Workflow Patterns in Orc

William R. Cook, Sourabh Patwardhan, and Jayadev Misra

Department of Computer Sciences, University of Texas at Austin
{wcook,sourabh,misra}@cs.utexas.edu

Abstract. Van der Aalst recently proposed a set of *workflow patterns* to characterize the kinds of control flow that appear frequently in workflow processes. These patterns are useful for evaluating the capabilities of workflow systems and models. In this paper we provide implementations of the workflow patterns in Orc, a new process calculus for orchestrating wide-area computations. A key feature of the Orc implementations is that they are expressed as *definitions* that can be reused as needed.

1 Introduction

The concept of workflow is familiar to anyone who has worked in an organization: achieving almost any goal requires coordination of multiple activities involving multiple participants. These activities are typically subject to many constraints and dependencies governing the order of activities and the capabilities of participants. Exceptional situations, interrupts, and failures must also be handled without losing sight of the end goal.

Despite the familiar and prosaic nature of workflow, developing formal models and languages for expressing workflows has proven to be a significant research challenge. The Workflow Management Coalition defines workflow informally as "The computerised facilitation or automation of a business process, in whole or part." [14] Their reference model defines vocabulary and identifies the interfaces into and out of a workflow system, but it does not provide a formal model of workflow.

Formal models of concurrency are being applied to the analysis of workflow. Petri Nets, which are a variant of finite state automata, have been used to model workflows for many years [1, 5]. Others have proposed using the π-calculus as a workflow model [9]. UML activity diagrams, which are a form of flowchart, have also been used extensively in analysis and design of workflows [4, 6]. There is as yet no widely-accepted formal model of workflow. The lack of a fundamental model of workflow makes it difficult to compare different models.

Recently van der Aalst proposed a set of workflow patterns [2] to characterize the kinds of control flow that appear frequently in workflow processes. The patterns facilitate comparison of very different workflow products and models: products can be compared quantitatively by counting the number of workflow patterns they can express directly, and qualitatively by examining the complexity of each pattern's implementation. The patterns have been implemented in a

wide range of systems, providing surprising range of solutions to these common problems [2, 11, 13, 9].

This paper shows how Orc [8], a new orchestration language, can be used to implement the workflow patterns. Orc is a process calculus in which basic services, like user interaction and data manipulation, are implemented by primitive *sites*. Orc provides constructs to orchestrate the concurrent invocation of sites to achieve a goal – while managing time-outs, priorities, and failure of sites or communication. Orc has already been used to implement a variety of traditional concurrent programming patterns [8], some of which overlap with the workflow patterns.

One difficulty in using van der Aalst's patterns is that the patterns are not formally defined. The informal descriptions are suggestive but in many cases admit several interpretations. The implementations in this paper are based on a study of the original pattern descriptions [2] and their implementation in [11].

2 Overview of Orc

An Orc program consists of a set of definitions and a *goal* expression which is to be evaluated. The evaluation of the goal expression calls sites (see below) and defined expressions, and *publishes* values. In this section, we give an informal overview of the programming model. For more a more detailed discussion and a formal semantics, see [8].

2.1 Syntax

In the following syntax, E is an expression name, M a site name, x a variable, c a constant, $\bar{p}$ a list of actual parameters and $\bar{q}$ a list of formal parameters.

$$\begin{array}{rl}
e, f, g, h \in \mathit{Expression} & ::= M(\bar{p}) \;\|\; E(\bar{p}) \;\|\; f > x > g \;\|\; f \mid g \;\|\; f \textbf{ where } x :\in g \;\|\; x \\
p \in \mathit{Actual} & ::= x \;\|\; M \;\|\; c \;\|\; f \\
q \in \mathit{Formal} & ::= x \;\|\; M \\
\mathit{Definition} & ::= E(\bar{q}) \;\underline{\Delta}\; f
\end{array}$$

An expression can be a site call $M(\bar{p})$, or a call to a defined expression $E(\bar{p})$. There are only three operators: $>x>$ for sequential composition, $\mid$ for parallel composition, and **where** for asymmetric parallel composition. The operators are listed in decreasing order of precedence, so that $f > x > g \mid h$ means $(f > x > g) \mid h$. The following sections discuss each kind of expression in turn. The syntax of Orc is extended here to include expressions as arguments in calls to definitions, using the same substitution semantics given in [8].

2.2 Site Call

The simplest Orc expression is a *site* call $M(\bar{p})$, where M is a site name and $\bar{p}$ is a list of actual parameters. A site is a separately defined procedure, like a web service. The site may be implemented on the client's machine or a remote machine. A site call elicits at most one response; it is possible that a site never responds to a call.

