REALIZING
THE SPECIFICATION AND EXECUTION OF WORKFLOWS
THROUGH THE EVENT CALCULUS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

HÜSEYİN YILMAZ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
THE DEPARTMENT OF COMPUTER ENGINEERING

DECEMBER 2006

Approval of the Graduate School of Natural and Applied Sciences

Prof. Dr. Canan ÖZGEN
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. Ayşe KİPER
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Nihan KESİM ÇİÇEKLİ
Supervisor

Examining Committee Members:

Assoc. Prof. Dr. Ali Hikmet DOĞRU (METU, CENG)

Assoc. Prof. Dr. Nihan KESİM ÇİÇEKLİ (METU, CENG)

Assoc. Prof. Dr. Ferda Nur ALPASLAN (METU, CENG)

Assist. Prof. Dr. Pınar ŞENKUL (METU, CENG)

Gökçe Banu LALECİ (METU, SRDC)

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: Hüseyin YILMAZ

Signature :

ABSTRACT

REALIZING THE SPECIFICATION AND EXECUTION OF WORKFLOWS THROUGH THE EVENT CALCULUS

YILMAZ, Hüseyin
M.S., Department of Computer Engineering
Supervisor: Assoc. Prof. Dr. Nihan KESİM ÇİÇEKLİ

December 2006, 92 pages

Workflow management promises a solution to an age-old problem: controlling, monitoring, optimizing and supporting business processes. What is new about workflow management is the explicit representation of the business process logic which allows for computerized support. In the light of this support, many researchers developed different approaches to model new systems with different capabilities to solve this age-old problem. One of the approaches is using logic-based methodology for the specification and execution of workflows. Here, the event calculus, a logic programming formalism for representing events and their effects especially in database applications, is used for this approach. It is shown that the control flow graph of a workflow specification can be expressed as a set of logical formulas and the event calculus can be used to specify the role of a workflow manager through a set of rules for the execution dependencies of activities. Constructed workflow formalization through Event Calculus is realized by using recent technologies, and the resulting product is named as EventFlow,

including some administrative interfaces to manage system and workflow engine. The thesis describes the architecture and implementation details of EventFlow, an editor developed for graphical representation of control flow graph, and technologies used in the implementation. And an example application is built to show the usability and execution of the implemented system.

Keywords : Workflow, Workflow Management System, Workflow formalization, The Event Calculus.

ÖZ

OLAY CEBİRİ ÜZERİNDEN İŞ AKIŞI TANIM VE UYGULAMASININ GERÇEKLENMESİ

YILMAZ, Hüseyin
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Nihan KESİM ÇİÇEKLİ

Aralık 2006, 92 sayfa

İş akışı yönetim sistemi asırlık bir problem olan iş süreçlerinin desteklenmesi, kontrolü, izlenmesi ve en uygun şekilde düzenlenmesi sorununa çözüm temin eder. Bu sistem ile ilgili yeni olan şey ise sistemin iş süreci mantığının açık tanımlamasına bilgisayar desteğinin sağlanmış olmasıdır. Bu destek ışığında, birçok araştırmacı bu asırlık problemin çözümünü sağlayacak farklı yeteneklerde sistemler modellemek için farklı yaklaşımlar geliştirmişlerdir. Bunlardan birisi de iş akışlarının yorumlanması, gerçekleştirilmesi ve biçimlendirilmesi için mantık tabanlı bir metodoloji kullanılmasıdır. Burada, daha çok veritabanı uygulamalarında olayları ve etkilerini belirtmek için kullanılan bir mantıksal programlama biçimi olan Olay Cebiri, bu yaklaşım için kullanılmıştır. Kontrol akış diyagramına ait işakışı tanımlamasının bir mantıksal formüller bütünü olarak ifade edilebileceği ve Olay Cebiri'nin, aktivitelerin uygulama bağımlılıkları için oluşturulacak kurallar bütünü aracılığı ile işakışı yöneticisi görevinin

tanımlanmasında kullanılabileceği gösterilmiştir. Oluşturulan Olay Cebiri tabanlı iş akışı tanımlaması, güncel teknolojiler kullanılarak gerçeklenmiş ve sonuçta oluşan, iş akışı motoru ve bazı sistem yönetim arayüzlerine sahip ürüne EventFlow ismi verilmiştir. İş akışı motoru ve grafiksel kontrol akışı düzenleyicisine ait mimari ve geliştirme detayları ile kullanılan teknolojilere ait detaylar verilmiştir. Ayrıca, oluşturulan sistemin kullanılırlığı ve çalışmasının gösterilmesi için örnek bir uygulama oluşturulmuştur.

Anahtar Kelimeler : İş Akışı, İş Akışı Yönetim Sistemi, İş Akışı Biçimlendirme, Olay Cebiri.

ACKNOWLEDGMENTS

I am grateful to my thesis supervisor Assoc. Prof. Dr. Nihan KESİM ÇİÇEKLİ for her guidance, motivation and support throughout this study.

I also want to thank my parents, my sisters and my wife for their motivating support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

APPL	:	Application
API	:	Application Programming Interface
Ax	:	Axiom
AxH	:	Axioms for Happens
AxIT	:	Axiom for Initiates/Terminates
AxS	:	Axiom for Scheduling
DBMS	:	Database Management System
EC	:	Event Calculus
ECA	:	Event-Condition-Action
EPC	:	Event-driven Process Chain
IT	:	Information Technology
J2EE	:	Java2 Platform, Enterprise Edition
WFMS	:	Workflow Management System
WfMC	:	Workflow Management Coalition
OS	:	Operating System
UIMS	:	User Interface Management System
YAWL	:	Yet Another Workflow Language
XPDL	:	XML Process Description Language
OMG	:	Object Management Group

CHAPTER 1

INTRODUCTION

In former times, information systems were designed to support the execution of individual tasks. Today's information systems need to support the business processes at hand. It no longer suffices to focus on just the tasks. The information system also needs to control, monitor and support the logistical aspects of a business process. In other words, the information system also has to manage the flow of work through the organization. Many organizations with complex business processes have identified the need for concepts, techniques, and tools to support the management of workflows. Based on this need the term *workflow management* was born.

Until recently there were no generic tools to support workflow management. As a result, parts of the business process were hard-coded in the applications. For example, an application to support task ***X*** triggers another application to support task ***Y***. This means that one application knows about the existence of another application. This is undesirable, because every time the underlying business process is changed, applications need to be modified. Moreover, similar constructs need to be implemented in several applications and it is not possible to monitor and control the entire workflow. Therefore, several software vendors recognized the need for *workflow management systems*. A **W**ork**f**low **M**anagement **S**ystem (*WFMS*) is a generic software tool which allows for the definition, execution, registration and control of workflows. At the moment many vendors are offering a workflow management system. This shows that the software industry recognizes the potential of workflow management tools.

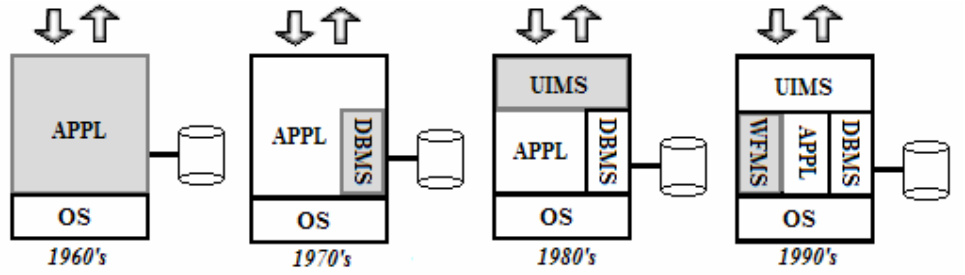


Figure 1.1 Workflow management systems in a historical perspective.

In order to become aware of the impact of workflow management in the near future, it is useful to consider the evolution of information systems over the last four decades [33,36]. **Figure 1.1** shows the phenomenon of workflow management in a historical perspective. The figure illustrates the evolution of information systems in the last four decades by describing the architecture of a typical information system in terms of its components. In the sixties an information system was composed of a number of stand-alone applications. For each of these applications an application-specific user interface and database system had to be developed, i.e., each application had its own routines for user interaction and data storage and retrieval. In the seventies data was pushed out of the applications. For this purpose **D**atabase **M**anagement **S**ystems (DBMS) were developed. By using a DBMS, applications were freed from the burden of data management. In the eighties a similar thing happened for user interfaces. The emergence of **U**ser **I**nterface **M**anagement **S**ystems (UIMS) enabled application developers to push the user interaction out of the applications. WFMS is the next step in pushing generic functionality out of the applications. The nineties are marked by the emergence of workflow software, allowing application developers to push the business procedures out of the applications.

