \in dom(STmap) \wedge t \in STmap(s) \cap dom(Trig)\}$$

4.2.2 Loops

Loops in *new*YAWL are based on PreTest and PostTest conditions associated with individual tasks. Depending on the combination of PreTest and PostTest associated with a task and whether it has any join or split behaviour, there are a series of alternate transformations that can be made in order to remove specific reliance on loop constructs in order to achieve task iteration. The various transformations are summarized in Figure 17. In essence, they involve varying the process model to construct looping structures with entry and/or exit conditions which are based on the conditions identified for the PreTests and PostTests[9].

[9]Note that in the diagrams, ∼PreTest(A) is assumed to be the logical negation of PreTest(A).

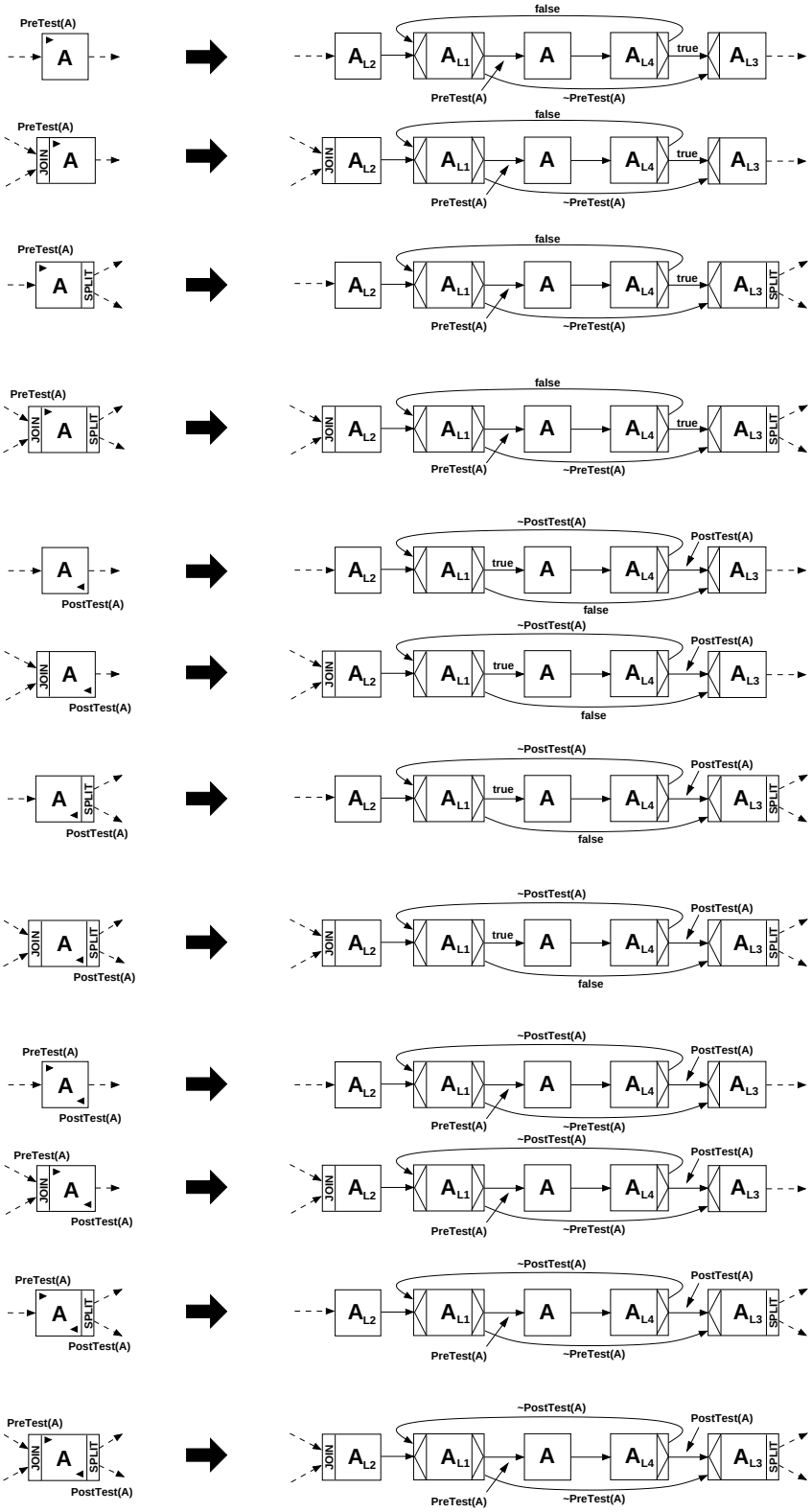


Figure 17: Transformation of pre-test and post-test loops

The specific transformations required are detailed below. They necessitate changes to each of the individual *new*YAWL-nets that contain loops to separate the join and split behaviour which may be associated with tasks possessing PreTest and/or PostTest conditions from the looping behaviour. As these transformations potentially necessitate the addition of new tasks, there are also changes to the Data passing model to ensure that any new tasks have access to the same data elements as the task from which they were derived and also to the *new*YAWL specification. Any new tasks are automatic hence there are also amendments to the Work distribution model.

Step 1: Transformations for *new*YAWL-net nid

Let $t \in dom(PreTest) \cup dom(PostTest)$, transforming t in net *nid* lead to the following changes:

$$
\begin{aligned}
T' &= T \cup \{t_{L1}, t_{L2}, t_{L3}, t_{L4}\} \\
T'_A &= T_A \cup \{t_{L1}, t_{L2}, t_{L3}, t_{L4}\} \\
F' &= (F \setminus \{(c,t) \mid c \in {\bullet t}\} \cup \{(t,c) \mid c \in t{\bullet}\}) \\
&\quad \cup \{(c, t_{L2}) \mid c \in {\bullet t}\} \cup \{(t_{L3}, c) \mid c \in t{\bullet}\} \\
&\quad \cup \{(t_{L2}, t_{L1}), (t_{L1}, t), (t, t_{L4}), (t_{L1}, t_{L3}), (t_{L4}, t_{L1}), (t_{L4}, t_{L3})\} \\
Split' &= (Split \setminus \{(x, Split(x)) \mid x \in dom(Split) \wedge x = t\}) \\
&\quad \cup \{(x_{L3}, Split(x)) \mid x \in dom(Split) \wedge x = t\} \\
&\quad \cup \{(t_{L1}, XOR), (t_{L4}, XOR)\} \\
Join' &= (Join \setminus \{(x, Join(x)) \mid x \in dom(Join) \wedge x = t\}) \\
&\quad \cup \{(x_{L2}, Join(x)) \mid x \in dom(Join) \wedge x = t\} \\
&\quad \cup \{(t_{L1}, XOR), (t_{L3}, XOR)\} \\
Default' &= (Default \setminus \{(x, Default(x)) \mid x \in dom(Default) \\
&\qquad \wedge Split(x) \in \{OR, XOR\} \wedge x = t\}) \\
&\quad \cup \{(x_{L3}, Default(x)) \mid x \in dom(Default) \\
&\qquad \wedge Split(x) \in \{OR, XOR\} \wedge x = t\} \\
&\quad \cup \{(t_{L1}, t), (t_{L4}, t_{L3})\} \\
<'_{XOR} &= (<_{XOR} \setminus \{(x, <^x_{XOR}) \mid x \in dom(Split) \wedge Split(x) = XOR \wedge x = t\}) \\
&\quad \cup \{(x_{L3}, <^x_{XOR}) \mid x \in dom(Split) \wedge Split(x) = XOR \wedge x = t\} \\
&\quad \cup \{(t_{L1}, \{(t, t_{L3})\}) \} \cup \{(t_{L4}, \{(t_{L1}, t_{L3})\})\} \\
Rem' &= ((Rem \setminus \{(x, Rem(x)) \mid x \in dom(Rem) \wedge x = t\}) \\
&\quad \setminus \{(x, Rem(x)) \mid x \in dom(Rem) \wedge t \in Rem(x)\}) \\
&\quad \cup \{(x_{L3}, Rem(x)) \mid x \in dom(Rem) \wedge x = t\} \\
&\quad \cup \{(x, Rem(x) \cup \{t_{L1}, t_{L2}, t_{L3}, t_{L4}\}) \mid x \in dom(Rem) \wedge t \in Rem(x)\} \\
Comp' &= (Comp \setminus \{(x, Comp(x)) \mid x \in dom(Comp) \wedge x = t\}) \\
&\quad \cup \{(x_{L3}, Comp(x)) \mid x \in dom(Comp) \wedge x = t\}
\end{aligned}
$$

$$
\begin{aligned}
Block' = ((&Block \setminus \{(x, Block(x)) \mid x \in dom(Block) \wedge x = t\}) \\
&\setminus \{(x, Block(x)) \mid x \in dom(Block) \wedge t \in Block(x)\}) \\
&\cup \{(x_{L2}, Block(x)) \mid x \in dom(Block) \wedge x = t\} \\
&\cup \{(x, ((Block(x) \setminus \{t\}) \cup \{t_{L2}\})) \mid x \in dom(Block) \wedge t \in Block(x)\}
\end{aligned}
$$

$$
\begin{aligned}
Disable' = (&Disable \setminus \{(x, Disable(x)) \mid x \in dom(Disable) \wedge x = t\}) \\
&\cup \{(x_{L3}, Disable(x)) \mid x \in dom(Disable) \wedge x = t\}
\end{aligned}
$$

$$
\begin{aligned}
Lock' = Lock &\cup \{(x_{L1}, Lock(x)) \mid x \in dom(Lock) \wedge x = t\} \\
&\cup \{(x_{L2}, Lock(x)) \mid x \in dom(Lock) \wedge x = t\} \\
&\cup \{(x_{L3}, Lock(x)) \mid x \in dom(Lock) \wedge x = t\} \\
&\cup \{(x_{L4}, Lock(x)) \mid x \in dom(Lock) \wedge x = t\}
\end{aligned}
$$

$$
\begin{aligned}
Thresh' = (&Thresh \setminus \{(x, Thresh(x)) \mid x \in dom(Thresh) \wedge x = t\}) \\
&\cup \{(x_{L2}, Thresh(x)) \mid x \in dom(Thresh) \wedge x = t\}
\end{aligned}
$$

$$
\begin{aligned}
ThreadIn' = (&ThreadIn \setminus \{(x, ThreadIn(x)) \mid x \in dom(ThreadIn) \wedge x = t\}) \\
&\cup \{(x_{L2}, ThreadIn(x)) \mid x \in dom(ThreadIn) \wedge x = t\}
\end{aligned}
$$

$$
\begin{aligned}
ThreadOut' = (&ThreadOut \setminus \{(x, ThreadOut(x)) \mid x \in dom(ThreadOut) \wedge x = t\}) \\
&\cup \{(x_{L3}, ThreadOut(x)) \mid x \in dom(ThreadOut) \wedge x = t\}
\end{aligned}
$$

$$
\begin{aligned}
ArcCond' = (&ArcCond \setminus \{((x, c), ArcCond(x, c)) \mid x \in dom(Split) \\
&\qquad \wedge Split(x) \in \{OR, XOR\} \wedge c \in x \bullet \wedge x = t\}) \\
&\cup \{((x_{L3}, c), ArcCond(x, c)) \mid x \in dom(Split)\} \\
&\qquad \wedge Split(x) \in \{OR, XOR \wedge c \in x \bullet \wedge x = t\} \\
&\cup \{((x_{L1}, x), PreTest(x)) \mid x \in dom(PreTest) \wedge x = t\} \\
&\cup \{((x_{L1}, x), true) \mid x \notin dom(PreTest) \wedge x = t\} \\
&\cup \{((x_{L1}, x_{L3}), \neg PreTest(x)) \mid x \in dom(PreTest) \wedge x = t\} \\
&\cup \{((x_{L1}, x_{L3}), false) \mid x \notin dom(PreTest) \wedge x = t\} \\
&\cup \{((x_{L4}, x_{L3}), PostTest(x)) \mid x \in dom(PostTest) \wedge x = t\} \\
&\cup \{((x_{L4}, x_{L3}), true) \mid x \notin dom(PostTest) \wedge x = t\} \\
&\cup \{((x_{L4}, x_{L1}), \neg PostTest(x)) \mid x \in dom(PostTest) \wedge x = t\} \\
&\cup \{((x_{L4}, x_{L1}), false) \mid x \notin dom(PostTest) \wedge x = t\}
\end{aligned}
$$

$$Pre' = Pre \cup \{(x_{L2}, Pre(x)) \mid x \in dom(Pre) \wedge x = t\}$$

$$Post' = Post \cup \{(x_{L3}, Post(x)) \mid x \in dom(Post) \wedge x = t\}$$

$$PreTest' = PreTest \setminus \{(x, PreTest(x)) \mid x \in dom(PreTest) \wedge x = t\}$$

$$PostTest' = PostTest \setminus \{(x, PostTest(x)) \mid x \in dom(PostTest) \wedge x = t\}$$

Step 2: Transformations for Data passing model for *new*YAWL-net nid

Let $t \in dom(PreTest) \cup dom(PostTest)$, transforming t in net nid leads to the following changes:

$$\begin{aligned}
InPar' = InPar \cup \{((x_{L1}, v), e) \mid x \in dom(PreTest) \cup dom(Lock) \\
\wedge ((x, v), e) \in InPar \wedge x = t\} \\
\cup \{((x_{L2}, v), e) \mid x \in dom(Pre) \cup dom(Lock) \\
\wedge ((x, v), e) \in InPar \wedge x = t\} \\
\cup \{((x_{L3}, v), e) \mid x \in dom(Split) \cup dom(Post) \cup dom(Lock) \\
\wedge ((x, v), e) \in InPar \wedge x = t\} \\
\cup \{((x_{L4}, v), e) \mid x \in dom(PostTest) \cup dom(Lock) \\
\wedge ((x, v), e) \in InPar \wedge x = t\}
\end{aligned}$$

$$\begin{aligned}
OptInPar' = OptInPar \cup \{(x_{L1}, v) \mid x \in dom(PreTest) \cup dom(Lock) \\
\wedge (x, v) \in OptInPar \wedge x = t\} \\
\cup \{(x_{L2}, v) \mid x \in dom(Pre) \cup dom(Lock) \\
\wedge (x, v) \in OptInPar \wedge x = t\} \\
\cup \{(x_{L3}, v) \mid x \in dom(Split) \cup dom(Post) \cup dom(Lock) \\
\wedge (x, v) \in OptInPar \wedge x = t\} \\
\cup \{(x_{L4}, v) \mid x \in dom(PostTest) \cup dom(Lock) \\
\wedge (x, v) \in OptInPar \wedge x = t\}
\end{aligned}$$

Step 3: Transformations for Work distribution model for *new*YAWL-net nid

$Auto' = Auto \cup (T'_{nid} \setminus T_{nid})$;

Step 4: Transformations for *new*YAWL specification

$TaskID' = \bigcup_{n \in NetID} T'_n$

$$\begin{aligned}
STmap' = (STmap \setminus \{(s, STmap(s)) \mid s \in dom(STmap) \\
\wedge STmap(s) \cap (dom(PreTest) \cup dom(PostTest)) \neq \emptyset\}) \\
\cup \{(s, STmap(s) \cup \{t_{L1}, t_{L2}, t_{L3}, t_{L4}\}) \mid s \in dom(STmap) \\
\wedge t \in STmap(s) \cap (dom(PreTest) \cup dom(PostTest))\}
\end{aligned}$$

4.2.3 Thread merge

The thread merge construct coalesces a specified number of execution threads from the same process instance. The transformation for this construct is illustrated in Figure 18. It essentially involves the creation of an AND-join precondition to the construct that can only fire when the required number of incoming tokens (i.e. incoming execution threads) have been received and there is one of the tokens in each of the conditions $C_{A_{M1}} ... C_{A_{Mn}}$ enabling the AND-join for task A to fire. We assume that there is a notion of "fairness" that applies to the model that will eventually result in the tokens being distributed across the input conditions in this way.

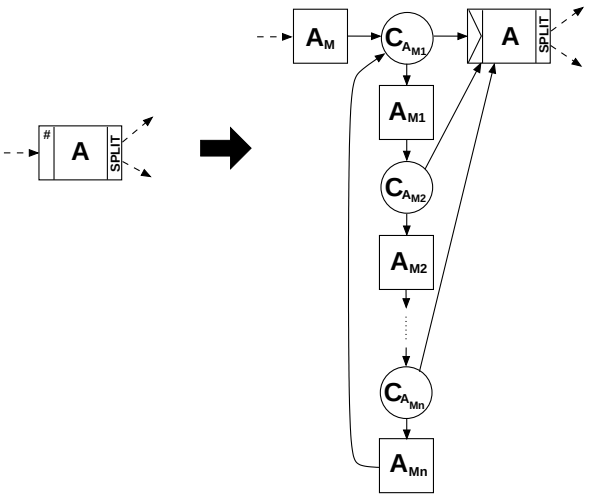


Figure 18: Transformation of thread merge construct

There are three steps involved in this transformation. First the thread merge tasks in each *new*YAWL-net are transformed into core *new*YAWL-net constructs, then the associated Work distribution models are transformed to ensure all added tasks are automatic and finally any new tasks are added to the *new*YAWL specification.

Step 1: Transformations for *new*YAWL-net nid to replace individual thread merges with core *new*YAWL constructs

Let $t \in dom(ThreadIn)$, transforming t in net nid leads to the following changes:

$$
\begin{aligned}
C' &= C \cup \{C_{tMi} \mid 1 \leq i \leq ThreadIn(t)\} \\
T' &= T \cup \{t_M\} \cup \{t_{Mi} \mid 1 \leq i \leq ThreadIn(t)\} \\
T'_A &= T_A \cup \{t_M\} \cup \{t_{Mi} \mid 1 \leq i \leq ThreadIn(t)\} \\
F' &= (F \backslash \{(x,t) \mid x \in {\bullet t}\}) \\
&\quad \cup \{(x, t_M) \mid x \in {\bullet t}\} \\
&\quad \cup \{(t_M, C_{t_{M1}})\} \\
&\quad \cup \{(C_{t_{Mi}}, t_{Mi}) \mid 1 \leq i \leq ThreadIn(t)\} \\
&\quad \cup \{(t_{M(i-1)}, C_{t_{Mi}}) \mid 2 \leq i \leq ThreadIn(t)\} \\
&\quad \cup \{(C_{t_{Mi}}, t) \mid 1 \leq i \leq ThreadIn(t)\} \\
&\quad \cup \{(t_{Mn}, C_{t_{M1}}) \mid n = ThreadIn(t)\} \\
Join' &= (Join \setminus \{(t, THREAD)\}) \cup \{(t, AND)\} \\
ThreadIn' &= ThreadIn \setminus \{(x, ThreadIn(x)) \mid x \in dom(ThreadIn) \wedge x = t\}
\end{aligned}
$$

Step 2: Transformations for Work distribution model for *new*YAWL-net nid

$Auto' = Auto \cup (T'_{nid} \setminus T_{nid})$;

Step 3: Transformations for *new*YAWL specification

$TaskID' = \bigcup_{n \in NetID} T'_n$

4.2.4 Thread split

The thread split construct diverges a single thread of execution into multiple concurrent threads, which initially flow through the same branch. Figure 19 illustrates how the transformation for this construct operates. Essentially it creates an AND-split in place of the thread split which has the required number of outgoing branches. These branches are subsequently joined at a common place (C_{AS}) and the associated tokens are passed on to subsequent tasks by task A_S on an as required basis.

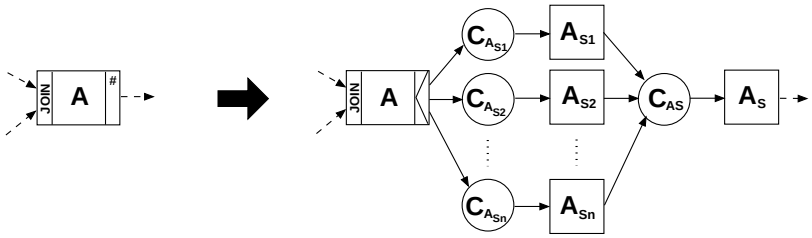


Figure 19: Transformation of thread split construct

The transformations associated with this proceed in three steps. First the thread split tasks in each *new*YAWL-net are transformed into Core *new*YAWL-net constructs, then any newly introduced tasks also added as automatic tasks to the Work distribution model associated with the *new*YAWL-net and finally any new tasks are added to the *new*YAWL specification.

Step 1: Transformations for *new*YAWL-net nid replace individual thread splits with core *new*YAWL constructs

Let $t \in dom(ThreadOut)$, transforming t in net nid leads to the following changes:

$$
\begin{aligned}
C' &= C \cup \{C_{t_S}\} \cup \{C_{t_{Si}} \mid 1 \leq i \leq ThreadOut(t)\} \\
T' &= T \cup \{t_S\} \cup \{t_{Si} \mid 1 \leq i \leq ThreadOut(t)\} \\
T'_A &= T_A \cup \{t_S\} \cup \{t_{Si} \mid 1 \leq i \leq ThreadOut(t)\} \\
F' &= (F \setminus \{(t,x) \mid x \in t\bullet\}) \\
&\quad \cup \{(t_S, x) \mid x \in t\bullet\} \\
&\quad \cup \{(C_{t_S}, t_S)\} \\
&\quad \cup \{(t, C_{t_{Si}}) \mid 1 \leq i \leq ThreadOut(t)\} \\
&\quad \cup \{(C_{t_{Si}}, t_{Si}) \mid 1 \leq i \leq ThreadOut(t)\} \\
&\quad \cup \{(t_{Si}, C_{t_S}) \mid 1 \leq i \leq ThreadOut(t)\}
\end{aligned}
$$

$Split' = (Split \backslash \{(t, THREAD)\}) \cup \{(t, AND)\}$

$ThreadOut' = ThreadOut \backslash \{(x, ThreadOut(x)) \mid x \in dom(ThreadOut) \wedge x = t\}$

Step 2: Transformations for Work distribution model for *new*YAWL-net nid

$Auto' = Auto \cup (T'_{nid} \setminus T_{nid})$;

The final step in the transformation process is ensure that all tasks that have been added are also included in the *new*YAWL specification.

Step 3: Transformations for *new*YAWL specification

$TaskID' = \bigcup_{n \in NetID} T'_n$

4.2.5 Partial join

The partial join construct has probably the most complex series of transformations associated with it. It is illustrated diagrammatically in Figure 20 and essentially involves replacing the partial join construct with the set of all possible AND-joins that would enable the set of input branches to trigger a join when the required threshold for the join was reached. For example where the partial join had four inputs and two were required for the join to proceed, then the partial join would be replaced by six two-input AND-joins, each of which is linked to one combination of incoming branches that could trigger the join as illustrated by the tasks prefixed A_J^z. As this is a combinatorial function, there are $\begin{pmatrix} m \\ n \end{pmatrix}$ of these tasks. Similarly, there are the same number of tasks prefixed A_R^z which reset the join and allow it to fire again. Only when one of the A_J^z joins has fired can the actual task A be enabled.

A feature of the partial join is the blocking link, which allows specified tasks to serve as gateways into the *blocking region* for a task. The blocking region is a group of preceding tasks and their associated branches where only one thread of execution should be active on each incoming branch to the partial join for each triggering of the join. The *block* function identifies the set of tasks that constitute the blocking region for a given task. In Figure 20, task X has a blocking link associated with it. Once task A has fired, task X is prevented from being enabled until inputs have been received on all branches to task A and it has reset. In the transformed *new*YAWL-net, each "blocking task" has a place associated with it (e.g. $C_{X_{B1}}$). Initially it has a token in it. This is removed when one of the permutations of input branches allows the (partial) join to fire and only when the partial join has reset, is the token replaced in the place allowing the "blocking task" to fire again.

The transformation proceeds in six stages. First, initialization conditions are inserted into any *new*YAWL-net that contains blocking tasks associated with partial joins. These conditions allow the blocking tasks to be enabled in a given net providing the associated partial join has not been enabled.

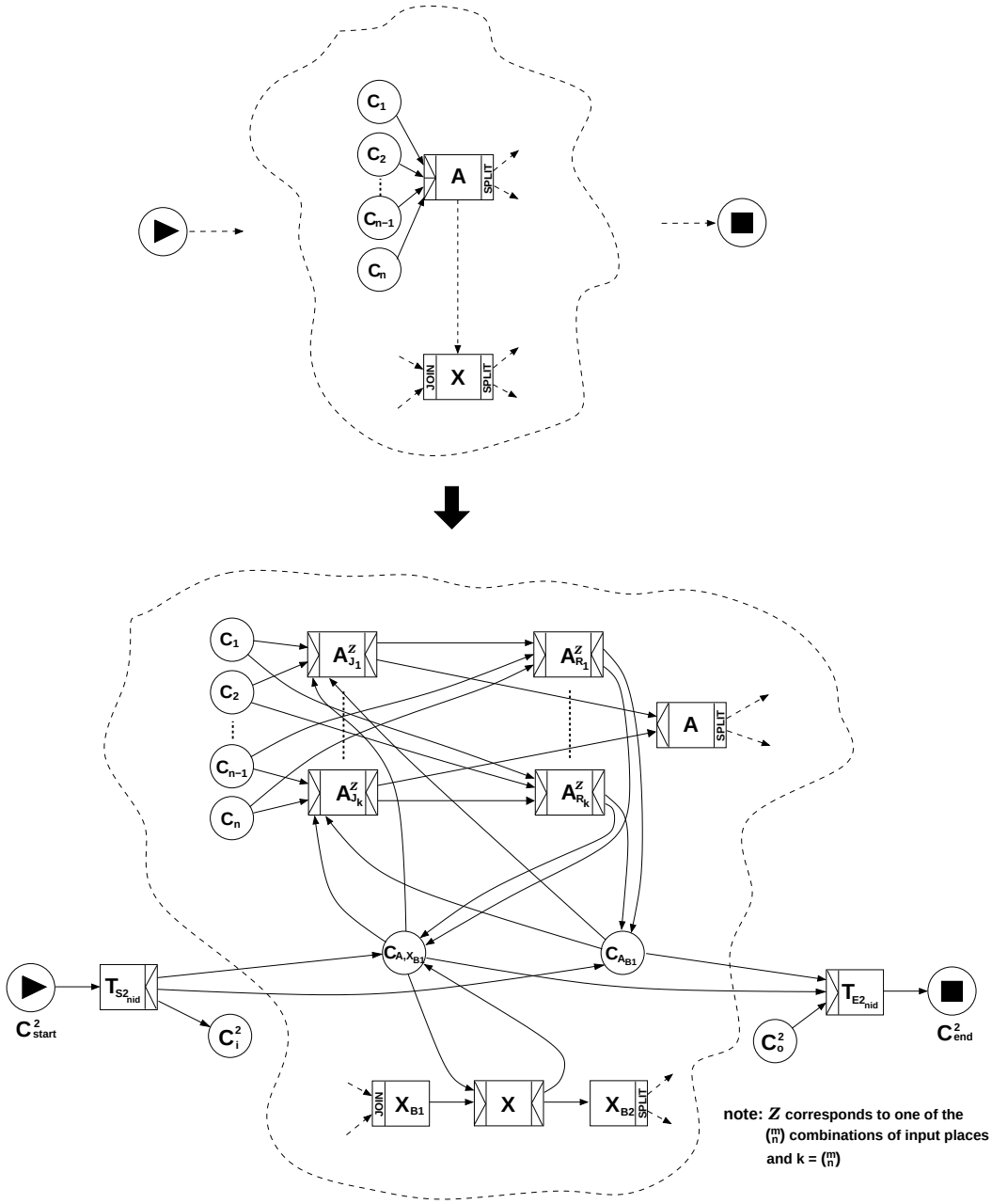


Figure 20: Transformation of partial join construct

Step 1: Initial transformation for *new*YAWL-net nid with partial joins

precondition: $ran(Block_{nid}) \neq \emptyset$

$C' = C \cup \{C^2_{start}, C^2_{end}\}$
$i' = C^2_{start}$
$o' = C^2_{end}$
$T' = T \cup \{T_{S2_{nid}}, T_{E2_{nid}}\}$

$$F' = F \cup \{(C^2_{start}, T_{S2_{nid}}), (T_{S2_{nid}}, i), (o, T_{E2_{nid}}), (T_{E2_{nid}}, C^2_{end})\}$$
$$\cup \{(T_{S2_{nid}}, C_{t_{B1}}) \mid t \in dom(Thresh)\}$$
$$\cup \{(C_{t_{B1}}, T_{E2_{nid}}) \mid t \in dom(Thresh)\}$$
$$Split' = Split \cup \{(T_{S_{nid}}, AND)\}$$
$$Join' = Join \cup \{(T_{E_{nid}}, AND)\}$$

The next step is to transform each of the partial joins in a given net. These need to be undertaken incrementally (on a task-by-task basis) and involve the insertion of a series of AND-join constructs ($A^{\mathcal{Z}}_{J1}...A^{\mathcal{Z}}_{Jk}$) such that a distinct AND-join is added for each combination of incoming paths that could enable the partial join. Associated with each AND-join is another AND-join ($A^{\mathcal{Z}}_{R1}...A^{\mathcal{Z}}_{Rk}$) that allows the partial join to be reset when execution threads have been received on the remaining incoming branches. There is also a condition ($C_{A_{B1}}$) inserted for each partial join to ensure that each of the reset tasks ($A^{\mathcal{Z}}_{R1}...A^{\mathcal{Z}}_{Rk}$) are on a path from the start to end condition in the *new*YAWL-net.

Step 2: Transformations for *new*YAWL-net nid' (from step 1) to replace individual partial joins with core *new*YAWL constructs

Let $t \in dom(Thresh')$, transforming t in net nid' leads to the following changes:

$$C'' = C' \cup \{C_{t_{B1}}\}$$
$$T'' = T' \cup \{t^{\mathcal{C}}_J \mid \mathcal{C} \in \mathbb{P}(\bullet t) \wedge |\mathcal{C}| = Thresh'(t)\}$$
$$\cup \{t^{\mathcal{D}}_R \mid \mathcal{D} \in \mathbb{P}(\bullet t) \wedge |\mathcal{D}| = |\bullet t| - Thresh'(t)\}$$
$$F'' = (F' \setminus \{(c,t) \mid c \in \bullet t\})$$
$$\cup \{(c, t^{\mathcal{C}}_J) \mid \mathcal{C} \in \mathbb{P}(\bullet t) \wedge |\mathcal{C}| = Thresh'(t) \wedge c \in \mathcal{C}\}$$
$$\cup \{(c, t^{\mathcal{D}}_R) \mid \mathcal{D} \in \mathbb{P}(\bullet t) \wedge |\mathcal{D}| = |\bullet t| - Thresh'(t) \wedge c \in \mathcal{D}\}$$
$$\cup \{(t^{\mathcal{C}}_J, t^{\mathcal{D}}_R) \mid \mathcal{C} \in \mathbb{P}(\bullet t) \wedge |\mathcal{C}| = Thresh'(t) \wedge \mathcal{D} = \bullet t \setminus \mathcal{C}\}$$
$$\cup \{(t^{\mathcal{C}}_J, t) \mid \mathcal{C} \in \mathbb{P}(\bullet t) \wedge |\mathcal{C}| = Thresh'(t)\}$$
$$\cup \{(T_{S2_{nid}}, C_{t_{B1}}), (C_{t_{B1}}, T_{E2_{nid}})\}$$
$$\cup \{(C_{t_{B1}}, t^{\mathcal{C}}_J) \mid \mathcal{C} \in \mathbb{P}(\bullet t) \wedge |\mathcal{C}| = Thresh'(t)\}$$
$$\cup \{(t^{\mathcal{D}}_R, C_{t_{B1}}) \mid \mathcal{D} \in \mathbb{P}(\bullet t) \wedge |\mathcal{D}| = |\bullet t| - Thresh'(t)\}$$
$$Split'' = Split' \cup \{(t^{\mathcal{C}}_J, AND) \mid \mathcal{C} \in \mathbb{P}(\bullet t) \wedge |\mathcal{C}| = Thresh'(t)\}$$
$$\cup \{(x^{\mathcal{D}}_R, AND) \mid x \in dom(Thresh') \cap dom(Block') \wedge \mathcal{D} \in \mathbb{P}(\bullet x)$$
$$\wedge |\mathcal{D}| = |\bullet x| - Thresh'(x) \wedge x = t\}$$
$$Join'' = (Join' \setminus \{(t, PJOIN)\})$$
$$\cup \{(t, XOR)\}$$
$$\cup \{(t^{\mathcal{C}}_J, AND) \mid \mathcal{C} \in \mathbb{P}(\bullet t) \wedge |\mathcal{C}| = Thresh'(t)\}$$
$$\cup \{(t^{\mathcal{D}}_R, AND) \mid \mathcal{D} \in \mathbb{P}(\bullet t) \wedge |\mathcal{D}| = |\bullet t| - Thresh'(t)\}$$
$$Block'' = (Block' \setminus \{(x, Block'(x)) \mid x \in dom(Block') \wedge t \in Block'(x)\})$$
$$\cup \{(x, (Block'(x) \setminus \{t\}) \cup \{t^{\mathcal{C}}_J\}) \mid x \in dom(Block') \wedge t \in Block'(x)$$
$$\wedge \mathcal{C} \in \mathbb{P}(\bullet t) \wedge |\mathcal{C}| = Thresh'(t)\}$$

$$Rem'' = (Rem' \setminus \{(x, Rem'(x)) \mid x \in dom(Rem') \wedge t \in Rem'(x)\}) \\ \cup \{(x, Rem'(x) \cup \{t_J^{\mathcal{C}}\} \cup \{t_R^{\mathcal{D}}\}) \mid \mathcal{C} \in \mathbb{P}(\bullet t) \wedge |\mathcal{C}| = Thresh'(t) \\ \wedge \mathcal{D} \in \mathbb{P}(\bullet t) \wedge |\mathcal{D}| = |\bullet t| - Thresh'(t) \wedge t \in Rem'(x)\}$$

$$Lock'' = Lock' \cup \{(t_J^{\mathcal{C}}, Lock'(t)) \mid \mathcal{C} \in \mathbb{P}(\bullet t) \wedge |\mathcal{C}| = Thresh'(t) \wedge t \in dom(Lock)\}$$

$$Pre'' = Pre' \cup \{(x_J^{\mathcal{C}}, Pre'(x)) \mid x \in dom(Pre') \wedge \mathcal{C} \in \mathbb{P}(\bullet x) \\ \wedge |\mathcal{C}| = Thresh'(x) \wedge x = t\}$$

$$Thresh'' = Thresh' \setminus \{(t, Thresh'(t))\}$$

The third step is to replace each blocking task with core *new*YAWL constructs. As each of these tasks may have joins and/or splits associated with them, it is necessary to move these to preceding and subsequent tasks in order to ensure that they are evaluated separately from the blocking action associated with each task under consideration. Similarly other constructs associated with each blocking task (e.g. arc conditions, evaluation sequence of arc conditions, default arcs etc.) may also need to be migrated and others (e.g. locks, preconditions, postconditions) may need to be replicated.

Step 3: Transformation for *new*YAWL-net nid" (from step 2) to replace individual blocking tasks with core *new*YAWL constructs

Let $b \in \bigcup_{t \in T_{nid''}} Block''(t)$, transforming b in net *nid* leads to the following changes:

$$C''' = C'' \cup \{C_{b,x_{B1}} \mid x \in dom(Thresh'') \wedge b \in Block''(x)\}$$

$$T''' = T'' \cup \{b_{B1}, b_{B2}\}$$

$$T_A''' = T_A'' \cup \{b_{B1}, b_{B2}\}$$

$$F''' = ((F'' \setminus \{(c, b) \mid c \in \bullet b\}) \setminus \{(b, c) \mid c \in b\bullet\}) \\ \cup \{(c, b_{B1}) \mid c \in \bullet b\} \cup \{(b_{B2}, c) \mid c \in b\bullet\} \cup \{(b_{B1}, b), (b, b_{B2})\} \\ \cup \{(C_{b,x_{B1}}, x_J^{\mathcal{C}}) \mid x \in dom(Block'') \wedge b \in Block''(x) \\ \wedge \mathcal{C} \in \mathbb{P}(\bullet x) \wedge |\mathcal{C}| = Thresh''(x)\} \\ \cup \{(x_R^{\mathcal{D}}, C_{b,x_{B1}}) \mid x \in dom(Block'') \wedge b \in Block''(x) \\ \wedge \mathcal{D} \in \mathbb{P}(\bullet x) \wedge |\mathcal{D}| = |\bullet x| - Thresh''(x)\} \\ \cup \{(T_{S2_{nid}}, C_{b,x_{B1}}) \mid x \in dom(Block'') \wedge b \in Block''(x)\} \\ \cup \{(C_{b,x_{B1}}, T_{E2_{nid}}) \mid x \in dom(Block'') \wedge b \in Block''(x)\} \\ \cup \{(C_{b,x_{B1}}, b) \mid x \in dom(Block'') \wedge b \in Block''(x)\} \\ \cup \{(b, C_{b,x_{B1}}) \mid x \in dom(Block'') \wedge b \in Block''(x)\}$$

$$Split''' = (Split'' \setminus \{(x, Split''(x)) \mid x \in dom(Split'') \wedge x = b\}) \\ \cup \{(x_{B2}, Split''(x)) \mid x \in dom(Split'') \wedge x = b\} \\ \cup \{(b, AND)\}$$

$$Join''' = (Join'' \setminus \{(x, Join''(x)) \mid x \in dom(Join'') \wedge x = b\}) \\ \cup \{(x_{B1}, Join''(x)) \mid x \in dom(Join'') \wedge x = b\} \\ \cup \{(b, AND)\}$$

$$
\begin{aligned}
Default''' &= (Default'' \setminus \{(x, Default''(x)) \mid x \in dom(Default'') \wedge x = b\}) \\
&\quad \cup \{(x_{B2}, Default''(x)) \mid x \in dom(Default'') \wedge x = b\} \\
<'''_{XOR} &= (<''_{XOR} \setminus \{(x, <''^{x}_{XOR}) \mid x \in dom(Split'' \rhd \{XOR\}) \wedge x = b\}) \\
&\quad \cup \{(x_{B2}, <''^{x}_{XOR}) \mid x \in dom(Split'' \rhd \{XOR\}) \wedge x = b\} \\
Rem''' &= ((Rem'' \setminus \{(x, Rem''(x)) \mid x \in dom(Rem'') \wedge x = b\}) \\
&\quad \setminus \{(x, Rem''(x)) \mid x \in dom(Rem'') \wedge b \in Rem''(x)\}) \\
&\quad \cup \{(x_{B2}, Rem''(x)) \mid x \in dom(Rem'') \wedge x = b\} \\
&\quad \cup \{(x, Rem''(x) \cup \{b_{B1}, b_{B2}\}) \mid x \in dom(Rem'') \wedge b \in Rem''(x)\} \\
Comp''' &= (Comp'' \setminus \{(x, Comp''(x)) \mid x \in dom(Comp'') \wedge x = b\}) \\
&\quad \cup \{(x_{B2}, Comp''(x)) \mid x \in dom(Comp'') \wedge x = b\} \\
Block''' &= Block'' \setminus \{(b, Block''(b))\} \\
Disable''' &= (Disable'' \setminus \{(x, Disable''(x)) \mid x \in dom(Disable'') \wedge x = b\}) \\
&\quad \cup \{(x_{B2}, Disable''(x)) \mid x \in dom(Disable'') \wedge x = b\} \\
Lock''' &= Lock'' \cup \{(x_{B1}, Lock''(x)) \mid x \in dom(Lock'') \wedge x = b\} \\
&\quad \cup \{(x_{B2}, Lock''(x)) \mid x \in dom(Lock'') \wedge x = b\} \\
ArcCond''' &= (ArcCond'' \setminus \{((x, c), cond1) \mid x \in dom(Split'' \rhd \{OR, XOR\}) \\
&\qquad \wedge ((x, c), cond1) \in ArcCond'' \wedge c \in x \bullet \wedge x = b\}) \\
&\quad \cup \{((b_{B2}, c), cond1) \mid x \in dom(Split'' \rhd \{OR, XOR\}) \\
&\qquad \wedge ((x, c), cond1) \in ArcCond'' \wedge c \in x \bullet \wedge x = b\}) \\
Pre''' &= Pre'' \cup \{(x_{B1}, Pre''(x)) \mid x \in dom(Pre'') \wedge x = b\} \\
Post''' &= Post'' \cup \{(x_{B2}, Post''(x)) \mid x \in dom(Post'') \wedge x = b\}
\end{aligned}
$$

The fourth step is to replicate any parameters passed to each blocking task to the (inserted) task preceding and following the blocking task. This is necessary as the preceding task may have associated preconditions or locks and the following task may have splits, postconditions or locks which rely on data elements passed to the blocking task in order to be evaluated.

