Preprint version of an article published as: Journal of Systems Architecture, Volume 50, Issue 6, June 2004, Pages 309-324, DOI:10.1016/j.sysarc.2003.09.004, Copyright © 2003 Elsevier B.V.

Developing a Reusable Workflow Engine

Diogo M. R. Ferreira[1], J. J. Pinto Ferreira[1,2]

[1]INESC Porto, Campus da FEUP, Rua Dr. Roberto Frias, nº 378, 4200-465 Porto, dmf@inescporto.pt
[2]Faculty of Engineering U. P., Rua Dr. Roberto Frias s/n, 4200-465 Porto, Portugal, jjpf@fe.up.pt

Keywords Workflow Management Systems, Workflow Engines, Component Reuse

Abstract Every time a workflow solution is conceived there is a large amount of functionality that is eventually reinvented and redeveloped from scratch. Workflow management systems from academia to the commercial arena exhibit a myriad of approaches having as much in common as in contrast with each other. Efforts in standardizing a workflow reference model and the gradual endorsement of those standards have also not precluded developers from designing workflow systems tailored to specific user needs. This article is written in the belief that an appropriate set of common workflow functionality can be abstracted and reused in forthcoming systems or embedded in applications intended to become workflow-enabled. Specific requirements and a prototype implementation of such functionality, named Workflow Kernel, are discussed.

1. Introduction

The Workflow Reference Model proposed by the Workflow Management Coalition [17] defines a framework for relating workflow management systems and their capabilities. Within this framework, supporting tools, execution services, client applications and external applications interact according to a set of interfaces. At the heart of this framework is the workflow enactment service, an execution service comprising one or more workflow engines. According to the reference model, a workflow engine is "a software service that provides the run time execution environment for a process instance" [17].

In this sense every workflow product, prototype or approach entails a workflow engine in one way or another. Interpreting a process definition, creating process instances from those definitions, and managing the execution of those instances are essential chores of every workflow management system. These capabilities represent functionality that is coded and embedded in every workflow solution. Notwithstanding, workflow systems are usually portrayed by supporting tools such as process editors or audit trail viewers. The important workflow functionality, however, is the one creating and managing the execution of process instances by iterating through individual tasks and triggering the appropriate actions.

Up to now, this functionality has been typically implemented over and over again as each workflow management system is developed. Existing standards or common views such as the ones proposed by the WfMC could lay down the guidelines for implementing workflow engines. And, to some extent, they do. But, as pointed out by [15], existing standards focus on the syntax of the reference model interfaces without clearly specifying the respective semantics and usage. Therefore, when confronted with specific user needs, developers often make use of standards according to their own interpretation. The previous authors go even further and compare the present situation with the early days of database management when,

in the absence of the relational and entity-relationship models, an incongruous set of database solutions coexisted.

In this respect, this article argues, as other authors have done, that Petri Net theory could become to workflow management what the relational model became to database management. It is also argued that a reasonable amount of common workflow functionality can be abstracted from existing approaches, and that from this abstraction it is possible to develop a Workflow Kernel that can be reused and embedded in workflow-enabled systems and applications, so as to prevent repeated and discordant implementations of general workflow features.

2. Common approaches and reusability

Successful workflow products such as Staffware™, InConcert™, or FlowMark® are an elaborate compound of user requirements. Throughout the years these and other leading products have been improved in order to fulfill or anticipate particular user needs. But although they share the common goal of business process integration, each product displays its own philosophy and approach. From the event-driven process chains (EPC) of ARIS® Toolset [12] to the four-stage workflow loop of ActionWorkflow™ [8] there are several approaches to workflow management. Many workflow solutions are built on a set of constructs that are believed to be appropriate to describe business processes. Staffware™ has its own constructs for modeling business processes, InConcert™ has another set of constructs, and ARIS® and ActionWorkflow™ have their own constructs too.

Furthermore, every product provides a process editor to graphically define or modify processes, provides support for monitoring process instances, and client applications to manage individual tasks. These support tools, which are the front-end of the workflow solution, are the functionality most seen by the user, and take a significant effort to develop. Still, behind this front-end each one of these solutions contains an engine capable of interpreting the process definition language, creating process instances from process definitions, and controlling the execution of these instances. Regardless of product philosophy, what lies in the heart of each workflow solution is an engine that glues everything together and makes the desired orchestration possible.

The WfMC's Workflow Reference Model describes the purpose of a workflow engine by a set of features that it is supposed to address. But then, if these features are well known and in fact provided by each workflow solution, why not isolate that functionality in a component that can be plugged in or embedded in every workflow application? Some proposals have been brought up as an answer to this question.

2.1. The Drala Workflow Engine

The Drala Workflow Engine [5] is an embeddable Java component that provides a comprehensive API for defining, executing and monitoring processes. It is not a workflow management system by itself; it is rather a core of workflow functionality which is intended to simplify the implementation of workflow management systems. The Drala Workflow Engine already includes some support tools such as a process editor, but it allows developers to replace these tools or to build additional user interfaces. Process definitions can be imported and exported in an XML format, and the Drala Workflow Engine supports the exchange of XML data between software applications built on top of that engine.

2.2. The Workflow Toolkit

The open-source Workflow Toolkit known as "wftk" [16] is an open-source project that is developing a function library (in ANSI C) to provide Web-based applications with workflow behavior. The wftk toolkit has two main components: the *workflow core*, which controls process and activity execution, and the *repository manager*, which stores and retrieves data for workflow activities. Both of these components rely on *adaptors* in order to interact with external systems, including databases, Web servers and third-party data formats such as process definition languages. Basically, an adaptor translates an external data format into a format that wftk is able to handle. Internally, the wftk manipulates objects represented by XML streams: processes, tasks, users, are all represented as XML strings. For example, a process is represented as a special type of XML format called a *datasheet*, which is a container for other XML objects. A process is also a single-threaded queue of tasks, which waits for the completion of each task to proceed to the next one.

2.3. The WorkMovr API

The WorkMovr API from A-Frame Software [3] is a complete, Java-based programming framework for workflow management systems. In fact, WorkMovr is a workflow management system by itself, and its entire functionality is accessible via external interfaces, which is called the WorkMovr API. The WorkMovr architecture has five layers: the Data Access Layer is the closest to the underlying database system; the Data Management Layer implements enhanced data manipulation routines; the Table API hides the details of the particular database schema, expressing it through a set of high-level Java objects; the Workflow API provides access to the internal workflow engine, which manages work queues; and the Web Management Layer can be used to implement Web-based front-ends. Each layer incorporates and builds upon the functionality of the preceding layer. The key feature is that it is possible to develop external applications that interface with the WorkMovr system directly at any of these layers.