Figure 1.1 clearly shows that, the WFMS is a generic building block to support business processes. Many information systems could benefit from such a building block, because many organizations are starting to see the need for advanced tools to support the design and execution of business processes. There are several

reasons for the increased interest in business processes [33,36]. First of all, management philosophies such as Business Process Reengineering and Continuous Process Improvement stimulated organizations to become more aware of the business processes. Secondly, today's organizations need to deliver a broad range of products and services. As a result the number of processes inside organizations has increased. Not only the number of products and services has increased, but also the lifetime of products and services has decreased in the last three decades. As a result, today's business processes are also subject to frequent changes. Moreover, the complexity of these processes increased considerably. All these changes in the environment of the information system in an average organization, have made business processes an important issue in the development of information systems. Therefore, there is a clear need for a building block named '*workflow management system'* [33].

The main purpose of a workflow management system is the support of the definition, execution, registration and control of *processes* [33]. Because processes are a dominant factor in workflow management, it is important to use an established framework for modeling and analyzing workflow processes. A workflow is a collection of cooperating, coordinated activities designed to accomplish a completely or partially automated process [19]. An activity in a workflow is performed by an agent that can be one or more software systems, one or a team of humans, or a combination of these. Human activities include interacting with computers closely (e.g., providing input commands) or loosely (e.g., using computers only to indicate activity progress). Examples of activities include updating a file or database, generating or mailing a bill, and laying a cable. In addition to a collection of activities, a workflow defines the order of activity invocation or condition(s) under which activities must be invoked, activity synchronization, and information/data flow. A workflow management system provides these and also provides support for modeling, executing and monitoring the activities in a workflow. Nowadays, there are many commercial products [8,18,23,24] to model and execute workflows and there have been many formal models proposed for the analysis and reasoning about the workflows.

[6,20] The most common frameworks for specifying workflows are graph-based, event-condition-action rules, and logic-based methods.

Graph-based approaches provide a good way to visualize the overall flow of control, where nodes are associated with activities and edges with control or data flow between activities. Petri nets and state charts are graph-based general-purpose process specification formalisms that have been applied to workflow specifications [33,23].

ECA rules have been widely used in active databases and they have been adopted in the specification of workflows as well. However, their expressive power is not as general as control flow graphs [5].

Logic-based formalisms, on the other hand, use the power of declarative semantics of logic to specify the properties of workflows and the operational semantics of logical systems to model the execution of workflows. Logic-based approaches mostly deal with the verification of workflows with global constraints [4,26].

The logic-based methods have the benefit of well-defined declarative semantics and well-studied computational models. In this thesis, a logic programming approach for the specification of control flow graphs, execution dependencies between activities and scheduling of activities within a workflow has been realized. The implementation includes the specification of main types of flow controls, such as sequential, concurrent and alternative execution of activities. Other issues such as representing the transactional properties of workflows, or temporal constraints (global constraints) between workflow activities are out of the scope of this thesis.

The main contribution of the thesis is to develop a workflow management system based on Event Calculus to realize the given formalization in [18]. As a proof of concept, a simple workflow management system is developed, to show the usability of the given formalization for a real-life application. Also, by providing some administrative capabilities, it will be easy to manage the system and define

different workflows to use for different applications or purposes. This system can also be used as a quick tool for simulation of real-life applications, and testing of different experimental workflows without implementing any logic for the activities used in workflow definition. Also it is useful to analyze the behaviour of workflows for different control flows with different number of agents and workflow instances. Also it may serve the need for querying the history of the workflow to analyze and assess the efficiency, accuracy and the timeliness of the activities by deriving the state of the workflow at any time in the past.

The rest of the thesis is organized as follows. ***Chapter 2*** contains a brief review of basic concepts of workflows and related works done on workflow management systems. In ***Chapter 3***, details of Event Calculus based formalization of specification and execution of workflows are given. ***Chapter 4*** gives the detailed information about the implementation and the architecture of the workflow management system (namely EventFlow) and the components used while developing that system. In ***Chapter 5***, a sample application is given. And, ***Chapter 6*** concludes the work by summarizing the features and possible future extensions of the implemented system.

CHAPTER 2

RELATED WORK

In this chapter, basic concepts which are related to workflow management systems and the **E**vent **C**alculus (EC) are first described briefly. Then some of the approaches for workflow modelling are summarized. Finally, some open source and commercial workflow system implementations are given as sample workflow systems.

2.1. Basic concepts

In this section, the definitions of basic concepts of the workflow systems and the EC will be described briefly to introduce the main parts of the framework presented in this thesis.

2.1.1. Workflow concepts

A *workflow* is a computerized facilitation or automation of a business process involving the coordinated execution of multiple activities performed by different processing entities. Processing of purchase orders over the Internet and insurance claims can be given as examples of workflows. An a*ctivity* (*task*) defines a logical step or description of a piece of work that contributes toward the achievement of a process like updating a database, generating a bill, mailing a form, etc. An *agent* is a processing entity that performs the defined activities in the workflow. It can be a hardware device, a person or a software system such as an application program, etc. A task defined to be done by human includes interacting with computers such as providing input commands. A *workflow instance* represents an instance of workflow definition which includes the automated aspects of a process

instance only. Running several concurrent instances of a workflow is possible. For example, a workflow manager can execute several processing orders at the same time.

A workflow management system is a software system that controls the *execution* of the multiple activities by different agents. *Specification (design)* of a workflow includes describing those aspects of its constituent activities and the agents that execute them. And also the relationships among activities and their execution requirements are defined.

A reference model that describes the major components and interfaces within a workflow architecture is defined by the **W**ork**f**low **M**anagement **C**oalition (WfMC) [14]. In a workflow, activities are related to one another via flow control conditions. Designing workflow with many different transition patterns is possible [34]. Accordingly the following basic routings among the activities are identified in this framework:

1. *Sequential*: Execution of activities sequentially (i.e. an activity is followed by the next activity).

2. *Parallel*: Execution of two or more activities parallelly. Two building blocks are identified for this kind of execution:

    (a) *AND-split* enables the concurrent execution of two or more activities after another activity has been completed.

    (b) *AND-join* synchronizes the parallel flows, and the next activity can start only after all activities in the join have been completed.

3. *Conditional:* Execution of one of the alternative activities. In order to model a choice among two or more alternatives following blocks can be used:

    (a) *XOR-split* enables the execution of only one of several branches of flows based on a condition check

(b) *XOR-join* re-converges the execution into a single thread of control without any synchronization.

4. *Iteration:* An activity cycle involving the repetitive execution of activities until a condition is met. Representation of that can be done by using XOR-split and XOR-join blocks.

Control flow graphs, most appropriate way of showing the execution dependencies of the activities in a workflow, provide a good way to visualize the overall flow of control. The vertices identify the names of corresponding activities in a control flow graph. And the edges represent the successor relation between th activities. Typically the initial and final activities of a workflow, the subsequent acivities for each activity, and whether all of these subsequent activities must be executed concurrently, or it is sufficient to execute only one branch depending on a condition is specified by a control flow graph.

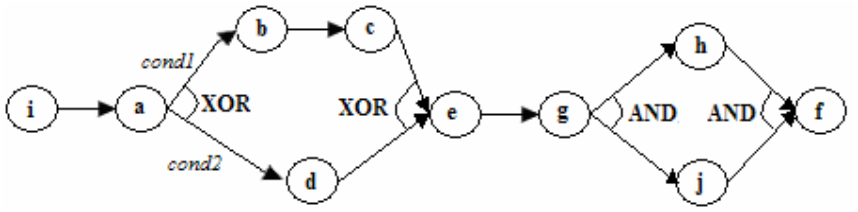


Figure 2.1 An example control flow graph

Figure 2.1 illustrates a control flow graph where the activity ***i*** is the initial task, and ***f*** is the final task. After the activity ***i*** is completed, next activity ***a*** will start. After completion of the activiy ***a***, one of the subsequent activities ***b***, and ***d*** will be started with respect to the evaluation result of the conditions *cond1* and *cond2*. This is indicated by the label "XOR". From the definition of "XOR-split", only one of the subsequent branches corresponding to the condition which is evaluated to true will be started. The conditions are based on workflow control data and applied to the current state of the workflow. The conditions can depend on some logical status, or output generated by some prior activity in the workflow, or on the value of some external variable (e.g. time). If the condition *cond1* is true, then

the activity ***b***, and just after the completion of it the activity ***c*** will be started, otherwise the activity ***d*** will be started. Activity ***e*** will be enabled immediately after either one of the activities ***c*** or ***d*** is completed. After completion of the activity ***e***, the activity ***g*** will be started and just after the completion of that activity, both of the subsequent activities ***h*** and ***j*** will be started concurrently. This is indicated by the label "AND". And the final activity ***f*** can only start after the completion of the parallel activities ***h*** and ***j***. And after the activity ***f*** is completed, the running instance workflow will be completed.