Step 4: Transformations for Data passing model for *new*YAWL-net nid

Let $t \in T_{nid}$, transforming t in net nid leads to the following changes:

$$
\begin{aligned}
InPar' &= InPar \cup \{((t_{B1}, v), e) \mid x \in dom(Block) \\
&\qquad \wedge t \in (dom(Pre) \cup dom(Lock)) \cap Block(x) \\
&\qquad \wedge ((t, v), e) \in InPar\} \\
&\quad \cup \{((t_{B2}, v), e) \mid x \in dom(Block) \\
&\qquad \wedge t \in (dom(Split \rhd \{OR, XOR\}) \cup dom(Post) \\
&\qquad\qquad \cup dom(Lock)) \cap Block(x) \\
&\qquad \wedge ((t, v), e) \in InPar\} \\
&\quad \cup \{((x^{\mathcal{C}}_{J}, v), e) \mid x \in dom(Thresh) \wedge \mathcal{C} \in \mathbb{P}(\bullet x) \\
&\qquad \wedge |\mathcal{C}| = Thresh(x) \wedge ((x, v), e) \in InPar \wedge x = t\}
\end{aligned}
$$

$$
\begin{aligned}
OptInPar' = OptInPar \cup \{(t_{B1}, v) \mid x \in dom(Block) \\
&\wedge t \in (dom(Pre) \cup dom(Lock)) \cap Block(x) \\
&\wedge (t, v) \in OptInPar\} \\
\cup \{(t_{B2}, v) \mid x \in dom(Block) \\
&\wedge t \in (dom(Split \rhd \{OR, XOR\}) \cup dom(Post) \\
&\quad \cup dom(Lock)) \cap Block(x) \\
&\wedge (t, v) \in OptInPar\} \\
\cup \{(x_J^{\mathcal{C}}, v) \mid x \in dom(Thresh) \wedge \mathcal{C} \in \mathbb{P}(\bullet x) \\
&\wedge |\mathcal{C}| = Thresh(x) \wedge (x, v) \in OptInPar \wedge x = t\}
\end{aligned}
$$

The fifth step is to transform the *Work distribution model* associated with the *newYAWL-net* in order to ensure all added tasks are automatic (i.e. do not need to be distributed to resources for execution).

Step 5: Transformations for Work distribution model for *new*YAWL-net nid

$$Auto' = Auto \cup (T''' \setminus T);$$

The final step in the transformation process is ensure that all tasks that have been added are also included to the *new*YAWL specification.

Step 6: Transformations for *new*YAWL specification

$$TaskID' = \bigcup_{n \in NetID} T_n'''$$

$$
\begin{aligned}
STmap' = ((STmap \setminus \{(s, STmap(s)) \mid s \in dom(STmap) \\
&\wedge STmap(s) \cap dom(Thresh) \neq \emptyset\}) \\
&\setminus \{(s, STmap(s)) \mid s \in dom(STmap) \\
&\wedge x \in dom(Block) \\
&\wedge STmap(s) \cap Block(x) \neq \emptyset\}) \\
&\cup \{(s, STmap(s) \cup \{t_J^{\mathcal{C}}\}) \mid s \in dom(STmap) \\
&\wedge t \in STmap(s) \cap dom(Thresh) \\
&\wedge \mathcal{C} \in \mathbb{P}(\bullet t) \wedge |\mathcal{C}| = Thresh(t)\} \\
\cup \{(s, STmap(s) \cup \{t_{B1}, t_{B2}\}) \mid s \in dom(STmap) \\
&\wedge x \in dom(Block) \\
&\wedge STmap(s) \cap Block(x) \neq \emptyset\}
\end{aligned}
$$

4.2.6 Tasks directly linked to tasks

*new*YAWL supports the direct linkage of one task to another in a process model. However, during execution, it is necessary that the state of a process instance can be completely captured. For this reason, an implicit condition is inserted between directly connected tasks, reflecting the Petri net foundations on which *new*YAWL is based. Figure 21 illustrates this transformation.

This transformation only applies to elements of a *newYAWL-net* as described in Definition 2. There are no changes to the other models.

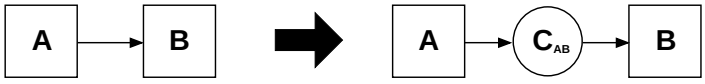


Figure 21: Inserting an implicit condition between directly linked tasks

Transformations for *new*YAWL-net nid to insert conditions between directly linked tasks

$$C' = C \cup \{C_{t_1,t_2} \mid (t_1,t_2) \in F \cap (T \times T)\}$$

$$\begin{aligned} F' = (F \backslash (T \times T)) \\ \cup \{(t_1, C_{t_1t_2}) \mid (t_1,t_2) \in F \cap (T \times T)\} \\ \cup \{(C_{t_1t_2}, t_2) \mid (t_1,t_2) \in F \cap (T \times T)\} \end{aligned}$$

$$\begin{aligned} Rem' = (Rem \backslash \{(t, Rem(t)) \mid (t_1,t_2) \in F \cap (T \times T) \wedge t \in dom(Rem) \\ \wedge \{t_1,t_2\} \subseteq Rem(t)\}) \\ \cup \{(t, Rem(t) \cup \{C_{t_1,t_2}\}) \mid (t_1,t_2) \in F \cap (T \times T) \wedge t \in dom(Rem) \\ \wedge \{t_1,t_2\} \subseteq Rem(t)\} \end{aligned}$$

$$\begin{aligned} ArcCond' = (ArcCond \setminus \{((t_1,t_2), cond1) \mid ((t_1,t_2), cond1) \in ArcCond \\ \wedge (t_1,t_2) \in F \cap (T \times T)\}) \\ \cup \{((t_1, C_{t_1t_2}), cond1) \mid ((t_1,t_2), cond1) \in ArcCond \\ \wedge (t_1,t_2) \in F \cap (T \times T)\} \end{aligned}$$

$$\begin{aligned} Default' = (Default \setminus \{(t_1, Default(t_1)) \mid t_1 \in dom(Default) \\ \wedge (t_1, Default(t_1)) \in F \cap (T \times T)\}) \\ \cup \{(t_1, C_{t_1t_2}) \mid t_1 \in dom(Default) \\ \wedge (t_1, Default(t_1)) \in F \cap (T \times T)\}) \end{aligned}$$

$$\begin{aligned} <'^{t}_{XOR} = \{(t, (<^{t}_{XOR} \cap (C \times C)) \\ \cup \{(x, C_{t,t'}) \mid t' \in T \cap t\bullet \wedge x \in C \wedge (x,t') \in <^{t}_{XOR}\} \\ \cup \{(C_{t,t'}, x) \mid t' \in T \cap t\bullet \wedge x \in C \wedge (t',x) \in <^{t}_{XOR}\} \\ \cup \{(C_{t,t'}, C_{t,t''}) \mid t' \in T \cap t\bullet \wedge t'' \in T \cap t\bullet \wedge (t',t'') \in <^{t}_{XOR}\}) \\ \mid t \in dom(Split) \wedge Split(t) = XOR\} \end{aligned}$$

4.3 Semantic model initialization

This section presents a series of *marking functions* which describe how a core *new*YAWL specification can be transformed into an initial marking of the *new*YAWL semantic model. The act of doing this prepares a *new*YAWL specification for enactment. Moreover, because the *new*YAWL semantic model is formalized using CP-nets, once the resultant initial marking is applied to the semantic model in the CPN Tools environment, it is possible to directly execute the *new*YAWL specification. The marking functions presented below operate between a core *new*YAWL specification and the semantic model presented in Section 5. They assume the existence of the auxiliary functions described subsequently.

4.3.1 Auxiliary functions

In order to describe the various transformations more succinctly, we first present eleven auxiliary functions. Throughout this section the designated value *procid* is assumed to be the identifier for the *new*YAWL specification under consideration.

$\mathcal{FUN}(fn(p_1, p_2, ...p_n)) = fn$ where $fn(p_1, p_2, ...p_n)$ is a function definition. $\mathcal{FUN}$ returns the name of the function;

$\mathcal{VARS}(fn(p_1, p_2, ...p_n)) = \{p_1, p_2, ...p_n\}$. $\mathcal{VARS}$ returns the set of data elements in the function definition;

$\mathcal{VAR}(fn(p_1)) = p_1$ where p_1 is of type *RecExpr*, i.e. p_1 is a record-based formal parameter containing a single data element in a tabular format for use with a multiple instance task. $\mathcal{VAR}$ returns the record-based data element;

$\mathcal{LCONDS}$ takes a task and returns a sequence of the link condition tuples for the outgoing arcs associated with the task, together with the set of variables and the condition used to evaluate whether the arc should be selected[10]. For XOR-splits, the sequence defines the order in which the arc conditions should be evaluated in order to determine the arc that will be enabled. The order is immaterial for OR-splits as several arcs can potentially be enabled.

$$\mathcal{LCONDS}(t) = \begin{cases} [(ArcCond(t,c), \mathcal{VARS}(ArcCond(t,c)), c) \mid c \leftarrow [t\bullet]^{<^t_{XOR}}] & \text{if } Split(t) = XOR \\ [(ArcCond(t,c), \mathcal{VARS}(ArcCond(t,c)), c) \mid c \leftarrow [t\bullet]] & \text{if } Split(t) = OR \end{cases}$$

$\mathcal{CAPVALS}(u) = [\{(c, v) \mid UserQual(u, c) = v\}]$, i.e. $\mathcal{CAPVALS}$ returns a list of the capability-value tuples corresponding to a nominated user u. The ordering of these tuples is arbitrary;
$\mathcal{VDEF}$ takes a variable v of type *VarID* and returns the static definition for the variable in the form used in the semantic model;

$$\mathcal{VDEF}(v) = \begin{cases} < \text{gdef:}(procid, VName(v)) > & \text{if } v \in VarID^{Global} \\ < \text{bdef:}(procid, VFmap(v), VName(v)) > & \text{if } v \in VarID^{Folder} \\ < \text{cdef:}(procid, VName(v)) > & \text{if } v \in VarID^{Case} \\ < \text{bdef:}(procid, VBmap(v), VName(v)) > & \text{if } v \in VarID^{Block} \\ < \text{sdef:}(procid, VSmap(v), VName(v)) > & \text{if } v \in VarID^{Scope} \\ < \text{tdef:}(procid, VTmap(v), VName(v)) > & \text{if } v \in VarID^{Task} \\ < \text{mdef:}(procid, VMmap(v), VName(v)) > & \text{if } v \in VarID^{MI} \end{cases}$$

$\mathcal{PUSAGE}$ identifies whether parameter p is a mandatory or optional parameter in the context of task t;

$$\mathcal{PUSAGE}(x,p) = \begin{cases} ''opt'' & \text{if } ((x,p) \in (dom(OptInPar) \cup dom(OptOutPar) \\ & \qquad \cup dom(OptInNet) \cup dom(OptOutNet)) \\ & \vee (x ='' null'' \wedge p \in dom(OptInProc) \\ & \qquad \cup dom(OptOutProc)) \\ ''mand'' & \text{otherwise} \end{cases}$$

[10] Details of the sequence comprehension notation used herein can be found in Appendix A.

$\mathcal{MTYPE}$ identifies whether task t is a singular or multiple-instance task;

$$\mathcal{MTYPE}(t) = \begin{cases} \text{''}multiple\text{''} & \text{if } t \in dom(Nofi) \\ \text{''}singular\text{''} & \text{otherwise} \end{cases}$$

$\mathcal{PREC}$ returns true if task x precedes task t (i.e. there is a path from x to t);

$\mathcal{PREC}(x,t) = (x,t) \in F^*$, where F^* is the reflexive transitive closure of the flow relation F;

The following three functions support the transformation of a *new*YAWL-net to a corresponding reset net that allows the enablement of an OR-join construct to be determined based on Wynn et al.'s algorithm [WEAH05]. These functions are based on the transformations illustrated in Figure 4.3.1.

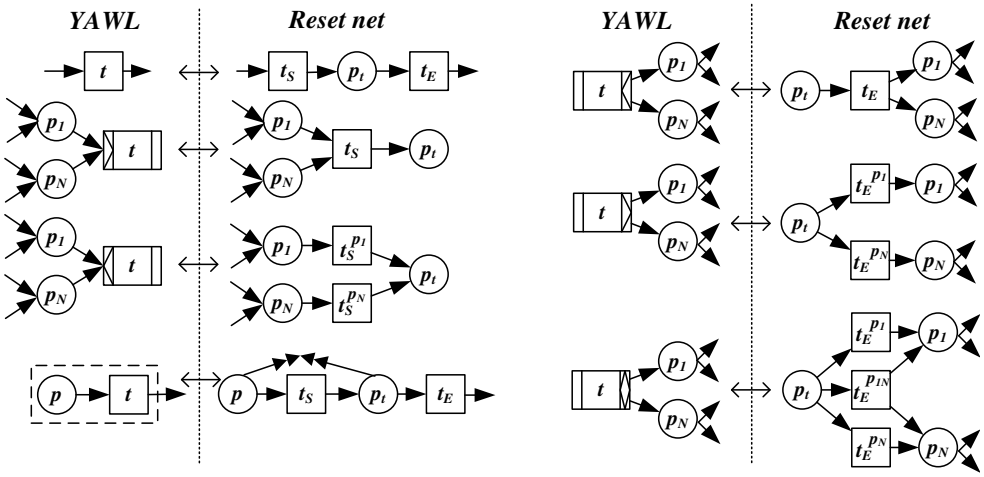


Figure 22: Reset net transformations for *new*YAWL constructs
(from [WEAH05])

$\mathcal{RNILS}(t_{OR})$ returns the set of input arcs to tasks in the reset net corresponding to the *new*YAWL-net of which the OR-join t_{OR} is a member. Only the tasks which precede t_{OR} are included in this set.

$$\begin{aligned}
&\mathcal{RNILS}(t_{OR}) = \\
&\quad \{(x, t_S) \mid t \notin dom(Join) \cup dom(Split) \wedge x \in {\bullet t} \wedge \mathcal{PREC}(t, t_{OR})\} \\
&\quad \cup \{(p_t, t_E) \mid t \in T \wedge t \in T \wedge t \notin dom(Join) \cup dom(Split) \wedge x \in {\bullet t} \wedge \mathcal{PREC}(t, t_{OR})\} \\
&\quad \cup \{(x, t_S) \mid t \in T \wedge Join(t) = AND \wedge x \in {\bullet t} \wedge \mathcal{PREC}(t, t_{OR})\} \\
&\quad \cup \{(p_t, t_E) \mid t \in T \wedge Split(t) = AND \wedge \mathcal{PREC}(t, t_{OR})\} \\
&\quad \cup \{(x, t_S^x) \mid t \in T \wedge Join(t) = XOR \wedge x \in {\bullet t} \wedge \mathcal{PREC}(t, t_{OR})\} \\
&\quad \cup \{(p_t, t_E^x) \mid t \in T \wedge Split(t) = XOR \wedge x \in t\bullet \wedge \mathcal{PREC}(t, t_{OR})\} \\
&\quad \cup \{(p_t, t_E^x) \mid t \in T \wedge Split(t) = OR \wedge x \in \mathbb{P}^+(t\bullet) \wedge \mathcal{PREC}(t, t_{OR})\}
\end{aligned}$$

$\mathcal{RNOLS}(t_{OR})$ returns the set of output arcs to tasks in the reset net corresponding to the *new*YAWL-net of which the OR-join t_{OR} is a member. Only the tasks which precede t_{OR} are included in this set.

$\mathcal{RNOLS}(t_{OR}) =$
$\{(t_S, p_t) \mid t \in T \wedge t \notin dom(Join) \wedge t \notin dom(Split) \wedge \mathcal{PREC}(t, t_{OR})\}$
$\cup \{(t_E, x) \mid t \in T \wedge t \notin dom(Join) \wedge t \notin dom(Split) \wedge x \in t\bullet \wedge \mathcal{PREC}(t, t_{OR})\}$
$\cup \{(t_S, p_t) \mid t \in T \wedge Join(t) = AND \wedge \mathcal{PREC}(t, t_{OR})\}$
$\cup \{(t_E, x) \mid t \in T \wedge Split(t) = AND \wedge x \in t\bullet \wedge \mathcal{PREC}(t, t_{OR})\}$
$\cup \{(t^x_S, p_t) \mid t \in T \wedge Join(t) = XOR \wedge x \in \bullet t \wedge \mathcal{PREC}(t, t_{OR})\}$
$\cup \{(t^x_E, x) \mid t \in T \wedge Split(t) = XOR \wedge x \in t\bullet \wedge \mathcal{PREC}(t, t_{OR})\}$
$\cup \{(t^x_E, y) \mid t \in T \wedge Split(t) = OR \wedge x \in \mathbb{P}^+(t\bullet) \wedge y \in x \wedge \mathcal{PREC}(t, t_{OR})\}$

$\mathcal{RNRLS}(t_{OR})$ returns the set of reset arcs to tasks in the reset net corresponding to the *new*YAWL-net of which the OR-join t_{OR} is a member. Only the tasks which precede t_{OR} are included in this set.

$\mathcal{RNRLS}(t_{OR}) =$
$\{(t_E, x) \mid t \in T \wedge t \notin dom(Split) \wedge \mathcal{PREC}(t, t_{OR})$
$\wedge x \in \{\{p_{t'} \mid t' \in Rem(t) \cap T \wedge \mathcal{PREC}(t', t_{OR})\}$
$\cup \{c \mid c \in Rem(t) \cap C \wedge \mathcal{PREC}(c, t_{OR})\}\}\}$
$\{(t_E, x) \mid t \in T \wedge Split(t) = AND \wedge \mathcal{PREC}(t, t_{OR})$
$\wedge x \in \{\{p_{t'} \mid t' \in Rem(t) \cap T \wedge \mathcal{PREC}(t', t_{OR})\}$
$\cup \{c \mid c \in Rem(t) \cap C \wedge \mathcal{PREC}(c, t_{OR})\}\}\}$
$\{(t^p_E, x) \mid t \in T \wedge Split(t) = XOR \wedge p \in t\bullet \wedge \mathcal{PREC}(t, t_{OR})$
$\wedge x \in \{\{p_{t'} \mid t' \in Rem(t) \cap T \wedge \mathcal{PREC}(t', t_{OR})\}$
$\cup \{c \mid c \in Rem(t) \cap C \wedge \mathcal{PREC}(c, t_{OR})\}\}\}$
$\{(t^y_E, x) \mid t \in T \wedge Split(t) = OR \wedge y \in \mathbb{P}^+(t\bullet) \wedge \mathcal{PREC}(t, t_{OR})$
$\wedge x \in \{\{p_{t'} \mid t' \in Rem(t) \cap T \wedge \mathcal{PREC}(t', t_{OR})\}$
$\cup \{c \mid c \in Rem(t) \cap C \wedge \mathcal{PREC}(c, t_{OR})\}\}\}$

$\mathcal{RNCS}(t_{OR})$ returns the set of conditions in the reset net corresponding to the *new*YAWL-net of which the OR-join t_{OR} is a member. Only the conditions which precede t_{OR} are included in this set.

$\mathcal{RNCS}(t_{OR}) = dom(RNILS(t_{OR})) \cup ran(RNOLS(t_{OR}))$;

4.4 *new*YAWL marking functions

This section presents a series of *marking functions* which describe how a core *new*YAWL specification can be transformed into an initial marking of the *new*YAWL semantic model. They assume the existence of the auxiliary functions described above. The population of a place in the CPN Tools environment is assumed to be a multiset however, for the purposes of these transformations, there is no requirement for multiplicity. As part of these transformations, it is assumed that a mechanism exists for mapping conditions that exist within a core *new*YAWL specification to corresponding ML functions that describe their evaluation in the CPN Tools environment.

The *process state* place records the *new*YAWL conditions in which tokens are present within a *new*YAWL specification. Initially this place is empty as there are no tokens yet.

pop(process state) = $\emptyset$

The *folder mappings* place records the correspondences between the folder names used when defining variable usage in a process definition and the actual folder IDs assigned to a process instance at initiation. Initially it is empty as the correspondences are recorded when a process instance is initiated.

$pop(\text{folder mappings}) = \emptyset$

The *scope mappings* place identifies the tasks which correspond to a given scope. It is populated from the *STmap* function in the abstract syntax model.

$pop(\text{scope mappings}) = \{< procid, s, STmap(s) > \mid s \in ScopeID\}$

inlinks and *outlinks* records the incoming and outgoing arcs for tasks in the flow relation. They are initially populated from the function F in the abstract syntax model;

$\text{inlinks} = \{< procid, c, t > \mid (c, t) \in F \wedge c \in C \wedge t \in T\}$
$\text{outlinks} = \{< procid, t, c > \mid (t, c) \in F \wedge c \in C \wedge t \in T\}$

The *flow relation* place is the aggregation of *inlinks* and *outlinks*.

$pop(\text{flow relation}) = \text{inlinks} \cup \text{outlinks}$

The *variable instances* place holds the values of variables instantiated during the execution of a process instance. Initially it is empty.

$pop(\text{variable instances}) = \emptyset$

The *variable declarations* place holds the static definitions of variables used during execution of the process. It is populated using the $\mathcal{VDEF}$ function along with data from the *PushAllowed* and *PullAllowed* attributes and the *DType* functions in the abstract syntax model.

$$\text{Let } VarT^X = \{< \mathcal{VDEF}(v), v \in PushAllowed, v \in PullAllowed, DType(v) > \mid v \in VarID^X\}$$

$$pop(\text{variable declarations}) = \{< VarT^{Global}, VarT^{Folder}, VarT^{Case}, VarT^{Block}, VarT^{Scope}, VarT^{Task}, VarT^{MI} >\}$$

The *lock register* place holds details of variables that have been locked by a specific task instance during execution. Initially it is empty as no variables exist.

$pop(\text{lock register}) = \emptyset$;

The *process hierarchy* place identifies the correspondences between composite tasks and their corresponding *new*YAWL-net decompositions. It is initially populated from T_n and the *TNmap* and *STmap* functions in the abstract syntax model.

$$\text{Let } Scopes(nid) = \{s \in ScopeID \mid \exists_{t \in STmap(s)}[t \in T_{nid}]\}$$

$$pop(\text{process hierarchy}) = \{< procid, t, i_n, o_n, T_n, Scopes(n) > \mid t \in dom(TNmap) \wedge n = TNmap(t)\}$$

tinpars, *toutpars*, *binpars*, *boutpars*, *miinpars*, *mioutpars*, *pinpars*, *poutpars* identify the input and output parameter mappings for task, block, multiple instance and process constructs respectively. They are populated from the *InPar, OutPar, InNet, OutNet, MIInPar, MIOutPar, InProc* and *OutProc* functions in the abstract syntax model.

$$\text{tinpars} = \{< \mathit{procid}, t, \mathcal{VARS}(e), \mathcal{FUN}(e), \{v\}, ''\text{invar}'', \mathcal{PUSAGE}(t, v), \mathcal{MTYPE}(t) > | \ (t, v, e) \in \mathit{InPar}\}$$

$$\text{toutpars} = \{< \mathit{procid}, t, \mathcal{VARS}(e), \mathcal{FUN}(e), \{v\}, ''\text{outvar}'', \mathcal{PUSAGE}(t, v), \mathcal{MTYPE}(t) > | \ (t, v, e) \in \mathit{OutPar}\}$$

$$\text{binpars} = \{< \mathit{procid}, n, \mathcal{VARS}(e), \mathcal{FUN}(e), \{v\}, ''\text{invar}'', \mathcal{PUSAGE}(n, v), \mathcal{MTYPE}(n) > | \ (n, v, e) \in \mathit{InNet}\}$$

$$\text{boutpars} = \{< \mathit{procid}, n, \mathcal{VARS}(e), \mathcal{FUN}(e), \{v\}, ''\text{outvar}'', \mathcal{PUSAGE}(n, v), \mathcal{MTYPE}(n) > | \ (n, v, e) \in \mathit{OutNet}\}$$

$$\text{miinpars} = \{< \mathit{procid}, t, \mathcal{VAR}(e), \mathcal{FUN}(e), v, ''\text{invar}'', ''\text{mand}'', \mathcal{MTYPE}(t) > | \ (t, v, e) \in \mathit{MIInPar}\}$$

$$\text{mioutpars} = \{< \mathit{procid}, t, \mathcal{VAR}(e), \mathcal{FUN}(e), v, ''\text{outvar}'', ''\text{mand}'', \mathcal{MTYPE}(t) > | \ (t, v, e) \in \mathit{MIOutPar}\}$$

$$\text{pinpars} = \{< \mathit{procid}, ''\text{null}'', \mathcal{VARS}(e), \mathcal{FUN}(e), \{v\}, ''\text{invar}'', \mathcal{PUSAGE}(''\text{null}'', v), ''\text{singular}'' > | \ (v, e) \in \mathit{InProc}\}$$

$$\text{poutpars} = \{< \mathit{procid}, ''\text{null}'', \mathcal{VARS}(e), \mathcal{FUN}(e), \{v\}, ''\text{outvar}'', \mathcal{PUSAGE}(''\text{null}'', v), ''\text{singular}'' > | \ (v, e) \in \mathit{OutProc} >\}$$

The *parameter mappings* place is populated from the aggregation of *tinpars*, *toutpars*, *binpars*, *boutpars*,*miinpars*, *mioutpars*, *pinpars* and *poutpars*.

$$\mathit{pop}(\text{parameter mappings}) = \text{tinpars} \cup \text{toutpars} \cup \text{binpars} \cup \text{boutpars} \cup \text{miinpars} \cup \text{mioutpars} \cup \text{pinpars} \cup \text{poutpars}$$

The *mi_a* place holds work items that are currently active. It is initially empty.

$$\mathit{pop}(\text{mi\_a}) = \emptyset;$$

The *mi_e* place holds work items that have been enabled but not yet started. It is initially empty as no work items are active in any process instances.

$$\mathit{pop}(\text{mi\_e}) = \emptyset;$$

The *exec* place holds work items that are currently being executed. It is initially empty.

$$\mathit{pop}(\text{exec}) = \emptyset;$$

The *mi_c* place holds work items that have been completed but have not yet exited (and triggered subsequent tasks). It is initially empty.

$$\mathit{pop}(\text{mi\_c}) = \emptyset;$$

atask, *ctask*, *mitask* and *cmitask* record details of atomic, composite, multiple-instance and composite multiple-instance tasks respectively that determine how they will be dealt with at runtime. They are populated from the T and M sets and the *TNmap* function in the abstract syntax model.

$$\text{atasks} = \{< \text{atask:}(\mathit{procid}, t) > | \ t \in T_A \backslash M\};$$

$$\text{ctasks} = \{< \text{ctask:}(\mathit{procid}, t, \mathit{TNmap}(t)) > | \ t \in T_C \backslash M\};$$

$$\text{mitasks} = \{< \text{mitask:}(\mathit{procid}, t, \mathit{min}, \mathit{max}, \mathit{th}, \mathit{sd}, \mathit{canc}) > | \ t \in M \backslash T_A \wedge \mathit{Nofi}(t) = (t, \mathit{min}, \mathit{max}, \mathit{th}, \mathit{sd}, \mathit{canc})\};$$

$$\text{cmitasks} = \{< \text{cmitask:}(\mathit{procid}, t, \mathit{TNmap}(t), \mathit{min}, \mathit{max}, \mathit{th}, \mathit{sd}, \mathit{canc}) > | \ t \in M \cap T_C \wedge \mathit{Nofi}(t) = (t, \mathit{min}, \mathit{max}, \mathit{th}, \mathit{sd}, \mathit{canc})\}$$

The *task details* place is populated from the aggregation of *atask*, *ctask*, *mitask* and *cmitask*.

pop(task details) = atasks $\cup$ ctasks $\cup$ mitasks $\cup$ cmitasks;

The *active nets* place identifies the particular *new*YAWL-nets that are active (i.e. have a thread of execution running in them). Initially it is empty as no process instances (and hence no particular nets) are active.

pop(active nets) = $\emptyset$;

The *preconditions* place identifies task and process preconditions that must be satisfied in order for a task instance or process instance to proceed. It is populated from the *Pre* and *WPre* functions in the abstract syntax model.

$$pop(\text{preconditions}) = \{< procid, t, \mathcal{FUN}(Pre(t)), \mathcal{VARS}(Pre(t)) > \\ \mid t \in dom(Pre) >\} \\ \cup \{< procid,'' \text{null}'', \mathcal{FUN}(WPre), \mathcal{VARS}(WPre) >\};$$

The *postconditions* place identifies task and process postconditions that must be satisfied in order for a task instance or process instance to complete execution. It is populated from the *Post* and *WPost* functions in the abstract syntax model.

$$pop(\text{postconditions}) = \{< procid, t, \mathcal{FUN}(Post(t)), \mathcal{VARS}(Post(t)) > \\ \mid t \in dom(Post) >\} \\ \cup \{< procid,'' \text{null}'', \mathcal{FUN}(WPost), \mathcal{VARS}(WPost) >\};$$

The *assign wi to resource* place holds work items that have been enabled but need to be distributed to users who can undertake them. Initially it is empty.

pop(assign wi to resource) = $\emptyset$;

The *wi started by resource* place holds work items that have been started by a user but not yet completed. Initially it is empty.

pop(wi started by resource) = $\emptyset$;

The *wi completed by resource* place holds work items that have been completed by a user but have not had their status changed to *completed* from a control-flow perspective. Initially it is empty.

pop(wi completed by resource) = $\emptyset$;

The *wi to be cancelled* place holds work items that are in the process of being distributed to users but now need to be cancelled. Initially it is empty.

pop(wi to be cancelled) = $\emptyset$;

asplits, *osplits* and *xsplits* identify AND, OR and XOR splits in a *new*YAWL-net. For OR and XOR splits details of the outgoing link conditions and default link conditions are captured as part of this definition. They are populated from the *Split* and *Default* functions in the abstract syntax model and the $\mathcal{LCONDS}$ condition.

asplits = $\{<$ asplit : $(procid, t) > \mid Split(t) = AND\}$
osplits = $\{<$ osplit : $(procid, t, \mathcal{LCONDS}(t), Default(t)) > \mid Split(t) = OR\}$
xsplits = $\{<$ xsplit : $(procid, t, \mathcal{LCONDS}(t)), Default(t)) > \mid Split(t) = XOR\}$

The *splits* place is populated from the aggregation of *asplits*, *osplits* and *xsplits*.

pop(splits) = asplits $\cup$ osplits $\cup$ xsplits;

ajoin, *ojoin* and *xjoin* identify AND, OR and XOR joins in a *new*YAWL-net. In the case of an OR-join, the details of the reset net which can be used to determine when the OR-join can be enabled is also recorded. They are populated from the *Join* function and in the case of the OR join, also utilize the $\mathcal{RNILS}$, $\mathcal{RNOLS}$, $\mathcal{RNRLS}$ and $\mathcal{RNCS}$ functions to determine the incoming, outgoing and reset links and the set of conditions associated with the corresponding reset net.

$$\text{ajoins} = \{< \text{ajoin} : (procid, t) > \mid Join(t) = AND\}$$
$$\text{ojoins} = \{< \text{ojoin} : (procid, t, \mathcal{RNILS}(t), \mathcal{RNOLS}(t), \mathcal{RNRLS}(t), \mathcal{RNCS}(t)) > \mid Join(t) = OR\}$$
$$\text{xjoins} = \{< \text{xjoin} : (procid, t) > \mid Join(t) = XOR\}$$

The *joins* place is populated from the aggregation of *ajoins*, *ojoins* and *xjoins*.

$pop(\text{joins}) = \text{ajoins} \cup \text{ojoins} \cup \text{xjoins}$;

The *task instance count* place records the number of instances of each task that have executed for a process. Initially it is empty.

$pop(\text{task instance count}) = \emptyset$;

The *required locks* place identifies the locks on specific variables that are required for a task before an instance of it can be enabled. The place is populated from the *Lock* function.

$pop(\text{required locks}) = \{< procid, t, \mathcal{VDEF}(v) > \mid v \in Lock(t)\}$;

The *cancel set* place identifies tasks instances that should be cancelled or force-completed when an instance on a nominated task in the same process instance completes. The place is populated from the *Rem* and *Comp* functions.

$$\begin{aligned} pop(\text{cancel set}) = \{&< procid, t, Rem(t) \cap C, Rem(t) \cap T, Comp(t) > \\ &\mid t \in dom(Rem) \cap dom(Comp)\} \\ \cup \{&< procid, t, Rem(t) \cap C, Rem(t) \cap T, \emptyset > \\ &\mid t \in dom(Rem) \backslash dom(Comp)\} \\ \cup \{&< procid, t, \emptyset, \emptyset, dom(Comp) > \\ &\mid t \in dom(Comp) \backslash dom(Rem)\} \end{aligned}$$

The *disable set* place identifies multiple tasks instances that should be disabled from being able to create further dynamic instances "on the fly" when an instance on a nominated task in the same process instance completes. The place is populated from the *Disable* function.

$pop(\text{disable set}) = \{< procid, t, Disable(t) > \mid t \in dom(Disable)\}$

The *chained execution users* place identifies which users are currently operating in *chained execution* mode. Initially it is empty.

$pop(\text{chained execution users}) = \emptyset$;

The *piled execution users* place identifies which users are currently operating in *piled execution* mode. Initially it is empty.

$pop(\text{piled execution users}) = \emptyset$;

The *distributed work items* place identifies work items that can be distributed to users. Each work item has had its routing determined, but the work items have not yet been distributed to specific users. Initially it is empty.

pop(distributed work items) = $\emptyset$;

The *task distribution details* place identifies the routing strategy to be used for distributing work items corresponding to a given task. It is populated from the *DistUser*, *DistRole*, *DistVar* and *Initiator* functions.

pop(task distribution details) =
$\{< procid, t, \text{users:}DistUser(t), o, a, s > \mid t \in dom(DistUser) \wedge (o, a, s) = Initiator(t)\}$
$\cup \{< procid, t, \text{roles:}DistRole(t), o, a, s > \mid t \in dom(DistRole) \wedge (o, a, s) = Initiator(t)\}$
$\cup \{< procid, t, \text{vars:}DistVar(t), o, a, s > \mid t \in dom(DistVar) \wedge (o, a, s) = Initiator(t)\}$
$\cup \{< procid, t, AUTO, ''\text{system}'', ''\text{system}'', ''\text{system}'' > \mid t \in Auto\}$

The *offered work items* place identifies work items that have been offered to users. Initially it is empty as there are no work items.

pop(offered work items) = $\emptyset$;

The *allocated work items* place identifies work items that have been allocated to users. Initially it is empty as there are no work items.

pop(allocated work items) = $\emptyset$;

The *started work items* place identifies work items that have been started by users. Initially it is empty as there are no work items.

pop(started work items) = $\emptyset$;

The *allocation requested* place identifies work items that users have requested to have allocated to them. Initially it is empty as there are no work items.

pop(allocation requested) = $\emptyset$;

The *in progress* place identifies work items that users are currently executing. Initially it is empty as there are no work items.

pop(in progress) = $\emptyset$;

The *logged on users* place identifies users that have logged on. Initially it is empty as no users are logged on.

pop(logged on users) = $\emptyset$;

The *logged off users* place identifies users who are not currently logged on (and hence cannot execute work items). Initially all users are deemed to be logged off.

pop(logged off users) = $UserID$;

The *task user selection basis* identifies a user routing strategy for specific tasks where they must be distributed to precisely one user. The place is populated from the *UserSel* function.

pop(task user selection basis) = $\{< procid, t, UserSel(s) > \mid t \in dom(UserSel)\}$

The *users* place identifies the users to whom work may be distributed. It is populated from the *UserID* type.

$pop(\text{users}) = UserID$;

The *user role mappings* place identifies the users that correspond to each role. It is populated from the *RoleUser* function.

