A Micro-Workflow Component for Federated Workflow*

Dragoş A. Manolescu and Ralph E. Johnson
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{manolesc,johnson}@cs.uiuc.edu

1 Introduction

Recent research has helped software developers to obtain better insights into workflow technology, understand its interdisciplinary nature, and realize its potential for building flow-independent applications. As Petrie and Sarin point out, workflow is no longer just "some sort of planned document routing" [14]. Therefore, an increasing number of developers are embracing workflow technology [20, 17, 11, 16, 2].

However, current workflow systems are based on assumptions and requirements that don't hold in the context of software development. On the one hand, they target non-programmers and therefore package a wide range of features. This focus produced heavyweight and monolithic architectures, which are hard to customize and extend. On the other hand, software developers want workflow systems that they can tailor and integrate with their applications. This mismatch forces developers to build home-made workflow solutions whenever they need workflow functionality within their applications.

Micro-workflow is a new workflow architecture that addresses this problem. It aims at software developers and focuses on providing the type of workflow functionality they need in object-oriented applications. Several research projects focus on a new generation of workflow architectures [5, 8]. We have taken an approach that combines techniques specific to object systems and compositional software reuse to solve workflow problems.

The micro-workflow architecture consists of components. Several core components provide basic workflow functionality. They allow developers to define and execute workflows. Other components implement advanced workflow features. Software developers select the features they need and add the corresponding components to the core through composition. We have implemented components for history, monitoring, persistence, manual intervention, worklists, and federated workflow. In this paper we describe the federated workflow component.

2 Federated Workflow

Federated workflow involves the integration of several workflow management systems into a global workflow. In effect, it breaks process *execution* into parts that run on different workflow systems. This requires the ability to have a subworkflow as an activity node in the process activity map, and support for distributed process execution. We refer to the first as *hierarchical workflow*, and the second as *distributed workflow*.

Federated workflow has several important benefits. First, it departs from the centralized model typical of many workflow architectures. This improves availability and scalability. Second, it opens the doors to processes that span across organizational boundaries. Multi-organizational processes become increasingly important in the context of the "networked economy" [18]. These benefits prompted us to study a federated workflow component for the micro-workflow architecture.

*Additional information about micro-workflow is available at http://micro-workflow.com/Research/.

Micro-workflow revolves around components that implement basic and advanced workflow features. The federated workflow component implements hierarchical and distributed workflow, thus adding support for transparent workflow distribution across the enterprise. Hierarchical workflow requires an activity type that allows developers to use a workflow as a single node in the process definition. Distributed workflow requires a mechanism that handles distribution. The next sections focus on these aspects.

2.1 Hierarchical Workflow

Micro-workflow uses an activity-based process model. The key abstraction at the core of the micro-workflow process component (the component that provides the abstractions that let developers define workflows) is a *procedure*. Procedures represent control structures. Software developers define workflows by configuring and connecting different procedure *types*. The micro-workflow process component defines seven types of procedures—primitive, sequence, iterative, conditional, repetition, fork, and join. (It is worth mentioning that unlike most workflow systems, micro-workflow allows developers to extend the process component with new procedures.)

For hierarchical workflow we introduce a new control structure that represents a subworkflow. Executing a subworkflow procedure fires off the execution of another *workflow system*. When this subworkflow completes execution, its results become available to the parent workflow.

However, hierarchical workflow brings in concerns that don't exist when a single workflow system executes the entire process. An important aspect lies in the sharing of data between the two workflow systems. Data sharing represents a significant concern in the context of multi-organizational federated workflows. Organizations share data only under the circumstances defined by the federated process: participating workflows must release and receive *only* the information required for their execution, as specified in the process definition [15]. Therefore, the subworkflow procedure should allow developers to specify the rules for data sharing among the two workflow systems, and then enforce these rules.

2.2 Distributed Workflow

Each subworkflow procedure is associated with a workflow system. When this type of procedure executes, it fires off its workflow through a local message send. This allows developers to break workflow execution among workflow systems that reside within the same address spaces. To add support for distribution (and thus completing the requirements for federated workflow) the procedure must be able to work with workflow systems regardless of whether they reside within the same address space, or a different one.

We introduce a new abstraction that addresses this problem. On the end that executes the parent workflow, a workflow facade acts as a *proxy*. At the other end, a second facade acts as a *manager*. When the subworkflow procedure executes, the interplay of the two facades handles remote workflow execution and the transport of data between the two address spaces.

The micro-workflow architecture strives to encapsulate individual design decisions in separate components. Our choice of using separate abstractions for hierarchical and distributed workflow is consistent with this style. In effect, developers can tailor one aspect without affecting the other. For example, switching from a distributed application architecture to another incurs changing only the workflow facades. In addition, the low coupling between the subworkflow procedure and the workflow system behind the facade facilitates federating heterogeneous workflow systems.

3 The micro-workflow federated workflow component

We have built an object-oriented framework for micro-workflow in VisualWorks Smalltalk [6]. Figure 1 shows the UML class diagram for the micro-workflow federated workflow component. This section discusses the implementation of this component.

