Generating Business Process Models from Object Behavior Models

Abstract

Object-oriented modeling is an established approach to document information systems. In an object model, a system is captured in terms of object types and associations, state machines and collaboration diagrams, among others. Process modeling on the other hand, provides a different approach whereby behavior is captured in terms of activities, flow dependencies, resources, etc. These two approaches have their relative advantages. Also, object models and process models lend themselves to different styles of implementation. In this paper we define a transformation from a meta-model for object behavior modeling to a meta-model for process modeling. The transformation relies on the identification of causal relations in the object model. These relations are encoded in a heuristics net from which a process model is derived and then simplified. Using this transformation, it becomes possible to apply established object-oriented techniques during system analysis and design, and to transform the resulting object models into executable process models that can be deployed in a workflow engine. The proposal has been implemented in an object modeling tool.

Keywords: process model, object model, model transformation.

1 Introduction

Object modeling and process modeling are two established approaches to describe information systems (Kueng, Bichler, Kawalek, & Schrefl, 1996). Each of these approaches adopts a different perspective and corresponds to a different way of thinking. Modeling an information system in terms of objects leads to the definition of object types, associations, intra-object behavior

and inter-object interactions, which are captured using notations such as Unified Modeling Language (UML) class, state and collaboration diagrams (Booch, Rumbaugh, & Jacobson, 1998). Object models group related data and behavior into classes, thus promoting modularisation and encapsulation. Purported advantages of this approach include reuse and maintainability. Object-oriented analysis and design techniques (e.g. those based on UML) are well-established and widely used in practice.

Meanwhile, process models are structured in terms of activities (which may be decomposed into sub-processes), events, control and data-flow dependencies, and associations between activities and resources. Business Process Modeling Notation (BPMN) (Object Management Group, 2006), UML activity diagrams, Business Process Execution Language (BPEL) (Andrews et al., 2003) and Yet Another Workflow Language (YAWL) (Aalst & Hofstede, 2005) are examples of notations that capture the behavior of a system in a process-oriented manner at various levels of details. Process models provide a holistic view on the activities and resources required to achieve a goal. Accordingly, they lend themselves to analysis through simulation and other quantitative analysis techniques, and they have proven instrumental in enabling communication between business and IT stakeholders (Becker, Kugeler, & Rosemann, 2003).

Moreover, object modeling and process modeling typically lead to different implementation styles. Whereas object modeling lends itself to implementation in an object-oriented programming environment, process models naturally lead to workflow applications or other types of process-aware information systems, which provide advanced monitoring and controlling functionality.

There is an opportunity to reconcile object-oriented and process-oriented approaches to information system engineering in order to benefit from their relative strengths. Each of these modeling approaches adopts a different perspective. In object models, behavior is split across object types, whereas in process models, behavior is captured along chains of logically related activities. Thus, information captured in one approach may be missing or only implicitly captured in the other approach. For example, a class in an object model may contain references to activities (e.g. in a sequence diagram), but objects are predominantly state-centric and do not explicitly define activities or control flow relations between them. Likewise, a process model contains implicit references to states or to classes representing resources, but a process model is predominantly activity-centric.

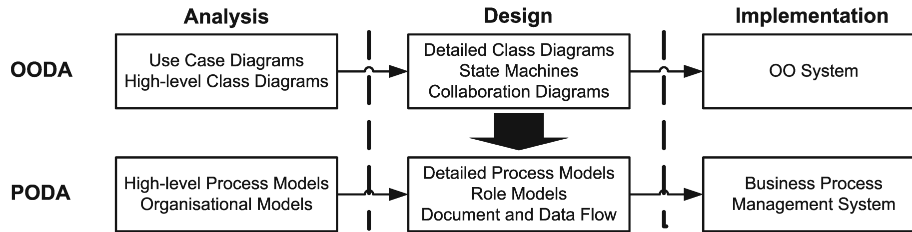

Figure 1: Transforming an object-oriented to a process-oriented approach

Figure 1 shows the typical phases and deliverables involved in the object-oriented development approach (OODA) and in the process-oriented development approach (PODA). In this paper, we investigate how to bridge the deliverables produced by the design phases of these two approaches. Specifically, we present a transformation from detailed object behavior models to process models. The transformation relies on the identification of causal relations in the object model. These relations are encoded in a causal matrix (also called heuristics net) from which we derive a process model represented in YAWL. We choose YAWL as a target language for representing process models for several reasons:

- Expressiveness: YAWL allows one to capture complex synchronization scenarios that may arise as a result of communication between objects;
- Native support for sub-processes, thus enabling us to modularize the generated process models
- Its executability, thus enabling us to test the resulting models.

Notwithstanding this choice, we argue that the proposed method could be adapted to other process modeling notations such as BPMN.

One usage scenario of the proposal is the following. As a result of a legacy of analysis and design initiatives, a substantial collection of object models of an organisation's software systems is available. However, a decision is taken to re-deploy some of the applications into a business process management system in order to benefit from the monitoring and controlling functionality that such a system provides, and to improve the level of alignment between the organisation's business processes and the supporting applications. The transformation method developed in this paper provides a foundation to migrate the existing object models into (executable) process models, thus

allowing developers to reuse the body of knowledge stored in these models. More generally, the co-existence of object-oriented and process-oriented approaches to system development may lead to situations where a project starts with a model corresponding to one approach, and needs to switch to a model corresponding to the other approach. The transformation method proposed in this paper provides a bridge between these two approaches.

The proposed method has been implemented as a prototype tool. This tool allows designers to edit object behavior models and to export them into process-oriented models that can be deployed into the YAWL of workflow system.

The paper is organised as follows. Section 2 introduces a motivating example. Section 3 defines a meta-model for object behavior modeling. Section 4 introduces an algorithm to transform an object behavior model into a process model and presents reduction rules for YAWL models. Section 5 discusses related work and Section 6 concludes the paper.

2 Example

In this section we introduce an example of an object model that we have used as a test scenario. The example deals with inspection and maintenance of heavy equipment such as open mine excavators and shipping container cranes. Such equipment is subjected to inspections at regular intervals when several issues requiring maintenance may be raised with the equipment. Depending on the severity of an issue and the criticality of the equipment some issues will be determined to be resolved with higher urgency than others. The application domain is presented as a high-level class diagram in Figure 2. Each class may have one or more state machines as discussed later.