2.4. Fujitsu's i-Flow™

The i-Flow™ from Fujitsu Software Corporation [6] is a Java-based workflow product that promotes reusability of all of its features. The product is sold as a package with source code included, and it can be used as-is (as an out-of-the-box solution), it can be customized to specific user requirements, or it can be used to build new workflow solutions. The i-Flow Server is a workflow engine that interacts with external resources such as file systems, directory services, databases, and e-mail servers by means of application wrappers called *integration adapters*. The engine can also be integrated with existing systems through a scripting language (JavaScript). i-Flow comes with a set of *reference clients*, which provide Web-based GUIs to define processes and administer the server, but it is possible to develop *customized clients* according to specific user needs, either built from scratch using the provided APIs or built as a functionality extension to any of the provided reference clients.

3. Guiding principles

The problem with current approaches towards reusability is that they aim at a full-fledged, all-encompassing set of workflow functionality, including process modeling, application wrappers and client applications. Instead, however, a reusable workflow engine should focus

exclusively on providing process execution capabilities, leaving all of the remaining functionality up to the implementation of a particular workflow management system. This shift towards a narrower focus is illustrated in figure 1, which depicts the Workflow Reference Model as defined by the WfMC. The purpose of developing a reusable workflow engine is to allow subsequent workflow management systems to be developed by plugging the appropriate functionality or components into that engine. From this it follows that a reusable workflow engine must be designed in a way that is independent of (1) particular modeling and monitoring tools, (2) particular ways of interacting with resources, and (3) particular ways of interoperating with other workflow enactment services.

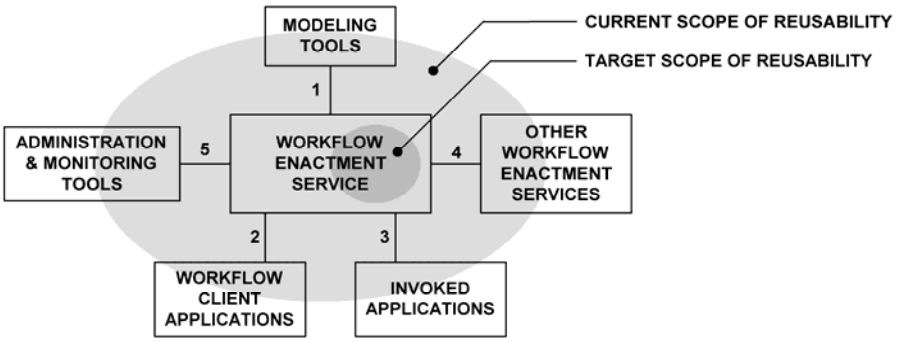

Figure 1. Narrowing the scope of reusability

3.1. The process representation language

Many workflow management systems devise their own process modeling language that seems appropriate for specific user needs, but is not backed by any formalism that can ascertain its properties or suitability. But a workflow engine that is intended to be reusable must make use of formal semantics in order to specify processes unambiguously and consistently, and to allow rigorous workflow analysis techniques to be employed. The main advantage is that if the workflow engine makes use of formal semantics then all workflow management systems based on this workflow engine will be able to rely on those semantics rather than developing proprietary modeling languages. It will then be possible to take advantage of formal analysis techniques to study the behavior of workflow processes. These analysis capabilities may be provided by the workflow engine itself, or by external components.

Realizing the need for a formal language, several authors have encouraged the use of Petri nets [4] for process modeling. The application of Petri nets to workflow management has several advantages [1]:

- Petri nets have formal semantics to describe workflow processes in a clear and precise way, and they also have an intuitive, graphical nature which is easy for end-users to grasp. Petri nets include basic constructs which can be used as building blocks to specify process definitions, and they also include a set of enhancements, such as color and time, which have a formal representation. Furthermore, Petri nets can even be used to define higher-level process modeling languages [2], if necessary.
- Petri nets have a solid mathematical foundation supported by decades of research. Several properties of Petri nets have been investigated, and many analysis techniques are available. These techniques can be employed to check process models for inconsistencies, such as deadlocks or infinite loops, and to compute performance measures, such as execution times and resource utilization. This way it is possible to evaluate alternative process models.
- Petri nets are a vendor-independent formalism to describe and analyze workflow processes. They are not based on a specific product or technology, and they are not affected

by product changes or upgrades. Besides, information on Petri nets is available from independent sources, and research on Petri nets will proceed regardless of any particular workflow management system.

3.2. Resource invocation

Many workflow management systems assume specific ways of interacting with resources and do not consider that, in an enterprise-wide environment, they may be required to interact with a variety of resources, from human to different systems, applications, or even machines. Each resource may require a different kind of interaction, from reply/request to iterated attempts of task acceptance and fulfillment. A reusable workflow engine cannot make no assumptions on the way resources are invoked, or on the way each resource carries out its work. In fact, a reusable workflow engine should not invoke any resource directly; rather, it should provide a mechanism that allows it to interact with any resource. This way it is possible to invoke any number or type of resources without having to make any changes to the way the workflow engine executes processes.

The systems described in section 2 cope with this requirement by introducing the concept of *adaptors* (wftk) or *adapters* (i-Flow™). In essence, these are application wrappers that allow the engine to invoke external components according to an engine-defined interface. However, the concept of adapter assumes that a certain kind of resources will be invoked. In the wftk, for example, the purpose of adaptors is to retrieve information from several kinds of data sources; a different adaptor would be required to dispatch a task to a user's worklist. But a reusable workflow enactment service cannot assume that certain applications are available, or that external programs can be invoked in a certain way, because these details differ from one workflow management system to another. Therefore, a reusable workflow engine requires a more general concept of resource invocation.

We propose that all tasks should be regarded as arbitrary *actions*. These actions are the source of *events*, signaling for example task completion, failure, or timeout. *Actions* produce *events*, which make process execution proceed to the following actions. In Petri Net terminology, events trigger *transitions* between *places*. *Places* are therefore associated with *actions*, representing tasks or process activities, while *transitions* are associated with *events*, standing for progression between consecutive tasks, as illustrated in figure 2. Everything the workflow enactment service should assume is that, when a token arrives on a place, a certain action must be invoked, regardless of what that action will effectively do. This action may dispatch a task to a workflow client application, it may request a remote machine operation, or it may perform some operation on a local database, for example.

