WORKFLOW MODELING AND SIMULATION USING AN EXTENSIBLE OBJECT-ORIENTED KNOWLEDGE BASE MANAGEMENT SYSTEM

Sunil Kunisetty
Database Research and Development Center
University of Florida
sk0@cis.ufl.edu

Abstract

Today's business enterprises must deal with global competition, reduce the cost of doing business, and rapidly develop new services and products. To address these requirements, enterprises must constantly reconsider and optimize the way they do business and make the best use of their information systems, applications and enterprise constraints and knowledge to support evolving business processes. Therefore, there is a need for the integration of workflow technology and knowledge management technology.

In this report, we present the design and implementation of a workflow modeling and simulation system (WFMSS) based on an extensible object-oriented knowledge base management system, OSAM*.KBMS. In particular, we add modeling constructs to the underlying object model of the KBMS for modeling the control structures of workflow processes and use event-condition-action-alternative action rules to specify constraints associated with these processes. For "every" participant in real life workflow process, a corresponding construct has been added, and because of this one-to-one correspondence, process modeling is made simpler. To facilitate workflow modeling, a graphical tool is being developed under a separate effort and the graphical representations of process models are used to generate their textual representations posed in a workflow definition language (WFDL).

Unlike most of the traditional workflow systems, which follow the *interpretive* approach for enacting a workflow, we follow the *compiled* approach. The data and process modeling facilities of the KBMS are used to model data entities and work processes and a Code-Generator module of the KBMS compile and generate an executable process controller for each workflow process model. In the simulation mode, the process controller is executed and data are gathered and analyzed to validate the corresponding workflow process. Based on the analysis, the process model can be modified to avoid bottlenecks or to improve performance. The validated process

model can then be used to control the activities of a real world business operation. In this work, workflow enactment is based on active rules. A developed active knowledge base management system is used for modeling and processing workflow processes.

1 Introduction

The competitiveness of many enterprises depends critically on the efficiency and quality of organizing the business processes within the enterprise [MCL93]. The term workflow management has been established to denote the computer-based support for the design, execution, and monitoring of business processes [GEO95, RUS94]. Workflow engines are commercially available, but their use is typically limited to local office environments with a few dozen users. In contrast, enterprise-wide workflow management system requires additional infrastructure software, often referred to as "middleware" [BER93], to ensure scalability in the presence of thousands of users and thousands of concurrently active workflows, to provide high availability in the presence of failures, and to cope with the inherent heterogeneity of large enterprises with high autonomous subsidiaries. Application examples that would enormously benefit from even partial solutions include credit processing in banks, insurance claim processing, health care administration, submission and processing of tax declarations, and so on.

Workflow modeling and simulation is emerging as a challenging area for knowledge bases, stressing knowledge base technology beyond its current capabilities. Workflow modeling and simulation systems need to be more integrated with knowledge management technology, in particular as it concerns the access to external knowledge bases and application systems. Thus, a convergence between workflow modeling and simulation and knowledge bases is occurring. In order to make the convergence effective, however, it is required to improve and strengthen the specification of workflows at the conceptual level, by formalizing within a unified model their "internal behavior" (e.g. interaction and cooperation between tasks), their relationships to the environment (e.g. the assignment of work task to agents) and the access to external knowledge bases.

Workflows [CAS95] are processes involving the coordinated execution of multiple tasks, called "activities" in our system, performed by different processing entities. An activity defines some work to be done by a person, by a software or by both of them. Specification of a workflow (WF) involves describing those aspects of its component activities (and the processing entities that execute them) that are relevant to control and coordinate their execution, as well as the relations between the activities themselves.

Intuitively, any activity must be enacted by the appropriate user to take input from previous activities and transform those inputs, adding values, to produce an output (data, event, etc.). This transformation is accomplished by a function (method) performed by the user, an application, a machine, etc. The function must meet some business and performance objectives. "Supplier activities" ensure that the input is of the desired quality and that the input criteria for "customer activities" are met. To make sure that the output is useful for other customer activities,

the output criteria should meet their input criteria . Thus, with each activity there needs to be a procedure which is enacted; that is, executed to meet the completion criteria.

Workflow modeling and simulation systems schedule activities in accordance with previously processed activities. Traditionally, the coordination of activities is performed by humans. Workflow modeling and simulation systems support a corporation e.g. enterprise or public administration, in automating these workflows in order to define well established "horizontal" processes or to reduce personnel or both. To do so, the various types of workflow in the respective environment have to be described. The resulting specifications are fed in a workflow modeling and simulation system and used for controlling work activities, either by launching automatic tools or by informing humans. Workflow modeling and simulation systems must autonomously recognize situations in which necessary actions have to be carried out.

A workflow modeling and simulation system in short should support :

- Modeling constructs to model a workflow.
- Enactment capabilities to execute a workflow.

Another issue which is important, but not mandatory is report generation capabilities. In general, real life processes such as manufacturing processes, business processes are very complex, huge, and expensive. Hence, they have to be foolproof, efficient, and fast. All these performances can be improved through process analysis and performance evaluations after each execution. At the same time it is neither recommended nor worthwhile to execute the whole process just for performance analysis, as executing these processes is a costly and a time-consuming affair. So it is suggested that we have a simulated prototype of the real life process, which is inexpensive and simple. The prototype can be subjected to regress testing and the ensuing results can be analyzed. Based on the analysis, the process can be re-modeled to improve the performance. Another advantage of simulated prototype is that reengineering of the workflow model is made easy. Reengineering increases customer satisfaction, improves efficiency of business operations, increases quality of products, reduces cost, and meets new business challenges and opportunities by changing existing services or introducing new ones.

The impetus for the design and development of a workflow modeling and simulating system (WFMSS) has been the aforementioned advantages. The WFMSS is developed based on an existing extendable object-oriented knowledge base management system, OSAM*.KBMS [SU92,SU93]. It is implemented by extending the underlying model of OSAM*.KBMS, XKOM, rather than built from scratch. This is because XKOM is reflexive and it supports rich features such as model extensibility, rules, methods and associations, with which the WFMSS can be built more easily. Since, the KBMS also supports persistence, querying, transaction management etc., the work is much simplified. For modeling a workflow, many new constructs have been added to the XKOM. These constructs are class types and association types.

Unlike traditional systems, which follow the *interpretive* approach for enacting a workflow, we follow the *compiled* approach. In our approach, the workflow developer models a business process using the modeling constructs provided in KBMS. The KBMS compiles the process model and generates process controller for the

model. The process controller is executed every time a request for an enactment is made, and the ensuing results are obtained and analyzed. Based on the analysis, the business process can be re-modeled for better performance. At run-time, the process controller controls and co-ordinates the activities of a business. In the interpretive approach, first, the workflow modeler models the business processes using the modeling constructs provided by a workflow modeling and simulation system to produce a number of process models at build-time. These process models are then interpreted at the run-time by the workflow modeling and simulation system. The block diagrams for the illustration of these two approaches are shown in Figure 1.

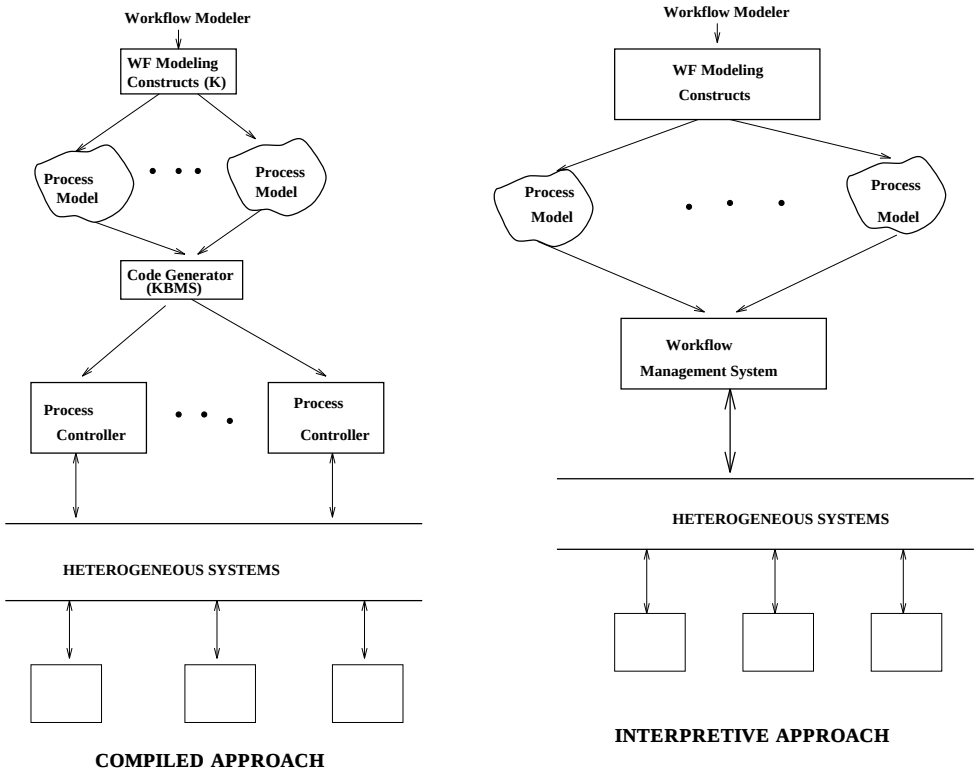


Figure 1: Interpretive and Compiled Approaches

Workflow enactment in our system is based on **rules**. This is because, active database technology is particularly suitable for providing an operational model of workflow enactment. Active rules follow the *event - condition - action - alternate action (ECAA)* paradigm: events are typically changes to the database content, conditions are database queries, and actions are arbitrary computations, possibly causing database changes. We describe how formal workflow descriptions can be used as input in order to semi-automatically generate both the schema of workflow data and the code of active rules for their management. In this operational model, workflow enactment is traced by means of manipulations of workflow data, and the behavior of the WFMSS is modeled by active rules and methods.

Essentially, there are two types of rules, Meta rules and User-defined rules. Meta rules govern the functioning of the WFMSS, according to the state diagram and the semantics of activation of activities in a WF according to our model. User-

defined rules are specified by the workflow developer. Meta rules are pre-defined as parameterized rules by the workflow customizer (or knowledge base customizer). These rules are converted to bound rules (or application specific rules), when an instance of workflow schema is created. This process is done by a Rule Binder module of the KBMS, with the help of a Dynamic Expression Evaluator.

The remainder of this report is organized as follows. Section 2 presents a case study of workflow modeling and simulation system : its evolution and its necessity. It also describes different types of uncertainties in a WFMSS and a classification of WFMSSs. Section 3 describes workflow modeling and the various constructs that are necessary for workflow modeling. In Section 4, the simulation of a process model is illustrated using an example. A description of the underlying model, XKOM, and a brief overview of model extensibility are given in Section 5. In Section 6, the design and implementation of the WFMSS is explained. Section 7 summarizes the presentation in the earlier sections and discusses possible directions for future research.

2 Workflow Modeling and Simulation Systems - A Case Study

In this section, we discuss the general concepts of a workflow modeling and simulation system, and it's evolution, applications, and need. In Subsection 2.3, we discuss various types of uncertainties in workflow applications. In Subsection 2.4, we elaborate on different types of workflow systems. The basic idea behind the introduction of this section is to give an overview and perspective view of a WFMSS.

2.1 The Evolution of Workflow

Workflow software products, like other software technologies, have evolved from diverse origins. While some offerings have been developed as pure workflow software technologies, many have evolved from image management systems, document management systems, relational or object database systems, and electronic mail systems. Vendors who have developed pure workflow offerings have invented terms and interfaces, while vendors who have evolved products from other technologies have often adapted terminologies and interfaces. Each approach offers a variety of strengths from which a user can choose, adding a standards-based approach which would allow a user to combine these strengths in one infrastructure.

2.2 The Need for Workflow Modeling and Simulation

The workflow concept has evolved from the notion of process [MED92, RAM94] in industry and office. Such processes have existed since industrialization and are products of a search to increase efficiency by concentrating on the routine aspects of work activities. They typically separate work activities into well-defined tasks, roles, rules, and procedures which regulate most of the work in industry and office. Initially, processes were carried out entirely by humans who manipulated physical objects. With the introduction of information technology, processes in the work

place are partially or totally automated by information systems,i.e., computer programs which perform activities and enforce rules that were previously implemented by humans.

2.2.1 Workflow Processes

Workflow processes [GEO95] can be classified as follows :

- **Material** processes relate to human activities that are rooted in the physical world. Such tasks include, moving, storing, transforming, measuring, and assembling physical objects.
- **Information** processes relate to automated activities(i.e., activities performed by programs) and partially automated activities(i.e., activities performed by humans interacting with computers) that create, process, manage, and provide information. Typically an information process is rooted in an organization's structure and/or the existing environment of information systems. Database, transaction processing, and distributed systems technologies provide the basic infrastructure for supporting information process.
- **Business** processes are market-centered descriptions of an organization's activities, implemented as information processes and/or material processes. That is, a business process is engineered to fulfill a business contract or satisfy a specific customer needs.

2.2.2 Reengineering

Once an organization captures its business in terms of business processes, it can reengineer each process to improve it or adapt it to changing requirements. Reasons cited for business process redesign include increasing customer satisfaction, improving efficiency of business operations, increasing quality of products, reducing cost, and meeting new business challenges and opportunities by changing existing services or introducing new ones. Business process reengineering involves explicit reconsideration and redesign of the process. Information process reengineering involves determining how to use legacy and new information systems and computers to automate the reengineering of business processes. These two activities can be performed iteratively to provide mutual feedback. While business process redesign can explicitly address the issues of customer satisfaction, the information process reengineering can address the issues of information system efficiency and cost, and take advantage of advancements in technology.

Workflow is a concept closely related to reengineering and automating business and information processes in an organization. A workflow may describe business process activities at a conceptual level necessary for understanding, evaluating , and redesigning the business process. Workflows can capture information process activities at a level that describes the process requirements for information system functionality and human skills.

Workflow modeling and simulation (WFMS) is a technology supporting the reengineering of these processes. It involves:

- Defining Workflows, i.e., describing those aspects of a process that are relevant to controlling and coordinating the execution of its tasks.

- Providing for fast (re)design and (re)implementation of the processes as business needs and information system change.
- Providing a way to control and coordinate processes using the workflow model.

2.3 Uncertainties in Workflow Applications

Early workflow models did not address data sharing, persistence and failure recovery. Recently, a number of models have been proposed which incorporate a number of new features not found in traditional models. These added features make it possible to address various problems arising from modern business applications such as cooperation and long running activities. However, there are some issues which are yet to be resolved.

Most workflows take a static view toward workflow applications. The static view requires that applications have the following two properties:

- All the components(activities, dependencies, etc.) are known in advance.
- The structural information alone can determine an expected execution path.