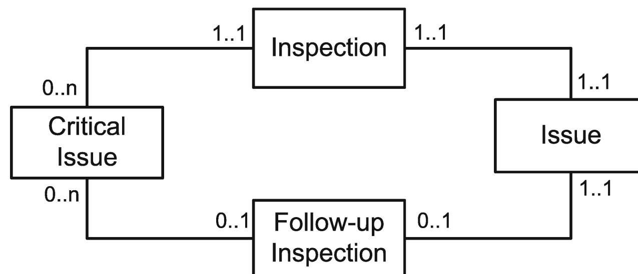

Figure 2: Asset inspection example – High-level class diagram

An *Inspection* for a particular piece of equipment may uncover zero or more *Issues* to be resolved. These are added to the set of existing *Issues* that can be derived for that piece of equipment. The main *Inspection* can only be completed after all *Issues* have been resolved and completed. A *Critical Issue* may also be detected during an *Inspection* or *Follow-up Inspection*. These *Critical Issues* are identified separately due to the elevated need to have them resolved. An *Issue* may raise a number of *Follow-up Inspections*, which in turn may lead to new (critical) *Issues* being raised, and so on.

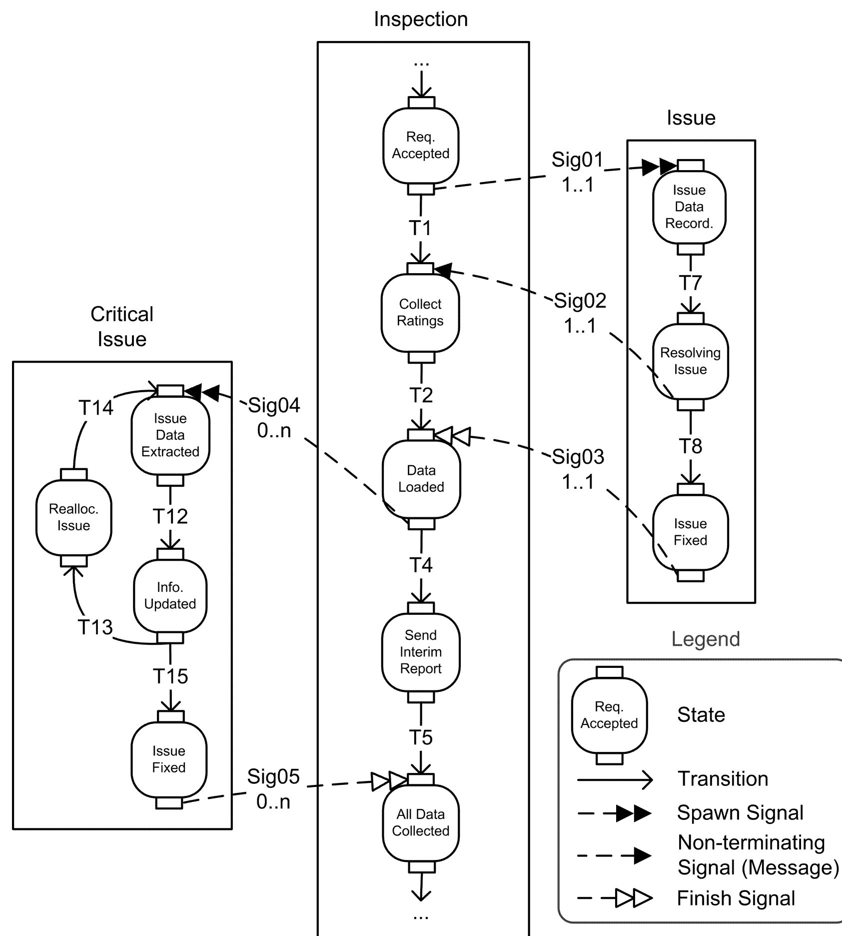

Figure 3: Example of an object behavior model

In Figure 3 a fragment of a state machine lifecycle corresponding to the *Inspection* class, as well as two related state machines corresponding to the

Critical Issue and the *Issue* classes are shown. In this example a request is accepted to begin an *Inspection* and an *Issue* is raised. The *Issue* continues to be processed until it is resolved. At this time a number of other *Critical Issues* may be identified with the equipment. During resolution of *Critical Issues* an interim report is sent to a manager. The *Critical Issues* are processed until they are resolved, at which time data is collected on all issues by the *Inspection* state machine. For reasons of space we have not included an example of a *Follow-up Inspection* lifecycle in the object behavior model.

3 Object Behavior Meta-Model

In order to present our transformation approach, we first need to agree on meta-models for representing object behavior models and process models. Meanwhile, to represent object models we adopt a meta-model inspired by FlowConnect (Shared Web Services Pty. Ltd., August, 2003), a system that supports the development of software applications based directly on executable object behavior models.

FlowConnect is an attractive source meta-model for our proposal for two reasons. Firstly, FlowConnect seamlessly integrates concepts from UML state diagrams with concepts from UML sequence diagrams, allowing us to capture both intra-object and inter-object behavior in the same model. Secondly, the FlowConnect-based meta-model is a representative of other object-oriented meta-models (e.g. Proclets (Aalst, Barthelmess, Ellis, & Wainer, 2001), Merode (Snoeck, Poelmans, & Dedene, 2000), OCoN (Wirtz, Weske, & Giese, 2001)), thus it is possible to adapt the results presented here to other meta-models.

The meta-model is presented as an Object Role Model (ORM) (Halpin, 2001) in Figure 4. At the highest level an **object model** is a container for all classes in an object-oriented model. The object model contains one or more **classes** that contain one or more **state machines**. A state machine contains one or more **states**. For example, *Request Accepted*, *Data Loaded* or *Collect Ratings* are states in an *Inspection* state machine, as shown in Figure 3.