A site call $CNN(d)$, where CNN is a news service and d is a date, may download the newspage for the specified date. Calling $Email(a, m)$ sends message m to address a, causing permanent change in the state of the recipient's mailbox, and returns a signal to the client to denote completion of the operation. Calling an airline flight-booking site returns the booking information and causes a tentative state change in the airline database.

Site calls are *strict*, i.e., a site is called only if all its parameters have values.

We define a few sites in Table 1 that are fundamental to effective programming in Orc. Additionally, **0** represents a site which never responds; it may be used to terminate certain parts of a computation. Orc expressions can use *Rtimer* to manage time, although none of the current workflow patterns require this ability.

$let(x, y, \cdots)$	Returns argument values as a tuple.
$if(b)$	Returns a signal if b is true, and it does not respond if b is false.
Signal	Returns a signal. It is same as $if(true)$.
$Rtimer(t)$	Returns a signal after exactly t time units.

Table 1. Fundamental Sites

We have made very few assumptions about the behaviors of sites because we want to orchestrate sites which may have unpredictable delays, including infinite delays, i.e., failing to respond. This generality allows us to regard humans (and their communication devices) as sites and include them in orchestrations. An Orc program may act as the director of a coordinated activity, such as 9-11 dispatching, in which it instructs humans (police, medical personnel) and listens to their responses.

A site M can have multiple entry points, denoted by $M.n$ where n is the name of a method in the site.

2.3 Composition Operators

As we have described earlier, evaluation of an Orc expression calls some sites and *publishes* a set of values. In Section 2.2, we considered simple expressions like $CNN(d)$; evaluation of this expression calls site CNN and publishes the value, if any, returned by the site. In this section, we discuss the syntax and semantics of general Orc expressions in informal terms.

There are three composition operators in Orc to combine expressions. Symmetric composition of f and g, written as $f \mid g$, evaluates f and g independently. The sites called by f and g are the ones called by $f \mid g$ and a value published by either f or g is a value published by $f \mid g$. Expressions f and g are evaluated

Condition : *set*, *wait*
A condition allows multiple activities to wait until an event happens. Before *set* is called, all calls to *wait* block. When *set* is called, all waiting activities are enabled and future calls to *wait* return immediately.

Buffer : *put*, *get*
The result of *Buffer* is a local buffer site with two operations, *put* and *get*. The *put* operation adds values to the buffer and publishes a signal on completion. The *get* operation returns an item from the buffer – it blocks until an item is available.

Lock : *acquire*, *release*
A lock has exactly one owner. When the lock is created it is not owned. An expression that acquires the lock becomes its owner, and all subsequent calls to *acquire* will block until the owner calls *release*. At that point, one of the blocked expressions, if any, will be given ownership and unblocked.

Fig. 1. Definition of three factory sites used in the workflow implementations. Each factory site returns a local site that implements one or more methods. The method names are listed in italics after the factory name.

independently. There is no direct communication or interaction between these two computations; the computations may interact only by accessing a common site. For example, f may write into a cell by calling site *Write* and g may read that cell by calling *Read*.

In $f > x > g$, expression f is evaluated, each value published by it initiates a fresh evaluation of g as a separate computation, and the value published by f is called x in g's computation. Variable x may be a parameter in a site call in g. Evaluation of f continues while (possibly several) evaluations of g are run. This is the only mechanism in Orc similar to spawning threads. If f is *silent* (i.e. publishes no value), g is never evaluated. If f publishes a single value, there is strict sequencing in the evaluations of f and g. The values published by the executions of g are the values published by $f > x > g$. As an example, the following expressions calls sites CNN and BBC in parallel to get the news for date d. Any results from either of these sites are bound to x and then site *email* is called to send the information to address a.

$$(CNN(d) \mid BBC(d)) > x > email(a, x)$$

The expression $f \gg g$ is a short-hand for $f > x > g$ when the variable x is not needed.

To evaluate (g **where** $x :\in f$), start by evaluating both f and g in parallel. Evaluation of parts of g which do not depend on x can proceed, but site calls in which x is a parameter are suspended until it acquires a value. In (($M \mid N(x)$) **where** $x :\in R$), for example, evaluation M can proceed even before x has a value. If f publishes a value, then x is assigned this value, f's evaluation is terminated and the suspended parts of g can proceed. This is the only mechanism in Orc to block and terminate parts of a computation.