However, it turns out that some applications do not have either or both of these properties. In these applications, some components may be either conditional or contingent. Also, the expected execution path may not be possible if only structural specification is present. Although we may circumvent the last problem by forcing the application to follow a predefined execution path, flexibility is at stake. We say that *uncertainty* [TAN95] exists in these applications, and they are of three types :

- **Domain Uncertainty :** Uncertainty in determining beforehand, the number of activities that will participate in a workflow.
- **Structural Uncertainty :** Uncertainty in occurrences of dependencies between tasks.
- **Implementation Uncertainty :** Refers to the phenomenon that there exist more than one way of enforcing a dependency, and the acceptable choices depend on certain conditions.

2.4 Workflow Systems

Workflow systems [TAN95] can be described according to the type of process they are designed to deal with. Thus we define three types of workflow systems:

- **Image-based Workflow Systems** are designed to automate the flow of paper through an organization, by transferring the paper to digital "images". These were the first workflow systems that gained wide acceptance. These systems are closely associated with "imaging" technology, and emphasize the routing and processing of digitized images.
- **Form-based Workflow Systems** are designed to intelligently route forms throughout an organization. These forms, unlike images, are text-based and consist of editable fields. Forms are automatically routed according to the information entered on the form. In addition, these form-based systems can notify or remind people when action is due. This can provide a higher level of capability than image-based workflow systems.

- **Coordination-based Workflow Systems** are designed to facilitate the completion of work by providing a framework for coordination of actions. The framework is aimed at addressing the domain of human concerns (business processes), rather than the optimization of information or material processes. Such systems have the potential to improve organizational productivity by addressing the issues necessary for customer satisfaction, rather than automating procedures that are not closely related to it.

3 Workflow Modeling

In this section, we describe the workflow modeling concepts, such as workflow definition language, activities, associations and other modeling constructs provided by our system.

3.1 Overview

We define a **WF** schema as a structure describing relations among activities (**WTs**) that are parts of the WF. In the schema, we describe which activities should be executed, in which order, who may be in charge of them, which operations should be performed on tables of external databases. WF schemas are described by means of a *WF Description Language* (**WFDL**) [CAS95].

A WF instance (or case) is a particular execution of a schema. For example, a WF schema may describe the process of reviewing submitted papers; a WF instance of that schema is created whenever an editor receives a new paper. Thus, normally, several WF instances of the same WF schema may be active at the same time.

3.2 Modeling Constructs

As it is very important for a workflow modeling and simulation system to support constructs for modeling a workflow, we have extended our meta-model to add these new constructs. For every "participant" (processes, agents, users etc.) in the real life workflow, we added a construct to our kernel meta-model. Since there is one-to-one correspondence between a real life and a modeling construct, specifying the workflow in our model is simplified and made easy. For example, a real life process such as an industry process or a manufacturing process can be modeled as a WF Schema and real life tasks can be specified as **activities** in our system. Tasks sometimes have many sub-tasks or operations. Sub-tasks can be specified as methods. Dependencies between tasks can be modeled as associations in our system. Since there are different types of dependencies, we have different association types, generally, one for each type. In order to do housekeeping work for each enactment of the workflow, we have another construct, named **project**. In addition to these, we also have many other utility classes to facilitate more expressibility and understanding. These utility classes help in detailed analysis of the workflow enactment.

3.3 Workflow Description Language

A Workflow Description Language describes activities to be performed during the WF execution and the mechanisms which are used for their activation and termination, both in normal and exceptional situations. Activity coordination is supported in a restricted number of alternate ways, thereby providing the classical constructs for parallelism, such as *fork* (**Parallel**) and *join* (**Synchronization**). In addition, the behavior of activities is formally described by listing their preconditions, their actions, and their exceptional conditions during their execution.

3.3.1 WF Schema

A real life **process** can be modeled as a WF schema in our WFMSS. The process can be of any magnitude or of any type. It can be either a manufacturing process, a business process, or a software process etc. A process is a set of inter-related tasks. A WF schema is composed of descriptions of flows, activities and sub-activities. Tasks are modeled as activities in our WFMSS. Detailed description of activities is given in the next subsection. A WF schema can have any number of activities. Relationship between these tasks (or activities) can be specified by control associations. Data can also be passed between tasks (or activities).

The attributes of a WF schema are *Start* and *Finish* activities. A WF schema has only one *start* activity, and an instance of this activity has to be activated when a new enactment of workflow is started. A WF schema can have any number of *finish* activities. If an enactment of a workflow ends in any one of these *finish* activities, then the enactment is said to be *complete*, otherwise *incomplete*. The *start* activity has at least one successor activity and each *finish* activity has several predecessor activities.

In the object-oriented representation, a WF schema is defined as a class and so is an activity. One reason for the introduction of this class is for the decomposition of an activity class. If an activity (or task) is complex, then it can be decomposed into a structure of finer activities defined in another WF schema. Even during the incremental development of the model, activities can be treated as black boxes in the initial stages and later can be decomposed into a structure of finer activities defined in another WF schema.

3.3.2 Activities

A real life *task* is defined as an activity in our WFMSS which is modeled by an *active* entity class. An activity can be a manufacturing activity, a business activity, or a software application activity in a real life process. Activities are the elementary work units that collectively achieve a WF goal. The WFMSS decides when a certain activity must start its execution. An instance of an activity is an activation, or execution of that activity, and can record different measurements and status, e.g. the start_time, finish_time of the activity instance, activity finish status, etc. The instance of an activity is recorded in the

knowledge base automatically during its activation and is uniquely identified by an *activationId*. Activities are inter-related and their inter-relationships can be specified by control associations. Information (through means of data) can be shared between activities.

Conceptually, activities are classified into four types :

- **Batch** : Does not need assistance from an agent, it can run by itself.
- **Interactive** : Needs an agent's assistance and needs to be activated by an agent.
- **Manual** : Totally human activity and no computer assistance is needed.
- **Dummy** : It does not do any concrete work. If an activity is decomposed into a schema, then that activity should not perform any action. Such activities are classified as *dummy*, as they do not have any concrete work to do.

Each activity has five major characteristics.

- **Name :** A string identifying the activity.
- **Data :** Attributes associated with the activity.
- **Description :** Few lines in natural language, describing its purpose and use.
- **Preconditions :** This is optional, if exists, it is a Boolean expression of simple conditions which must yield a truth value before the actions in the activity can be executed. Simple conditions are nothing but boolean expressions expressible in our programming language called **K**.
- **Actions :** Sequence of statements which serves as a *specification* of the intentional behavior of the activity. These statements describe data manipulations of temporary and persistent WF data occurring while the activity is active.

The status of an activity instance should be one of the following.

- **Inactive :** When it is created but not yet activated.
- **Active :** When it is being activated or executed.
- **Finish :** When the activation is completed.
- **Inhibit :** If the pre-condition of an activity instance is false, then the execution of that activity instance is skipped. However, the workflow enactment continues. Status of such activity instances are set to *inhibit*.
- **Suspend :** Instead of waiting for some action (like input), the activation can be terminated successfully, but flagged as suspended.

The status of the activity instance changes from one to another as the execution proceeds. Graphical representation of transition is shown in Figure 2.

3.3.3 Project

When a workflow is enacted, it is important that, we record the metrics (such as start time, activate time, finish time etc.) for the activation of each activity instance. And, we also need to keep track of the activity instances that have

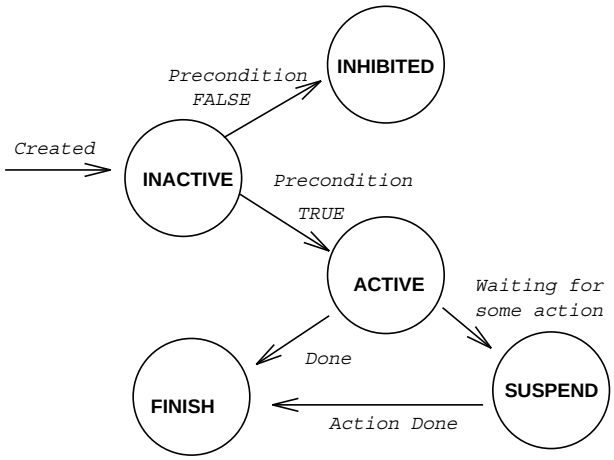


Figure 2: State Transition Diagram

been activated and those that are to be activated etc. All this housekeeping work for every enactment of a workflow is done by this class. For every WF schema there is an associated Project, as we cannot create an instance of a WF schema. Whenever a new request for an enactment of a workflow is made, an instance of the Project associated with the WF schema is created. This project instance is initialized and started. Once started, initial preparations for the project are made and an instance of the *start* activity of the WF schema is created and activated. This activity instance in turn activates other activities and the cycle continues, thus completing the enactment. Throughout, the execution metrics are recorded and computed. Once the enactment is complete, based on the metrics recorded, a report is generated. Detailed analysis can be performed using the report.

3.3.4 Associations

Connections between activities are defined as associations in our model. They define dependencies among activities. These connections have both linguistic description (in WFDL) and graphical description. Each activity can have more than one input connection and more than one output connection. If there is an association between activities **A** and **B**, then they are directly connected by an edge in the workflow diagram. If the edge is from **A** to **B**, then **A** is called *predecessor* of **B** and **B** is called *successor* of **A**. Several association types have been designed keeping in view the real life processes. These association types are specified using WFDL (see appendix A.1). They can be broadly classified as :

- Decomposition
- Data Flow
- Control

Control Associations are once again classified as :

- Parallel

- Sequential
- Testing
- Synchronization
- Case
- Loop

Decomposition A decomposition association, labeled with a **"D"**, is used to decompose an activity class (the defining Class) into a more detailed level of specification represented by a schema class (the constituent class). The schema class into which an activity class is decomposed contains a number of detailed activity classes, along with data entities and associations among them.

The activity class which is decomposed is referred to as the high-level activity class, and the schema class containing the high-level activity is referred to as the high-level schema class. The schema class into which the high-level activity class is decomposed is referred to as low-level schema class, and an activity class in a low-level schema class is referred to as a low-level activity class. An activity class which is not decomposed is referred to as a leaf activity class.

An activity class can be decomposed into a hierarchical structure of schema classes until a set of methods can be used to model the operational properties of a set of low-level activity classes. There can be atmost one decomposition association for each activity class, i.e. an activity class can not be decomposed into more than one low-level schema class. Hence, there should be only one association link for this association type. Decomposition of an activity to a schema class is basically for better presentation and comprehension of the workflow. The graphical representation of this association type in a workflow diagram is shown in Figure 3.

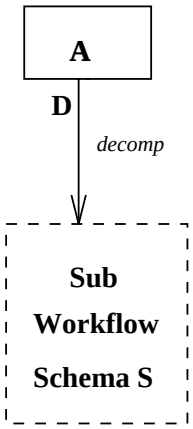


Figure 3: Decomposition Association

Dataflow A dataflow association, labeled with an **"F"**, is used to define the dataflow relationship among activities. Using this association type, activities can communicate and exchange information. This is very important

and useful because in real life processes, tasks need to communicate with each other and also sometimes their execution depends on the data they exchange. Information is passed by means of the data attributes of the activity. In the OSAM* model, aggregation associations are used to define the data attributes of activities. Since the dataflow can be from one activity to any other collection of activities, and vice versa, there can be any number of links for this association type. The graphical representation of this association type in a workflow diagram is shown in Figure 4.

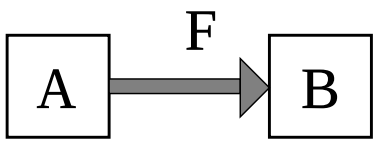


Figure 4: Dataflow Association

Sequential It is very common in real life processes that once, a task is completed, another task is activated, i.e., sequential execution of tasks. To model such dependencies we have introduced the sequential association type. A sequential association is labeled with an "S" in a workflow diagram. There can not be more than one sequential association per activity. Thus, there can be atmost one association link for this type in the specification of an activity. The graphical representation of this association type in a workflow diagram is shown in Figure 5.

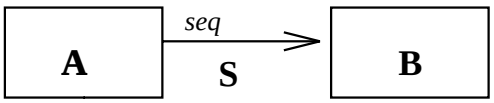


Figure 5: Sequential Association

Parallel When real life tasks do not depend and conflict with each other, then it would be very efficient to activate them concurrently. Running tasks concurrently has many benefits; efficiency and throughput improvement. To model this type of execution, we have introduced the parallel association type in our system. Activities in a parallel construct, run concurrently and are not dependent on each other. Since there can be any number of activities running concurrently, there can be many association links for this association type, i.e., one association link for each concurrent activity. A parallel association is labeled with a "P" in a workflow diagram.
A constraint can be defined on a parallel association. For example, one can specify that either B or C can be activated upon the termination of A. If B is chosen, C is not necessary to be executed, and if C is chosen, B is not necessary to be executed. The constraint can be generalized to chose 1 out of 3, 2 out of 3 parallel activities etc. The graphical representation of this association type is shown in Figure 6.

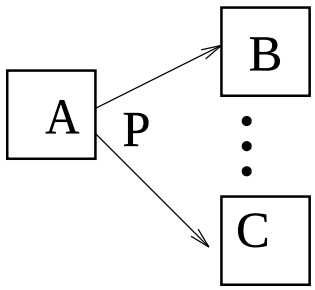


Figure 6: Parallel Association

Synchronization It is used to define a rendezvous point of activities executing in parallel. If there is a synchronization association from activities A and B (or more) to activity C, it means that upon the termination of both A and B, C is activated, and A and B should have the same *enactment identifiers (ids)*. Same as a sequential association, a new instance of C is created and the *enactment id* is passed along. Since an activity can be synchronized from many activities, there can be any number of association links in this construct. A synchronization association is labeled with a "Y" in a workflow diagram.

The synchronization association is different from a set of sequential associations. For instance, if activity C has a synchronization association with activity A and B, C can be executed only after both A and B are completed, and the *enactment ids* of A and B should be the same. Whereas, if A has a sequential association with C, and B has a sequential association with C, C can be executed after either A and B is completed. The *enactment ids* of A and B need not be the same. Moreover C could be executed twice, once activated by the termination of A, and once by the termination of B.

Similar to a parallel association, a special constraint, n out of m, can be defined on a synchronization association. If there is a synchronization association from activities A, B, C (or more) to an activity D, and if 2 out of 3 constraint is specified, it means that D is activated upon the termination of either A and B, B and C, or C and A, the *enactment ids* of A and B (or B and C, C and A) must be the same. In our implementation, the first two activities that are completed will activate D. The graphical representation of this association type in a workflow diagram is shown in Figure 7.

