Workflow Enactment with Continuation and Future Objects

Dragos A. Manolescu
dam@micro-workflow.com

ABSTRACT

An increasing number of software developers are turning to workflow to separate the logic and the control aspects in their applications, thus making them more amenable to change. However, in spite of recent efforts to standardize and provide reusable workflow components, many developers build their own. This is a challenging endeavor and involves solving problems which seem incompatible with the object paradigm and current object-oriented programming languages. In the context of an object-oriented workflow framework, this paper demonstrates a novel approach that resolves this impedance mismatch with techniques drawn from programming language theory. This successful cross-pollination narrows the gap between the results of decades of research in programming languages and developers working hard to cope with change.

Categories and Subject Descriptors

D.1.5 [**Software**]: Programming Techniques—*Object-oriented Programming*; D.3.3 [**Software**]: Language Constructs and Features—*Classes and objects, Frameworks*

General Terms

Design, Languages

Keywords

Workflow, continuations, trampolined style, future objects, micro-workflow.

1. INTRODUCTION

Business changes. Software must keep up with change at an affordable cost. Programmers are currently tackling this challenge on several fronts. From a methodology perspective, they are moving towards lightweight, agile methodologies. From a programming perspective, they are embracing new styles that facilitate change. This latter trend has increased (among other things) the importance of process support within applications in general, and of the supporting technology (i.e., workflow) in particular. The recent adoption of a workflow management facility by the Object Management Group (OMG) shows a strong interest in workflow technology from the object community [34].

This paper describes in detail some of the most interesting aspects of an object-oriented workflow framework written in VisualWorks Smalltalk. However, as I will discuss later, the techniques explained here (originally developed in Lisp and Scheme) are applicable to an increasing number of languages. Therefore, the contributions of this paper are relevant outside the Smalltalk community. The paper has two goals. First, it shows one way of building basic workflow support with objects. At first sight object and workflow technologies seem incompatible: the former *deemphasizes the flow of control*, while the latter *explicitly represents it*. Likewise, while most object-oriented languages implement *synchronous message sends*, workflow involves *asynchronous processing*. This paper provides a solution to this impedance mismatch. Second, the paper demonstrates that concepts from programming languages (e.g., continuations, trampolined style, and futures) provide viable solutions in the context of workflow management.

The paper has two parts. The first part begins with a discussion of workflow in the context of object-oriented software development, and then introduces micro-workflow, a workflow architecture aimed at developers. The second part focuses on the design and implementation of the micro-workflow components providing basic workflow support. Using these components as a vehicle I introduce techniques from programming language theory to workflow management.

The above structure targets two audiences. The first part of the paper aims at developers who are building applications that must accommodate business changes. Developers not familiar with workflow will learn what workflow is and how they could use it to facilitate change in their applications. Likewise, developers already familiar with workflow will learn how to implement a lightweight workflow core for use within applications. The second half of the paper aims at object-oriented developers who are interested in techniques applicable to workflow, but that might appear in other domains. The workflow architecture presented in the first part provides the context for introducing them.

2. WHAT IS WORKFLOW?

Despite being around since the 1970s, workflow technology is still a fuzzy area to many people. Some wrongfully regard it as a novel idea, while others have very different interpretations of what workflow means.

During the past few years a common workflow terminology and standards have been on the agenda of several organizations, including the Workflow Management Coalition (WfMC) and the OMG. In the context of this paper, workflow coordinates activities per-

OOPSLA'02, November 4-8, 2002, Seattle, Washington, USA.

formed by various *participants*[1] towards a business goal—this view is similar to WfMC's definition of workflow. Coordination involves activity ordering and the interdependencies between them, synchronization with external events, and delegation to *participants*. Examples include the billing process within a telecommunication system, or the followup process for newborns with certain predispositions.

Let's consider a short example that corresponds to a simplified workflow from the medical world—the treatment of strep throat. The strep throat workflow begins with a patient who suspects that she may have strep and goes to the doctor to seek medical attention. The doctor examines the patient and tests whether she has strep throat. If the results are positive, the doctor prescribes a treatment. Based on the patient's medical records, the doctor can treat strep throat in two different ways. If the patient is not allergic to penicillin (an antibiotic), the doctor prescribes this treatment. Otherwise, he prescribes the sulfa drug. Next a nurse takes the prescription and instructs the patient how to follow the treatment. If the prescription contains penicillin, she also warns the patient about the possibility of an allergic reaction to antibiotics. The patient goes home and starts taking the pills. Two days after the beginning of the treatment the nurse checks with the patient to see whether there have been any improvements. She also reminds the patient to continue taking the pills even if her condition has improved. At the end of the treatment, the nurse checks the state of the patient again.

This workflow involves two participants, the doctor and the nurse. They provide medical knowledge and expertise. The doctor *examines* the patient and *prescribes* the treatment. Likewise, the nurse *administers* the prescription and *checks* the patient's condition.

But is this really that different from what happens in an object system, where programs consist of objects which perform computations in response to messages? Here are some of the characteristics that set workflow management apart:

- Workflow involves long-running, slow business processes.[2] Mortgages, for example, typically last for 15 or 30 years.
- The workflow participants carrying out the activities can be software (e.g., objects, components, applications) as well as people (as in the strep throat example). Some of these activities execute asynchronously.
- Workflow users may want to take control over activity ordering of the process as it unfolds. For example, the strep throat patient may develop an allergic reaction, in which case the workflow doesn't proceed as above.
- The workflow system saves history information about the workflows it executes. People have many uses for this information, including auditing, process analysis and improvement. The workflow system may also use it for recovery.
- Workflow users can monitor the progress of the entire process. Knowing who is doing what at any moment helps with resource allocation.

3. WHY WORKFLOW?

Workflow is no longer just "some sort of planned document routing" [35]. Workflow technology and process support has shifted from end-user applications to a key ingredient of the "networked economy" [37]. Currently workflow lies at the center of enterprise application architectures. The two characteristics that make workflow a valuable technology for building agile applications are flow- and domain-independence.

3.1 Flow-Independence

Software developers have long recognized the benefits of separating different concerns. For example, data management and user interface represent two of the aspects that many applications have to deal with. Good developers aim at building software such that every design decision is encapsulated into a component. This lets them revisit individual decisions and make changes without affecting other parts of the application.

Over 20 years ago Kowalski argued that separating the *what*, which specifies the "knowledge to be used in solving problems," from the *how*, which determines "the problem solving strategies by means of which that knowledge is used," will make programs more readily adapted to new problems, thus improving modifiability [24]. However, most developers intermix these two aspects within their software because doing so is intuitive (requiring little analysis). The intertwined control and logic (referred to as flow-dependency in the context of workflow management [25]) becomes a hindrance when developers change one or the other.

Kowalski recommended that programming languages provide explicit support for the separation of the logic and the control aspects:

> Computer programs will be more often correct, more easily improved, and more readily adapted to new problems when programming languages separate logic and control. (R. Kowalski [24])

However, programming languages haven't really evolved in this direction. Although aspect-oriented programming (AOP) focuses on separating cross-cutting concerns (i.e., aspects) [22], aspect systems like Aspect-J don't regard the control flow as an AOP-style aspect. Software developers looking for ways to separate the flow between an application's objects/components from the application are turning to workflow products like for example Versata Process Logic Engine [41] or Drala Workflow Engine [9].