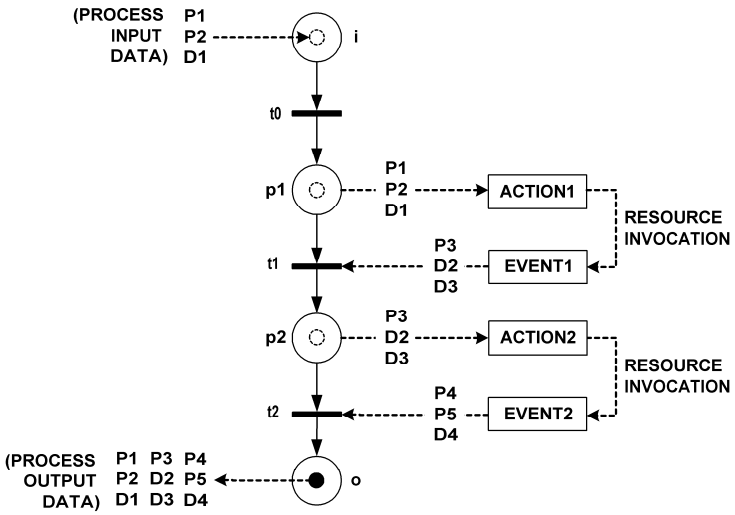

Figure 2. Associating actions with places and events with transitions

The input data for any action, as well the output data brought back to the Workflow Kernel by means of events, are represented as set of *properties* and *documents*. A property is a name-value pair, where name is a string and the value can be of any type. A document is a name-location pair, where location is the fully-qualified path or URL to a file. When the action is complete (for example, when the user has completed the task), an event brings back a number of output properties and documents to the Workflow Kernel. Tokens are the carriers of those information items: they give properties and documents to actions, and take properties and documents from events. Whenever a transition is fired, the new tokens are loaded with the event's properties and documents. A process may also have its own information items: the first token that is inserted into the process carries the process input properties and documents.

3.3. Interoperability with external components

Many workflow systems provide some possibilities of integration with other applications or existing systems but provide no way to extend the functionality of the workflow system itself. In general, the interoperability of workflow management systems is restricted to a set of interfaces, which the system exposes in order to allow external application code to invoke part or all of the system's functionality. This has been the approach within the standardization efforts of the WfMC. The Object Management Group (OMG) has also proposed a standardized interface for workflow management systems, called the *Workflow Management Facility* [11]. The Workflow Management Facility specifies a set of CORBA interfaces to provide access to the runtime environment of a workflow management system.

A reusable workflow engine, however, must be interoperable and extensible. This means that the workflow engine, besides exposing a set of interfaces that allow external application code to invoke its functionality, must also provide a mechanism that allows external application code to be invoked during process execution. In part, the concept of having actions encapsulating resource invocation already addresses this requirement: if an action is defined as an interface IAction that the workflow engine is able to invoke, then any external component that implements this interface can be invoked during process execution. But this solves only part of the problem, since actions are invoked only when a new token is inserted in the corresponding place. Furthermore, due to the long-running nature of workflow activities, another interface, which will be called INotifySink, is required so that actions can notify the workflow engine of events, as illustrated in figure 3(a).

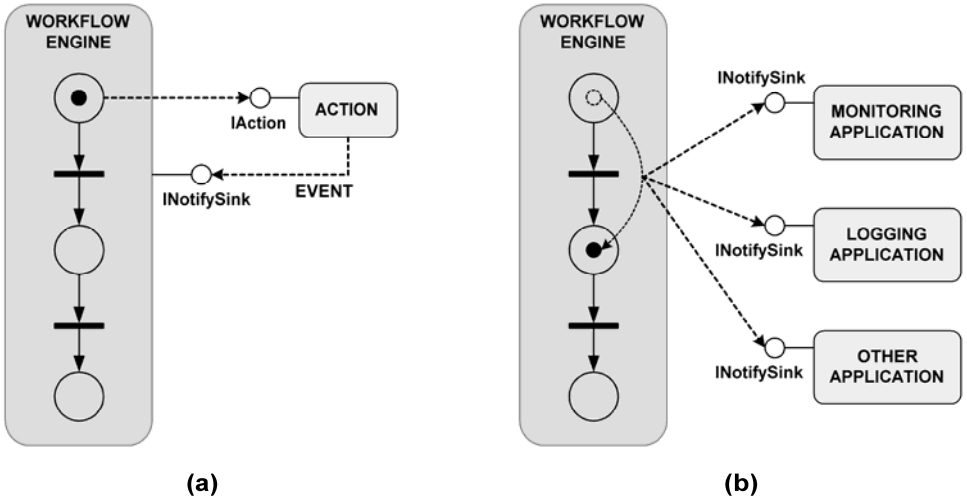

Figure 3. Extending the workflow engine's functionality

In order to extend the functionality of the workflow engine, it may be necessary to invoke external application code, not only when a token is inserted into a place, but in other circumstances as well, for example when a token is removed from a certain place. It should also be possible to invoke external components during process modeling, for example when a new place is added to a process. In this case, the purpose of the external component could be to check for consistency while the process is being modeled. These scenarios suggest that it should be possible to invoke external application code in response to any event, where "event" refers not to action-produced events but to any change that occurs inside the workflow engine. Only then it is possible to develop external modules that react to a set of particular circumstances. This is essential to ensure the reusability of the workflow engine, since a workflow management system based on that engine must be able to blend the reusable workflow engine together with application-specific functionality.

A reusable workflow engine must therefore be able to invoke external components that were unknown at the time when the engine was conceived. The solution is to specify a callback interface that external components must implement in order to be invoked by the workflow engine. In order to avoid defining more than one event notification interface, this callback interface should be the same as that used by actions to communicate events to the workflow engine, i.e., it should be the same as INotifySink, as suggested in figure 3(b). Therefore, INotifySink must be designed in such a way that it is able to convey any kind of change that occurs within the workflow engine. Possible changes thus include (1) insertion and removal of elements such as places, transitions, actions and events, (2) changes to an element's attributes, and (3) triggering of actions and generation of action-specific events. These change events can be identified by means of an enumeration value, which is passed as an input parameter when invoking INotifySink. Any external module or application can listen to changes being made to a given process, as long as it provides a reference to its own INotifySink implementation.