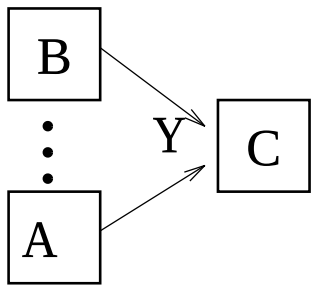


Figure 7: Synchronization Association

Testing Upon the activation of an activity, we may have to chose between two activities to select one as the next activity for activation based on the value of a predicate, i.e, if the value of the predicate is TRUE then one activity is activated, otherwise another activity is activated. To model such inter-relationships, *testing* association type is added into our system. A testing association is labeled with a "T" in a workflow diagram. Testing association has only two association links. The links names for these two association links are two standard keywords, *IfTrue* and *IfFalse*. A Testing association has a *condition* clause associated with it, and this clause takes a predicate as its argument. A predicate is a **K** Boolean expression. When evaluated, this expression returns TRUE or FALSE. If the expression evaluates to TRUE, then the activity in IfTrue link is activated, otherwise the one in IfFalse link is activated.

The graphical representation for this association type in the workflow diagram is shown in Figure 8.

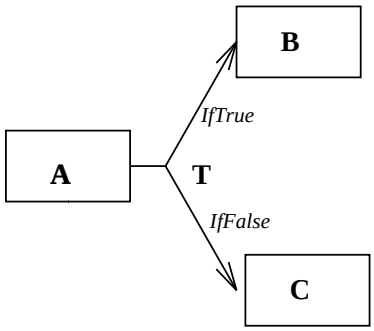


Figure 8: Testing Association

Case The Case association type is a general case of the testing association type. In the testing association type, choice is made only between two activities. However, if we want to chose one or more activities among several activities available for activation, then we need to extend the testing association type. To facilitate such an option, we have introduced the **case** association type. This association type can have any number of links. All but one has a condition clause associated with it, and each of them takes a predicate as its argument. If the predicate evaluates to *true* then that activity is activated. Any number of activities can be activated as long as the predicates associated with them evaluate to *true*. One particular (generally the last one) link doesn't have a predicate clause to it, and that one is termed as *default*. If none of the predicate expressions evaluated to *true*, then this activity is activated. The Case association type is denoted by **C** in workflow diagram. The graphical representation of this association type in a workflow diagram is shown in Figure 9.

Loop It is often necessary for an activity to be activated a number of times. To model such dependencies we have introduced the **Loop** association type. It

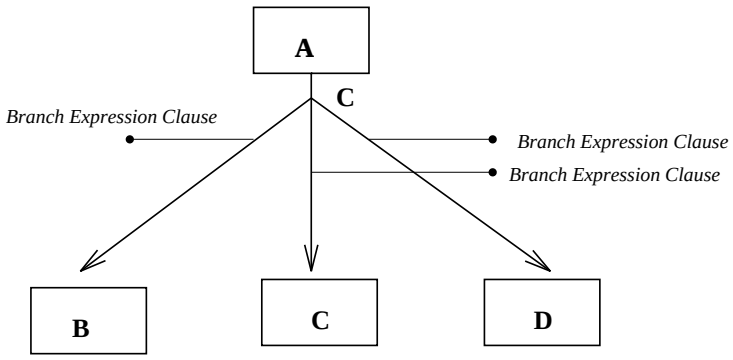


Figure 9: Case Association

is denoted by **L**. The loop association type has a condition clause. As long as the condition is **true** the body is repeatedly executed. Initializations, if any, for the condition clause is specified by an initialization clause of the association type. The loop increment is specified by the loop increment clause. The body of the loop is an activity which has to be activated every time the body is executed. Since the body of the association has to be only one activity, this association type should have only one link. The graphical representation of this association in a workflow diagram is shown in Figure 10.

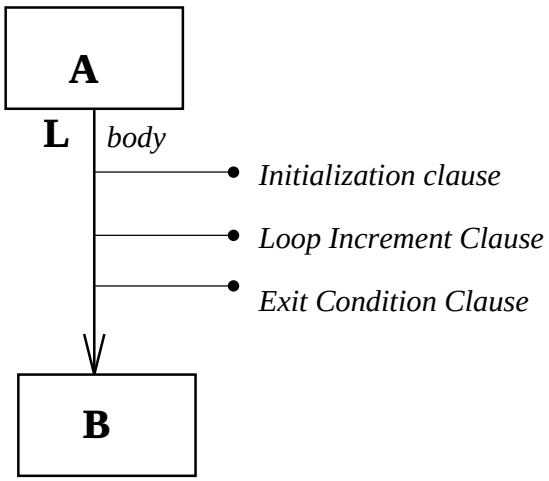


Figure 10: Loop Association

3.4 Utilities

Utility classes have also been provided in our system for better expressibility and understanding. They are used for better presentation of the workflow model.

3.4.1 User

User in a workflow management is an agent who is responsible for the enaction, either on the computer or manually, of any activity within the scope of modeled

workflow. A user can play more than one Role.

3.4.2 Role

A role is associated with a group of users who are collectively responsible for the enaction of a coherent set of activities. Usually Role has a name or a title, such as manager, programmer, controller, etc. The relationship between user and role is M:N, i.e., a user can play different roles, and a role can be played by different users. A user needs authorization to play a certain role in a workflow modeling and simulation system.

3.4.3 View

A view consists of a set of activities in a WF model for which a certain group of users playing certain role are responsible. They show what activities, different groups of users playing different roles, are responsible for. The relationship between a role and a view is 1:1. Views can overlap, i.e., an activity can be in different views.

Views are introduced only for presentation purpose. During modeling, it is often convenient to define the individual views and then construct the complete workflow from the views. During the analysis of the workflow model, focus can also be made on one or more views, rather than the entire model. Results of the activation of activities in a particular view can also be obtained during the workflow enactment. Grouping a set of activities to form a view can be based on their objectives, information they use, activation times, applications they form, and enaction and decision support.

4 Workflow Simulation

4.1 Need for Simulation

Real life processes such as manufacturing processes, business processes, etc., are very complex, huge, and expensive, hence they have to be foolproof, efficient, and fast. Their performances can be improved through process analysis and performance evaluations after each execution. However, it is neither recommended nor worthwhile to execute a process real legacy systems, since its execution is a costly and time consuming task and the data changes made to the legacy system are difficult to rollback. Hence, it is simpler and less costly to evaluate a process model by simulation. The process model can be subjected to regress testing, and analysis can be made from these results. Based on the analysis, the process model can be modified to improve it's performance and to avoid bottlenecks. Another advantage of simulation is that reengineering of the workflow model is made easy. Reengineering increases customer satisfaction, improves efficiency of business operations, increases quality of products, reduces cost, and meets new business challenges and opportunities by changing existing services or introducing new ones.

4.2 An Example

We shall use an example to explain the modeling and simulation of a workflow.

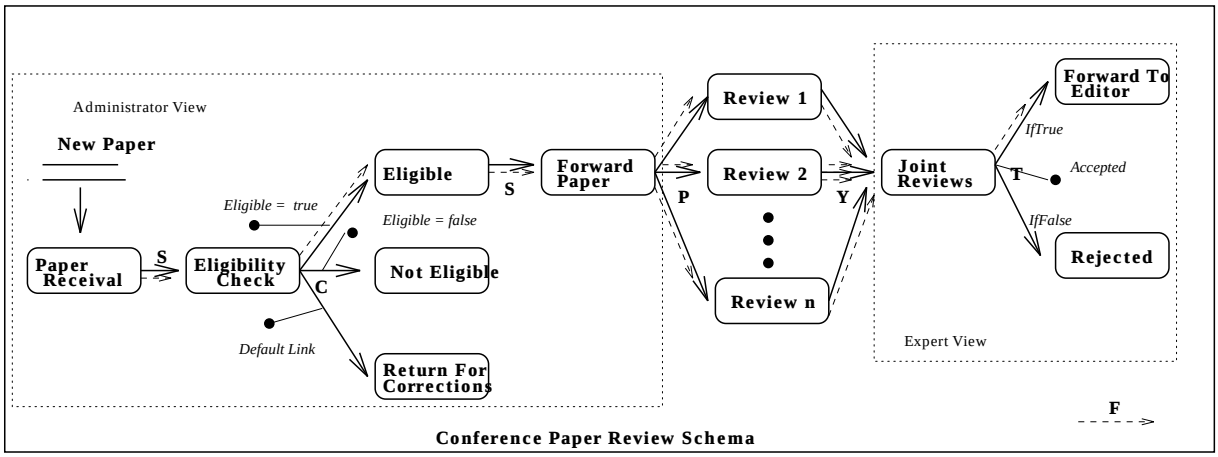
4.2.1 Description

The workflow under discussion involves a review process for conference papers. The review process is to receive a paper, record its information, check its eligibility, distribute the paper to the selected reviewers, have the reviewers perform the reviews and collaborate in producing a joint review document, and finally forward it to the editor, if accepted.

4.2.2 Modeling

Once the application process is well-defined, the next stage is to specify that in our model. This is done in steps, which are as follows.

- Step 1 is to **identify activities**. Based on the above description, the whole process is divided into tasks, such as, receiving a paper, checking its eligibility, rejecting the non eligible paper, returning it back to the author, if corrections are to be made, selecting reviewers for the eligible paper, forwarding it to reviewers, preparing a joint review report based on all the individual review reports, and forwarding the accepted paper to the editors For each task, we define an activity class in our model. These activity classes are denoted by rectangular boxes in our workflow diagram. The behavior of each activity class is defined by a number of methods and knowledge rules. Since, the review task is very complex, it is further decomposed into a number of activities, such as, reviewing the structure, layout, and style, reviewing the contents, and preparing a review report. All these activities are defined in another schema into which the *review* activity decomposes. A set of related activities are grouped into a view, if their actions represent the collective responsibility of a group of users playing a certain role. In this application, we define 3 views, administrator's view, reviewer's view, and expert's view. In the administrator view, all the activities which have to be done (or activated) by an administrator can be framed. Similarly, the other two views are defined. For each view, we associate a role, such as, administrator, expert, and reviewer. All the users of this model should play at least one of these roles.
- Step 2 is to **identify dependencies** between activities identified in step 1. For example, once the first activity, i.e., receiving a paper, is finished, then it has to be checked for eligibility. Thus, there should be a sequential association from activity *PaperReceival* to activity *Eligibility Check*. Following this approach, we need to establish relationships between all the tasks.
- Step 3 is to use a GUI tool to **draw a workflow diagram** based on information from steps 1 and 2. The workflow diagram for review paper application schema is shown in Figure 11.
- Step 4 is to **specify** this workflow diagram in our **language using WFDL**. The language can be automatically generated based on the information con-



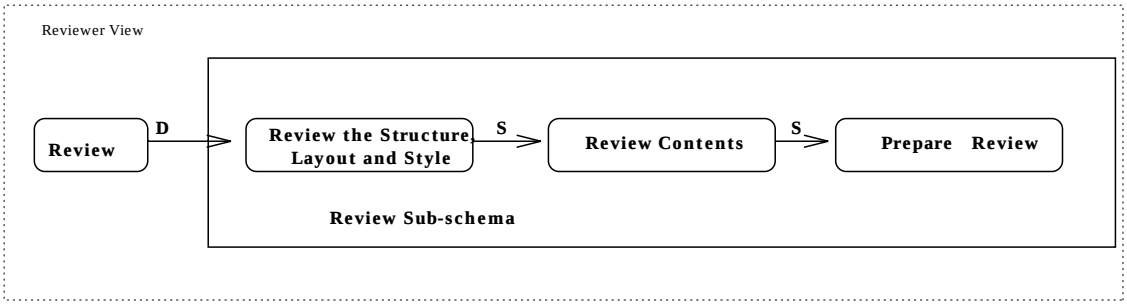


Figure 11: Workflow Diagram For Paper Review Application Process

tents of workflow diagram. The schema for this example is given in Appendix A.3.

These 4 steps complete the workflow modeling stage.

4.2.3 Writing Simulation Code

The above modeling step results in a process model of the workflow. It is not an executable model until all the methods associated with all the activity classes have been implemented in executable code. For the purpose of simulating the process model to identify potential bottleneck(s) and to test its performance, simulation code (not the real programs) can be written for the methods and some estimated time information for performing the activities can be specified by the process modeler. This step produces an "executable process model" of the workflow.

4.2.4 Generating Workflow Process Controller

The executable process model is then compiled and translated into executable code. The executable code serve as the workflow process controller which activates information systems and interact with the agents involved in the workflow. Whenever, a request for a paper (to be published in the conference), is made, the process controller is enacted (executed). The main method of the starting activity class is executed first. The subsequent processing of the process controller follows the semantics of the association types and the structure of the process model until a finish activity is reached. The WFMSS will produce a report of the execution. The report includes information of the activity instances that have been executed, general information about the process, metrics of individual activations, metrics for the whole enactment, status, etc. Reports can also be generated for views, roles, and users individually. (See Appendix A.4 for a sample report).

4.2.5 Analysis

Analysis can then be made from the reports that were generated for different enactments. If necessary, changes can be made to the process model by redesigning and re-implementing it. This whole cycle is repeated until a desirable executable process model is obtained. Finally, when the model with the desired output is obtained, the corresponding process controller can be used to control and coordinate the real life application or business activities.

4.2.6 Execution of the Process Model

Once a request for a workflow enactment is made, first, an instance of the project class that is associated with this WF Schema is created. The new project instance is initialized and activated. This project instance creates a new instance of the *start* activity of the WF Schema. The new *start* activity instance is initialized and activated. Thus, the enactment process is started. Once the enactment starts, other activities are created and executed following the semantics of their association links. All during the enactment, different metrics are recorded, and activity instances

status is changed based on its execution stage. The whole process can be explained in-detail using the model (Conference Paper Review Model) under consideration.

If a paper to be submitted to the conference is received, then an instance of project class P (the project class associated with this schema) is created and the status is set as inactive. The *startExec* method of this instance is called, before that the status of this instance is changed to active. The *startExec* method creates an instance of the activity class *paperReceival* and also calls the *activate* method of this instance. Immediately after an activity instance is created, the status of it is set to inactive and just before the method *activate* is called the status is set to active. The *activate* method does the intended action of the activity, i.e, updates its database with the paper title, author, reference id, etc. The code that enforces this is in the method *activate* of the activity class. Once the method is executed, the status is set to finish. Since there is a sequential association in the activity class *paperReceival*, the rule associated with this association type is triggered after the method *activate* is executed. This rule will create an instance of the activity class *EligibilityCheck*, passes the enactment Id value to the new instance, and finally activates this instance. The *activate* method of the activity class *EligibilityCheck* checks whether the paper meets all the requirements. If so, an instance of the activity class *Eligible* is created and activated. If the paper does not meet the requirements, then an instance of the activity class *NotEligibile* is created and activated. If it meets the requirements, but some changes have to be made, then an instance of the activity class *ReturnForCorrections* is created and activated. Once the paper is accepted, then reviewers for the paper are identified and the paper is forwarded to them for review. This action is done by the activity class *ForwardReport*. Since the review process is complex, it is decomposed into finer activities defined in another schema *ReviewPaperScehma*. Enactment of this schema follows the same line as that of the schema *ConPaperSchema*. Reviews made by different reviewers are collected and a joint report is made by the activity class *JointReviews*. If the paper is selected, then this activity will send it to the editor for publishing. Otherwise, the paper is sent back to the author informing him that it was not accepted. This action is performed by the activity *Rejected*. Thus, the request is served.

