In Proceedings of the 6th International Enterprise Distributed Object Computing Conference (EDOC 2002), 17-20 September 2002, Lausanne, Switzerland,

Workflow-based composition of web-services: a business model or a programming paradigm?

Dinesh Ganesarajah[1]
Orbis, NDS,
London, UK
dganesar@orbisuk.com

Emil Lupu
Department of Computing
Imperial College, London, UK
e.c.lupu@doc.ic.ac.uk

Abstract

While SOAP/XML is perceived as the appropriate interoperability level for web-services, companies compete to provide workflow-based tools for web-service integration. This paper presents the design and implementation of a prototype workflow management system for building new web-services from a workflow of existing web-services. This enables the creation of multiple layers of value-added service providers and provides fast service creation, customisation and deployment. The system caters for multiple workflow paradigms, provides an extensible language for workflow specification and emphasises encapsulation and tight constraints on workflow execution. To expose a workflow of web-services as a web-service, several design steps have been required including the deployment as a web-service of the generic workflow engine and a generalisation of the Visitor Pattern to concurrent visitors.

1. Introduction

In the early days the Web was mainly used to publish information. Then, application servers were used to offer services to human customers. We now witness development of the Services Web where the services can be accessed programmatically and application servers collaborate with each other, typically using the Simple Object Access Protocol (SOAP). The Services Web evolves out of the desire to perform transactions in an open and automated environment with ubiquitous services, rather than using other mechanisms such as EDI or manual processing. The Services Web environment typically exhibits the following characteristics:

- Web Services are black-box components that encapsulate behaviour. The underlying object model and implementation technology are hidden though the functionality is not.
- Web Services interact using SOAP over HTTP thus providing, through the use of wrappers, interoperation between technology specific components such as DCOM, CORBA components or Enterprise Java Beans (EJBs).
- Web Services can be discovered at run-time for dynamic binding. Their APIs are published in a standard format (WSDL) that can be inspected and invoked dynamically; in essence a liberal form of reflection.
- Several Web Services can be orchestrated to perform a series of functions in a workflow.

Web Services are units of extremely low coupling; they communicate with each other with low dependency on the other party. In many respects they are akin to deployed components; a similarity that has created both confusion and controversy. Can web services be composed like components? Can the result be exposed as a web service, thus providing hierarchical composition? What are the restrictions imposed by such strict encapsulation? One possibility is to manually program in a Web Service its interactions with other Web Services, or to implement in a Web Service the functionality needed to find other services and bind to them. However, Web Services are expected to become ubiquitous. In Hewlett-Packard's Cool Town project [1] every entity has a URI and can be represented by a Web Service. This ranges from businesses to handheld devices and even people. In such a world, the ability to compose Web Services through workflows rapidly and with less effort than through general programming will be of substantial value. The potential for this evolution is considerable. Multiple layers of value-added service providers could easily be formed where providers aggregate existing services into new web-services. Services customised to each client's specific needs could be easily created and deployed. And even customers themselves could aggregate existing services into new more convenient applications.

[1] Work undertaken while at Imperial College, London, UK

Workflows emphasize the separation of control and information flows between components from the actual execution of the code in the underlying components. This separation provides the ability to easily rearrange and change the components.

By and large, current workflow languages consider workflow as a graph problem, with control flow and data flow described in terms of lines in a graph; this is exemplified by the Web Service Flow Language (WSFL) [2]. However, such an approach produces systems that are difficult to maintain and modify because control lines are similar to programming with *goto* statements.

The approach to workflow specification we consider here is that of encapsulating boxes of control. Each 'box', defines a Non-Terminal Expression of the workflow language, which determines control within that box. For example, a Sequence Expression entails that all constituent expressions are executed in order, while a Concurrent Expression entails that all constituent expressions are executed in parallel. Boxes (non-terminals) can then be recursively encapsulated avoiding the need for lines that define the control path between the expressions. A substantial advantage is that modifications to any part of the workflow are contained within the expression concerned, and Non-Terminals act as a structuring mechanism for decomposing large complex problems, and providing scalability for large workflows; a form of encapsulation. Additionally, an entire workflow or any subsection can be deployed as a new Web Service, for use in another workflow or in the workflow itself, and thus providing composition.

This paper presents a complete web-service environment having the characteristics mentioned above. The design and implementation cover the workflow service implementation as well as the specification language, workflow execution, monitoring tools and visualisation.

The paper is organised as follows: Section 2 will present the relevant related work from both the Web-Service environments and workflow management systems; Section 3 will present the overall system architecture; Section 4 will focus on the workflow language while Sections 5 and 6 will focus on the workflow engine and the user interface respectively.

2. The Scene

The success of Web Services will be determined by the availability of tools and product support for building, enacting and interoperating between web services. All major software vendors have tried to position themselves into the field by developing their own solutions.

As a first step, software development environments have been extended to facilitate the development and deployment of web-services. In Microsoft's .NET framework, programming code can be written in almost any language, including Microsoft's C#, and targeted for deployment on a variety of mediums including Web Forms and Web Services [4]. The cornerstone of the architecture is that all the deployment models produce self-describing components that are not directly dependent on other components.

IBM, who has played an active role in the development of standards like UDDI, SOAP and WSDL, has integrated support for creating web applications in its VisualAge and WebSphere products and now provides a suite of freely available test tools for web-service development through these products and from AlphaWorks.

The Sun ONE architecture [3] provides a method of web service development based on Java that permits the deployment as Web Services and Applications of macro services composed from developer written pre-built components (micro services).

Hewlett Packard has developed a Web Services Platform that includes tools for the graphical specification, creation and management of Web Services. Specifically, the HP Services Composer can be used to automatically create WSDL files and deploy Java Beans as Web Services. The Web Services work has evolved from HP's previous work on E-Speak [5].