$pop(\text{user role mappings}) = \{< r, RoleUser(r) > \mid r \in RoleID\}$;

The *four eyes constraints* place identifies task pairs within a process instance that cannot be executed by the same user. It is populated from the *FourEyes* function.

$pop(\text{four eyes constraints}) = \{< procid, t, FourEyes(t) > \mid t \in dom(FourEyes)\}$

The *retain familiar constraints* place identifies task pairs within a process instance that must be executed by the same user. It is populated from the *SameUser* function.

$$pop(\text{retain familiar constraints}) = \{< procid, t, SameUser(t) > \mid t \in dom(SameUser)\}$$

The *org group mappings* place identifies the type and parent group (if any) for each organizational group. It is populated from the *GroupType* and *OrgStruct* functions.

$$\begin{aligned} pop(\text{org group mappings}) = \\ \{< og, GroupType(og), OrgStruct(og) > \\ \mid og \in OrgGroupID \cap dom(OrgStruct)\} \\ \cup \{< og, GroupType(og), '' \text{null}'' > \\ \mid og \in OrgGroupID \backslash dom(OrgStruct)\} \end{aligned}$$

The *user job mappings* place identifies the jobs that a given user possesses. It is populated from the *UserJob* function.

$pop(\text{user job mappings}) = \{< u, j > \mid u \in UserID \wedge j \in UserJob(u)\}$

The *org job mappings* place identifies the organizational group and superior job (if any) for a given job. It is populated from the *JobGroup* and *Superior* functions.

$$\begin{aligned} pop(\text{org job mappings}) = \{< j, JobGroup(j), Superior(j) > \mid j \in dom(Superior)\} \\ \cup \{< j, JobGroup(j), '' \text{null}'' > \mid j \in JobID \backslash dom(Superior)\} \end{aligned}$$

The *work item event log* place holds a list of significant work item events. Initially it is empty.

$pop(\text{work item event log}) = \emptyset$;

The *organizational task distributions* place holds details of tasks that are to be distributed using an organizational distribution function. It is populated from the *OrgDist* function.

$$pop(\text{organizational task distributions}) = \{< procid, t, \mathcal{FUN}(OrgDist(t)) > \mid t \in dom(OrgDist)\}$$

The *historical task distributions* place holds details of tasks that are to be distributed using a historical distribution function. It is populated from the *HistDist* function.

$$pop(\text{historical task distributions}) = \{< procid, t, \mathcal{FUN}(HistDist(t)) > \mid t \in dom(HistDist)\}$$

The *capability task distributions* place holds details of tasks that are to be distributed using a capability-based distribution function. It is populated from the *CapDist* function.

pop(capability task distributions) =
$$\{< procid, t, \mathcal{FUN}(CapDist(t)) > \mid t \in dom(CapDist)\}$$

The *user capabilities* place identifies capabilities and their associated values that individual users possess. It is populated using the $\mathcal{CAPVALS}$ function.

pop(user capabilities) = $\{< u, \mathcal{CAPVALS}(u) > \mid u \in UserID\}$

The *failed work items* place identifies work items that could not be routed to any user. Initially it is empty.

pop(failed work items) = $\emptyset$;

The *user privileges* place identifies the privileges associated with each user. It is populated from the *UserPriv* function.

pop(user privileges) = $\{< u, UserPriv(u) > \mid u \in dom(UserPriv)\}$

The *user task privileges* place identifies the privileges associated with each user in relation to a specific task. It is populated from the *UserTaskPriv* function.

pop(user task privileges) = $\{< u, t, UserTaskPriv(u, t) >$
$\mid (u, t) \in dom(UserTaskPriv)\}$

The *requested* place holds the identity work items identified for being upgraded or downgraded in a user's work list. Initially it is empty.

pop(requested) = $\emptyset$;

The *work list view* place provides a list of the work items currently in a nominated user's work list. Initially it is empty.

pop(work list view) = $\emptyset$;

The *new offerees* place identifies an alternate set of users to whom a work item may be offered. Initially it is empty.

pop(new offerees) = $\emptyset$;

5 Semantics

The preceding sections have laid the groundwork for *new*YAWL describing its objectives, proposing a new set of suitable language primitives that enable the broadest range of the patterns to be supported, detailing an abstract syntax model and describing how a candidate process model captured using the abstract syntax can be mapped into an initial marking of the semantic model thus allowing it to be directly executed. This section presents the semantic model for *new*YAWL, using a formalization based on CP-nets. This model has been developed using *CPN Tools*[11] and hence offers the dual benefits of both providing a detailed definition of the operation of *new*YAWL and also supporting the direct execution of a process model that is captured in this format. This provides an excellent basis for investigating and validating the operation of individual language elements as well as confirming they can be coalesced into a common execution environment and effectively integrated. Furthermore, such a

[11] For interested readers, the CP-net model for *new*YAWL is available from the YAWL website: see `http://www.yawl-system.com/newYAWL` for more details.

model provides an extremely effective platform for reasoning about the operation of specific constructs as well as investigating potential execution scenarios in *real-world* *new*YAWL models.

This section is organized in four parts. First an overview of the semantic model is presented. Then the core operational concepts underpinning the model are introduced. The third section describes the manner in which the control-flow and data perspectives are operationalized. Finally the fourth section introduces the work distribution facilities in *new*YAWL.

5.1 Overview

The CP-net model for *new*YAWL logically divides into two main parts: (1) the control-flow and data sections and (2) the work distribution, organizational model and resource management sections. These roughly correspond to definitions 1 and 2 and definitions 3, 4 and 5 of the abstract syntax model respectively, which in turn seek to capture the majority of control-flow and data patterns and the resource patterns.

Figure 23, which is the topmost CP-net diagram in the semantic model, provides a useful summary of the major components and their interrelationship. The various aspects of control-flow, data management and work distribution are encoded into the CP-net model as tokens in individual places. The toplevel view of the lifecycle of a process instance is indicated by the transitions in this diagram connected by the thick black line. First a new process instance is started, then there are a succession of `enter`→`start`→`complete`→`exit` transitions which fire as individual task instances are enabled, the work items associated with them are started and completed and the task instances are finalized before triggering subsequent tasks in the process model. Each atomic work item needs to be distributed to a suitable resource for execution, an act which occurs via the `work distribution` transition. This cycle repeats until the last task instance in the process is completed. At this point, the process instance is terminated via the `end case` transition. There is provision for data interchange between the process instance and the environment via the `data management` transition. Finally where a process model supports task concurrency via multiple work item instances, there is provision for the dynamic addition of work items via the `add` transition.

The major data items shared between the activities which facilitate the process execution lifecycle are shown as shared places in this diagram. Not surprisingly, this includes both *static* elements which describe characteristics of individual processes such as the flow relation, task details, variable declarations, parameter mappings, preconditions, postconditions, scope mappings and the hierarchy of processes and subprocesses which make up an overall process model, all of which remain unchanged during the execution of particular instances of the process. It also includes *dynamic* elements which describe how an individual process instance is being enacted at any given time. These elements are commonly known as the *state* of a process instance and include items such as the current marking of the place in the flow relation, variable instances and their associated values, locks which restrict concurrent access to data elements, details of subprocesses currently being enacted, folder mappings (identifying shared data folders assigned to a process instance) and the current execution state of individual work items (e.g. *enabled*, *started* or *completed*).

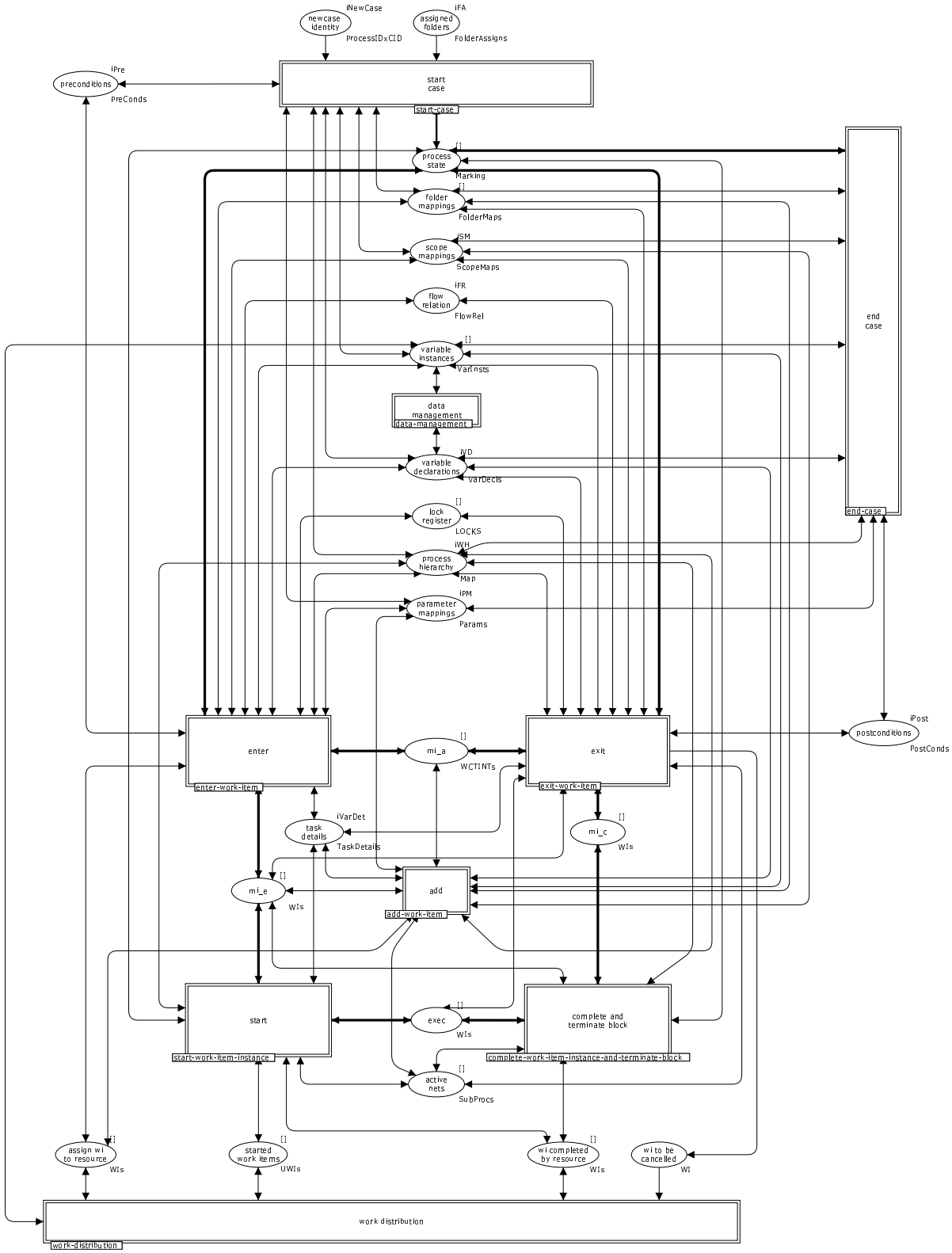


Figure 23: Overview of the process execution lifecycle

There is relatively tight coupling between the places and transitions in Figure 23, illustrating the close integration that is necessary between the various aspects of the control-flow and data perspectives in order to enact a process model. The coupling between these places and the `work distribution` transition however is more sparse. There are no static aspects of the process that are shared with other transitions in the model and other than the places which serve to communicate work items being distributed to resources for execution (and being started, completed or cancelled), the `variable instances` place is the only aspect of dynamic data that is shared with the `work distribution` subprocess.

All of these transitions in Figure 23 are substitution transitions (as illustrated by the double borders) for significantly more complex subprocesses that we will discuss in further detail in subsequent sections. These discussions will seek to both describe how the various language primitives in the *new*YAWL syntax are implemented as well as more generally explaining how the various patterns in the control-flow, data and resource perspectives are realized.

5.2 Core concepts

The CP-net model for *new*YAWL assumes some common concepts that are adopted throughout the semantic model. In the main, these extend to the way in which core aspects such as tasks, work items, subprocesses, data elements, conditional expressions and parameters are identified and utilized. Each of these issues is discussed below in more detail.

5.2.1 Work item characterization

In order to understand the issues associated with identifying a work item, it is first necessary to recap on some basic assumptions in regard to process elements that were made in the abstract syntax model and continue to hold in the semantic model. As previously indicated in this section, a process is assumed to be identified by a distinct *ProcessID*. Similarly each block or subprocess within that model is also assumed to be uniquely identified within the model by a unique *NetID*. The combination *ProcessID* $\times$ *NetID* therefore precisely identifies a given process model whether it is the toplevel process or a subprocess or block within a process model. (It should be noted that where a given block is the toplevel block in a process model then *ProcessID* = *NetID*). Individual tasks in a process model are uniquely identified by a distinct *TaskID*. A task can only appear once in a given process model[12] and it corresponds to one of the four *new*YAWL task types: i.e. *atomic*, *composite*, *multiple-instance* or *composite multiple-instance*. Each executing instance of a process is termed a *case* or *process instance* and is identified by a unique case identifier or *CID*. When a given task is instantiated within an executing process instance, a new instance of the task termed a *work item* is created. Each time a given task within a process is instantiated, it is given a unique instance identifier *Inst*. This is necessary to differentiate between distinct instances of the same task such as might occur if the task is part of a loop. The use of instance

[12]Note that as a task name is assumed to be a handle for a given implementation – whether it is atomic or composite in form – this does not preclude the associated implementation from being utilized more than once in a given process, it simply restricts the number of times a given task identifier can appear.

identifiers allows distinct instances of an atomic task to be identified, however in the situation where the task has multiple instances, each of these instances also needs unique identification, hence each instance of a multiple instance task is assigned a unique task number (denoted *TaskNr*) when it is created. Hence, in order to precisely identify a work item a five part work item identifer is necessary composed as follows: $ProcessID \times CID \times TaskID \times Inst \times TaskNr$.

5.2.2 Subprocess characterization

For the same reason that it is necessary to uniquely identify each work item, the same requirement also exists for each instantiation of a given block within a process model. Although each block within a process model can be uniquely identified, it is possible for more than one composite task to have a given block as its subprocess decomposition. Moreover, the use of recursion within a process model (an augmented control-flow pattern supported by *new*YAWL) gives rise to the possibility that a given block may contain a composite task that has that block as its subprocess decomposition. In order to distinguish between differing execution instances of the same block, the notion of a subprocess case identifier is introduced[13]. In this scheme, the case identifier for a subprocess is based on the *CID* with which the relevant composite task is instantiated together with a unique suffix. Two examples of this are shown in Figure 24. In the first of these, composite task `C` is instantiated with CID = 3. It has block `X` as its subprocess decomposition which is subsequently instantiated with (unique) CID = 3.1. At the conclusion of block `X`, the thread of control is passed back to composite task `C` which then continues with CID = 3. In the second example, composite multiple-instance task `F` is instantiated with CID = 4.5. The data passed to this task causes three distinct instances of it to be initiated. Each of these has a distinct subprocess CID, these being 4.5.1, 4.5.2 and 4.5.3 respectively.

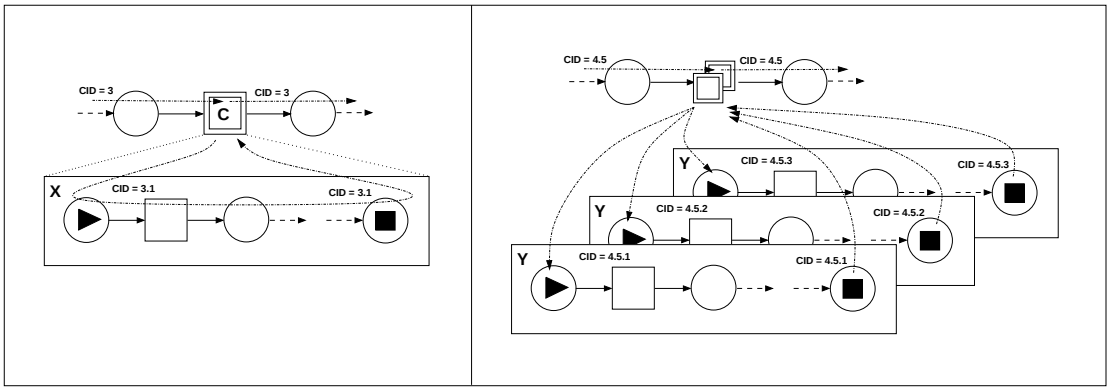


Figure 24: Subprocess identification

5.2.3 Data characterization

Seven distinct data scopings are supported in *new*YAWL: global, folder, case, block, scope, task and multiple-instance together with support for interaction with external

[13]This is not a new concept but rather a variant of the scheme first proposed in the original YAWL paper [AH05].

data elements. Individual data elements are identified by a static definition which precisely identifies the process element to which the data element is bound and the scope of its visibility. Table 7 outlines each of the types supported, the specific element to which they are bound in the static process definition and the scope of their visibility when instantiated at runtime.

Data Element Type	**Binding Element**	**Scope of Visibility**
Global	Process definition	Accessible to all work items in all instances of the process
Folder	Data folder	Accessible to all work items of process instances to which the folder is assigned at runtime
Case	Process definition	Accessible to all work items in a given process instance
Block	Specific block in a process definition	Accessible to all work items in a specific instantiation of the nominated block
Scope	Specific scope in a process definition	Accessible to all work items contained within the nominated scope in a specific instantiation of the block to which the scope is bound
Task	Specific task in a process definition	Accessible to a specific instantiation of a task
MI Task	Specific multiple instance task in a process definition	Accessible to a specific instance of a multiple instance task

Table 7: Data scopes in *new*YAWL

The identification required for data elements varies by data element type. Furthermore, a deterministic means is required to allow the static declaration for a given variable to be transformed into its runtime equivalent. Table 8 illustrates the static and dynamic naming schemes utilized for the various data types supported in *new*YAWL.

5.2.4 Conditional expression characterization

Several aspects of a *new*YAWL model utilize conditional expressions to determine whether or not a specific course of action should be taken at runtime:

– Processes and tasks can have preconditions which determine whether a specific process or task instance can commence;
– Processes and tasks can have postconditions which determine whether a specific process or task instance can conclude;
– OR-splits can have link conditions on individual arcs which determine which of them will be enabled. If none of them evaluate to true, the default arc is taken; and

Data Element Type	**Static Identification**	**Runtime identification**
Global	ProcessID × VarName	ProcessID × VarName
Folder	ProcessID × FolderID × VarName	ProcessID × FolderName × VarName
Case	ProcessID × VarName	ProcessID × CID × VarName
Block	ProcessID × NetID × VarName	ProcessID × CID × NetID × VarName
Scope	ProcessID × ScopeID × VarName	ProcessID × CID × ScopeID × VarName
Task	ProcessID × TaskID × VarName	ProcessID × CID × TaskID × Inst × VarName
MI Task	ProcessID × TaskID × VarName	ProcessID × CID × TaskID × Inst × TaskNr × VarName

Table 8: Data element identification in *new*YAWL

– XOR-splits can have link conditions on individual arcs which are evaluated in a specific order until the first of them evaluates to true allowing that specific branch to be enabled. If none of them evaluate to true, the default arc (being that with the lowest priority in the ordering) is taken.

In each of these cases, the relevant conditions are expressed in terms of a specific function name and a set of input data elements. The function is passed the values of the input data elements on invocation and must return a Boolean result.

5.2.5 Parameter characterization

Formal parameters are used in *new*YAWL to describe the passing of data values to or from a process instance, block instance (where the parameters are associated with a composite task which has the block as its associated subprocess decomposition), or task instance at runtime. Parameters have three main components, a set of input data elements, a parameter function and one (or several in the case of multiple instance parameters) output data element. All data passing is by value and is based on the evaluation of the parameter function when the values for each of the nominated input data elements are passed to it. The resultant value is subsequently assigned to the nominated output data element. Each of the input data elements must be accessible to the process element to which they are bound when it is instantiated (in the case of input parameters) or when it concludes (in the case of output parameters). Parameters can be specified as mandatory or optional. For mandatory parameters, all input data elements must have a defined value before the parameter evaluation can occur. This is not necessary in the case of optional parameters which are evaluated where possible. Generally, the output data elements for a parameter mapping reside in the same block as the process element to which the parameter is bound however for parameters passed to a composite task, the resultant output value will be assigned to a data element in the block instance to which the composite task is mapped. At the conclusion of this block, the output parameters for the composite task will map data elements from the block instance back to the data elements accessible in the block to which the composite task is bound. The potential range of input and output data elements for specific parameter types is summarized in Table 9

Parameter Type	Binding Element	Input Data Elements	Output Data Elements
Incoming	Process	Global, Folder	Case, Block, Scope
	Block (via composite task)	Global, Folder, Case, Block, Scope	Block, Scope
	Task	Global, Folder, Case, Block, Scope	Task
	MI Task	Global, Folder, Case, Block, Scope	Task, Multiple Instance
Outgoing	Process	Global, Folder, Case, Block, Scope	Global, Folder
	Block (via composite task)	Global, Folder, Case, Block, Scope, Task	Global, Folder, Case, Block, Scope
	Task	Task	Global, Folder, Case, Block, Scope
	MI Task	Multiple Instance	Global, Folder, Case, Block, Scope

Table 9: Parameter passing supported in *new*YAWL

As indicated in Figure 23, the *new*YAWL semantic model essentially divides into two main parts: control-flow and data handling, and work distribution. The following sections focus on these areas.

5.3 Control-flow & data handling

This section presents the operational semantics for control-flow and data handling in *new*YAWL. This involves consideration of the following issues: case start and completion, task instance enablement and work item creation, work item start and completion, task instance completion, multiple instance activities and data interaction with the external environment. Each of these areas is discussed in detail.

5.3.1 Case commencement

The `start case` transition handles the initiation of a process instance. It is illustrated in Figure 25. Triggering a new process instance involves passing the *ProcessID* and *CID* to the transition together with a list of any data folders that are to be assigned to the process instance during execution. There are three prerequisites for the `start case` transition to be able to fire:

1. The precondition associated with the process instance must evaluate to true;
2. Any data elements which are inputs to mandatory input parameters must exist and have a defined value (i.e. they must not have the *UNDEF* value); and
3. All mandatory input parameters must evaluate to defined values.

Once these prerequisites are satisfied, the process instance can commence. This involves:

1. Placing a token representing the new CID in the input place for the process;
2. Creating variable instances for any case data elements, block data elements for the topmost block and scope data elements for scopes in the top-level block;
3. Mapping the results of any input parameters for the process to the relevant output data elements created in step 2; and
4. Adding folder mappings to identify the folders assigned to the process instance during execution.

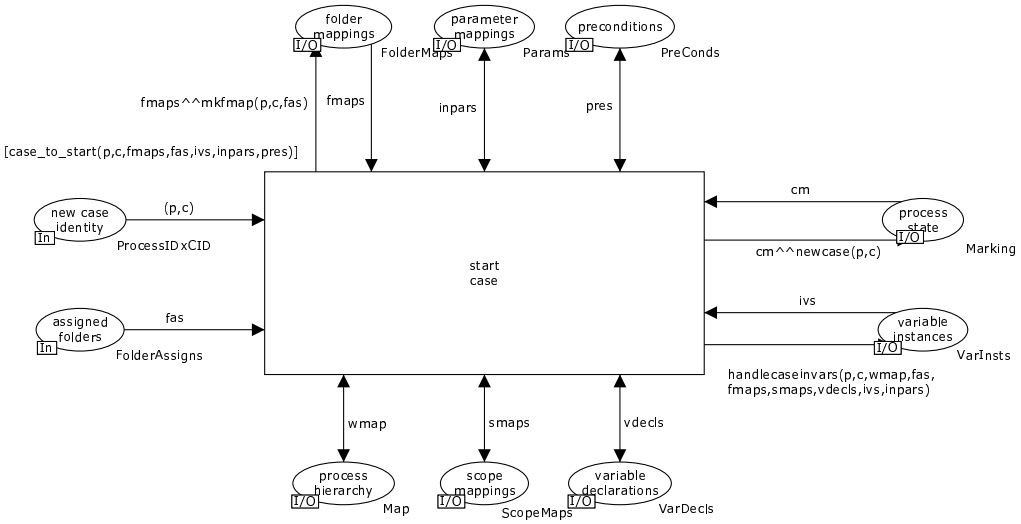


Figure 25: Start case process

5.3.2 Case completion

The `end case` transition is analogous in operation to the start-case transition described above except that it ends a case. In order for a process instance to complete, three prerequisites must be satisifed:

1. The postcondition associated with the process instance must evaluate to true;
2. Any data elements which are inputs to mandatory output parameters must exist and have a defined value; and
3. All mandatory output parameters must evaluate to defined values.

Once these prerequisites are satisfied, the process instance can complete. This involves:

1. Placing a token in the output place for the process;
2. Cancelling any work items or blocks in the process instance that are still executing;
3. Mapping the results of any output parameters for the process to the relevant output data elements;
4. Removing any remaining variable instances for the process instance; and
5. Removing any folder mappings associated with the process instance.

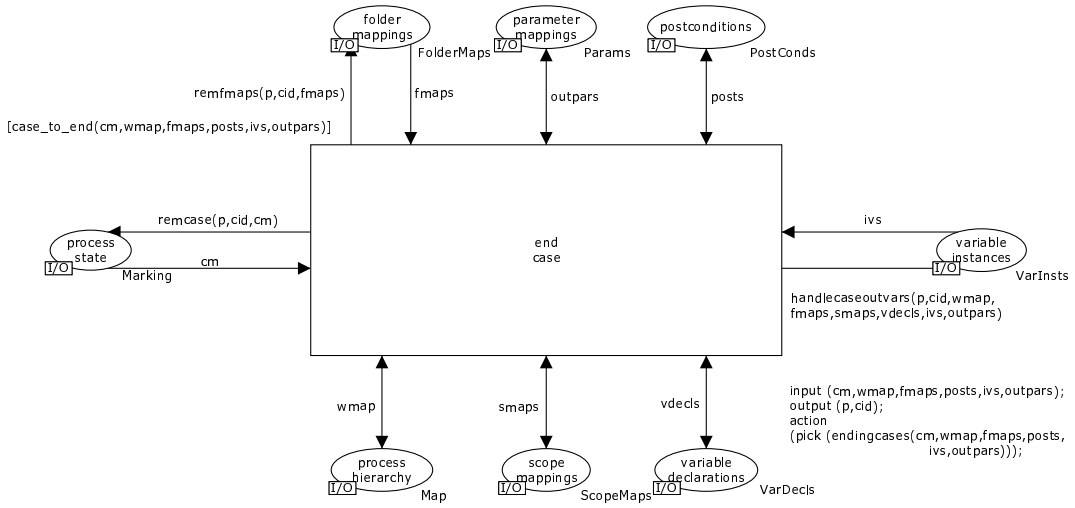


Figure 26: End case process

The execution of a work item involves the execution of a sequence of four distinct transitions. The naming and coupling of these transitions is analogous to the formalization proposed in the original YAWL paper [AH05]. In the *new*YAWL semantic model, the same essential structure for work item enactment is maintained however this structure is augmented with a complete executable model defining precisely how the control-flow, data and resource perspectives interact during work item execution. This sequence is described in detail in the following four sections.

5.3.3 Task instance enablement & work item creation

Task instance enablement is the first step in work item execution. It is depicted by the `enter` transition in Figure 27. The first step in determining whether a task instance can be enabled is to examine the marking of the input places to the task. There are four possible scenarios:

- If the task has no joins associated with it, then the input condition to the task simply needs to contain a token;
- If the task has an AND-join associated with it, each input condition needs to contain a token with the same *ProcessID* $\times$ *CID* combination;
- If the task has an XOR-join associated with it, one of the input conditions needs to contain a token; and
- If the task has an OR-join associated with it, one (or more) of the input conditions needs to contain a token and a determination needs to be made as to whether in any future possible state of the process instance, the current input conditions can retain at least one token and another input condition can also receive a token. If this can occur, the task is not enabled, otherwise it is enabled. This issue has been subject to rigorous analysis and an algorithm has been proposed [WEAH05] for determination of exactly when an OR-join can fire. The *new*YAWL semantic model implements this algorithm.

Depending on the form of task that is being enabled (singular or multiple-instance), one or more work items may be created for it. If the task is atomic, the work item(s)

is created in the same block as the task to which it corresponds. If the task is composite, then the situation is slightly more complicated and two things occur: (1) a "virtual" work item is created in the same block for each instance of the task that will be initiated (this enables later determination of whether the composite task is in progress or has completed) and (2) a new subprocess decomposition (or a new block) is started for each task instance. This involves the placement of a token in the input place to the subprocess decomposition which has a distinct subprocess CID. Table 10 indicates the potential range of work items that may be created for a given task instance.

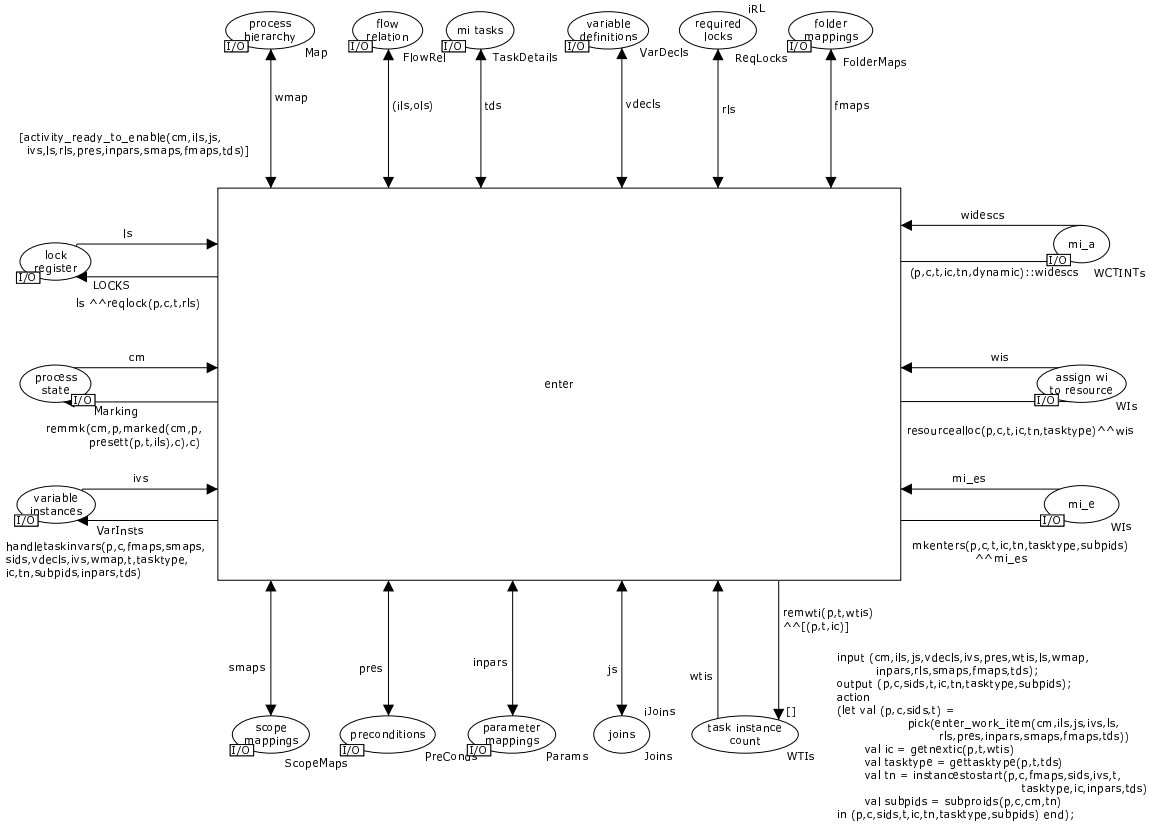


Figure 27: Enter work item process

In order for a task to be enabled, all prerequisites associated with the task must be satisfied. There are five prerequisites for the `enter` transition to be able to fire:

1. The precondition associated with the task must evaluate to true;
2. All data elements which are inputs to mandatory input parameters must exist and have a defined value;
3. All mandatory input parameters must evaluate to defined values;
4. All locks which are required for data elements that will be used by the work items associated with the task must be available; and
5. If the task is a multiple instance task, the multiple instance parameter when evaluated must yield a number of rows that is between the minimum and maximum number of instances required for the task to be initiated.

Once these prerequisites are satisfied, task enablement can occur. This involves:

Task Type	Instances Initiated at Commencement	
	Singular	*Multiple Instances*
Atomic	Single work item created in the same block.	Multiple work items created in the same block, each with a distinct *TaskNr*.
Composite	Single "virtual" work item created in the same block and a new subprocess is initiated for the block assigned as the task decomposition.	Multiple "virtual" work items created in the same block. Additionally a distinct subprocess is initiated for each work item created, each with a distinct subprocess *CID* and *TaskNr*

Table 10: Task instance enablement in *new*YAWL

1. Removing the tokens marking input conditions to the task for the instance enabled. The exact number of tokens removed depends on whether there is a join associated with the task or not and occurs as follows:
   - *No join*: one token corresponding to the *ProcessID* × *CID* combination triggered is removed from the input condition to the task;
   - *AND-join*: one token corresponding to the *ProcessID* × *CID* combination triggered is removed from each of the input conditions to the task;
   - *XOR-join*: one token corresponding to the *ProcessID* × *CID* combination triggered is removed from *one* of the input conditions to the task; and
   - *OR-join*: one token corresponding to the *ProcessID* × *CID* combination triggered is removed from any of the input conditions to the task which currently contain tokens of this form.
2. Determining which instance of the task this is. The instance identifer must be unique for each task instance and all work items and data elements associated with this task instance in order to ensure that they can be uniquely identified. A record is kept of the next available instance for a task in the `task instance count` place.
3. Determining how many work item instances should be created. For a singular task (i.e. an atomic or composite task), this will always be a single work item, however for a multiple instance task (i.e. an atomic or composite multiple instance task), the actual number started will be determined from the evaluation of the multiple instance parameter which will return a composite result containing a number of rows of data. The number of rows returned indicates the number of instances to be started. In all of these situations, individual work items are created which share the same *ProcessID*, *CID*, *TaskID* and *Inst* values, however the *TaskNr* value is unique for each work item and is in the range *1...number of work items created*;
4. For all work items corresponding to composite tasks, distinct subprocess CIDs need to be determined to ensure that any variables created for subprocesses are correctly identified and can be accessed by the work items for the subprocesses that will subsequently be triggered;
5. Creating variable instances for data elements associated with the task. This varies depending on the task type and the number of work items created for the task:

- For atomic tasks which only have a single instance, this will involve the creation of relevant task variables.
- For atomic multiple instance tasks, this will involve the creation of both task variables and multiple instance variables for each task instance. The required multiple instance variables are indicated by the output data elements listed for the multiple instance parameter. and this set of variables is created for each new work item.
- For composite tasks that only have a single instance, any required task variables are created in the subprocess decomposition that is instantiated for the task. Also, there may be block and scope variables associated with the subprocess decomposition that need to be created; and
- For composite multiple instance tasks, any required block, scope, task variables and multiple instance variables are created for each subprocess decomposition that is initiated for the task.

6. Mapping the results of any input parameters for the task instance to the relevant output data elements. For multiple instance parameters, this can be quite a complex activity as illustrated in Figure 12;
7. Recording any variable locks that are required for the execution of the task instance;
8. For all work items corresponding to atomic tasks (other than for automatic tasks which can be initiated without distribution to a resource), requests for work item distribution need to be created. These are routed to the `assign wi to resource` place and are subsequently dealt with by the `work distribution` transition; and
9. Finally, work items with an *enabled* status need to be created for this task instance and added to the `mi_e` place in accordance with the details outlined in Table 10.

5.3.4 Work item start

The action of starting a work item is denoted by the `start` transition in the *new*YAWL semantic model as shown in Figure 28. For a work item that corresponds to an atomic task, the work item can only be considered to be *executing* when a notification has been received that a resource has commenced executing it. This event is indicated by the presence of a token for the work item in the `wi started by resource` place. When this occurs, the `start` transition can fire and the work item token can be moved from the `mi_e` place to the `exec` place.

A work item corresponding to a composite task can be started at any time. This simply involves changing the state of the work item from *entered* to *executing*, noting that a new subprocess has been initiated by adding an entry to the list in the `active nets` place and initiating a new block (or subprocess decomposition) for the work item by placing a token indicating the subprocess CID in the input condition for the subprocess. Note that for work items corresponding to composite tasks, the `start` transition receives a notification to start the subprocess, hence the `exec` place is updated with a work item corresponding to the CID of the parent work item that initiated the subprocess and the list in the `active nets` place links the parent work item to the subprocess that has been newly initiated by this transition.

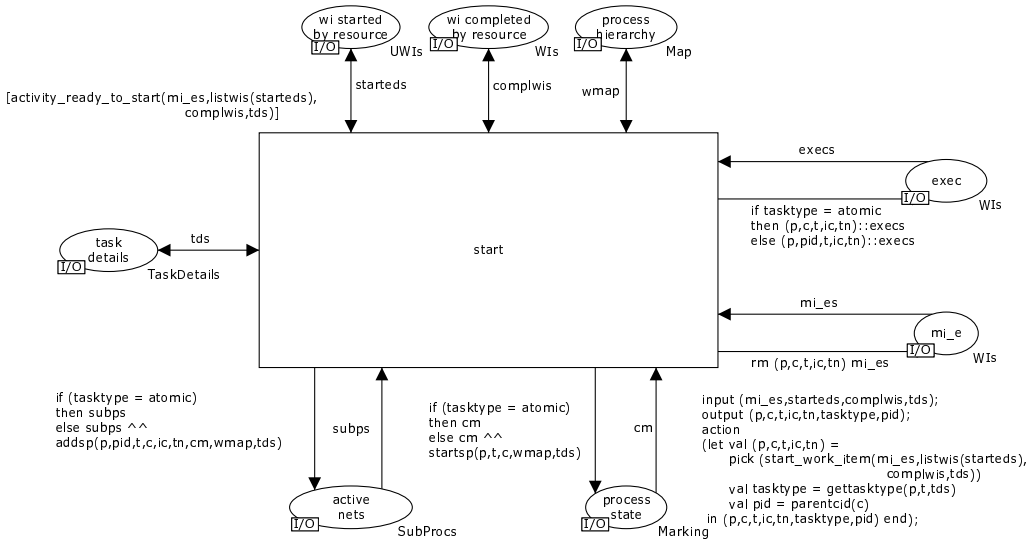


Figure 28: Start work item instance process

5.3.5 Work item completion

There are two distinct (but interrelated) transitions for completing an individual work item. These are the `terminate block` transition that completes a work item corresponding to a composite task and the `complete` transition that finishes a work item corresponding to an atomic task. Both of these transitions are illustrated in Figure 29.