5 THE EXTENSIBLE KERNEL OBJECT MODEL (XKOM)

In this section, we explain how we extend an extensible kernel object model (XKOM) to include association types which are used to model the dependency relationships among activities of a process model. In Subsection 5.1, we describe the features of XKOM. In Subsection 5.2, we briefly explain the meta-model (model of the kernel model) and the extended meta-model respectively. XKOM is important because, our extensions are based on this model. In Subection 5.3, we discuss the concept and technique of model extensibility.

5.1 XKOM

The requirements of an extensible kernel object model, can be summarized as follows:

1. The kernel data model [AGR89] must provide a set of basic modeling constructs. Ideally, these constructs represent a set of core constructs that are common to the range of existing object and semantic data models. In our kernel data model, we provide the capability to define the structural and behavioral properties of objects and the inheritance mechanism.
2. The kernel model must be extensible. It must provide mechanisms which allow a knowledge base customizer (KBC) to declaratively extend the kernel model to include other modeling constructs required by an enterprise.

The basic structural and behavioral constructs of XKOM and the diagrammatic representation of an XKOM schema are explained in the following subsubsections.

5.1.1 Structural Constructs

Objects are the basic units in XKOM. XKOM supports two fundamental types of objects: self-naming objects and system-named objects. Self-naming objects (e.g., integer 5 or string "John") have a basic data type such as integer, string, etc. System-named objects are used to model entities of interest in the application and each is assigned a globally unique object identifier (OID).

A XKOM class is an abstraction that encapsulates the structural and behavioral semantics of a set of *like* objects. Two general types of classes are supported in XKOM: *Entity (E-)classes* and *Domain (D-)classes*. An *E-class* represents system-named objects. A *D-class* represents self-naming objects. *E-classes* have an associated set of materialized object instances, whereas *D-classes* serve to declare domains of values for materializing *E-class* objects.

The notion of relationships or associations among objects is an important concept in data modeling [CHE76, SU89]. The salient feature of the XKOM in this respect is that it supports the notion of different types of class associations corresponding to the different types of relationships in the real world. XKOM provides two fundamental types of associations, namely generalization (G) and aggregation (A) [SMI77]. Generalization represents the super class-sub class relationship ("is-a" relationship) between two classes. Aggregation is an association that defines an "a-part-of", "a-function-of" or "is-characterized-by" relationship between a defining class and a constituent class.

5.1.2 Behavioral Constructs

XKOM provides two forms of behavioral abstractions: methods and rules.

- **Methods:** Like conventional object-oriented models, methods are provided in XKOM as a procedural form of behavioral abstraction. Methods represent the interface to the objects of a class. Methods are specified as a part of the class specification.

- **Rules:** Rules are provided as a declarative form of behavioral abstraction in XKOM. The declarative nature of rules allows for convenient and flexible specification of behavioral semantics. Rules in XKOM are event-condition-action-alternative action (ECAA) rules.

5.1.3 Summary

The kernel data model (XKOM), minus the rule specification component, is a "conventional" message passing object model. Also, it is a basic object model similar to the one adapted by the Object Management Group (OMG) for the specification of the Object Request Broker (ORB) ([OBJ92, COR92]) . In other words, it has a small number of basic concepts: objects, classes, associations (aggregation and generalization), methods, and rules. What it does not have is a set of semantically-rich constructs found in some of the existing data models (e.g., see survey in [HUL87].), constraints (such as primary key constraint, cardinality constraint, non-null constraint, etc.) or association types (such as "relationship" in the ER model) found in database management. However, the addition of the rule specification component in XKOM allows the semantics of these and other new constraint and association types to be explicitly defined by rules and be incorporated into the application schema by means of a *model extensibility* mechanism.

5.2 Meta-Model

The meta-model is shown in Figure 12. We have shown in the figure only the parts of the schema which are relevant to our discussion. An interested reader should refer to [YAS91a, JAV92] for a more fundamental treatment of the meta-model. As described in Subsection 5.1, in XKOM, everything is an object. Therefore, the root of the class hierarchy is the meta-class *Object* which represents that concept. An object is either a system-named or a self-named object, represented by the subclasses (of *Object*) *E-Class Object* and *D-Class Object*. A set of like objects are grouped in a class, represented by the meta-class *Class*. The important meta-classes are discussed in the following subsubsections. The meta-model has been extended to support workflow constructs. The extended meta-model is shown in Figure 13.

5.2.1 Class

The meta-class *Class* encapsulates classes as objects. Objects in *Class* may include application classes or meta-classes. A *Class* has a class name, a set of methods, a set of rules and a set of associations.

As described in Subsection 5.1, there are 2 types of XKOM classes, represented by the subclasses *Domain (D-Class)* and *Entity (E-Class)*. Unlike *D-Class*, an *E-Class* has a set of instances of *E-Class Objects*.

5.2.2 Association

Another key construct of the XKOM is association, as represented by the meta-class *Association*. Objects in this class include all the associations among applications classes (e.g., all the A- and G-associations shown in Figure 12) and meta-classes.

Each association has a defining class and one or more association links (each association link connects to one constituent class). The A- and G-associations are binary associations and so they have only one constituent class. *Association* class has the attribute *assocType* which specifies the type (e.g., aggregation, generalization or others) of association. An instance of the meta-class *Association* relates a defining class to a set of constituent classes (through *AssocLink*). The attribute *definingClass* specifies the defining class to which the association object belongs. The kernel system supports A- and G-associations. Thus, *Association* has been subclassed into *Aggregation* and *Generalization*. Other association types are introduced through model extensions.

The class *Association* and its subclasses are important meta-classes, since their specifications define the structural and behavioral properties of all association types in the system, and their implementation realize these properties.

5.2.3 AssocLink (Association Link)

An association has one or more links. Each link is represented by an object in the class *AssocLink*. This class contains the name of the link and the constituent class. The attribute *definingAssoc* specifies the association to which the association link object belongs.

5.2.4 Rule

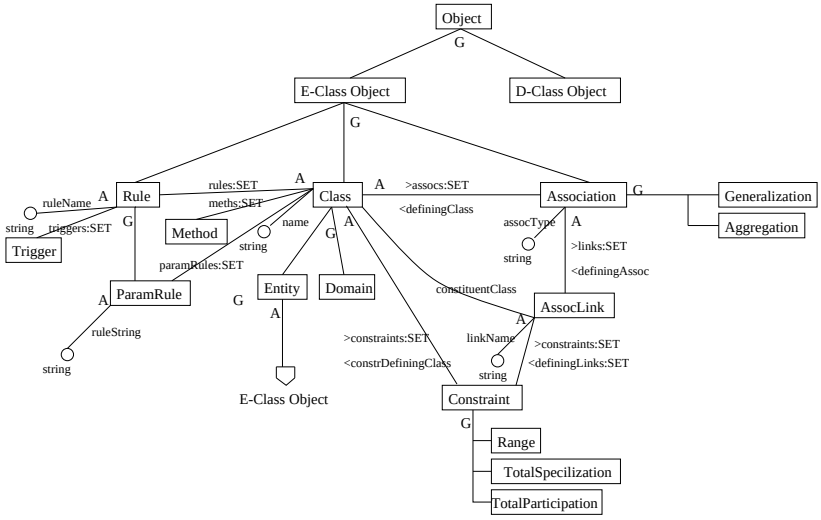
All rules are stored as objects of the meta-class *Rule*. A rule has a rule name and a set of triggers. A trigger specifies the trigger time (before, after, immediate_after) and the name of the event (e.g., update).

5.2.5 ParamRule (Parameterized Rule)

A class can have a set of parameterized rules. The use of parameterized rules is one of the key mechanisms used for achieving model extensibility. This class is a subclass of *Rule*. A parameterized rule (in the form of a string) is stored in *ruleString*. This will be used by a rule binder to generate bound rules.

5.2.6 Constraint

A class may also contain a set of constraints. An association link also can have a set of constraints. So a constraint object can either be associated with a *Class* or with an *AssocLink*. If the constraint object is associated with a *Class*, then the attribute *constrDefiningClass* specifies the class with which the constraint object is associated with. If the constraint object is associated with an *AssocLink*, then the attribute *definingLinks* specifies the association link object with which the constraint object is associated with. The attribute *definingLinks* is set-valued because one constraint can be associated with more than one association link.



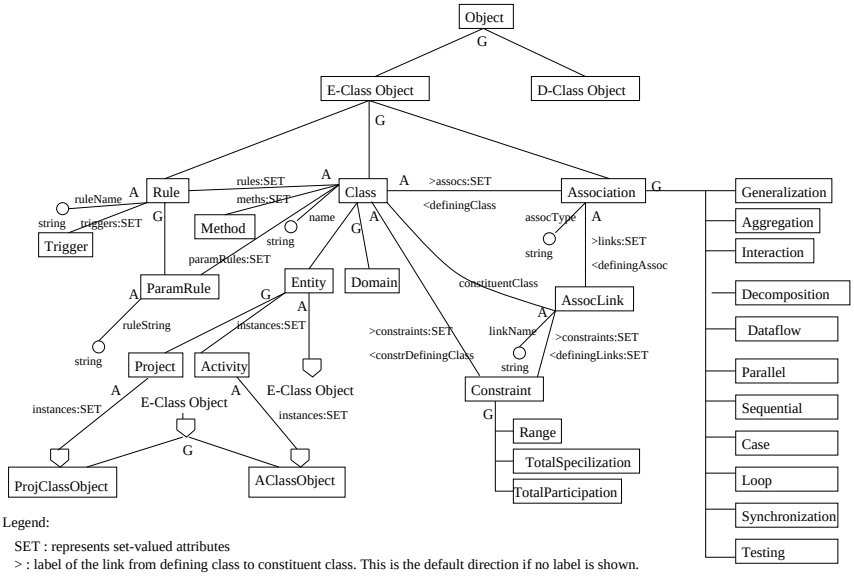
Legend:

SET : represents set-valued attributes
> : label of the link from defining class to constituent class. This is the default direction if no label is shown.
< : label of the link from constituentt class to defining class

: a connector to another class in the schema

Meta-Model

Figure 12: Meta-Model



Legend:

SET : represents set-valued attributes
> : label of the link from defining class to constituent class. This is the default direction if no label is shown.
< : label of the link from constituentt class to defining class

: a connector to another class in the schema

Extended Meta-Model

Figure 13: Extended Meta-Model

5.3 Model Extensibility

In this subsection, we describe how we achieve model extensibility in the OSAM*.KBMS. This is the procedure that is followed in adding new data and control flow associations to KBMS.

5.3.1 Approach

The semantics of a new construct (constraint type/association type) can be specified using rules. These rules, when triggered, enforce the semantics, thus supporting the new construct.

The users themselves can write rules inside their application schema to enforce the semantics of a construct. But if this construct is commonly used, then it is better to incorporate it in the underlying data model (e.g., XKOM) and the corresponding language (e.g., K). The advantages of incorporating a commonly used construct inside the language are :

- Users can readily make use of the construct, instead of writing rules for supporting it, which results in less development time and smaller code (so less maintenance).
- If different users write rules to support the same construct, then it results in replication of the same code.

As explained earlier, a new construct can be incorporated in the OSAM*.KBMS system by specifying its semantics using parameterized rules. When the user's application schema makes use of the new construct, the Rule Binder generates bound rules (by translating the corresponding parameterized rules) and incorporates them inside the user's classes (without the knowledge of the user) as if the user himself/herself has written those rules in the user class. The extended user's class (extended with rules that support the new construct) is compiled to enforce the semantics of the new construct. The block diagram of this process is shown in Figure 14.

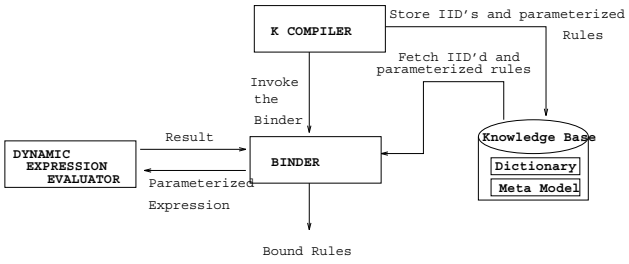


Figure 14: Block Diagram showing the process of Binding

5.3.2 Steps to be followed

- **Customizing the meta-model**

  This is the first step. The KBC studies the semantics of a new construct and writes parameterized rules that specify those semantics.

- **Compilation**

  When the user uses the new construct in his program, and when the program is compiled, the semantic specification of the construct is evaluated by the K compiler using the Dynamic Expression Evaluator, which results in the following

  - The OID of the new construct object is stored in the dictionary of OSAM*.KBMS (The OID's of all created objects, which contain parameterized rules, are stored in the dictionary).
  - The meta-model is populated with the relevant information. In this case, the new construct object that is created is stored and the association links between the association link object and the construct object are updated. At the end of compilation, the Rule Binder is invoked.

- **Binding**

  This is the third step. In this step, the Binder fetches the OID of the new construct object, that has been created, from the dictionary. It also fetches the corresponding parameterized rules from the meta-model. The Binder then processes those parameterized rules, with respect to the newly created construct object, to generate bound rules. These generated rules are incorporated into the user's class definition thus extending the user's class definition. The Binder invokes the K compiler to compile the generated bound rules.

- **Enforcing the Semantics**

  As already mentioned, the OSAM*.KBMS system has a rule processing facility and so is capable of processing the bound rules (enforcing the semantics specified by the rules).

  When the K compiler compiles a bound rule, it generates C++ code for processing that rule. This generated C++ code is compiled with a C++ compiler to form a library file. This library file is linked with the user's programs and the Query Processor. When the user program (or the Query Processor) is executed, the Rule Processor detects the occurrence of events and triggers the actions as specified in the bound rule. Thus the semantics of the new constraint or association type are enforced.

6 THE DESIGN AND IMPLEMENTATION OF A WFMSS

In this section we discuss the design and implementation of a Workflow Modeling and Simulation System (WFMSS). In Subsection 6.1, we explain the overview and logic of the design. In Subsection 6.2, the conceptual architecture of the WFMSS is discussed. In Subsection 6.3, justifications for the use of KBMS are cited. In Subsection 6.4, workflow enactment and association execution processes are discussed. In the rest of the section, implementation aspects are discussed.