As databases and user interface frameworks remove data- and UI-dependencies from application code, workflow makes applications *flow-independent*. Software developers implement the control aspect of their applications with workflow technology, which removes the knowledge of activity sequencing and their interdependencies from application objects. In effect, changing the control no longer affects the logic (i.e., flow-independence). Additionally, application objects can become more reusable since they make fewer assumptions about the control context in which they operate.

The sketch from Figure 1 illustrates the key difference between flow-dependent and flow-independent applications. A flow-dependent application scatters the control among the objects implementing the logic. Changing one aspect typically impacts the other. In contrast, a flow-independent application makes a clear separation between its control and logic, organizing the workflow and application objects into two tiers. This separation of concerns facilitates changes in either tier and improves reuse. Workflow experts predict that the discovery of benefits of flow-independence will foster the use of workflow management systems for building flexible applications [25].

Additionally, workflow allows application developers to use workflow-specific features that otherwise would be too expensive to hand craft every time they build a new application. Some examples include the ability to take over the execution ordering at run time,

[1]Throughout this document, the terminology of the Workflow Management Coalition (WfMC) standards appears in *slanted* fonts.

[2]I use the term business process to avoid the confusion with operating system processes. Note, however, that workflow is applicable outside the business realm.

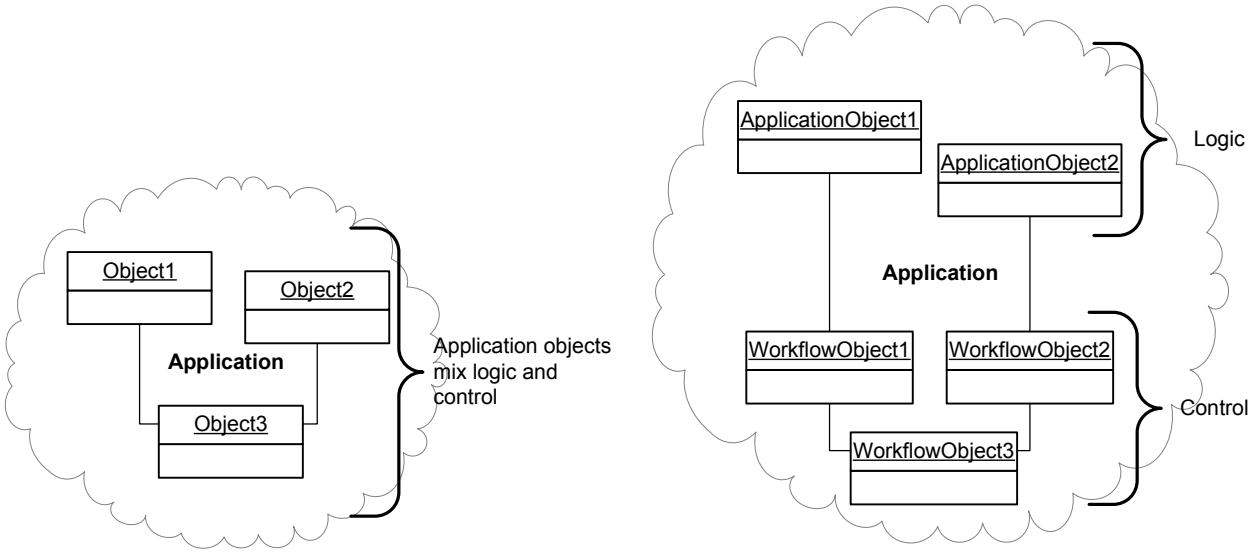

Figure 1: Flow-dependent (left) and flow-independent (right) applications.

or the ability to know the current state of the process (in workflow parlance these are known as manual intervention and monitoring).

3.2 Domain-Independence

The partitioning typical of flow-independent applications (Figure 1, right) keeps the workflow (along with the business process support that runs it) outside the application domain. Thus applying workflow to a particular application domain requires providing components that perform domain-specific work. This characteristic makes workflow technology applicable to a large number of application domains.

For example, Jackson [20] and Georgakopoulos and colleagues [15] discuss examples from the telecommunications industry. Dinkhoff and colleagues [7] apply workflow to administrative processes for property management. Vossen and Weske [42] use workflow technology for geoprocessing applications. McClatchey and colleagues [29] and Kováks [23] employ workflow in the context of the Compact Muon Solenoid high energy physics experiment. Yang and Papazoglou [43] identify workflow as part of the reference architecture for interoperable e-commerce applications. Leymann and Roller [25] discuss the application of workflow technology for software distribution, security management, and business-process-oriented systems management. The key point here is that a wide range of application domains can benefit from workflow technology.

4. THE MICRO-WORKFLOW ARCHITECTURE

To accommodate the requirements of object-oriented developers who need workflow functionality within their applications, a workflow system must [26]: (1) Allow developers to pick and choose the workflow features they need; (2) Let them customize existing features and add new ones through object and class composition; (3) Integrate with custom components, subsystems, and frameworks; and (4) Support incremental integration with existing systems and applications. However, most workflow products are incompatible with these requirements. Current workflow systems are heavyweight and package a comprehensive set of features in an all-or-nothing manner. The narrow purpose design of traditional workflow architectures limits their applicability to the types of applications for which they have been tailored. For example, Muth and colleagues [31] observe that "most workflow management systems, both products and research prototypes, are rather monolithic and aim at providing full fledged support for the widest possible application spectrum." Additionally, current workflow systems are hard to integrate with other environments.

To achieve the goals listed above I have designed micro-workflow, a new workflow architecture [26]. Micro-workflow bridges the gap between the workflow functionality object-oriented developers need in their applications and the functionality provided by traditional workflow management systems. The combination of microkernel and object-oriented architectural styles [5], and ideas from compositional software reuse solve many of the problems of traditional workflow architectures. The resulting architecture can be integrated within object-oriented applications, can be tailored to specific domains, and was designed from the ground up to accommodate *organic growth* [4].

At the focal point of the micro-workflow architecture, a lightweight *core* provides basic workflow functionality, allowing developers to define and execute workflows. *Additional components* implement advanced workflow features like monitoring, history, manual intervention, worklists, federated workflow, persistence, and so on. This represents a radical departure from the traditional workflow architectures. Figure 2 sketches the structure of the micro-workflow architecture.