Existing **Business Workflow** frameworks are not always suitable for orchestrating Web Services. Firstly, because Web Services do not immediately fit into the current workflow components. Secondly, because Web Services have properties like reflection that are not readily considered in business workflow systems. Finally, because business process workflows are geared for dealing with human users whereas Web Service workflows need only interact with other automated services – human users are implicitly represented through the Web Services they use. However, the techniques used in business workflow systems and Enterprise Application Integration teach valuable lessons for the orchestration of Web Services.

By and large Workflow Management Systems (WFMS) have similar structures, irrespective of the application domain or type. The general pattern used is characterized by the Workflow Management Coalition's Workflow Reference Model [6].

OTSArjuna [7], a WFMS for CORBA-based environments uses a graph-like notation to represent workflows. The graph is made up of nodes denoting tasks, which represent units of computation. Each task has a group of input and output sets. At runtime, a Workflow Repository service holds schemas of different workflows. A Workflow Execution service co-ordinates the workflow and delegates responsibility for executing and managing tasks to Task Controllers associated with each task, thus decentralizing the management of the workflow.

METEOR$_2$ [8] uses a more declarative language approach towards workflow specification than in OTSArjuna. Each task is associated with a directed graph representing the states into which the task can change to. Changes from one state to another arise through inter-dependencies to other tasks. These inter-dependencies are described using the Workflow Intermediate Language (WIL), which is specific to **METEOR$_2$**. Both control and data inter-dependencies can be specified and the specification can be generated from a GUI application.

The creators of **RainMan** [9] argue against centralized Workflow Management Systems and consider some novel use-cases that are typical in an Internet environment. These include the ability to download and run workflow schemas, to reconfigure the workflow at runtime and to cater for devices that can go offline but that can still be assigned to tasks. RainMan workflows have Performers and Sources. Sources request Performers to complete tasks and can hold activities, which are essentially workflow schemas and comprise several tasks. Each Performer has a list of the different tasks it has been asked to perform. The RainMan system defines interfaces for Performers and Sources, which can be implemented in different ways. RainMan has been implemented as a RainMan Builder, which is essentially an applet for creating workflow specifications, which can also act as a Source and can monitor the completion of the activities.

WSFL [2] has two models of 'flow', Flow Model and Global Model. The former is concerned with describing workflow between several parties. The latter describes interfaces for Web Services and patterns of interaction between them. Conceptually, a Flow Model is made of Control Links between activities which represent business tasks within a process. Activities can be interpreted as a method call within a conversation of methods calls in the business process. Control Links can have transition conditions and data links are superimposed over control links.

The Workflow Management Facility specification for CORBA accepted by the OMG is **jointFlow** [10], which describes a set of interfaces that can be implemented to create WFMS systems built on parts that can interoperate with other WFMS systems. The interfaces include *WfRequesters* that have *WfProcesses*, representing workflow schemas. WfProcesses hold a set of *WfActivities*, which are the tasks to be completed. Each WfActivity is assigned to a *WfResource*.

Microsoft's **BizTalk** [11] Orchestration framework provides the means to coordinate in a workflow the applications and components it supports including SOAP accessible components. At first sight BizTalk orchestration has similar objectives to those presented in this paper. However, BizTalk Orchestration lacks flexibility and some of the advanced aspects presented here. The language supports concurrent tasks, synchronization and dynamic task assignment to components. However, functionality such as choice is not provided in the workflow. Furthermore, BizTalk Orchestration focuses on enterprise level workflow between components, and does not consider some of the more general use-cases of Web Service workflow. During workflow execution, monitoring is limited to querying the state of each component. The system offers a simple approach to the Web Service workflow but it is very constrained and does not solve other problems considered here, such as recursive workflow encapsulation.

3. Architecture

The ability to provide arbitrary web-service composition through workflow requires a WFMS architecture that caters for the creation of workflows, which can be used by other WFMS systems. Thus, other WFMS systems should be able to invoke this WFMS to enact workflows created with this system. To satisfy these requirements the fundamental design approach is to treat a workflow over web-services as a composite component, thus placing strict restrictions on the workflow encapsulation. The Workflow Engine, which enacts the workflow, is also implemented as a Web Service and accessible via SOAP. Thus, SOAP clients can be other workflow engines that access the engine enacting each workflow as a web-service. In turn, this requires an execution environment that caters for the concurrent execution of different workflow schema or several instances of the same schema over the same web services. The execution environment encapsulates the interpreters for the workflow schemas and provides persistency for schemas that have been created and deployed (Figure 1).

A User Application interacts with the *WorkflowEngine* Web Service using standard SOAP messages in order to: deploy a workflow schema, invoke the relevant workflow, obtain information on the progress of the workflow enactment and perform other operations such as retracting the workflow. The User Application provides both a graphical and a textual interface. Different user-interface applications can be used, so long as the same protocol is used to interact with the Workflow Service (Figure 1).

The workflow engine, which enacts the workflow, is itself a Web Service, provides the mechanism to recursively encapsulate workflows within other workflows and can itself can be called by the workflow.

Much depends on the workflow specification language which expresses the workflow structure, dependencies and concurrency. Its design and constituent elements are described in the next section.

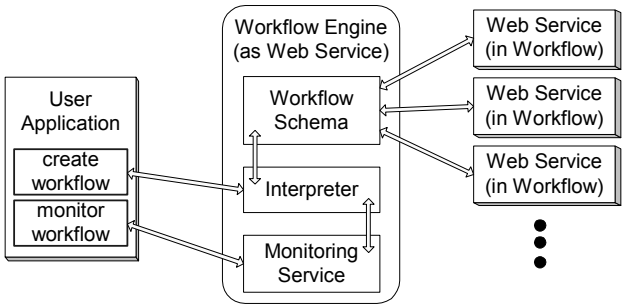

Figure 1 Overall System Architecture

4. Workflow Language

The workflow language is designed to be used on both the workflow engine and in the user application with minimal modifications. The language caters for sequencing of web service invocations, parallel execution, choice and synchronisation mechanisms. The design of the language emphasises its flexibility and reusability. The main motivation is to be able to extend the workflow language with new primitives with minimal impact on the existing ones.