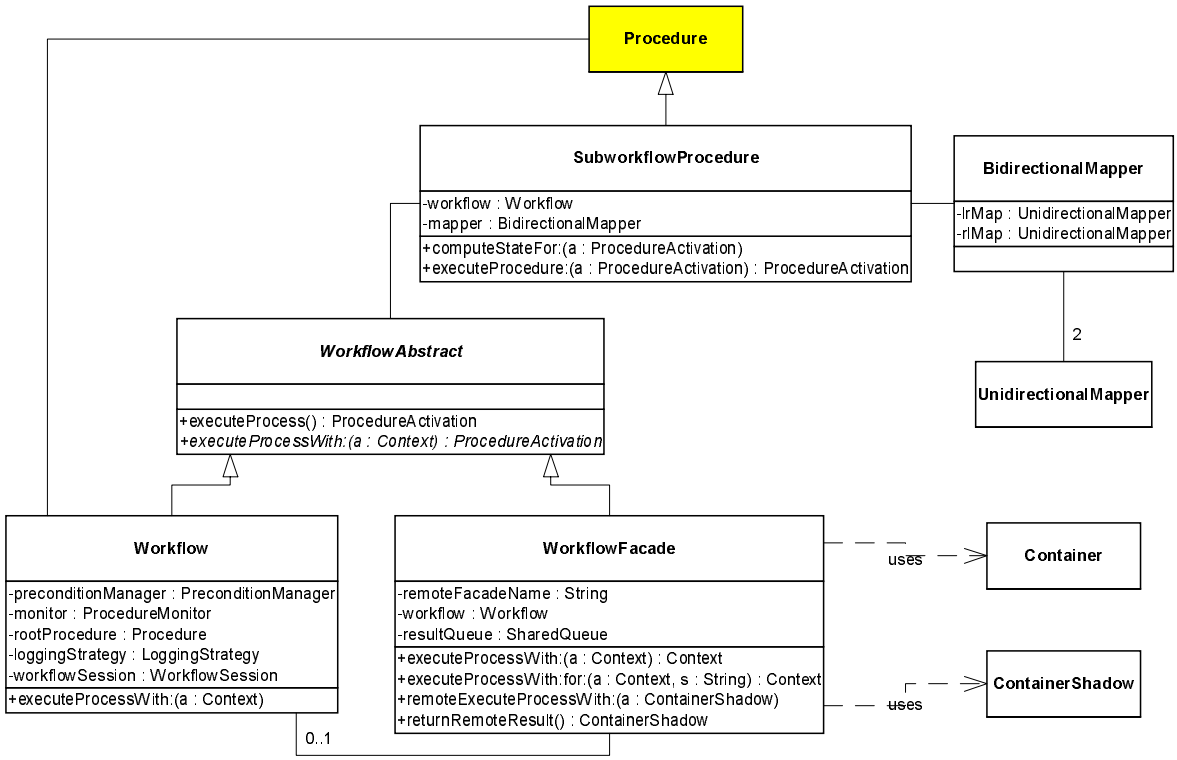

Figure 1: Class diagram for the micro-workflow federated workflow component. The colored/shaded class belongs to a different micro workflow component.

3.1 Hierarchical workflow

The Workflow class represents a workflow system. Users provide the process definition and the code that initializes the workflow runtime data. Instances of this class hold the objects required for workflow execution (e.g., process activity map). But unless framework users need to customize these objects, they don't interact directly with them. The Workflow class provides a process execution template that orchestrates workflow execution. It also manages the other framework components (history, persistence) or their user interfaces (monitoring, manual intervention, and worklists). The template starts the runtime services required to execute a workflow, fires off the process, and shuts down the runtime services after the process finishes execution. Developers control what micro workflow components they use by tailoring the process execution template.

SubworkflowProcedure provides an abstraction for a subworkflow. It enables developers to use a workflow system (i.e., instance of Workflow) as an activity node in the process definition. SubworkflowProcedure encapsulates the data flow between the two workflows, and the execution of the subworkflow. The transfer of control from the workflow to the subworkflow exposes only the objects needed by the subworkflow. Likewise, the return of control passes only the objects required by the workflow.

3.2 Inter-workflow data flow

Usually the data flow between two different address/name spaces involves translation. For example, when both workflow systems use contexts with named slots (as micro-workflow does), processing may require mapping between different slots. Figure 2 illustrates the data flow between a SubworklfowProcedure and the subworkflow associated with it. The input involves mapping the contents of the workflow's slot "4" into the subworkflow's slot "foo" (since the subworkflow expects to find at "foo" the object that the workflow stores at the slot "4") along with slots "1" and "2"

which don't require mapping. Once the subworkflow completes, its output slots "AA" and "CC" map into the workflow's "B" and "A." Consequently, SubworkflowProcedure requires framework users to program the data flow between the workflow name space and the subworkflow name space.

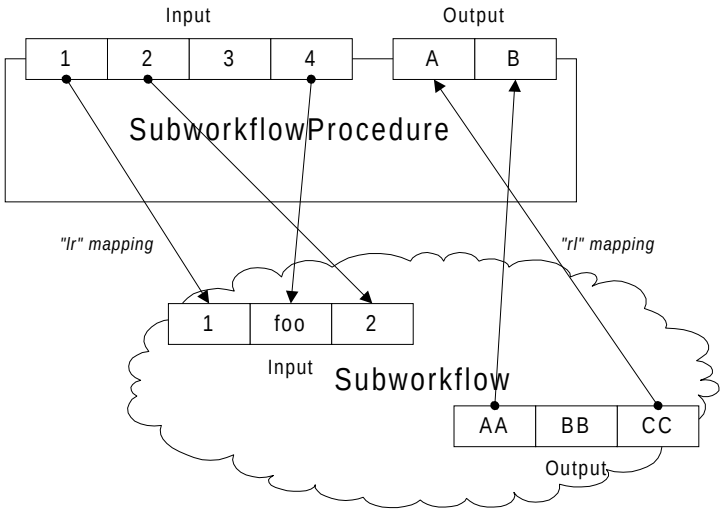

Figure 2: Data flow between a SubworkflowProcedure and a subworkflow.