2.4 Definitions

Declaration $E(\bar{q}) \,\underline{\Delta}\, f$ defines expression E whose formal parameter list is $\bar{q}$ and body is expression f. A call $E(\bar{p})$ is evaluated by replacing the formal parameters $\bar{q}$ by the actual parameters $\bar{p}$ in the body of the definition f. Sites are called by value, while definitions are called by name.

2.5 Local Sites

A *local site* is a site that is created during execution of an expression. A local site is constructed by a *factory* site, which publishes a site when called. The factory sites used in the workflow implementations are defined in Fig. 1. The sites returned by the factory contain multiple methods. For example, the *Buffer* factory returns a site with *put* and *get* methods.

The following Orc expression illustrates the use of local sites. It creates a buffer, then executes three expressions in parallel, two of which insert numbers into the buffer while the other attempts to read from the buffer:

$$\mathit{Buffer} > b > (b.\mathit{put}(3) \mid b.\mathit{put}(5) \mid b.\mathit{get})$$

The value obtained by $b.get$ is either 3 or 5. Expression $b.get$ is blocked until one of the first two expressions is completed.

2.6 Synchronous Execution

We impose the following constraints on the Orc semantics: (1) a site is called as soon as possible, and (2) response from a site is processed only if there is no pending site call to be made. Therefore, initially, Orc calls all sites which can be called, and then it waits to receive a response. On receiving a response, it may publish some values and call some sites and waits for the next response. An expression publishes a (possibly empty) stream of values (position in the stream depends on the time of publication). The synchronous semantics ensures that in (g **where** $x :\in f$), the first value published by f is assigned to x.

3 Workflows in Orc

A workflow consists of a set of activities generating output in the form of data or events which may trigger further actions. These activities can be executed in sequential or parallel order. A workflow can be represented by a composition of elementary patterns as discussed in the subsequent sections. These patterns are modeled by composition of basic Orc expressions and Orc site calls. An Orc expression or site call may publish (produce) zero or more values as output.

We will use the workflow term "activity" to refer to an Orc expression that publishes at most one value and stops execution after this value is produced: an activity is complete when it publishes its value. Orc expressions that produce more then one value, or continue to call sites after producing a value, are not considered well-formed activities, but they can be converted into proper form by terminating them after the first value is produced.

Some patterns also use activities to signal *events*. In this case the event occurs when the activity publishes its value.

The following sections correspond to the patterns defined by van der Aalst [2]. We assume that f and g represent well-formed activities, unless stated otherwise.

WP 1: Sequence "An activity in a workflow process is enabled after the completion of another activity in the same process. Example: After the activity *order registration* the activity *customer notification* is executed."[11]

Sequential execution is a built-in feature of Orc.

$$\mathit{Seq}(f, g) \,\underline{\Delta}\, f \gg g$$

If f and g are activities, then the sequential composition is an activity. Note that if f is not an activity (i.e. it produces more than one value) g will be executed more than once.

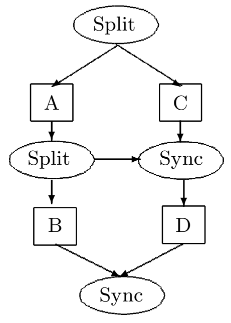
WP 2: Parallel Split "A point in the process where a single thread of control splits into multiple threads of control which can be executed in parallel, thus allowing activities to be executed simultaneously or in any order. Example: After activity *new cellphone subscription order* the activity *insert new subscription in Home Location Registry application* and *insert new subscription in Mobile answer application* are executed in parallel."[11]

The ability to run activities in parallel is an inherent feature of Orc.

$$\mathit{Par}(\bar{f}) \,\underline{\Delta}\, f_1 \mid \cdots \mid f_n$$

A bar over an expression $\bar{x}$ represents a list of items $x_1, \ldots, x_n$. The expression created by *Par* is not a well-formed activity, however, because it produces more than one value. The Discriminator pattern discussed in Section 3 can be used to model a well-formed activity by ensuring termination after the first value has been produced.

WP 3: Synchronization "A point in the process where multiple parallel branches converge into one single thread of control, thus synchronizing multiple threads. ... Example: Activity *archive* is executed after the completion of both activity *send tickets* and activity *receive payment*."[11]



(a) *Example from [11]*

$$\begin{array}{l} \mathit{Condition} >M> \\ \quad \mathit{Sync}(\,A \gg \mathit{Par}(M.\mathit{set}, B), \\ \qquad\quad \mathit{Sync}(C, M.\mathit{wait}) \gg D) \end{array}$$

(b) *Orc implementation of Fig. 2(a)*

$$\begin{array}{l} \mathit{Condition} >M> \\ \quad \mathit{Sync}(\,A \gg M.\mathit{set} \gg B, \\ \qquad\quad C \gg M.\mathit{wait} \gg D) \end{array}$$