The `terminate block` transition fires when a subprocess decomposition corresponding to a composite task has completed. This is indicated by a token for the subprocess CID in the output condition for the block. When this occurs, the `terminate block` transition can fire and in doing so, any work items that are currently executing for this subprocess (or any children of it resulting from composite tasks that it may contain) are removed, similarly any markings in places in the process model for this subprocess (or its children) are also removed. Finally a work item is added to the `mi_c` place indicating that the parent work item corresponding to the composite task which launched this subprocess is complete.

The `complete` transition fires when a work item corresponding to an atomic task is complete. This occurs when a notification is received (via a work item token in the `wi completed by resource` place) that a work item assigned to a resource for execution has been completed. When this occurs, the state of the work item is changed from *executing* to *completed* by moving the work item from the `exec` to the `mi_c` place.

5.3.6 Task instance completion

The act of completing a task instance is illustrated by the `exit` transition in Figure 30. It is probably the most complex transition associated with the execution of a process instance. In order for the `exit` transition to fire for a given task instance, a series of prerequisites must be met. These are:

1. The work item corresponding to a given task instance must have completed or, in

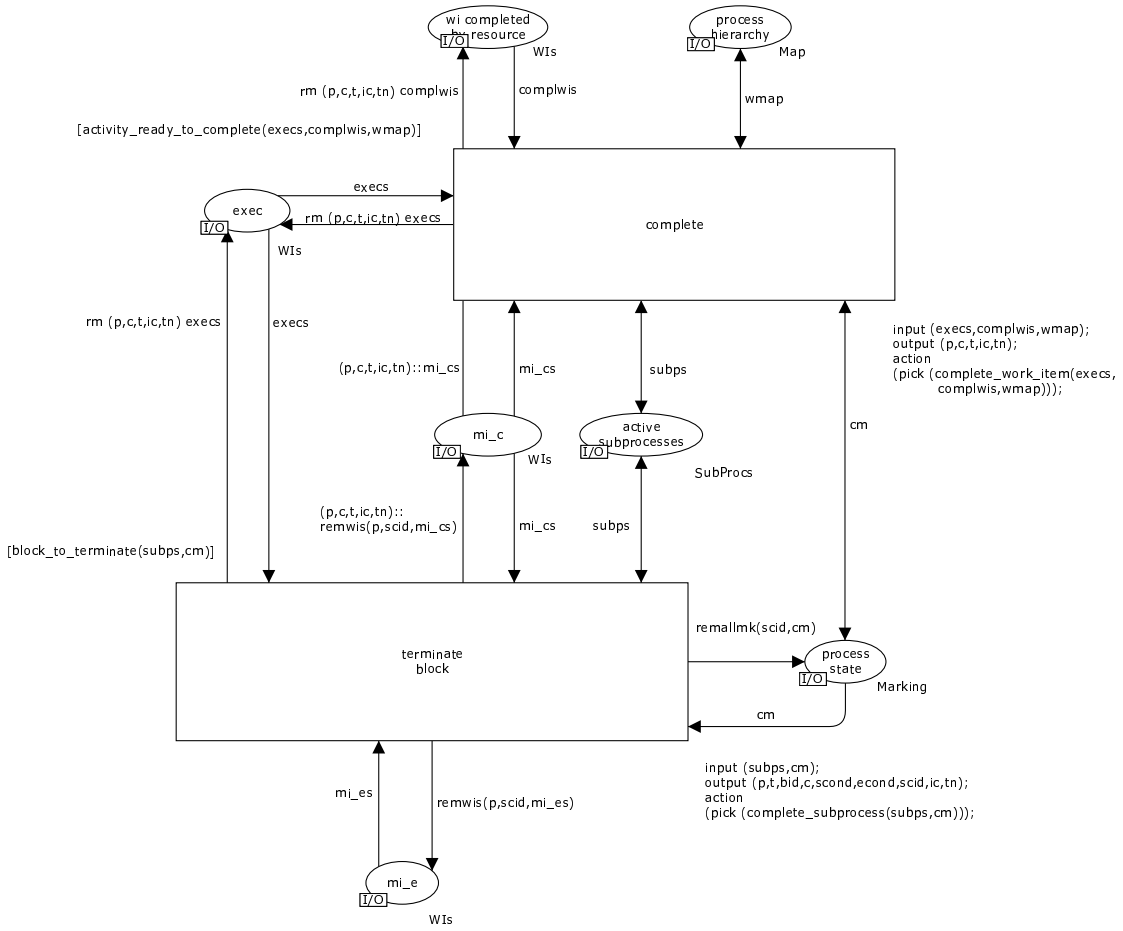


Figure 29: Complete work item instance and terminate block process

the case of a multiple instance task, at least as many work items as the *threshold* value for the task must have completed execution;
2. All data elements which are outputs to mandatory output parameters must exist and have a defined value; and
3. All mandatory output parameters must evaluate to defined values.

When the `exit` transition fires for a given task instance, a series of actions occur:

– The set of work items which cause this task instance to complete are determined;
– The set of work items which are to be cancelled as a consequence of the completion of this task instance are determined;
– The set of work items which are to be force completed as a consequence of the completion of this task instance are determined;
– The set of subprocess instances that are to be terminated as a consequence of the completion of this task instance are determined, e.g. for a partial join composite multiple instance task;
– All work items (including those in subprocesses of this task instance) currently in the *enabled*, *executing* and *completed* states that are to be cancelled or force completed are removed from the lists in the `mi_e`, `exec` and `mi_c` places;

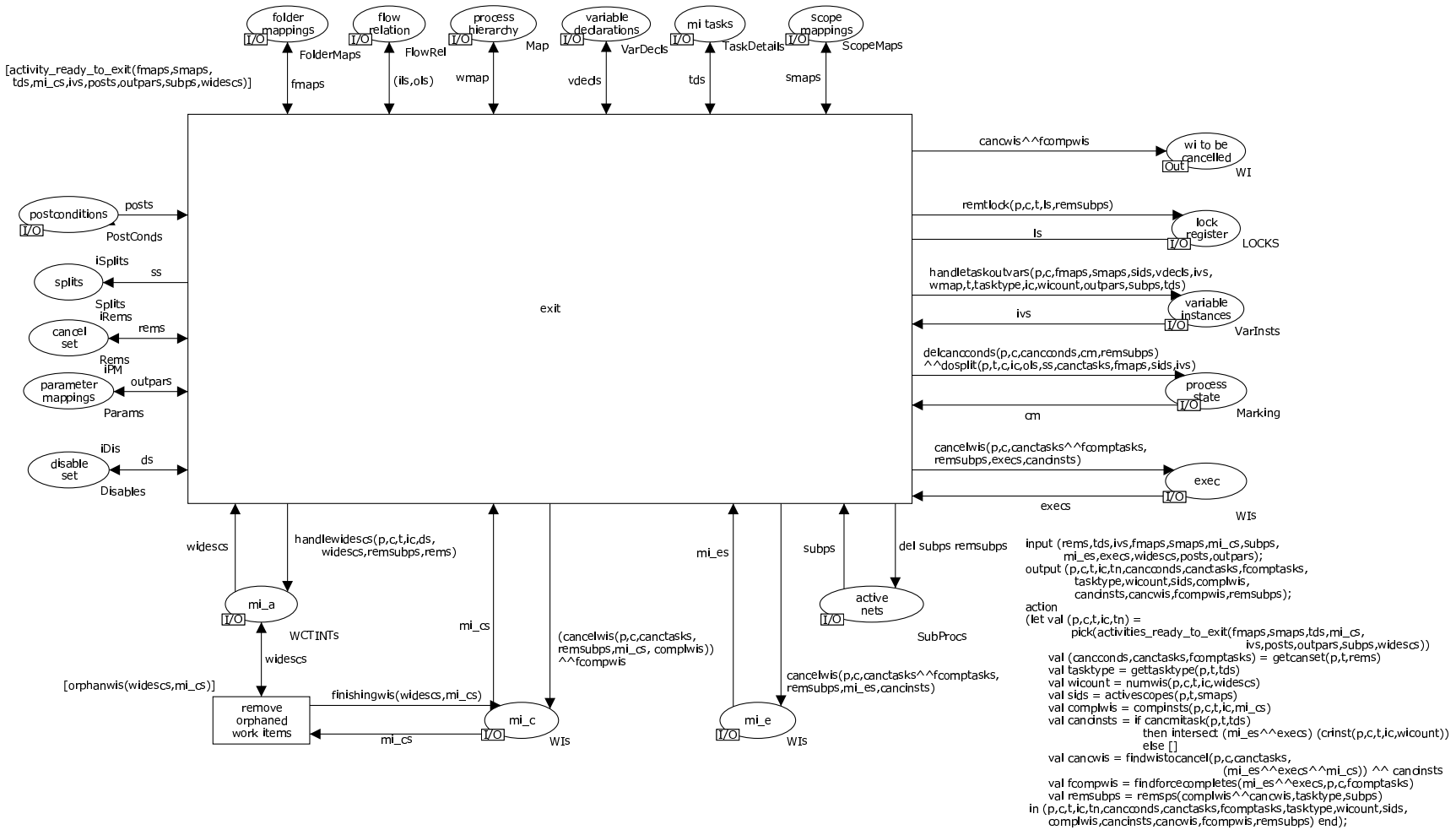


Figure 30: Exit work item process

– All subprocesses instances which are completed or terminated by this task instance are removed from the list in the `active nets` place;
– All output parameter mappings for this task instance are evaluated and the relevant output data elements are updated. In the case of a task instance which has multiple work items, the multiple instance data elements from individual instances are coalesced as illustrated in Figure 13. Note that it is possible that some data elements may be undefined where not all work items corresponding to the task instance have completed;
– All data elements for work items associated with this task instance as well as those for subprocesses of this task instance and work items to be cancelled (but not force-completed) by this task instance are destroyed and removed from the list in the `variable instances` place;
– Any locks on variables that were held by the task instance are released;
– Cancellation requests are sent to the `work distribution` transition for any atomic work items that are enabled or currently executing and are being cancelled as a consequence of the completion of this task instance;
– Any tokens in conditions in the process model to be cancelled by the completion of this task instance are removed (including those associated with subprocesses);
– Any conditions on outgoing links from this task are evaluated and tokens are placed in the associated output places. For a task without any splits or with an AND-split this means tokens are placed in all output conditions. Where the task has an associated OR-split, they are placed in the output condition for each link which has a link condition which evaluates to true (or the default condition if none of them evaluate to true) and for a task with an XOR-split a token is placed in the condition associated with the first link-condition to evaluate to true (or the default if none evaluate to true).

In general, the `exit` transition results in the completion of all work items associated with a task instance, however in the situation where a task has multiple instances and its specified completion threshold is lower than the maximum number of instances allowed and the remaining instances are not cancelled when the task instance completes, it is possible that some related work items may continue executing after the task instance has completed. These work items are allowed to continue execution but their completion will not result in any further action or other side-effects, in particular the exit transition will not fire again and no data elements will be retained for these work items.

5.3.7 Multiple instance activities

Where a task is specified as being *multiple-instance*, it is possible for several work items associated with it to execute concurrently and also for additional work item instances to be added dynamically (i.e. during task execution) in some situations. The `add` transition in Figure 31 illustrates how an additional work item is added. The prerequisite conditions for the addition of a work item are:

– The task must be specified as a *multiple-instance* task which allows for dynamic instance addition; and
– The number of associated work items currently executing must be less than the maximum number of work items allowed for the task.

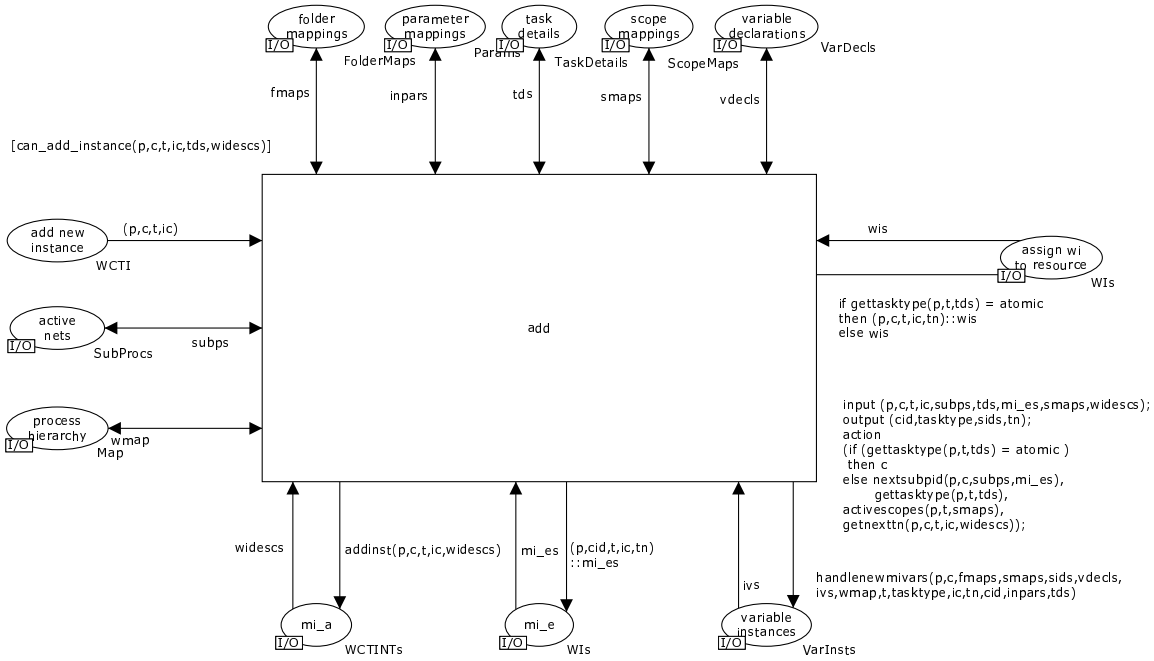


Figure 31: Add work item process

Where an additional work item is created, the following actions occur:

1. The task instance is recorded as having an additional work item associated with it;
2. An additional work item is created with an *enabled* status and added to the `mi_e` place (note that this work item shares the same *ProcessID*, *CID*, *TaskID* and *Inst* as the other work items for this task instance but it has a unique *TaskNr*); and
3. If the task is atomic, then a request is made to the `work distribution` transition for allocation of the work item to a resource for execution.

5.3.8 Data interaction with the external environment

Support is provided via the `data management` process for data interaction between a given process instance and the operating environment via *push* and *pull* interactions with variables supported by the *new*YAWL environment or those which may exist outside of the process environmnent. A *push* interaction allows an existing variable instance to be updated with a nominated value from another location. A *pull* interaction allows its value to be accessed and copied to a nominated location. *Push* and *pull* interactions are supported both in both directions for *new*YAWL data elements. Each variable declaration allows these operations to be explicitly granted or denied on a variable-by-variable basis. Figure 32 illustrates how these interactions occur.

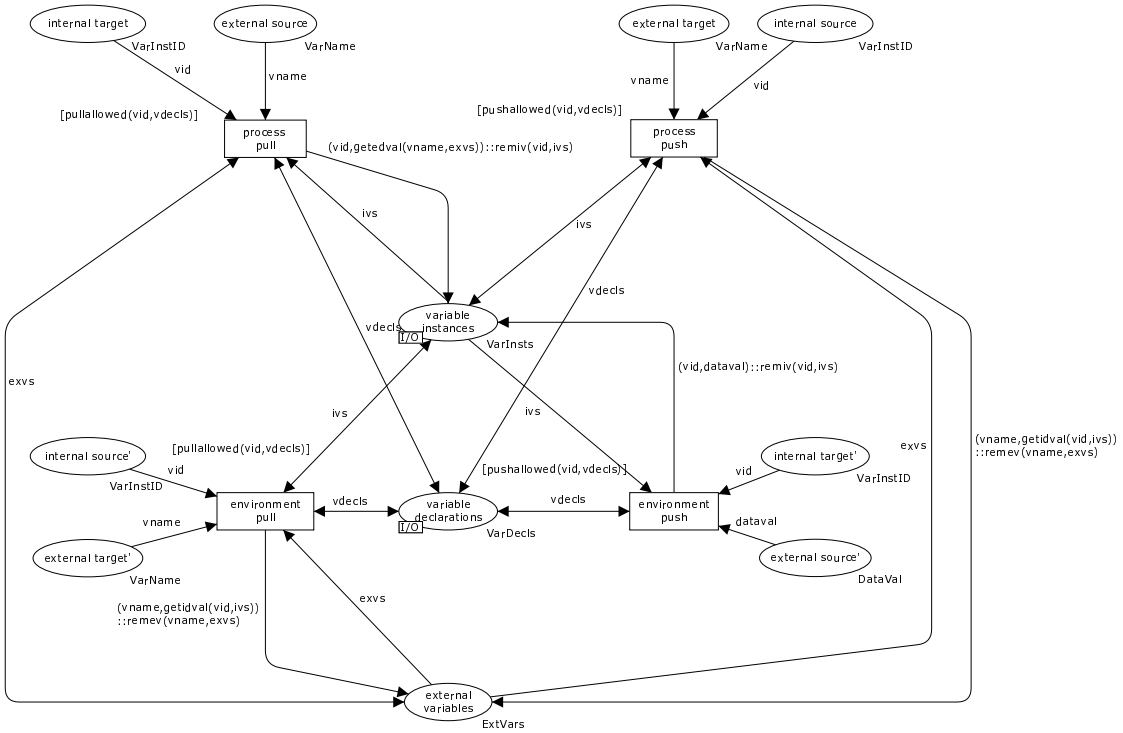


Figure 32: Data management – interactions between *new*YAWL and the operating environment

5.4 Work distribution

The main motivation for PAIS is achieving more effective and controlled distribution of work. Hence the actual distribution and management of work items are of particular importance. The process of distributing work items is summarized by Figure 33[14]. It comprises four main components:

- the `work item distribution` transition, which handles the overall management of work items though the distribution and execution process;
- the `work list handler`, which corresponds to the user-facing client software that advises users of work items requiring execution and manages their interactions with the main `work item distribution` transition in regard to committing to execute specific work items, starting and completing them;
- the `management intervention` transition, that provides the ability for a process administrator to intervene in the work distribution process and manually reassign work items to users where required; and
- the `interrupt handler` transition that supports the cancellation, force completion and force fail of work items as may be triggered by other components of the process engine (e.g. the control-flow process, exception handlers).

[14]Note that the high-level structure of the work distribution process is influenced by the earlier work of Pesic and van der Aalst [PA05].

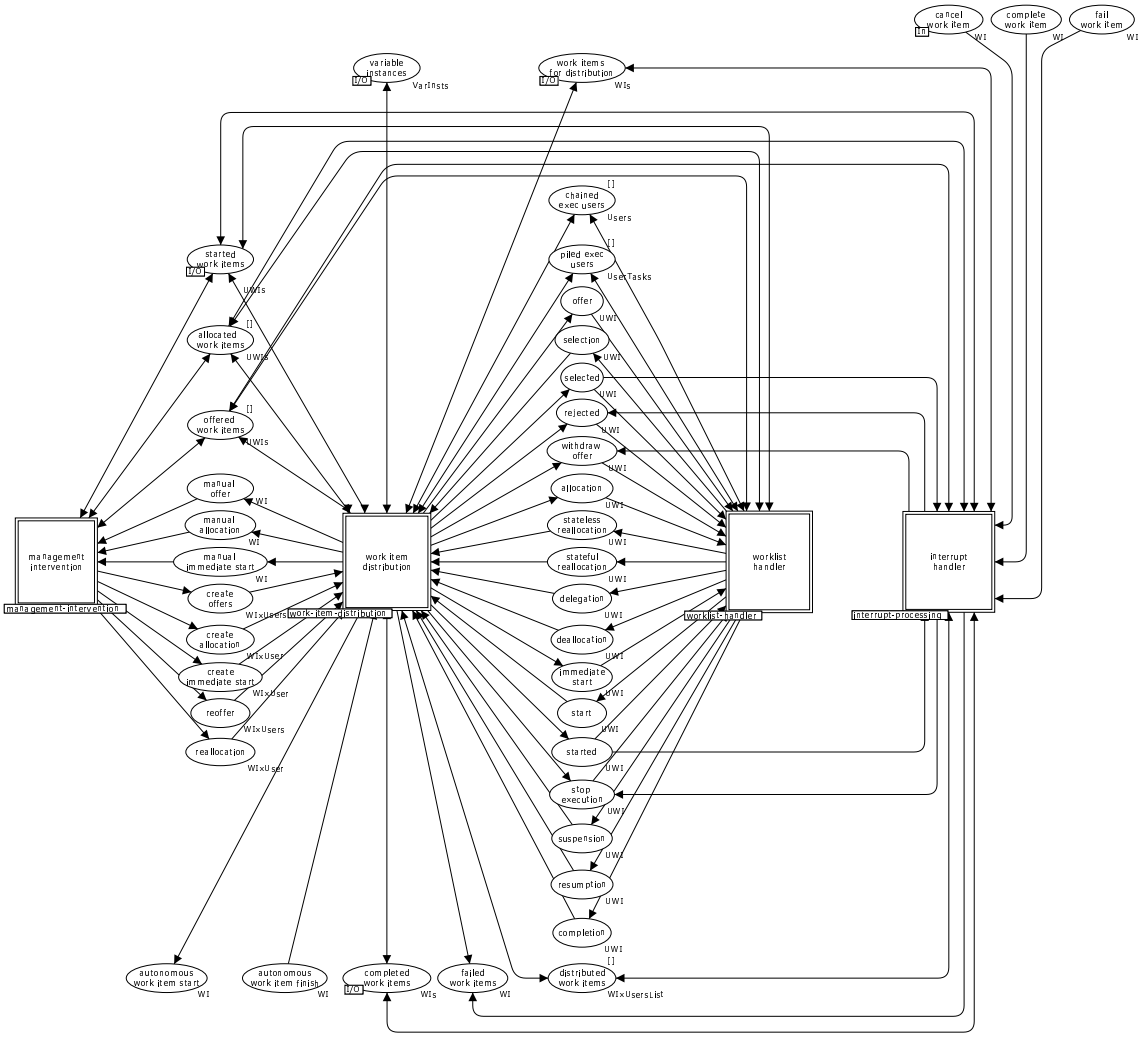


Figure 33: Top level view of the main work distribution process

Work items that are to be distributed through this process are added to the `work items for distribution` place. This then prompts the `work item distribution` transition to determine how they should be routed for execution. This may involve the services of the process administrator in which case they are sent to the `management intervention` transition or alternatively they may be sent directly to one or more users via the `worklist handler` transition. The various places between these three activities correspond to the range of requests that flow between them. There is a direct correspondence between these place names and the interaction strategies illustrated in Figure 14. In the situation where a work item corresponds to an *automatic* task, it is sent directly to the `autonomous work item start` place and no further distribution activities take place. An automatic task is considered complete when a token is inserted in the `autonomous work item finish` place.

A common view of work items in progress is maintained between the `work item distribution`, `worklist handler` and `management intervention` transitions via the `offered work items`, `allocated work items` and `started work items places`. There is also shared information about users in advanced operating modes that is

recorded in the `piled exec users` and `chained exec users` places. Although there is significant provision for shared information about the state of a work items, the determination of when a work item is actually complete rests with the `work item distribution` transition and when this occurs, it inserts a token in the `completed work items` place. Similarly, work item failures are notified via the `failed work items` place. The only exception to these arrangements are for work items that are subject to some form of interrupt (e.g. an exception being detected and handled). The `interrupt handler` transition is responsible for managing these occurrences on the basis of cancellation, force completion and failure requests received in the `cancel work item`, `complete work item` and `fail work item` places respectively.

All of the activities in the `work distribution` process are illustrated by substitution transitions indicating that each of them are defined in terms of significantly more complex subprocesses. The following sections present the CP-net models for each of them.

5.4.1 Work item distribution

The `work item distribution` process, illustrated in Figure 34 and 35, supports the activities of distributing work items to users and managing interactions with various worklist handlers as users select work items offered to them for later execution, commence work on them and indicate that they have completed them. It also support various "detours" from the normal course of events such as deallocation, reallocation and delegation of work items. It receives work items to be distributed from the `enter` or `add` transitions which forms part of the main control-flow process and sends back notifications of completed work items to the `complete` transition in the same process. The main paths through this process are indicated by thick black arcs. In general, the places on the lefthand side of the process correspond to input requests from either the main control-flow process or from the `management intervention` or `worklist handler` processes that require some form of action. Typically this results in some form of output that is illustrated by the places on the righthand side of the process.

One of the most significant aspects of this process is its ability to manage work item state coherence between itself, the `worklist handler` and `management intervention` process. This is a particular problem in the face of potential race conditions that the various parties (i.e. users, the process engine, the process administrator) involved in work distribution and completion may invoke. This is managed by enforcing a strict interleaving policy to any work item state changes where user-invoked changes are handled first (and are reflected back to the user) prior to any process engine or process administrator initiated changes. External interrupts override all other state changes since they generally result in the work item being cancelled.

5.4.2 Worklist handler

The `worklist handler` process, illustrated in Figure 36, describes how the user-facing process interface (typically a worklist handler software client) operates and interacts with the `work item distribution` process. Once again, the main path through this process are indicated by the thick black arcs. There are various transitions that make up the process, these correspond to actions that individual users can request in order to alter the current state of a work item to more closely reflect their

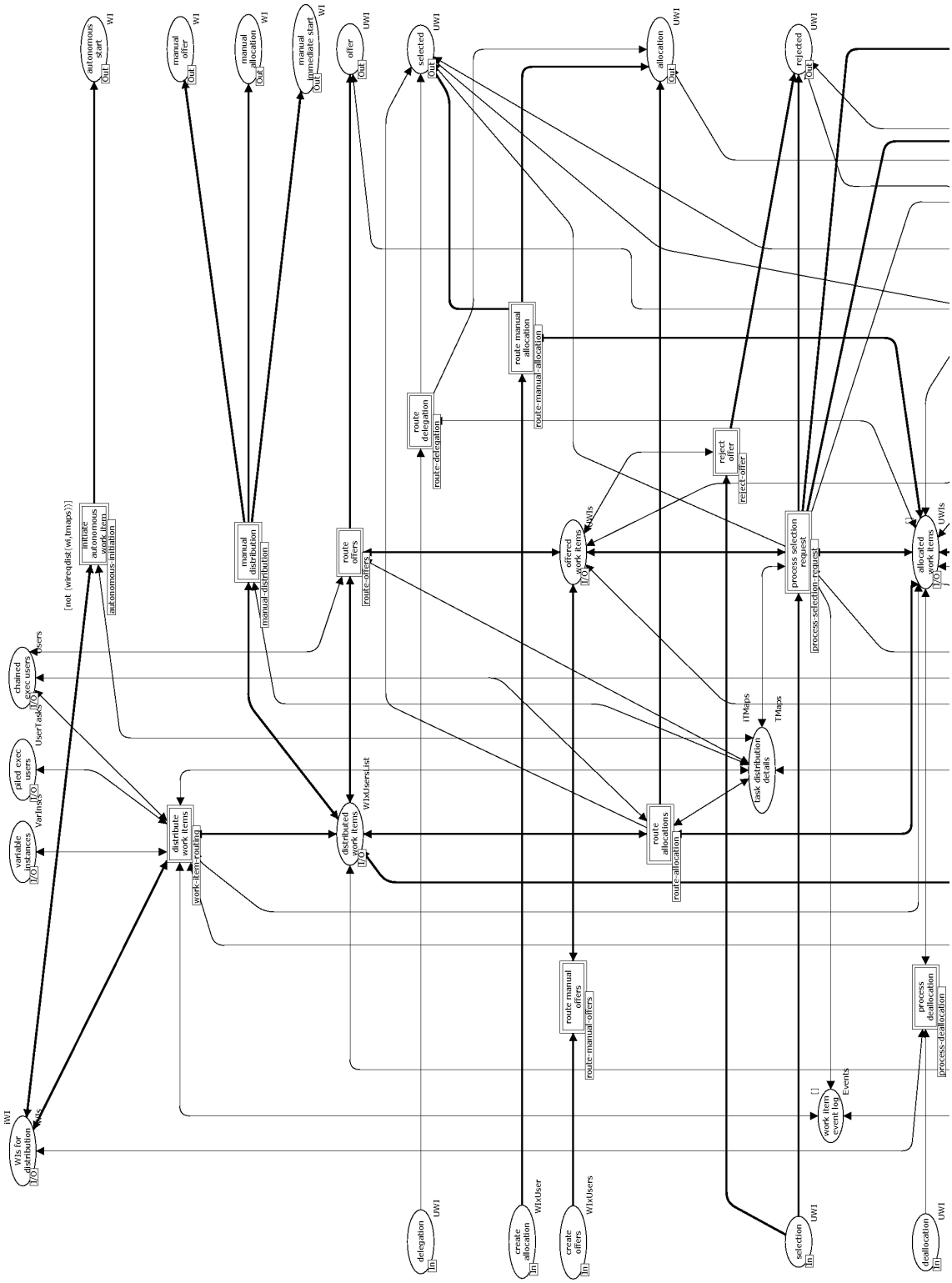


Figure 34: Work item distribution process (top half)

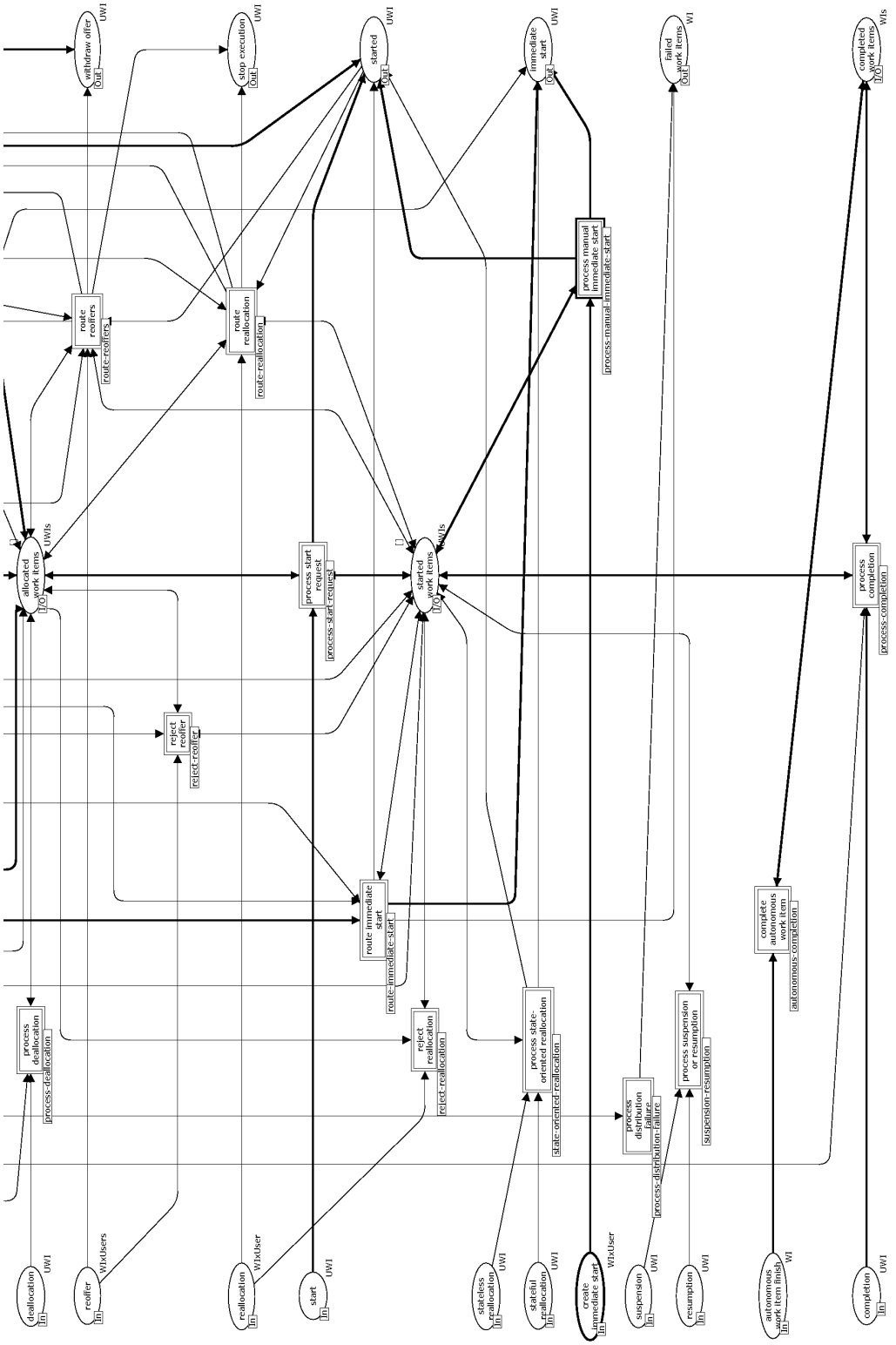


Figure 35: Work item distribution process (bottom half)

current handling of it. These actions may simply be requests to start or complete it or they may be "detour" requests to reroute it to other users e.g. via delegation or deallocation.

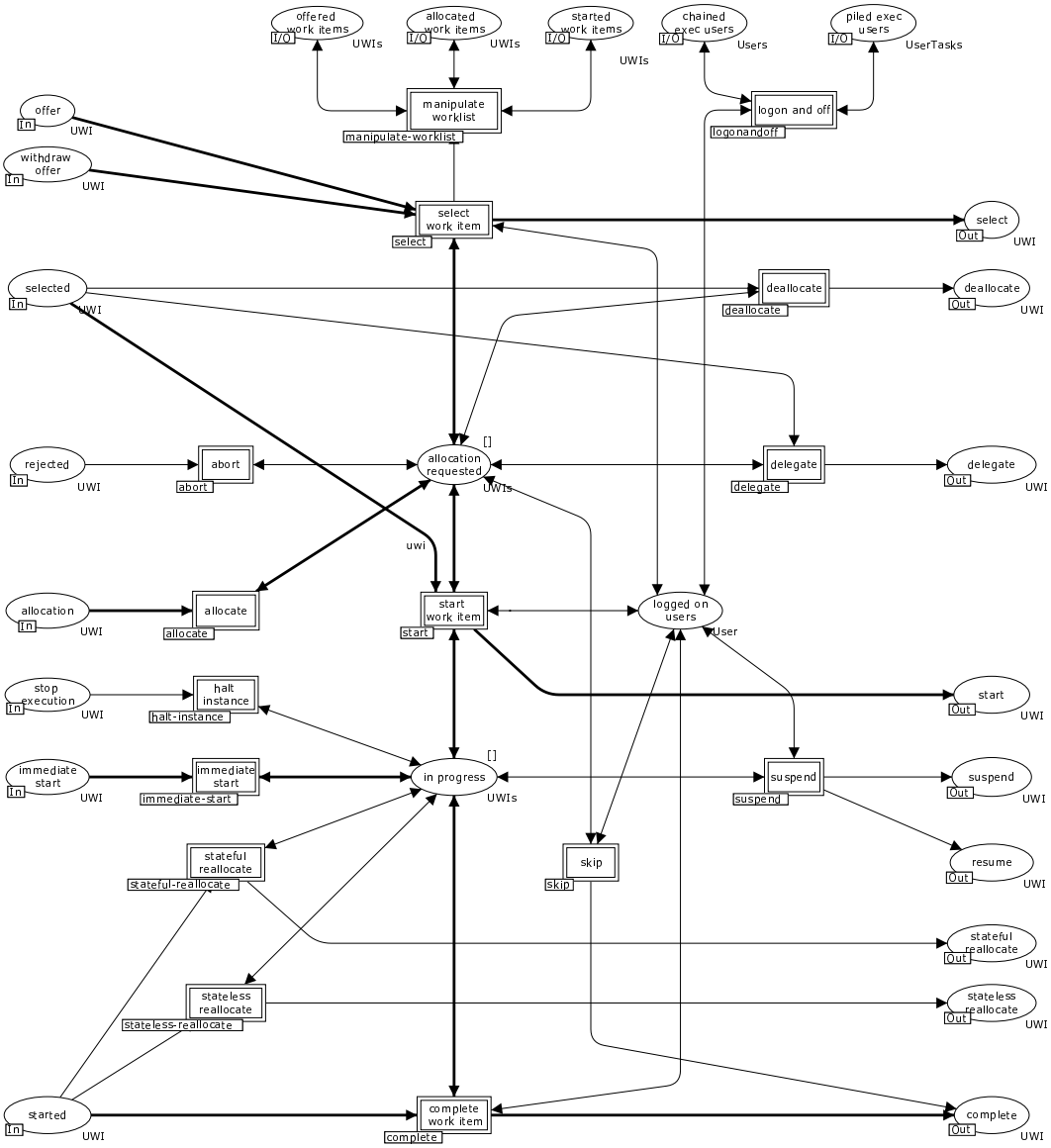


Figure 36: Work list handler process

5.4.3 Management intervention

The `management intervention` process, illustrated in Figure 37, provides facilities for a *process administrator* to intervene in the distribution of work items both for clarifying which users specific work items should be distributed to and also for escalating non-performing work items by removing them from one (or several) user's worklist and placing them on others, possibly also changing the state of the work item whilst doing so (e.g. from *offered* to *allocated*).