This callback-based architecture allows external applications to react to workflow events, widening the possibilities of extending the engine's functionality. While specifying the Workflow Management Facility, the OMG had realized this same convenience and has proposed a callback interface called WfRequester [11]. Still, the proposed INotifySink callback interface exhibits some advantages over the WfRequester interface. First, an unspecified number of event sinks may be notified, rather than a single WfRequester instance. Second, not only a process instance may generate events, but also every element inside that

process instance may produce events. Third, event types are defined by an enumeration value instead of object type; hence, INotifySink can handle new event types without having to define a new callback interface. The only drawback of this approach is that a small number of external applications may be enough to significantly decrease the engine's performance, as several listeners may have to be notified of each single event.

4. Designing the Workflow Kernel

The *Workflow Kernel* is a prototype for a reusable workflow engine, which essentially comprises a set of objects that are inter-related according to the semantics described in the last section. At the top of the class hierarchy, class Manager provides access to the set of currently active processes, as depicted in figure 4. The Manager object behaves as a *singleton* object [7]: it is the single point of entry for every other application accessing the Workflow Kernel. The Manager object contains a list of references to Process objects. In turn, a Process is comprised of a set of Place and Transition objects that reference each other according to the way the Petri Net is connected. A Place references its input and output Transition objects, while a Transition references its input and output Place objects. A Place also references a set of Token objects if there are any tokens inside that Place. A Place will usually contain an Action, the object that encapsulates the invocation of some resource. On the other hand, a Transition is assigned an Event with a certain event code, meaning that if an Event with such an event code occurs while the transition is enabled, the transition will be fired. Action objects, Event objects and Token objects make use Properties and Documents.

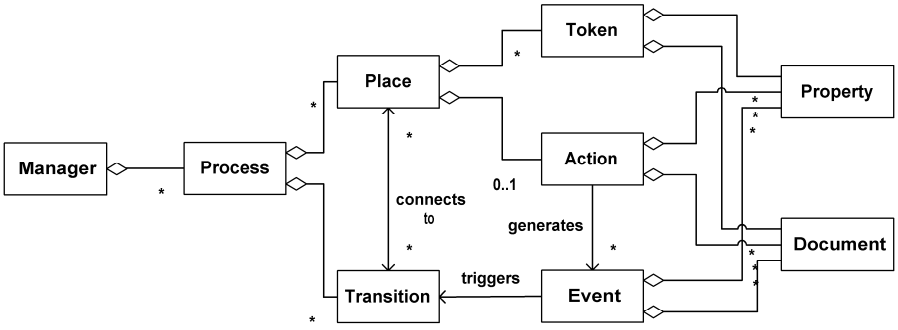

Figure 4. Simplified class diagram for the Workflow Kernel

4.1. The IFeatures base interface

Each of the objects depicted in figure 4 implements its own interface. For example, the Manager object implements the IManager interface, the Process object implements the IProcess interface, the Place object implements the IPlace interface, the Property object implements the IProperty interface, and so on. Despite having different purposes, some of these objects have attributes in common. For example, most objects have a name attribute, Tokens and Events have a color attribute in order to distinguish between different process instances, and other objects such as Places and Transitions have graphical position coordinates, so that the process has the same appearance in different process modeling tools.

To avoid repeating attributes across objects and interfaces, it makes sense to encapsulate these common attributes in base interfaces, and to let each interface expose only those attributes which are unique to their corresponding objects. But if a new base interface

were to be defined for each set of common attributes, this would significantly increase the overall complexity of the Workflow Kernel interfaces. So, in order to simplify these interfaces but still have some sort of reusability, a single base interface was defined for all those objects, as shown in figure 5. This interface, called IFeatures, contains all the attributes that are common to at least two objects. This means that IFeatures may contain attributes that do not belong to all objects. For example, the color attribute (which is a string value) is only applicable to Tokens and Events, whereas the xpos and ypos attributes are only applicable to Places and Transitions, if these are the only objects that can be represented graphically.

In some cases, such as the Process, Place and Transition objects, the purpose of the name attribute is to provide a human-readable designation for those objects. In other cases, namely in the Document and Property objects, the name is a fundamental piece of information: a property must have a name in order to become a name-value pair, and a document requires a neutral designation other than the name of the file it represents. Besides, some objects also have a human-readable description. All Workflow Kernel objects have a unique identifier, which is necessary in order to serialize process objects. These objects are interrelated, e.g., places are connected to transitions. Since object references are volatile, the relationships between objects are stored as relationships between object identifiers.

Besides the identifier, name, description, color and position attributes, the IFeatures base interface contains a *simulation mode* attribute. This attribute is intended for Process and Action objects only, to allow them to run in a protected mode without interacting with resources, which is called the simulation mode. The main purpose of the simulation mode is to test process behavior prior to actual execution, in order to ensure that the process behaves as expected and that the actions have been correctly configured.

In order to support process nesting, the Workflow Kernel allows a Process to be run as a sub-process inside another Process. A sub-process can be regarded as being just a special kind of Action which, like other Actions, must be associated with a Place. But since all Action objects implement the IAction interface, then Process objects must also implement this interface, as suggested in figure 5 by the inheritance relationship between IAction and IProcess. In this case, IAction's Start() and Stop() methods invoke IProcess's Start() and Stop() methods.