Two classes provide the mechanism for the inter-workflow data flow and mapping between the context of the calling workflow and the context of the called subworkflow. UnidirectionalMapper maps into one direction, from a source to a destination. BidirectionalMapper aggregates two UnidirectionalMapper instances, one for each direction. Messages prefixed with "lr" (e.g., lrDirectMap:, lrMap:to:, and lrMapFrom:) affect the mapping from the workflow to the subworkflow. Likewise, messages with the "rl" prefix affect the mapping from the subworkflow back to the workflow (e.g., rlDirectMap:, rlMap:to:, and rlMapFrom:) The two mappings are annotated in Figure 2, and Figure 3 shows the Smalltalk code that configures the bidirectional mapping.

```
| mapper |
    mapper := BidirectionalMapper new.
    mapper
      rlDirectMap: #1;
      rlDirectMap: #2;
      rlMap: #4 to: #foo;
      lrMap: #AA to: #B;
      lrMap: #CC to: #A
```

Figure 3: Configuring the inter-workflow mapping.

Notice, however, that the Workflow and SubworkflowProcedure classes assume that the parent workflow and the subworkflow reside in the same address space (i.e., Smalltalk virtual machine).

3.3 Distributed workflow

The WorkflowFacade class provides an abstraction for workflow systems within different address spaces and adds support for distribution. Its clients interact with it through an IDL-style (i.e., message name and parameters) interface. This design doesn't expose any details about the workflow system WorkflowFacade represents, thus facilitating the

integration (through white-box techniques) of potentially heterogeneous systems. For example, the facade can encapsulate either object-oriented or legacy workflow systems (the latter wrapped to present an object-like interface). In effect, this class completes the functionality required by federated workflow. Because we didn't have a commercial workflow system, the federated workflow component described here assumes a homogeneous federation. However, since the design aims at offering a consistent view *above* the WorkflowFacade, this assumption *doesn't affect* the framework—the glue code connecting a workflow system to a facade is not reusable anyways. Additionally, having the same system implement both the workflow and the subworkflow reveals the issues that have to be dealt with at both ends.

The WorkflowFacade class leverages the CORBA-based distributed application architecture provided by VisualWorks Opentalk [3]. It enables a workflow running within one address space (i.e., the server) to execute a subworkflow within a different address space (i.e., the client):

- On the server, a WorkflowFacade instance represents a *Proxy* [4] for a remote workflow. Instances of this class replace Workflow instances transparently.
- On the client, another WorkflowFacade instance represents a *Manager* [19] that accepts requests from remote facades and executes workflows on their behalf.

Figure 4 shows an instance diagram that depicts the server and client facades.

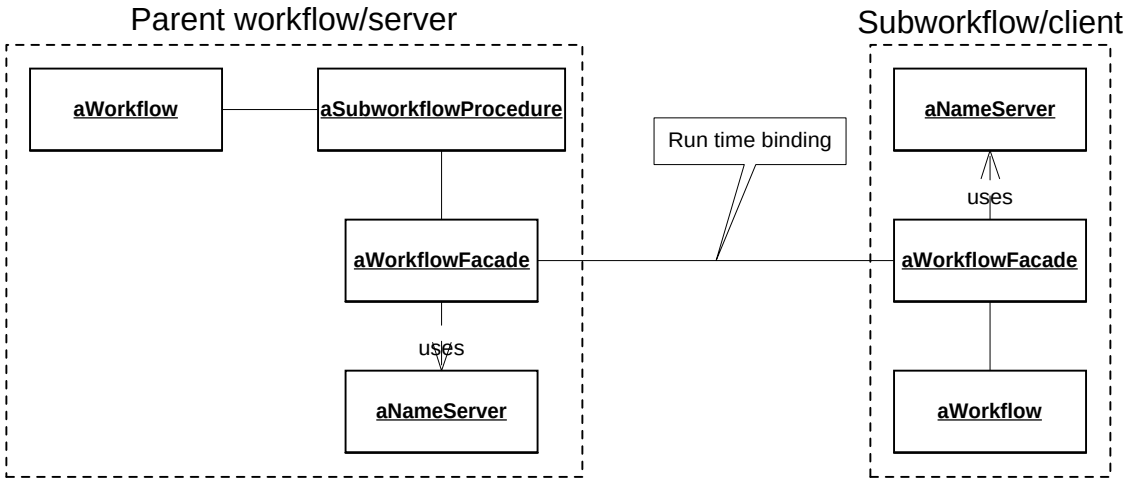

Figure 4: Federated workflow component, instance diagram.