2.1.2. The Event Calculus

The EC is a logic programming formalism for representing events and their effects, especially in database applications [21]. One of the EC dialects is based on a later simplified version presented in [22]. There are two assumptions made in this simplified version of the EC:

- *The events have no extended duration*
- *The properties initiated by events, hold in the period that event initiates and contain the said event*.

The formulation and implementation of the EC are simplified by these assumptions; otherwise nothing essential depends on them.

The EC is based on general axioms concerning notions of events, properties and the periods of time for which the properties hold. The events initiate and/or terminate periods of time in which a property holds. As events occur in the domain of the application, the general axioms imply new properties that hold true in the new state of the world being modeled, and infer the termination of properties that no longer hold true from the previous state. The main axiom used by the event calculus to infer that a property holds true at a time is described as follow:

holds_at(Property, Time) ←
happens(Event, Time1),

Time1 ≤ *Time,*
initiates(Event, Property),
not broken(Property, Time1, Time). (Ax1)

In ***(Ax1)***, the predicate *holds_at(Property, Time)* represents that the property ***Property*** holds at time ***Time***, and the other predicates in that axiom represent the following:

- *happens(Event, Time1)* : The event ***Event*** occurs at time point ***Time1***;
- *initiates(Event, Property)* : The event ***Event*** initiates a period of time during which the property ***Property*** holds;
- *broken(Property, Time1, Time)* : The property ***Property*** ceases to hold between time point ***Time1*** and time point ***Time*** (inclusive) due to an event which terminates it.

The time points are ordered by the usual comparative operators. The ***not*** operator is interpreted as negation-as-failure. The use of negation-as-failure gives a form of default persistence into the future. Thus, the persistence axiom states that once a property ***Property*** is initiated by an event ***Event*** at time point ***Time1***, it holds for an open period of time containing time point ***Time1*** (i.e. *[Time1,Time)*), unless there is another event happened at some point of time after time point ***Time1***, that breaks the persistence of property ***Property***.

Other axioms used in the body of this axiom are defined as follows. The axiom for *happens(Event, Time)* is usually defined as an extensional predicate symbol that records the happening of the event ***Event*** at time point ***Time***. A particular course of events that occur in the real world being modeled is represented with a set of such extensional predicates. The axiom for *broken(Property, Time1, Time2)* is defined by the following clause:

broken(Property, Time1, Time2) ←
happens(Event, Time), terminates(Event, Property),
Time1 ≤ *Time* ≤ *Time2.* (Ax2)

That is, the persistence of the property ***Property*** is broken at time point ***Time2*** if a distinct event ***Event*** that happened at time ***Time*** between ***Time1*** and ***Time2*** terminates the persistence of property ***Property***. Here the predicate *terminates(Event, Property)* represents that the event ***Event*** terminates any ongoing period during which property ***Property*** holds. Finally the axioms for *initiates* and *terminates* are specific to the application at hand. The problem domain is captured by a set of *initiates* and *terminates* clauses.

The EC is defined as a collection of all types of axioms described above. The state of the system at any point of time until the time point ***t*** can be computed by using the *holds_at* predicate, if the event occurrences until time ***t*** are known. The event occurrences are recorded as an extensional database and snapshots of the database state can be derived at any time using this history of events. Also, it is possible to extend the EC by adding the definition of other predicates such as *holds_for* to find out the period of time for which a property holds.

2.2. Modeling workflows

There are many approaches studied by researchers for the modeling of long running sequence of activities, in other words, workflows. Some of them are briefly described in the following sections.

2.2.1. Using the Temporal Logic

In [3], workflows are modeled as a set of inter-task dependencies. Both local and global constraints are modeled in this way and, therefore, the control-flow graph is not represented explicitly. The tasks in a workflow are described in terms of significant events. A typical event is the beginning or termination of a task, but it can also be some other thing like printing a report, etc.

When an event is received for execution, it is checked against every dependency and based on that the event might be accepted, rejected, or delayed and scheduled later. The dependencies are specified as formulae in Computational Tree Logic. The scheduler enforces these dependencies by converting them into automata and ensuring that the sequence of scheduled events is accepted by all these automata.

This work does not explicitly deal with the verification issues, such as whether the given set of constraints implies some other constraints.

2.2.2. Using the Event Calculus

Actually, the EC is a simple temporal formalism designed to model situations characterized by a set of events, whose occurrences have the effect of initiating or terminating the validity of determined properties. Given a description of when these events take place and of the properties they affect, it is able to determine the maximal validity intervals over which a property holds uninterruptedly. It uses a polynomial algorithm for the verification or calculation of the maximal validity intervals and its axioms can easily be implemented as a logic program.

The EC provides mechanisms for storing and querying the history of all known events. Once the event occurrences until time ***t*** are known, the state of the system can be computed at any point of time until ***t***. In order to be able to model the invocation of activities in a workflow, we need to be able to represent that certain type of event invariably follows a certain other type of event, or that a certain type of event occurs when some property holds [19]. In this framework events are treated as triggers that denote the start or end times of activities. Once we know the history of all events either explicitly recorded or automatically generated by the system, the modeling of workflow execution becomes the computation of new events from the history and thus executing new activities until the end of the workflow is reached. The most important result made possible by this approach is the definition of the operational semantics of event detection, condition verification and activity scheduling in terms of a well-defined semantics, which can be computed by that of a deductive system and queries.

[19] presents a simple scheduling algorithm in which it is possible to model agents as separate entities and assign agents to certain activities based on their cost. The workflow manager is designed to choose the best agent to perform the next scheduled activity among all available agents qualified to do that activity. The representation of events, activities and agents in presented framework makes

it also possible to model the execution of concurrent workflow instances over a single workflow specification.

2.2.3. Using Event-Condition-Action Rules

Event-**C**ondition-**A**ction (ECA) rules are a way of modeling dependencies between workflow activities. As the name specifies, each rule consists of three components: events, conditions and actions. *Event* corresponds to the notion of significant event like beginning or ending of a transaction. A *condition* is a query over the database state. The condition is satisfied if the query evaluates to true. An *action* is a program which can be either a database operation or an external operation such as an application program. If the condition is satisfied for a rule then the action associated with the rule is fired. ECA rules are expressive enough to model global constraints between tasks.

In [15], Vortex, a programming paradigm for modeling workflows, is defined. In this model, a workflow is specified in terms of *modules* and *attributes* values which are to be computed. Vortex paradigm is especially suitable for modeling dynamic, data-driven workflows. Modules correspond to workflow activities. The task of the modules is to compute the values for the specified attributes. Global constraints can be expressed in a Vortex workflow specification. To verify Vortex workflows, model checking techniques can be applied on a symbolic representation of a workflow specified using event-condition-action rules.

In [7], triggers are used to model workflows. Workflows are represented as activities with dependencies between them. The workflow activities are represented as transactions and the dependencies between the transactions are represented by *triggers* which are simple ECA rules. The scheduler executes the transactions in a nested transaction model, and defines mechanisms to serialize concurrently executing rules.

2.2.4. Using Petri Nets

Petri Nets are an established way of modeling and verifying process behavior. A Petri Net is a directed bipartite graph consisting of two types of nodes, called

places and *transitions*. Edges go either from places to transitions or from transitions to places. At any time a place contains zero or more *tokens*. The state of the Petri Net, referred to as *marking*, denotes the distribution of tokens over places. Since workflows are models of complex processes, it seems natural to try to formalize workflows in a Petri Net setting. The graphical nature of Petri Nets also makes them appealing as a modeling tool.

Petri Nets are used to model workflow tasks and dependencies between these tasks in [1]. Logical operators have been used to specify relationships between multiple dependencies. The classical Petri Net has been extended with time to model temporal dependencies. It is possible to verify safety and liveness properties of the workflow specification on the resulting Petri Net model. It is also possible to check the consistency of the dependencies specified. However, even though it is possible to check whether a workflow specification can be scheduled, there is no scheduler to actually schedule the tasks.

In [33], workflows have been modeled as tasks and transitions between these tasks. Join and split constructs are used to model constraints between these tasks. However, it is possible to specify only local constraints using these constructs. Triggers have been used to model constraints arising out of external conditions. In order to model constraints based on attribute values and time, a higher level Petri Net extended with the semantics of token color and time are used. The Petri Net model of a task is simpler than in [1]. However, the use of higher level Petri Nets provides more abstraction of the workflow specification than in [1]. It is possible to check for deadlock, live-lock and proper termination on the Petri Net model of the workflow. Special structural characterizations of Petri Nets have been provided where these properties can be verified in polynomial time. As in [1], there is no scheduler to actually schedule the different tasks according to the constraints.

2.3. Sample Workflow Systems

There are many commercial [8,11,18,23,24,27] and open source implementations [26] of workflow management system software. Before starting the implementation of EventFlow system, some of them are briefly examined from the point of their used technologies and capabilities given for modeling and executing workflows.