The language is based on the principles of Java Beans, self-contained reusable software components. Each expression in the language is a Bean definition with properties that can be set to appropriate values. For example, an instance of a composition expression will be an instance of a Bean with its properties set to the constituent expressions that are composed.

4.1 Language design and dialects

The starting point of the language is the Interpreter Pattern [12]. However, the traditional implementation of the pattern defines an Abstract Expression at the highest level of the hierarchy, descending into Terminal and Non-Terminal Expressions. The top-level expression of the language (e.g. 'class' or 'module') will usually then descend from the *Non-Terminal* expression. However, in our implementation the top-level node, a *WorkflowService*, extends directly from the Abstract Expression (Figure 2). This design permits the implementation, of more than one language-type, or dialect, by providing different descendents to the abstract *WorkflowService* class. These dialects can then share some of the expressions within the Terminal and Non-Terminal descendants (Figure 2).

In the current implementation there are two different descendents of the WorkflowService:

- *SafeFlow* which provides a structured approach to workflow design, with tight execution control.
- *CircusFlow* which is a dialect concerned with information flow rather than execution control.

The *WorkflowService* class acts as an interface or marker class for both, rather than implementing specific functionality. Although the model suggests that all non-terminals are shared between the two dialects the user application will restrict each dialect to the appropriate subset of what is available.

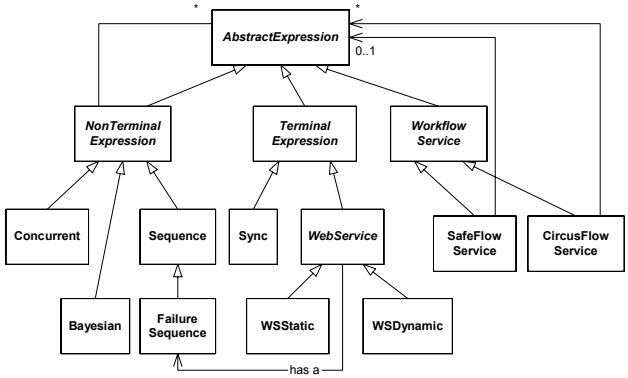

Figure 2 Language Design

In addition to *Concurrent* and *Sequence* composition the non-terminals include *FailureSequence* and *Bayesian*. The former is a mechanism for dealing with failures, while the latter is a complex choice mechanism based on probabilistic inference. If an invoked Web Service fails, *FailureSequence* defines the actions necessary to overcome the failure. The terminals are either *Sync*, for synchronisation, or *Web Service* (see Section 4.2).

The enactment of *SafeFlow* and *CircusFlow* schemas, by their respective interpreters on the Workflow Engine, is different, particularly in terms of concurrency. The SafeFlow dialect provides explicit control of concurrency through the *Concurrent* and *Sequence* non-terminals. All non-terminals are mapped into components and thus, a *Concurrent* expression is implemented as a component which contains only sub-expressions that will be executed simultaneously. All sub-expressions in concurrent expressions must terminate before computation can proceed. Similarly, the *Sequence* expression component will contain only sub-expressions executed in sequence. Note, that the above is more restrictive that general workflow expressions based on arbitrary graphs. In particular, an expression such as the one shown in Figure 3 cannot be represented.

However, in SafeFlow, expressions, particularly Non-Terminals, can hide all their internal workings from all other external expressions. The *AbstractExpression*, from which all expressions descend, provides the common means for one expression to interact with any other expression, in a black-box way, without knowing what the

expression is. Thus, Workflow Services themselves can be encapsulated and reused in other workflow schemas.

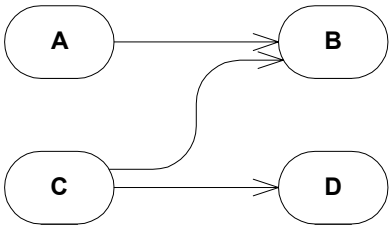

Figure 3 Graph prevented by strict encapsulation

Note that control in SafeFlow departs from traditional workflows such as OTSArjuna or WSFL. While in OTSArjuna or WSFL control is implicit and described by lines in SafeFlow all control is explicit and determined by component types and their encapsulation. Although OTSArjuna allows encapsulation of workflow as intermediate composite tasks, these do not represent control but denote perimeter sub-graphs of control and data-lines. To our knowledge none of the current available ORB workflow languages use the structured Interpreter Pattern style approach to their design, which is largely exploited in SafeFlow, and prefer a more scripting oriented approach. Even when some means of workflow composition are integrated in to these languages, control, by and large, remains unstructured.

The *CircusFlow* dialect adopts a more liberal approach disregarding control flow over information flow. Thus, in CircusFlow, all constituent expressions of a workflow are executed as soon as the data representing their input parameters is available. The *sync* terminal expression provides explicit synchronisation when needed. Note, that neither the Concurrent nor the Sequence non-terminals are used in CircusFlow since these provide control specifications. The idea behind CircusFlow can also be encountered in *data-flow* based computing, primarily used in multithreaded execution, signal processing and reconfigurable computing. OTSArjuna can also be seen as similar to a certain extent. Although, in OTSArjuna control and data flow are mixed, with control implemented as notification requests. Control and data are then implicitly synchronised at each task or composite task.

CircusFlow provides a high degree of concurrency and overcomes the limitation identified for SafeFlow. However, this is at the expense of control which is entirely omitted. Additionally, recursive encapsulation is not possible. Note however that SafeFlow and CircusFlow can be used in conjunction to overcome the problems of both. In particular, web services deployed using CircusFlow can be included in a SafeFlow and vice-versa.