Framework users specify that a SubworkflowProcedure triggers the execution of a remote workflow by building the procedure on a WorkflowFacade instance instead of a Workflow instance. The code fragment from Figure 5 shows the difference between using a local workflow system and a remote workflow system.

WorkflowFacade is a concrete subclass of WorkflowAbstract (Figure 1) and therefore has the same workflow execution interface. SubworklfowProcedure fires off subclasses of WorkflowAbstract through the executeProcessWith: message. This involves sending the remoteExecuteProcessWith: message to a facade residing in a different Smalltalk image. Next the local facade obtains the results of the subworkflow from a shared queue. Reading from this queue blocks execution until the remote workflow completes and sends back its output through the returnRemoteResult: message. In effect, the queue provides a synchronization point between the workflow and the subworkflow. Figure 6 shows the implementation of these messages and illustrates that the Opentalk STST framework makes remote message invocation (sending remoteExecuteProcessWith:) transparent.

The client facade executes the workflow in response to the remoteExecuteProcessWith: message. However, this message shouldn't wait for the return value since typically workflow execution takes much longer than message sends. Opentalk expects objects to process message sends synchronously, in a timely manner. To compensate for

```
localLabScreeningSubprocess

    | mapper |
    mapper := BidirectionalMapper new.
    mapper rlMap: #testResult to: #screening2.
    ^(SubworkflowProcedure on: LabScreening new)
        mapper: mapper;
        yourself

remoteLabScreeningSubprocess

    | mapper |
    mapper := BidirectionalMapper new.
    mapper rlMap: #testResult to: #screening2.
    ^(SubworkflowProcedure on: WorkflowFacade instance)
        mapper: mapper;
        yourself
```

Figure 5: Building a procedure that fires off a local and a remote workflow.

```
WorkflowFacade>>executeProcessWith: aContext
    | outgoingContainer containerShadow incomingContainer |

    outgoingContainer := Container fromContext: aContext.
    outgoingContainer callerName: self name.
    self remoteFacade remoteExecuteProcessWith: outgoingContainer asShadow.
    containerShadow := resultQueue next.
    incomingContainer := Container fromShadow: containerShadow.
    ^incomingContainer context

WorkflowFacade>>returnRemoteResult: aContainerShadow
    resultQueue nextPut: aContainerShadow
```

Figure 6: Subworkflow execution, server side.

the impedance mismatch between workflow and object time scales, the facade executes the workflow in a separate thread, through the executeProcessWith:for: message. This mechanism allows the remoteExecuteProcessWith: message to return control immediately. Additionally, combined with a pass by value data transfer mechanism, it requires only temporary connections between the server and the client—once to fire off the subworkflow, and once to transfer back its results. This characteristic has several consequences. First, it enables disconnected workflow execution, an important feature that is missing from many current workflow systems [1, 12, 13, 10]. For example, the subworkflow can run on a laptop computer which is removed from the network once the subworkflow starts execution. Second, as Hagen observes in his PhD thesis [5], this type of architecture facilitates maintenance tasks. While the subworkflow executes on the client, workflow execution on the server can be suspended for maintenance and updates.

On the client side, executeProcessWith:for: starts the execution of the local workflow, adding the objects passed by the server (i.e., the arguments) to its context. Once execution completes, it resolves the name of the caller and sends it the returnRemoteResult: message. This transfers the subworkflow's results to the parent workflow. Figure 7 shows the implementation of these message.

```
WorkflowFacade>>remoteExecuteProcessWith: aContainerShadow
remoteExecuteProcessWith: aContainerShadow
    | incomingContainer |
    incomingContainer := Container fromShadow: aContainerShadow.
    [self
        executeProcessWith: incomingContainer context
        for: incomingContainer callerName] fork

WorkflowFacade>>executeProcessWith: aContext for: aName
    | result outgoingContainer serverFacade |
    result := workflow executeProcessWith: aContext.
    outgoingContainer := Container fromContext: result.
    serverFacade := NameServiceRoot default resolveLeaf: aName.
    serverFacade returnRemoteResult: outgoingContainer asShadow
```

Figure 7: Subworkflow execution, client side.

Instances of the WorkflowFacade class use the Opentalk naming service to find their peers. At run time, the facades resolve the name of their peers with the naming service, which returns references to registered objects. For example, the server-side facade uses the name service to obtain the target for the remoteExecuteProcessWith: message (Figure 6). Likewise, once the subworkflow completes execution, the client-side facade resolves the receiver of the returnRemoteResult: message (Figure 7).

The UML sequence diagram from Figure 8 shows how the two WorkflowFacade instances orchestrate the remote workflow execution. The federated workflow component uses the STST Opentalk framework to send the remoteExecuteProcessWith: and returnRemoteResult: messages across Smalltalk images.

3.4 Data transport

By default, Opentalk uses the CORBA 2.0 pass by reference mechanism [9] between Smalltalk images. Sending messages to objects passed by reference incurs overhead and takes longer since Opentalk transfers the messages to the virtual machine where the object resides. Sometimes applications can't afford this impedance mismatch between local and remote message sends and therefore require passing objects by value (i.e., migrate the object from an address space to another) instead of by reference. To accommodate this situation, Opentalk also provides a mechanism that implements pass by value.