(c) *Simplified form of Fig. 2(b)*

Fig. 2. Unstructured workflow example

Synchronization is a standard pattern in concurrent systems; its implementation in Orc was presented in [8].

$$\begin{array}{rcl} \mathit{Sync}(\bar{f}) & \triangleq & \mathit{let}(x_1) \gg \cdots \gg \mathit{let}(x_n) \\ & & \textbf{where } x_1 :\in f_1 \\ & & \quad \cdots \\ & & \textbf{where } x_n :\in f_n \end{array}$$

This expression uses asymmetric parallel composition to run the expressions f_i in parallel. The output of each expression is captured in a corresponding variable x_i, which is undefined until f_i publishes its value. The body of the **where** expression calls *let* on each variable: since site calls are strict, the sequence of calls will block until all the variables $\bar{x}$ are defined – that is, it will block until all the activities f_i are complete.

Synchronization of multiple activities is always a well-formed activity, even if f_i may produce more than one value. This is because *Sync* takes just the first value of each sub-expression and then terminates the sub-expression.

The previous example is a *structured* workflow, because the structure of synchronization matches the control flow structure: the expressions being synchronized are defined within the same composition operator. In an *unstructured* workflow, the expressions being synchronized appear in different places in the flow of control. Unstructured workflows are frequently more difficult to describe than structured workflows. Van der Aalst gives an example of an unstructured workflow, reproduced in Fig. 2(a), in which the synchronization path does not follow the structure of sub-expressions. This workflow cannot be expressed using only structured workflow constructs. In Orc, it requires a local site to express the communication between parallel branches, as defined in Fig. 2(b). The expression first creates a *Condition*, a local site defined in Section 2.5. The first *Sync*

expression represents the Split/Sync nodes at the top and bottom of Fig. 2(a). This is a structured synchronization. The left path A/Split/B is implemented by $A \gg (M.\mathit{set} \mid B)$, which executes A and then sets the condition to true and executes B. The right path C/Sync/D is implemented by $\mathit{Sync}(C, M.\mathit{wait}) \gg D$, which uses *Sync* to wait for C to complete and the condition to be set. When these two events have been synchronized, D is executed.

The expression in Fig. 2(b) corresponds closely to the diagram in Fig. 2(a), but it can be simplified to a more readable from in Fig. 2(c). This simplification replaces parallel execution with sequential execution. But the overall effect is the same if *set* and *wait* are instantaneous: instead of executing them in parallel with B or C, they can simply executed sequentially (before B and after C, respectively). Such transformations can be obtained through algebraic manipulation of Orc expressions.

WP 4: Exclusive Choice "A point in the process where, based on a decision or workflow control data, one of several branches is chosen. Example: The manager is informed if an order exceeds $600, otherwise not."[11]

An exclusive choice is simply a conditional, or "if" statement.

$$\mathit{XOR}(b, f, g) \;\triangleq\; \mathit{if}(b) \gg f \mid \mathit{if}(\neg b) \gg g$$

The built-in *if* site (see Table 1) does not publish a value when the condition is false, so only one of the two parallel alternatives will execute. Exclusive choice, like other patterns above, naturally generalizes to a choice between a set of options, also known as a case statement. Nested conditional constructs can also be represented using *XOR*. An example is given below.

$$\mathit{XOR}(b_1, f, \mathit{XOR}(b_2, g, \mathit{XOR}(b_3, h, i)))$$

WP 5: Simple Merge A *merge* is "a point in the workflow process where two or more alternative branches come together ... Example: After the payment is received or the credit is granted the car is delivered to the customer."[2] A simple merge assumes that only *one* of the expressions being merged is executing, so synchronization is not needed. Petri nets represent merges explicitly, while in Orc a merge is implicit in the structure of an expression. In the following, we assume that only of the f_i expressions will produce a value.

$$\mathit{Merge}(\bar{f}, h) \;\triangleq\; \mathit{Par}(\bar{f}) \gg h$$

WP 6: Multi-Choice "A point in the process, where, based on a decision or control data, a number of branches are chosen and executed as parallel threads. Example: After executing the activity *evaluate damage* the activity *contact fire department* or the activity *contact insurance company* is executed. At least one of these activities is executed. However, it is also possible that both need to be executed."[11]

A Multi-Choice is a non-exclusive choice. A separate condition controls the execution of each choice, and multiple conditions can be true.