A **transition** connects two states within a state machine. In our asset inspection example the transition *T1* connects the *Request Accepted* state to the *Collect Ratings* state in the *Inspection* state machine. A transition may optionally be labelled by an Event-Condition-Action (ECA) rule. The

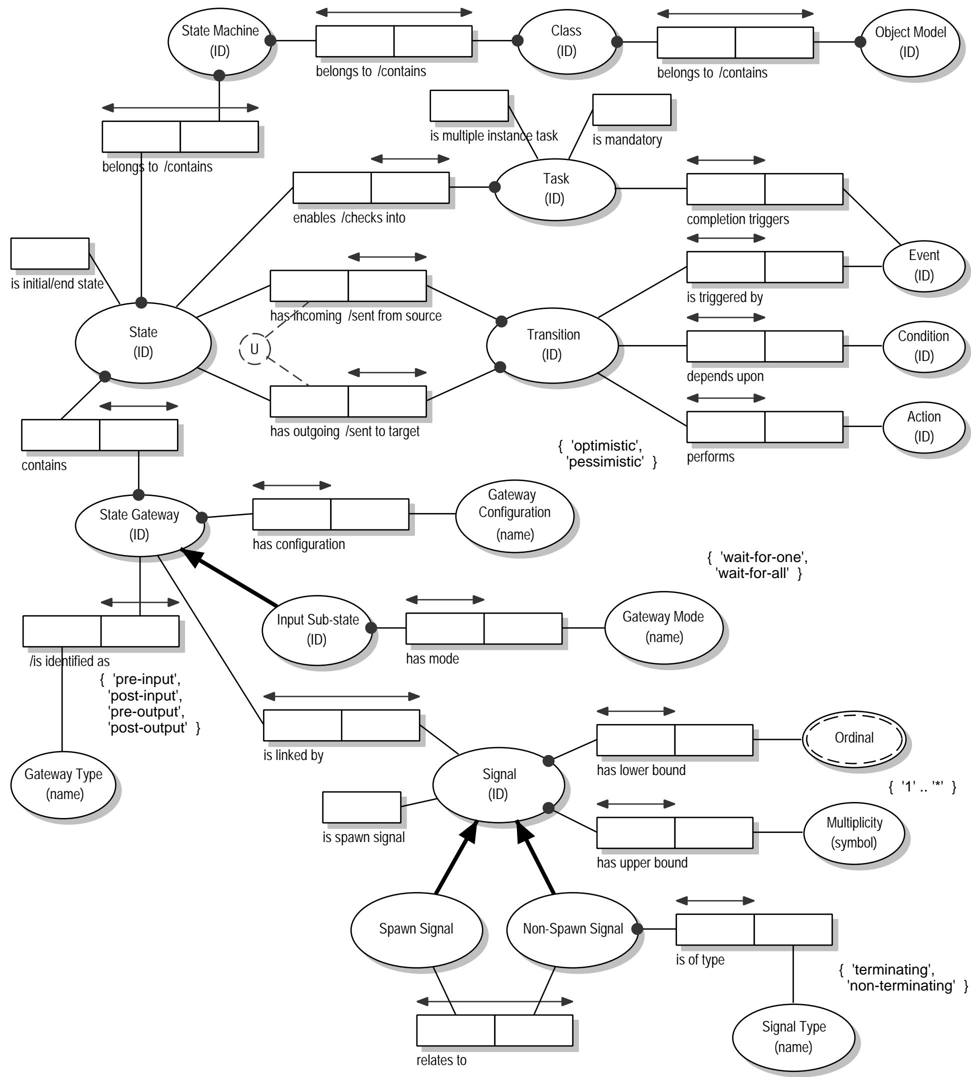

Figure 4: Object behavior meta-model

occurrence of an **event** will cause the transition labeled by that event to be performed provided that the **condition** associated with that event is

satisfied. When a transition is performed it may also execute an **action**. The details of an ECA rule language are not specified in this article since this is out of scope of this work.

Each state in a state machine contains three sub-states: a **pre-gateway**, a **main processing sub-state** and a **post-gateway**. The pre- and post-gateways are the entry and exit points of the main processing sub-state respectively. The main processing sub-state is where work is completed in a state and it may contain zero or more atomic **tasks**. A state that contains zero tasks is empty and will pass control flow directly through the main processing sub-state from the pre-gateway to the post-gateway.

A **signal** connects two gateways in different state machines together, thus capturing an interaction between objects. Signals are denoted by a dashed line. There are three types of signals: a **spawn** signal (double-filled arrowhead) starts an instance of a state machine; a **finish** signal (double-empty arrowhead) indicates that a state machine has completed its execution; and a **message** signal (single solid arrowhead) corresponding to a (non-terminating) interaction between two state machines. The output end of a spawn signal can only be connected to the pre-gateway of the initial state in the target state machine. Likewise, the input end of a finish signal can only be connected to a final state post-gateway. A signal has a lower and upper bound, which are the minimum and maximum number of times it can be sent, i.e. the spawn signal *Sig04* with a lower bound of 1 and upper bound of 'n' can create between 1 to 'n' *Critical Issues*. For example, a message signal *Sig02* sent by the post-gateway in the *Resolving Issue* state in the *Issue* state machine can only occur following a spawn signal *Sig01* sent by the post-gateway in the *Req. Accepted* state in the *Inspection* state machine.

Since signals can be both sent and received by a pre-gateway or post-gateway, all gateways consist of two parts; an input part and an output part. The input part of a gateway receives incoming signals from gateways in other state machines. The output part allows a gateway to send outgoing signals to gateways in other state machines. The order in which signals are sent or received by a gateway depends on the gateway configuration: a *pessimistic* gateway waits to receive all signals that it expects before sending any (the input part comes before the output part), whereas an *optimistic* gateway sends signals before waiting to receive any (the output part comes before the input part). A pre-gateway has an additional mode to specify whether it should wait for the first signal (*wait-for-one*) or all signals (*wait-for-all*) before control flow will be released by the gateway. An example of a

pessimistic gateway in *wait-for-all* mode is the pre-gateway of the *All Data Collected* state of the *Inspection* state machine.

4 From Object Behavior Models to Process Models

In this section we introduce a proposal to transform an object behavior model into a process model. The aim of the transformation is to preserve the causal dependencies in the original object model. Transitions in each individual state machines capture such causal dependencies. But in addition, a signal between different state machines also caputures a (partial) causal dependency between an action in the source state machine and an action in the target state machine.

Accordingly, the essence of the transformation is to analyse the object behavior model in order to extract a set of elementary causal dependencies between events and signals. These elementary causal dependencies are represented as a *causal matrix*, also known as a *heuristics net* (Aalst, Medeiros, & Weijters, 2005). From the causal matrix it is possible to obtain a Petri net, which is then transformed into a YAWL net. The idea of using a heuristics net comes from the ProM framework (Dongen, Medeiros, Verbeek, Weijters, & Aalst, 2005), where heuristics nets are used as an intermediate step to construct a Petri net from an event log.