The federated workflow component knows how to use the Opentalk pass by value mechanism. Programmers can specify which workflow context objects the micro workflow framework should pass by value. This requires creating

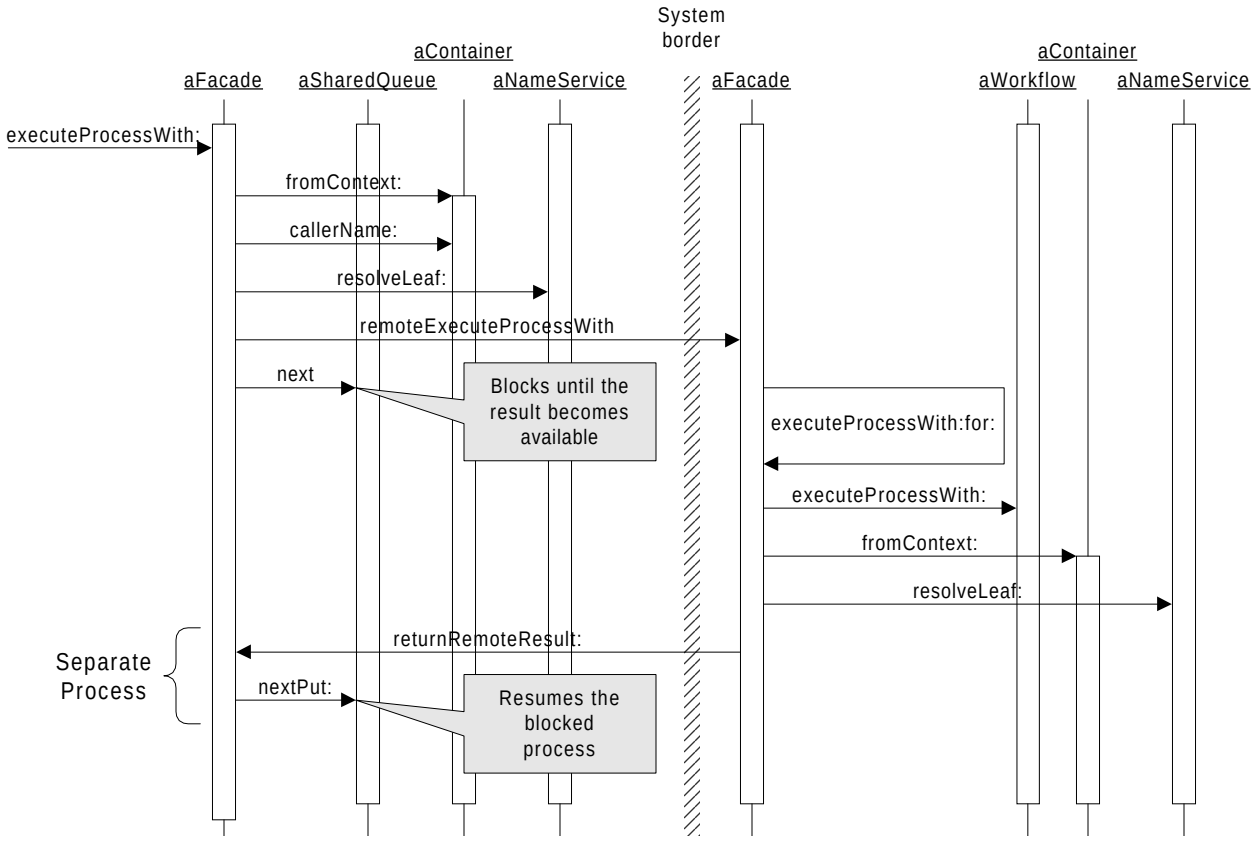

Figure 8: Remote workflow execution, UML sequence diagram.

a subclass of the Opentalk Shadow class for each class whose instances should be passed by value. Subsequently, Opentalk marshals and unmarshals instances of these Shadow subclasses. Each subclass provides the instance variables of the class it mirrors, along with accessors and mutators. The micro workflow framework hooks workflow objects and their shadows to the Opentalk marshaling and unmarshaling mechanism through the asShadow and unshadow messages, respectively. Therefore, each class that requires its instances passed by value should implement asShadow, and its shadow class should implement unshadow. Figure 9 shows how the Container class sends the asShadow and unshadow messages to domain objects and their shadows.

4 Example

4.1 The Newborn Followup Process

This workflow involves an administrative process from the Newborn Screening Program at the Illinois Department of Public Health (IDPH). Hospitals throughout the state collect blood samples from newborn babies and send them to a Chicago-based laboratory for testing. The followup process is triggered only when the lab returns anomalous test results. The objective of the process is to keep track of these problems, and ensure that they are dealt with according to the state law. Figure 10 shows the two workflows residing at different locations.

```
Container>>asShadow
        | shadowContext |
        shadowContext := IdentityDictionary new.
        context keysAndValuesDo: [:key :value | shadowContext at: key put:
            ((value respondsTo: #asShadow)
                ifTrue: [value asShadow]
                ifFalse: [value])].
        ^(ContainerShadow new) context: shadowContext; yourself

Container>>initializeFromDictionary: aDictionary
        aDictionary keysAndValuesDo: [:key :value | self at: key put:
            ((value respondsTo: #unshadow)
                ifTrue: [value unshadow]
                ifFalse: [value])]
```

Figure 9: The Container class uses the Opentalk shadow mechanism to pass objects by value.