4.2 Terminals

WebService expressions (see Figure 2) are Beans which contain the fields necessary to invoke a Web Service namely: the URL of the Web Service, the particular Service Name, the Method Name to be called, the HTTP SOAP Action (header information on call intent) and the XML Namespace defining the encoding style of the call. With WebService expressions, *import* parameters are cast into Apache SOAP parameters and sent to the Web Service. After the invocation, the returned parameters are converted to *export* parameters (see Section 4.3). The *WebService* expression is an abstract class deferring instantiation to a more specialised expression which can be one of the following:

- *WSStatic*, used when the details of the Web Service are defined at design time, as fields.
- *WSDynamic*, used when the details of the Web Service are passed as special ImportParameters at runtime using the names '#soapend', '#nameuri', '#method', '#encode' and '#action'.

Sync Beans are markers for synchronisation in CircusFlow interpreters, and do not have specific data.

4.3 Guards and Parameters

Interactions between expressions in the workflow language are achieved through *import* and *export* parameters. Both types of parameters are implemented as a specialisation of the *Parameter* bean defined in the Apache SOAP implementation. Each parameter is characterised by its name, value, type and XML namespace to which it belongs. In addition, parameters have a *Reference* field that specifies from which neighbouring or other expression the value for this parameter can be derived from. Only one of the Reference or Value fields will usually have an assignment.

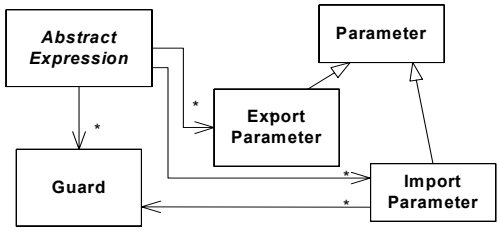

Figure 4 Expressions, parameters and guards

The ability to express *choice* in workflow languages is important in order to provide multiple alternatives in different sets of circumstances. In our language choice is expressed through the more general mechanism of *guards*. In addition to the parameters each *AbstractExpression* also maintains a list of *guards* and the expression is evaluated only if all the guards are satisfied. Each *guard* is

```
<SafeFlowService id="somewsid">
  <ImportParam name="sourcedata" value="" type="String" encodingStyleURI="" ref="#import/data" />
  <Sequence id="someschema">
       <ImportParam name="sourcedata" value="" type="String" encodingStyleURI=""
                          ref="#import/sourcedata" />
       <WSStatic id="someproxy" soapend="http://services.xmethods.net:80/perl/soaplite.cgi"
                          nameuri="urn:xmethodsBabelFish" method="BabelFish" encode="">
            <ImportParam name="translationmode" value="en_de" type="String" ref="" />
            <ImportParam name="sourcedata" value="" type="String" ref="#import/sourcedata" />
            <ExportParam name="return" value="" type="String" encodingStyleURI="" ref="#details/return" />
       </WSStatic>
       <WSStatic id="someproxy2" soapend="http://services.xmethods.net:80/perl/soaplite.cgi"
                 nameuri="urn:xmethodsBabelFish" method="BabelFish" encode="">
            <ImportParam name="translationmode" value="de_fr" type="String" ref="" />
            <ImportParam name="sourcedata" value="" type="String" ref="#someproxy/return" />
            <ExportParam name="return" value="" type="String" encodingStyleURI="" ref="#details/return" />
       </WSStatic>
         <ExportParam name="return" val="" type="String" encodingStyleURI="" ref="someproxy/return" />
         <ExportParam name="return2" val="" type="String" encodingStyleURI="" ref="someproxy2/return" />
  </ Sequence >
  <ExportParam name="return" val="" type="String" encodingStyleURI="" ref="someschema/return" />
  <ExportParam name="return2" val="" type="String" encodingStyleURI="" ref="someschema/return2" />
</SafeFlowService>
```

Example 1 Representing a SafeFlow schema in XML

associated with an *import* parameter, which can have several guards (Figure 4).

4.4 The Role of Data Binding

In terms of internal representation, the language is based on Java Beans that encapsulate data only. Decorators and visitors are then used to add functionality to the Beans as required by the program using them.

The transport used between the workflow engine and the user application (Figure 1) is an XML representation that has a direct one-to-one mapping with the Bean representation. This tight mapping enables the conversion process between Beans and XML to be independent of the Beans and XML themselves; it describes only how to map any of the language's Bean to an equivalent XML form and vice-versa. The language can therefore be extended by simple addition of Beans, without modifying the mapping to and from the transport. In essence, this is a custom and specialised implementation of Sun's Java-XML Data Binding [13], which was not yet available at the time of the implementation.

4.5 XML and Graphical Specification

The graphical representation of the language (also inspired from the graphical representation of component-based systems) has a direct one-to-one mapping to the XML representation. Each workflow expression is represented by a box where the import parameters are represented on the left hand side and the output parameters on the right hand side. Figure 5 gives the graphical representation for the workflow described in more detail in the Example 1 below.

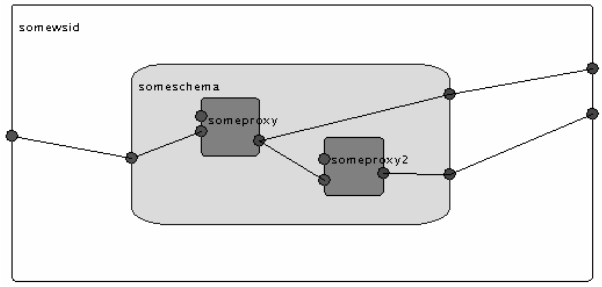

Figure 5 Visual Representation for SafeFlow

Example 1 below gives an outline of an XML schema for a SafeFlow Service with a Sequence expression that has two Static Web Services. The second bolded text import parameter refers to the parameter imported from the outer expression, using the reserved *#import* directive, and the name of the parameter, *sourcedata*. The first highlighted import parameter follows a similar convention except that *data* refers to the name of the parameter passed in at runtime when the method for the workflow is called on the Web Service Engine. Export parameters use a different referencing convention. The export parameter for *someproxy* refers to the parameter called *return* returned from the Web Service. This return value is accessed using the reserved *#details*. The import parameter for *someproxy2* refers to the export parameter from *someproxy*. The export parameters for the Sequence and Workflow Service refer by convention to export parameters from their constituent expressions. The

export parameters for the SafeFlow Service, *return* and *return2* near the bottom, refer to the workflow results.