$$\begin{aligned} \mathit{MultiChoice}(\bar{b}, \bar{f}) &\triangleq \mathit{IfDo}(b_1, f_1) \mid \cdots \mid \mathit{IfDo}(b_n, f_n) \\ \mathit{IfDo}(b, f) &\triangleq \mathit{if}(b) \gg f \end{aligned}$$

WP 7: Synchronizing Merge "A point in the process where multiple paths converge into one single thread. Some of these paths are active (i.e. they are being executed) and some are not. If only one path is active, the activity after the merge is triggered as soon as this path completes. If more than one path is active, synchronization of all active paths needs to take place before the next activity is triggered. ... Example: After either or both of the activities *contact fire department* and *contact insurance company* have been completed (depending on whether they were executed at all), the activity *submit report* needs to be performed (exactly once)."[11]

This pattern is implemented in Orc by modifying the *IfDo* expression to always publish a signal when it completes, even if the condition is false. The resulting conditional activities can then be synchronized.

$$\begin{aligned} \mathit{SyncMerge}(\bar{b}, \bar{f}) &\triangleq \mathit{Sync}(\mathit{IfSignal}(b_1, f_1), \cdots, \mathit{IfSignal}(b_n, f_n)) \\ \mathit{IfSignal}(b, f) &\triangleq \mathit{if}(b) \gg f \mid \mathit{if}(\neg b) \end{aligned}$$

Van der Aalst creates an unstructured example of synchronizing merge by replacing the Split at the top of Fig. 2(a) with a Multi-Choice, and the two Synchronize nodes with Synchronizing Merges. He says "then the process must somehow keep track of the activation of the left thread in order to determine whether activity D should be activated immediately after activity C completes, or whether it should also wait for activity A to complete."[11] Assuming that the left and right conditions for the Multi-Choice are α and β respectively, the resulting workflow can be expressed by making appropriate changes to workflow can be encoded easily in Orc:

$$\begin{aligned} &\mathit{Condition} > M > \mathit{Sync}(\, \mathit{XOR}(\alpha, A \gg M.\mathit{set} \gg B, \mathit{Signal} \mid M.\mathit{set}), \\ &\qquad\qquad\qquad\qquad \mathit{IfSignal}(\beta, C \gg M.\mathit{wait} \gg D)) \end{aligned}$$

The call to XOR (see WP 4) either executes the left path $A \gg M.\mathit{set} \gg B$ or else sets the condition M and signals completion, so that the condition is set in both alternatives. The right-hand path is the same as in Fig. 2(c) with the addition of the *IfSignal* (see WP 7) condition for β. This expression cannot be written using *SyncMerge* because of the additional call to $M.\mathit{set}$ when α is false. Orc cannot fully encapsulate this pattern as a definition. Some mechanism would be needed to track the collection of active synchronization variables, so that they can be *set* in the false branch of conditionals.

WP 8: Multi-Merge A multi-merge allows multiple branches to converge without synchronization. "If more than one branch gets activated, possibly concurrently, the activity following the merge is started for every activation of every

incoming branch."[11] The sequential composition operator in Orc supports this behavior directly.

$$\begin{aligned} \mathit{MultMerge}(\bar{f}, h) &\triangleq (f_1 \mid \cdots \mid f_n) \gg h \\ &\equiv \mathit{Par}(\bar{f}) \gg h \end{aligned}$$

WP 9: Discriminator "A point in the workflow process that waits for one of the incoming branches to complete before activating the subsequent activity. From that moment on it waits for all remaining branches to complete and ignores them. ... Example: To improve query response time a complex search is sent to two different databases over the Internet. The first one that comes up with the result should proceed the flow."[11]

A discriminator returns the first value produced by a set of expressions but allows the remaining expressions to continue executing. To implement this behavior, Orc uses a local channel S created by a *Buffer* site.

$$\mathit{Discr}(\bar{f}) \triangleq \mathit{Buffer} > S > (\mathit{Par}(\bar{f}) > x > S.\mathit{put}(x) \mid S.\mathit{get})$$

The discriminator publishes only the first value that is placed in the buffer by $\bar{f}$, but allows $\bar{f}$ to continue running.

When applied to any expression $\bar{f}$, *First* terminates the computation of $\bar{f}$ after its first value is produced:

$$\mathit{First}(\bar{f}) \triangleq \mathit{let}(x) \ \mathbf{where} \ x :\in \bar{f}$$

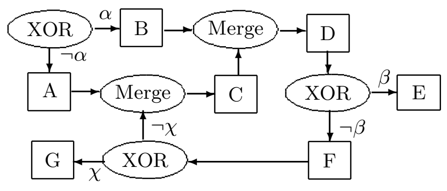
First can be used to ensure termination of any expression f after it has produced its first value.