Background: Heuristics nets

A heuristics net is composed of a set of transitions, which we call "tasks" to put them in the context of this paper. Each task has an *input* and an *output*. The input of a task T represents the different ways in which task T can be started. Concretely, the input of a task is a set. If this set is empty, it means that the task can be started even if no other task has been completed (i.e. this is the initial task in the process model). If the input of a task is not empty, it contains one of several *disjunctions*. Each of these disjunctions should be read as an "Or" of several tasks. For example, the Petri Net in Figure 5 has a disjunct { F,B,E } meaning that a choice between either task F or task B or task E is made after task A has been completed.

The different disjunctions in an input are implicitly linked through an "And", meaning that each disjunct must be satisfied before the target task

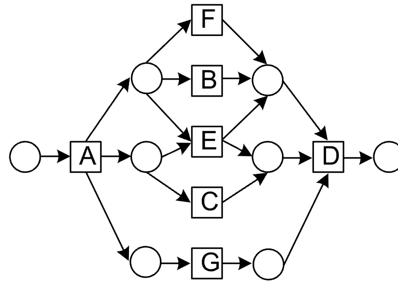

Figure 5: Example heuristics net shown as a Petri net

can be started. For example, the input of task D is { {F,B,E}, {E,C}, {G} }. For task D to be executed, either F or B or E must be completed, and either E or C must be completed, and G must be completed.

Symmetrically, the output of a task determines which other tasks can be executed after a given task completes. An empty output denotes a final task in the process. Meanwhile, a non-empty output must be read as a set of disjuncts and can contain multiple disjunctions. In Figure 5 the output of task A is { {F,B,E}, {E,C}, {G} }, which means that after A is completed, either F or B or E will be executed, and either E or C will be executed, and G will be executed. Readers familiar with Petri nets will recognise that a heuristics net is a Petri net of a particular form.

Transformation procedure

The transformation procedure consists of the following three steps, which are performed in the order depicted in Figure 6:

I - Generate a heuristics net from an object model/state machine diagrams.

II - Generate a Petri net from a heuristics net.

III - Transform the Petri net into a YAWL process model.

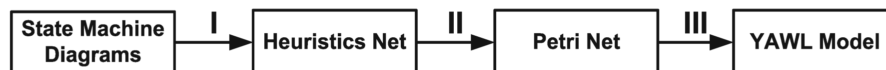

Figure 6: An overview of the transformation procedure

Below, we present an algorithm that automates **Step I**. For each state in an object model, this algorithm generates two tasks corresponding to the pre- and post-gateway. In other words, each pre- or post-gateway in the object model will lead to one task in the generated heuristics net. The algorithm then identifies causal dependencies between these tasks. Causal dependencies are derived from the transitions within a state machine, but also from the interactions (i.e. signals) between state machines. Thus, indirectly, interactions between object types are mapped into control-flow relations in the resulting process model.

Algorithm 1 takes as input an object model and produces the corresponding heuristics net. The algorithm iterates over each state gateway in order to generate an input set (preTask) and output set (postTask). Because there is a one-to-one mapping between state gateways and tasks, the algorithm treats them interchangeably, meaning that it uses the identifiers of gateways in the source object model as identifiers of tasks in the generated heuristics net.

The following auxiliary functions are used in *Algorithm 1*:

- *states* : ObjectModel → Set of State, is the set of states in an object model.
- *pre, post* : State → Gateway, yields the pre or post gateway of a state.
- *inputTransitions, outputTransitions* : State → Set of Transition, yields the set of input/output transitions.
- *source, target* : Transition → State, yields a transition's source/target.
- *inputSignals, outputSignals* : Gateway → Set of Signal, yields a gateway's input/output signals.
- *mode* : Gateway → GatewayMode, yields a gateway's mode.
- *explode* : Set of Signal → Set of Set of Signal. explode($\{e_1, e_2, \dots, e_n\}$) = $\{\{e_1\}, \{e_2\}, \dots, \{e_n\}\}$.

Algorithm 1: Generation of a heuristics net

Input: om : ObjectModel
Output: preTask, postTask : Task → Set of Set of Task
predecessors, successors : Set of Gateway
foreach $s \in$ *states(om)* **do**
    predecessors := { post(source(t)) | t ∈ inputTransitions(s) };
    successors := { pre(target(t)) | t ∈ outputTransitions(s) };
    preInputSignals := { source(g) | g ∈ inputSignals(pre(s)) };
    preOutputSignals := { source(g) | g ∈ inputSignals(post(s)) };
    postInputSignals := { target(g) | g ∈ outputSignals(pre(s)) };
    postOutputSignals := { target(g) | g ∈ outputSignals(post(s)) };
    if *mode(pre(s)) = wait-for-one* **then**
        preTask(pre(s)) := { predecessors, preInputSignals };
    else
        preTask(pre(s)) := { predecessors } ∪
        explode(preInputSignals);
    postTask(pre(s)) = { { post(s) }, postInputSignals(s) };
    if *mode(post(s)) = wait-for-one* **then**
        preTask(post(s)) := { { pre(s) }, preOutputSignals(s) };
    else
        preTask(post(s)) := { { pre(s) } } ∪
        explode(preOutputSignals(s));
    postTask(post(s)) := { successors } ∪
    explode(postOutputSignals(s));
end

To analyse the inbound and outbound causal dependencies of a gateway, we conceptually decompose each gateway into two parts: the input and the output. The input corresponds to the signals the gateway has to wait for, while the output corresponds to the signals it has to send out. Figure 7 depicts the decomposition of the pre- and post-gateways of a state into an input and output part.