5. Workflow Engine

The workflow engine encapsulates the functionality of the workflow enactment and permits the encapsulated system to be easily accessible while respecting low coupling with clients. It also permits the deployment of several workflow schemas which are enacted through delegation to the appropriate workflow.

The WorkflowEngine is deployed as a Web Service, and allows for several WorkflowServices to be deployed on it. Each WorkflowService is an instance of a workflow schema and is associated with an Interpreter (which may create further child interpreters) to enact it. Clients (including the user interface) can create several WorkflowServices corresponding to possibly different schemas and deploy them on the WorkflowEngine for enactment. The WorkflowEngine is installed as a Web Service on a server supporting a SOAP implementation that allows for the deployment of SOAP services – in our case a servlet that allows for the invocation of the Java Beans that it holds.

As shown in Figure 6, the WorkflowEngine is implemented as a single Java Bean with static methods and a persistent Hashtable that maintains the services. Each deployed schema behaves like a new method of the WorkflowEngine which can be invoked with parameters that become ImportParamters during enactment. The WorkflowEngine bean provides the methods necessary to:

- *Add* a new Workflow schema by providing an XML encoding of the schema.
- *Remove* a Workflow schema.
- *Run* a Workflow schema. An XML String version of an *EMethod* object is passed to the *enact* method. After parsing the string to an internal representation, the *enact* method retrieves the service name and passes the *EMethod* object as a set of import parameters to it. The set of export parameters returned from the WorkflowService is converted to an XML version of a new *EMethod* object and returned to the client.
- *List* the WorkflowServices deployed, together with their required parameters.
- *Enact* a workflow service. This method is similar to the *Run* method, but is used only by clients capable of using the monitoring protocol.
- *Update* returns monitoring information on the current state of a WorkflowService that is being enacted. When the WorkflowService has finished executing, this method returns the final result. SOAP over HTTP does not allow for call-backs, so relaying of calls to Update is necessary.

Note that more complex management of the Workflow Schema is possible by combining these operations in a Web Service that acts as a wrapper for the WorkflowEngine Web Service. Although alternatives to the use of the *EMethod* object were investigated, for example by dynamically adding a new method to the workflow engine for each schema, this would have required stopping and restarting the server, which would have been unacceptable.

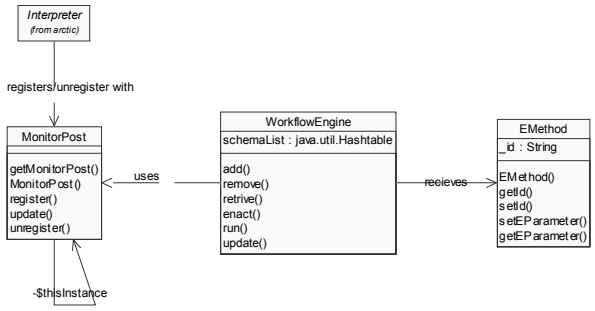

Figure 6 Workflow Engine Bean

Update calls are made on a shared *MonitorPost* component implemented as a singleton. When entering an AbstractExpression, interpreter instances enacting the workflow register with the *MonitorPost* the identity of the AbstractExpression they have entered, together with the workflow schema. When the enactment is finished the AbstractExpression is unregistered. When an update call comes from the WorkflowEngine, with a specified workflow schema, the MonitorPost returns an EMethod object identifying a list of active AbstractExpressions for the schema. This system for monitoring the workflow enactment is simple, but provides all the necessary functionality. However, the singleton instance means that the system is constrained in terms of scalability and a more complicated implementation using Publisher-Subscribers is desirable in future developments.

6. Interpreter – A Concurrent Visitor

Once the schema is converted from XML to the internal representation, the enactment of the workflow is performed by an Interpreter, implemented as a Visitor that traverses the hierarchy of Beans which forms the internal representation of the workflow, as described in the Visitor Pattern [12]. This allows the encapsulation of interpretation control and co-ordination in one logical object and avoids placing interpretation code across the different expression beans, thus making future modifications difficult because of interdependencies.

However, the implementation of the Visitor Pattern is modified, in that each time a new bean is visited, a new

instance of the interpreter evaluates it rather than the one evaluating the current bean. This design decision was made in order to: (*i*) provide support for concurrent expressions, which require several threads of execution, (*ii*) provide "natural" concurrency within the design and (*iii)* cater for several instantiations of the workflow by different clients, all represented by their own sets of Interpreters. In this way, the information held in the Bean, during interpretation is not changed and the interpreter holds temporary information derived from processing the Bean. If instead a single instance of an Interpreter were to visit all the different Beans, it would be necessary to use a stack to hold temporary information that must be saved before traversal returns to the Expression where that temporary information was needed.

When the workflow is invoked in a WorkflowService, such as SafeFlowService or CircusFlowService, a new instance of the Interpreter is created for the first time, and the *accept* function of the top-level WorkflowService schema Bean is passed the instance. Within the Bean, the Interpreter has its visit function invoked with the Bean itself, passed as a parameter (Figure 7).