6.1 Overview

Active database technology is particularly suitable for providing an operational model of workflow enactment. Active rules follow the *event - condition - action - alternate action (ECAA)* paradigm: events are typically changes to the database content, conditions are database accesses, and actions are arbitrary computations, possibly causing database changes. We describe how formal workflow descriptions can be used as input in order to semi-automatically generate both the schema of workflow data and the code of active rules for their management. In this operational model, workflow enactment is traced by means of manipulations of workflow data, and the behavior of the WFMSS is modeled by active rules. In addition, the active rule paradigm provides a convenient formalism for expressing reactive computations which normally are influenced by *events* generated outside of the WFMSS, such as exceptions or changes in the truth values of task preconditions. Execution of these rules is done by the rule processor module of the KBMS.

Basically, there are two types of rules :

- **Meta** (also known as **Generic**) rules are common to any WF schema; they are part of the kernel (meta model) of the WFMSS and are never modified. Since they are defined in the meta-classes, and hence stored in the meta model, they are called Meta rules. Such rules govern the functioning of the WFMSS, according to the state diagram and the semantics of activation of activities in a WF according to our model. For instance, one rule states that, when a new workflow schema is created, its starting task becomes ready.
- **Specific** (also known as **User-defined**) rules depend on the characteristics of the WF Schema; therefore, they are added/removed when a workflow schema is created/deleted.

Specific rules are coded by the Workflow Developer. Workflow Developer is the one, who specifies a real life process in our workflow model. It is presumed that he/she is well acquainted with our workflow modeling and simulation system. He/she, first, studies the real life process, then denotes it with a workflow diagram and finally specifies it in our workflow model using WFDL. Meta rules are of two types.

- General rules which implement the semantics of activation of activities. For example, one rule states that, immediately after the activation of an activity, call the *postExecution* method of that activity.
- Rules that define the semantics of the control and dataflow association types.

Both these types are defined in the meta model as parameterized rules (also called as rule templates), i.e., they are not bound to any specific WF schema. But, once a new WF schema is created, rules have to be specific and bound to this WF schema. In order to achieve this, first, rule templates are written and stored in the KBMS. Secondly, whenever a WF Schema is created, application specific rules are to be generated based on the parameterized rules that are embedded into the system. As the name suggests, these rules are specific to a WF schema and are bound to the objects defined in the WF schema. This translation of parameterized rules into application specific rules, called bound rules in our system, is done by the *Rule*

Binder. Helping rule binder in this translation is the *Dynamic Expression Evaluator* (DEE) that evaluates a **K** language expression dynamically.

Application specific rules along with generic rules provide the operation model for the workflow enactment. These rules along with other specifications are compiled by the Code-Generator to generate a workflow process controller. The workflow process controller is executed every time a workflow enactment is made. During it's execution, it interacts with KBMS and other application systems. Block Diagram of this compilation process is shown in Figure 15 and that of run time process is shown in Figure 16.

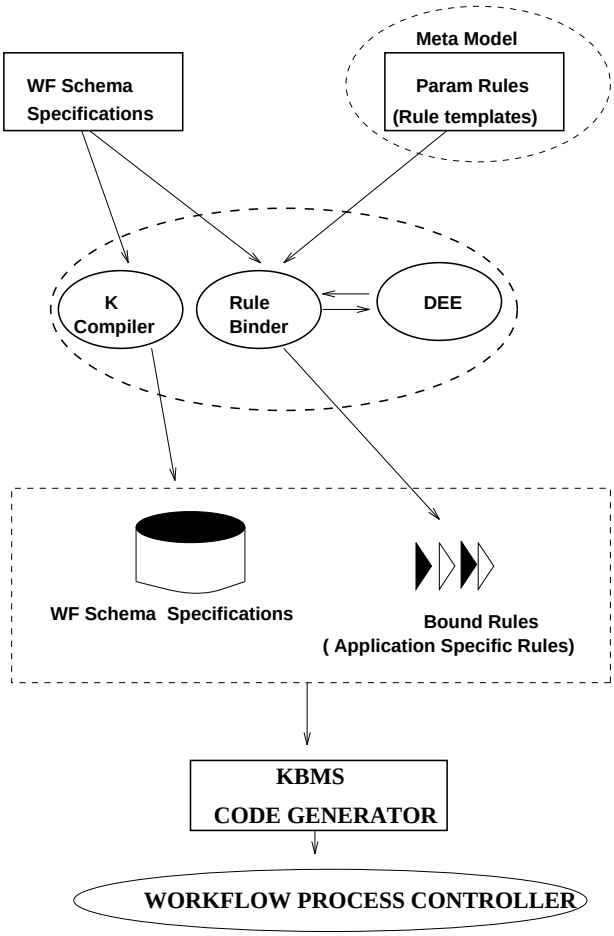


Figure 15: Workflow Compilation

6.2 Conceptual Architecture

The conceptual architecture of the system gives a conceptual overview of the system. It gives an overview of all the constructs in the system and dependencies among them. A block diagram depicting the conceptual architecture of the system is shown in Figure 17. Understanding the conceptual architecture gives a vivid picture of the enactment process. This figure shows all the constructs and their relationships in our WFMSS. The block diagram can be summarized as follows.

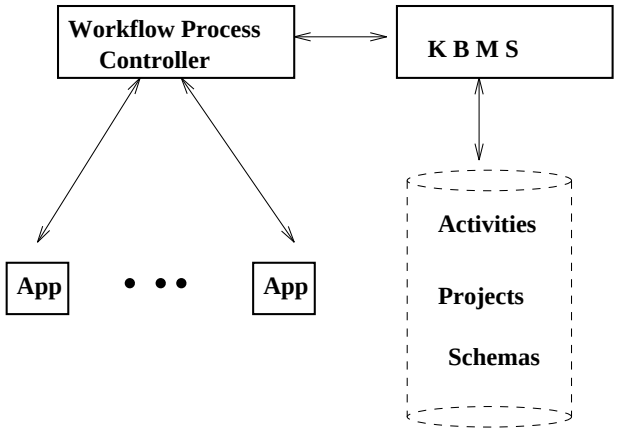


Figure 16: Run-time Process

The most important construct is a WfSchema, and for every WfSchema there is an associated *Project* that handles the enactment of the WfSchema. A WfSchema consists of a structure of Activities. Activities are inter-related and their relationships are specified by different associations. If an activity is complex, it can be decomposed and defined by another WfSchema. An activity consists of sub-activities, which are implemented in our system as methods. Each activity is modeled by an object class. As such, it can have a set of attributes, a set of methods and a set of user-defined rules. Data used in an activity are stored as attribute values. Every activity must be in at least one view and a view consists of any number of activities. Views can also overlap. For every view there is a corresponding role. A role is played by one or more users. A user can be in more than one role.

6.3 Features in OSAM*.KBMS

The reason why we chose to extend OSAM*.KBMS rather than developing a WFMSS from scratch is that, XKOM, underlying model of OSAM*.KBMS, has rich modeling features with which we can implement a WFMSS much easier. The following features of OSAM*.KBMS help in the development of the WFMSS.

- It is **reflexive**, i.e., the functionalities of the KBMS are "driven" by the meta model. Any change or extensions to the meta model will automatically alter the functionalities of the system.
- It supports **system** extensibility, i.e, architecture of the system is developed in a modular and open manner so that the software system components can be interchanged or easily extended. New modules can be added without internal modifications.
- It has **model extensibility**, i.e., the modeling constructs of the underlying data model can be extended and customized. Because of this facility, new association and constraint types can be added to the underlying model very easily. Cashing on this, new control and data associations and constraint types are added.

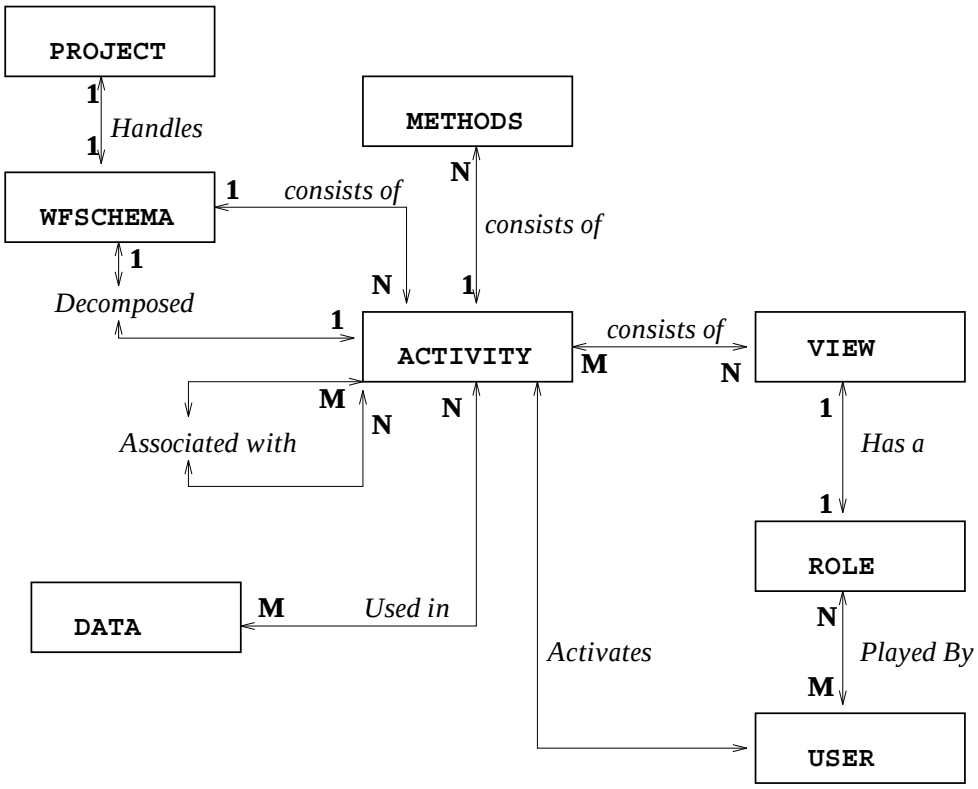


Figure 17: Conceptual Architecture

- It supports **rules**. As explained in Subsection 6.1, active database technology is well suited for providing the operational model for workflow enactment.
- Has querying ability.
- Persistence data and objects can be defined and declared.
- Transaction management is supported.
- Activity is a subclass of Entity Class in the extended meta model. Thus, each Activity class can have attributes, rules, associations and constraints which define the semantics of an activity.
- Has a unified object model, i.e., data associated with activities can be defined in the same modeling framework. Because of this, consistency between workflow and data models can be more easily verified. Also, testing and checking during reeenginering of the workflow model is made easy.

Weighing all these, we feel that it is justified to use OSAM*.KBMS as its underlying system.

6.4 Workflow Enactment and Association Execution Processes

In this subsection, we discuss the process (steps) for workflow enactment and the execution of a workflow model based on the different control association types that

link the activities of the model.

6.4.1 Workflow Enactment

Once a request for a workflow enactment is made, the following steps are followed in the sequence.

- An instance of the project class that is associated with a WF Schema is created.
- The new project instance is initialized and executed (i.e., the **startExec** method is called. This is the method in the project class that starts the workflow enactment).
- This project instance creates a new instance of the *start* activity of the WF Schema.
- The new *start* activity instance is initialized and activated (i.e., the **activate** method is called. This is the actual method that does the intended action of the activity).

Thus, the enactment process is started. Once the enactment starts, other activities are created and executed following the semantics of their association links.

6.4.2 Association Priorities & Execution

Control associations listed in Subsection 3.3.4 are once again classified as *sink* and *source* control associations. Only synchronization comes under source category, as the rule that defines the semantics of this association type is executed **before** an activity is activated. While all other control associations fall in sink category, as the rules that define the semantics of these associations are executed **after** an activity has been activated. In an activity, there can be atmost one decomposition, one dataflow, one source control association and one sink control association. We cannot have two sink control associations due to their conflicting semantics. And, if a synchronization association exists, then the rule that defines the semanctions of this association type is triggered **before** an activity is activated, this rule makes sure that all the predecessor activities have been activated. The rules that define the semantics of all other associations are executed **after** an activity has been activated, which is the defining class of these associations. If an activity has a decomposition association, a dataflow association, and a control association, first the decomposition association, then the dataflow and finally the control association are executed. The executions of different association types are described below.

Decomposition When a decomposition association is seen by the compiler, then an instance of the project class associated with the low-level WFSchema is created and then started. This project instance will create an instance of the *start* activity of the low-level WFschema and activates it. Thus, the low-level WfSchema is enacted.

Dataflow If there is a dataflow association from activity A to activity B, it means that upon the termination of the activity A, some data objects should be sent from A to B, and these data objects are available till the activation of B consumes them. The

communication is specified by a construct (constraint) FLOW_IS_FROM (see Appendix A), in terms of the data attributes of these activities . In the OSAM*.KBMS model, aggregation associations are used to define the data attributes of activities. Such attributes are used to describe an activity in terms of the data it manipulates. All the attributes are stored in the KBMS. If the data attributes' underlying domains are entity classes, the corresponding **oids** are passed and stored in the instances of the two associated activities. If the data attributes' underlying domains are domain classes, the corresponding values are passed and stored.

The construct FLOW_IS_FROM takes a string as its argument. The string specifies the source and the destination of the data flow. For example, if the string is "a1:b1;a2:b2,b3", then the value of a1 is copied to b1 and value of a2 is copied to b2 and b3.

Sequential If there is a sequential association from an activity A to an activity B, it means that upon the termination of the activity A, activity B is to be activated. If the activity B has a synchronization association in it, then there could be an instance of B which has been created by another predecessor class of B waiting for the completion of A. If such an instance exits, update that instance and if that instance is ready for activation according to the synchronization constraint, activate it. If activity A has a dataflow association link to B in addition to a control association link, then there is a possibility of an instance of B which has been created to receive the dataflow from A, but not yet activated. If such an instance exists, activate that instance. If there is no instance of B created, then create a new instance of B, initialize and activate it by executing the method *activate*.

Parallel If there is a parallel association from an activity A to activities B and C (or more), it means that upon the termination of A, both B and C can be activated in parallel. While activating activities B and C, we follow the same procedure as for the sequential association.

A construct **ANY_OF(x)** is imposed on this association type. It takes an integer as an argument. For example, if ANY_OF(2) is imposed on a parallel association having 3 (or more) constituent classes, then any two of the 3 (or more) constituent activities are executed.

Synchronization If there is a synchronization association from activities A and B (or more) to activity C, it means that upon the termination of both A and B, C is activated, and A and B should have the same *enactment identifiers (ids)*. Only the first predecessor activity instance (in this example either A or B) that has completed its execution will create an instance of the activity in which this association type has been defined (in this example it is C). All other predecessor activity instance will make an attempt to activate the new instance. The last predecessor activity instance that has completed its execution will actually activate this instance. The construct **ANY_OF(x)** can also be imposed on this association type. It takes an integer as an argument. For example, if ANY_OF(2) is imposed on a synchronization association having 3 (or more) constituent classes, and if any two of the 3 (or more) constituent classes have done with their executions, then only activate this instance.