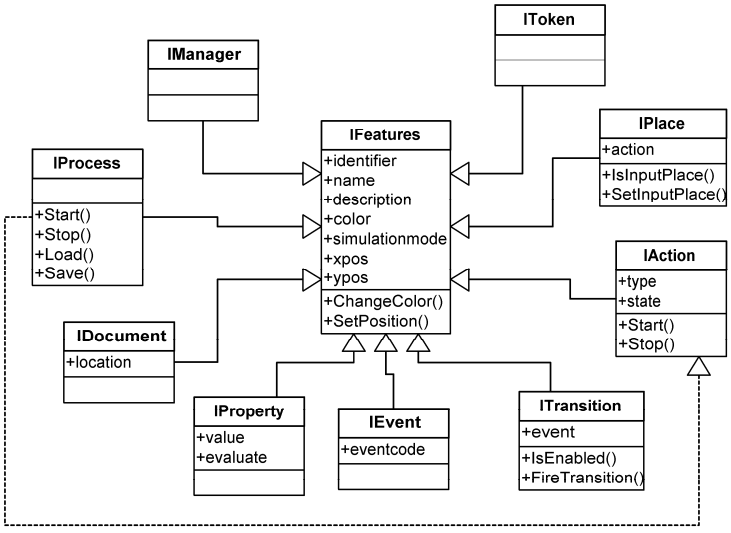

Figure 5. The IFeatures base interface and the object-specific interfaces

4.2. The IEnumerator base interface

Most Workflow Kernel objects contain other objects, as shown in figure 4. For example, each Process contains a set of Places, each Place contains a set of Tokens, and each Token contains a set of Properties and a set of Documents. This means that each object may have to maintain one or more collections of other objects. In practice, these will be collections of object references rather than collections of objects, because each object is created separately. Each of these collections must allow the container object to add, remove and retrieve object references that are stored in that collection. For example, the Process object is a container of Place and Transition references, and a Token object is a container of Property and Document references.

Since most Workflow Kernel objects require the ability to maintain a collection of object references, it would make little sense to specify object-specific methods to deal with collections. Instead, just like IFeatures defines attributes that are common to all objects, there is a base interface that provides methods that are common to all containers, and that allow a container to add, remove and retrieve references from an object collection. This base interface is called IEnumerator, and it is depicted in figure 6. The IEnumerator interface is the base interface for all collection containers which, incidentally, do not have any attributes or methods beyond those provided by IEnumerator. Thus, the methods that give access to a collection are the same for all containers. But each container deals with a different type of objects; for example, a Place container has a collection of Place references, and a Transition container has a collection of Transition references. Therefore, the use of IEnumerator is only possible if all objects share a common base interface, so that IEnumerator's methods refer to this base interface rather than referring to object-specific interfaces.

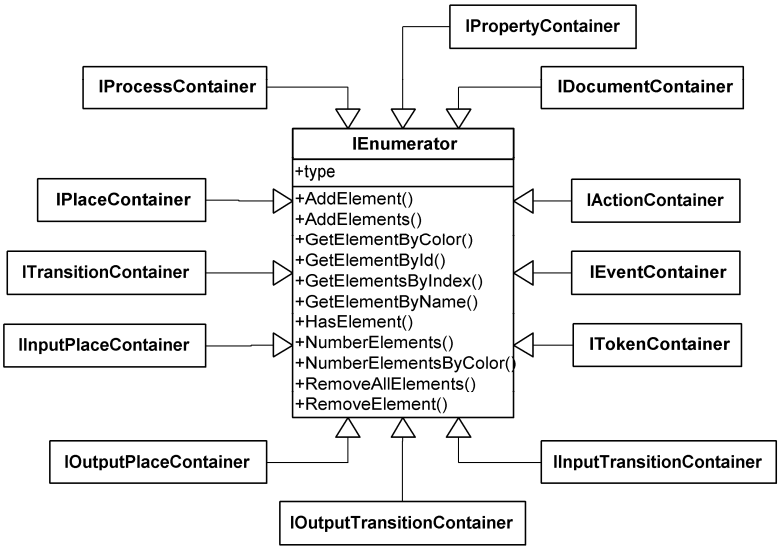

Figure 6. The IEnumerator base interface

In fact, Workflow Kernel objects share a common base interface – IFeatures – so IEnumerator's methods take pointers to IFeatures as input and output parameters. In other words, IEnumerator and all other container interfaces provide access to a collection of references to objects which implement the IFeatures interface. For IPlaceContainer, for example, these objects are supposed to be Places; for ITransitionContainer, the objects are supposed to be Transitions. If an application obtains an IEnumerator pointer, then it can use IEnumerator's type attribute to find out which type of objects the collection holds. The type

attribute is a numeric value that determines whether the IEnumerator pointer actually represents an IProcessContainer, an IPlaceContainer, an ITransitionContainer, etc.

Some Workflow Kernel objects implement more than one container interface. The Manager object is a Process container so it implements IProcessContainer; in addition, it implements IActionContainer in order to maintain a list of all the available types of Action objects in the system. The Process object is a container of Places and Transitions, so it must implement both IPlaceContainer and ITransitionContainer; and it also implements IEventContainer in order to maintain a list of all Events produced by Actions within that Process. A Place may have input and output Transitions, so it implements both IInputTransitionContainer and IOutputTransitionContainer; a similar situation occurs with Transition objects, which implement both IInputPlaceContainer and IOutputPlaceContainer.

4.3. The INotifySink callback interface

In essence, and to be as generic as possible, the Workflow Kernel should be able to notify any change, in any of its objects, to any object that implements INotifySink. Therefore, every object within the Workflow Kernel becomes an event source, since it must produce a notification event whenever one of its attributes changes or whenever one of its object collections changes. For example, a Property will produce a notification event if its name, value or evaluate attribute changes; a Place will produce a notification event when it is given a new Token, or when its input or output Transitions change. In general, both the object-specific interfaces and the container interfaces define methods that allow an object to be changed in some way; each of these changes will produce a notification event.

The INotifySink interface has a single callback method called OnNotify() with five input parameters: (1) a Process reference, (2) an enumeration value describing the type of object that generated the notification, (3) a reference to the object that generated the notification, (4) an enumeration value describing the type of event, and (5) an arbitrary value comprising event-relevant data. The enumerations *obj_type* and *ntf_type* comprise two sets of enumeration values that represent object type (parameter 2) and event type (parameter 4), respectively. By implementing this interface, an object is able to receive and react to events generated by another object.

However, this event notification approach has a major drawback: since all Workflow Kernel objects are event sources, an event sink may have to subscribe to several event sources in order to receive all the events it is interested in. For example, a monitoring tool that keeps track of process execution would perhaps be interested in receiving all events produced within a given Process, so it would have to subscribe to all event sources, from Places and Transitions down to individual Properties and Documents. This is not only impractical but also difficult to implement: if new elements, such as Places or Transitions, are added to the Process then the monitoring tool would have to subscribe to events produced by these new elements.