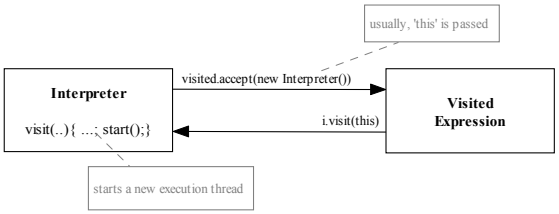

Figure 7 Interpreter and Visited Exrpression

While interpreting a parent expression that contains several child expressions, an interpreter will be created for each child expression. The reference to the child interpreter will be passed to the child expression through its *accept* method which in turn will call the child's interpreter *visit* method with the child expression bean as parameter. The visit method will call the *start* method, thus forking execution and then an *interpret* method which is overloaded for the different types of expression. Once execution has forked, the parent's execution thread will be put to sleep. In case of a sequence expression the parent interpreter will sleep until the child interpreter finishes and then start visitation of the following child. In case of a concurrent expression the visitation of the other child expressions will start immediately. All instances of child interpreters are grouped in a Thread Group, and when the Thread Group indicates that all its threads are dead and all the data from their enactment accumulated, the original execution of the hosting Interpreter continues, effectively synchronising all constituent Expressions. In the case of CircusFlow, if any of the child expressions has sufficient data to execute the visitation process starts for it. All child expressions are checked each time a child interpreter finishes and produces additional data.

According to this enactment pattern several concurrently acting interpreters can be spawned. However the Interpreters also need to communicate to each other the import parameters and export parameters as they become available. This was achieved by using a double dispatched Publisher-Subscriber version of the Observer pattern between the visitors.

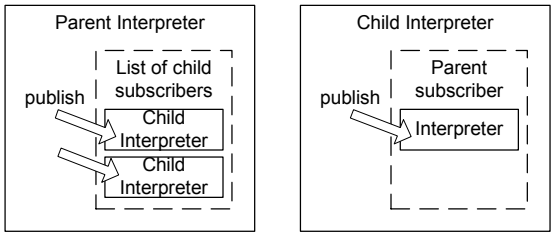

Figure 8 Publish-Subscribe coordination

The Parent Interpreter is registered with the Child Interpreters, and the Child is also registered with the Parent (Figure 8). The Parent publishes to its Children the ImportParameter data necessary for a Child to execute. The method the Child listens on (i.e. the method invoked by the Parent) is protected as a synchronised method, for concurrency purposes. Once a Child has finished interpreting, it publishes the ExportParameter data to the Parent. The actual co-ordination of information is dependent on the nature of the expression. In concurrent expressions (SafeFlow) the parent publishes its own ExportParameters only once all the Children have published their ExportParameters. In sequence expressions the ExportParameters of a child are made available to the next child through the parent. In CircusFlow export parameters are published to the next children as soon as they are available. Finally, the Parent can publish any data it has to any of its Parents.

7. The User Interface

Although XML can be used for workflow schema specification, XML is far from being a user-friendly language. A user interface was therefore developed which permits workflow specification, workflow enactment and monitoring. The user interface follows a similar approach to that used in component environments such as Darwin [14] and in many respects it was designed to resemble an Integrated Development Environment. The hierarchical language data structure allows to maintain a tight integration between the graphical specification and its XML textual description, thus allowing knowledgeable users to manipulate text directly.

The user interface comprises: a top window defining the workspace area, one or several graphical composition windows, a property window and a monitor browser (Figure 9). The graphical composition windows permit a top-down workflow specification by allowing various language constructs to be selected from a toolbar, drawn on the canvas and then linked to the components already present. The properties window displays the properties of the current selected element. The Monitor Browser permits workflow deployment (and retraction) on a workflow engine, and workflow invocation. When the Monitor Browser is in use the graphical composition window is used to display workflow execution state by highlighting the elements currently being enacted.

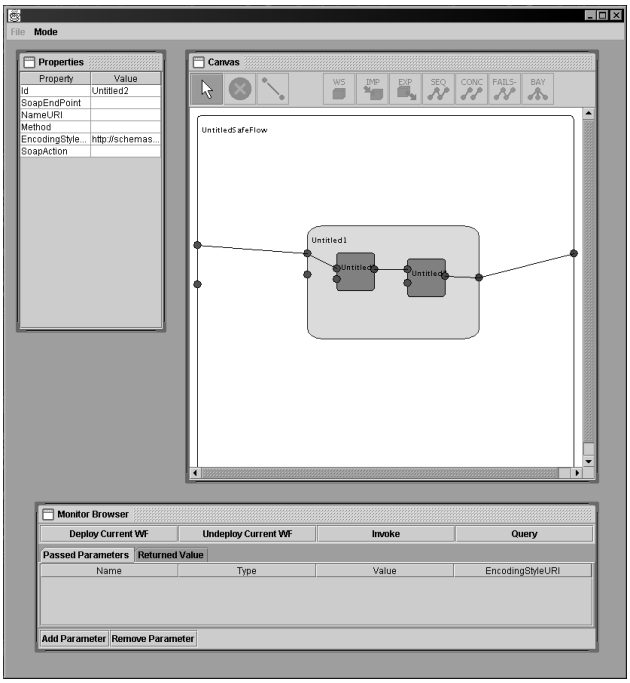

Figure 9 Graphical specification tool

The top-down approach to workflow specification was inspired by the B Method which describes a process of progressive refinement of a formal specification into more concrete descriptions. At the lowest level, implementations such as pre-built machines can be substituted to the components. This is not unlike our environment where existing Web Services behave like pre-built machines that provide the leaf nodes in the hierarchical design. Thus, the workflow can be designed at a higher level and decomposed into Non-Terminals and predefined workflows that can be Web Services. This can be a recursive process.