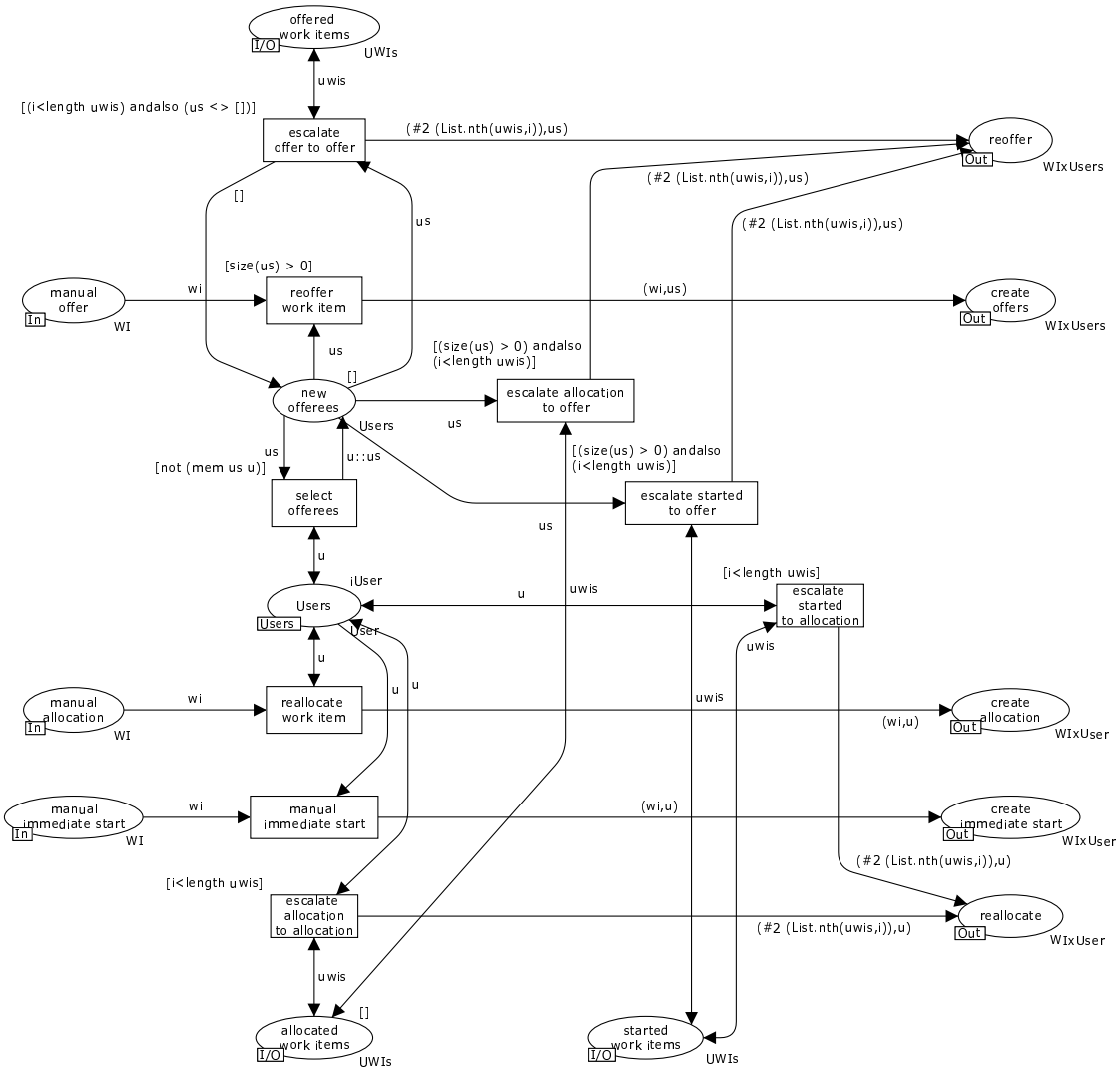


Figure 37: Management intervention process

5.4.4 Interrupt handler

The `interrupt handler` process, illustrated in Figure 38, provides facilities for intervening in the normal progress of a work item where required as a result of a request to cancel, force-fail or force-complete a work item received from external parties.

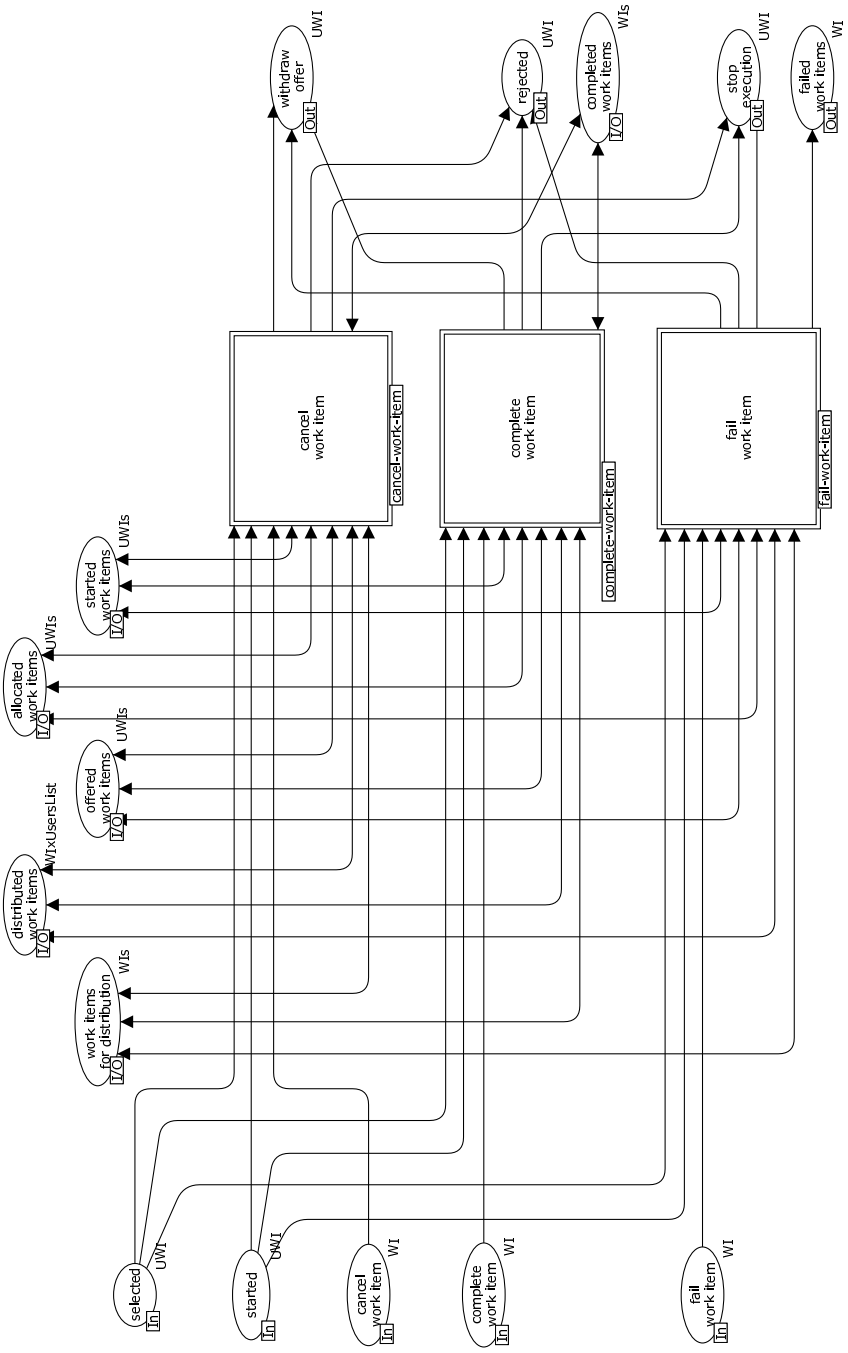


Figure 38: Interrupt handler process

5.4.5 Work item distribution process – individual activities

Work item distribution involves the routing of work items to users for subsequent execution. Each work item corresponds to an instantiation of a specific task in the overall process which has specific distribution information associated with it, in particular this identifies potential users who may execute the work item and describes the manner in which the work item should be forwarded to the user (e.g. offered to a range of users on a non-binding basis, allocated to a specific user for execution at a later time, allocated to a user for immediate start or sent to the process administrator who can make appropriate routing decisions at run-time).

The `work item routing` activity is the first part of the distribution process. It takes a new work item and determines the population to whom it should be distributed. This decision process takes into account a number of distribution directives associated with the work item including constraints based on who executed preceding work items in the same case, organizational constraints which limit the population to those who have specific organizational characteristics (e.g. job roles, seniority, members of specific organizational units), historical constraints which limit the population on the basis of previous process executions and capability-based constraints limiting the population to users who possess specific capabilities. Chained and piled execution modes (i.e. subsequent work items in a case being immediately started for a given user when them have completed a preceding item or work items corresponding to a specific task always being allocated to the same user) are also catered for as part of the distribution activity. Finally there are also facilities to limit the population to a single user on a random, round-robin or shortest-queue basis.

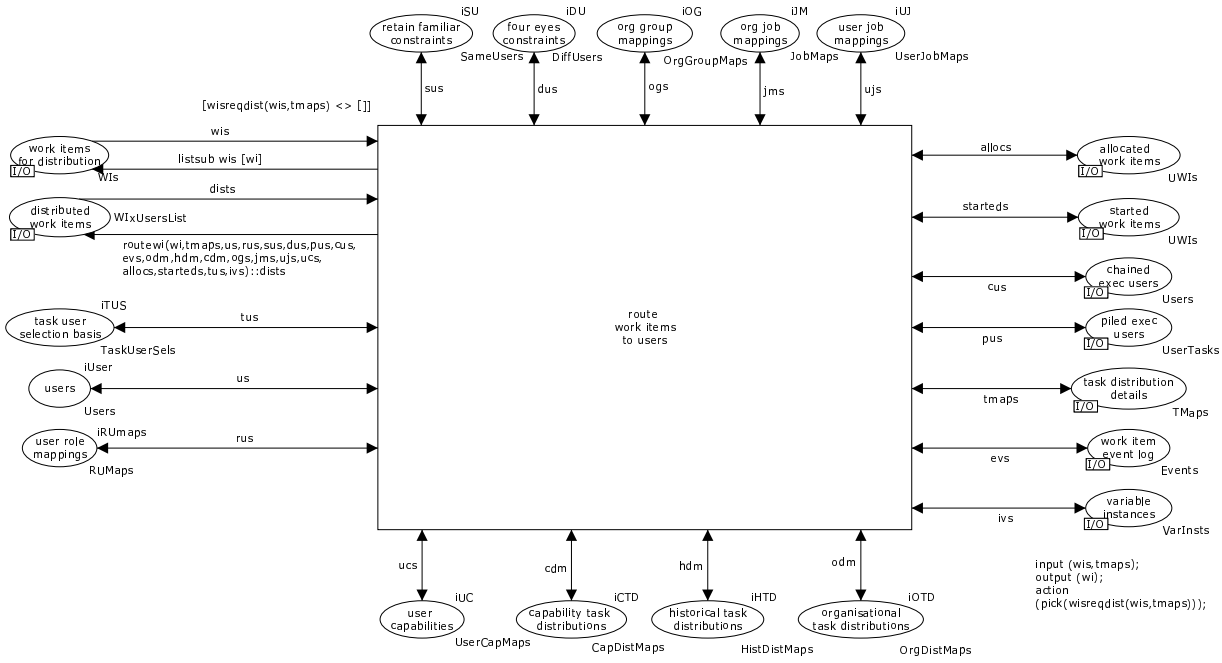


Figure 39: work item routing activity

The `process distribution failure` activity caters for the situation where the `work item routing` activity fails to identify any users to distribute a work item to. In this situation, the work item is simply noted as having failed and is not subject to any further distribution activities.

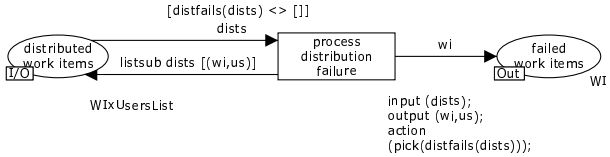


Figure 40: process distribution failure activity

The `route offers` activity takes a distributed work item which is be offered to one or more users and (based on the population identified for the work item) creates work items for each of these users which are then forwarded for insertion in their respective work lists. It is not possible for the activity to execute for a work item that is to be distributed to a user operating in chained execution mode (regardless of the original distribution scheme specified for the work item). This is necessary as any work items intended for chained execution users must be started immediately and not just offered to users on a non-binding basis. It also ensures that the work engine records the fact that the work item has been offered to the respective users.

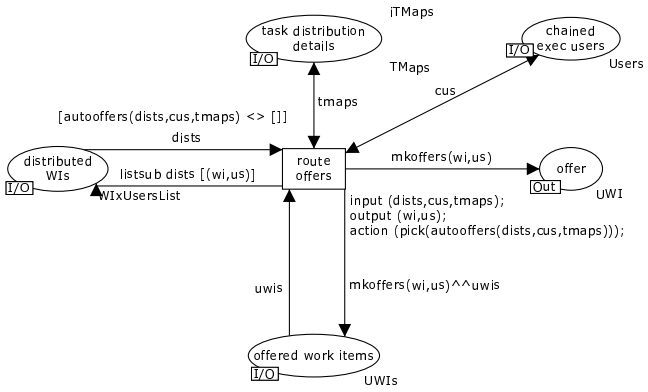


Figure 41: route offers activity

The `route allocation` activity takes a distributed work item which is be directly allocated to a user and creates a work item for insertion in that users's worklist. It also ensures that the work engine records the fact that the work item has been allocated to the user. As for the `route offers` activity, it is not possible for this activity to execute for a work item that is to be distributed to a user operating in chained execution mode (regardless of the original distribution scheme specified for the work item).

The `route immediate start` activity takes a distributed work item which is be directly allocated to a user and started immediately, and it creates a work item for insertion in that users's worklist. It also ensures that the work engine records the fact that the work item has been allocated to the user and automatically started. As for the `route offers` and `route allocation` activities, it is not possible for this

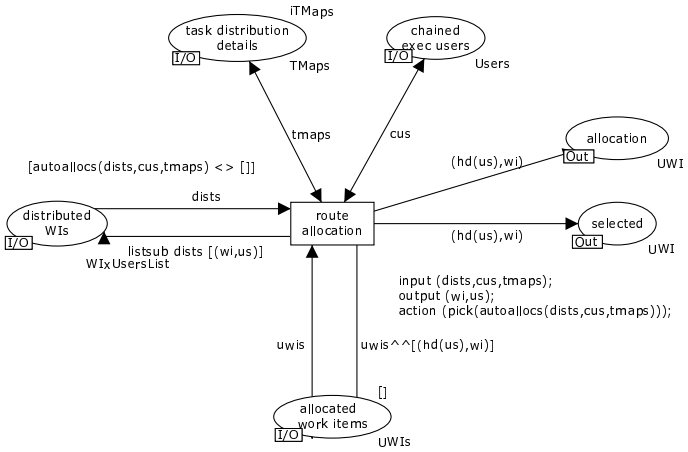


Figure 42: route allocation activity

activity to execute for a work item that is to be distributed to a user operating in chained execution mode as this is handled by a distinct activity. In order for the work item to be automatically started for the user, that user must either have an empty worklist or have the *concurrent* privilege allowing them to execute multiple work items simultaneously. If a user does not have this privilege and already has an executing item in their worklist, then the work item is regarded as having failed to be distributed and no further action is taken with it.

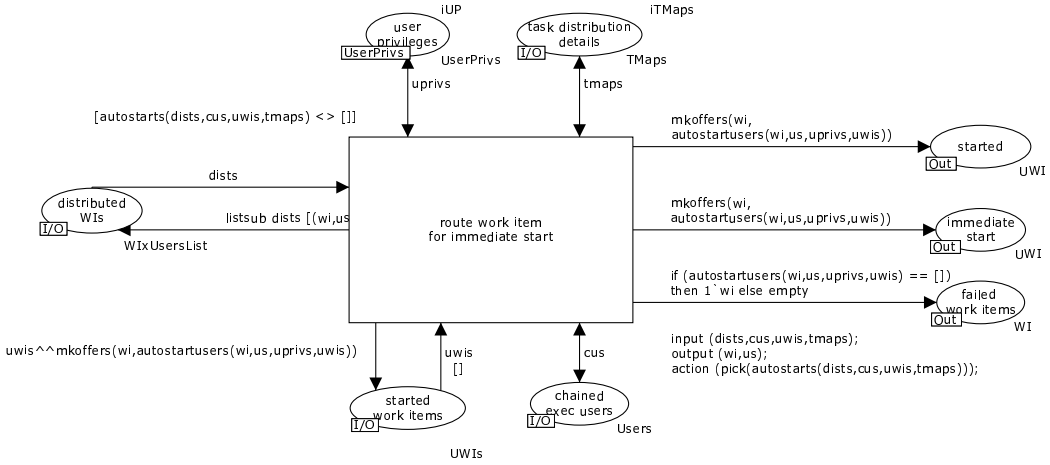


Figure 43: route immediate start activity

The `manual distribution` activity is responsible for forwarding any work items that are recorded as requiring manual distribution to the process administrator for action.

The `route manual offers` activity takes any work items that have been identified by the process administrator to be offered to users and creates work items for those users for subsequent insertion in their worklists with an *offered* status.

The `route-manual-allocation` activity takes any work items that have been identified by the process administrator to be allocated to a user and creates a work item for subsequent insertion in the user's worklist with an *allocated* status.

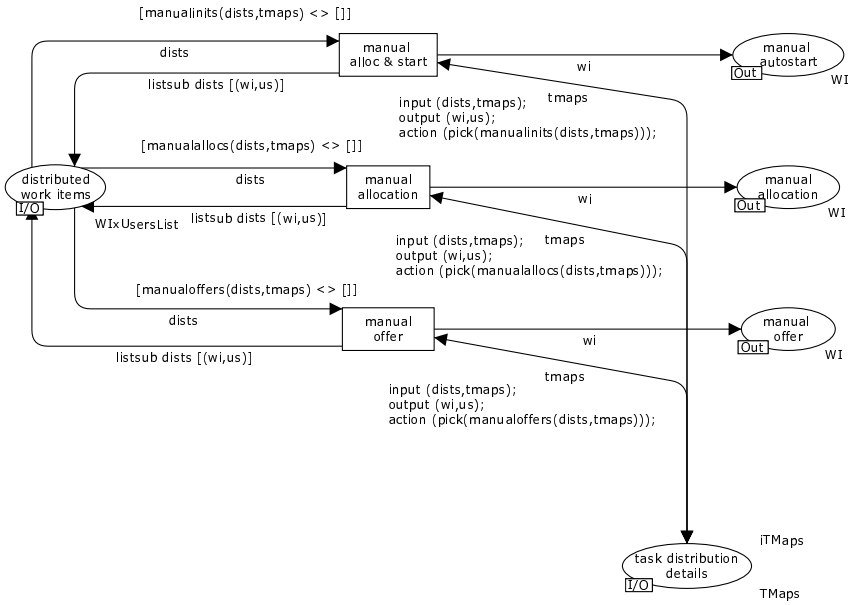


Figure 44: manual distribution activity

Figure 45: route manual offers activity

The `process manual immediate start` activity takes any work items that have been identified by the process administrator to be allocated to a user and directly started, and creates a work item for subsequent insertion in the user's worklist with a *started* status.

The `autonomous initiation` activity provides an *autonomous start* trigger to work items that execute automatically and do not need to be assigned to a user. It is anticipated that this trigger would be routed outside of the process environment in order to initiate a remote activity such as a web service.

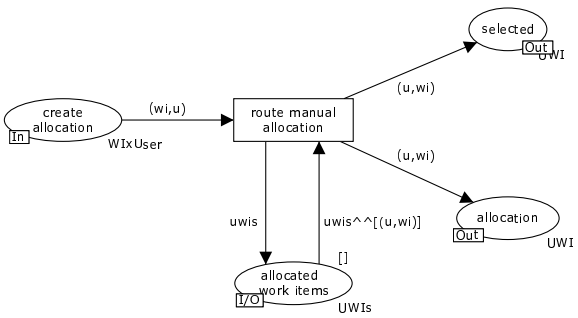


Figure 46: route manual allocation activity

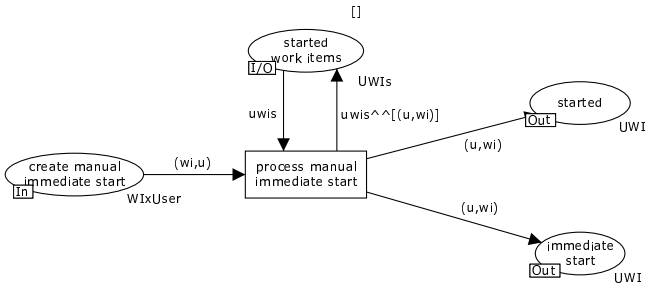


Figure 47: process manual immediate start activity

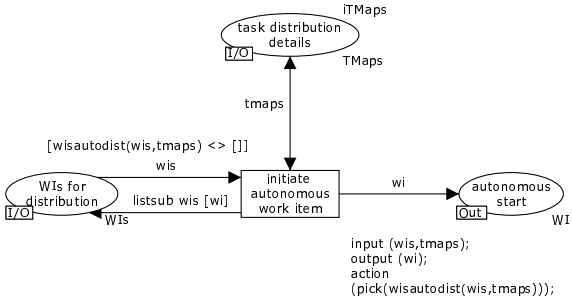


Figure 48: autonomous initiation activity

The `autonomous completion` activity responds to a trigger from an external autonomous activity indicating that it has finished. It forwards this response to the process engine allowing it to enable subsequent activities in the process.

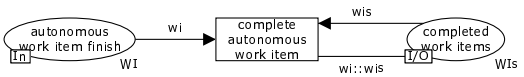


Figure 49: autonomous completion activity

The `process selection request` activity responds to a request from a user for a work item to be allocated to them. If the work item is one that is defined as being started by the user, then the work item is recorded as having an *allocated* status and the user is advised that it has been allocated to them. If is one that is automatically started on allocation, then it is recorded as having a *started* status and the user is advised of this accordingly. In both cases, any offers pending for this work item for other users are withdrawn thus ensuring that these users can not request that the work item be subsequently allocated to them.

The `reject offer` activity handles requests from users for a work item previously offered to them to be allocated to them where the work item in question has already been allocated to a different user. These requests are rejected and the requesting user is advised accordingly.

The `process start request` activity responds to a request from a user to start a work item. It can only start the work item if it has been previously allocated to the same user. Where it has, the status of the work item is changed from *allocated* to *started* and the user is advised accordingly.

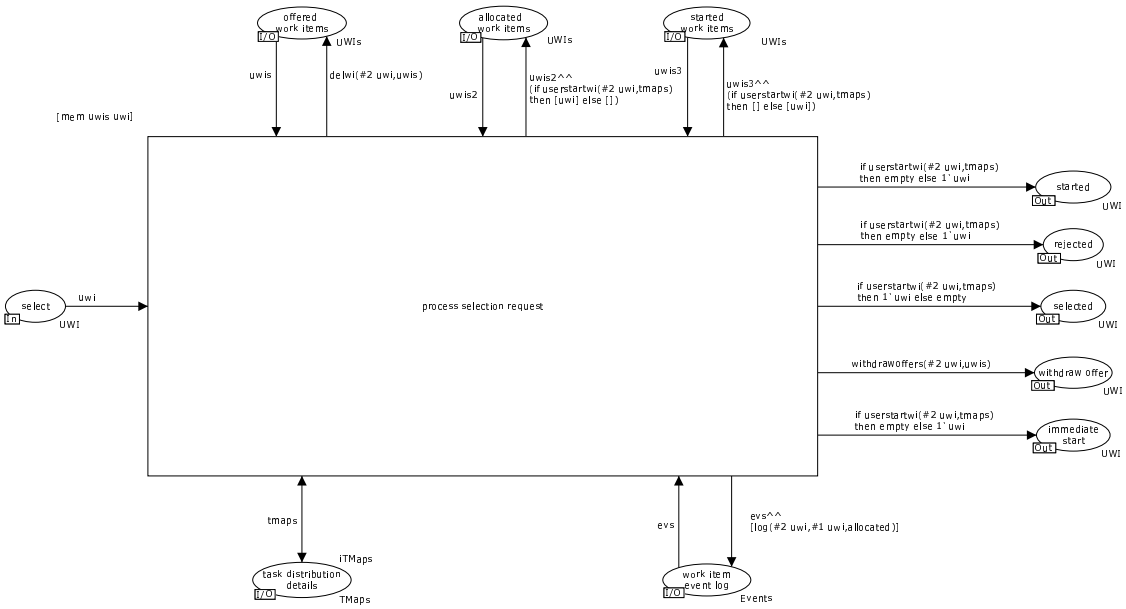


Figure 50: process selection request activity

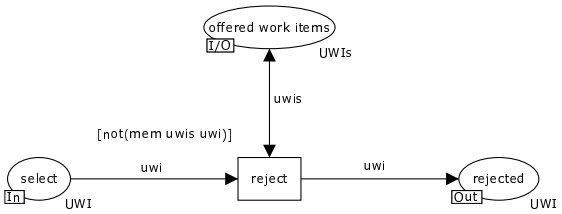


Figure 51: reject offer activity

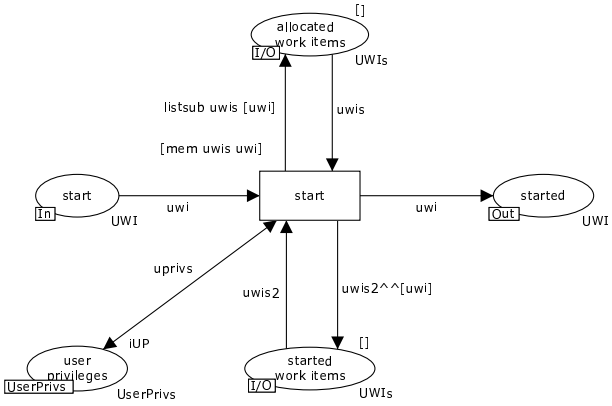


Figure 52: process start request activity

The `suspension resumption` activity responds to suspend or resume requests from a user, causing a specified work item to be recorded as *suspended* or (re-)*started*.

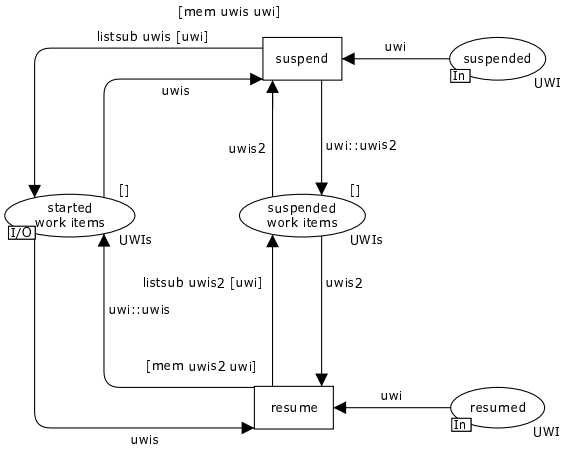


Figure 53: suspension resumption activity

The `process completion` activity responds to user requests to record as *completed* work items that are currently recorded as being executed by them. Once this is done, the fact that the work item has been completed is signalled to the high-level process controller enabling subsequent work items to be triggered. The completion of the work item is also recorded in the execution log. For the purposes of this model, we only record instances where a work item is completed as these are relevant to other work distribution activities however in actual practice, it is likely that all of the activities identified in this work distribution model would be logged.

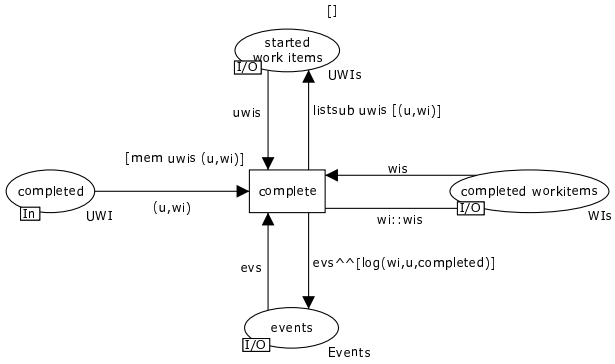


Figure 54: process completion activity

The `route delegation` activity responds to a request from a user to allocate a work item currently recorded as being allocated to them to another user. As part of this activity, the record of the work item being allocated to the current user is removed.

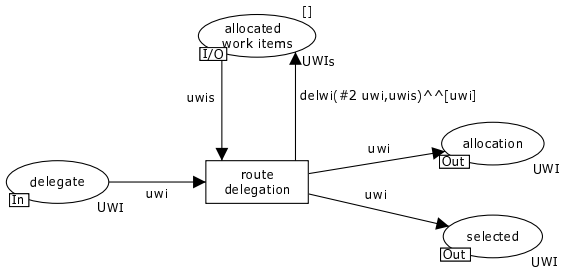


Figure 55: route delegation activity

The `process deallocation` activity responds to a request from a user to make a work item recorded as being allocated to them available for redistribution. As part of this activity, the record of the work item being allocated to the current user is removed.

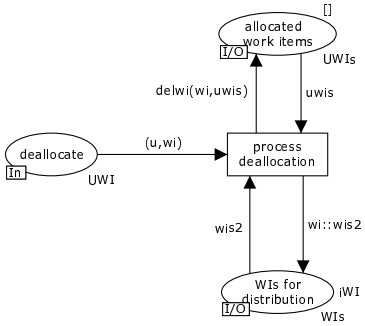


Figure 56: process deallocation activity

The `state oriented reallocation` activity responds to a request from a user to migrate a work item that they are currently executing to another user. Depending on whether the activity is called on a stateful or stateless basis, the current working state of the work item is either retained or discard when it is passed to the new user respectively. As part of this activity the work item is directly inserted into the worklist of the new user with a *started* status.

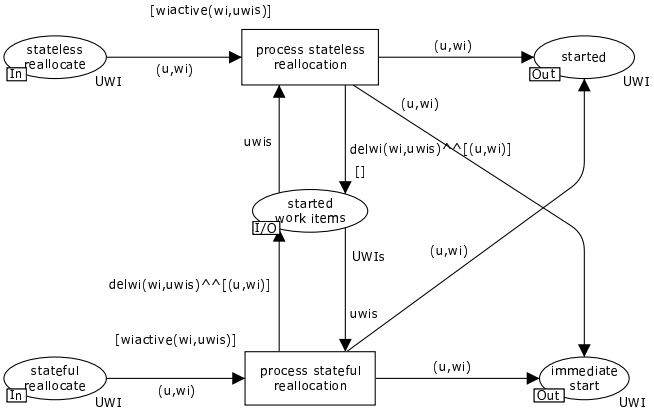


Figure 57: state oriented reallocation activity

The `route reoffers` activity responds to requests from the process administrator to escalate specific work items that are currently in a *offered*, *allocated* or *started* state by repeating the act of offering them to prospective users. The process administrator identifies who the work item should be offered to and suitable work items are forwarded to their work lists. Any existing worklist entries for this work item are removed from user's work lists. The state of the work item is also recorded as being *offered*.

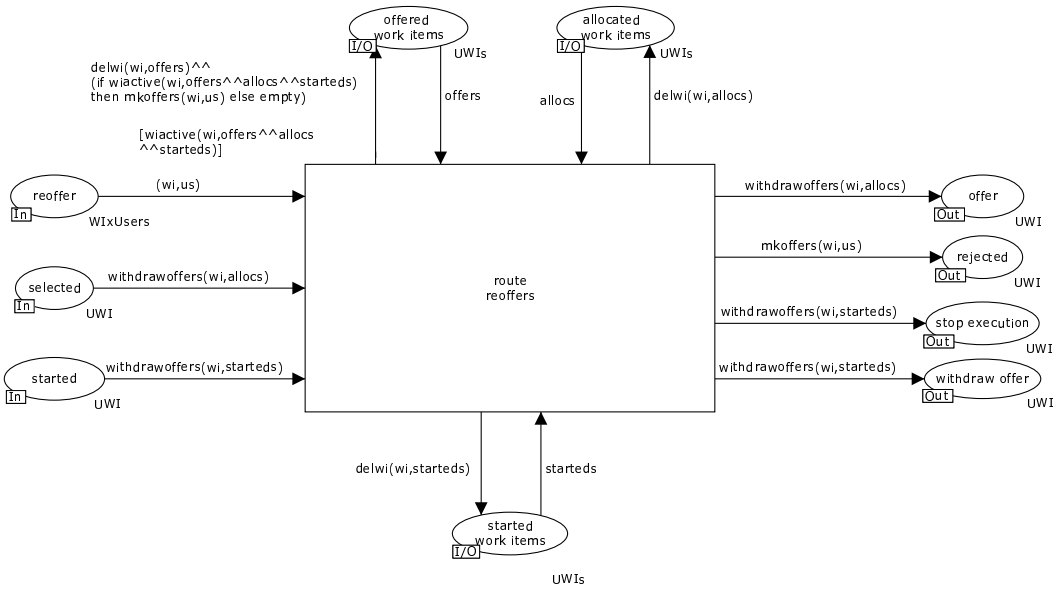


Figure 58: route reoffers activity

The `route reallocation` activity responds to requests from the process administrator to escalate specific work items that are currently in a *offered*, *allocated* or *started* state by directly allocating them to a specific (and most likely different) user. The process administrator identifies who the work item should be allocated to and a suitable work item is forwarded to their work list. Any existing worklist entries for this work item are removed from user's work lists. The state of the work item is also recorded as being *allocated*.

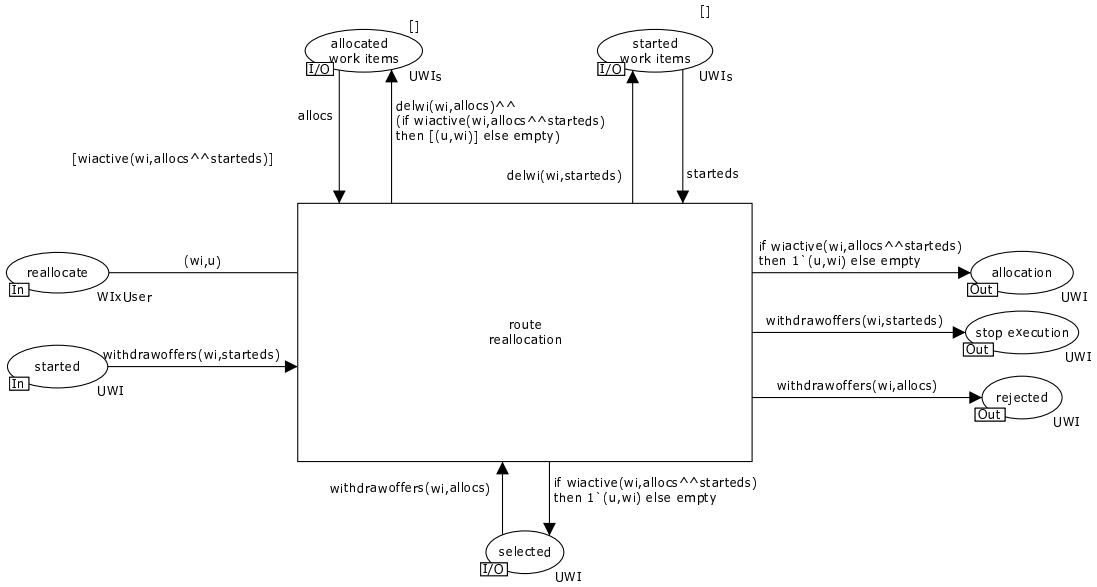


Figure 59: route reallocation activity

The `reject reoffer` activity responds to requests from the process administrator to escalate a work item by repeating the act of offering it where the work item is not in the *offered*, *allocated* or *started* state. This may be because the work item has failed, already been completed or is currently being redistributed at the request of a user. In any of these situations, the action is to simply ignore the *reoffer* request.

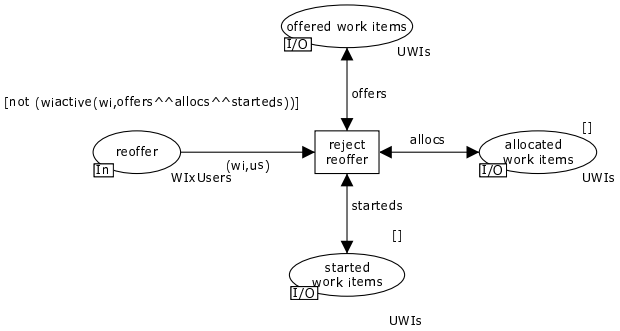


Figure 60: reject reoffer activity

The `reject reallocation` activity responds to requests from the process administrator to escalate a work item by directly allocating it to a user it where the work item is not in the *allocated* or *started* state. This may be because the work item has failed, already been completed or is currently being redistributed at the request of a user. In all situations, the action is to simply ignore the *reallocate* request.

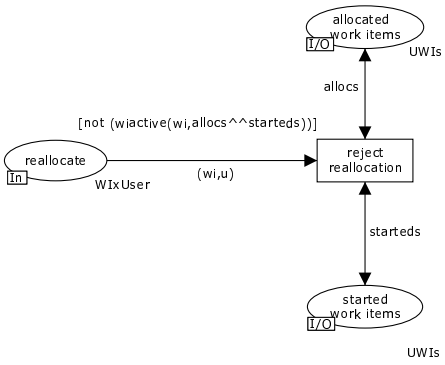


Figure 61: reject reallocation activity

5.4.6 Work list handler activities

The `select` activity handles the insertion of work items offered by the process engine into user worklists. It also handles the deletion of previously offered work items that have been subsequently allocated to a different user.

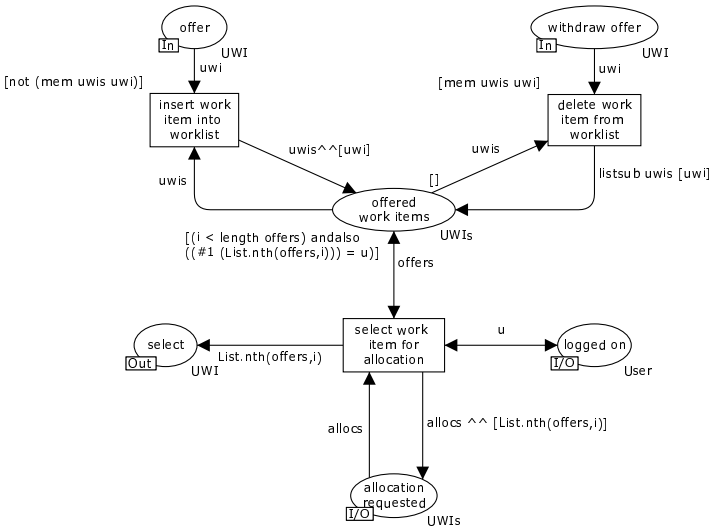


Figure 62: select activity

The `allocate` activity manages the insertion of work items directly allocated to a user into their work list.