The input and output sets are generated as follows. The pre-gateway input set is the union of the source of each incoming transition with the source of each incoming signal, depending on the gateway mode (i.e. *waits-for-one* or *waits-for-all*). If the gateway mode is *waits-for-one* then the preTask set is the set of input transition sources (predecessors) and the set of pre-gateway input signal sources. However if the gateway mode is *waits-for-all* then the

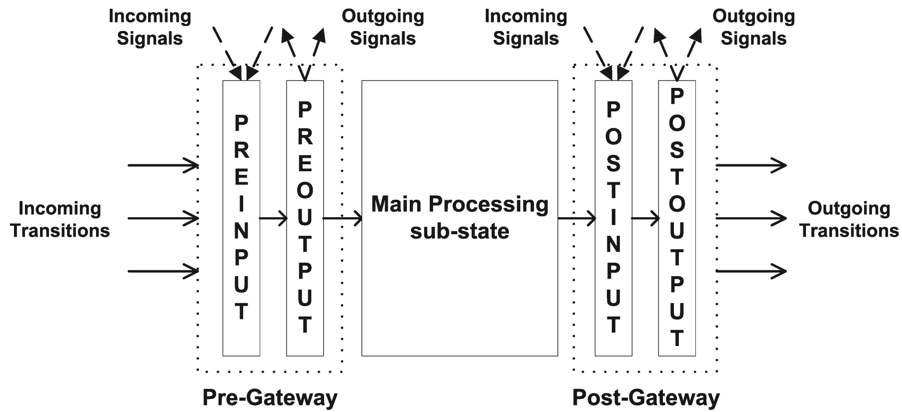

Figure 7: Pre- and post-gateways of a state

preTask set is the union of the set of input signal sources converted to a set of set of signals by the *explode* function with the set of predecessors. The post-Task set consists of the post-gateway and the targets of all outgoing signals sent from the pre-gateway. The procedure for constructing the input and output sets for a post-gateway is symmetric to the corresponding procedure for a pre-gateway. After the application of *Algorithm 1* a heuristics net is constructed by inserting the input and output set as new rows in the net.

A heuristics net is a 'flat' representation of a process since it does not capture sub-processes. When converting a heuristics net to a YAWL net, it is desirable to incorporate sub-processes in the generated process model to achieve some degree of modularity. This is done by identifying "sub-process delimiters" in the object behavior model. Sub-process delimiters are the points where an instance of a state machine is created by a spawn signal, and the point(s) where a state machine finishes by sending a terminating signal back to its parent process. In the absence of any message signals between the parent and the child state machines, and assuming there is a single finish signal located at the end of the of the child state machine, then the child state machine can be modeled as a sub-process invoked by the parent process.

YAWL has two kinds of sub-processes: composite tasks (which capture a simple sub-process invocation) and multiple-instance composite tasks (a sub-process that is executed multiple times concurrently). If the multiplicity of the spawn signal from the parent to the child state machine equals "one", the child state machine is embedded into a composite task in the parent process.

If it is greater than one, the sub-process is embedded into a multiple instance composite task.

In **Step II** of the transformation procedure the heuristics net is passed to the heuristics net conversion tool in ProM to obtain a Petri net. **Step III** is performed using a workflow net conversion plugin in the ProM framework to combine the sub-process delimiters with the derived Petri net to create an 'unflattened' YAWL process model where properties such as sub-process definitions and task multiplicity are restored in the YAWL model as shown in Figure 8. This step is merely a syntactic transformation that aims to exploit the constructs in YAWL because any Petri net can be seen as a YAWL model.

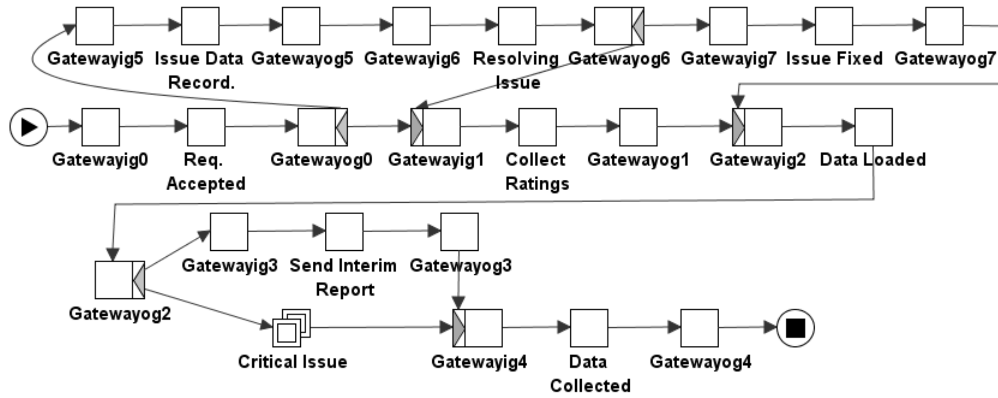

Figure 8: Inspection process model in YAWL

YAWL Model Reduction Rules

The YAWL nets generated by the transformation method presented above may contain tasks with empty decompositions, which we hereby call *epsilon tasks*. Indeed, in *Algorithm 1* gateways in an object-oriented model are converted to tasks in the heuristics net. These tasks then become empty transitions when the heuristics net is converted to a Petri net, and these empty transitions become epsilon tasks when the Petri net is converted to YAWL. The resulting epsilon tasks thus correspond to state pre- and post-gateways in the original model. For example, the epsilon task "Gatewayog5" in Figure 8 corresponds to the post-gateway of state "Issue Data Record" in Figure 3.

Below, we present four reduction rules to post-process the generated YAWL nets in order to eliminate epsilon tasks. The following notation is used to represent these reduction rules:

- ε: An epsilon task that has no decomposition.
- T: A task that may or may not have a decomposition.
- X: a placeholder for a control flow join decorator which may be bound to an XOR-join, AND-join, or an OR-join decorator when the corresponding rule is applied.
- Y: a placeholder for a control flow split decorator which may be bound to an XOR-split, AND-split, or an OR-split decorator when the corresponding rule is applied.

We now define and illustrate the following four reduction rules:

1. Task Input Combination.
2. Task Output Combination.
3. Task Join Combination.
4. Task Split Combination.

1. Task Input Combination: If a control node ε is connected via an outgoing arc to a task node T, where T has no more than one incoming arc from ε and ε has no other outgoing arcs, then the incoming arcs of the join X become incoming arcs to T as shown in Figure 9.

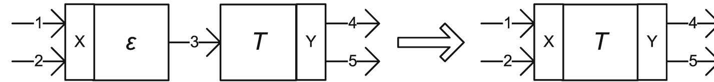

Figure 9: Task Input Combination Reduction Rule

2. Task Output Combination: If a task node T is connected via an outgoing arc to a control node ε where T has no more than one outgoing arc to ε and ε has no more than one incoming arc from T, then the outgoing arcs of the split Y become outgoing arcs of T, as shown in Figure 10.