2.3.1. Commercial Workflow Systems

COSA [29] is a Petri-net-based workflow management system developed by Ley GmbH, a German company based in Pullheim. The modeling language of COSA consists of two types of building blocks: activities (i.e., Petri net transitions) and conditions (i.e. Petri net places). *COSA* extends the classical Petri net model with control data to allow for explicit choices based on information and decisions. Unfortunately, only safe Petri nets are allowed, i.e., it is not allowed to have multiple tokens in one place. Therefore, *COSA* is unable to support multiple instances directly. The only way to deal with multiple instances is to use workflow triggers. Every subprocess in *COSA* has a unique start activity and a unique end activity. As a result, only highly structured subprocesses are possible and termination is always explicit. The main feature of the workflow language of *COSA* is that it allows for the explicit representation of states.

Lotus Domino Workflow [23] is the workflow extension of the groupware product Lotus Domino/Notes (Lotus/IBM). Clearly, the tight integration with the groupware product is one of the attractive features of this product. The marriage between groupware (Lotus Domino/Notes) and workflow (Domino Workow) allows for partly structured workflows. There are various types of resource classes, e.g., person (singleton), workgroup (including inheritance and many-to-many relationships), department (only one-to-many relationships, however with inheritance), and roles. Each routing relation is of one of the following types:

1. Always (for AND-split)

2. Exclusive choice (for XOR-split made by the user at the end of the activity)

3. Multiple choice (for OR-split made by the user after completing the activity)

4. Condition (automatically evaluated on the basis of data elements)

5. Else (only taken if none of the other routing relations is activated).

Each activity can serve as a join. The type of join is determined implicitly. Joins are either enabled or disabled. If a join is disabled, it serves as an XOR-join, i.e., the activity is enabled the moment one of the preceding activities completes. If the join is enabled, it continuously checks whether potentially it can receive more inputs in the future without activating itself. This way it is possible to make AND-joins or use more advanced synchronization mechanisms.

MQSeries/Workflow [8] is the successor of IBM's workflow offering, FlowMark. FlowMark was one of the first workflow products that was independent from document management and imaging services. It has been renamed to *MQSeries/Workflow* after a move from the proprietary middleware to middleware based on the *MQSeries* product. The workflow model consists of activities linked by transitions. Other than a decomposition block, few other special modeling constructs are available. The workflow engine of *MQSeries/Workflow* has unique execution semantics in that it propagates a False Token for every transition with a condition evaluating to False. This allows for every activity that has more than one incoming transition to act as a synchronizing merge (AND-join). Other than the synchronizing merge, which is a natural construct for *MQSeries/Workflow*, there is no way to directly implement any of the other advanced synchronization patterns [34]. Support for multiple instances is provided through the Bundle construct although it is not suitable if the number of instances is not known at any point prior to generating the instances involved. Arbitrary loops are not supported. An explicit termination point is not required and the workflow process will terminate when there is nothing else to be executed. There is no direct way to

model the cancellation patterns [34]. There is a global data container for the running workflow instance, and also for each activity, one can define an input and an output data.

Visual WorkFlo [11,12] is part of the FileNet's Panagon suite (Panagon WorkFlo Services) that includes also document management and imaging servers. *Visual WorkFlo* is one of the oldest products on the market. The workflow modeling language of Visual WorkFlo is structured and is a collection of activities and routing elements such as Branch (XOR-split), While (structured loop), Static Split (AND-split), Rendezvous (AND-join), and Release. *Visual WorkFlo* does not directly support any of the advanced synchronization patterns. It requires the model to have structured loops only and one, explicit, termination node thus limiting the suitability of the resulting specifications. Direct support for Multiple Instances is possible through the Release construct as long as there is no further synchronization required. There is no direct way to implement any of the state-based patterns. There is no explicit support for the cancellation patterns.

SAP R/3 Workflow [27] is an integrated workflow component within SAP's R/3 software suite. *SAP R/3 Workflow* imposes a number of restrictions on the use of **E**vent-driven **P**rocess **C**hains (EPC). EPCs that are used for workflow modeling consist of a set of functions (activities), events and connectors (AND, XOR, OR). However, in *SAP R/3 Workflow* not the full expressive power of EPCs can be used, as there are a number of syntactic restrictions similar in vein to the restrictions imposed by *Filenet Visual Worko* (e.g. every workflow needs to have a unique starting and a unique ending point, and-splits are always followed by and-joins, or-splits by or-joins). As such, there is no direct provision for the advanced synchronization constructs, multiple instances, arbitrary loops, state-based or cancellation patterns [34].

NovaManage [24] is an integrated document management and workflow solution designed to meet the needs of highly regulated and quality controlled industries, such as the pharmaceutical and medical device sectors. Activity can be assigned to one or more agent (person), and the group of agents. Parallel-split (AND-split)

and Decide (XOR-split) nodes are supported directly. If there is a decide node after an activity currently executed by the agent, the agent will be asked to select the next activity or activities to enable for execution in currently running workflow instance. For each split node, there must be a corresponding join node. Each workflow template must have a unique starting and a unique ending activity. There is no support for state-based and cancellation patterns [34]. There is no data container defined for the workflow.

2.3.2. Open Source Workflow Systems

***Y**et **A**nother **W**orkflow **L**anguage* (YAWL), [38] an open source workflow language/management system, is based on a rigorous analysis of existing workflow management systems and workflow languages. YAWL extends Petri-Nets as its modeling approach. Unlike traditional systems it provides direct support for most of the workflow patterns. YAWL supports the control-flow perspective, the data perspective, and is able to interact with web services declared in **W**eb **S**ervice **D**efinition **L**anguage (WSDL). It is based on a distributed, web-friendly infrastructure.

The Enhydra Shark project [8] delivers a workflow server with a difference. It is an extendable and embeddable Java Open Source workflow engine framework including a standard implementation completely based on **W**ork**f**low **M**anagement **C**oalition (WfMC) specifications using **X**ML **P**rocessing **D**escription **L**anguage (XPDL) as its native workflow process definition format and the WfMC "ToolAgents" **A**pplication **P**rogramming **I**nterface (API) for serverside execution of system activities.

Every single component (persistence layer, assignment manager, etc.) can be used with its standard implementation or extended/replaced by project specific modules. This way Enhydra Shark can be used as a simple "Java library" in servlet or swing applications or running in a J2EE container supporting a session beans API, Corba ORB or accessed as a web service.

WfMOpen [37] is a J2EE based implementation of a workflow facility (workflow engine) as proposed by the WfMC and the **O**bject **M**anagement **G**roup (OMG). Workflows are specified using WfMC's XPDL with some extensions.

CHAPTER 3

WORKFLOW FORMALIZATION USING THE EVENT CALCULUS

Before giving the architectural and the implementational details of the developed workflow system (namely EventFlow), the association between the constructs of the **E**vent **C**alculus (EC), and also the details of using the EC in the specification and the execution of workflows are described in this chapter.

3.1. Formalizing the control flow graphs

A set of predicates in first-order-logic can be used to represent a given control flow graph. In this thesis, it is considered that there are five different successor relations between activities. These relations with separate predicate symbols are described in **Table 3.1**.

Table 3.1 Successor relationships between activities

Predicate	*Description*
initial_activity(***Activity***)	***Activity*** is the first activity in the workflow
sequential(***Activity1, Activity2***)	***Activity2*** follows ***Activity1*** unconditionally
and_split(***Activity, ListOfActivities***)	***Activity*** is followed by a list of activities in ***ListOfActivities***
xor_split(***Activity, ActCondPairs***)	***Activity*** is followed by ***Activityx*** in list ***ActCondPairs*** if condition ***conditionx*** is true
and_join(***ListOfActivities, Activity***)	***Activity*** starts after all the activities in ***ListOfActivities*** completed
xor_join(***ListOfActivities, Activity***)	***Activity*** starts after one of the activity in ***ListOfActivities*** completed
final_activity(***Activity***)	***Activity*** is the last activity in the workflow

The developed framework must be able to express the execution of concurrent workflow instances over the same specification. For example, if the workflow describes the activities in an order processing application, there may be more than one order being processed at the same time. In order to be able to model such concurrent instances of a given workflow and the execution of the same activities for different workflow instances, a special naming convention must be used. Thus, each workflow instance is given a unique identity.

This unique identity is an atomic term and it is generated by the system when the workflow instance is started. Since each activity is executed at different times for different workflow instances, their names must be associated with that unique identity of workflow instance to identify each of these executions. In its simplest form, this identity will be the workflow instance id. Thus, in first order predicate form of the workflow, each activity execution is represented by a term *"act(AcivitytName, EID)"* where ***ActivityName*** is the name of the activity given by the user at the specification, and ***EID*** is the execution id of the activity generated by the system for the workflow instance being run.

For example, an execution of activity ***e*** in **Figure 2.1**, in a workflow instance ***w1*** can be represented by the term *act(e,w1)*, and when it is completed it can trigger the execution of the activity ***g*** with the same workflow id, i.e. *act(g,w1)*.