Testing If there is a testing association from an activity A to activities B and C, it means that the control will branch to either B or C, depending on the TRUE or FALSE value of the predicate defined in the activity A. If the predicate is evaluated to TRUE, then the activity with the linkname *IfTrue* is activated, otherwise the other activity is activated. While activating B or C we follow the same procedure as for the sequential association. The predicate is specified by the clause (constraint) EXPRESSION_IS (see Appendix A), which takes an expression as its argument. The expression is evaluated to TRUE or FALSE.

Loop If there is a loop association from activity A to activity B, then a instance of activity B is activated as long as the *condition* associated with the loop is true. If there is an instance of B already created, but not yet activated, either because A has a dataflow link to B or B has a synchronization, then that instance is to be activated. Otherwise, create a new instance of B and activate it. The loop association also has an initialization clause and a loop increment clause for initializing and incrementing loop variable(s), respectively.

Case A Case association can have any number of links (successor activities). All but one will have BRANCH_EXPRESSIONs associated with them. The one without the expression is called a default link. BRANCH_EXPRESSION (see Appendix A) takes a predicate as its argument, which evaluates to TRUE or FALSE. Each link is considered one at a time, if the predicate associated with it is evaluated to TRUE then the activity associated with that link is activated. Since there could be more than one link for which the predicate returns TRUE, there can be any number of activities that are activated. If none of them is activated, and if the association has a default link, then the activity associated with the default link is activated.

6.5 Extensions to Meta Model

The meta-model (Figure 12) has been extended so as to allow the Workflow Developer to model workflow processes in our system using the new constructs. Many new meta-classes have been added to the underlying model. Some changes have been made to the existing meta-classes also.

6.5.1 Activity Class Type

As discussed in the earlier section, to model a real life task, we have introduced Activity Class. Definition of this class is done in a two layered approach. All the attributes , methods and rules that are common to all instances of all activities are defined in the class *AClassObject*. And attributes, rules and methods that characterize an activity class are defined in the class *Activity*. Each instance of the class Activity is the definition of an Activity class. The class AClassObject is a sub_class of class *EClassObject*, thus inherits all the properties of an EClassObject. The class Activity is a sub_class of class Entity. The base_class for class Activity is class AClassObject. Hence, for every instance of an Activity class, an instance of AClassObject is created.

Important Attributes in the class AClassObject are:

- **enactId :** Enactment Id is stored in this attribute. Its value is the Id of the project instance that has created and activated this instance.
- **enactStatus :** The status of the activation of this instance. It can be INACTIVE, SUSPEND, ACTIVE, INHIBITED or FINISH.
- **activationId :** A unique number to identify this instance. It is the oid of the instance (object).
- **parActivatedId :** Activation Id of the activity that has created this instance.
- **createTime :** The system time at which this activity instance has been created.
- **activateTime :** The system time at which this activity instance has been activated.
- **finishTime :** The system time at which this activity instance has finished its activation.

Important attributes in the class Activity are:

- **desc :** Brief description about the activity.
- **vName :** This is an aggregate attribute. Has the names of all the views in which this activity is present.
- **manType :** Type of activity (Manual, Interactive, Batch or Dummy).

Important methods in AClassObject :

- **preInitialize :** This method is called immediately after the activity instance is created. Does some initializations, such as, assigning a unique activationId, initializing all attributes to their default values, etc.
- **preExecution :** If an activity has a synchronization association, then this method makes sure that all the predecessor activities have been activated before activating this activity instance.
- **activate :** This is the actual method that does the intended action of the activity.
- **postExecution :** This method finishes the execution sequence.

6.5.2 Project Class Type

Just like Activity Class, Project Class is also defined in a two-step manner. All the attributes, methods, rules that are common for every instance of all project classes are defined in the class *ProjClassObject*. And attributes, methods, and rules that *characterize* a project class are defined in the class *Project*. By making ProjClassObject as a base class of Project Class, for every instance of a Project class, an instance of ProjClassObject is created. The reason for implementing Project as a class type rather than entity class is, the name of the *start* activity varies from one WF schema to other, thus, we can not hard-code it in the rules of the project entity class. We need this because, once a new project instance is started, we should create a new instance of the *start* activity and activate it. And also, **K** language

doesn't support *dynamic* binding. The remedy for this is to define it as a class type and instead of defining these semantics in rules, we can define it in parameterized rules. While translating these rules into bound rules, we can extract the name of the *start* activity.

Important methods in this class are :

- **initialize :** This method is called immediately after a project instance is created. Does some initializations, such as, assigning enactId a unique number, initializing all attributes to their default value, etc.
- **assignParActId :** Decomposition association calls this method, when a project instance has to be created for enacting a low-level schema. Assigns the activity name and Id to the new project instance.
- **preExecution :** Before the workflow enactment is started this method is called.
- **startExec :** This is the main function that starts the workflow enactment.
- **postExecution :** After the workflow enactment is finished this method is called.

6.5.3 WF Schema Type

To support the construct WF Schema, we have introduced a class WFSchema. It is a sub_class of Class Schema. The definition is as follows :

```
define WFSchema : ClassType
   associations:
     public:
     Generalization <- { Schema };
     Aggregation
     {
        sActivity    : String;
        fActivities  : Set<String>;
     };
end;
```

As explained in Subsubsection 3.3.1, WF Schema has two attributes, *start* and *finish* activities. The attribute *sActivity* represents the *start* activity and the attribute *fActivities* represents the set of finish activities in a WF Schema.

6.5.4 AssocLink

Many new methods, rules, and parameterized rules have been added to this class. Most important change of all is the addition of parameterized rules. Thus, for every association link of an association type, bound rules are generated. Though this is an overhead, the reason for this implementation is that, most of the control and data association types have more than one assoc link and for each of them bound rules have to be generated. If the parameterized rules were written in the Assoc class rather than AssocLink class, then it would have been impossible and inconsistent to

generate different bound rules for each association link based on one parameterized rule in Assoc class. So parameterized rules were written in this class for all data and control association types which can have more than one association link. Since synchronization and sequential have only one association link each, parameterized rules for these were not written in this class to improve the efficiency of the system.

New methods that have been added are :

- **defAssocType :** Returns the name of the association type that defines this association link.
- **defClassName :** Returns the name of the class that defines this association link.
- **linkName :** Returns the link name of the association link.
- **constClassName :** Returns the constituent class name.

6.5.5 Assoc

Some new methods that have been added to this class are as follows.

- **getCondExpr :** If there is a CONDITION clause (constraint) associated with an association, then, this method returns the predicate, the CONDITION clause has takes as it's argument. Otherwise, returns an *error*. This method is used in the param rule that defines the semantics of a Loop association type (see Appendix A.2).
- **getInitExpr :** If there is a INITIALIZATION clause (constraint) associated with an association, then this method returns the predicate, the INITIALIZATION clause has takes as it's argument. Otherwise, returns an *error*. This method is used in the param rule that defines the semantics of a Loop association type (see Appendix A.2).
- **getLoopIncrExpr :** If there is a LOOP_INCR clause (constraint) associated with an association, then this method returns the predicate, the LOOP_INCR clause has takes as it's argument. Otherwise, returns an *error*. This method is used in the param rule that defines the semantics of a Loop association type (see Appendix A.2).
- **noOfSynchLinks :** If the type of this association is Synchronization, then this method returns the number of synchronization predecessors to this activity. Otherwise, this method returns an *error*.

6.6 Extensions to K Compiler

In order to support workflow constructs and control and dataflow associations type, a few extensions have been made to **K** compiler also. Brief descriptions of the changes are given below.

6.6.1 New Domain Type

TimeStamp Activation of activities sometimes depends on the time at which they are created. To keep track of the time at which these instances are created and also to calculate and analyze metrics for simulation purposes we have introduced this new domain type.

6.6.2 New Entity Classes

User A new entity class named User has been created to support the *User* agent construct. Attributes of this class are userName and userId.

View To support the concept of *View*, this new entity class has been created. Attributes of this class are viewName, viewDesc and viewId.

Role The *Role* construct is implemented using this entity class. It's attributes are roleName and roleId.

ROLE_VIEW As mentioned earlier, there is a 1:1 relationship between a role and a view. A role is associated with a view by creating a new instance of this class This instance contains a roleId and a viewId. In this entity class, role and view are associated by an *interaction* association.

USER_ROLE As mentioned earlier, there is a M:N relationship between a role and a user. A user is made a part of a role by creating a new instance of this class which contains the userId and the roleId. In this entity class, role and user are associated by an *interaction* association.

6.7 Parameterized Rules for New Association Types

As mentioned earlier, rule templates for all the new association types are written and embedded into the kernel. Rule templates in our system are called parameterized rules. As the name indicates, they are generic and not bound to any specific

object. In this subsection, we explain in detail the parameterized rule for sequential association. Parameterized rules for other associations are listed in Appendix A.2.

6.7.1 Parameterized Rule for Sequential Association

```
 1    param_rule Sequential
 2    bind_classes @definingClass()
 3    triggered immediate_after postExecution()
 4    action
 5        if ( askForActivation() = FALSE ) then
 6           return;
 7        end_if;
 8        context c:@constituentClasses()
 9           where ( (c.activityStatus = "INACTIVE") and
10                   (c.synchSet.member(this.actName) <= 0) and
11                   (c.synchFlag = 1) and
12                   (c.aSchName = this.aSchName) and
13                   (c.enactId = this.enactId) )  do
14             c.synchSet.insert(this.actName,0);
15             c.parActivatedId.insert(this.activationId,0);
16             c.preExecution();
17             return;
18        end;
19        context c:@constituentClasses()
20            where ( (c.activityStatus = "INACTIVE") and
21                   (c.aSchName = this.aSchName) and
22                   (c.parActivatedId.member(this.activationId) > 0) and
23                   (c.enactId = this.enactId) )  do
24              c.preExecution();
25              return;
26        end;
27        this.seq := @constituentClasses().pnew();
28        if ( this.seq.aSchName = this.aSchName)  then
29          if ( this.seq.synchFlag = 1) then
30             this.seq.synchSet.insert(this.actName,0);
31          end_if;
32          this.seq.initialize(enactId,activationId);
33          this.seq.preExecution();
34        else
35          this.seq.del();
36        end_if;
37    end;
```

6.7.2 Explanation

Line 1 is the header of the parameterized rule, wherein the name is declared. Line 2 says that this parameterized rule has to be bound to the activity which has a sequential association with another activity. Line 3 tells when this rule has to be triggered: Immediately after the post_execution method of the activity in which this association is specified. The remaining part of the param_rule is the body of the rule that has to be executed. Lines 5-7 check whether the enactment is running in the simulation mode or the interactive mode. If it is in the interactive mode,

then the system interacts with the user who activates the instance. Otherwise, it continues. Lines 8-18 check whether the successor activity has a synchronization association in it or not. If it does not have a synchronization association, execution of the rule continues. Otherwise, it checks whether there exists a successor activity instance having the same enactment Id as this activity instance, whose activity status is "INACTIVE", and waits for this activation to complete. If such an instance exists, update the successor instance indicating that this instance is done. If all the predecessors activity instances have been activated then activate the successor activity instance. Lines 19-26 check whether there exists a successor (constituent class/Activity) activity instance that has already been created, either because of a dataflow association in this activity or due to a synchronization in the successor activity and it's activity status should be "INACTIVE". Moreover, it should have the same *enactment Id* as this activity instance, and the activity instance that has created this instance should be the predecessor activity instance. If one such activity instance is found, then activate that one. Finally lines 27-37 create a new instance of the successor activity, and initialize it and start its activation. Thus, the semantics of sequential association are enforced.

6.8 Summary

Characteristics of the WFMSS under discussion can be summarized as follows :

- Unlike traditional WFMSS, there is no server engine.
- Workflow enactment is by active rules, i.e., the semantics of data and control association types are defined by rules.
- Reconfiguring the workflow model is easy, i.e. statically changing the model is easy. We only need to compile only the part that has been changed or modified.
- Generates a report for each workflow enactment. Based on the report we can analyze the model and enhance it accordingly.
- Easily extendable, i.e, new control associations can be added easily as the underlying model is extendable.

7 SUMMARY, CONCLUSIONS AND FUTURE WORK

In this report, we present the design and implementation of a workflow modeling and simulation system (WFMSS) based on an extensible object-oriented knowledge base management system, OSAM*.KBMS. In particular, we added modeling constructs so that a real life process can be specified, simulated, executed, and reengineered in our system. For every "participant" in a real life workflow process, a corresponding construct has been added, and because of this one-to-one correspondence, workflow process modeling is simplified. For a user to perceive the model easily, we can denote it with a workflow diagram using the graphical notations provided for various constructs. Once the process is modeled, it is specified in our system using the workflow definition language (WFDL).

Unlike traditional workflow modeling and simulation systems, which follow the *interpretive* approach for enacting a workflow, we follow the *compiled* approach. The modeling tool of the KBMS is used to produce a workflow process model, and the model is **compiled** to generate a workflow process controller for controlling and coordinating the application process. The workflow process controller is executed every time a request for an enactment is made, and the ensuing results are obtained and analyzed. Based o n the analysis, the real life process can be re-modeled for better performance. Workflow enactment in our system is based on **active rules**. This is because, active database technology is particularly suitable for providing an operational model of workflow enactment.

Workflow modeling and simulation is still in its infancy and we feel that our work constitutes some new ideas and implementation techniques for improving the features offered by workflow modeling and simulation systems. By no means that our work is exhaustive. A lot more efforts are needed both from research and implementation point of view before workflow modeling and simulation systems become as commonly used as database systems in enterprise management.

Extensions that could be done to our system in the future are.

- Dynamic changes in schema specifications.
- Adding exceptions to activities.
- Parallel execution of activities.