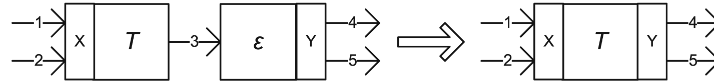

Figure 10: Task Output Combination Reduction Rule

3. Task Join Combination: If a control node ε is connected to a task T and ε has more than one incoming arc and both ε and T have the same join type (e.g. AND-join), then ε and T can be combined to become T. The incoming arcs to both ε and T are combined to become incoming arcs to the join X at T (minus the arc that connects ε to T) as shown in Figure 11.

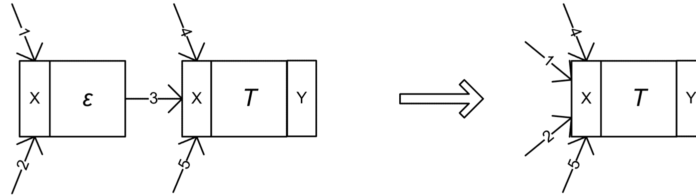

Figure 11: Task Join Combination Reduction Rule

4. Task Split Combination: If a task T is connected to a control node ε and T has more than one outgoing arc and both nodes have the same split type (e.g. AND-split) and ε has no more than one incoming arc from T, then the outgoing arcs from T and ε (minus the arc connecting T to ε) can be combined to become outgoing arcs of T as shown in Figure 12.

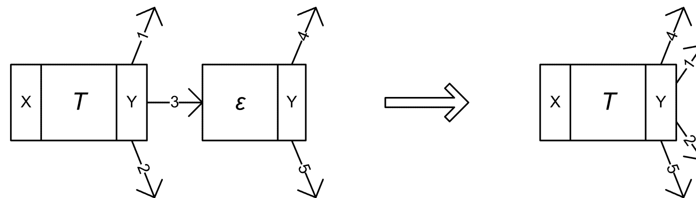

Figure 12: Task Split Combination Reduction Rule

Additional considerations are needed to deal with the conditions on the arcs of the resulting split if Y is an XOR-split or an OR-split decorator. Let us first examine the case where Y is an XOR-split. In YAWL an evaluation order is associated with outgoing arcs of an XOR-split. At runtime control is passed along the first arc in this evaluation order for which the condition

evaluates to true. If there is no arc for which the condition evaluates to true, then control is passed along the arc that is last in the evaluation order, regardless of its associated condition. Let us call r the arc that joins that two tasks to be merged, and c the condition associated with r. We distinguish the case where r is the last arc in the evaluation order, from the case where r is not the last arc. If r is the last arc, the evaluation order of the arcs of the combined XOR-split should be the following: first should come the arcs of the first XOR-split until arriving at arc r (and no including arc r which is removed because of the merger). The arcs of the second XOR-split should then be evaluated in their original order.

If r is not last in the evaluation order, the condition of each arc associated with the second XOR-split should be changed to reflect that control can only be passed along that arc if both condition c and the original condition associated to that arc evaluate to true. For example, if the tasks in Figure 13 are combined using the Task Join Combination rule where the arcs of the first and second XOR-split are evaluated from top to bottom, then the evaluation order of the combined task is s, u, v, t (r is replaced by u, v) and the conditions associated with u and v become conjunctions consisting of their original condition and condition c. In contrast, if r was last in the evaluation order, the conditions associated with the arcs of the second XOR-split do not need to be changed because control would pass along r even if c evaluated to false.

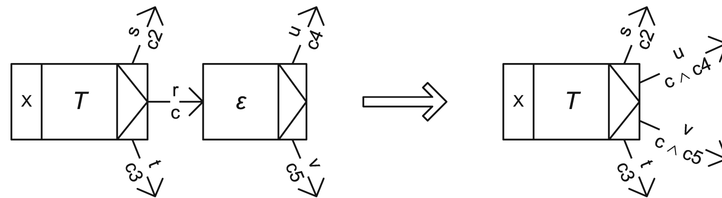

Figure 13: Task Split Combination Reduction Rule (XOR-split)

Let us now consider the case where Y is an OR-split. In YAWL, an OR-split has a designated outgoing arc, namely the *default arc*, along which the control flow is passed at runtime if none of the conditions associated with its outgoing arcs evaluates to true (including the condition of the default arc). We distinguish the case where arc r is the default arc and the case where r is not the default arc.

In the former case, control would pass to the second node in the original

model if either all conditions associated with the outgoing arcs of the first OR-split evaluate to false, or the condition associated with the default arc of this first OR-split evaluates to true. Accordingly, the condition associated with an arc that originates from the second OR-split should be equal to the conjunction of the original condition and a disjunction consisting of c and a conjunction where the elements are negations of the conditions associated with the outgoing arcs of the first OR-split (excluding r). For example, if the tasks in Figure 14 are combined using the Task Join Combination rule where r is the default arc of the first OR-split and u is the default arc of the second OR-split, the conditions of the arcs u and v become $c4 \wedge (c \vee (\neg c2 \wedge \neg c3))$ and $c5 \wedge (c \vee (\neg c2 \wedge \neg c3))$ respectively. In addition, the default arc of the second OR-split becomes the default arc of the OR-split resulting from the reduction. In this example, u becomes the default arc.

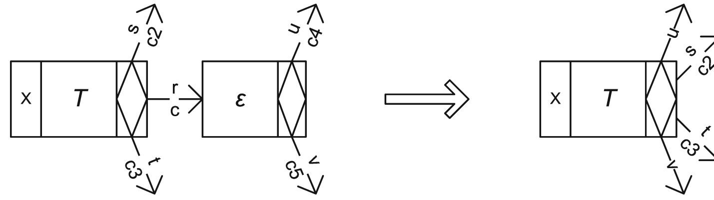

Figure 14: Task Split Combination Reduction Rule (OR-split)

The situation is simpler if r is not the default arc. In this case, the condition of an arc originating from the second OR-split is equal to the conjunction of c and the original condition. Meanwhile, the conditions in the arcs originating from the first OR-split are carried over unchanged. Finally, the OR-split resulting from the reduction is equal to the default arc from the first OR-split.