The sample workflow shown in **Figure 2.1** is actually translated into the following first order predicates in the implemented framework, using the naming conventions described above:

initial_activity(act(i, EID)).
sequential(act(i, EID), act(a, EID)).
xor_split(act(a, EID), [(act(b, EID), cond1), (act(d, EID), cond2)]).
sequential(act(b, EID), act(c, EID)).
xor_join([act(c, EID), act(d, EID)], act(e, EID)).
sequential(act(e, EID), act(g, EID)).
and_split(act(g, EID), [act(h, EID), act(j, EID)]).
and_join([act(h, EID), act(j, EID)], act(f, EID)).

final_activity(act(f, EID)).

The graphical structure of the control flow graph can be directly mapped into a set of logic formulas by the above set of predicates. In this thesis, this mapping is automatically done by the EventFlow Editor while saving the built workflow template to the disk or the system database. (*see Section 4.2.1*) The workflow manager determines the actual execution order of activities. The execution dependency rules are used by the workflow manager to determine which activity needs to be scheduled next. The execution dependency rules are various scheduling pre-conditions and they are described as axioms within the framework of the EC.

The main concern of this thesis is the design of a workflow manager within the framework of the EC and the implementation of that system using different technologies. So, before going into the details of the technologies used to implement that system, the EC and the usage of the EC to specify and execute workflows over the implemented framework must be described.

While implementing the framework, some additions and also modifications are done on these predicates. For example, because of the possibility of non-terminating loop problem described in the same paper, new predicate *happened* is used instead of the predicate *happens*, which is the main predicate of the EC. The main difference between these to predicates is that the predicate *happened* checks only the events that are known to have happened while the predicate *happens* checks all possible events. Modified predicates will be used in the axioms used to define workflow manager through the EC and the original rules in [19] are put in **Appendix A**.

3.2. Specification of workflows using the Event Calculus

This section presents a summary of the work in [19] to explain the specification of workflows in a logical framework and the rules to specify the execution requirements of workflows.

3.2.1. Activities by events

The occurrences of events are considered as instantaneous happenings in the EC, so that the events have no duration. But, from the workflow point of view, agents need some time to carry out their tasks, so the activities must have some time duration. Depending on the nature of the activity, the period of time needed to finish an activity can be either fixed or varying amount. As an example of activity needing fixed amount of time, an automatic mechanical task can be given. But, an activity performed by a human may need varying amount of time to complete the task.

Generally in workflow systems, the details of the internal operations of the activities are not interested by a workflow specification, but the way the activities are sequenced is. A workflow manager is concerned only with those aspects of an activity that are externally visible on the workflow level. So that, from the workflow manager point of view, an activity can be in one of the possible execution states and state transitions are enabled in terms of externally observable events.

In the implemented framework each activity is initiated by an event and its termination is regarded as another event that records the completion of that activity. Once the occurrence times of these events are known, the duration of the activity can be derived easily. The internal operation of the activity is unknown to the workflow manager, and the activity is in execution state between these two special events.

The workflow manager assigns activities to agents and the activities are executed by the corresponding agents. By recording the times of occurrences of the starting and ending events for an activity, the workflow manager can maintain the state of an activity. The starting event of an activity is triggered by the workflow manager, and the ending event of an activity is sent by the agent to the workflow manager. The conditions that describe the end of the activity may be produced by the agent performing the activity. For instance, the activity may be a computer program and

it may finish only when the user of the program fills in and submits a form. Such an input can be considered as an external event. Then the agent will terminate its execution by sending end activity event to the workflow manager. The execution duration of an activity is therefore application dependent and the activity must be designed to inform the workflow manager of its completion.

The activities are viewed as independent modules executed by proper agents and the implementation details of activities are not described in detail. Only their interfaces with the workflow manager in terms of their starting time, ending time and any relevant data that they generate to affect the workflow execution are described.

3.2.2. Activity scheduling

The execution order of activities depends on the successor relation among activities, and conditions that are currently satisfied on the system state. To establish the local execution dependencies between the activities within the same workflow instance, the unique identity (workflow instance id) given to the workflow instance by the workflow manager is used.

The predicate *follows* is used to define the execution dependencies between the activities:

> *follows(Activity1, Activity2, WorkflowInstance, Time):* The activity ***Activity2*** follows the activity ***Activity1*** in the workflow instance **WorkflowInstance** at time point **Time**.

The rules for the predicate *follows* for each successor relation considered in this work are described below. These rules, mainly, describe the scheduling pre-conditions of activities and therefore they are named as axioms for scheduling (AxS).

Figure 3.1 shows a graphical representation of sequential routing of activities. When activity act_i finishes, the next activity act_j can start unconditionally.

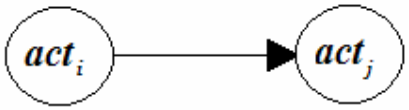


Figure 3.1 Activity act_j starts when activity act_i finishes

For sequential activities, execution dependency rule is written as:

follows(Activity1, Activity2, WorkflowInstance, Time) ←
sequential(Activity1, Activity2),
happened(end(Activity1, _, WorkflowInstance), Time). (AxS1)

In a workflow instance ***WorkflowInstance*** at a time point ***Time***, ***Activity2*** follows ***Activity1*** unconditionally when ***Activity1*** is completed by any of the qualified agents in the same workflow instance at the time point ***Time***. By using *happened* predicate, only the events that are known to have happened are checked and the possibility of having endless loops because of the call to the predicate *happens* is eliminated.

Activities after an AND-split are scheduled to be executed concurrently in a workflow. An AND-split is illustrated in **Figure 3.2-(a)**. When the activity act_i finishes, activities act_{a1}, act_{a2} ..., act_{an} will start concurrently. **Figure 3.2-(b)** illustrates *AND-join*. The activity act_j will start when all the preceding activities act_{b1}, act_{b2} ..., act_{bm} are finished by the corresponding agents.

All subsequent activities will be scheduled when the end of activity act_i is recorded by the workflow manager. Also, the activity act_j can only be scheduled by the workflow manager when the ending events of all its predecessor activities are recorded.

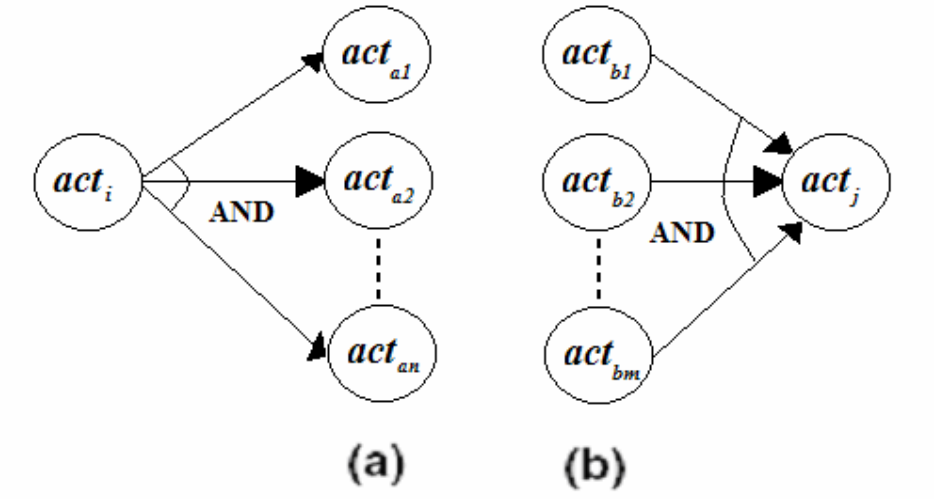


Figure 3.2 (a) AND-split and (b) AND-join

Thus the representation of the execution dependency of an AND-split is described by the following rule:

follows(Activity1, Activity2, WorkflowInstance, Time) ←
and_split(Activity1, ActivityList),
happened(end(Activity1, _, WorkflowInstance), Time),
member(Activity2, ActivityList). (AxS2)

If ***Activity2*** is a member of the activity list ***ActivityList*** in AND-split, the predicate *member* will be *true*. Each activity in the activity list ***ActivityList*** follows ***Activity1*** concurrently in a workflow instance ***WorkflowInstance*** at a time point ***Time***, when ***Activity1*** is completed by any of the qualified agents in that workflow instance at the given time point.