A SYNTAX AND PARAMETERIZED RULES FOR WORKFLOW ASSOCIATIONS

A.1 Syntax of Workflow Associations Using Examples

A.1.1 Decomposition

The WFDL statement to specify a decomposition association type is given below with an example. This association type has only one association link. The name of this link is *decomp*, and this is a reserved word in our language. Constituent class should be of schema type.

```
define  A  : Activity in <WFSchemaName>
   associations:
     Decomposition
     {{
         decomp : S1;
     }};
end;
```

A.1.2 Dataflow

The WFDL statement to specify a dataflow association type is given below with an example. This association type can have any number of association links. Constituent classes should be of activity types. For each association link, a constraint, FLOW_IS_FROM, is associated. This constraint, takes a string as it's argument, which when parsed, specifies the objects that are flowing and their direction of flow.

```
define  A  : Activity in <WFSchemaName>
   associations:
     Dataflow
     {{
         bb : B where  FLOW_IS_FROM("a2:b1;a3:b2,b3");
         cc : C where  FLOW_IS_FROM("a1:c1");
     }};
end;
```

A.1.3 Sequential

The WFDL statement to specify a sequential association type is given below with an example. This association type has only one association link. The name of this link is *seq*, and this is a reserved word in our language. Constituent class should be of activity type.

```
define  A  : Activity in <WFSchemaName>
   associations:
     Sequential
     {{
         seq : B;
     }};
end;
```

A.1.4 Parallel

The WFDL statement to specify a parallel association type is given below with an example. This association type can have any number of association links. Constituent classes should be of activity types. If needed, a constraint, ANY_OF, can be specified on this association type.

```
define  A  : Activity in <WFSchemaName>
   associations:
     Parallel
     {{
         branch1 : B ;
         branch2 : C ;
         branch3 : D ;
     }};
end;
```

A.1.5 Synchronization

The WFDL statement to specify a synchronization association type is given below with an example. This association type can have any number of association links. Constituent classes should be of activity types. If needed, a constraint, ANY_OF, can be specified on this association type.

```
define  C  : Activity in <WFSchemaName>
   associations:
     Synchronization
     {{
         branch1 : A ;
         branch2 : B ;
     }};
end;
```

A.1.6 Testing

The WFDL statement to specify a testing association type is given below with an example. This association type has only two association links. And their link names are *IfTrue* and *IfFalse*, and both these are reserved words. Constituent classes should be of activity types. A constraint, CONDITION, is associated with this association. This constraint takes an expression as it's argument, which evaluates to true or false value.

```
define  A  : Activity in <WFSchemaName>
   associations:
     Testing
     {{
        IfTrue  : B;
        IfFalse : B;
     }} where { CONDITION(a=b) };
end;
```

A.1.7 Case

The WFDL statement to specify a case association type is given below with an example. This association type can have any number of association links. Constituent classes should be of activity types. For all but one association link, constraints,

BRANCHEXPR, are associated. This constraint, takes an expression as it's argument, which evaluates to true or false value. The association link without the constraint is known as *default* link.

```
define  A  : Activity in <WFSchemaName>
   associations:
     Case
     {{
         branch0 : B where { BRANCHEXPR(cstmt=2) };
         branch1 : C where { BRANCHEXPR(cstmt=3) };
         branch2 : D;
     }};
end;
```

A.1.8 Loop

The WFDL statement to specify a loop association type is given below with an example. This association has only one association link. And, the name of the link is *body*, and this is also a reserved word in our language. Associated with this association type, we have, INITIALIZATION, INCR_IS, and CONDITION constraints.

```
define  A  : Activity in <WFSchemaName>
   associations:
     Loop
     {{
        body :  B;
     }} where {INITIALIZATION(a:=1),INCR_IS(a:=a+1),CONDITION(a > 5)};
end;
```

A.2 Parameterized Rules for Workflow Associations

A.2.1 Decompostion

```
param_rule decomp_rule
bind_classes @definingClass()
triggered immediate_after activate()
action
  local
    _kpId : Integer,
    _kaId : Integer;
  begin
    _kpId := this.enactId;
    _kaId := this.activationId;
    @getCorProj().pnew() { assign(_kpId,_kaId) , startExec() };
  end;
end;
```

Brief Description : Creates a new instance of the project class associated with the sub-schema. The method, *getCorProj()*, retrieves the name of the project class associated with the sub-schema. The new project instance is assigned with enactId and name of the activity instance in which this association was specified. Finally, this project instance is started (i.e., the start method is called).

A.2.2 Dataflow

```
param_rule data_flow_rule
```

```
bind_classes @defClassName()
bind_if @defAssocType() = "DataFlow"
triggered before postExecution()
action
   if ( askForActivation() = FALSE ) then
            return;
   end_if;
   ext c: @constClassName()
      where ( (c.activityStatus = "INACTIVE") and
             (c.synchSet.member(this.actName) <= 0) and
             (c.aSchName = this.aSchName) and
             (c.synchFlag = 1) and
             (c.enactId = this.enactId) )  do
         c.synchSet.insert(this.actName,0);
         c.parActivatedId.insert(this.activationId,0);
         @getDataFlowStmt("c",getDataFlowExpr());
         return;
    end;
    this.@linkName() := @constClassName().pnew();
    if ( this.@linkName().aSchName = this.aSchName) then
         this.@linkName().initialize(enactId,activationId);
         if ( this.@linkName().synchFlag = 1) then
            this.@linkName().synchSet.insert(this.actName,0);
         end_if;
         @getDataFlowStmt(linkName(),getDataFlowExpr());
    else
         this.@linkName().del();
    end_if;
end;
```

Brief Description : This rule check whether the successor activity has a synchronization association in it or not. If it does not have a synchronization association, execution of the rule inues. Otherwise, it checks whether there exists a successor activity instance having the same enactment Id as this activity instance, whose activity status is "INACTIVE". If one such instance exists, then the data flow statements given as an argument to the constraint FLOW_IS_FROM are extracted from the kernel by the method *getDataFlowStmt()*. Otherwise, create a new instance of the successor activity and extract the dataflow statements.

A.2.3 Parallel

```
param_rule parallel
bind_classes @defClassName()
bind_if @defAssocType() = "Parallel"
triggered immediate_after postExecution()
condition (noParaLinks < @getParaConst() )
action
   if ( askForActivation() = FALSE ) then
            return;
   end_if;
   noParaLinks := noParaLinks + 1;
   ext c:@constClassName()
      where ( (c.activityStatus = "INACTIVE") and
             (c.synchSet.member(this.actName) <= 0) and
             (c.synchFlag = 1) and
             (c.aSchName = this.aSchName) and
             (c.enactId = this.enactId) )  do
       c.synchSet.insert(this.actName,0);
```

```
            c.parActivatedId.insert(this.activationId,0);
            c.preExecution();
            return;
         end;
         ext c:@constClassName()
            where ( (c.activityStatus = "INACTIVE") and
                     (c.parActivatedId.member(this.activationId) > 0) and
                     (c.aSchName = this.aSchName) and
                     (c.enactId = this.enactId) )  do
               c.preExecution();
               return;
         end;
         this.@linkName() := @constClassName().pnew();
         if (this.@linkName().aSchName = this.aSchName) then
            if ( this.@linkName().synchFlag = 1) then
                 this.@linkName().synchSet.insert(this.actName,0);
            end_if;
            this.@linkName().initialize(enactId,activationId);
            this.@linkName().preExecution();
         else
            this.@linkName().del();
         end_if;
      end;
```

Brief Descripiton : The parameterized rule for this association type is almost the same as the one defined for sequential association type, except for the fact that it is bound to all the association links in the association. Because of this, the parameterized rule is defined in the class AssocLink, whereas the one defined for sequential association type is defined as a new association type. The other difference is the existence of the condition clause, this is needed when we have an ANY_OF constraint on this association type.

A.2.4 Loop

```
param_rule loop_rule
bind_classes @definingClass()
triggered immediate_after postExecution()
action
 local
  flag : Boolean;
 begin
   if ( askForActivation() = FALSE ) then
      return;
   end_if;
   if ( @noOfLinks() != 1 ) then
     "\n  Invalid Specification of Loop Association \n".display();
     abort;
   end_if;
   @getInitExpr();
   while not ( @getCondExpr() )  do
     flag := FALSE;
     ext c:@constituentClasses()
        where ( (c.activityStatus = "INACTIVE") and
               (c.synchSet.member(this.actName) <= 0) and
               (c.aSchName = this.aSchName) and
               (c.synchFlag = 1) and
               (c.enactId = this.enactId) )  do
          if ( flag = FALSE ) then
```

```
            flag := TRUE;
            c.synchSet.insert(this.actName,0);
            c.parActivatedId.insert(this.activationId,0);
            c.preExecution();
          end_if;
      end;
      if ( flag = FALSE) then
        ext c:@constituentClasses()
          where ( (c.activityStatus = "INACTIVE") and
                  (c.aSchName = this.aSchName) and
                  (c.parActivatedId.member(this.activationId) > 0) and
                  (c.enactId = this.enactId) )  do
           if ( flag = FALSE ) then
             flag := TRUE;
             c.preExecution();
           end_if;
        end;
      end_if;
      if ( flag = FALSE ) then
        this.body := @constituentClasses().pnew();
        if ( this.body.aSchName = this.aSchName) then
          if ( this.body.synchFlag = 1) then
             this.body.synchSet.insert(this.actName,0);
          end_if;
          this.body.initialize(enactId,activationId);
          this.body.preExecution();
          end_if;
        else
          this.body.del();
        end_if;
      @getLoopExpr();
    end_while;
  end;
end;
```

Brief Description : The methods *getInitExpr()*, *getCondExpr()*, and *getLoopExpr()* extracts the initialization, loop-termination and loop-increment clauses respectively of the loop association type. As long as the loop condition is *true*, the same procedure we followed for sequential association is also followed.

A.2.5 Testing

```
param_rule testing_FALSE
bind_classes @defClassName()
bind_if @defAssocType() = "Testing" and @linkName() = "IfFalse"
triggered immediate_after postExecution()
condition @getTestExpr()
otherwise
    if ( askForActivation() = FALSE ) then
            return;
    end_if;
    ext c:@constClassName()
      where ( (c.activityStatus = "INACTIVE") and
              (c.synchSet.member(this.actName) <= 0) and
              (c.synchFlag = 1) and
              (c.aSchName = this.aSchName) and
              (c.enactId = this.enactId) )  do
        c.synchSet.insert(this.actName,0);
```

```
            c.parActivatedId.insert(this.activationId,0);
            c.preExecution();
            return;
        end;
        ext c:@constClassName()
           where ( (c.activityStatus = "INACTIVE") and
                   (c.parActivatedId.member(this.activationId) > 0) and
                   (c.aSchName = this.aSchName) and
                   (c.enactId = this.enactId) )  do
             c.preExecution();
             return;
        end;
        this.IfFalse := @constClassName().pnew();
        if ( this.IfFalse.aSchName = this.aSchName) then
            if ( this.IfFalse.synchFlag = 1) then
               this.IfFalse.synchSet.insert(this.actName,0);
            end_if;
            this.IfFalse.initialize(enactId,activationId);
            this.IfFalse.preExecution();
        else
            this.IfFalse.del();
        end_if;
    end;
```

Brief Description : This association type has two parameterized rules, one for the *true* value of the condition predicate and other for the *false* value. The one that was shown above is executed (or bounded) if the predicate value is *true*. Except for the condition clause, everything else is same as that of the sequential association type.

A.2.6 Synchronization

```
param_rule synch_rule
bind_classes  @definingClass()
triggered before activate()
condition (synchSet.size() != synchLinks)
action
   activatable := 0;
   actName.fdisplay("* This instance of %s is Not Yet Ready for Execution *\n");
   activityStatus := "INACTIVE";
otherwise
   this.activate();
end;
```

Brief Description : This rule is triggered before the *activate* method is called. This rule is activated only if the activity class has a synchronization association in it and number of predecessor activity instances that have completed their execution is less than the total number of constituent activity classes. If both the above conditions are satisfied, then it implies that this instance is not yet ready for execution as all the predecessor activity instances have not completed their executions. Hence, the status is set to inactive and activatbale flag to zero.

A.2.7 Case

```
param_rule case_stmt
bind_classes @defClassName()
bind_if @defAssocType() = "Case"
```

```
triggered immediate_after postExecution()
condition @getBranchExpr()
action
  if ( askForActivation() = FALSE ) then
            return;
  end_if;
  doDefault := 1;
  ext c:@constClassName()
     where ( (c.activityStatus = "INACTIVE") and
             (c.synchSet.member(this.actName) <= 0) and
             (c.synchFlag = 1) and
             (c.aSchName = this.aSchName) and
             (c.enactId = this.enactId) )  do
        c.synchSet.insert(this.actName,0);
        c.parActivatedId.insert(this.activationId,0);
        c.preExecution();
        return;
  end;
  ext c:@constClassName()
     where ( (c.activityStatus = "INACTIVE") and
             (c.parActivatedId.member(this.activationId) > 0) and
             (c.aSchName = this.aSchName) and
             (c.enactId = this.enactId) )  do
         c.preExecution();
         return;
  end;
  this.@linkName() := @constClassName().pnew();
  if ( this.@linkName().aSchName = this.aSchName) then
       if ( this.@linkName().synchFlag = 1) then
          this.@linkName().synchSet.insert(this.actName,0);
       end_if;
       this.@linkName().initialize(enactId,activationId);
       this.@linkName().preExecution();
  else
       this.@linkName().del();
  end_if;
end;
```

Brief Description : If the predicate in the branch expression constraint associated with the association link evaluates to true, then an instance of the constituent class is created and activated in the same lines as that of the sequential association type. The method *getBranchExpr()* extracts the predicate specified in the branch expression constraint from the kernel.