Several types of objects need to be drawn on the canvas, ranging from AbsractExpressions like Web Services, which have no constituent expressions, to ImportParameters, which do not descend from AbsractExpression, but are distinct fields of AbstractExpressions, and need to be drawn as well. The existing data structure of beans used for the language is therefore decorated with the necessary graphical information. However, this is not straightforward. Although, a single Decorator Class, for AbstractExpressions or EParameter, can be defined, it is also necessary to traverse the hierarchy of language Beans during graphical operations, for example to add constituent expressions to the right Non-Terminals. While a Decorator can refer to a language Bean, and the language Bean to its constituents, it is also necessary to derive the Decorator for the constituents as well, for example when resizing an inner expression. To achieve this the Visitor Pattern style double dispatching was applied between each object that can be drawn on the canvas and the Decorator Class that contains that object's graphic information. AbstractExpressions and EParameters implement the *Drawable* interface which requires that each implementing Bean provide methods for getting and setting a VisualExpr decorator associated with it. The VisualExpr decorator manages the graphical and language data associated with each *Drawable* and becomes the single point of contact for manipulating language or graphical data associated with it. The *VisualExpr* delegates all functionality such as expression formatting to other classes which are therefore isolated from the data elements and can be changed independently. Amongst the different delegated decorator classes *VisualExprLogic* is used to manipulate the hierarchy of expressions while enforcing some of the language restrictions e.g., FailureSeqs can only be added to Web Services, Terminals can only be added to Non-Terminals, etc. Another, the *VisualExprFormatter,* caters for the visual manipulation of expressions on the canvas enforcing graphical constraints e.g., parameters remain on the expression boundary and expressions cannot move out of parent expressions.

By taking advantage of the graphical representation already used, it is possible to provide a simple way to observe the enactment of the workflow, and its results. In the same way as a train monitoring system shows the progress of the trains between departure and destination, the enactment of a workflow can be shown on the canvas on which the workflow is drawn by highlighting the elements currently active. Note that while in '*Monitoring'* mode the canvas cannot be simultaneously used for editing. The output of the workflow itself is displayed in a form similar to a web-browser integrated in the monitor browser component. The monitor browser also integrates the functionality of a MonitorManager component which caters for the deployment, revocation and invocation of the workflow. While the workflow is executing, the

monitor manager receives continuous updates on its execution state, which can be passed onto the graphical display window.

8. Discussion

When applied to Web Service orchestration the fundamental limitation of current workflow languages is that they largely approach workflow definition as a graph problem; nodes that perform work and lines of control between them that describe how execution flows. This means that in large workflows, managing all the different nodes can become difficult as the workflow definition graphically starts to look like a complex spider diagram. Within the context of distributed systems, OTSArjuna, METEOR2, RainMan and WSFL, which are probably some of the most significant references for comparison, are to an extent graph-based.

SafeFlow's distinguishing characteristic is that it takes a component-style programmatic approach to workflow design. Whereas all the above-mentioned languages approach workflow as a graph problem, SafeFlow is motivated by encapsulation and tight control over the enactment of the workflow. This means that control is considered visually as a box with nodes in it, rather than as a line between two nodes. The exact nature of the control, such as concurrency or sequence is dependent on the type of box used. Boxes can be nested and provide encapsulation by hiding the inner workings of a particular box, which will represent a sub-workflow. This structured approach allows the workflow definition to be made in a similar way to component composition where a higher level component hides the lower level details of the components it contains. At a lower level, within a box, changes will impact only the other components within that box. With a graph-based approach, manipulating whole sub-workflows at a high level is difficult because there may be many tangled inter-dependencies. Changing the workflow at a low level is also difficult, as the workflow designer needs to be aware of dependencies of a particular node across the entire workflow, rather than just in the local vicinity. The encapsulating component-style composition approach to workflow design is largely due to the language's origins from the Interpreter Pattern - an approach seemingly not considered in the other workflow languages.

WSFL and OTSArjuna are the strongest related to graph problems. The former lays control out as a graph and overlays a data flow graph above it that respects the control graph. OTSArjuna, $METEOR_2$ and RainMan mix both control and data flow in the same graph. Although Composite Tasks (OTSArjuna), which define sub-graphs that can be represented as nodes, can be used as a structuring mechanism, the underlying approach remains graph-based. Graph-based approaches cause difficulties in the context of maintainability and problem structuring – a graph of control lines is analogous to 'goto' statements. In contrast, the argument for representing control as a graph is that of increased flexibility and expressiveness.

SafeFlow uses recursively encapsulated control boxes, which dictate the control within the box. The significant advantage of such an approach is that any encapsulating box can be easily removed and replaced without effect to the rest of the workflow. Furthermore, SafeFlow allows for high scalability, due to the natural presence of recursive encapsulation. Thousands of nodes of Web Services, or other distributed component, could be organised effectively. Subsections within a workflow can also be effectively reused, or the entire workflow itself.

SafeFlow has some synchronisation limitations owing to the strict encapsulation. We have shown how these limitations can be overcome by using an unstructured workflow (CircusFlow) encapsulated in a web service and used within the context of the SafeFlow structure. While in essence, this may seem a way of circumventing the problem, it contains the "uncontrolled" part of the workflow in a strictly encapsulated manner.

The entire system described in this paper was implemented and tested for varying case scenarios. However, there are few freely available web-services, which provide meaningful services that can be composed in realistic experiments of large scale. If the momentum gathered towards the Services Web environment is sustained, we will be in a better position to test the framework in larger scale environments.

9. Conclusions and Future Work

Workflow Management Systems have been traditionally difficult to manage and evolve according to changing business requirements. This problem will acquire an entire new dimension in a Services Web environment if significant parts of the business model are based on the aggregation of existing services. Changes in business requirements, availability of suppliers, personalisation of services, mergers with other businesses and acquisitions are only few amongst the factors that will require changes in the workflows used. In the absence of a structured environment providing strict encapsulation, the impact of change will often be unforeseen and sometimes unforeseeable. Workflows originated in an organisational management background and typically lack the structures and models that programming languages have evolved to. This work has investigated adding structure and approaching workflow as a programming problem; the SafeFlow language described here can be reasoned for workflow properties.

Workflow encapsulation is a powerful structuring mechanism, which provides a way of managing complexity that would otherwise be difficult with standard approaches. Because each constituent part in a workflow can be closed as a black box, either using a Non-Terminal expression or by outsourcing to another Web Service standard refinement and decomposition techniques can be applied.