The following rule is used to represent the execution of an *AND-join* of activities:

follows(Activity1, Activity2, WorkflowInstance, Time) ←
and_join(ActivityList, Activity2),
findActEndTimePairs(ActivityList, WorkflowInstance, ActEndTimePairs),
actWithMaxEndTime(ActEndTimePairs, Activity1, Time). (AxS3)

The predicate *findActEndTimePairs* is used by the rule to find out whether all predecessor activities in ***ActivityList*** are completed in a workflow instance ***WorkflowInstance*** or not. If this predicate holds, ***ActEndTimePairs*** will be the list of all predecessor activities of act_j together with their ending times. Then the predicate *actWithMaxEndTime* picks the predecessor activity with the latest ending time from the list. In **Figure 3.2-(b)**, activity act_j must wait for the completion of all predecessor activities act_{b1}, act_{b2} ..., act_{bm}. The last conjunct in this rule ensures that act_j is scheduled at the time of the last ending activity among activities act_{b1}, act_{b2} ..., act_{bm}.

The 3-argument predicate *findActEndTimePairs* (see **Appendix A** for the original predicate definition, and see **Appendix C** for modified version) finds the ending times of all predecessor activities in an *AND-join*. The third argument is a list of (activity, ending time) pairs if all the incoming activities have completed their executions. The 3-argument predicate *actWithMaxEndTime* (see **Appendix A** for the original predicate definition) simply calls its 4-argument definition in order to find the maximum ending time in the list of (activity, ending time) pairs. The subsequent activity in an *AND-join* can start execution only if all incoming activities are completed. Therefore the maximum ending time is found to determine the starting time of the subsequent activity.

Depending on the evaluation of the conditions, one of the alternative activities executed in a workflow instance if there is XOR-split after the activity currently executing.

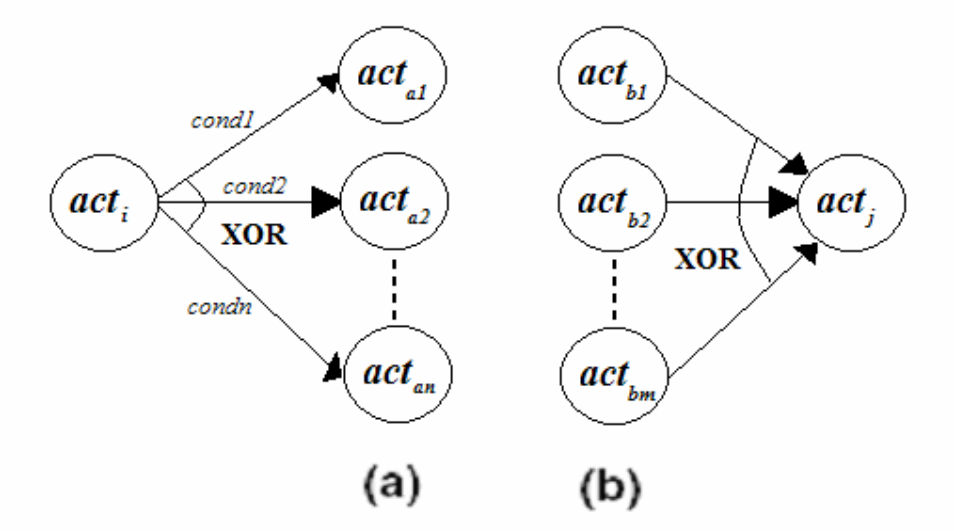


Figure 3.3 (a) XOR-split and (b) XOR-join

In an *XOR-split*, when the activity act_i ends, one of the activities act_{a1}, act_{a2}, ..., act_{an} will start depending on the condition satisfied at that time.

follows(Activity1, Activity2, WorkflowInstance, Time) ←
xor_split(Activity1, ActCondPairs),
happened(end(Activity1, _, WorkflowInstance), Time),
member((Activity2, Condition2), ActCondPairs),
initiates(Event, Condition2),
happened(Event, Time2),
max([Time1\Time2], Time),
holds_at(Condition2, Time). (AxS4)

One of the conditions at the split should evaluate to true. If not, then none of the branches can be chosen by the workflow manager. By using *member* predicate, each activity-condition pair is picked from the list of activities ***ActCondPairs*** and checked whether the corresponding condition is evaluated to true or not. The picked activity ***Activity2*** will be scheduled by the workflow manager in a workflow instace ***WorkflowInstance*** at time point ***Time*** only if ***Time*** is the later of the two time points:

- The ending time of activity ***Activity1***
- The time of the event that initiates the condition ***Condition2*** for activity ***Activity2.***

Also ***Condition2*** must be checked to see whether the condition still holds at time point ***Time***.

In an *XOR-join*, if any one of the incoming activities is finished, the activity at the join can start executing. Thus the *XOR-join* is represented by the following rule:

follows(Activity1, Activity2, WorkflowInstance, Time) :-
xor_join(ActivityList, Activity2),
findOneActEndTimePair(ActivityList,WorkflowInstance, Activity1, Time)

(AxS5)

The rule uses the predicate *findOneActEndTimePair* which holds when one of predecessor activities in ***ActivityList*** is completed in a workflow instance ***WorkflowInstance***. If this predicate holds, ***Activity1*** will be the completed predecessor activity and ***Time*** will be its ending time. Thus, the subsequent activity is scheduled at time point ***Time*** of the first ending activity. The 3-argument predicate *findOneActEndTimePair* (see **Appendix A** for the original predicate definition, and see **Appendix C** for the modified version of it) finds the predecessor activity, that has been completed in an *XOR-join*, with its ending time. It simply checks each activity in the *XOR-join* with the predicate *member* to see whether it has been finished.

The event occurrences for the activities carried out by the corresponding agents must also be identified uniquely as done for the activities of concurrent instances of same workflow. One activity may be executed by different agents in different instances of workflow which are running concurrently. So the agent assignment must also be considered in the naming of the events. For this purpose, the unique

identifier of the workflow instance is used in the predicate used to describe events such as the one used in above predicate. (i.e., *end(Activity, Agent, WorkflowInstance)*)

3.2.3. Workflow state

The specification and execution of activities must be permitted by the workflow management systems. The axioms necessary for the specification of workflow activities and the description of scheduling pre-conditions among the activities are presented so far within the current logical framework. In this section, the execution semantics of the workflows through the EC will be explained. The representation of the system state maintained by the workflow manager, and the rules for the execution of activities by appropriate agents are described in the following sub-sections.

At any time the execution state of a workflow can be defined as a collection of states of its constituent activities and agents. The occurrences of events and the execution of activities cause changes in the state of workflow. The EC axioms are used to derive the state of the workflow. The workflow manager makes an agent assignment for the activities and also schedules new activities according to the specification. At any point in time, one can check activities executing or completed at that time point, or agents assigned to any task.

Each activity is characterized by a set of executable states and transitions between these states. An activity may be in either of the following states:

- ***waiting****:* An initial state of an activity. An activity is put into the waiting list of the agent(s), capable of doing that activity and it is waiting for any of the assigned agents to execute it.

- ***active****:* The executing state of an activity. One of the assigned agent is currently executing that activity.

- ***completed****:* Done state of an activity. One of the assigned agent is finished that activity.

The activity enters in *waiting* state, when the workflow manager determines the next activity to be executed, and puts that activity into the worklists of all agents that can perform that activity. If an agent retrieves the activity from its worklist and starts executing it, then the activity enters in *active* state at that time point. When the agent finishes executing the activity, the activity enters the *completed* state finally.

Each agent has a worklist showing which activities are waiting for that agent. The property *waiting* is also used to represent the worklists of agents since it includes the information about which activity is waiting for which agent. The property *waiting(Activity, Agent, WorkflowInstance, Time)* describes that activity ***Activity*** is waiting for agent ***Agent*** in a particular workflow instance ***WorkflowInstance***. The time variable ***Time*** denotes the point of time at which the activity started waiting for the agent.

An agent can be in either of the following two states:

- ***idle:*** An agent is in *idle* state when there is no activity in the worklist of the agent and the agent is not assigned to any activity. This state of an agent is described by a predicate *idle(Agent).*

- ***assigned:*** The agent is in *assigned* state when an activity is in *active* state with that agent. This state of an agent is described by a predicate *assigned(Agent, Activity, WorkflowInstance).*

The state of the agent is changed by the following events:

- *assign(Agent, Activity, WorkflowInstance):* An agent ***Agent*** is assigned to an activity ***Activity*** in a workflow instance ***WorkflowInstance.***

- *release(Agent, Activity, WorkflowInstance):* An agent ***Agent*** is released an activity ***Activity*** in a workflow instance ***WorkflowInstance.***

In addition to the time dependent description of the workflow state, there are also static properties of the workflow. The agent definitions, the activities for which

they are qualified are static properties of the workflow and they are defined in the workflow specification. In order to represent the relationship between the activities and agents we use the predicate *qualified(Agent, Activity).*

The time-dependent states for activities and agents together with the events causing the transitions between these states are summarized in **Table 3.2** and **Table 3.3** respectively.