As the topic of this paper lies on the border between workflow and programming languages, note that the key ideas of this approach (i.e., lightweight core and organic growth) are in line with recent research on workflow architectures [30, 16] and trends in programming language design:

> From now on, a main goal in designing a language should be to plan for growth. The language must start small, and the language must grow as the set of users grows. (G. Steele [38])

I have implemented the micro-workflow architecture as an object-oriented framework in VisualWorks Smalltalk [6], with GemStone/S [14] (an object-oriented persistent store) and OpenTalk (a distributed application architecture) for persistence and distribution [26]. The

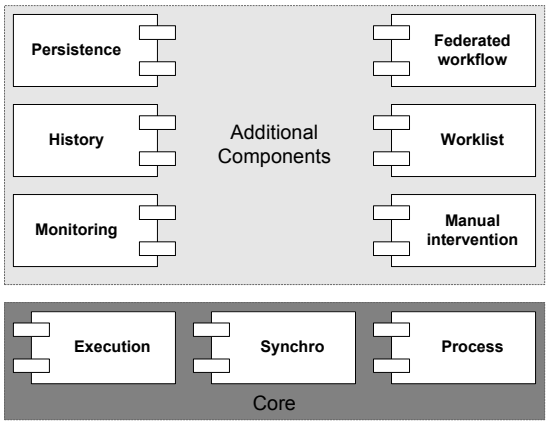

Figure 2: The micro-workflow architecture

following sections focus on the core of the micro-workflow framework, particularly on the process and execution components (see Figure 2). The components for advanced workflow features are beyond the scope of this paper.

5. THE DESIGN AND IMPLEMENTATION OF THE MICRO-WORKFLOW CORE

The decomposition into classes typical of object-oriented applications deemphasizes the control flow and distributes it among different objects. Thus, the global control flow and behavior are less visible than in procedural programs. Therefore, one of the challenges of building an object-oriented workflow system lies in providing abstractions that maintain an explicit representation of the control flow without violating the principles of good object-oriented design.

One way to implement a workflow system would be to add workflow-specific features to a general-purpose programming language. Languages with metaobject protocols like CLOS and Smalltalk are amenable to domain-specific extensions [21]. However, I have adopted a different approach. Instead of changing an existing language (and thus creating a dialect), micro-workflow lies above the implementing language (Smalltalk). Although this approach has several liabilities (Noble has analyzed its tradeoffs in a different context [33]), here its benefits outweigh them. First, the implemented system (i.e., workflow) is under programmers' control. Second, it fosters integrability with existing systems and applications (one of the initial goals). Finally, it yields language portability, thus making the contributions of this paper relevant to object-oriented developers regardless of the language and environment they use.

5.1 The Process Model

At the core of every workflow system, a process model provides key process abstractions and their relationships. Workflow users define workflows with these abstractions.

Most workflow systems use one of the three types of process models: activity-, artifact-, or communication-based; a combination of these (i.e., a hybrid model) is also possible. The models focus on different aspects:

- The *activity-based process model* provides key abstractions that capture how to coordinate the process activities. It shows which activities execute as the workflow unfolds in time.
- The *artifact-based process model* focuses on the artifacts that are created, modified or used by the workflow. The focus is on data flow, and the control flow emerges as the process generates new artifacts.
- Finally, the *communication-based process model* is centered around agreements between participants. This model focuses on how the participants fulfill their commitments and advance the process.

The explicit representation of control flow makes an activity-based process model appropriate for separating the control from the logic. Micro-workflow uses this process model. The primary concept of the activity-based process model is a *task*, which is a multi-entity collaborative activity. A task consists of *activities*, each of which is an atomic unit of work performed by a workflow *participant*.

The activity-based model represents workflows with directed graphs called *activity networks*. One way to organize the networks is to place activities in the network nodes with the data passed between activities on the arcs connecting these nodes. Figure 3 shows a workflow definition consisting of 6 activity nodes. Although conceptually some activity nodes use application objects, these are not part of the definition.

Workflow enactment corresponds to workflow execution, i.e., at run time the workflow system instantiates a workflow definition, creating a *workflow case*. For an activity-based process model, the workflow enactment mechanism navigates the activity graph passing the control flow to nodes according to the workflow definition. For example, a sequence node passes the control to its subtrees sequentially, in a predefined order (e.g., from left to right). Likewise, a conditional node passes the control to its unique subtree only if the condition associated with it is satisfied. Thus, navigation depends on the types of activity nodes, and the workflow data. The navigation begins in the root node and continues until the enactment mechanism completes the workflow, or until the user aborts execution. For example, the execution of the workflow depicted in Figure 3 begins with the node labeled `Sequence1` and ends with the node labeled `Primitive4`.

The micro-workflow framework represents each node of the workflow definition with an activity object. Several types of activity objects provide a basic set of control structures—primitive, sequence, conditional, repetition, etc. For example, the workflow from Figure 3 consists of 4 primitives and 2 sequences. As typical of white-box object-oriented frameworks, developers add new structures through inheritance [36].

5.2 Workflow Enactment with Message Sends

A simple and intuitive way to implement workflow navigation is through message sends between activity objects [27]. In the example from Figure 3, `Sequence1` starts execution by sending the executeActivity message to its first step, `Sequence2`. In turn, this executes its first step `Primitive1` by sending it the executeActivity message, and so on.

This implementation translates workflow enactment into a chain of executeActivity messages. Each message must preserve the current control context; upon its return, program execution should continue from the point following the message send. Therefore message sends incur an accumulation of control context until they return. The diagram from Figure 4 shows the sequence of messages corresponding to the enactment of the workflow example from Figure 3. The control context corresponding to each executeActivity message (left to right solid arrow) must be preserved until the message returns (right to left dashed arrow). The control context grows with the number of chained messages, as the navigation moves from the initial executeActivity message.

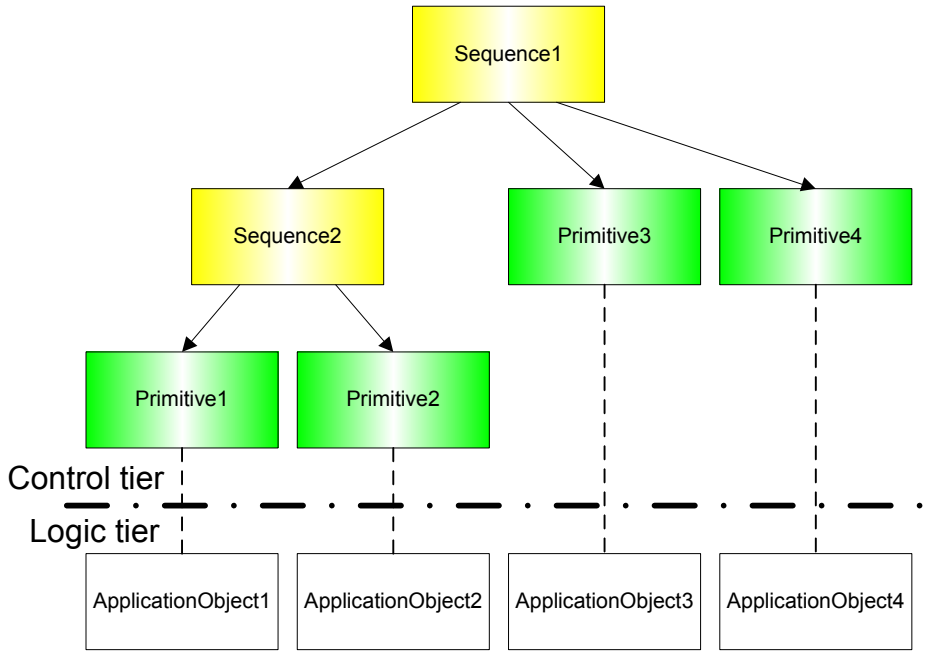

Figure 3: Activity network; the sequence nodes execute their children left-to-right. The application objects from the logic tier are not part of the workflow definition.