When the implementation of the basic framework was completed, our programming landscape had somewhat changed. Far from fuelling the component/web-service debate [15],[16], the framework emphasised some of the analogies. By using a structured (component-based) approach to workflow specification and workflow as a constrained form of programming it was possible to build new components and deploy them in a uniform way. The framework's usability, greatly helped by the graphical specification tool, constituted one of the main temptations. In essence, it became easy to aggregate, deploy and re-use web-services at will to build potentially large-scale systems. Is this approach suitable for application development in general, thus providing a new programming paradigm? Workflows however, are not a general-purpose programming tool, and in any realistic setting it is unlikely that all of the required functionality of a new service or application can be found by simply composing existing web-services.

The environment provides a high degree of concurrency as multiple workflows and web-services can be executed concurrently. However, intensive network use and remote communications increase the delay experienced substantially. These performance considerations impact the granularity at which services can be composed and restrict the settings in which this approach can be adopted. However, it is relatively easy to redistribute coordination work by redeploying the workflows "closer" to the underlying services they use, thus minimising the impact of those communications with large delays.

The framework described in this paper constitutes a first step, and many improvements remain to be made both in the workflow language and in the implementation. In particular, the extensibility of the workflow language has not been exploited and we intend to develop additional language elements and dialects. For example, a new dialect could combine aspects of both SafeFlow and CircusFlow. In this model, Non-Terminal control expressions do not synchronise data as it enters and leaves the expression boundaries. In Concurrent Expressions, constituent Web Service expressions that have sufficient data to enact will do so, and for any other inner Non-Terminal Expressions data will be passed straight through the boundaries of that expression to any Web Services that need it. Concurrent Expressions are similar to CircusFlow, but the boundaries for intermediate Non-Terminal composites like Concurrent / Sequence are not synchronising. This creates a workflow model that is like a graph of data, but where sub-graphs can be perimetered into sections. The flexibility of the sub-graph does not change, though this allows the sub-graph to be encapsulated, and replaced with sub-graphs with the same endpoints. Sequence expressions force ordering on the constituent expressions.

The specification toolset also needs additional improvements particularly for the manipulation of visual expressions and to provide better support for debugging.

Business in an open environment raises issues relating to the reliability, availability, and quality of service provided by the services. These issues give rise to contracts, trust models and preferences, which not only complicate the model but also require greater adaptability, tolerance to failures and performance deteriorations. Further work is needed in order to provide support for such scenarios within our framework.

Historically, workflow originates from business and management as a way of modelling business processes that could wholly or partially be automated. However, the graph based models used have largely not evolved. Programming is similar to the methods of describing workflow, but has evolved incredibly to encapsulate complexity and allow for greater manageability and maintainability. This paper describes how adopting some of the lessons learnt from programming can improve business modelling using workflow.

10. References

[1] T. Kindberg and J. Barton. 'A Web-based Nomadic Computing System', Computer Networks, 35(4):443-456, March 2001.

[2] F. Leymann, 'Web Services Flow Language'. IBM Software Group specification, May 2001. http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf)

[3] 'Open Net Environment (ONE) Software Architecture', Sun Microsystems, http://www.sun.com/software/sunone/, 2001.

[4] M. Kirtland, 'The Programmable Web: Web Services Provides Building Blocks for the Microsoft .NET Framework' MSDC magazine, Sept. 2000. available at:http://msdn.microsoft.com/msdnmag/issues/0900/WebPlatform/WebPlatform.asp

[5] Hewlett-Packard. 'E-Speak Architecture Specification, version 2.2, 1999. see also www.e-speak.net.

[6] D. Hollingsworth, 'Workflow Management Coalition The Workflow Reference Model' (1995), http://www.wfmc.org/standards/docs/tc003v11.pdf

[7] Frédéric Ranno, Santosh K. Shrivastava, Stuart M. Wheater, 'A Language for Specifying the Composition of Reliable Distributed Applications', Proc. 18th Int. Conf. on

Distributed Computing Systems (ICDCS '98), Amsterdam, The Netherlands, May 26 - 29, 1998.

[8] A. Sheth and K. J. Kochut. Workflow Application to Research Agenda: Scalable and Dynamic Work Coordination and Collaboration Systems. In Workflow Management and Interoperability. A. Dogac et al. (eds.). Springer Verlag, 1999, pp. 35-59. also see: http://lsdis.cs.uga.edu/proj/meteor/meteor.html

[9] S. Paul, E. Park, and J. Chaar, 'RainMan: A Workflow System for the Internet', Proc. USENIX Symp. on Internet Technologies and Systems, December 8-11, 1997, Monterey, California.

[10] Object Management Group. 'Workflow Management Specification v1.2', available from: www.omg.org

[11] BizTalk Orchestration White Paper (July 1999), Microsoft,http://www.microsoft.com/biztalk/techinfo/biztalkorchestration.htm

[12] E. Gamma, R. Helm, R. Johnson and J. Vlissides. 'Design Patterns', Addison Wesley Longman Publishing, 1994

[13] M. Reinhold, 'An XML Data-Binding Facility for the Java Platform' (30 July 1999), Core Java Platform Group Java Software, Sun Microsystems, Inc., http://java.sun.com/xml/jaxp-docs-1.0.1/docs/bind.pdf

[14] Ng, K., Kramer, J. and Magee, J., 'Automated Support for the Design of Distributed Software Architectures', Journal of Automated Software Engineering (JASE), 3 (3/4), Special Issue on CASE-95, (1996), pp. 261-284.

[15] C. Szyperski. 'Components and Web Services', Software Development Magazine, August 2001. available from: www.sdmagazine.com

[16] B. Meyer. 'Product or Service'. Software Development Magazine, October 2001. available from: www.sdmagazine.com