Table 3.2 Execution states of activities

State of Activity	***Meaning***	***Initiating Event***
active(***Act, Ag, W***)	***Act*** is being executed by ***Ag*** in workflow instance ***W***	start(***Act, Ag, W***)
completed(***Act, Ag, W***)	***Act*** is completed in workflow instance ***W***	end(***Act, Ag, W***)
waiting(***Act2, Ag2, W, T***)	***Act2*** is in worklist of ***Ag2*** in workflow instance ***W*** with timestamp ***T***	release(***Ag1, Act1, W***)

Table 3.3 States of agents

States of Agents	***Meaning***	***Initiating Event***
idle(***Ag***)	***Ag*** is idle	release(***Ag, Act, W***)
assigned(***Act, Ag, W***)	***Ag*** is carrying out ***Act*** in workflow instance ***W***	assign(***Ag, Act, W***)

The rules to describe how these events cause state transitions are presented in the **Appendix A** part of this thesis and these rules are named as axioms for initiates/terminates (AxIT) for reference purposes.

After the starting event of an activity is recorded by the workflow manager, it becomes active in a corresponding workflow instance.(AxIT1) Also the recording of an end event for that activity sets up a completed state for it (AxIT2), terminating its active state (AxIT3). If an agent starts to execute one of the

activities from its worklist, it is not in idle state any more (AxIT4) and it is assigned to the executed activity until finishing that activity (AxIT5). An agent becomes idle, after finishing or releasing the activity currently executing by that agent (AxIT6, AxIT7). If there is no activity in the worklist of the agent, it will remain in the idle state. If there are one or more activities waiting for that agent in the agent's worklist, the agent will be assigned to the next activity in its worklist.

The property *waiting(Activity, Agent, WorkflowInstance, Time)* is used to represent both the state of an activity, and the worklists of agents. An agent is released when it completes an activity and the subsequent activity is enabled by the workflow manager. The subsequent activity is inserted to the worklists of all agents qualified to do that activity (AxIT8). When an activity is assigned to an agent, the activity is no longer in waiting state (AxIT9). This activity no longer exists in the worklists of other agents that are not currently executing it.
A workflow manager generates a selection event for an activity ***Activity*** with condition evaluating to true, and this activity is put into all of the qualified agents' worklists. Ending event for that activity terminates the selection of that activity. These rules are added to the system to be able to implement the XOR condition evaluation operation. Because the data values needed to evaluate the conditions are not known by the EC, this evaluation is done by the manager and by using event *select*, the manager indicates that the condition for the activity ***Activity*** is evaluated to true. (see Section 4.2.2)

initiates(select(Agent, Activity, WorkflowInstance),
selected(Activity, WorkflowInstance)).
terminates(end(Activity, _, WorkflowInstance),
selected(Activity, WorkflowInstance)).

3.2.4. Execution of workflows

It is a critical issue for the workflow manager to assign an activity to appropriate agents in order to execute workflow. If an activity is not an automated one, it will be in *waiting* state till any of the agent having that activity in its worklist starts to

execute that activity. The axioms given in this section are used to record new event occurrences in the history through the predicate *happened.* Therefore the rules are named as axioms for happens (AxH).

The execution of an activity can start only when an agent is assigned to that activity. As soon as the agent is assigned, the starting event of the activity is generated, which is described by the following rule (AxH1):

happens(start(Activity, Agent, WorkflowInstance), Time) ←
happened(assign(Agent, Activity, WorkflowInstance), Time).

In a workflow instance ***WorkflowInstance***, the happening of the assign event of an activity ***Activity*** for an agent ***Agent*** means that the happening of the starting event of that activity is occurred at the same time point ***Time***. And, when an activity is completed, the ending event of the activity is recorded and the agent that completed the activity is released (AxH2).

happens(release(Agent, Activity, WorkflowInstance), Time) ←
happened(end(Activity, Agent, WorkflowInstance), Time).

The workflow manager is an interpreter to generate events that start and assign agents to activities through the event generation rules. In order to start generating the events (and thus, start the execution of workflow instances), the manager needs to know what initiates the workflow and also the initial state of the system. In this framework there must be an external event to start the workflow. This initial event must be defined in the workflow specification. In addition, all agents are in idle state at the beginning. In order to set all agents idle initially, an event having affect of initiating the idle property for all agents, called *free_agent(Ag)*, is defined (AxIT10). The manager starts a workflow instance when an initial external event happens. When that starting external event is recorded, the manager schedules the first activity of the workflow by inserting it into the worklists of all

agents qualified to perform that activity. The workflow manager will keep scheduling the next activity for each completed activity using the execution dependency rules (AxS1 – AxS8) and event generation rules (AxH1 – AxH2) until the end of the workflow is reached (or until the current time). In order to start this process, following rule is written, so that when the initial event happens, the first activity can be scheduled:

initiates(Event, waiting(Activity, Agent, WorkflowInstance, Time)) ←
initial_activity(Activity),
starts(Event, WorkflowInstance),
happened(Event, Time),
setEID(Activity, WorkflowInstance),
qualified(Agent, Activity). (AxIT13)

The starting event is defined with the predicate *starts*. The predicate *starts* also generates a unique workflow instance id ***WorkflowInstance***. Thus, this rule represents that when the event which starts the workflow instance ***WorkflowInstance*** happens at time point ***Time***, the first activity of the workflow starts waiting for all qualified agents. The predicate *setEID* sets the execution id of the initial activity of the workflow instance to the workflow id ***WorkflowInstance***.

CHAPTER 4

THE ARCHITECTURE AND IMPLEMENTATION OF EVENTFLOW

The **W**ork**f**low **M**anagement **S**ystems (WFMS) provide the procedural automation of a business process by the management of the sequence of work activities and the invocation of appropriate human and/or automated software program associated with the various activity steps. An individual business process may have a life cycle ranging from minutes to days (or even months), depending upon its complexity and the duration of the various constituent activities. Such systems may be implemented in a variety of ways. Despite this variety, all WFMSs exhibit certain common characteristics, which provide a basis for developing integration and interoperability capability between different products. In the **W**ork**f**low **M**anagement **C**oaliation (WfMC) Reference Model [14], a common model for the construction of workflow system is described.

At the highest level, all WFMSs may be characterised as providing support in three functional areas [14]:

- *Build-time functions*, concerned with defining, and possibly modelling, the workflow process and its constituent activities. *EventFlow Editor* provides that functionality in our system.

- *Run-time control functions*, concerned with managing the workflow processes in an operational environment and sequencing the various activities to be handled as part of each process. *EventFlow Engine* provides that functionality.

- *Run-time interactions* with human users and **I**nformation **T**echnology (IT) application tools for processing the various activity steps. *Worklist and activity implementations* provide this functionality.

Figure 4.1 illustrates the basic characteristics of WFMSs and the relationships between these main functions.

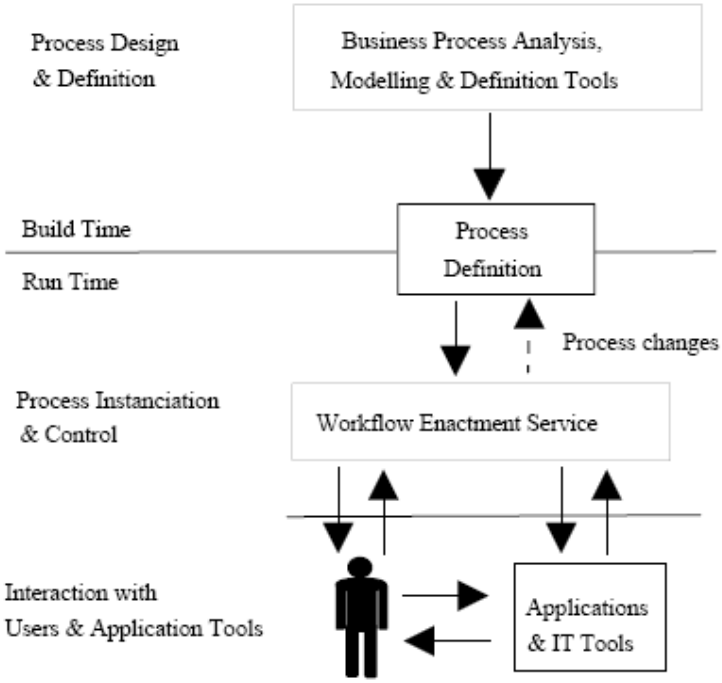


Figure 4.1 Workflow System Characteristics

In this chapter, the computational aspects of the logical description discussed in the previous chapter, including details of the system architecture and the system implementation (such as technology used to develop system, open source components used for the main parts of the system, etc.), are discussed. The next chapter presents a case study which is designed with the implemented system. Also there is a brief user manual for EventFlow Editor in **Appendix D** of this thesis.

4.1. Architecture

Figure 4.2 depicts the components of the system architecture for EventFlow. The workflow state is described as a deductive database. The records of event occurrences are considered to be an extensional database, called the history. The intentional database includes the event calculus rules, workflow specification and activity execution dependency rules, and workflow execution rules. The set of known events and the set of possible workflow states are immediately characterized in terms of the set of all logical consequences of this deductive database. All these are kept by XSB, an open source Prolog interpreter, and an open source library InterProlog is used by the EventFlow engine to communicate with this component.