The `start` activity provides the ability for a user to commence execution of a work item that has been allocated to them. In order to start it, they must be logged in, have the *choose* privilege which allows them to select the next work item that they will execute and either not be currently executing any other work items or have the *concurrent* privilege allowing them to execute multiple work items at the same time.

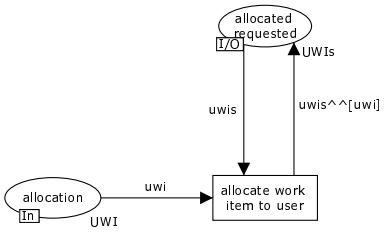


Figure 63: allocate activity

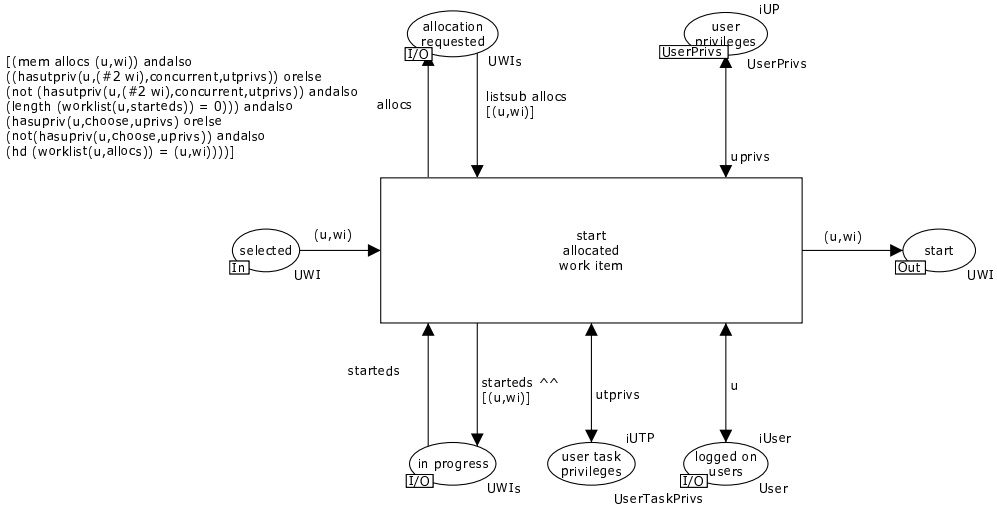


Figure 64: start activity

The `immediate start` activity allows work items to be directly inserted in a user's worklist by the process engine with a currently executing status. As with the `start` activity, this is only possible if the user is not currently executing any other work items or has the *concurrent* privilege.

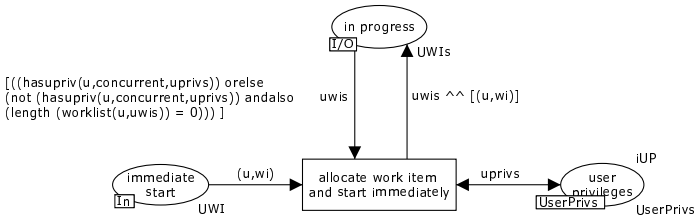


Figure 65: immediate start activity

At the request of the process engine, the `halt instance` activity immediately removes a specified work item from a user's worklist.

The `complete` activity allows a user to mark a work item as complete, resulting in it being removed from their worklist. This can only occur if the user is logged in and the process engine has recorded the fact that the user has commenced work on the activity.

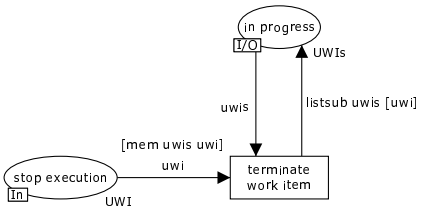


Figure 66: halt instance activity

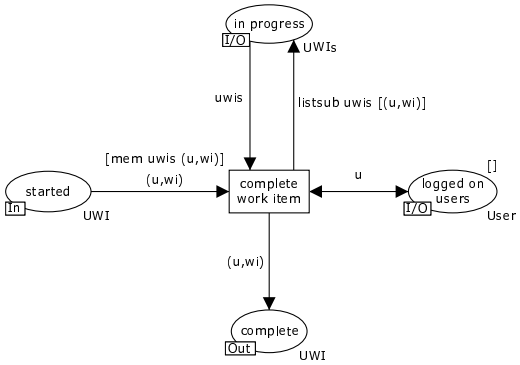


Figure 67: complete activity

The `suspend` activity provides the ability for a user to cease work on a specified work item and resume it at some future time. In order to suspend a work item, they must possess the *suspend* privilege. To resume it, they must either have no other currently executing work items in their worklist or possess the *concurrent* privilege.

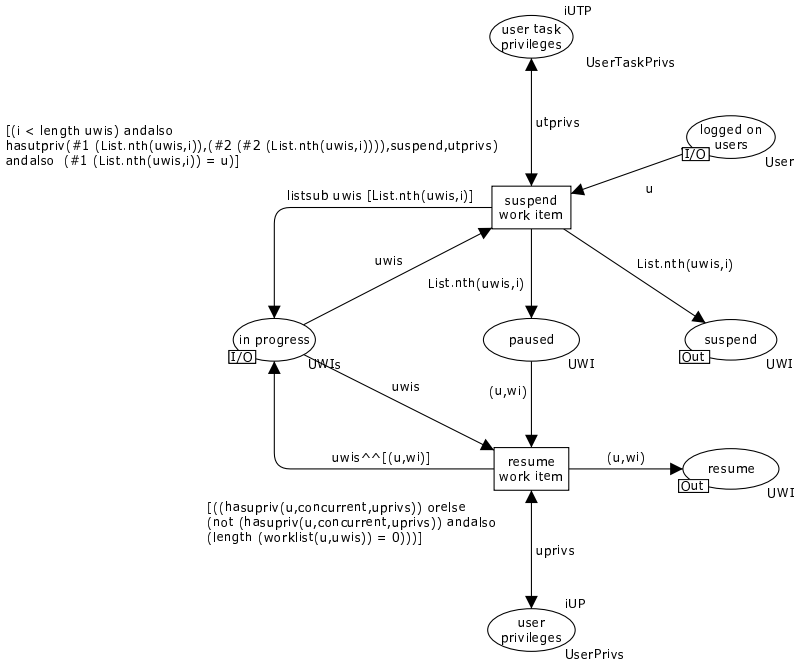


Figure 68: suspend activity

The `skip` activity provides the user with the option to *skip* a work item that is currently allocated to them. This means it transitions from a status of *allocated* to one of *completed*. They can only do this if they have the *skip* privilege.

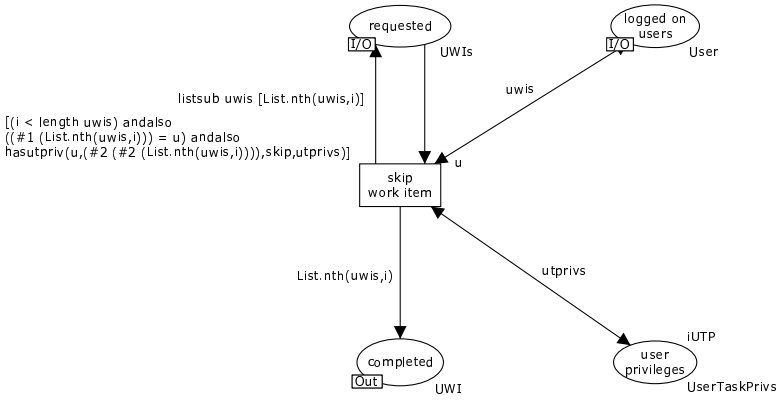


Figure 69: skip activity

The `delegate` activity allows a user to pass a work item currently allocated to them to another user. They can only do this if they have the *delegate* privilege and the process engine has recorded the fact that the work item is allocated to them.

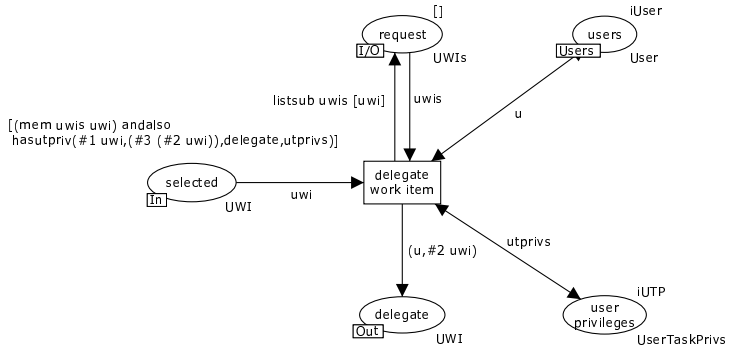


Figure 70: delegate activity

The `abort` activity operates at the request of the process engine and removes a work item currently allocated to a user from their worklist. If they have also requested that its execution be started, this request is also removed.

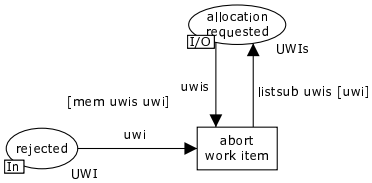


Figure 71: abort activity

The `stateful reallocate` activity allows a user to reassign a work item that they are currently executing to another user with complete retention of a state i.e.

the next user will continue executing the work item from the place at which the preceding user left off. In order for the user to execute this activity, the work item must be recorded by the process engine as currently being executed by them and they must possess the *reallocate* privilege.

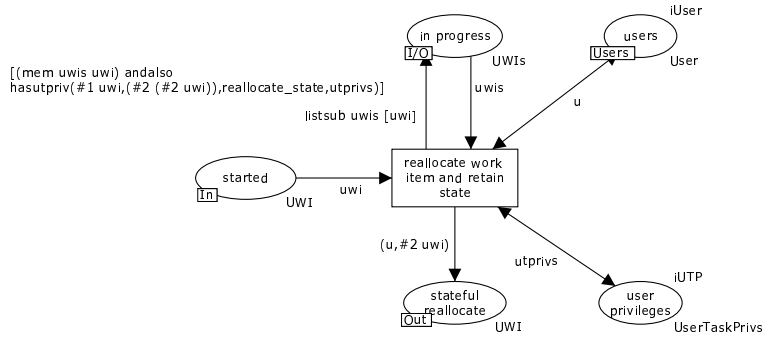


Figure 72: stateful reallocate activity

The `stateless reallocate` activity operates in a similar way to the `stateful reallocate` activity although in this case, there is no retention of state and the subsequent user effectively restarts the work item. In order for the user to execute this activity, the work item must be recorded by the process engine as currently being executed by them and they must possess the *delegate* privilege.

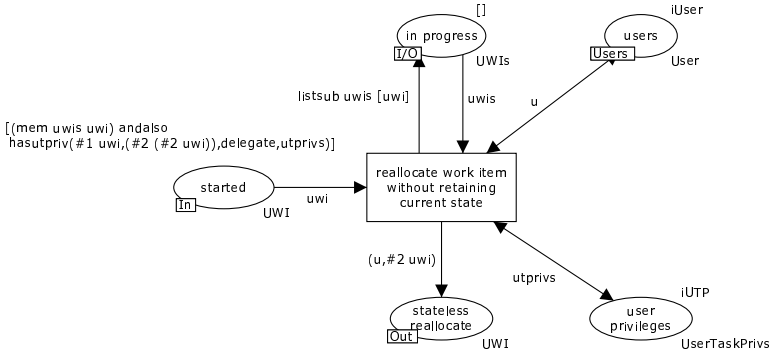


Figure 73: stateless reallocate activity

The `manipulate worklist` activity provides the user with the facility to reorder the items in their work list and also to restrict the items that they view from it. In order to fully utilize the various facilities available within this activity, the user must possess the *reorder*, *viewoffers*, *viewallocs* and *viewexecs* privileges.

The `logonandoff` activity supports the logonand logoff of users from the process engine. A number of worklist functions require that they be logged in in order to operate them. Depending on the privileges that they possess, the user is also able to trigger chained and piled execution modes as part of this activity. This requires that they possess the `chainedexec` and `piledexec` privileges respectively. Only one user can be nominated to execute a given task in piled execution mode at any given time.

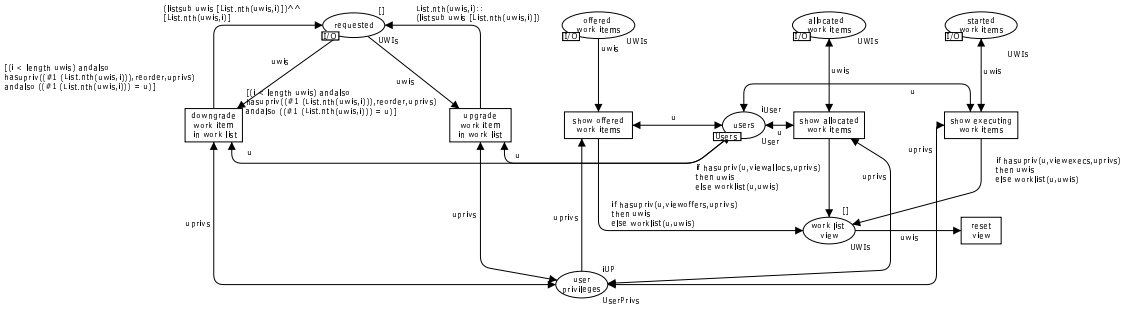


Figure 74: manipulate worklist activity

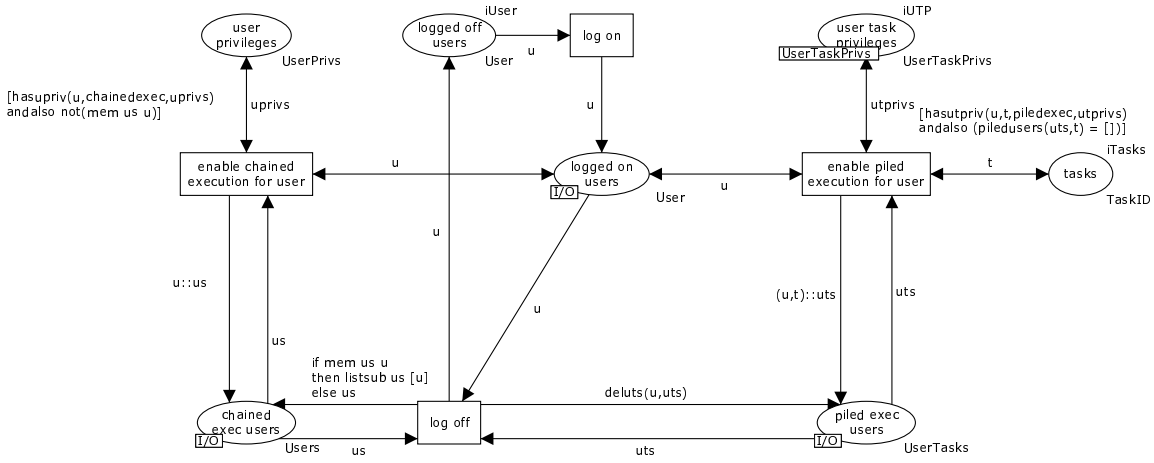


Figure 75: logonandoff activity

5.4.7 Interrupt handling processing

The `cancel work item` process provides facilities to handle a cancellation request initiated from outside the work distribution process. It enables a work item currently being processed (and in any other state than *completed*) to be removed from both the work distribution and worklist handler processes. No confirmation is given regarding work item cancellation.

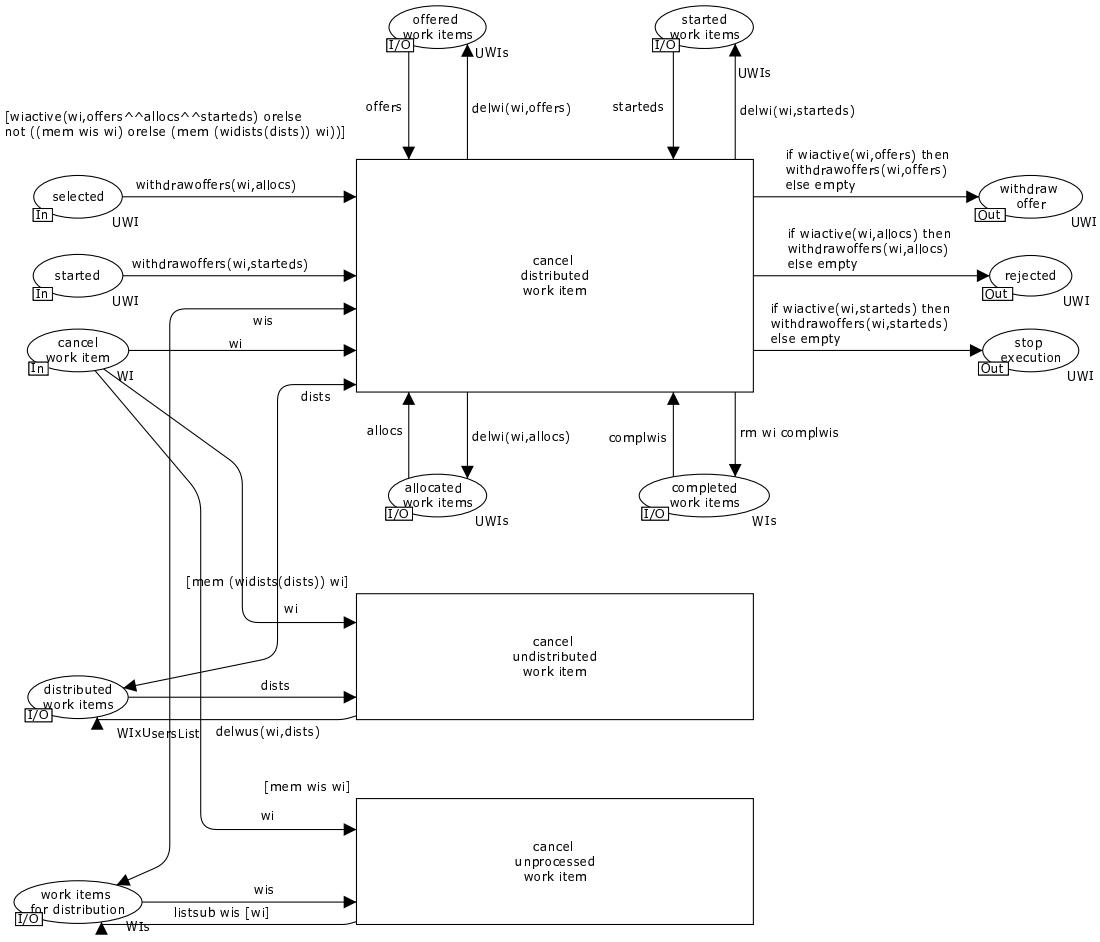


Figure 76: cancel work item process

The `complete work item` process provides facilities to handle a *force-complete* request initiated from outside the work distribution process. It enables a work item currently being processed (and in any other state than *completed*) to be marked as having completed and for it to be removed from both the work distribution and worklist handler processes. If the request is successful, the work item completion is indicated via a token in the `completed work items` place which is passed back to the high-level `process execution` process (i.e. as illustrated in Figure 23).

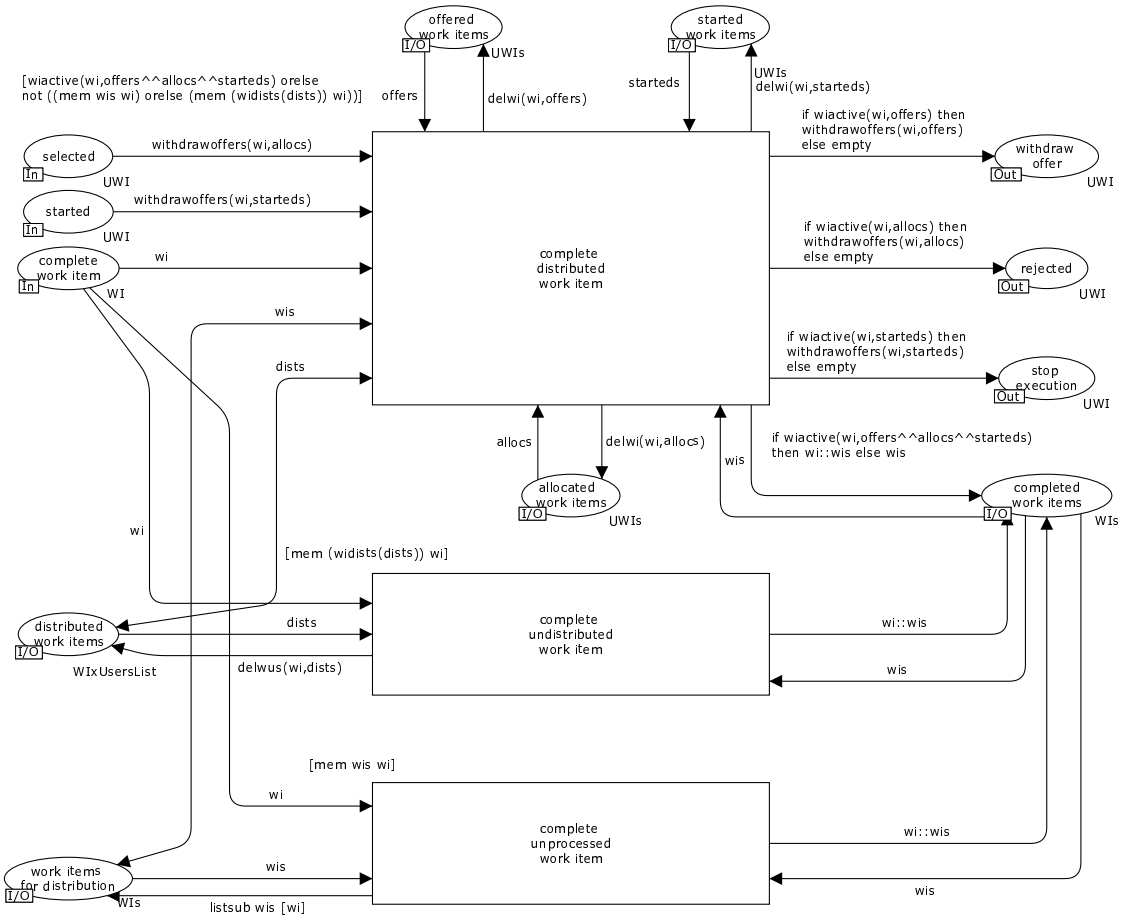


Figure 77: complete work item process

The `fail work item` process provides facilities to handle a force-fail request initiated from outside the work distribution process. It enables a work item currently being processed (and in any other state than *completed*) to be marked as having failed and for it to be removed from both the work distribution and worklist handler processes. If the request is successful, the work item failure is indicated via a token in the `failed work items` place which is passed back to the high-level `process execution` process.

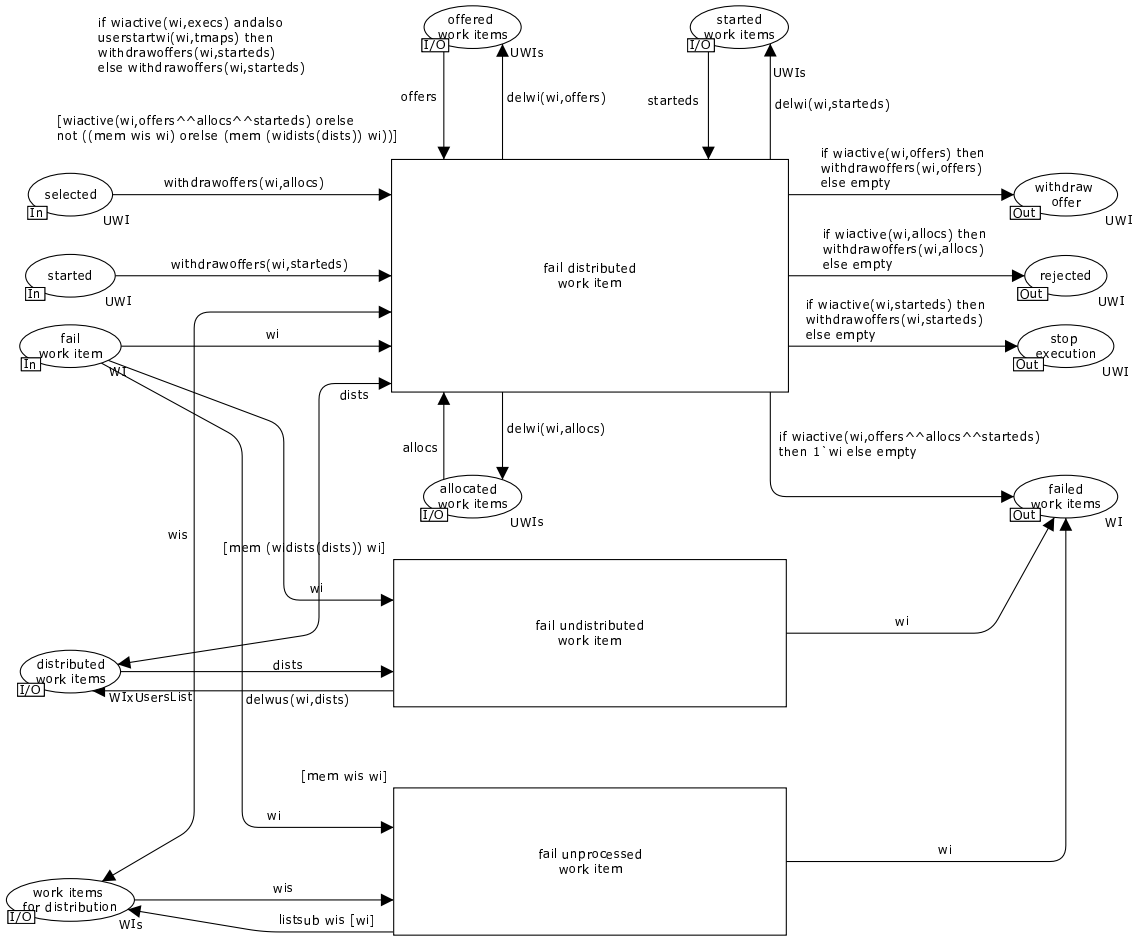


Figure 78: fail work item process

6 Worked Example

In this section, we provide a worked example of a complete *new*YAWL model in order to clarify the correspondences between the abstract syntax and semantic models and also to illustrate the operation of the various transformation functions which allow some of the new language elements to be catered for within the existing YAWL framework. Having mapped a candidate *new*YAWL model from a static design time representation to an instance of the semantic model (in the form of an initial marking), we can then *execute* the *new*YAWL model. This occurs in the CPN Tools environment and allows us to examine how individual *new*YAWL language elements are catered for and how they are integrated in order to facilitate the execution of process instances.

Figure 79 illustrates the *new*YAWL model that we will examine for the purposes of this exercise. It includes several of the new language elements in a model that comprises two distinct nets.

The data elements defined for the model are described in Table 11. The specific construct to which each of them is bound is identified along with those that are passed as parameters to or from any of the tasks during execution.

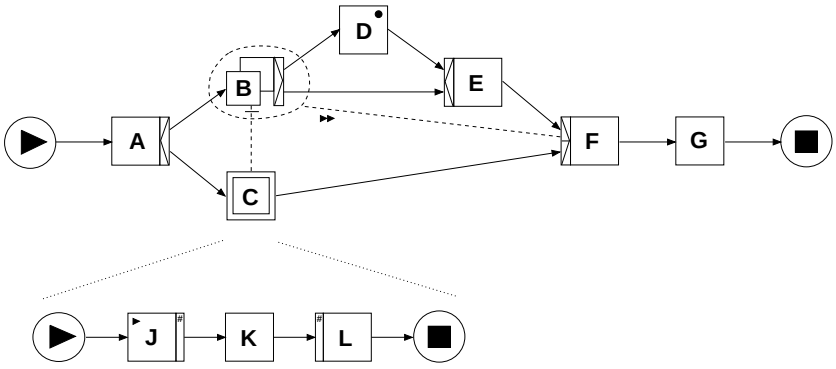


Figure 79: Working model

VarID	Name	Type	Binding	InPar	OutPar
v1	wv1	int	p1		
v2	wv2	emprecs	p1	A	A
v3	fv1	string	f1	C	C
v4	fv2	int	f1		
v5	cv1	string	p1	F	F
v6	bv1	string	p2	J	J
v7	sv1	string	s1	E	E
v8	tv1	int	A		
v9	tv2	string	B		
v10	tv3	string	E		
v11	tv4	int	F		
v12	tv5	string	J		
v13	mv1	string	B		
v14	mv2	int	B		
v15	mv3	string	B		

Table 11: Data elements in working model

There is also a simple work distribution model which comprises six users: *user1* ... *user6* organized into three roles: *role1* ... *role3*. Each user is associated with one or more jobs *j1* ... *j5* each of which are attached to a specific organizational unit. The specific work distribution criteria for each task are listed in Table 12

The complete population of the abstract syntax model corresponding to this model is listed in Appendix B. As well as mirroring the control-flow elements depicted in Figure 79, it also includes details of the data and resource (i.e. work distribution) considerations associated with this model.

In order to utilize this model, it first needs to be "unfolded" using the transformation functions described in Section 4.2. This yields an augmented control-flow model as illustrated in Figure 80. The persistent trigger, partial join and while loop constructs appearing in the initial model have been transformed out of this model. This has necessitated some changes to the process model, in particular some tasks and new arcs have been added. This also requires additional work distribution directives (in

Task	Routing	Interaction Strategy	Other Constraints
A	user1, user2, user3, user4, user5, user6	system,system,system	org dist function org1()
B	role1, role3	system,resource,resource	hist dist function lastA()
D	user1, user2, user3, user4	resource,system,resource	
E	role1, role2, role3	system,system,resource	
F	user1	resource,resource,resource	
G	AUTO	system,system,system	
J	user1, user2, user3, user4	system,resource,system	cap dist function capval1()
K	user1, user2, user3, user4	system,resource,resource	same user as J
L	role2, role3	system,resource,resource	different user to K

Table 12: Work distribution in working model

most cases added tasks are AUTO) and some additional data passing specifications (e.g. task J_{L1} shares access to the same data elements as task J). Specific additions or changes to the abstract syntax model are identified in Appendix C.

Finally, the complete unfolded *new*YAWL specification is transformed to an initial marking for the semantic model (included in Appendix D) based on the transformations identified in section 4.2. By including this set of markings in the CPN model on which the *new*YAWL semantic model is based, it is possible to execute the process model. In the following sections, we illustrate the operation of some of the more significant new constructs supported by *new*YAWL.

6.1 Initiation of process instance

Figure 81 illustrates the manner in which a new process instance is started. Two distinct inputs are required for initiation: the *ProcessID* and *CID* to identify the particular instance being started and the list of folders being assigned to it. The absence of a bold border around the `start case` transition indicates that it is not yet able to fire as a precondition for its enablement has not been satisfied. In this case, the precondition that the *wv2* variable have a value is not met as it does not yet exist.

Once a value is inserted into the `variable instances` place for this data element, the transition is enabled and can fire. Figure 82 illustrates the results of the case being enabled: (1) a token is placed in the input condition for the process (see the `process state` place) and the required case and scope variables are created.

6.2 Task enablement and initiation

Once a token exists in the input condition for the process, the first task can potentially be enabled subject to preconditions, locks and mandatory parameters being available.

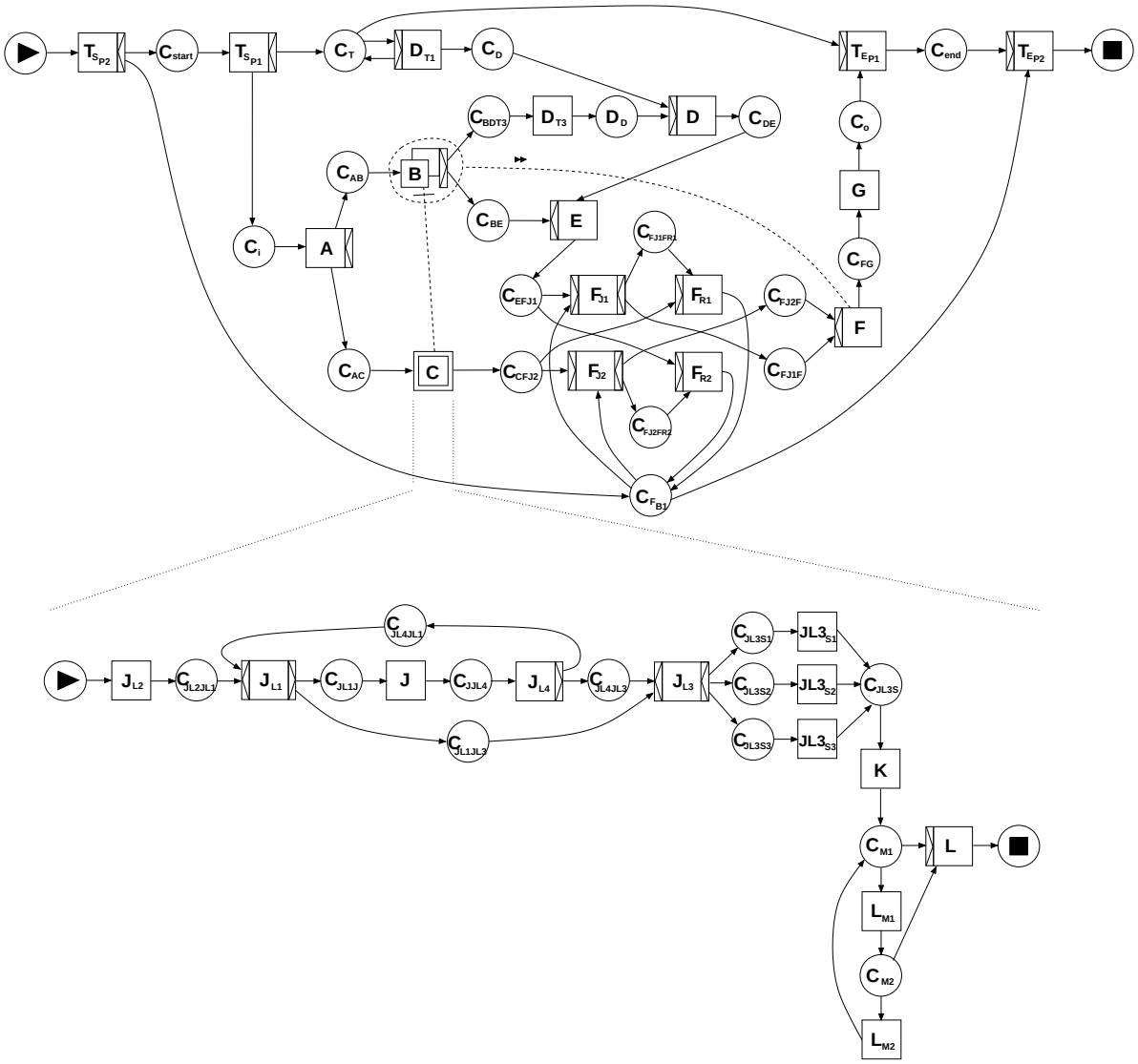


Figure 80: Unfolded working model

The enablement of a specific task instance is handled by the `enter` transition. In Figure 83, we consider the triggering of a work item for task A. Initially, this cannot proceed because variable *wv1* does not have a defined value. Once a value is inserted for this variable, all preconditions for the task are met and it is enabled as illustrated in Figure 84. Upon the firing of the `enter` transition, a work item is created for the task which can subsequently be assigned to a resource for execution. The relevant data elements required for the work item are created. In this case, task variable *tv1* is created and is assigned a value based on applying the `inc` function to global variable *wv1*.

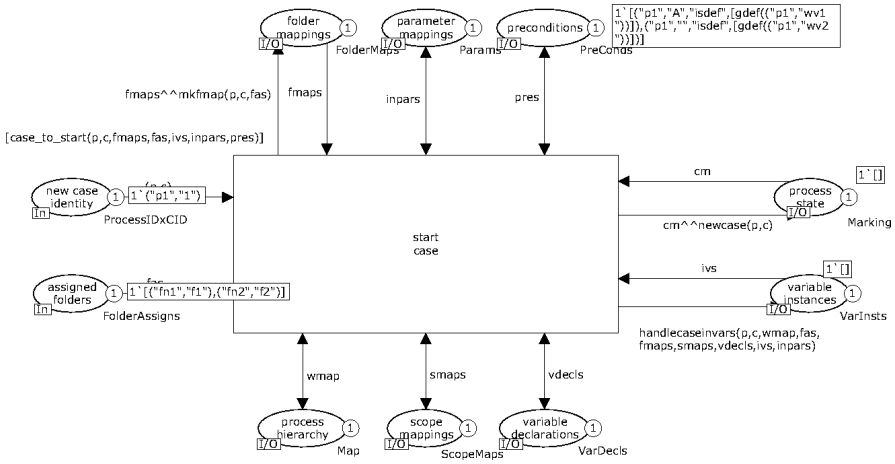


Figure 81: Working model – case start

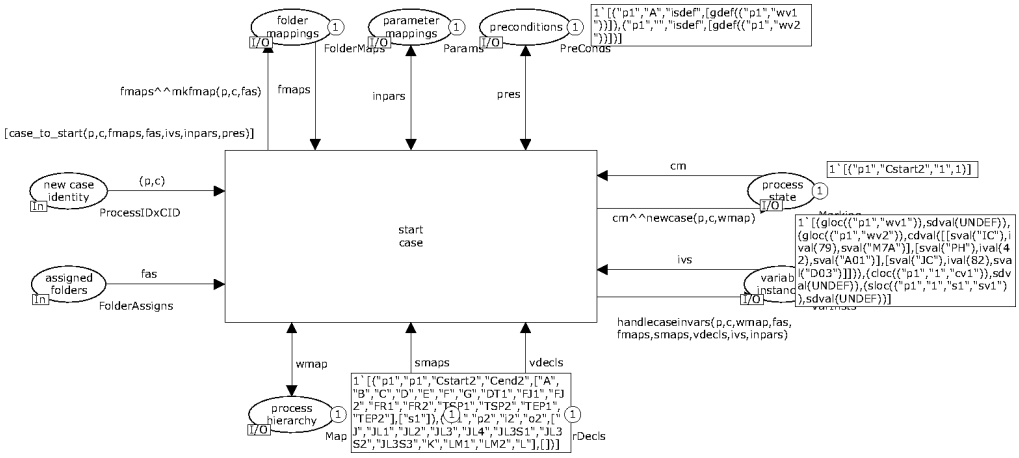


Figure 82: Working model – case start after enablement

6.3 Work item routing

Immediately after a work item has been enabled, it is routed to an appropriate resource for subsequent execution. The transition which does this is illustrated in Figure 86. The routing decision is based on a number of factors as illustrated by the number of input places to the transition. In this case, the work item corresponds to task A and will be offered to all users providing are members of organizational group A. An organizational task distribution function (org1) is used to determine the appropriate group of users. The results of apply this function as part of the work item routing are indicated in Figure 87 and the work item is subsequently offered to *user1* and *user2*.