Programming languages (Smalltalk for the implementation discussed here) typically preserve the control context on the call stack. The context is pushed on the stack just before the control passes to the receiver of the message, and popped once the message completes execution. In the example from Figure 3 this means that the executeActivity message sent to `Sequence1` doesn't return until its three steps complete execution. Likewise, the executeActivity message that `Sequence1` sends to its first step `Sequence2` doesn't return until the latter's two steps complete execution. In other words, this implementation fires off the enactment mechanism with a message send that *doesn't return until the workflow completes execution*. The stack of the implementing language holds the control context throughout workflow execution.

In spite of chained messages being quite common in object systems, the enactment mechanism presented here has problems in the context of workflow. The problems stem from the tight coupling between the implemented system (i.e., workflow) and the implementing language. Most programming languages provide only indirect access to the call stack. In other words, programmers can't extract or manipulate the control context associated with the workflow. This limitation hinders implementing workflow features like recovery and manual changes:

- If the computer executing the workflow crashes, the call stack is gone unless you can save it. This requires the ability to extract the workflow control context from the programming language call stack.
- Sometimes workflow users want to take over activity ordering, overriding the workflow definition. This requires the ability to manipulate the call records on the stack.

The solution lies in separation, a widely-used design operation [3]. This involves building an enactment mechanism that no longer uses the call stack to save the workflow control context; in other words, it factors the management of workflow control context out of the implementing system's call mechanism. A separate workflow stack can provide this separation; I discussed this solution elsewhere [26]. The next section presents an alternative without an explicit workflow stack.

5.3 Workflow Enactment with Continuation Objects

The chained message sends of the solution presented in the previous section are responsible for the accumulation of control context. The enactment mechanism presented in this section breaks the chained message sends with continuation objects. As I will discuss later, the solution *uses continuation objects without language support for continuations*, and thus is language-neutral.

A continuation represents work that has to be done. Consider the strep throat treatment example from Section 2. The first activity of the workflow involves the doctor, who examines the patient and tests whether she has strep throat. A continuation corresponding to this step is a function that takes one argument and carries out the remainder of the process—prescribing the treatment depending on the test results and on the patient's allergies; following up with the patient; and closing the case. The argument of this function would be the outcome of the first step—in this case, the test results. Language designers have used continuations to implement interpreters, backtracking, multi-threading, exceptions, compilation, and optimization [39, 11, 2]. Here continuations eliminate the growth of control context associated with cascaded message sends, and lay the groundwork for implementing dynamic changes.

The continuation-based enactment mechanism executes the workflow in discrete steps, each of which corresponds to an activity object. Instead of executing an action, activity objects build and return continuation objects. These objects represent a reification of the processing that should be carried out in each activity node. For example, a sequence activity no longer responds to an execute-like message by executing its steps. Instead, it creates a continuation object that will carry out the sequential execution of each step

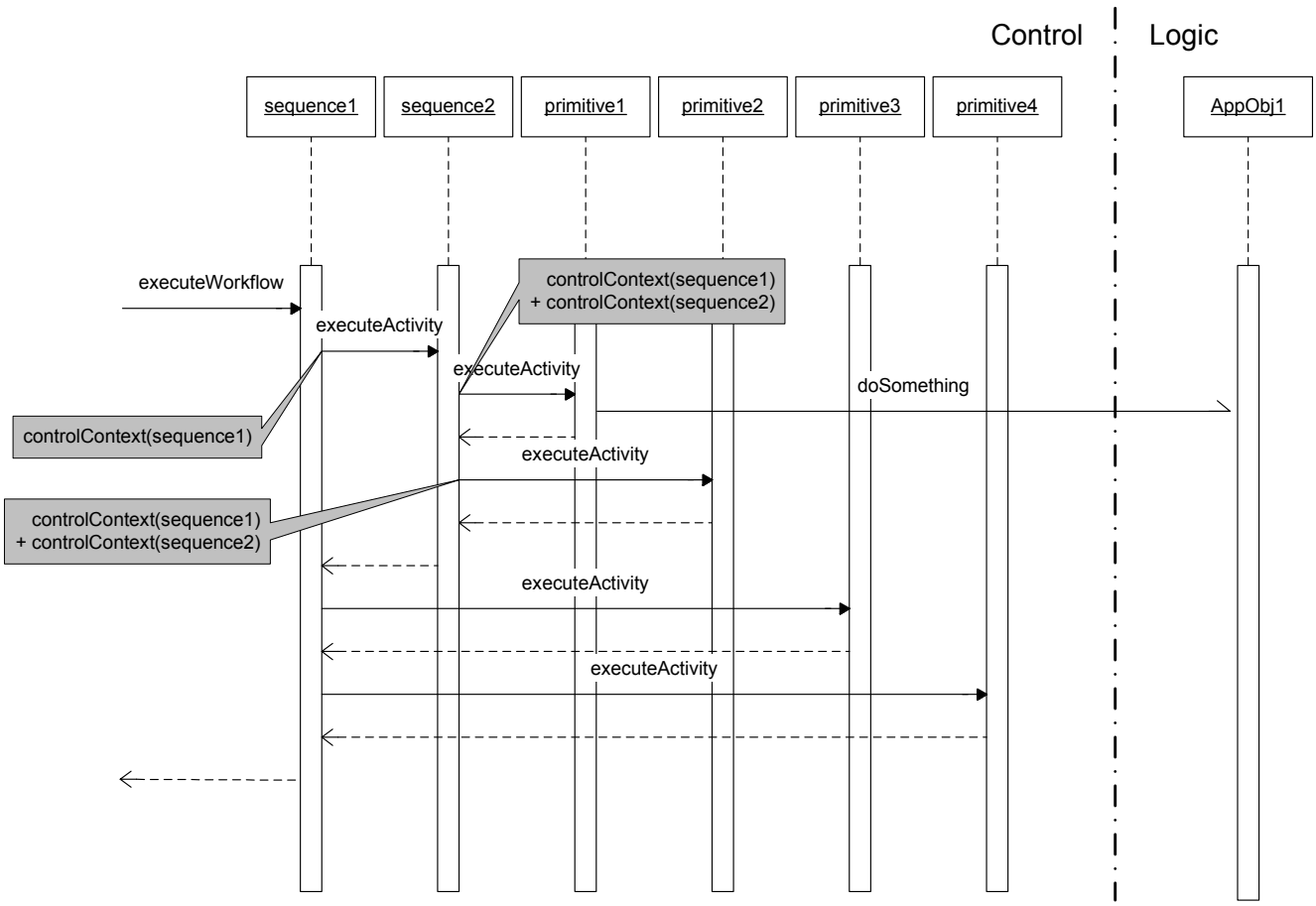

Figure 4: Activity map navigation through message sends between the activity nodes. Chained executeActivity: messages cause the growth of the control context. For simplicity, only one application object is shown in the logic tier.

when supplied with a control context. Therefore, continuation objects represent an abstraction of the control context associated with each activity. Workflow enactment takes place in a loop that iterates over continuations and supplies them with the control context. Developers familiar with design patterns can regard the continuations as command objects [12]. Command objects also factor out the control, but typically they're not used in the chained fashion typical of continuation-passing style.