A.3 WFDL for the Confrence Paper Receival Schema

```
define ConPaperSchema : WFSchema
  where:
    sActivity   := "PaperReceival",
    fActivities := "Rejected, ForwardToEditor";
end;
define ReviewPaperSchema : WFSchema
  where:
    sActivity   := "ReviewStructure",
    fActivities := "PrepareReview";
```

```
end;
define P : Project
  PROJECT_INITIALIZATIONS;
  where:
     assocSchemaName := "ConPaperSchema";
end;
define SP : Project
  PROJECT_INITIALIZATIONS;
  where:
     assocSchemaName := "ReviewPaperSchema";
end;
define Request : Entity
  associations:
    public:
    Aggregation
    {
      title    : String;
      author   : String;
      fileName : String;
      status   : String;
      refNo    : Integer;
      reviewers: Set<String>;
    };
 methods:
   public:
   method getInput()
      "\n Give the name of the author => ".display();
      author.read();
      "\n Give the title of the paper => ".display();
      title.read();
      "\n Give the name of the file   => ".display();
      fileName.read();
      refNo := oid();
      status := "Under Processing";
      refNo.fdisplay.("\n Ref No is %d \n");
   end;
end;
define PaperReceival : Activity in ConPaperSchema
  ACTIVITY_INITIALIZATIONS;
  associations:
    public:
      Aggregation
      {
         paperNo : Integer;
      };
      Dataflow
      {{
```

```
        ec : EligibilityCheck where {FLOW_IS_FROM(paperNo : paperNo)};
      }};
      Sequential
      {{
          seq : EligibilityCheck;
      }};
  methods:
     method activate()
      local
        r : Request;
      begin
        r.getInput();
        paperNo := r.reqNo;
      end;
     end;
  where:
    vName := "AdministratorView",
    desc  := "Receives the request and updates the information of the author,
              title, and file name of the paper";
end;
define EligibilityCheck  : Activity in ConPaperSchema
  ACTIVITY_INITIALIZATIONS;
  associations:
    public:
      Aggregation
      {
          eligible : Integer;
          paperNo  : Integer;
      };
      Dataflow
      {{
         e : Eligible where {FLOW_IS_FROM(paperNo : paperNo)};
      }};
      Case
      {{
          b1 : Eligible     where {BRANCHEXPRESSION(eligible = 1)};
          b2 : NotEligible  where {BRANCHEXPRESSION(eligible = 2)};
          b3 : ReturnForCorrections;
      }};
  methods:
     method activate()
      "\n Is the paper eligible for submission => ".display();
      eligible := 0;
      eligible.read();
     end;
  where:
    vName := "AdministratorView",
```

```
    desc  := "The administrator checks whether the paper is eligible for
              submission or not";
end;
define NotEligible  : Activity in ConPaperSchema
  ACTIVITY_INITIALIZATIONS;
  methods:
     method activate()
       "\n The paper didn't meet all the requirements, hence REJECTED \n".display();
     end;
  where:
    vName := "AdministratorView",
    desc  := "If the paper is not eligible, then it is rejected";
end;
define ReturnForCorrections  : Activity in ConPaperSchema
  ACTIVITY_INITIALIZATIONS;
  methods:
     method activate()
       "\n The paper meets the requirements, but some corrections have to be
        made. \n Make the corrections and send it \n".display();
     end;
  where:
    vName := "AdministratorView",
    desc  := "If the paper meets the requirements but if some corrections have
              to be made, then it is returned to the author";
end;
define Eligible  : Activity in ConPaperSchema
  ACTIVITY_INITIALIZATIONS;
  associations:
    public:
      Aggregation
      {
         paperNo : Integer;
      };
      Dataflow
      {{
         fp : ForwardPaper where {FLOW_IS_FROM(paperNo : paperNo)};
      }};
      Sequential
      {{
          seq : ForwardPaper;
      }};
  methods:
     method activate()
       "\n The paper has met all the requirements, its being forwarded \n".display();
       context r:Request
          where r.reqNo = this.paperNo do
          r.status := "ACCEPTED";
```

```
       end;
     end;
  where:
    vName := "AdministratorView",
    desc  := "If the paper is eligible, then its forwarded";
end;
define ForwardPaper  : Activity in ConPaperSchema
  ACTIVITY_INITIALIZATIONS;
  associations:
    public:
      Parallel
      {{
          b1   : Review_1;
          b2   : Review_2;
      }};
      Aggregation
      {
         paperNo : Integer;
      };
      Dataflow
      {{
         r1 : Review_1 where {FLOW_IS_FROM(paperNo : paperNo)};
         r2 : Review_2 where {FLOW_IS_FROM(paperNo : paperNo)};
      }};
  methods:
     method activate()
       local
         rn : String;
       begin
         context r:Request
            where r.reqNo = this.paperNo do
              "\n Give name of reviewer 1 => ".display();
              rn.read();
              r.reviewers(rn,0);
              "\n Give name of reviewer 2 => ".display();
              rn.read();
              r.reviewers(rn,0);
              r.status := "ACCEPTED";
          end;
       end;
     end;
  where:
    vName := "AdministratorView",
    desc  := "Reviewers for the paper are selected and being informed";
end;
define Review_1  : Activity in ConPaperSchema
  ACTIVITY_INITIALIZATIONS;
```

```
  associations:
    public:
      Aggregation
      {
         paperNo  : Integer;
         comments : String;
      };
      Dataflow
      {{
         ec : JointReviews where {FLOW_IS_FROM(paperNo : paperNo;
                                               comments : comments1)};
      }};
      Decomposition
      {{
         decomp : ReviewPaperSchema;
      }};
      Sequential
      {
        seq : JointReviews;
      };
  methods:
     method activate()
     end;
  where:
    vName := "ReviewerView",
    desc  := "A Reviewer reviews the paper and gives his comments";
end;
define Review_2  : Activity in ConPaperSchema
  ACTIVITY_INITIALIZATIONS;
  associations:
    public:
      Aggregation
      {
         paperNo  : Integer;
         comments : String;
      };
      Dataflow
      {{
         ec : JointReviews where {FLOW_IS_FROM(paperNo : paperNo;
                                               comments : comments2)};
      }};
      Decomposition
      {{
         decomp : ReviewPaperSchema;
      }};
      Sequential
      {
```

```
        seq : JointReviews;
      };
  methods:
     method activate()
     end;
  where:
    vName := "ReviewerView",
    desc  := "A Reviewer reviews the paper and gives his comments";
end;
define JointReviews  : Activity in ConPaperSchema
  ACTIVITY_INITIALIZATIONS;
  associations:
    public:
      Aggregation
      {
         paperNo   : Integer;
         accepted  : Boolean;
         comments1 : String;
         comments2 : String;
         fReview   : String;
      };
      Synchronization
      {
       {
          b1 : Review_1;
          b2 : Review_2;
       };
      };
      Testing
      {
       {
          IfFalse : Rejected;
          IfTrue  : ForwardToEditor;
       }
      } where {EXPRESSION_IS(accepted)};
  methods:
     method activate()
       "\n Is the paper accepted \n".display();
       accepted.read();
     end;
  where:
    vName := "ExpertView",
    desc  := "A joint report based on both the reviews is made";
end;
define Rejected  : Activity in ConPaperSchema
  ACTIVITY_INITIALIZATIONS;
  methods:
```

```
  method activate()
    "\n The paper was rejected \n".display();
  end;
  where:
    vName := "ExpertView",
    desc  := "Informs the author that the paper is not accepted";
end;
define ForwardToEditor  : Activity in ConPaperSchema
  ACTIVITY_INITIALIZATIONS;
  methods:
  method activate()
    "\n The paper was forwarded to editor \n".display();
  end;
  where:
    vName := "ExpertView",
    desc  := "Paper is sent to the editor for publishing";
end;
define ReviewStructure  : Activity in ReviewPaperSchema
  ACTIVITY_INITIALIZATIONS;
  associations:
    public:
      Aggregation
      {
         paperNo   : Integer;
         comments  : String;
      };
      Dataflow
      {{
         rc : ReviewContents where {FLOW_IS_FROM(paperNo : paperNo;
                                                 comments : comments)};
      }};
      Sequential
      {
        seq : ReviewContents;
      };
  methods:
     method activate()
       local
         cts : String;
       begin
         "\n Give the comments \n".display();
         cts.reads();
         comments.app(cts);
       end;
     end;
  where:
    vName := "ReviewerView",
```

```
    desc  := "Reviews the structure of the paper and add comments";
end;
define ReviewContents  : Activity in ReviewPaperSchema
  ACTIVITY_INITIALIZATIONS;
  associations:
    public:
      Aggregation
      {
         paperNo   : Integer;
         comments  : String;
      };
      Dataflow
      {{
         pr : PrepareReview where {FLOW_IS_FROM(paperNo : paperNo;
                                                comments : comments)};
      }};
      Sequential
      {
        seq : PrepareReview;
      };
  methods:
     method activate()
       local
         cts : String;
       begin
         "\n Give the comments \n".display();
         cts.reads();
         comments.app(cts);
       end;
     end;
  where:
    vName := "ReviewerView",
    desc  := "Reviews the contents of the paper and add comments";
end;
define PrepareReview  : Activity in ReviewPaperSchema
  ACTIVITY_INITIALIZATIONS;
   associations:
    public:
      Aggregation
      {
         paperNo   : Integer;
         comments  : String;
         review    : String;
      };
  methods:
     method activate()
       "\n Prepare a Review based on comments \n".display();
```

```
        suspend();
      end;
   where:
     vName := "ReviewerView",
     desc  := "Prepares a review report";
end;
```

A.4 A Sample Report

```
*** Generating the Report for the Enactment of an instance of  SusProject ***
  Enactment Id is : 15977
  Enactment Status is : ACTIVE
  Schema Associated with this enactment is : TempSchema
  Name of all the activities associated with this enactment are :
  A
  B
  C
  D
  Total no of Activities are 4
   ## Statistics of Activity Instances  Enactments ##
 * For this instance of A :
 Actual Time is  :    0 /  0 /  0 -  0 :  0 :  0 :     0
 Latency Time is :    0 /  0 /  0 -  0 :  0 :  0 :     0
 Wait    Time is :    0 /  0 /  0 -  0 :  0 :  1 :   870
 Total  Time is  :    0 /  0 /  0 -  0 :  0 :  1 :   870
 * For this instance of B :
 Actual Time is  :    0 /  0 /  0 -  0 :  0 :  7 :   792
 Latency Time is :    0 /  0 /  0 -  0 :  0 :  0 :   345
 Wait    Time is :    0 /  0 /  0 -  0 :  0 :  7 :   226
 Total  Time is  :    0 /  0 /  0 -  0 :  0 : 15 :    18
 * For this instance of C :
 Actual Time is  :    0 /  0 /  0 -  0 :  0 :  0 :    30
 Latency Time is :    0 /  0 /  0 -  0 :  0 :  0 :   193
 Wait    Time is :    0 /  0 /  0 -  0 :  0 : 15 :   555
 Total  Time is  :    0 /  0 /  0 -  0 :  0 : 15 :   585
 * For this instance of D :
 Actual Time is  :    0 /  0 /  0 -  0 :  0 :  0 :     7
 Latency Time is :    0 /  0 /  0 -  0 :  0 :  0 :   202
 Wait    Time is :    0 /  0 /  0 -  0 :  0 : 16 :   166
 Total  Time is  :    0 /  0 /  0 -  0 :  0 : 16 :   173
 Total no of Activity Instances Enacted are 4
               Time Statistics for this Enactment
 Actual Time  =>   0 /  0 /  0 -  0 :  0 :  7 :   829
 Total  Time  =>   0 /  0 /  0 -  0 :  0 :  0 :    46
```

References

[AGR89] Agrawal, R. and Gehani, N.H., "ODE (Object Database & Environment): The Language and the Data Model," Proceedings of the ACM SIGMOD International Conference on Management of Data, Portland, OR, May 1989.

[ALA89] Alashqur, A.M., Su, S.Y.W., and Lam, H., "A Query Language for Manipulating Object-Oriented Databases," Proceedings of the Fifth International Conference on VLDB, Amsterdam, The Netherlands, August 1989.

[BER93] Bernstein, P.A., "Middleware: An Architecture for Distributed System Services," Techinical Report, Digital Corporation, Cambridge Research Laboratory, 1993.

[CAS95] Casati, F., Ceri, S., Pernici, B., and Pozzi., G., "Conceptual modeling of workflows," Proceedings of the O-O ER'95, Gold Coast, Australia, Springer Verlag, Dec. 12-15, 1995.

[CHA89] Chakravarthy, S., "Rule Management and Evaluation: An Active DBMS Perspective," SIGMOD Record, Vol. 18. No. 3, September 1989, pp. 20-28.

[CHE76] Chen, P.P., "The Entity-Relationship Model: Towards a Unified View of Data," ACM Transactions on Database Systems, Vol. 1, No. 1, 1976.

[COR92] Object Management Group and x/OPEN, *The Common Object Request Broker:Architecture and Specification*, John Wiley & Son, Inc., New York, 1992.

[ELL95] Ellis, S., Keddara, K., and Rozenburg, G., "Dynamic Change within Workflow systems," ACM Conference on Organizational Computing Systems (COOSS95), 1995.

[GEH91] Gehani, N.H. and Jagadish, H.V., "ODE as an Active Database: Constraints and Triggers," Proceedings of the 17th Conference on Very Large Data Bases, Barcelona, September 1991, pp. 327-336.

[GEO95] Georgakopoulas, D., Hornick, M., and Sheth, A., "An Overview of Workflow Management: from Process Modeling to Workflow Automation Infrastructure," Distributed and Parallel Databases, Vol. 3, No. 2, April 1995.

[HSU93] Hsu, M., "Special Issue on Workflow and Extended Transaction Systems," Data Engineering Bulletin, Vol. 16, No. 2, June 1993.

[HSU95] Hsu, M., "Special Issue on Workflow Systems," Data Engineering Bulletin, Vol. 18, No. 1, 1995.

[HUL87] Hull, R. and Roger, K., "A Tutorial on Semantic database Modeling," ACM Computing Sciences, Vol. 19, No. 3, September 1987.

[JAV92] Javier, A., "The Design and Implementation of K.1: A Third Generation DataBase Programming Language," M.S.Thesis, Department of Electrical Engineering, University of Florida, 1992.

[KAM95] Kamath, M., Alonso, G., Gunthor, R., and Mohan, C., "Providing High Availability in Workflow Management Systems," Technical Report, IBM Almaden Research Center, 1995.

[MCL93] Malone, T.W., Crowston, K., Lee, J., and Pentland, B., "Tools for Inventing Organizations: Toward a Handbook of Oraganizational Processes," Technical Report, MIT Center for Coordination Science, 1993.

[MED92] Medina-Mora, R., Wong, H., and Flores, P., "The Action Workflow Approach To Workflow Management," Proceedings of the 4th Conference on Computer-Supported Cooperative Work, June 1992.

[OBJ92] Object Management Group, *Object Management Architecture Guide*, John Wiley & Son, Inc., New York, September 1992.

[RAM94] Ramanathan, J., "Support for Workflow Process Collaboration," in *Mechanical Design*: *Theory and Methodology,* Waldron, M.B., Waldron, K.J. (editors), Springer-Verlag, 1994.

[RUS94] Rusinkiewicz, M., and Sheth, A., "Specification and Execution of Transactional Workflows," Kim, W. (editor), Modern Database Systems: The Object Model, Interoperability, and Beyond, ACM Press, 1994.

[SMI77] Smith, J. M. and Smith, D. C. P., "Database Abstractions: Aggregation and Generalization," ACM Transactions on Database Systems, Vol. 2, No. 2, June 1977, pp. 105-133.

[STO92] Stonebraker, M., "The Integration of Rule Systems and Database Systems," IEEE TKDE, Vol. 4, No. 5, October 1992.

[SU89] Su, S.Y.W., Krishnamurthy, V. and Lam, H., "An Object-oriented Semantic Association Model (OSAM*)," in *AI in Industrial Engineering and Manufacturing*: *Theoretical Issues and Applications,* Kumara, S. and Kashyap, R.L. (eds.), American Institute of Industrial Engineering, Norcross, GA, 1989.

[SU93] Su, S.Y.W. and Fang, S.C., "A neutral Semantic Representation for Data Model and Schema Translation," Technical Report, Department of Computer and Information Sciences, University of Florida, July 1993.

[TAN95] Tang, J., and Veijalainen, J., "Enforcing inter-task dependencies in transaction workflows," Proc. of the 3rd Intl. Conf. on Cooperative Information Systems, May 1995.

[VOS91] Vossen, G., "Bibliography on Object-Oriented Database Management," SIGMOD Record, Vol.20, No.1, March 1991, pp. 24-46.

[WOR94] Workflow Management Coalition, "Workflow Managemnt Reference Model", 1994.

[YAS91a] Yaseen, R.M., "An Extensible Data Model and Extensible System Architecture for Building Advanced KBMS," Ph.D. Dissertation, Department of Computer and Information Sciences, University of Florida, 1991.

[YAS91b] Yaseen, R.M.,Su, S.Y.W. and Lam, H., "An Extensible Kernel Object Management System," Proceedings of the ACM Conference on OOPSLA, Phoenix, AZ, October 1991.