There is a separation of concerns between the main `work item distribution` process which normally occurs as part of (and resides with) the main process engine and the `work list handler`. The `work list handler` is the user-facing software that manages their interaction with the process environment. Unlike the `work item distribution` process, which runs in response to distribution requests and reactions from users, the `work list handler` runs under the auspices of an individual users based on their response to individual requests. The interface between the two pro-

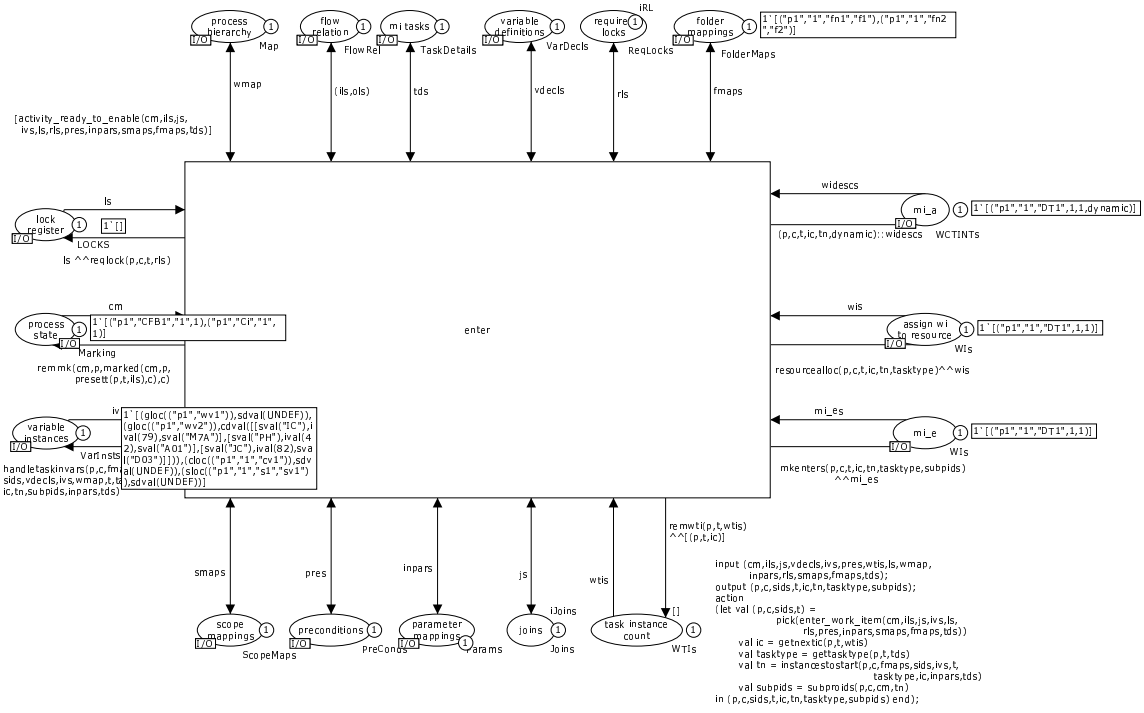


Figure 83: Working model – prior to enablement of task A

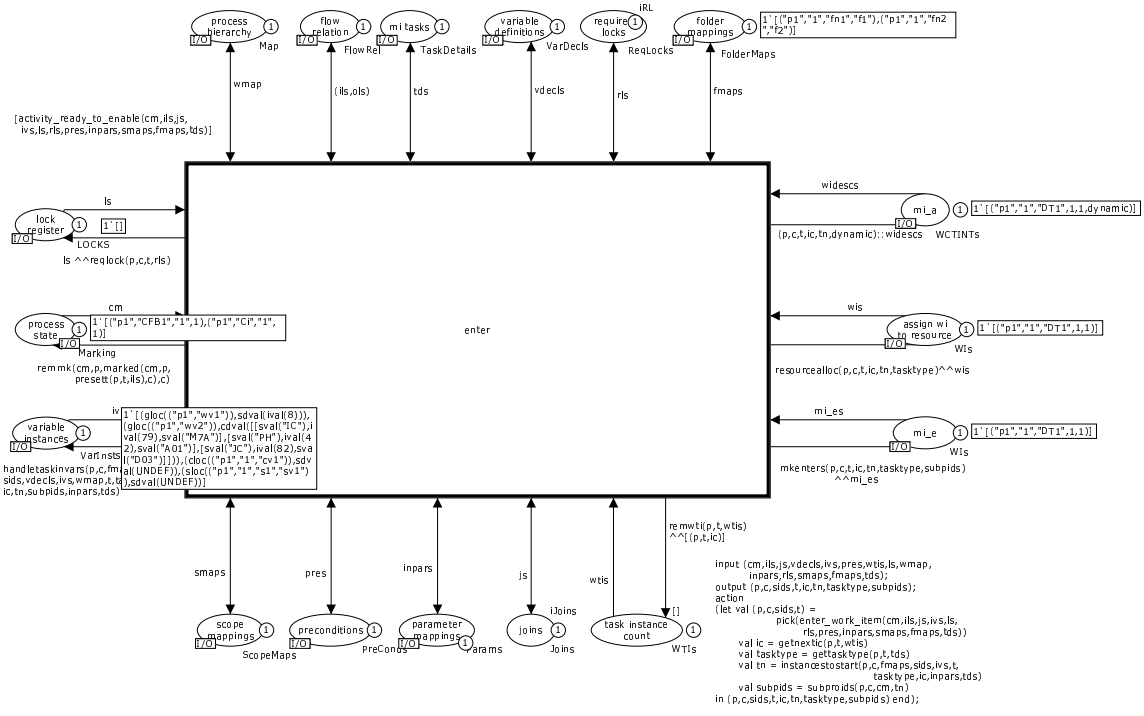


Figure 84: Working model – enablement of task A

cesses is defined by the places in the two substitution transitions that represent them in the `work distribution` process. Continuing with the distribution of work items

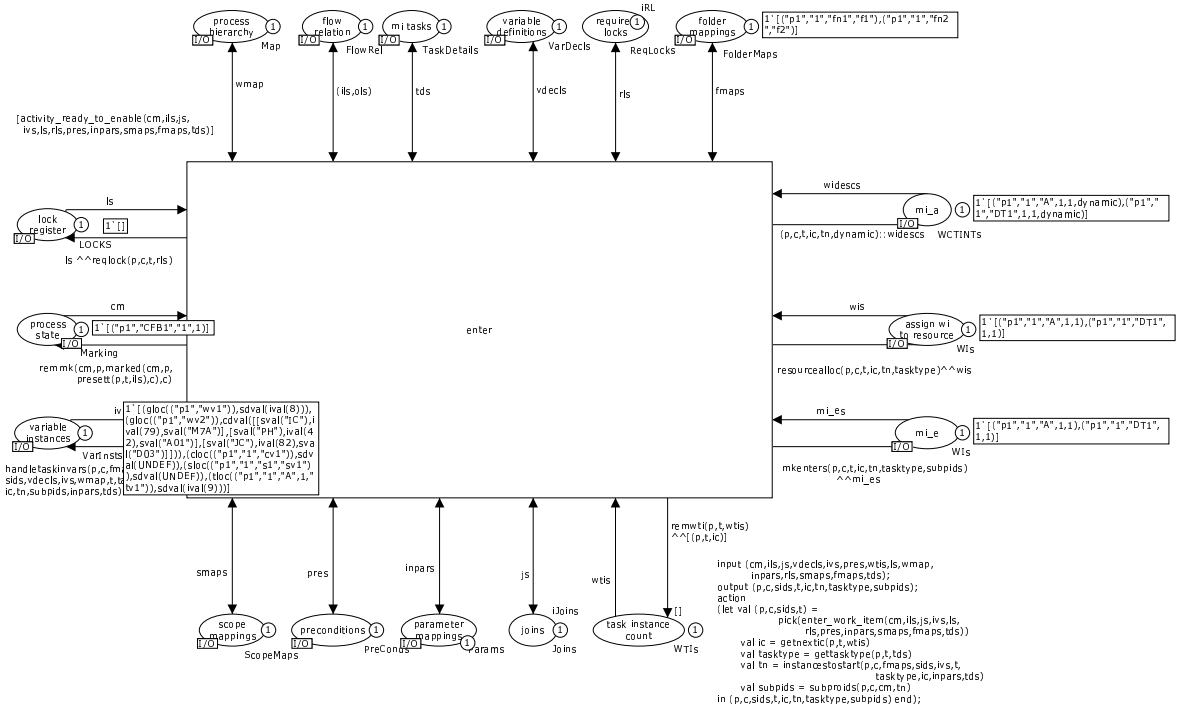


Figure 85: Working model – initiation of task A & work item creation

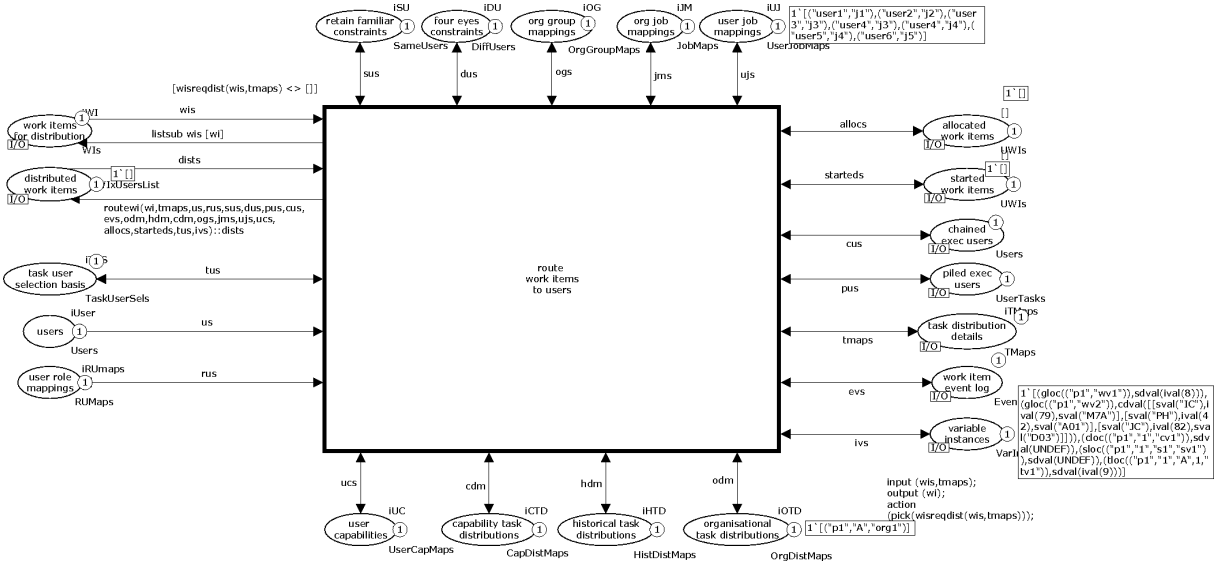


Figure 86: Working model – work item routing

associated with task A, the offer of these work items is illustrated in Figure 88 with the offer requests indicated by the two tokens in the offer place.

Upon receipt by the respective worklist handler, these requests are recorded in the work queue for the relevant user with a status of *offered*. A user can indicate their intention to execute the work item at some future time by initiating the `select work item for allocation` transition.

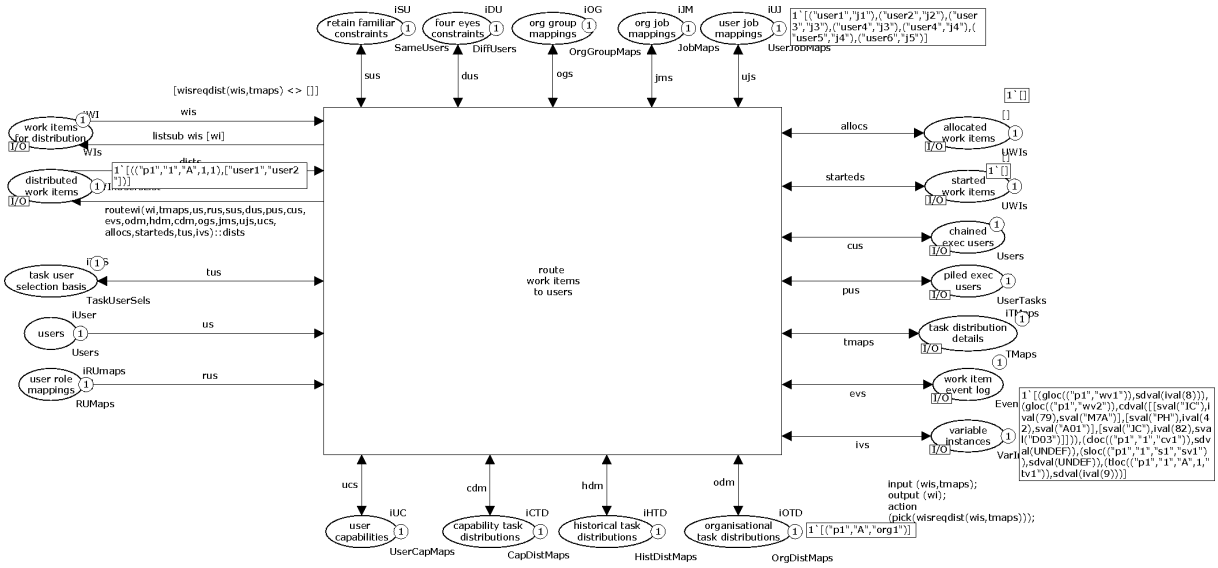


Figure 87: Working model – routing based using an organizational distribution function

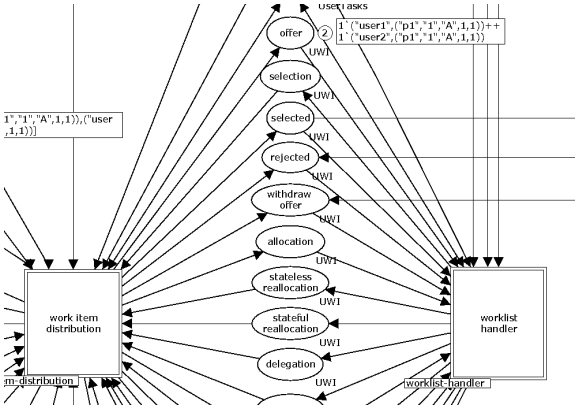


Figure 88: Working model – distribution of requests between work item distribution and work list handler processes

6.4 Rerouting of work items

Although the distribution and execution of work items is generally fairly predictable in that they are offered or assigned to users who at some future time execute and complete them, there are occasions where this sequence of events needs to be varied and the ultimate user who will execute them needs to be varied. The variation can be initiated by a user, the process engine or by a system administrator and various permutations of reassignments are possible. In this section we consider the user-initiated delegation of a work item allocated to them that they do not wish to undertake. The effects of *user1* initiating the delegation of work item A to *user6* are illustrated in Figure 89 with the delegation request flowing from the worklist handler to the work item distribution process. This is subsequently confirmed by the work item distribution process (as shown in Figure 90) and the work item is reassigned to *user6* with an *allocated* status as illustrated in Figure 91.

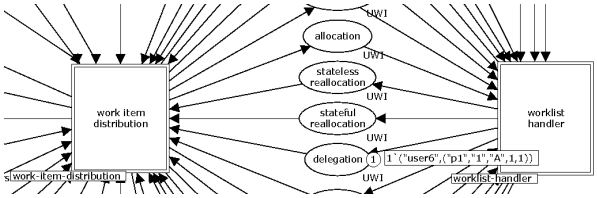


Figure 89: Working model – delegation request from work list handler to work item distribution process

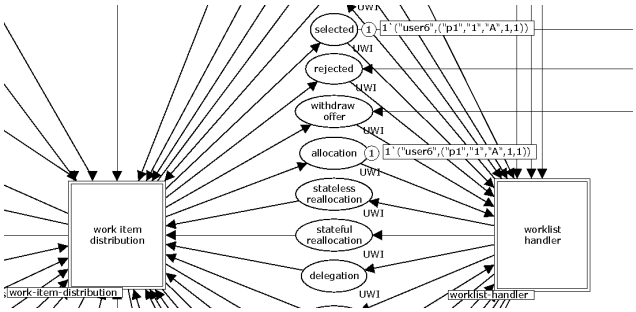


Figure 90: Working model – delegation confirmation from work item distribution process to work list handler

Once the work item corresponding to task A is complete, two subsequent tasks can potentially be enabled as task A is associated with an AND-split. Both of these have interesting characteristics: task B is a multiple instance task and task C is composite in form. We discuss each of them below.

6.5 Multiple instance tasks

The precursor to a multiple instance task is that the complex data element referred to by the multiple invar parameter exists and does indeed hold data in the required tabular form indicated by the parameter. Figure 92 illustrates this situation, the incoming parameter to task B will pass data to three multiple instance task variables from global variable *wv2* and it holds data of the required form. As there are no other preconditions to the task, it is enabled.

Once the `enter` transition has fired for the task, the number of task instances created is determined from the number of lines resulting from the evaluation of the incoming multiple instance parameter. In this case, there are three lines, resulting in three work items for subsequent assignment and nine multiple instance data elements are created as there are three data values for each instance. Figure 93 illustrates the firing of the transition for task B.

6.6 Composite tasks

A composite activity has a subprocess (i.e. another net) associated with it. The enablement of a composite task occurs in the same way as that for a normal atomic tasks. It can have preconditions and parameters associated with it, however in this case the parameters map data elements to the subprocess that implements the task rather than to the task itself. Figure 94 illustrates this process. A key part of it is

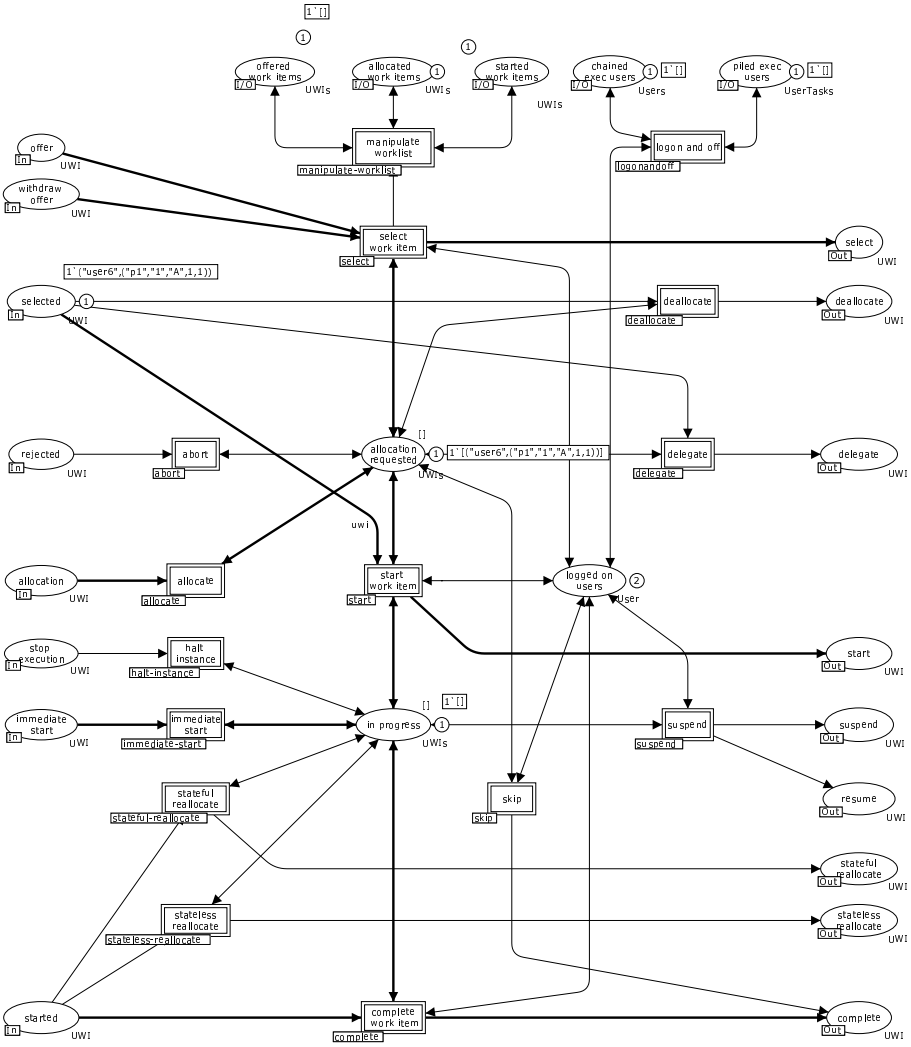


Figure 91: Working model – delegation handing in work list handler

the determination of the subprocess CID which is used to differentiate a subprocess instance from an execution thread in the net from which it was created. In this case, the subprocess CID is 1.1. Any variables created for the subprocess are created with this CID as illustrated by the block variable *bv1* which receives its value from folder variable *fv1*.

The act of starting a composite task corresponds to the placement of token with the subprocess *CID* in the input place for the net corresponding to the subprocess decomposition. This is illustrated in Figure 95.

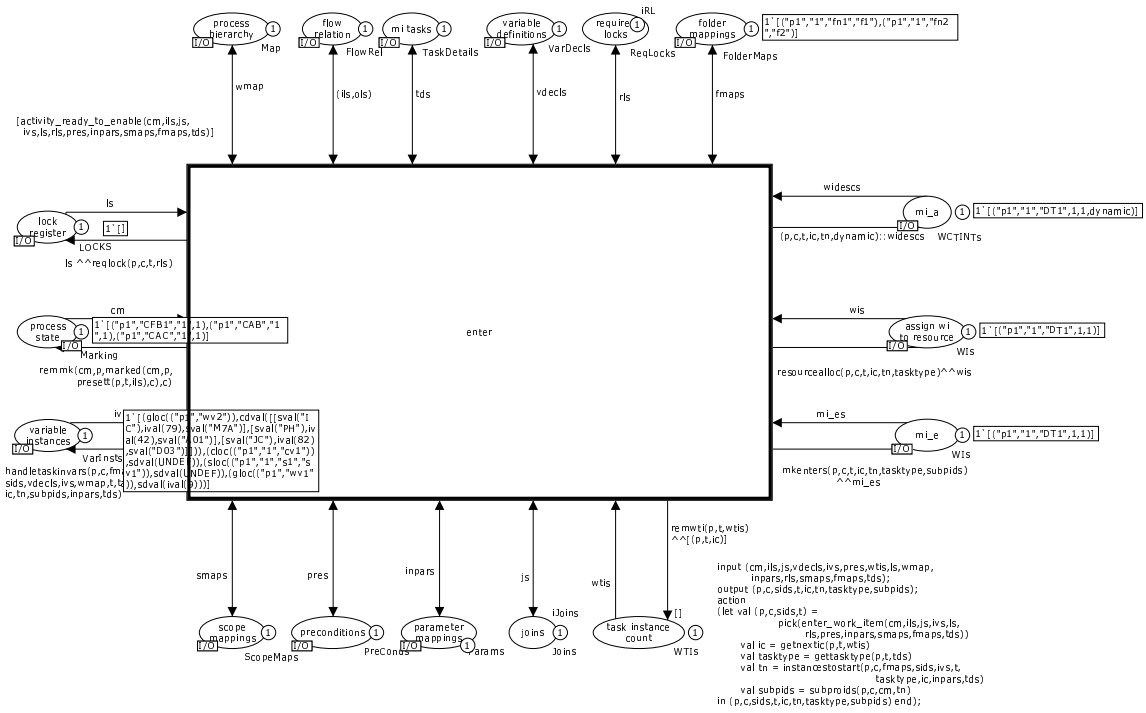


Figure 92: Working model – precursor to enablement of a multiple instance task

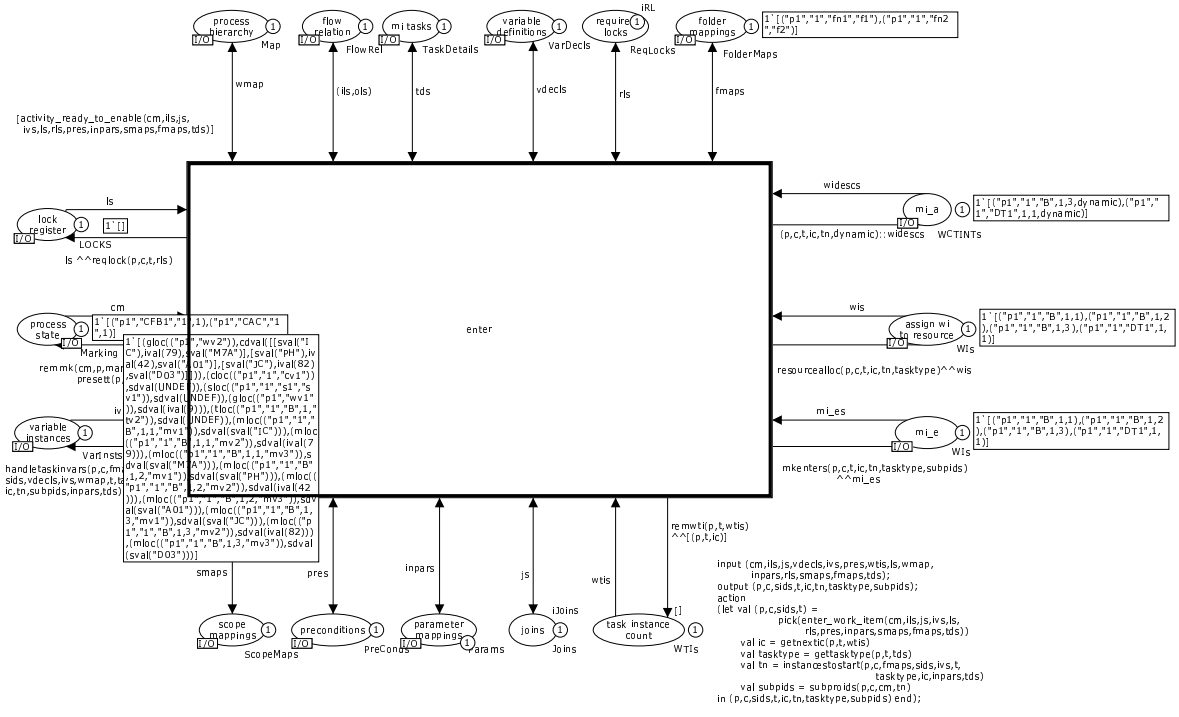


Figure 93: Working model – initiation of a multiple instance task

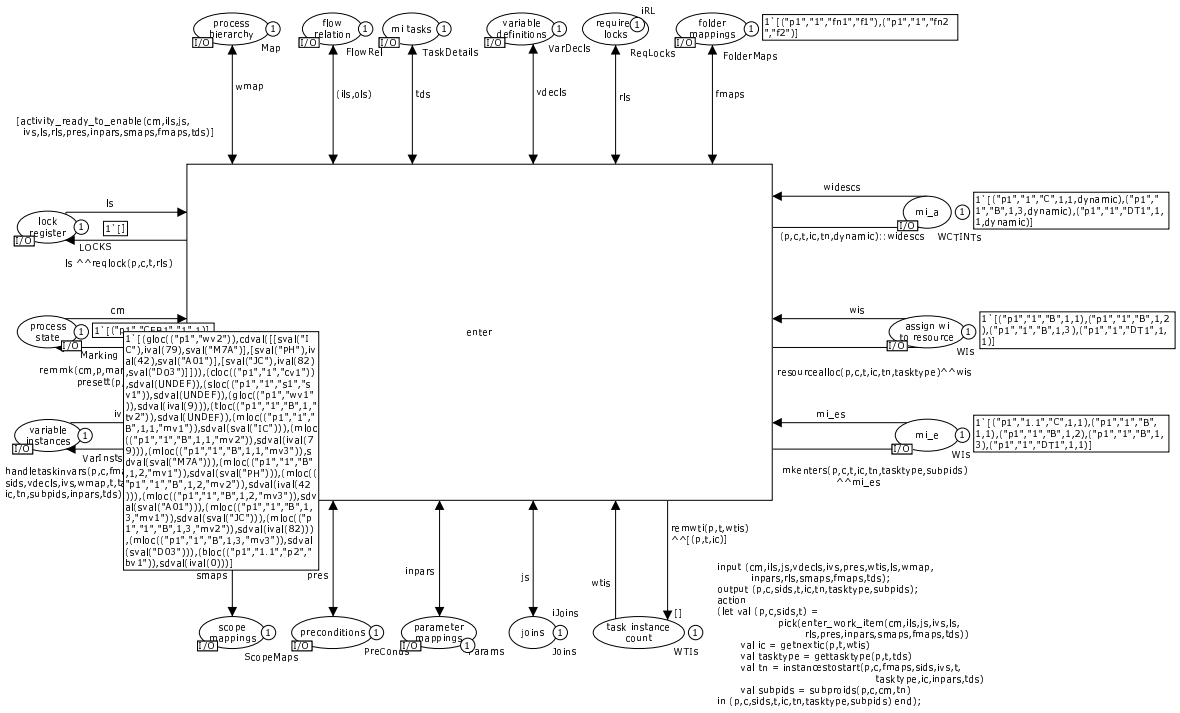


Figure 94: Working model – initiation of a composite task

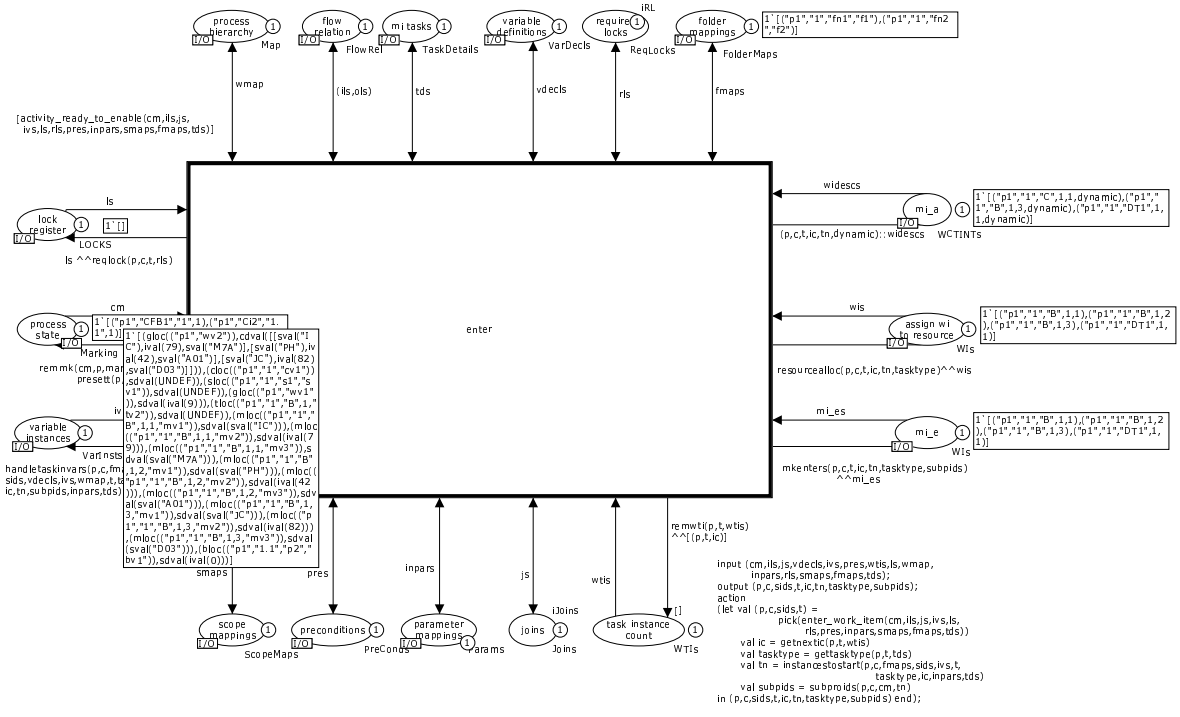


Figure 95: Working model – subprocess initiation

7 Pattern Support in *new*YAWL

As a benchmark of the capabilities of *new*YAWL, this section presents the results of a patterns-based evaluation of the language using the control-flow, data and resource patterns. For comparative purposes, this evaluation is contrasted with one for YAWL. Although there is a formal definition for YAWL, it is limited to the control-flow perspective. For this reason, the evaluation results in this section are based on the latest version of the YAWL System (Beta 8). The *new*YAWL results are based on the capabilities demonstrated by the abstract syntax and the semantic models presented earlier. Subsequent sections in this section present the results of pattern evaluations in three perspectives: control-flow, data and resource.

7.1 Control-flow perspective

Table 13 identifies the extent of support by YAWL and *new*YAWL for the control-flow patterns. YAWL supports 19 of the 20 original control-flow patterns. The only

Nr	Pattern	YAWL	*new*YAWL	Nr	Pattern	YAWL	*new*YAWL
	Basic Control				New Control-Flow Patterns		
1	Sequence	+	+	21	Structured Loop	–	+
2	Parallel Split	+	+	22	Recursion	–	+
3	Synchronization	+	+	23	Transient Trigger	–	+
4	Exclusive Choice	+	+	23	Persistent Trigger	–	+
5	Simple Merge	+	+	25	Cancel Region	+	+
	Adv. Branching & Synchronization			26	Cancel MI Activity	+	+
6	Multiple Choice	+	+	27	Complete MI Activity	–	+
7	Structured Synchronising Merge	+	+	28	Blocking Discriminator	–	+
8	Multiple Merge	+	+	29	Cancelling Discriminator	+	+
9	Structured Discriminator	+	+	30	Structured Partial Join	–	+
	Structural			31	Blocking Partial Join	–	+
10	Arbitrary Cycles	+	+	32	Cancelling Partial Join	–	+
11	Implicit Termination	+/–	–	33	Generalized AND-Join	+	+
	Multiple Instance			34	Static Partial Join for MIs	+	+
12	MI without Synchronization	+	+	35	Static Canc. Partial Join for MIs	+	+
13	MI with a priori Design Time Knowl.	+	+	36	Dynamic Partial Join for MIs	–	+
14	MI with a priori Runtime Knowl.	+	+	37	Acyclic Synchronizing Merge	+	+
15	MI without a priori Runtime Knowl.	+	+	38	General Synchronizing Merge	+	+
	State-based			39	Critical Section	+	+
16	Deferred Choice	+	+	40	Interleaved Routing	+	+
17	Interleaved Parallel Routing	+	+	41	Thread Merge	–	+
18	Milestone	+	+	42	Thread Split	–	+
	Cancellation			43	Explicit Termination	–	+
19	Cancel Activity	+	+				
20	Cancel Case	+	+				

Table 13: Support for control-flow patterns in Original YAWL vs *new*YAWL

omission being *Implicit Termination* which, although being the preferred method of process termination, is not fully implemented in the YAWL system. It fares less well in terms of support for the new control-flow patterns. The availability of the cancellation region construct allows YAWL to support the *Cancel Region*, *Cancel MI Task* and *Cancelling Discriminator* patterns. Similarly, the availability of a multiple instance

task construct ensures the *Static Partial Join for MIs* and the *Static Cancelling Join for MIs* patterns are supported. Considerable work [WEAH05] has gone into OR-join handling in YAWL, hence the *Acyclic* and *General Synchronizing Merge* are supported. Finally the Petri net foundation for YAWL provide a means of supporting the *Critical Section* and *Interleaved Routing* patterns. None of the other new control-flow patterns are supported.

In contrast, *new*YAWL supports 42 of the 43 patterns. The only omission is the *Implicit Termination* pattern which is not supported as the termination semantics of *new*YAWL are based on the identification of a single defined end node for processes. When the thread of control reaches the end node in a *new*YAWL process instance, it is deemed to be complete and no further work is possible. Thus it achieves a full support rating for the *Explicit Termination* pattern which is a distinction from previous versions of the YAWL System which were based on *Implicit Termination*.

7.2 Data perspective

Table 14 illustrates data pattern support in YAWL and *new*YAWL. The data model in YAWL is based on the use of net variables which are passed to and from tasks using XQuery statements. These can be used for data passing to all forms of task hence the *Block Data* and *Multiple Instance Data* patterns are directly supported together with the various *Data Interaction* patterns relevant to these constructs. The use of XQuery for data passing ensures the *Data Transfer by Value – Incoming* and *Outgoing* patterns are also supported together with the corresponding *Data Transformation* patterns. There is also support for *Data-based Routing.*

Nr	Pattern	YAWL	*new*YAWL	Nr	Pattern	YAWL	*new*YAWL
	Data Visibility				Data Interaction (Ext.) (cont.)		
1	Task Data	–	+	21	Env. to Case – Push-Oriented	–	+
2	Block Data	+	+	22	Case to Env. – Pull-Oriented	–	+
3	Scope Data	–	+	23	Workflow to Env. – Push-Orient.	–	+
4	Multiple Instance Data	+	+	24	Env. to Workflow – Pull-Orient.	–	+
5	Case Data	–	+	25	Env. to Workflow – Push-Orient.	–	+
6	Folder Data	–	+	26	Workflow to Env. – Pull-Orient.	–	+
7	Workflow Data	–	+		Data Transfer		
8	Environment Data	–	+	27	by Value Incoming	+	+
	Data Interaction (Internal)			28	by Value Outgoing	+	+
9	between Tasks	+	+	29	Copy In/Copy Out	–	+
10	Block Task to Sub-work. Decomp.	+	+	30	by Reference – Unlocked	–	–
11	Sub-work. Decomp. to Block Task	+	+	31	by Reference – Locked	–	+/–
12	to Multiple Instance Task	+	+	32	Data Transformation – Input	+	+
13	from Multiple Instance Task	+	+	33	Data Transformation – Output	+	+
14	Case to Case	–	+		Data-based Routing		+
	Data Interaction (External)			34	Task Precondition – Data Exist.	–	+
15	Task to Env. – Push-Oriented	–	+	35	Task Precondition – Data Val.	–	+
16	Env. to Task – Pull-Oriented	–	+	36	Task Postcondition – Data Exist.	–	+
17	Env. to Task – Push-Oriented	–	+	37	Task Postcondition – Data Val.	–	+
18	Task to Env. – Pull-Oriented	–	+	38	Event-based Task Trigger	–	+
19	Case to Env. – Push-Oriented	–	+	39	Data-based Task Trigger	–	+
20	Env. to Case – Pull-Oriented	–	+	40	Data-based Routing	+	+

Table 14: Support for data patterns in Original YAWL vs *new*YAWL

*new*YAWL markedly improves on this range of capabilities and supports all but two of the patterns. One of the two remaining patterns – *Data Transfer by Reference – Locked* – receives a partial support rating. This is a consequence of the value-based interaction strategy that *new*YAWL employs for data passing. *new*YAWL does provide locking facilities for data elements, hence it is possible to prevent concurrent use of a nominated data element thus achieving the same operational effect as required for support of this pattern however as it is not directly implemented in this form, it only achieves a partial support rating.