Ganz, Friedman, and Wand [13] call the above technique *trampolined style*, and describe its use for implementing multithreading. The trampoline (i.e., a loop) drives the entire computation by bouncing from one continuation to the next. In the context of micro-workflow, the trampolined style replaces the chain of message sends with a single message send from the trampoline to the executing activity node. This translates into limiting the amount of control context that the enactment mechanism accumulates as it navigates the activity map. The diagram from Figure 5 sketches this situation for the example workflow from Figure 3.

5.4 Asynchronous Message Sends with Future Objects

Although most developers building workflow select an object-oriented language that supports threads, popular languages have minimal support for concurrency. Unlike full-fledged concurrent languages [19], general-purpose programming languages like Smalltalk or Java implement message sends in a synchronous manner. In other words, upon sending a message, the sender waits until the receiver processes it and returns. However, synchronous processing is not suitable for workflow. Workflow participants may respond to requests asynchronously. Typically this happens when workflows involve actions from humans, external systems, or objects/components that are not available continuously. Additionally, some workflow participants may take a long time to complete processing. If messages are synchronous, once the enactment mechanism sends a message to an object representing one of these participants, it must wait until the processing completes. To deal with these circumstances a workflow system must support asynchronous message sends from the workflow object to the application object. Since typically the implementing object-oriented language provides only synchronous messages, this requirement represents another obstacle to building a workflow system with object technology.

Micro-workflow supports asynchronous messages with future objects, a technique from concurrent programming languages [17, 1, 40]. Future objects provide placeholders for objects whose identity is determined after the future objects representing them are created. An asynchronous message returns to the sender immediately, without waiting for the receiver to complete processing. However, the return value is a future object instead of the real result of the operation. Once the real result becomes available, it replaces the future transparently [32].

For example, assume that a workflow activity involves the result

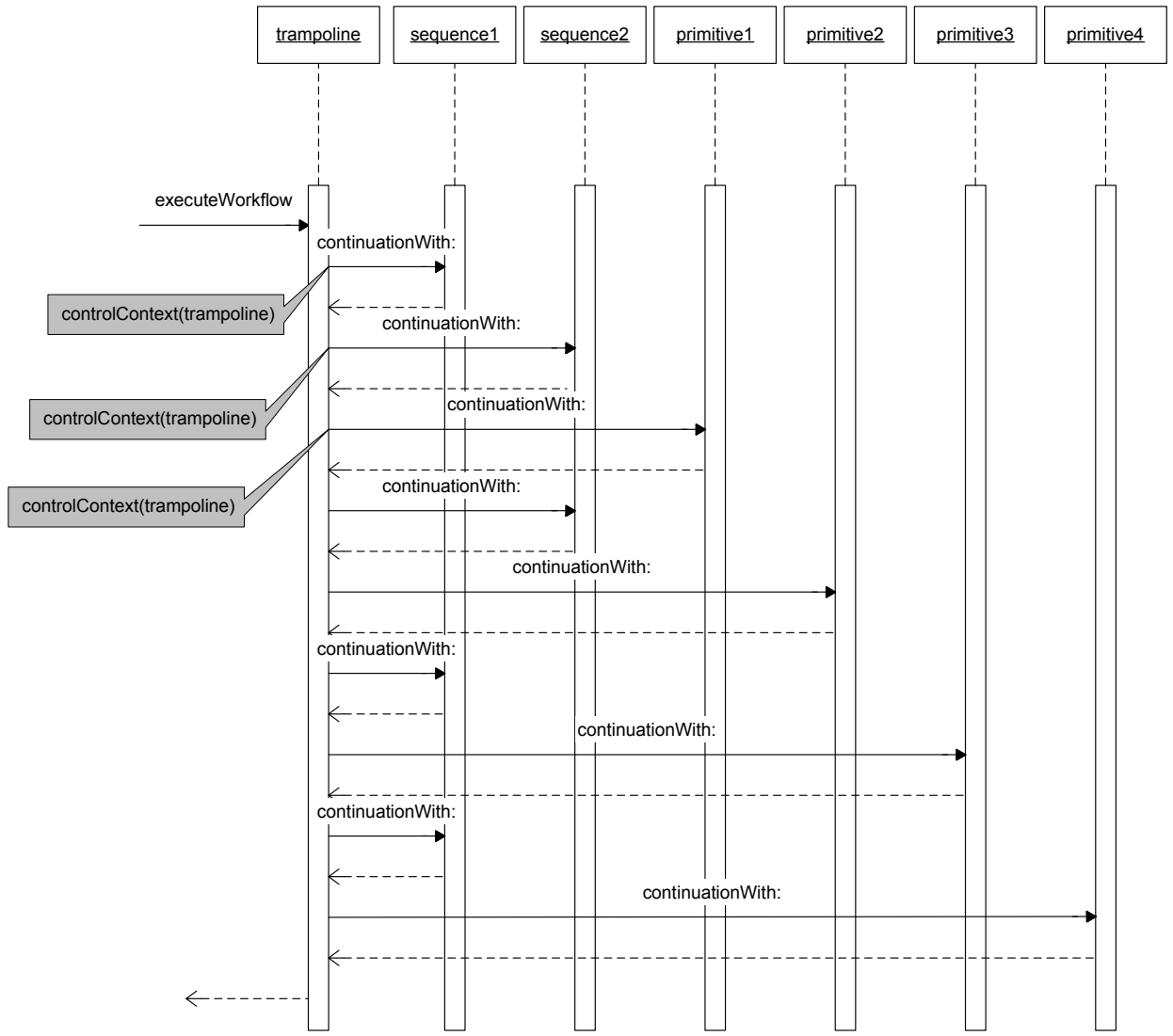

Figure 5: Activity map navigation, trampolined style; note the absence of chained message sends. For simplicity, only the objects on the control tier are shown.

of a long batch job. With synchronous messaging, the enactment mechanism sends a message to an object from the logic tier requesting the batch job. This message starts the job, but it doesn't return the control to the enactment mechanism until the batch job finishes. With asynchronous messaging, the enactment mechanism receives a future object in response to its request and continues execution. The batch job starts executing in a separate thread; upon its completion, the result object replaces the future returned in its place.

The above mechanism maintains the appearance of synchronous message sends. However, what happens if other objects attempt to use a future before it has been replaced with the result object returned from the logic tier? Message sends to future objects indicate that the workflow has reached a point where it involves objects which are not yet available. In response to these messages, futures suspend the execution within the workflow domain until the asynchronously-produced application objects requested to process the message become available.

The solution described above involves two tricky aspects. First, future objects require the ability to replace an object with another such that all references to the replaced object point to the replacement. Second, futures require the ability to trap message sends to the future object and suspend them until the future object is replaced. Reflective facilities like the ones available in Smalltalk–80 [10] provide elegant solutions for both requirements. Alternatives that don't rely so heavily on reflection but introduce additional objects are also viable (e.g. forwarding proxies). I'll present my solution in the context of the activity type that uses futures, in Section *5.6.1*.[3] First let's look at the implementation of the trampoline.