Also, the proposed reduction rules only deal with patterns consisting of combinations of tasks (with or without decorators). They do not deal with cancellation regions, but this is not an issue since the YAWL nets generated by the proposed transformation do not have cancellation regions. Similarly, the generated YAWL models do not have explicit conditions (except for start and end conditions) and therefore we do not need to deal with explicit conditions in the reduction rules.

The application of the model reduction rules to the YAWL model from the asset inspection example (Figure 8) is shown in Figure 15. In this example it has been possible to apply the reduction rules to merge all of the control

nodes (pre- and post-gateways) with tasks. The model is now clearer to read and the original control flow behavior has been maintained.

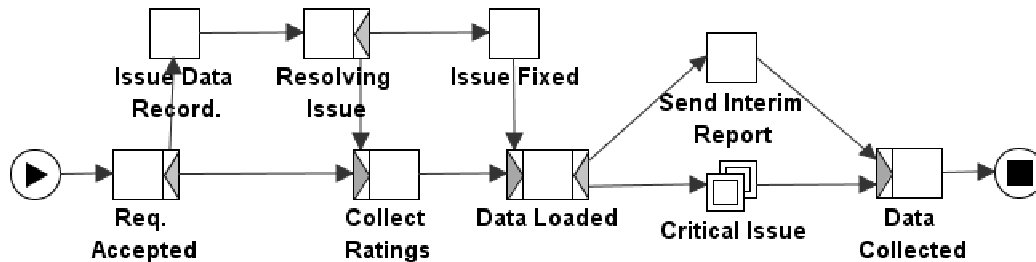

Figure 15: Reduced Inspection process model in YAWL

Other model reduction rules have been proposed in the literature: these include Murata's Petri net transformation rules (Murata, 1989) and Wynn's state-space reduction rules (Wynn, 2006). Murata's rules allow one to transform a Petri net while preserving soundness: if the original net is sound, the net resulting after applying a transformation rule is also sound. Murata's rules can also be used to eliminate tau-transitions (i.e. transitions without labels) in a net while preserving its semantics. However, Murata's rules are defined on Petri nets and need some adaptation to be applicable to YAWL nets. This is the idea followed by Wynn et al. (Wynn, 2006) who propose a set of reduction rules for cutting down large YAWL nets to ease verification. Again, these rules could potentially be used to eliminate tasks with empty decompositions in YAWL models. However Wynn's reduction rules are much more complex than what we need to post-process YAWL nets. This is the reason why we designed a specific set of reduction rules.

Implementation and testing

The proposal including the model reduction rules has been implemented in Java on top of the Eclipse platform.[1] The tool includes a graphical editor for object behavior models and a module that supports the transformation of state-based object behavior models to YAWL nets. The tool implementation relies on libraries from the ProM framework[2] to perform the transformation

[1]http://www.eclipse.org/platform/

[2]http://www.processmining.org

from heuristic nets to Petri nets and from Petri nets to flat YAWL models. Subsequently, these YAWL models are unflattened and then reduced according to the reduction rules as explained above.

The modeling tool, the model transformation technique and the reduction rules have been tested using several variants of an asset inspection example. The working example shown in this paper is one of the simplest variants. Other variants of different sizes and levels of complexity were also used in the tests including one with 53 states and 13 objects.

Future plans for the tool includes adding support for data flow modeling and resource modeling and enhancing the model transformation technique to cater for these other modeling perspectives.

5 Related Work

Object-oriented (OO) design methodologies that use the UML to design and develop Information Systems have been proposed such as OCoN (Wirtz et al., 2001). These proposals link UML diagrams to phases of the process development lifecycle to produce a schema as output at the conclusion of the lifecycle. Our proposed approach extends these design methodologies by allowing process analysts and designers to produce a completely process-oriented view of an OO model.

An architecture for mapping between OO and activity-oriented process modeling approaches has been proposed by Snoeck et al. (Snoeck et al., 2000). Object associations and business rules are captured using object-relationship diagrams and an object-event table models the behavior of domain objects, which are similar to our mapping artifacts. Aspects that appear to be missing from this architecture include a consideration of the various kinds of control flow splits and joins between objects. We consider that this control flow information is necessary for all process specifications and this information should be derived directly from an object-oriented analysis.

The object behavior model that we consider in this paper is based on the one supported by FlowConnect: a workflow management system developed by Shared Web Services.[3] The FlowConnect meta-model is an example of an executable object behavior (meta-)model tailored to support the design and automation of business processes. Another notable example is the Business State Machines model supported by IBM Websphere Process Server (IBM

[3] http://www.sws.com.au

Corporation, 2005), which also relies on a paradigm based on communicating state machines.

Proclets (Aalst et al., 2001) are a formal (Petri net-based) model for representing workflows in terms of collections of modules, each of which captures the behavior of a class of objects. Proclets provide a formal basis for reasoning about behavioral correctness, in particular, to identify deadlocks. Another formal approach to object-oriented business process modeling, namely *artifact-centric process modeling*, has been proposed by Bhattacharya et al. (Bhattacharya, Gerede, Hull, Liu, & Su, 2007), based on earlier work reported in (Nigam & Caswell, 2003). An artifact-centric process model is composed of a collection of artifact schemas and service schemas, where services act upon (and coordinate) artifacts. The behavior of artifacts and services is captured as state-transition systems. Bhattacharya et al. provide a computational complexity analysis of static analysis problems over these models. In our work, we have not considered static analysis issues, although we hint that static analysis could be performed on the YAWL nets generated from an object model.

Reijers et al. proposed a methodology for Product-Based Workflow Design (PBWD) that presented an analytical clean-sheet approach for process design specified by the bill-of-material for products that are affected by a process (Reijers, Limam, & Aalst, 2003). However, PBWD focuses on inferring the high-level structure of a business process, and not its detailed behavior. In particular, the PBWD approach does not consider the issue of describing the behavior of product lifecycles in an executable manner (e.g. as state machines), and automatically generating business process models from such lifecycles.