7.3 Resource perspective

Table 15 illustrates the extent of resource pattern support by the two offerings. YAWL provides relatively minimal consideration of this perspective supporting only 8 of the 43 patterns.

Nr	Pattern	YAWL	*new*YAWL	Nr	Pattern	YAWL	*new*YAWL
	Creation Patterns				Pull Patterns (cont.)		
1	Direct Allocation	+	+	24	System-Determ. Wk Queue Cont.	–	+
2	Role-Based Allocation	+	+	25	Resource-Determ. Wk Queue Cont.	–	+
3	Deferred Allocation	–	+	26	Selection Autonomy	–	+
4	Authorization	–	+		Detour Patterns		
5	Separation of Duties	–	+	27	Delegation	–	+
6	Case Handling	–	–	28	Escalation	–	+
7	Retain Familiar	–	+	29	Deallocation	–	+
8	Capability-Based Allocation	–	+	30	Stateful Reallocation	–	+
9	History-Based Allocation	–	+	31	Stateless Reallocation	–	+
10	Organizational Allocation	–	+	32	Suspension/Resumption	–	+
11	Automatic Execution	+	+	33	Skip	–	+
	Push Patterns			34	Redo	–	–
12	Distrib. by Offer - Single Resource	+	+	35	Pre-Do	–	–
13	Distrib. by Offer - Multiple Resources	+	+		Auto-Start Patterns		
14	Distrib. by Allocation - Single Resource	–	+	36	Commencement on Creation	–	+
15	Random Allocation	–	+	37	Creation on Allocation	–	+
16	Round Robin Allocation	–	+	38	Piled Execution	–	+
17	Shortest Queue	–	+	39	Chained Execution	–	+
18	Early Distribution	–	–		Visibility Patterns		
19	Distribution on Enablement	+	+	40	Conf. Unalloc. Work Item Visibility	–	+
20	Late Distribution	–	+	41	Conf. Alloc. Work Item Visibility	–	+
	Pull Patterns				Multiple Resource Patterns		
21	Resource-Init. Allocation	+	+	42	Simultaneous Execution	+	+
22	Resource-Init. Exec. - Alloc. Wk Items	+	+	43	Additional Resource	–	–
23	Resource-Init. Exec. - Offer. Wk Items	–	+				

Table 15: Support for resource patterns in Original YAWL vs *new*YAWL

In contrast, *new*YAWL supports 38 of the 43 resource patterns. The five that are not supported are:

– *Case Handling* – as this implies that complete process instances are allocated to users rather than individual work items as is the case in the majority of current PAIS;
– *Early Distribution* – as work items in *new*YAWL can only be allocated to users once they have been enabled;

– *Pre-Do* – as work items can only be executed at the time they are enabled and cannot be allocated to a resource prior to this time;
– *Redo* – as work items cannot be re-allocated to a user at some time after their execution has been completed; and
– *Additional Resource* – as work items are only ever undertaken by a *single* resource in *new*YAWL and there is no provision for the additional involvement of non-human resources.

8 Epilogue

The main objectives of this research initiative were to provide a definitive semantic meaning for each of the catalogue of workflow patterns in the control-flow, data and resource perspectives and to demonstrate that they could be implemented on a common platform in an integrated manner. The workflow patterns embody individual recurring constructs that have generic applicability when modelling and enacting process-aware information systems. They are derived from empirical observations of actual process modelling formalisms and execution environments hence it is to be expected that they are able to be formally defined and enacted as this research initiative has illustrated. In addition to successfully achieving these goals, we have also shown how these patterns can be enacted in an integrated operational environment. This work forms the basis of the design for the *new*YAWL offering, a reference language for process-aware information systems. An implementation of which is currently being developed by the BPM Group at QUT.

One of the main propositions underlying the approach taken to completing this research has been that the design and development of large-scale software initiatives such as *new*YAWL should be based on formal specification of the intended functionality prior to the actual software development. This notion is not novel and whilst widely lauded in various sections of the software community, it has been extremely difficult to achieve. The motivations for pursuing a formal approach to software specification lie in the belief that the underlying design will be both correct and consistent. Moreover the use of formal techniques should allow for the experimentation with a wider range of design alternatives and the selection of the most appropriate of them at an earlier stage in the overall software development process.

We have been able to successfully produce a complete formal design for the *new*YAWL offering based on formal principles. However, whilst we agree with the need for the use of formal techniques in software development, there are two immediate observations that are pertinent to this work: (1) in order to prevent the need to develop a design completely from first principles, the approach used for specifying the design must be based on a formal underlying model that itself is both self-consistent and capable of embodying various levels of abstraction and (2) there must be an environment available in which the model can developed, tested for consistency and executed. Unless it is possible to satisfy both of these requirements, it is not possible to produce a complete design of a complex software offering such as *new*YAWL in a tractable and reliable manner.

For our purposes we selected the *Coloured Petri-Net* modelling formalism and used the *CPN Tools* offering to develop the formal semantic model for *new*YAWL. This decision has suited our purposes admirably. We have been able to build a

comprehensive model of *new*YAWL in a relatively efficient manner. Indeed the overall design effort has been less than 10% of the total development budget for the initial version of the YAWL System, which has a significantly smaller range of functionality. CPN Tools provides an interactive development environment for modelling that allows candidate models to be executed. This provides immediate feedback on the efficacy and relative merit of potential solutions to a specific problem and allows specific design alternatives to be tested and decided on at a relatively early stage in the software process with some certainty of the actual impact of these choices from an execution standpoint.

The hierarchical nature of the CPN model has proven to be extremely effective in delineating the various structural components of *new*YAWL and the interfaces between them. Moreover, it allows different processing paradigms to be embodied within the same model. For example: the control-flow and data perspectives in *new*YAWL are tightly interrelated and involve a limited number of interactions between a large range of data resources where as the resource perspective involves a relatively broad range of possible interactions between a smaller range of data resources. In visual terms, the former perspectives are embodied by CPN models involving a wide range of persistent data elements linked to a small number of transitions whilst the latter leads to process models which have a much larger number of transitions (and possible execution paths linking those transitions) but involve a more limited range of data elements.

An additional benefit of the hierarchical modelling approach is the ability to clearly delineate common functionality and the interfaces between various structural components. As an example, the control-flow and data perspectives communicate with the resource perspective through four specific interaction points. Similarly, the interface between the *work item distribution*, *work list handler* and *management intervention* processes that comprise the resource perspective are also clearly delineated and the specific range of interactions that are allowed between these components can be defined accordingly.

The overall modelling process for *new*YAWL, whilst extremely complex, has been relatively time-efficient. Much of this progress is attributable to the design decisions taken early in the project and the capabilities of the modelling tools chosen. Whilst extremely powerful, there are several aspects of the CPN Tools environment that would benefit from the inclusion of additional capabilities. In particular, the ability to incrementally wind back the execution state of a given execution would be useful, as would the ability to save an execution state for later execution. The interaction facilities for the CPN model are particularly effective, however there are less features provided for tracing and altering the execution of ML code segments that form part of a CPN model. The inclusion of features such as these in future *CPN Tools* releases would be extremely beneficial.

A Mathematical Notations

This appendix outlines mathematical notations used in this report that are not in general use and hence merit some further explanation.

In the context of a *new*YAWL net, where $t \in T$ is a task, $\bullet t$ denotes the input conditions or tasks (as in *new*YAWL, tasks can be directly linked to tasks) to the task and $t\bullet$ denotes the output conditions or tasks. In a more formal sense, $\bullet t = \{x \in T \cup C \,|\, (x,t) \in F\}$ where T is the set of tasks, C the set of conditions and F the flow relation (i.e. the set of arcs) associated with a net. Similarly, $t\bullet = \{x \in T \cup C \,|\, (t,x) \in F\}$.

In the context of a function $f\colon A \to B$, range restriction of f over a set $R \subseteq B$ is defined by $f \rhd R = \{(a,b) \in f \mid b \in R\}$.

$\mathbb{P}(X)$ denotes the power set of X where $Y \in \mathbb{P}(X) \Leftrightarrow Y \subseteq X$.

$\mathbb{P}^+(X)$ denotes the power set of X without the empty set ie. $\mathbb{P}^+(X) = \mathbb{P}(X) \backslash \{\emptyset\}$.

Let $V = \{v_1, ...v_n\}$ be a non-empty) set and $<$ a strict total order over V, then $[V]^<$ denotes the sequence $[v_1, ...v_n]$ such that $\forall_{1 \leq i \leq j < n}[v_i < v_j]$ and every element of V occurs precisely once in the sequence. $[v]$ denotes the sequence in arbitrary order. Sequence comprehension is defined as $[E(x) \mid x \leftarrow [V]^<]$ yielding a sequence $[E(v_1)...E(v_n)]$.

B *new*YAWL Specification for Working Example

```
newYAWL-net = p1
NetID = {p1,p2}
FolderID = {fn1,fn2}
ProcessID = {p1}
TaskID = {A,B,C,D,F,F,G,J,K,L}
ScopeID = {s1}
VarID = {v1,v2,v3,v4,v5,v6,v7,v8,v9,v10,v11,v12,v13,v14,v15}
TriggerID = {t1}
TNmap = {((p1,C),p2)}
NTmap = {(p1,{A,B,C,D,F,F,G}),(p2,{J,K,L})}
STmap = {(s1,{B,D,E})}
VarNames = {wv1,wv2,fv1,fv2,cv1,bv1,sv1,tv1,tv2,tv3,tv4,tv5,mv1,mv2,mv3}
DataTypes = {string,int,bool,emprecs}
VName = {(v1,wv1),(v2,wv2),(v3,fv1),(v4,fv2),(v5,cv1),(v6,bv1),(v7,sv1),
(v8,tv1),(v9,tv2),(v10,tv3),(v11,tv4),(v12,tv5),(v13,mv1),(v14,mv2),
(v15,mv3)}
DType = {(v1,int),(v2,emprecs),(v3,string),(v4,int),(v5,string),(v6,string),
(v7,string),(v8,int),(v9,string),(v10,string),(v11,int),(v12,string),
(v13,string),(v14,int),(v15,string)}
VarType = {(v1,Global),(v2,Global),(v3,Folder),(v4,Folder),(v5,Case),
(v6,Block),(v7,Scope),(v8,Task),(v9,Task),(v10,Task),(v11,Task),(v12,Task),
(v13,MI),(v14,MI),(v15,MI)}
VGmap = {(v1,p1),(v2,p2)}
VFmap = {(v3,fn1),(v4,fn2)}
VCmap = {(v5,p1)}
VBmap = {(v6,p2)}
VSmap = {(v7,s1)}
VTmap = {(v8,A),(v9,B),(v10,E),(v11,F),(v12,J)}
VMmap = {(v13,B),(v14,B),(v15,B)}
PushAllowed = {v1,v2,v3,v4,v5,v6,v7,v8,v9,v10,v11,v12}
PullAllowed = {v1,v2,v3,v4,v5,v6,v7,v8,v9,v10,v11,v12}



nid = p1
C = {Ci,Co}
i = Ci
o = Co
T = {A,B,C,D,E,F,G}
F = {(Ci,A),(A,B),(A,C),(B,D),(B,E),(D,E),(E,F),(C,F),(F,G),(G,Co)}
Split = {(A,AND),(B,XOR)}
join = {(E,XOR),(F,PJOIN)}
Default = {(B,D)}
<XOR = {(B,{(1,D),(2,E)}])}
Rem = {}
Comp = {(F,{B})}
Block = {}
Disable{(C,{B})}
Nofi = {(B,1,5,2,dynamic,cancelling)}
Lock = {(A,v2)}
Thresh = {(F,1)}
ThreadIn = {}
```

```
ThreadOut = {}
ArcCond = {((B,D),undef(v9)),((B,E),isdef(v9))}
Pre = {(A,isdef(v1))}
Post = {(G,isdef(v4))}
WPre = isdef(v2)
WPost = true
PreTest = {}
PostTest = {}


nid = p2
C = {Ci2,Co2}
i = Ci2
o = Co2
T = {J,K,L}
F = {(Ci2,J),(J,K),(K,L),(L,Co2)}
Split = {(J,THREAD)}
Join = {(L,THREAD)}
Default = {}
<XOR = {}
Rem = {}
Comp = {}
Block = {}
Disable{}
Nofi = {}
Lock = {}
Thresh = {}
ThreadIn = {(L,3)}
ThreadOut = {(J,3)}
ArcCond = {}
Pre = {}
Post = {}
WPre = true
WPost = true
PreTest = {(J,less3(v12))}
PostTest = {}



InPar = {((A,v8),inc(v1)),((E,v10),copy(v7),((F,v11),copy(v5))),((J,v12),copy(v6))}
OutPar = {((A,v1),copy(v8)),((E,v7),addc(v10)),((F,v5),copy(v11)),((J,v6),copy(v12))}
OptInPar = {(F,v5),(J,v6)}
OptOutPar = {(J,v12)}
InNet = {((C,v6),copy(v3))}
OutNet = {((C,v3),copy(v6))}
OptInPar = {}
OptOutPar = {}
MIInPar = {((B,{v13,v14,v15}),copy(v2))}
MIOutPar = {(B,v2),copy(v13,v14,v15))}
InProc = {}
OutProc = {(v1,inc(v4))}
```

```
UserID = {user1,user2,user3,user4,user5,user6}
RoleID = {role1,role2,role3}
CapabilityID = {cap1,cap2,cap3}
OrgGroupID = {og1,og2,og3}
JobID = {j1,j2,j3,j4,j5}
CapVal = {val1,val2,val3,val4,val5}
GroupType = {(og1,team),(og2,team),(og3,department)}
JobGroup = {(j1,og1),(j2,og1),(j3,og2),(j4,og3),(j5,og3)}
OrgStruct = {(og1,og2),(og2,og3)}
Superior = {(j1,j2),(j2,j4),(j3,j4),(j4,j5)}
RoleUser = {(role1,{user1,user2,user3,user4}),(role2,{user2,user4,user6}),
(role3,{user5,user6})}
UserPriv = {(user1,{choose}), (user2,{choose,concurrent,reorder}),
(user3,{choose,viewoffers,viewallocs,viewexecs}), (user4,{choose}),
(user5,{choose,concurrent,reorder,viewoffers,viewallocs,viewexecs,chainedexec}),
(user6,{choose,concurrent,reorder,viewoffers,viewallocs,viewexecs,chainedexec})}
UserTaskPriv = {((user1,A),{suspend,reallocate,reallocate\_state,deallocate,
delegate,skip,piledexec}),
((user2,A),{piledexec}),((user3,J),{piledexec}),((user3,K),{piledexec}),
((user3,L),{piledexec}),
((user6,A),{suspend,reallocate,reallocate\_state,deallocate,delegate,skip,piledexec}),
((user3,B),{suspend,reallocate,reallocate\_state,deallocate,delegate,skip,piledexec})}
UserQual = {((user1,cap1),val1),((user1,cap2),val2),((user2,cap1),val3),
((user3,cap2),val4),((user4,cap1),val1),((user4,cap2),val2),((user5,cap1),val1),
((user5,cap2),val3),((user6,cap1),val1),((user6,cap2),val2)}
UserJob = {(user1,{j1}),(user2,{j2}),(user3,{j3}),(user4,{j3,j4}),(user5,{j4}),
(user6,{j5})}
HistExpr = {lastA}
OrgExpr = {org1}
CapExpr = {capval1}
Auto = {G}
Manual = {A,B,D,E,F,G,J,K,L}
Initiator = {(A,(system,system,system)),(B,(system,resource,resource)),
(D,(resource,system,resource)),(E,(system,system,resource)),
(F,(resource,resource,resource)),(G,(system,system,system)),
(J,(system,resource,system)),(K,(system,resource,resource)),
(L,(system,resource,resource)),}
DistUser = {(A,{user1,user2,user3,user4,user5,user6}),(D,{user1,user2,user3,user4}),
(F,{user1}),(J,{user1,user2,user3,user4}),(K,{user1,user2,user3,user4})}
DistRole = {(B,{role1,role3}),(E,{role1,role2,role3}),(L,{role2,role3})}
DistVar = {}
SameUser = {(K,J)}
FourEyes = {(L,K)}
HistDist = {(B,lastA())}
OrgDist = {(A,org1())}
CapDist = {(J,capval1())}
UserSel = {(E,sortestqueue),(B,random)}
```

C Unfolded *new*YAWL Specification for Working Example

```
TaskID = {A,B,C,D,E,F,G,J,K,L,TSP1,TSP2,TEP1,TEP2,DT1,DT3,FJ1,FJ2,FR1,FR2,JL1,JL2,
JL3,JL4,JL3S1,JL3S2,JL3S3,LM1,LM2}
TriggerID = {}
STmap = {(s1,{B,D,DT3,E})}

nid = p1
C = {Cstart,Cstart2,Ci,CT,CBDT3,DD,CAB,CAC,CBD,CBE,CEFJ1,CFB1,CCFJ2,CFJ1FR1,CFJ2FR2,
CFJ2F,CFJ1F,CFG,Co,Cend,Cend2}
i = Cstart2
o = Cend2
T = {A,B,C,D,E,F,G,J,K,L,TSP1,TEP1,TSP2,TEP2,DT1,DT3,FJ1,FJ2,FR1,FR2}
F = {(Cstart2,TSP2),(TSP2,Cstart),(TSP2,CFB1),(Cstart,TSP1),(TSP1,CT),(TSP1,Ci),
(CT,DT1),(DT1,CT),(CT,TEP1),(TEP1,Cend),(Cend,TEP2),(CFB1,TEP2),(TEP2,Cend2),
(Ci,A),(A,CAB),(CAB,B),(B,CBDT3),(CBDT3,DT3),(DT3,DD),(B,CBE),(DD,D),
(D,CDE),(CDE,E),(CBE,E),(E,CEFJ1),(A,CAC),(CAC,C),(C,CCFJ2),(CCFJ2,FR1),
(CCFJ2,FJ2),(FJ2,CFJ2F),(FJ2,CFJ2FR2),(CFJ2FR2,FR2),(CEFJ1,FJ1),(FJ1,CFJ1F),
(CEFJ1,FR2),(FJ1,CFJ1FR1),(CFJ1FR1,FR1),(CFB1,FJ1),(CFB1,FJ2),(FR1,CFB1),
(FR2,CFB1),(CFJ1F,F),(CFJ2F,F),(F,CFG),(CFG,G),(G,Co),(Co,TEP1)}
split = {(A,AND),(B,XOR),(DT1,AND),(FJ1,AND),(FJ2,AND),(TSP1,AND),(TSP2,AND)}
join = {(E,XOR),(F,XOR),(FJ1,AND),(FJ2,AND),(FR1,AND),(FR2,AND),(TEP1,AND),
(TEP2,AND)}
<XOR = {(B,{(1,CBDT3),(2,CBE)}])}
ArcCond = {((B,CBDT3),undef(v9)),((B,CBE),def(v9))}
Thresh = {}

nid = p2
C={Ci2,CJL2JL1,CJL1J,CJJL4,CJL1JL3,CJL4JL1,CJL4JL3,CJL3S1,CJL3S2,CJL3S3,CJL3S,
CM1,CM2,Co2}
T = {J,JL1,JL2,JL3,JL4,JL3S1,JL3S2,JL3S3,K,L,LM1,LM2}
F = {(Ci2,JL2),(JL2,CJL2JL1),(CJL2JL1,JL1),(JL1,CJL1J),(JL1,CJL1JL3),(CJL1J,J),
(J,CJJL4),(CJJL4,JL4),(JL4,CJL4JL1),(CJL4JL1,JL1),(JL4,CJL4JL3),(CJL4JL3,JL3),
(CJL1JL3,JL3),(JL3,CJL3S1),(JL3,CJL3S2),(JL3,CJL3S3),(CJL3S1,JL3S1),
(CJL3S2,JL3S2),(CJL3S3,JL3S3),(JL3S1,CJL3S),(JL3S2,CJL3S),(JL3S3,CJL3S),(CJL3S,K),
(K,CM1),(CM1,L),(CM1,LM1),(LM1,CM2),(CM2,LM2),(LM2,CM1),(CM2,L),(L,Co2)}
split = {(JL1,XOR),(JL4,XOR),(JL3,AND)}
join = {(JL1,XOR),(JL3,XOR),(L,AND)}
<XOR = {(JL1,{(1,CJL1J),(2,CJL1JL3)}),(JL4,{(1,CJL4JL1),(2,CJL4JL3)})}
ThreadIn = {}
ThreadOut = {}
ArcCond = {((JL1,CJL1J),less3(v12)),((JL1,CJL1JL3),nless3(v12)),
((JL4,CJL4JL1),true),((JL4,CJL4JL3),false)}
PreTest = {}

InPar = {((A,v8),inc(v1)),((E,v10),copy(v7),((F,v11),copy(v5))),((FJ1,v11),copy(v5))),
((FJ2,v11),copy(v5))),((J,v12),copy(v6)),((JL1,v12),copy(v6)),((JL3,v12),copy(v6))}
OptInPar = {(F,v5),(FJ1,v5),(FJ2,v5),(J,v6),(JL1,v6),(JL3,v6)}

Auto = {G,TSP1,TSP2,TEP1,TEP2,DT1,DT3,FJ1,FJ2,FR1,FR2,JL1,JL2,JL3,JL4,JL3S1,JL3S2,JL3S3,
LM1,LM2}
```

D Initial Marking for Working Example

```
val iTMaps = [("p1","A",  users ["user1","user2","user3","user4","user5","user6"],
system,resource,resource),
("p1","B", roles ["role1","role3"], system,resource,resource),
("p1","D", users ["user1","user2","user3","user4"], resource,system,resource),
("p1","E", roles ["role1","role2","role3"], system,system,resource),
("p1","F", users ["user1"], resource,resource,resource),
("p1","G", AUTO, system,system,system),
("p1","J", users ["user1","user2","user3","user4"], system,resource,system),
("p1","K", users ["user1","user2","user3","user4"], system,resource,resource),
("p1","L", roles ["role2","role3"], system,resource,system),
("p1","TSP1", AUTO, system,system,system),
("p1","TSP2", AUTO, system,system,system),
("p1","TEP1", AUTO, system,system,system),
("p1","TEP2", AUTO, system,system,system),
("p1","DT1", AUTO, system,system,system),
("p1","FJ1", AUTO, system,system,system),
("p1","FJ2", AUTO, system,system,system),
("p1","FR1", AUTO, system,system,system),
("p1","FR2", AUTO, system,system,system),
("p1","JL1", AUTO, system,system,system),
("p1","JL2", AUTO, system,system,system),
("p1","JL3", AUTO, system,system,system),
("p1","JL4", AUTO, system,system,system),
("p1","JL3S1", AUTO, system,system,system),
("p1","JL3S2", AUTO, system,system,system),
("p1","JL3S3", AUTO, system,system,system),
("p1","LM1", AUTO, system,system,system),
("p1","LM2", AUTO, system,system,system)];

val iTasks = ["A","B","C","D","E","F","G","J","K","L","DT1","DT3","TSP1","TSP2",
"TEP1","TEP2","FJ1","FJ2","FR1","FR2","JL1","JL2","JL3","JL4","JL3S1","JL3S2",
"JL3S3","LM1","LM2"];

val iRUmaps = [("role1",{"user1","user2","user3","user4"}),
("role2",{"user2","user4","user6"}),
("role3",["user5","user6"])];

val iUser = ["user1","user2","user3","user4","user5","user6"];

val iUP = [("user1",[choose]), ("user2",[choose,concurrent,reorder]),
("user3",[choose,viewoffers,viewallocs,viewexecs]), ("user4",[choose]),
("user5",[choose,concurrent,reorder,viewoffers,viewallocs,viewexecs,chainedexec]),
("user6",[choose,concurrent,reorder,viewoffers,viewallocs,viewexecs,chainedexec])]

val iUTP = [("user1","A",[suspend,reallocate,reallocate_state,deallocate,
delegate,skip,piledexec]),
("user2","A",[piledexec]),("user3","J",[piledexec]),("user3","K",[piledexec]),
("user3","L",[piledexec]),
("user6","A",[suspend,reallocate,reallocate_state,deallocate,delegate,skip,
piledexec]),("user3","B",[suspend,reallocate,reallocate_state,deallocate,delegate,
skip,piledexec])];
```

```
val iTUS = [("p1","E",shortestqueue),("p1","B",random)];

val iRL = [("p1","A",[gdef(("p1","wv2"))])];

val iSplits = [asplit(("p1","A")),
xsplit(("p1","B",[("undef",[tdef("p1","B","tv2")],"CBDT3"),
("isdef",[tdef("p1","B","tv2")],"CBE")],"CBDT3")),
asplit(("p1","DT1")),asplit(("p1","FJ1")),asplit(("p1","FJ2")),
asplit(("p1","TSP1")),asplit(("p1","TSP2")),
xsplit(("p1","JL1",[("less3",[tdef("p1","JL1","tv5")],"CJL1J"),
("nless3",[tdef("p1","JL1","tv5")],"CJL1JL3")],"CJL1J")),
xsplit(("p1","JL4",[("true",[],"CJL4JL1"),("false",[],"CJL4JL3")],"CJL4JL1")),
asplit(("p1","JL3"))];

val iJoins = 1`[xjoin(("p1","E")),xjoin(("p1","F")),ajoin(("p1","FJ1")),
ajoin(("p1","FJ2")),ajoin(("p1","FR1")),ajoin(("p1","FR2")),ajoin(("p1","TEP1")),
ajoin(("p1","TEP2")),xjoin(("p1","JL1")),xjoin(("p1","JL3")),ajoin(("p1","L"))];

val iSM = 1`[("p1","s1",["B","D","E"])];

val iUC = [("user1",[("cap1",sval("val1")),("cap2",sval("val2"))]),
("user2",[("cap1",sval("val3"))]),("user3",[("cap2",sval("val4"))]),
("user4",[("cap1",sval("val1")),("cap2",sval("val2"))]),
("user5",[("cap1",sval("val1")),("cap2",sval("val3"))]),
("user6",[("cap1",sval("val1")),("cap2",sval("val2"))])];

val iUJ = [("user1","j1"),("user2","j2"),("user3","j3"),("user4","j3"),
("user4","j4"),("user5,"j4"),("user6","j5")];

val iFR = [([("p1","Cstart2","TSP2"),("p1","Cstart","TSP1"),("p1","CT","DT1"),
("p1","CT","TEP1"),("p1","Co","TEP1"),("p1","Cend","TEP2"),("p1","CFB1","TEP2"),
("p1","Ci","A"),("p1","CAB","B"),("p1","CAC","C"),("p1","CD","D"),("p1","DD","D"),
("p1","CBDT3","DT3"),("p1","CBE","E"),("p1","CDE","E"),("p1","CEFJ1","FJ1"),
("p1","CEFJ1","FR2"),("p1","CCFJ2","FR1"),("p1","CCFJ2","FJ2"),
("p1","CFJ1FR1","FR1"),("p1","CFJ2FR2","FR2"),("p1","CFJ1F","F"),("p1","CFJ2F","F"),
("p1","CFG","G"),("p1","Ci2","JL2"),("p1","CJL2JL1","JL1"),("p1","CJL4JL1","JL1"),
("p1","CJL1J","J"),("p1","CJJL4","JL4"),("p1","CJL1L3","JL3"),("p1","CJL4JL3","JL3"),
("p1","CJL3S1","JL3S1"),("p1","CJL3S2","JL3S2"),("p1","CJL3S3","JL3S3"),
("p1","CJL3S","K"),("p1","CM1","L"),("p1","CM2","L"),("p1","CM1","LM1"),
("p1","CM2","LM2")],[("p1","TSP2","Cstart"),("p1","TSP2","CFB1"),("p1","TSP1","CT"),
("p1","TSP1","Ci"),("p1","TEP1","Cend"),("p1","TEP2","Cend2"),("p1","DT1","CT"),
("p1","DT1","CD"),("p1","A","CAB"),("p1","A","CAC"),("p1","B","CBDT3"),("p1","DT3","DD")
("p1","B","CBE"),("p1","C","CCFJ2"),("p1","D","CDE"),("p1","E","CEFJ1"),
("p1","F","CFG"),("p1","G","Co"),("p1","FJ1","CFJ1F"),("p1","FJ1","CFJ1FR1"),
("p1","FJ2","CFJ2F"),("p1","FJ2","CFJ2FR2"),("p1","FR1","CFB1"),("p1","FR2","CFB1"),
("p1","CFB1","FJ1"),("p1","CFB1","FJ2"),("p1","JL2","CJL2JL1"),("p1","JL1","CJL1J"),
("p1","JL1","CJL1JL3"),("p1","J","CJJL4"),("p1","JL4","CJL4JL1"),("p1","JL4","CJL4JL3"),
("p1","JL3","CJL3S1"),("p1","JL3","CJL3S2"),("p1","JL3","CJL3S3"),("p1","JL3S1","CJL3S"),
("p1","JL3S2","CJL3S"),("p1","JL3S3","CJL3S"),("p1","K","CM1"),("p1","LM1","CM2"),
("p1","LM2","CM1"),("p1","L","Co2")])];

val iVD = [(gdef("p1","wv1"),true,true,"int"),
(gdef("p1","wv2"),true,true,"emprecs"),
```

```
(fdef("p1","fn1","fv1"),true,true,"string"),
(fdef("p1","fn2","fv2"),true,true,"int"),
(cdef("p1","cv1"),true,true,"string"),
(bdef("p1","p2","bv1"),true,true,"string"),
(sdef("p1","s1","sv1"),true,true,"string"),
(tdef("p1","A","tv1"),true,true,"int"),
(tdef("p1","B","tv2"),true,true,"string"),
(tdef("p1","E","tv3"),true,true,"string"),
(tdef("p1","F","tv4"),true,true,"int"),
(tdef("p1","J","tv5"),true,true,"string"),
(mdef("p1","B","mv1"),false,false,"string"),
(mdef("p1","B","mv2"),false,false,"int"),
(mdef("p1","B","mv3"),false,false,"string")];

val iWH = [("p1","p1","Cstart2","Cend2",["A","B","C","D","E","F","G",
"DT1","DT3","FJ1","FJ2","FR1","FR2","TSP1","TSP2","TEP1","TEP2"],["s1"]),
("p1","p2","Ci2","Co2",["J","JL1","JL2","JL3","JL4","JL3S1","JL3S2","JL3S3","K","LM1",
"LM2","L"],[])];

val iPM = [("p1","A",[gdef("p1","wv1")],"inc",[tdef("p1","A","tv1")],invar,mand,single),
("p1","E",[bdef("p1","p1","bv1")],"copy",[tdef("p1","E","tv3")],invar,mand,single),
("p1","F",[cdef("p1","cv1")],"copy",[tdef("p1","F","tv4")],invar,opt,single),
("p1","FJ1",[cdef("p1","cv1")],"copy",[tdef("p1","F","tv4")],invar,opt,single),
("p1","FJ2",[cdef("p1","cv1")],"copy",[tdef("p1","F","tv4")],invar,opt,single),
("p1","J",[bdef("p1","p2","bv1")],"copy",[tdef("p1","J","tv5")],invar,opt,single),
("p1","JL1",[bdef("p1","p2","bv1")],"copy",[tdef("p1","JL1","tv5")],invar,opt,single),
("p1","JL3",[bdef("p1","p2","bv1")],"copy",[tdef("p1","JL1","tv5")],invar,opt,single),
("p1","A",[tdef("p1","A","tv1")],"copy",[gdef("p1","wv1")],outvar,mand,single),
("p1","E",[tdef("p1","E","tv3")],"addc",[sdef("p1","s1","sv1")],outvar,mand,single),
("p1","F",[tdef("p1","F","tv4")],"copy",[cdef("p1","cv1")],outvar,opt,single),
("p1","J",[tdef("p1","J","tv5")],"copy",[bdef("p1","p2","bv1")],outvar,opt,single),
("p1","C",[fdef("p1","fn1","fv1")],"copy",[bdef("p1","p2","bv1")],invar,mand,single),
("p1","C",[bdef("p1","p2","bv1")],"copy",[fdef("p1","fn1","fv1")],outvar,mand,single),
("p1","B",[gdef("p1","wv2")],"copy",[mdef("p1","B","mv1"),mdef("p1","B","mv2"),
mdef("p1","B","mv3")],invar,mand,multiple),
("p1","B",[mdef("p1","B","mv1"),mdef("p1","B","mv2"),mdef("p1","B","mv3")],
"copy",[gdef("p1","wv2")],outvar,mand,multiple)];

val iVarDet = [atask("p1","A"),mitask("p1","B",1,5,2,dynamic,cancelling),
ctask("p1","C","p2"),atask("p1","D"),atask("p1","DT1"),atask("p1","DT3"),
atask("p1","E"),atask("p1","F"),atask("p1","FJ1"),atask("p1","FJ2"),
atask("p1","FR1"),atask("p1","FR2"),atask("p1","G"),atask("p1","J"),
atask("p1","JL1"),atask("p1","JL2"),atask("p1","JL3"),atask("p1","JL4"),
atask("p1","JL3S1"),atask("p1","JL3S2"),atask("p1","JL3S3"),atask("p1","K"),
atask("p1","L"),atask("p1","LM1"),atask("p1","LM2"),atask("p1","TSP1"),
atask("p1","TSP2"),atask("p1","TEP1"),atask("p1","TEP2")];

val iPre = [("p1","A","isdef",[gdef("p1","wv1")]),
("p1","","isdef",[gdef("p1","wv2")])];

val iPost = [("p1","G","def",[fdef("p1","fn2","fv2")])];

val iRems = [("p1","F",[],[],["B"])];
```

```
val iDis = [("p1","C",["B"])];

val iSU = [("p1","K","J")];

val iDU = [("p1","L","K")];

val iOG = [("og1",team,"og3"),("og2",team,"og3"),("og3",department,"og3")];

val iJM = [("j1","og1","j2"),("j2","og1","j4"),("j3","og2","j4"),("j4","og3","j5"),
("j5","og3","j5")];

val iHTD = [("p1","A","lastA")];

val iOTD = [("p1","B","org1")];

val iCTD = [("p1","J","capval1")];
```

References

[AH02] W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet another workflow language. Technical Report QUT Technical Report, FIT-TR-2002-06, Queensland University of Technology, 2002.

[AH05] W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet another workflow language. *Information Systems*, 30(4):245–275, 2005.

[Jen97] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts.* Monographs in Theoretical Computer Science. Springer-Verlag, Berlin, Germany, 1997.

[Kas06] P.S. Kastner. The business process management benchmark report. Technical report, Aberdeen Group, 2006.

[Kie03] B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows.* PhD thesis, Queensland University of Technology, Brisbane, Australia, 2003.

[MAHR06] N. Mulyar, W.M.P. van der Aalst, A.H.M. ter Hofstede, and N. Russell. A critical analysis of the 20 classical workflow control-flow patterns. Technical report, BPM Center Report BPM-06-18, 2006.

[OADH06] C. Ouyang, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. Translating BPMN to BPEL. Technical Report BPM-06-02, 2006. `www.BPMcenter.org`.

[PA05] M. Pesic and W.M.P van der Aalst. Toward a reference model for work distribution in workflow management. In E. Kindler and M. Nüttgens, editors, *Proceedings of the First International Workshop on Business Process Reference Models (BPRM'05)*, Nancy, France, 2005.

[RAHE05] N. Russell, W.M.P. van der Aalst, A.H.M. ter Hofstede, and D. Edmond. Workflow resource patterns: Identification, representation and tool support. In O. Pastor and J. Falcão e Cunha, editors, *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05)*, volume 3520 of *Lecture Notes in Computer Science*, pages 216–232, Porto, Portugal, 2005. Springer.

[RAHW06] N. Russell, W.M.P van der Aalst, A.H.M. ter Hofstede, and P. Wohed. On the suitability of UML 2.0 activity diagrams for business process modelling. In M. Stumptner, S. Hartmann, and Y. Kiyoki, editors, *Proceedings of the Third Asia-Pacific Conference on Conceptual Modelling (APCCM2006)*, volume 53 of *CRPIT*, pages 95–104, Hobart, Australia, 2006. ACS.

[RHAM06] N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, and N. Mulyar. Workflow control-flow patterns: A revised view. Technical Report BPM-06-22, 2006. `http://www.BPMcenter.org`.

[RHEA05] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow data patterns: Identification, representation and tool support. In L. Delcambre, C. Kop, H.C. Mayr, J. Mylopoulos, and O. Pastor, editors, *Proceedings of the 24th International Conference on Conceptual Modeling (ER 2005)*, volume 3716 of *Lecture Notes in Computer Science*, pages 353–368, Klagenfurt, Austria, 2005. Springer.

[WEAH05] M.T. Wynn, D. Edmond, W.M.P. van der Aalst, and A.H.M. ter Hofstede. Achieving a general, formal and decidable approach to the OR-join in workflow using Reset nets. In G. Ciardo and P. Darondeau, editors, *Proceedings of the 26th International Conference on Application and Theory of Petri nets and Other Models of Concurrency (Petri Nets 2005)*, volume 3536 of *Lecture Notes in Computer Science*, pages 423–443, Miami, USA, 2005. Springer-Verlag.

[WH06] C. Wolf and P. Harmon. The state of business process management. Technical report, BPTrends, 2006. `http://www.bptrends.com/surveys_landing.cfm`.