Van der Aalst uses discriminator to create another variation on the unstructured workflow in Fig. 2(a), by replacing the synchronize node between C and D with a discriminator. This means that D can start as soon as A or C completes. This example is easily defined in Orc, by simply replacing the corresponding *Sync* with *Discr*:

$$\begin{aligned} &\mathit{Condition} > M > \mathit{Sync}(\, A \gg M.\mathit{set} \gg B, \\ &\qquad\qquad\qquad\qquad \mathit{Discr}(C, M.\mathit{wait}) \gg D) \end{aligned}$$

WP 10: Arbitrary Cycles Workflows with arbitrary cycles and loops are easily created in Orc using recursive definitions. Fig. 3(a) gives a workflow from van der Aalst [2]. The diagram is a Petri Net, which can be understood as a form of flowchart. An XOR node is an exclusive choice in which the outgoing branches are labeled by a condition. A Merge node is a simple merge (WP 5).

Fig. 3(b) is an implementation of this flowchart in Orc. Each node is translated to a definition. An arc to a node in the flowchart is translated to a call to the corresponding definition. These expressions are equivalent to loops, because Orc is defined to optimize tail calls. Note that the Merge nodes are modelled implicitly.



(a) *Fig. 6 of [2]*

$$
\begin{aligned}
P &\triangleq XOR(\alpha, PB, PA)\\
PA &\triangleq A \gg PC\\
PB &\triangleq B \gg PD\\
PC &\triangleq C \gg PD\\
PD &\triangleq D \gg XOR(\beta, E, PF)\\
PF &\triangleq F \gg XOR(\chi, G, PC)
\end{aligned}
$$

(b) *Orc implementation of (a)*

Fig. 3. Arbitrary cycles example

Arbitrary cycles can be difficult to model when computations can only be structured as iterations with one entry and exit point [7]. Although Orc is highly structured, this example illustrates the use of recursion to define loops, which do not suffer from the problems of structured iteration.

Simple while loops can also be easily created. In the following definition, g publishes a boolean that controls execution of the loop. The call to *IfSignal* (see WP 7) evaluates $f \gg Loop(g, f)$ if b is true, and produces a signal otherwise.

$$Loop(g, f) \triangleq g > b > IfSignal(b, f \gg Loop(g, f))$$

WP 11: Implicit Termination "A given subprocess should be terminated when there is nothing else to be done. In other words, there are no active activities in the workflow and no other activity can be made active (and at the same time the workflow is not in deadlock)."[2]

Implicit termination simply means that an expression continues running as long as there is more work to do, and that no explicit "stop" action is required. Since there is no explicit stop action in Orc, it supports implicit termination.

WP 12-15: Multiple Instances There are three patterns covering creation of multiple instances of a workflow, one *without* synchronization, and two more with and without a priori *design time* knowledge.

The use of "process instance" in van der Aalst's patterns is probably influenced by his work on Petri nets: since Petri nets are (traditionally) understood as a form of finite state machine, they do not have the concept of block structure and instantiation as in process calculi like CCS, π-calculus and Orc.

Multiple threads are created using parallel composition *Par* (WP 12). If the list of instances is known at design time, then they can be synchronized by using *Sync* instead of *Par* (WP 13). For WP 14, the number of instances is known as a runtime quantity before the instances are created. We represent this runtime knowledge as a list in Orc, using a notation borrowed from Haskell [3]. An activity is started for each item in the list, and all the activities are synchronized using *Sync*.

$$
\begin{aligned}
SyncList(F, []) &\triangleq Signal\\
SyncList(F, a : as) &\triangleq Sync(F(a), SyncList(F, as))
\end{aligned}
$$

Finally, WP 15 allows creation of instances where the number of instances is not known in advance: more instances may be created until some condition is satisfied. One implementation is a synchronized form of while loop. *ParLoop* is the same as *Loop* (see WP 10) except that iterations of the loop are performed in parallel and synchronized.

$$ParLoop(g, f) \triangleq g > b > IfSignal(b, Sync(f, ParLoop(g, f)))$$

WP 16: Deferred Choice "A point in a process where one among several alternative branches is chosen based on information which is not necessarily available when this point is reached. This differs from the normal exclusive choice, in that the choice is not made immediately when the point is reached, but instead several alternatives are offered, and the choice between them is delayed until the occurrence of some event. Example: When a contract is finalized, it has to be reviewed and signed either by the director or by the operations manager, whoever is available first. Both the director and the operations manager would be notified that the contract is to be reviewed: the first one who is available will proceed with the review."[11]

Deferred choice happens when a set of *events* is used to select an alternative: the first event that fires causes its corresponding action to be activated. Deferred choice is called *arbitration* in [8]. Assume that the events are specified by a set of Orc expressions $\bar{e}$ and that the actions are defined by the Orc expressions $\bar{f}$. Note that, in Orc, the firing of an event is represented in terms of a site call to the environment. This enables the environment to participate in making a choice.