One of the closest works to ours is that of Küster et al. (Küster, Ryndina, & Gall, 2007) who also define a transformation from object behavior models to process models. Küster et al. represent object behavior models (called object lifecycles) as state machines and process models as UML activity diagrams. In addition to the difference in the target language (activity diagrams versus YAWL), we see two other differences between our proposal and that of Küster et al. First, the process models generated by the transformation method of Küster et al. are flat (i.e. they do not contain sub-processes). Second, the object behavior meta-model that we consider is more sophisticated. In our object meta-model, synchronisation dependencies are captured as asynchronous signals that are sent and received before and/or after each state. The meta-model we consider also allows us to capture signals that

one object of one type sends to multiple objects of another type (i.e. 1-n relations). In contrast, in the model considered in Küster et al., synchronisation dependencies are captured as *synchronisation events* which are akin to synchronous message exchanges between one object of one type and one object of another type.

This article consolidates and extends the work that we previously presented in (Redding, Dumas, Hofstede, & Iordachescu, 2008). In this previous publication, we did not consider the issue of generating YAWL models with sub-processes (instead we generated flat models) and we did not consider the possibility of using reduction rules to improve the readability of the generated YAWL models. Moreover, in this extended version, we also report on an implementation of the proposal in the form of a modeling tool that exports object behavior models as YAWL nets.

6 Conclusions and Future Work

Object technology is a mainstream approach to implementing Information Systems. Mainstream object-oriented analysis and design practices (e.g. those based on UML) are based on concepts of objects whose structure is captured as classes and whose behavior and interactions are captured as state machines, sequence diagrams and similar notations. On the other hand, recent trends have seen an uptake of approaches to Information Systems engineering that treat processes as a central concept throughout the system development lifecycle.

The co-existence of these two approaches may lead to situations where a project starts with a model corresponding to one approach and needs to switch to a model corresponding to the other approach. In this paper, we have proposed an approach to help bridge these differences in terms of the control flow logic and discussed how the conversion technique has been implemented.

Future work will continue on the topic of transforming object-oriented models to process-oriented models and vice-versa. There is a need to cover not only the control-flow aspects as outlined in this paper, but also data flow and resource allocation. We also note that a similar problem arises in the opposite direction, i.e. moving from a process-oriented to object-oriented design for the purpose of implementing process-oriented design models using object-oriented technology. Therefore another future challenge will be a proposal of a reverse transformation from process-oriented to object-oriented

models.

References

Aalst, W. M. P. van der, Barthelmess, P., Ellis, C. A., & Wainer, J. (2001). Proclets: A Framework for Lightweight Interacting Workflow Processes. *International Journal of Cooperative Information Systems*, *10*(4), 443–481.

Aalst, W. M. P. van der, & Hofstede, A. H. M. ter. (2005). YAWL: Yet Another Workflow Language. *Information Systems*, *30*(4), 245–275.

Aalst, W. M. P. van der, Medeiros, A. K. A. de, & Weijters, A. J. M. M. (2005, June 20-25). Genetic Process Mining. In *Applications and Theory of Petri Nets, 26th International Conference, ICATPN 2005* (pp. 48–69). Miami, USA: Springer.

Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., et al. (2003). *Business Process Execution Language for Web Services, Version 1.1.* (http://dev2dev.bea.com/webservices/BPEL4WS.html)

Becker, J., Kugeler, M., & Rosemann, M. (2003). *Process Management*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Bhattacharya, K., Gerede, C., Hull, R., Liu, R., & Su, J. (2007, September 24-28). Towards Formal Analysis of Artifact-Centric Business Process Models. In *Proceedings of the 5th International Conference on Business Process Management (BPM)* (pp. 288–304). Brisbane, Australia: Springer.

Booch, G., Rumbaugh, J., & Jacobson, I. (1998). *The Unified Modeling Language User Guide*. Boston, MA, USA: Addison-Wesley Professional.

Dongen, B. F. van, Medeiros, A. K. A. de, Verbeek, H. M. W., Weijters, A. J. M. M., & Aalst, W. M. P. van der. (2005, June 20-25). The ProM Framework: A New Era in Process Mining Tool Support. In *26th International Conference on Applications and Theory of Petri Nets (ICATPN)* (pp. 444–454). Miami, USA: Springer.

Halpin, T. (2001). *Information modeling and relational databases: from conceptual analysis to logical design*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

IBM Corporation. (2005). *Business State Machines.* `http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/topic/com.ibm.wbit.help.ae.ui.doc/topics/cundstat.html`.

Kueng, P., Bichler, P., Kawalek, P., & Schrefl, M. (1996). How to compose an object-oriented business process model? In *Proceedings of IFIP TC8, WG8.1/8.2 working conference on method engineering* (pp. 94–110). London, UK: Chapman & Hall, Ltd.

Küster, J. M., Ryndina, K., & Gall, H. (2007, September 24-28). Generation of Business Process Models for Object Life Cycle Compliance. In *Proceedings of the 5th International Conference on Business Process Management (BPM)* (pp. 165–181). Brisbane, Australia: Springer.

Murata, T. (1989, April). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE (invited paper)*, *77*(4), 541-580.

Nigam, A., & Caswell, N. S. (2003). Business artifacts: An approach to operational specification. *IBM Systems Journal*, *42*(3), 428-445.

Object Management Group. (2006). *Business Process Modelling Notation, Ver 1.0.* (http://www.bpmn.org)

Redding, G., Dumas, M., Hofstede, A., & Iordachescu, A. (2008, February). Transforming Object-oriented Models to Process-oriented Models. In *Business Process Management Workshops – Proceedings of the International Workshops, BPI, BPD, CBP, ProHealth, RefMod, semantics4ws* (Vol. 4928). Brisbane, Australia: Springer.

Reijers, H. A., Limam, S., & Aalst, W. M. P. van der. (2003). Product-Based Workflow Design. *Journal of Management Information Systems*, *20*(1), 229–262.

Shared Web Services Pty. Ltd. (August, 2003). *FlowConnect Model.* (http://www.flowconnect.com.au)

Snoeck, M., Poelmans, S., & Dedene, G. (2000, August). An architecture for bridging OO and business process modelling. In *33rd International Conference on Technology of Object-Oriented Languages (TOOLS)* (pp. 132–143). Mont-Saint-Michel, France: IEEE Computer Society.

Wirtz, G., Weske, M., & Giese, H. (2001). The OCoN Approach to Workflow Modeling in Object-Oriented Systems. *Information Systems Frontiers*, *3*(3), 357–376.

Wynn, M. T. K. (2006). *Semantics, Verification, and Implementation of Workflows with Cancellation Regions and OR-joins*. Unpublished doctoral dissertation, Queensland University of Technology.