5.5 Implementing Trampolined Workflow Enactment

At the heart of the enactment mechanism, the Workflow class provides the loop driving the computation. This class represents

[3]The micro-workflow worklist component also uses the asynchronous invocation mechanism to add support for human workers. This paper doesn't cover the worklist component; I have discussed it elsewhere [26].

workflow definitions and provides the means to execute them. Programmers start a *workflow case* through sending the execute message to an instance of this class. Workflow execution uses a slotted environment (like Smalltalk's IdentityDictionary) for data flow. The enactment mechanism provides this environment to each continuation as it passes control to it. In turn, the continuation obtains the domain objects it needs from the environment; some continuations may also extend the environment by adding new slots. In effect, this scheme pushes the data through the workflow.

The executeWith: message implements the trampoline. Each step processes the continuation at the top of a continuation FIFO queue and puts the result (also a continuation) back on the queue. The loop exits when it reaches a particular continuation that marks that there's nothing else to do. Here's the relevant Smalltalk code:

```
Workflow>>executeWith: aContext
    | k |
    k := self firstContinuation.
    [k shouldStop] whileFalse: [k := self bounce: k with: aContext].
    ^k applyContinuationIn: aContext

Workflow>>bounce: k with: aContext
    kqueue nextPut: (k applyContinuationIn: aContext).
    ^kqueue next
```

The micro-workflow core uses several types of continuations corresponding to each activity type. Their base class Continuation defines the trampoline interface:

```
Smalltalk.Microworkflow defineClass: #Continuation
    superclass: #Core.Object
    indexedType: #none
    private: false
    instanceVariableNames: 'continuation workflow '
    classInstanceVariableNames: ''
    imports: ''
    category: 'Workflow-Execution'

Continuation>>applyContinuationIn: aContext
    ^self subclassResponsibility

Continuation>>shouldStop
    ^false
```

InitialContinuation marks the beginning of the computation. i.e., there's nothing else to do with the context passed to it as the argument of applyContinuationIn:. It reimplements the shouldStop test message accordingly. The enactment mechanism introduces an instance of InitialContinuation in the loop when it starts executing a workflow. Once the loop exits, InitialContinuation responds to (the final) applyContinuationIn: by returning its argument—the slotted environment for data flow. This represents the return value of the trampoline.

The code of the key messages implemented by InitialContinuation follows:

```
InitialContinuation>>shouldStop
    ^true

InitialContinuation>>applyContinuationIn: aContext
    ^aContext
```

5.6 A Process Model with Continuations

Each type of activity node builds and returns a different continuation to the trampoline. The activity-specific processing takes place in these continuations, in response to the applyContinuation: message.

Two parallel class hierarchies provide the functionality required to define and execute workflows. The first hierarchy belongs to the *process realm* and provides the abstractions for the activity-based process model. Developers use instances of these classes to define workflows. At the root of this hierarchy, the Activity abstract class defines the interface used for navigating the workflow definition:

```
Activity>>continuationWith: k
    ^self subclassResponsibility
```

The second hierarchy belongs to the *enactment realm* and provides the continuations that encapsulate activity-specific execution mechanisms.

The Continuation superclass described in the previous section defines only the trampoline interface (applyContinuationIn:). The following examples show the implementation of three control structures (primitive, sequence, and conditional) by specializing this class.

5.6.1 Primitive

A Primitive is an abstraction of a piece of work performed by an object from the logic tier. Instances of this class handle the execution of application-specific actions through delegation to application objects from the logic tier. They also have the possibility of pulling *Workflow Relevant Data* (i.e., application-specific information) into the workflow runtime.

In the process realm, the Primitive class holds the information required to send a message to an application object, crossing the boundary between logic and control. This consists of the object's slot name, the message, its arguments, and the result slot name (which is optional, to accommodate messages without return values). In response to continuationWith:, the Primitive returns a continuation of the appropriate type.

```
Primitive>>continuationWith: k
    | rk |
    rk := k makePrimitiveContinuation.
    rk
        subject: subject;
        selector: selector;
        arguments: arguments;
        result: result.
    ^rk
```

In the enactment realm the PrimitiveContinuation class implements the corresponding execution mechanism. The delegation to application objects uses the asynchronous invocation mechanism and thus depends on future objects.

As Section 5.4 has mentioned, future objects require the ability to transparently replace an object with another, and to intercept message sends. My implementation of future objects uses several reflective facilities available in Smalltalk–80. Note, however, that the implementation described here is otherwise language-independent.

The replacement of a future object with a result object relies on Smalltalk's oneWayBecome: message. The virtual machine replaces the receiver of this message with its argument such that all references to the receiver (i.e., future object) point now to the argument (i.e., return object). Note that a similar mechanism could also be supplied by the implemented system (thus in a language-neutral manner) by updating the appropriate slot of the workflow environment and enforcing a single point of access.

To intercept messages addressed to result objects I have tailored the message dispatch mechanism of the Future class. Instances of this class have no inherited behavior (in Smalltalk their superclass is nil). Consequently, the doesNotUnderstand: mechanism traps the messages that require the availability of the result object and delays their processing. The key aspects of this implementation of future objects follow.

```
Smalltalk.Microworkflow defineClass: #Future
    superclass: nil
    indexedType: #none
    private: false
    instanceVariableNames: 'semaphore '
    classInstanceVariableNames: ''
    imports: ''
    category: 'Workflow-Support'

Future>>doesNotUnderstand: aMessage
    self semaphore wait.
        ^self perform: aMessage selector withArguments: aMessage arguments

Workflow>>replaceWithWorkflowRelevantData: anObject
    |localSemaphore signalsToSend|
    localSemaphore := self semaphore.
    signalsToSend := self waitingProcesses.
    self oneWayBecome: anObject.
    signalsToSend timesRepeat: [localSemaphore signal]

Future class>>asyncPerform: aSelector with: arguments on: anObject
    | future |
    future := Future new.

    [| return |
    return := arguments notNil
            ifTrue: [anObject perform: aSelector with: arguments]
            ifFalse: [anObject perform: aSelector].
    future replaceWithWorkflowRelevantData: return]
                fork.
    ^future
```

In response to the applyContinuationIn: message, the continuations returned by primitive activity nodes obtain the application object from the workflow environment and send the message in a separate thread, or delegate the asynchronous processing to the Future class.

```
Primitive>>applyContinuationIn: aContext
    | target |
    target := aContext at: subject.
    result isNil
          ifTrue: [self asyncExecuteOn: target]
            ifFalse: [aContext at: result put: (self asyncExecuteAndReturnFrom: target)].
    ^self continuation

Primitive>>asyncExecuteOn: target
    [arguments isNil
          ifTrue: [target perform: selector]
          ifFalse: [target perform: selector with: arguments]]
                fork

Primitive>>asyncExecuteAndReturnFrom: target
    ^Future asyncPerform: selector with: arguments on: target
```

Note that Primitive performs application-specific work whereas other types of activity nodes merely coordinate the computation. Therefore, Primitive instances can appear only as leaf nodes in the activity map. applyContinuationIn: returns the caller's continuation, which is available to all continuation classes as an instance variable of the Continuation abstract class. This return value provides the context that requested the application domain action.