If there is an event sink that is interested in receiving all events from a given Process, it makes more sense to allow this event sink to subscribe only once to events produced by that Process, regardless of how many elements that Process contains. In the same line of thinking it seems to be appropriate, for example, that if an event sink subscribes to events produced by an Action, it should also receive notification events from all Properties and Documents that the Action contains. In general, if an event sink subscribes to events from a certain object, then it should also receive events from all event sources which that object contains. As a consequence, the highest amount of notification events will be received by subscribers of the top-level Manager object, which is the ultimate container for all objects within the Workflow Kernel, as suggested in figure 4.

To enable this kind of behavior, it was established as a mandatory requirement for the Workflow Kernel that all collection containers must automatically subscribe to events produced by the objects they contain. This way, if an external event sink subscribes to an event source, it is effectively subscribing to the events produced by that event source and by all other objects that event source contains. In addition, in some associations between objects, notably the association between a Place and an Action, an object subscribes to events produced by the object it is associated with. In general, any object that acquires a reference to another object will subscribe to events produced by this object. This requirement implies that events will flow across the Workflow Kernel, down from Properties and Documents up to the singleton Manager object.

4.4. Process execution

The Workflow Kernel can take advantage of this event notification mechanism by using it to implement process execution behavior. Figure 7 illustrates an example of an event notification sequence that takes place during process execution. In this example, an Action A1 has been invoked and it is now returning an Event object. The Action notifies this event to Place P1 which is associated with A1 (step 1). For this purpose, A1 invokes P1's OnNotify() method, where the first parameter, an IProcess pointer, remains unspecified (it will be filled in later on as the notification sequence progresses). Optionally, Action A1 may also notify the same event to an external event sink that has explicitly subscribed to this Action's events.

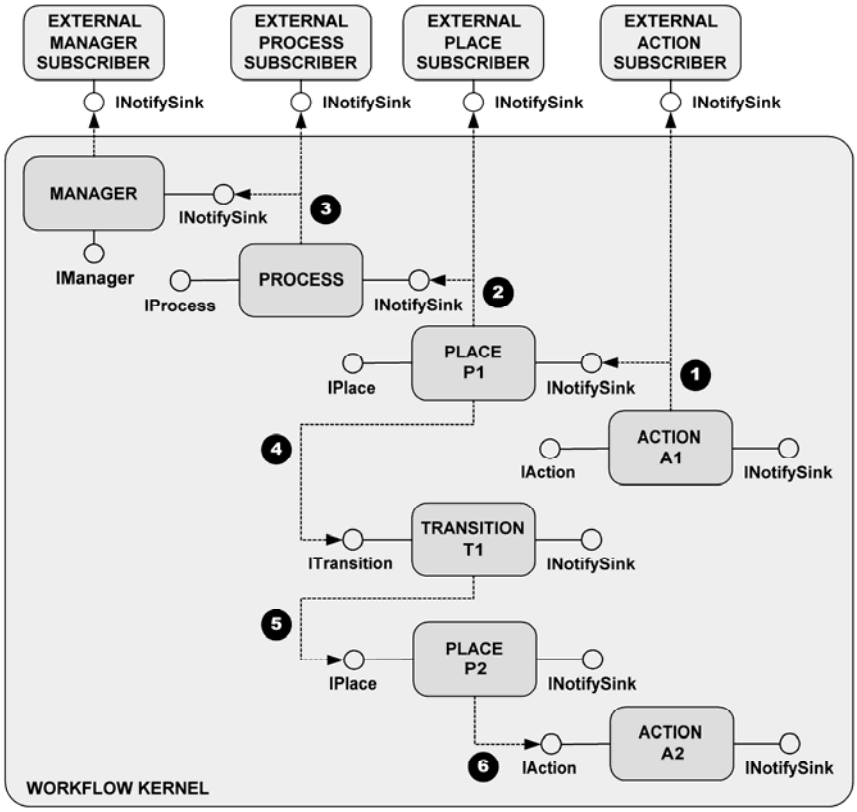

Figure 7. Event notification sequence

Place P1 will forward this event to the Process object, which is the Place's container (step 2). For this purpose, P1 makes a similar method call as A1 did, but now invoking the Process's OnNotify() method; for the time being, the first parameter still remains unspecified.

Optionally, the Place may notify event sinks other than the Process. Once it is notified, the first thing the Process object does is to forward this event to its subscribers (step 3), which include the singleton Manager object and any external event sinks. Hence, the Process object invokes the OnNotify() method of those objects, passing its own interface pointer as the first parameter.

After this chain of events is completed, the Place walks through all of its output Transition objects in order to find out which Transitions should be fired for that Event. In this example, the Place object finds that it should fire Transition T1, so it invokes T1's FireTransition() method, passing the received Event as input parameter (step 4). When the Transition fires, it removes Tokens from its input places (not shown in figure 7) and inserts Tokens into its output places (step 5). When the output place P2 receives a new Token, it starts its associated Action (step 6). It should be noted that steps 5 and 6 will result in new events being generated; these events will follow a similar notification sequence as in steps 2 and 3 (for step 5), and as in steps 1, 2 and 3 (for step 6).

6. Implementing the Workflow Kernel

Given the purpose and approach of the Workflow Kernel, it becomes clear that it should be implemented with a distributed object middleware technology such as COM, CORBA or Java RMI. There are some comparison studies for these technologies, for example [13], which highlight their differences and similarities. Regarding the implementation of the Workflow Kernel, COM seemed to be the most advantageous due to the following reasons:

- *application scenario* – The Workflow Kernel is a workflow enactment service, which is an inherently centralized component within a workflow management system. Therefore, the Workflow Kernel will be embedded either in a centralized application or in a single node of a distributed application. Furthermore, the Workflow Kernel will have to be integrated with desktop applications such as modeling and monitoring tools. Therefore, the Workflow Kernel lends itself more to a single-system, desktop component architecture such as COM, rather than to a distributed component architecture such as CORBA, which excels in heterogeneous environments.
- *reference counting* – In COM, IUnknown is the base interface for every other COM interface. (In CORBA, there is similar base interface which is called CORBA::Object.) One of two fundamental features of IUnknown is that it allows a client object to query for a specific object interface from the set of all interfaces that a COM object implements. The other fundamental feature of IUnknown is that it specifies a basic reference counting mechanism that all object interfaces must support, and which becomes essential in order to keep track of all the objects within the Workflow Kernel.
- *connection points* – COM specifies a mechanism that allows objects to implement the observer pattern [7]. This mechanism is known as *connectable objects* or *connection points* [9]. A connectable object is a COM object that defines one or more callback interfaces, in addition to its own external interfaces. In COM terminology, these callback interfaces are referred to as *outgoing interfaces*, and they must be implemented by clients of the connectable object. In this case, the connectable object is the event source, whereas the client object is the event sink.
- *aggregation vs. inheritance* – The design of the Workflow Kernel makes extensive use of interface inheritance. As a consequence, all Workflow Kernel objects will have to implement base interface methods, and this means implementing the same behavior in several objects. Both CORBA and COM support interface inheritance, but they do not support implementation inheritance. Despite this fact, COM does provide an attractive alternative,

which enables implementation reuse by means of *aggregation*. Aggregation allows an outer object to contain one or more inner objects, and to present the inner objects' interfaces as if they were implemented by the outer object.