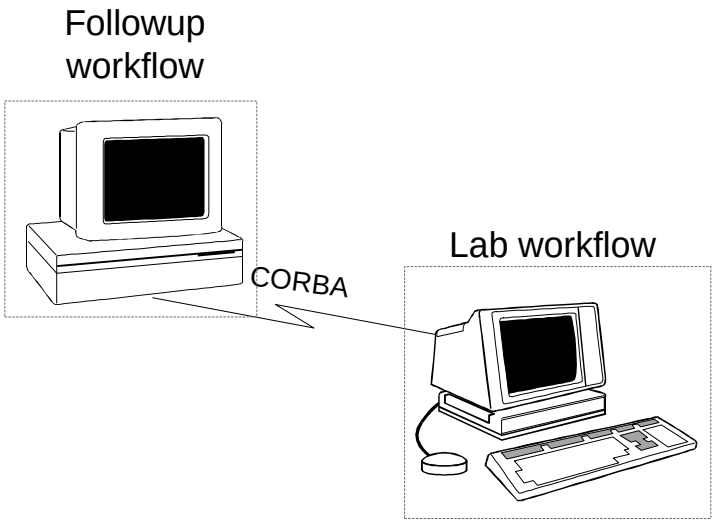

Figure 10: The followup workflow involves a subworkflow that runs at a different site.

4.1.1 Process Overview

The IDPH Newborn Followup Process starts when the test results indicate a potential problem. Upon receiving an abnormal test result, an IDPH employee contacts the newborn's physician and asks her to obtain a new blood specimen for a second test. Once the phone call completes, IDPH also sends the physician a letter with the information communicated over the phone. The physician collects the sample on a special filter paper and sends it to the Chicago laboratory for testing.

At the Chicago lab, a clerk acknowledges the receipt of the blood specimen by scanning the barcode printed on the envelope. Next a lab technician performs the test. Once the test completes, the lab supervisor certifies the result. The supervisor may ask the technician to redo the test if the results are on the borderline, if they are completely inconsistent with the first results, or if a problem occurred during testing—e.g., the punched paper was dropped into a different well of the batch board. Once certified, the lab releases the test result into the system. The certification improves the accuracy of the test results and ensures that the testing procedure doesn't miss any positives.

When the second test results confirm the problem, the IDPH employee assigned to the case contacts the physician again and provides a list with consultants who can handle the case. Once the phone call completes, IDPH also sends a letter with the information exchanged over the phone to the physician.

The physician refers the parents/guardian to a consultant. Seven days after the consultant reports that he is handling the case, the IDPH employee contacts him to check on the status of the case. He keeps contacting the consultant every seven days, until the case is either solved, or the consultant can't contact the parents/guardian (e.g., they have moved out of state).

4.1.2 Domain Objects and Workflow Actors

The IDPH Newborn Followup process involves domain objects as well as workflow actors (humans):

IdphSystem represents the IDPH system. It has the following responsibilities: print out and send the physician a letter requesting a second blood sample; print out and send the physician a letter with the list with consultants; and update the records.

IdphStaff (human) is the IDPH employee. It performs the following tasks: calls the physician to request a second blood sample; calls the physician to provide the list with consultants; and calls the consultant to check the status.

LabSystem represents the lab system. It is responsible for updating the lab records.

LabStaff represents a lab clerk or a lab technician. As the lab clerk, this domain object is responsible for acknowledging the receipt of blood specimens. As the lab clerk, it is responsible for performing a test involving a dry blood sample.

LabSupervisor (human) is the lab supervisor in charge with test certification.

4.1.3 Workflow Definition

The top level of the IDPH Newborn Followup Process executes at the IDPH site and consists of four steps:

1. Following the notice that the first blood screening indicates an abnormal result, the IDPH employee notifies the physician over the phone. Once the phone call completes, the system prints and mails a form letter. This step involves a sequence with two primitives. The first primitive queues a work item in the employee's worklist, and the second step handles the letter. Through a Future object, the object passed from the first step to the second (at the slot firstCallOk) ensures that the system sends the letter only after the employee closes the work item. Figure 11 shows the definition of this step.

```
firstScreeningProcess
    | call send |
    call := PrimitiveProcedure
                sends: #callAbnormalResult1For:
                with: #(#infant)
                to: #idphstaff
                result: #firstCallOk.
    send := PrimitiveProcedure
                sends: #sendAbnormalResult1For:callOk:
                with: #(#infant #firstCallOk)
                to: #idphsystem
                result: #firstLetterOk.
    ^SequenceProcedure with: call with: send
```

Figure 11: Newborn Followup Workflow—Notification of Abnormal Test Result.

2. The lab system receives the blood sample and runs the tests. This step involves the federated workflow component, which executes the lab workflow at the lab site. Let's first talk about what happens there.

   The lab workflow consists of a SequenceProcedure with three steps. First the lab clerk acknowledges the receipt of the blood specimen. This step involves a primitive. Next the lab technician performs the test and the lab supervisor certifies the results. Since the technician and the supervisor can repeat the testing and certification several times, this step involves a repetition with a two-step sequence as its body. Finally the lab sends out the certified test result. This last step involves another primitive. Figure 12 shows the definition of this workflow.