5.6.2 Sequence

Sequence represents an activity that has a number of steps, each of which is another activity. Software developers use instances of this class to specify a temporal ordering between workflow activities. The enactment mechanism navigates the steps sequentially.

In the process realm Sequence is a subclass of Activity that represents a composite [12], with other activities as its components. This property is key for the hierarchical decomposition of workflows, which allows developers to split a workflow definition into subworkflows. Subworkflows break down a workflow definition into pieces that may be reused. At run time the enactment mechanism treats the workflow and its subworkflows as a whole—they all execute as a single *workflow case*.

Sequence preserves the temporal ordering between its steps by the means of an OrderedCollection. Developers add steps through the add: message, in the order in which the enactment mechanism should execute them. In response to continuationWith:, Sequence returns a SequenceContinuation initialized with its steps.

```
Sequence>>add: anActivity
    steps add: anActivity

Sequence>>continuationWith: k
    | rk |
    rk := k makeSequenceContinuation.
    rk steps: steps readStream.
    ^rk
```

SequenceContinuation represents the sequence control context in the enactment realm. In response to applyContinuationIn: it returns the continuation of the next step. Once it executes its last step, applyContinuationIn: returns the caller's continuation—i.e., the control context that required the execution of the sequence of activities.

```
SequenceContinuation>>applyContinuationIn: aContext
    steps atEnd
        ifTrue: [^self continuation]
        ifFalse: [^self continuationForStep: steps next]

SequenceContinuation>>continuationForStep: anActivity
    ^anActivity continuationWith: self
```

Note that since instances of Sequence are composite activities, they can't be leaf nodes in the activity map representing the workflow definition.

5.6.3 Conditional

A Conditional enables developers to alter the control flow. At run time instances of this class determine how the enactment mechanism navigates the activity map based on *Workflow Control Data*. They implement a control structure of the type **IF** `condition` **THEN** `activity1` **ELSE** `activity2`.

In the process realm, the Conditional class holds the information required to branch based on data from the workflow environment.

This consists of the slot name for the *Workflow Control Data*, and the activities for the two branches. In response to continuationWith:, this activity type returns a properly initialized ConditionalContinuation.

```
Conditional>>continuationWith: k
    | rk |
    rk := k makeConditionalContinuation.
    rk
        subject: subject;
        thenBranch: self thenBranch;
        elseBranch: self elseBranch.
    ^rk
```

Note that not both branches are required. When the `activity2` branch is missing, the conditional corresponds to an **`IF`** `condition` **`THEN`** `activity` control structure (e.g., a guarded command). Likewise, when the `activity1` branch is missing, the conditional corresponds to an **`UNLESS`** `condition activity` control structure. The branch accessors return a NullActivity object if a branch has not been specified. This is a concrete Activity subclass which responds to continuationWith: by returning the parent continuation.

```
Conditional>>elseBranch
    ^elseBranch isNil ifTrue: [NullActivity new] ifFalse: [elseBranch]

Conditional>>thenBranch
    ^thenBranch isNil ifTrue: [NullActivity new] ifFalse: [thenBranch]

NullActivity>>continuationWith: k
    ^k
```

In the enactment realm the ConditionalContinuation fetches the *Workflow Control Data* from the runtime and takes one branch or the other depending on its value. Like for other composite activities (e.g., sequence), this involves building the appropriate continuation and returning it to the trampoline.

```
ConditionalContinuation>>applyContinuationIn: aContext
    ^((aContext at: subject) ifTrue: [thenBranch] ifFalse: [elseBranch])
        continuationWith: continuation
```

5.7 Summary

The enactment mechanism presented above uses continuations to reify the control flow. Continuation objects abstract the control context associated with the activities of a workflow. Programmers can manipulate these objects directly. This makes dealing with long running jobs feasible, and facilitates implementing the workflow features that involve full access to the control context (i.e., history, recovery, manual changes/ad hoc workflow, etc.).

6. LANGUAGE REQUIREMENTS

The micro-workflow components described in the previous sections employ techniques typically used in the context of functional and concurrent programming languages. The enactment mechanism uses *continuation objects* to break chained message sends, and thus prevents the growth of control context. Likewise, the process model uses *future objects* to provide an asynchronous invocation mechanism on top of synchronous message sends. I have shown a Smalltalk implementation for these components. Can object-oriented developers who use other languages leverage these techniques? What kind of language features are required to implement a micro-workflow architecture? The answer to these questions determines what language features developers should look for if they want to use micro-workflow in their applications. It also shows language designers what features they should consider for new languages.

Trampolined style workflow enactment doesn't require language support for continuations. The continuation objects used by the workflow enactment mechanism are in fact command objects [12]. As such they rely on polymorphism. Therefore, the micro-workflow execution component is practical for developers regardless of what object-oriented language they use.

Future objects require some language support. Reflective capabilities like Smalltalk's oneWayBecome: provide an elegant way of replacing an object with another and updating all references transparently. Forwarding proxies can provide a similar mechanism in a language-neutral manner. However, the ability to intercept message sends to future objects requires access to the message sending mechanism of the implementing language. In other words, the micro-workflow process component requires reflection.

To summarize, developers who want to implement workflow enactment with continuation and future objects in a general-purpose programming language need reflective capabilities. Although reflection has been around for many years, popular languages like Java have started adding the type of reflection capabilities discussed here only recently (e.g., Java's dynamic proxies, introduced with JDK 1.3, let developers create objects that selectively forward or intercept invocations). Therefore, as language designers continue to grow and refine the reflective capabilities of their languages, the above techniques are becoming accessible to a wider audience.

7. CONCLUSION

Adapting software to evolving business requirements is hard and thus expensive. Although people have recognized that flow-independence—the separation of the logic and control aspects—makes applications easier to modify and change, most programming languages do not support it. In fact, with the distribution of control among different classes, the object paradigm goes in the opposite direction. Therefore, an increasing number of developers have started to implement the control aspect within their applications with workflow. As a sign of this trend, workflow has recently been at the focus of intense attention from the OMG.

Unfortunately, most workflow architectures have been designed under different assumptions and are ill-suited for integration within applications. Consequently, developers who seek lightweight and customizable workflow functionality must hand-craft their own solutions. This is not trivial. Additionally, several characteristics of workflow (e.g., explicit representation of control flow and asynchronous processing) seem incompatible with the object paradigm and with current object-oriented languages. Is there an impedance mismatch between object and workflow technologies?

In this paper I've discussed some of the most interesting aspects of implementing basic workflow support with object technology. I have introduced micro-workflow, a lightweight workflow architecture. Micro-workflow aims at object-oriented developers and represents a radical departure from traditional workflow architectures. More importantly, I've shown that techniques specific to programming languages provide answers to the impedance mismatch between workflow and objects. Just as continuation and future objects are good for building programming languages, they're also good for implementing workflow enactment. Finally, I have discussed what kind of language features developers implementing workflow might need to look for when they select a language. I've also identified the kind of features language designers should consider for new languages aimed at developers building agile software in general, and lightweight workflow functionality in particular.