In the following definitions, *Which* produces an index identifying which event signalled; the call to *First* terminates the remaining events. The *Select* expression then runs the selected action.

$$
\begin{aligned}
DefChoiceTerm(\bar{e}, \bar{f}) &\triangleq Which(\bar{e}) > k > Select(k, \bar{f})\\
Which(\bar{e}) &\triangleq First(e_1 \gg let(1) \mid \cdots \mid e_n \gg let(n))\\
Select(k, \bar{f}) &\triangleq if(k = 1) \gg f_1 \mid \cdots \mid if(k = n) \gg f_n
\end{aligned}
$$

WP 17: Interleaved Parallel Routing "A set of activities is executed in an arbitrary order: Each activity in the set is executed, the order is decided at run-time, and no two activities are executed at the same moment (i.e. no two activities are active for the same workflow instance at the same time)."[2]

This pattern is essentially an example of mutual exclusion between concurrent processes.

$$\begin{aligned} \mathit{Interleave}(\bar{f}) &\triangleq \mathit{Lock} >M> (\mathit{wait}(M, f_1) \mid \cdots \mid \mathit{wait}(M, f_n)) \\ \mathit{wait}(M, f) &\triangleq M.\mathit{acquire} \gg f >x> M.\mathit{release} \gg \mathit{let}(x) \end{aligned}$$

WP 18: Milestone "A given activity can only be enabled if a certain milestone has been reached which has not yet expired. ... Example: After having placed a purchase order, a customer can withdraw it at any time before the shipping takes place."[2]

Consider three Orc activities f, g, and e. The completion of activity f enables g. Let e be an event that is raised when g is no longer allowed to run. Thus f precedes g and e, while e can interrupt g.

$$\begin{aligned} \mathit{Milestone}(f, g, e) &\triangleq f \gg \mathit{Interrupt}(g, e) \\ \mathit{Interrupt}(g, e) &\triangleq \mathit{First}(g \mid e) \end{aligned}$$

This simple definition does not fully express the intent of the pattern: the intent is for f and e to be part of one workflow, while g is a part of another workflow. The workflows should communicate through channels, not be defined in a single expression. An improved definition uses two conditions, S and E. The S condition signals the start of the milestone, while E signals the end of the milestone.

$$\begin{aligned} \mathit{Notify}(f, S, E, e) &\triangleq f \gg S.\mathit{set} \gg e \gg E.\mathit{set} \\ \mathit{Listener}(S, E, g) &\triangleq S.\mathit{wait} \gg \mathit{Interrupt}(g, E.\mathit{wait}) \\ \mathit{Milestone}(f, e, g) &\triangleq \mathit{Condition} >S> \mathit{Condition} >E> \\ &\qquad (\mathit{Notify}(f, S, E, e) \mid \mathit{Listener}(S, E, g)) \end{aligned}$$

Van der Aalst also considers the case where g may be repeated arbitrarily after f and before e: this is done by replacing g by $\mathit{Loop}(\mathit{true}, g)$.

WP 19/20: Cancel Activity/Case Cancelling can apply to an activity that is part of a workflow an entire workflow case. The *Interrupt* operator can be applied to a part of a workflow or the entire workflow to cancel part or all of the activity. This will cancel any activity immediately. *Interrupt* and *Condition* can be used in conjunction to model a set of cancellable activities.

4 Related Work

Orc implementations of the workflow patterns are most similar to BPML [11] and π-calculus. However, BPML is much more verbose than Orc. The mechanism for creating reusable definitions is also more cumbersome. There does not seem

to be a mechanism analogous to local sites, so Interleaved Parallel Routing (WP 17) does not have a clean solution.

The π-calculus [9] versions of the workflow patterns are similar in structure to the Orc implementations. One significant difference is that π-calculus uses channels for all communication and synchronization. Orc expressions, on the other hand, embody structured forms of communication and control, so explicit channels (local sites) are needed only for unstructured workflows. The π-calculus explanation of the cancellation pattern is incomplete, because π-calculus does not provide built-in support for terminating a process, and the proposed encoding of a "global cancel trigger" is left undefined. The synchronizing merge pattern also does not specify how it is determined which processes are active.