   The followup (parent) workflow requires the result of the lab workflow at the slot screening2. But as Figure 12 shows, the lab workflow puts the screening results into a slot named testResult. Thus the activity of the followup workflow that fires off the lab subworkflow configures a BidirectionalMapper instance to transfer data between the two slots. SubworkflowProcedure intermixes freely with the other control structures, and WorkflowFacade hides the fact that the subworkflow resides on a remote machine. These abstractions let the workflow designer focus on the process definition as a whole. Figure 13 shows the definition of this step.

3. The IDPH employee checks the results of the second screening. If the test results confirm the initial problem, he contacts the physician with the referral information. Once the phone call completes, the system prints and mails letter with the list with consultants communicated over the phone. The physician calls back with the name of the consultant who is going to handle the case. Next the IDPH employee will keep contacting the consultant every seven days until the case is closed. Therefore, this step involves a conditional (which checks the lab results) with a sequence as its body. The sequence has two primitives which handle the communication with the physician (phone and mail), and a repetition which handles contacting the consultant. A precondition ensures that the body of the repetition (another primitive) executes every seven days. Figure 14 shows the definition of this step.

4. Finally, the IDPH employee updates the records. This involves a primitive and Figure 15 shows the definition of this step.

In the second step of the followup, the lab subworkflow returns a blood test result object. This object becomes part of the case data in the followup workflow. Therefore, the federated workflow component must pass it by value instead of by reference. Figures 16–17 show the type of code required to pass a blood test result object by value. Figure 16 shows the definition of the BloodTestResult class and sketches how the domain object packs its state into a BloodTestResultShadow instance in response to the asShadow message. Likewise, Figure 17 shows the definition of the BloodTestResultShadow class and how it reconstructs a BloodTestResult instance in response to the unshadow message.

```
buildWorkflow
    | ack test certify rep send |
    ack := PrimitiveProcedure
                sends: #acknowledgeReceipt:
                with: #(#bloodSpecimen)
                to: #labclerk
                result: #bloodSpecimen.
    test := PrimitiveProcedure
                sends: #runTest:for:
                with: #(#testType #bloodSpecimen)
                to: #labtech
                result: #uncertifiedTestResult.
    certify := PrimitiveProcedure
                sends: #certifyTest:
                with: #(#uncertifiedTestResult)
                to: #labsup
                result: #certifiedResult.
    rep := RepetitionProcedure
                repeat: test , certify
                until: [:arg | arg isCertified or: [arg number = 5]]
                for: #certifiedResult.
    send := PrimitiveProcedure
                sends: #updateRecordsFor:
                with: #(#certifiedResult)
                to: #labsystem
                result: #testResult.
    self rootProcedure: ack , rep , send
```

Figure 12: Newborn Followup Workflow—Lab Workflow Definition.

```
labScreeningSubprocess
    | mapper |
    mapper := BidirectionalMapper new.
    mapper rlMap: #testResult to: #screening2.
    ^(SubworkflowProcedure on: WorkflowFacade instance)
        mapper: mapper;
        yourself
```

Figure 13: Newborn Followup Workflow—Lab System Subworkflow.

```
testSecondScreeningProcess
	| test |
	test := ConditionalProcedure
			if: [:arg | arg isAbnormal]
			for: #screening2
			execute: self secondScreeningProcess.
	test precondition: (Precondition withSubject: screening2
			block: [:screening | screening haveResults]).
	^test

secondScreeningProcess
	| call send contact loop |
	call := PrimitiveProcedure
			sends: #callAbnormalResult2For:
			with: #(#infant)
			to: #idphstaff
			result: #secondCallOk.
	send := PrimitiveProcedure
			sends: #sendAbnormalResult2For:callOk:
			with: #(#infant #secondCallOk)
			to: #idphsystem
			result: #secondLetterOk.
	contact := PrimitiveProcedure
			sends: #contactConsultantFor:
			with: #(#infant)
			to: #idphstaff
			result: #consultantResult.
	contact precondition: (Precondition withSubjectAt: #screening2
			block: [:screening | calendar
			currentDay ~= 0 and: [calendar currentDay \\ 7 = 0]]).
	loop := RepetitionProcedure
			repeat: contact
			until: [:arg | arg]
			for: #consultantResult.
	^call , send , loop
```

Figure 14: Newborn Followup Workflow—Processing of Second Screening Results.

```
updateProcess
	^PrimitiveProcedure
		sends: #updateRecordsFor:
		with: #(#infant)
		to: #idphsystem
		result: #finalresult
```

Figure 15: Newborn Followup Workflow—Updating the Records.

```
Smalltalk defineClass: #BloodTestResult
	superclass: #Core.Object
	indexedType: #none
	private: false
	instanceVariableNames: 'type abnormal certification number '
	classInstanceVariableNames: ''
	imports: ''
	category: 'Workflow-Example Followup'

BloodTestResult>>asShadow
	^(BloodTestResultShadow new)
		type: type;
		abnormal: abnormal;
		certification: certification;
		number: number;
		yourself
```

Figure 16: Opentalk pass by value, domain object side (VisualWorks 5i-style class definition).

```
Smalltalk defineClass: #BloodTestResultShadow
	superclass: #Opentalk.Shadow
	indexedType: #none
	private: false
	instanceVariableNames: 'type abnormal certification number '
	classInstanceVariableNames: ''
	imports: ''
	category: 'Workflow-Example Followup'

BloodTestResultShadow>>unshadow
	^BloodTestResult fromShadow: self
```

Figure 17: Opentalk pass by value, shadow side (VisualWorks 5i-style class definition).