• *implementation support* – In spite of the previous advantages, it is clear that the use of COM-specific features, such as reference counting and connection points, requires a significant amount of what is usually referred to as "plumbing code" or "boilerplate code". Reference counting requires all objects to implement IUnknown and to maintain an internal reference count; connection points require objects to implement some standard interfaces such as IConnectionPointContainer and IConnectionPoint; and aggregation requires IUnknown method calls to be delegated to the outer object's IUnknown interface. The COM Active Template Library (ATL) [14] relieves software developers from the burden of having to deal with these technology-specific details.

• *security model* – COM makes use of existing security capabilities without requiring additional development efforts. Within a single host, COM relies on security capabilities provided by the operating system, which are common to all desktop applications. Distributed COM (DCOM) is the extension of COM technology to support distributed systems. In this case, the available security capabilities are based on Kerberos [10].

The Workflow Kernel implementation has been divided into three modules: WfBase, WfKernel and WfAction. The WfBase module is a COM in-process server, i.e., a DLL that is attached to the WfKernel module. The WfBase module defines common Workflow Kernel interfaces: IFeatures, IEnumerator and all of the container interfaces shown in figure 6. It also defines INotifySink and the two enumeration types used by the OnNotify() method. In addition, the WfBase module implements two aggregatable COM objects – Features and Enumerator – which implement the IFeatures and IEnumerator interfaces, respectively.

The WfKernel module is the central module in the Workflow Kernel. It has been implemented as a Windows NT/2000 service, and it can run both as a local COM server and as a remote COM server via DCOM. The WfKernel module defines the object-specific interfaces shown in figure 5, and it implements the corresponding objects except for the Action object. The WfKernel module implements the following COM objects: Manager, Process, Place, Transition, Token, Event, Property and Document. The Manager object is implemented as a singleton COM object, so every client obtains the same object reference when it attempts to create a new Manager object.

To enable the event flow mechanism described in the previous section, each WfKernel object must be simultaneously an event source and an event sink. This means that each WfKernel object not only implements a connection point mechanism, but it must also behave as a client of that same mechanism. For example, both Processes and Places are event sources, so both of them implement connection points for INotifySink; but a Process is also an event sink for events generated by Places, and a Place is an event sink for events generated by Actions, so both of them must implement INotifySink to receive those events.

The WfAction module is a placeholder for Action objects, which depend on the particular workflow management system the Workflow Kernel is built into. Action objects can be implemented outside the WfAction module, as long as they implement the IAction interface. Like the WfKernel object, the WfAction module has also been implemented as a Windows NT/2000 service. It defines two enumeration types – one for IAction's type attribute and another for IEvent's eventcode attribute – and it includes some sample Action objects that have been used during the development and testing of the Workflow Kernel. These sample objects can be used as a starting point for the implementation of application-specific Actions.

7. Reusing the Workflow Kernel

The Workflow Kernel provides two kinds of interfaces that allow external applications to be integrated with it. On one hand, the object-specific interfaces shown in figure 5 allow external application code to invoke and manipulate Workflow Kernel objects. For example, modeling tools will invoke Workflow Kernel objects via their interfaces in order to define Processes and the Actions and Events associated with Places and Transitions. On the other hand, the callback interface described in section 4.3 allows external application code be notified of changes in those objects. For example, monitoring tools will make use of the Workflow Kernel's event notification mechanism in order to keep track of process execution as it unfolds. These two kinds of interfaces are almost analogous counterparts: just like an modeling tool may invoke a Process's interface or get into more detail by invoking a Place's interface, so too can a monitoring tool subscribe to events from a Process or only from a Place within that Process.

An interesting possibility is that the Workflow Kernel allows an external application to do both things at the same time, i.e., to manipulate objects and to receive events from those objects simultaneously. Thus, it is possible to have a modeling tool that is also a monitoring tool, or the other way around. On one hand, this means that it is possible to have two modeling tools displaying the same Process, and the changes inserted via one modeling tool are immediately reflected onto the other. On the other hand, it also means that process modeling and process monitoring no longer have a clear-cut gap; in fact, they may even overlap. For example, while a process is being defined or refined, some of its process instances may be already running and they could be monitored within the modeling tool itself. This is possible because the process definition (the Petri net) is exactly the same data structure in which process execution is recorded.

Figure 8 shows the visual appearance of two graphical components developed specifically for the Workflow Kernel. Since the Workflow Kernel is based on COM, these two components have been implemented as ActiveX controls, so they can be integrated with other COM-based applications. In figure 8, these components are embedded in the Web browser by means of an HTML page. On the left-hand side there is the WfKMonitor component, which basically presents a tree with all Workflow Kernel objects and their contained objects. On the right-hand side there is the WfKEditor component, which allows Processes to be defined and edited, namely by adding or removing Places and Transitions, by connecting Places to Transitions, by associating Actions to Places and Events to Transitions, by configuring the Properties and Documents for Actions and Events, and by adding or removing Tokens to the Process.