8. ACKNOWLEDGMENTS

Bill Burdick, Brian Foote, Daniel Friedman, Adrian Kunzle, and Rich MacDonald have read and commented on drafts of this paper. I am grateful to them all.

9. REFERENCES

[1] G. A. Agha. *Actors—A Model of Concurrent Computation is Distributed Systems*. The MIT Press, Cambridge, Massachusetts, 1986.

[2] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley, 1998.

[4] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1995.

[5] F. Buschman, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture—A System of Patterns*. John Wiley & Sons, July 1996.

[6] Cincom, Inc. Cincom Smalltalk. On the Web at `http://www.cincom.com/scripts/smalltalk.dll/index.asp`.

[7] G. Dinkhoff, V. Gruhn, A. Saalmann, and M. Zielonka. *Entity-Relationship Approach–ER'94, Business Modelling and Re-engineering*, chapter Business Process Modeling in the Workflow Management Environment *Leu*, pages 46–63. Number 881 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

[8] A. Doğaç, L. Kalinichenko, M. T. Özsu, and A. Sheth, editors. *Workflow Management Systems and Interoperability*, volume 164 of *NATO Advanced Science Institutes (ASI), Series F: Computer and Systems Sciences*. Springer-Verlag, August 1998.

[9] Drala Software, Inc. Drala workflow engine. Available from `http://www.dralasoft.com/products/workflow/index.html`.

[10] B. Foote and R. E. Johnson. Reflective facilities in Smalltalk–80. In *Proceedings of OOPSLA'89*. ACM, 1989.

[11] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. The MIT Press, Cambridge, MA, second edition, 2001.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[13] S. E. Ganz, D. P. Friedman, and M. Wand. Trampolined style. In *Proc. International Conference on Functional Programming*, pages 18–27, Paris, September 1999. ACM Press.

[14] GemStone Systems. GemStone/S Smalltalk Application Server. On the Web at `http://www.gemstone.com/products/s/index.html`.

[15] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases, an International Journal*, 3:119–153, 1995. Available on the Web at `ftp://ftp.gte.com/pub/dom/reports/GEOR95a.ps`.

[16] C. J. Hagen. *A Generic Kernel for Reliable Process Support*. PhD thesis, Swiss Federal Institute of Technology, Zürich, Switzerland, 1999.

[17] R. Halstead, Jr. MultiLISP: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7:501–538, October 1985.

[18] N. Harrison, B. Foote, and H. Rohnert, editors. *Pattern Languages of Program Design 4*. Software Patterns Series. Addison-Wesley, 2000.

[19] Y. Ishikawa and M. Tokoro. Orient84/k: An object-oriented concurrent programming language for knowledge representation, 1987.

[20] M. Jackson and G. Twaddle. *Business Process Implementation–Building Workflow Systems*. Addison-Wesley, 1997. ISBN 0-201-177684.

[21] G. Kiczales and J. des Rivieres. *The art of the metaobject protocol*. MIT Press, Cambridge, MA, USA, 1991.

[22] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP '97—Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241, pages 220–242. Springer-Verlag, New York, NY, 1997.

[23] Z. Kováks. *The Integration of Product Data with Workflow Management Through a Common Data Model*. PhD thesis, Faculty of Computer Studies and Mathematics, University of the West of England, Bristol, Apr. 1999.

[24] R. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, July 1979.

[25] F. Leymann and D. Roller. *Production Workflow—Concepts and Techniques*. Prentice-Hall, Upper Saddle River, New Jersey, 2000.

[26] D.-A. Manolescu. *Micro-Workflow: A Workflow Architecture Supporting Compositional Object-Oriented Software Development*. PhD thesis, University of Illinois, Urbana-Champaign, October 2000. Available as Computer Science Technical Report UIUCDCS-R-2000-2186. On the Web from `http://micro-workflow.com/`.

[27] D.-A. Manolescu and R. E. Johnson. A micro workflow framework for compositional object-oriented software development. OOPSLA'99 Workshop on the Implementation and Application of Object-Oriented Workflow Management Systems II, Nov. 1999. Available on the Web from `http://micro-workflow.com/`.

[28] R. C. Martin, D. Riehle, and F. Buschmann, editors. *Pattern Languages of Program Design 3*. Software Patterns Series. Addison-Wesley, October 1997.

[29] R. McClatchey, J.-M. L. Goff, N. Baker, W. Harris, and Z. Kovács. *A Distributed Workflow and Product Data Management Application for the Construction of Large Scale Scientific Apparatus*, pages 18–34. Volume 164 of Doğaç et al. [8], August 1998.

[30] P. Muth, J. Weissenfels, M. Gillmann, and G. Weikum. Mentor-lite: Integrating light-weight workflow management systems within business environments (extended abstract), October 1998. Available on the Web from `http://www-dbs.cs.uni-sb.de/~mlite/`.

[31] P. Muth, J. Weissenfels, M. Gillmann, and G. Weikum. Workflow history management in virtual enterprises using a light-weight workflow management system. In *Proc. 9th International Workshop on Research Issues in Data Engineering*, Sydney, Australia, March 1999. Available on the Web from `http://www-dbs.cs.uni-sb.de/~mlite/`.

[32] J. Noble. Arguments and results. *The Computer Journal*, 43(6):439–450, July 2000.

[33] J. Noble. *Prototype-based Object System*, chapter 5. In Harrison et al. [18], 2000.

[34] Workflow management facility specification. OMG Document Number bom/98–03–01, 1998. Available on the Web at ftp://ftp.omg.org/pub/docs/bom/98-03-01.pdf.

[35] C. Petrie and S. Sarin. Controlling the flow. *IEEE Internet Computing*, 4(3):34–36, May–June 2000.

[36] D. Roberts and R. Johnson. *Evolving Frameworks—A Pattern Language for Developing Object -Oriented Frameworks*, chapter 26. In Martin et al. [28], October 1997.

[37] A. P. Sheth, W. van der Aalst, and I. B. Arpinar. Processes driving the networked economy. *IEEE Concurrency*, pages 18–31, July–September 1999.

[38] G. L. Steele. Growing a language. In *Addendum to the 1998 proceedings of the conference on Object-oriented Programming, Systems, Languages, and Applications*, New York, NY, 1998. ACM Press.

[39] G. L. Steele and G. J. Sussman. Lambda—the ultimate imperative. MIT AI Memo No. 353, March 1976.

[40] K. Taura, S. Matsuoka, and A. Yonezawa. ABCL/f: A future-based polymorphic typed concurrent object-oriented language—its design and implementation. In *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms*, 1994.

[41] Versata, Inc. Versata process logic engine. Available from http://www.versata.com/products/inSuite/logic.addon.html.

[42] G. Vossen and M. Weske. *The WASA Approach to Workflow Management for Scientific Applications*, pages 145–164. Volume 164 of Doğaç et al. [8], August 1998.

[43] J. Yang and M. P. Papazoglou. Interoperation support for electronic business. 43(6):39–47, June 2000.