Van der Aalst et al. defined a new workflow language, YAWL[10, 12], specifically to support the workflow patterns. The language is based on Petri nets, but is extended with special constructs for creating multiple instances and cancelling tokens in a group of nodes. The mechanism for multiple instantiation is analogous to Orc's sequential composition, but provides built-in synchronization. The node grouping and cancellation construct is similar to Orc's **where** operator. Rather than build specific workflow patterns into the language, Orc provides few fundamental primitives with a mechanism to define new operators for user-defined composition patterns.

5 Conclusion

We have implemented a set of standard workflow patterns using Orc, a new orchestration language. The solutions are generally easy to read and understand.

There is no reason to assume that the workflow patterns proposed by van der Aalst are complete. Orc has already been used to implement common concurrency patterns, like Priority and Timeout [8]. The Implicit Termination pattern suggests a need for an Explicit Termination pattern, in which an activity explicitly signals when it is complete. This pattern can also be implemented in Orc, although the machinery to do so is somewhat more complex.

One novelty of our approach is the encapsulation of pattern as *reusable definitions*. These definitions can be used to create larger programs; this technique is illustrated many times in this paper. Van der Aalst argues that all the workflow patterns should be expressed directly in the workflow language *without any encodings*. In his summary of the workflow patterns supported by various commercial systems, a pattern is marked as "not supported" if any form of encoding is required. In Orc some of the patterns require a combination of operators to implement – however, the pattern itself can be expressed as a *definition*, which can be reused whenever that pattern is required. Thus the pattern becomes another composition operator that can be used in larger programs. The operators that define patterns are reused extensively in this paper. This demonstrates the power of a language that can grow by adding new definitions, rather than requiring building a fixed set of primitives that cannot be easily extended by new definitions.

References

1. W. Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
2. W. M. P. V. D. Aalst, A. H. M. T. Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.
3. R. Bird. *Introduction to Functional Programming using Haskell*. International Series in Computer Science, C.A.R. Hoare and Richard Bird, Series Editors. Prentice-Hall International, 1998.
4. M. Dumas and A. H. ter Hofstede. UML Activity Diagrams as a Workflow Specification Language. Technical report, Cooperative Information Systems Research Centre, Queensland University of Technology GPO Box 2434, Brisbane QLD 4001, Australia, Nov. 2003.
5. R. Eshuis and J. Dehnert. Reactive petri nets for workflow modeling. In W. M. P. van der Aalst and E. Best, editors, *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets (ICATPN 2003)*, volume 2679 of *Lecture Notes in Computer Science*, pages 296–315. Springer-Verlag, June 2003.
6. R. Eshuis and R. Wieringa. Comparing petri net and activity diagram variants for workflow modelling - a quest for reactive petri nets. In H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors, *Petri Net Technology for Communication-Based Systems*, volume 2472 of *Lecture Notes in Computer Science*, pages 321–351. Springer-Verlag, November 2003.
7. B. Kiepuszewski, A. H. M. ter Hofstede, and C. Bussler. On structured workflow modelling. In *Conference on Advanced Information Systems Engineering*, pages 431–445, 2000.
8. J. Misra and W. R. Cook. Computation orchestration: A basis for wide-area computing. To appear in the Journal of Software & Systems Modeling, 2006.
9. F. Puhlmann and M. Weske. Using the π-calculus for formalizing workflow patterns. In *Proceedings of the 3rd International Conference on Business Process Management*, volume 3649 of *Lecture Notes in Computer Science*, 2005.
10. W. van der Aalst and A. ter Hofstede. YAWL: Yet Another Workflow Language. Technical report, Department of Technology Management, Eindhoven University of Technology P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands, Nov. 2003.
11. W. M. van der Aalst, M. Dumas, A. H. ter Hofstede, and P. Wohed. Pattern Based Analysis of BPML (and WSCI). Technical report, Department of Technology Management Eindhoven, University of Technology, The Netherlands, nov 2003.
12. W. M. P. van der Aalst, L. Aldred, M. Dumas, and A. H. M. ter Hofstede. Design and implementation of the yawl system. In A. Persson and J. Stirna, editors, *CAiSE*, volume 3084 of *Lecture Notes in Computer Science*, pages 142–159. Springer, 2004.
13. P. Wohed, W. M. van der Aalst, M. Dumas, and A. H. ter Hofstede. Pattern based analysis of BPEL4WS. Technical Report FIT-TR-2002-04, Queensland University of Technology, 2002.
14. The Workflow Reference Model. The Workflow Management Coalition, Jan. 1995.