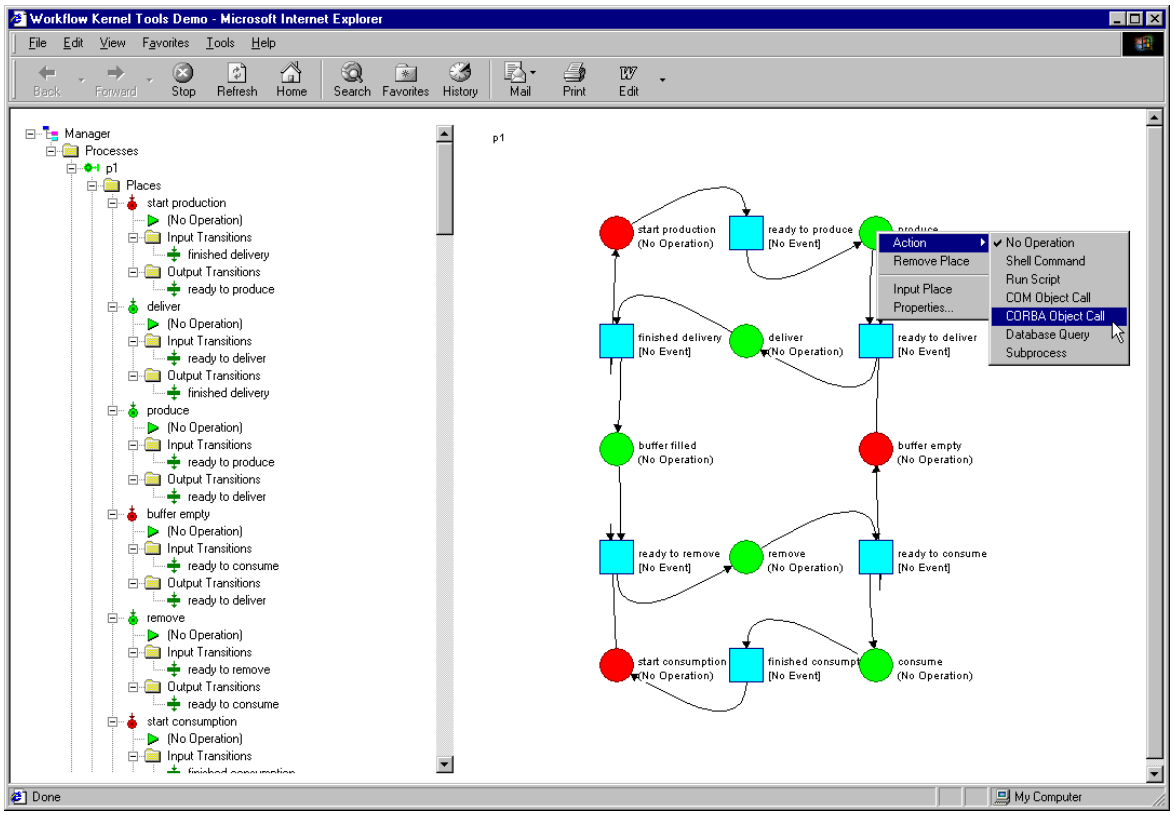

Figure 8. The WfKMonitor and the WfKEditor sample components

Both of these components subscribe to events from the Workflow Kernel: the WfKEditor subscribes only to the events produced within the Process being edited, whereas the WfKMonitor subscribes to events from the Manager object, so the WfKMonitor effectively receives all events produced within theWorkflow Kernel. As a result, any change introduced via the WfKEditor – or, as a matter of fact, via any other local or remote application connected to the Workflow Kernel – is immediately reflected on WfKMonitor. Additionally, if another application introduces a change to the Process being edited, this change is also immediately reflected on WfKEditor.

8. Conclusion

Instead of developing every workflow management system from the ground up, it should be possible to come up with a generic and reusable set of functionality that provides the basic capabilities of a workflow engine. This paper has focused on the development of such a reusable workflow engine. To ensure that the workflow engine is independent of any particular workflow system architecture, the system makes use of Petri nets and extends them with the concepts of Actions and Events. On one hand, Petri nets provide a solid foundation, formal semantics and pave the way for process analysis and verification. On the other hand, the concepts of Actions and Events insulate the system from particular mechanisms of resource invocation. By means of a set of external and callback interfaces, it is possible to develop components that augment the functionality of the Workflow Kernel, until the result is a system that covers the entire functionality of a workflow management system.

Acknowledgement

The first author gratefully acknowledges the support from *Fundação para a Ciência e a Tecnologia* (http://www.fct.mct.pt), which is funding his research.

References

[1] W. van der Aalst, The Application of Petri Nets to Workflow Management, Journal of Circuits Systems and Computers, 8(1) 21-66, 1998

[2] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, A. Barros, Workflow Patterns, BETA Working Paper Series, WP 47, Eindhoven University of Technology, Eindhoven, Netherlands, 2000

[3] A-Frame Software, WorkMovr API Set Overview, http://www.a-frame.com/HTMDocs/PDF/APIset.pdf, 2001

[4] R. David, H. Alla, Petri Nets and Grafcet: Tools for modelling discrete event systems, Prentice-Hall, 1992

[5] Drala Software, Drala Workflow Engine, http://www.dralasoft.com/products/workflow/, 2001

[6] Fujitsu Software Corporation, i-Flow™ Architecture White Paper, http://www.i-flow.com/, 2001

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, Addison-Wesley, 1995

[8] R. Medina-Mora, H. Wong, P. Flores, ActionWorkflow as the Enterprise Integration Technology, Bulletin of the Technical Committee on Data Engineering, IEEE Computer Society, 16(2), 1993

[9] Microsoft, DEC, The Component Object Model Specification, Version 0.9, http://www.microsoft.com/com/resources/comdocs.asp, 1995

[10] B. Neuman, T. Ts'o, Kerberos: An Authentication Service for Computer Networks, IEEE Communications, 32(9) 33-38, 1994

[11] OMG, Workflow Management Facility Specification V1.2, ftp://ftp.omg.org/pub/docs/formal/00-05-02.pdf, 2000

[12] A.-W. Scheer, ARIS – Business Process Modeling, Springer Verlag, 2000

[13] G. Raj, A Detailed Comparison of CORBA, DCOM and Java/RMI, http://my.execpc.com/~gopalan/misc/compare.html, 1998

[14] B. Rector, C. Sells, ATL Internals, Addison-Wesley, 1999

[15] Sheth, A., Aalst, W. van der, Arpinar, I., Processes Driving the Networked Economy, IEEE Concurrency, pp. 18-31, July-September 1999

[16] Vivtek, wftk: Open-source Workflow Toolkit, http://www.vivtek.com/wftk/, 2001

[17] WfMC, The Workflow Reference Model, Document Number TC00-1003, http://www.wfmc.org/, 1995