5 Conclusion and future research

The micro-workflow architecture enables object-oriented developers to add workflow features by adding components to the micro-workflow core. In the context of the federated workflow component, this has several advantages over the monolithic approach adopted by current workflow architectures.

First, developers plug in a component only when they need the feature it implements. This allows them to tailor the workflow functionality to the requirements of each application. For example, applications that don't require federated workflow functionality don't use the federated workflow component.

Second, having separate components implement advanced workflow features localizes customizing a feature to that component. For example, although the federated workflow component uses CORBA for distribution, developers can change it to use Microsoft's DCOM or Java's RMI. The micro-workflow architecture makes these changes transparent for the other components.

Third, this design also localizes the changes required to federate heterogeneous workflow systems. The indirection provided by WorkflowFacade isolates the micro-workflow components as well as the classes of the federated workflow component from the specifics of the workflow system behind the facade.

We would also like to point out that the approach we've taken withthe micro-workflow architecture reduced the impact of adding the federated workflow component had on the other components. The SubworkflowProcedure class required adding one additional message to its superclass, Procedure (in the execution component). The remaining seven classes of the federated workflow component (see Figure 1) don't affect the other framework components, and therefore incurred no modifications.

One of the characteristics that set micro-workflow apart from other workflow architectures is that it allows its users to customize and extend it according to their needs. Therefore, we expect this architecture to grow. Evolution will bring changes, which means that it will be hard to declare that the work is finished. For example, the abstractions provided by the federated workflow component hide the workflow system executing a subworkflow behind a facade. This decreased coupling between the invoking workflow system and the invoked workflow system facilitates heterogeneous federations. We limited the discussion to homogeneous workflow systems and claimed that the abstractions provide enough separation. But this claim is not supported by empirical evidence. Future work should pursue this direction further and test the claim.

References

[1] G. Alonso, D. Agrawal, A. E. Abbadi, and C. Mohan. Functionality and limitations of current workflow management systems, 1997. Available on the Web at `http://www.almaden.ibm.com/cs/exotica/wfmsys.ps`.

[2] G. A. Bolcer. *Flexible and Customizable Workflow on the WWW*. PhD thesis, University Of California, Irvine, 1998.

[3] Cincom Systems, Inc. *VisualWorks Opentalk Application Developer's Guide*, 1999. Part Number P46–0131–00, Software Release 5i.1.

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

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

[6] 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, Nov. 1999. Available on the Web at `http://www.uiuc.edu/ph/www/manolesc/Workflow/PDF/oopsla99.pdf`.

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

[8] P. Muth, J. Weissenfels, M. Gillmann, and G. Weikum. Mentor-lite: Integrating light-weight workflow management systems within business environments (extended abstract). Available on the Web at `http://www-dbs.cs.uni-sb.de/~gillmann/Publications/Mentor-Lite-Zurich.p%s`.

[9] R. Orfali, D. Harkey, and J. Edwards. *Instant CORBA*. John Wiley & Sons, 1997.

[10] A. M. Ouksel and J. James Watson. The need for adaptive workflow and what is currently available on the market—perspectives from an ongoing industry benchmarking initiative. CSCW Towards Adaptive Workflow Systems Workshop, Seattle, WA, November 1998. Available on the Web at `http://ccs.mit.edu/klein/cscw-ws.html`.

[11] M. Papazoglou, A. Delis, A. Bouguettaya, and M. Haghjoo. Class library support for workflow environments and applications. *IEEE Transactions on Computers*, 46(6):673–686, June 1997.

[12] S. Paul, E. Park, and J. Chaar. Essential requirements for a workflow standard. OOPSLA'97 Business Object Workshop, 1997. Available on the Web from `http://jeffsutherland.org/oopsla97/`.

[13] S. Paul, E. Park, and J. Chaar. RainMan: A workflow system for the Internet. In USENIX, editor, *USENIX Symposium on Internet Technologies and Systems Proceedings, Monterey, California, December 8–11, 1997*, pages 159–170, Berkeley, CA, USA, 1997. USENIX.

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

[15] G. Piccinelli. Interaction modelling in federated process-centered environments. Technical Report HPL–98–54, HP Laboratories, Bristol, UK, March 1998.

[16] F. Ranno, S. K. Shrivastava, and S. M. Wheather. A system for specifying and coordinating the execution of reliable distributed applications. Technical report, Department of Computing Science, University of Newcastle upon Tyne, 1998. Available on the Web at `http://arjuna.ncl.ac.uk/group/papers/p062.ps`.

[17] S. Rausch-Schott. TRIGS$_{flow}$—*Workflow Management Based on Active Object-Oriented Database Systems and Extended Transaction Mechanisms*. PhD thesis, Institute of Applied Computer Science, Johannes Kepler University, Linz, Austria, Feb. 1997. Published by Trauner Verlag, Linz, ISBN 3-85320-991-2.

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

[19] P. Sommerland. *Manager*, chapter 2, pages 19–28. In Martin et al. [7], October 1997.

[20] P. S. C. Young. *Customizable Process Specification and Enactment for Technical and Non-Technical Users*. PhD thesis, University Of California, Irvine, 1994.