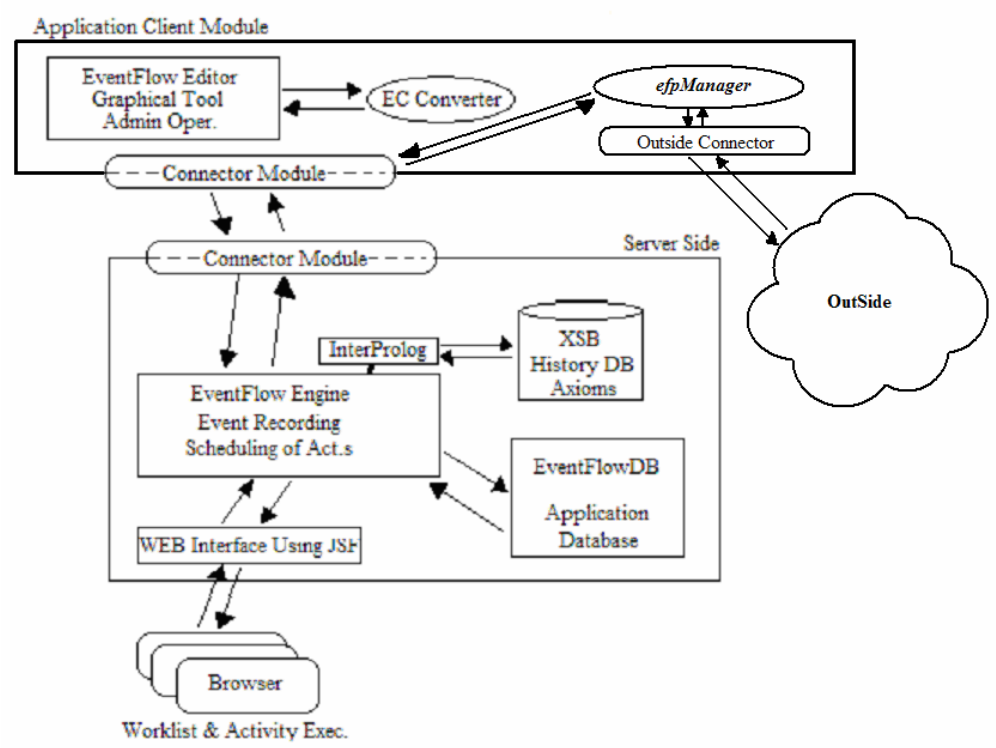


Figure 4.2 EventFlow System Architecture

Conceptually speaking the database states need not be independently stored, since they follow logically from the history. The history only needs appending event occurrences to, in order to record that some event has happened in the modeled

reality. But, to be able to store workflow application data and to be able to keep the event history even the system crashes, a permanent database is needed. This database is used to store data from applicational point of view and for administrative purposes. For this purpose, *Apache Derby*, an open source relational database developed by Java, is used.

In order to give the user the ability of defining workflows with a graphical user interface in an easy way, a visual workflow editor tool (EventFlow Editor) is developed by using Java Swing components and an open source Java graph library JGraph. This tool also provides a capability to define new agents and new activities. Also, a Graph-to-EC-Axioms converter module has been implemented. This converter automatically generates the first order predicate form of the workflow drawn by the user when it is saved to the disk or the system database. It uses Connector Module to send data to or get data from the EventFlow Engine.

Also the end user/agent can get his worklist and do the appropriate operation for the selected/assigned activity via Web interface of the system implemented using Java Server Faces Technology.

The module called *efpManager* is used to run the automated activities. These automated activities may need to run an application to communicate with the outside world and according to the result taken from that application, the state of the activity can be changed and the next activity can be scheduled by the manager. Automated activities are assigned to the agent *eventflow* with the role *manager.*

A typical cycle in this architecture can be described as follows. The environment notifies the system the start of a new workflow instance by appending an external event that initiates the workflow. At the same time this is logged into the EventFlowDB. Since the set of known events (i.e. history) now includes at least one event, the interpreter reacts to this change by scheduling the first activity in the workflow. The first activity is placed to the worklists of qualified agent(s). Agent queries its worklist and sees the *waiting* activities, and checks out one of them from the list. This means that an *assign* event is occurred and the activity

state is changed to active. After the agent finishes the required operation, it sends the *finish* activity request. The end of the activity is recorded in the history and also in the event log at the applicational database. Then, the interpreter uses the execution dependency rules and agent assignment rules to put the next activity to the corresponding agent(s)' worklist. Meanwhile, the environment may record the beginning of another workflow instance, or the executed activities may insert new (external) events to the history. The interpreter proceeds to coordinate the activities by reacting these new happenings until a saturation state is reached in which all possible events have been derived.

4.2. Implementation Details of EventFlow System

The logic-based formalization of the workflow using the EC as base can be implemented using various techniques. To support the basic characteristics given at the beginning of this chapter, different software components are brought together while developing the EventFlow system. For example, the given axioms are implemented directly in Prolog, although, it is possible to implement these by using Java as the other parts of the system.

For the implementation of the system, an Enterprise Application Project named *EFProject* created according to the J2EE 1.3 specifications with basic modules included. These modules are shown in **Figure 4.3,** and **Figure 4.4** shows the J2EE hierarchy of these modules. Here, *EFProjectApplicationClient* includes the implementation of administrative purpose user interfaces and a workflow editor which gives the user the ability of defining new workflow templates. Also the editor has an ability to convert graphical representation of the workflow into first order predicate form. *EFProjectConnector* module is responsible for providing communication functionality between the server part and the client part of the system. *EFProjectCommons* contains the common objects, used by all or some of the other modules, such as an object to implement *static* values or to carry data between server side and client. *EFProjectManager* includes manager that is responsible for executing automated tasks assigned to the agent *eventFlow,* and also the outside connector in that module gives an ability to communicate with

outside world such as the database, an application program or a Web service. *EFProjectEJB* and *EFProjectWeb* construct the server part of the project together. *EFProjectEJB* contains all the logic to implement the workflow engine, and provides communication with the external components like Prolog environment and the database. *EFProjectWeb* provides an interface for the browser and supports communication between *EFProjectEJB* and client part (such as EventFlow Editor or worklist opened by a user using browser).

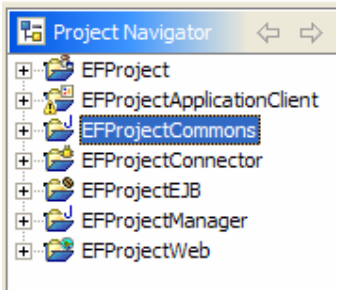


Figure 4.3 EFProject Modules

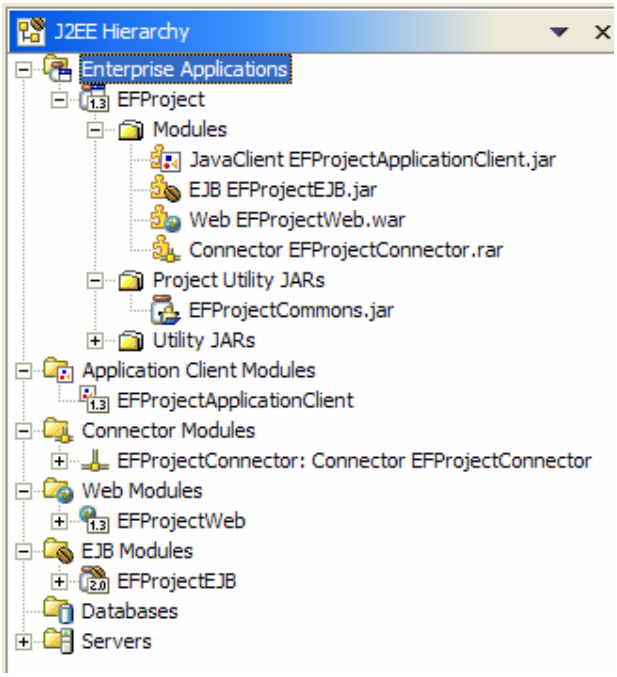


Figure 4.4 J2EE Hierarchy for EFProject Modules

As described before, our system supports three main functional areas as the other WFMSs. These areas and the system components implemented to support given functionality are described in the following sections.

4.2.1. Build-time Functionality

The *Build-time functions* are those which result in a computerised definition of a business process. During this phase, a business process is translated from the real world into a formal, computer processable definition by the use of one or more analysis, modelling and system definition techniques. A process definition normally comprises a number of discrete activity steps, with associated computer and/or human operations and rules governing the progression of the process through the various activity steps. The process definition may be expressed in textual or graphical form or in a formal language notation.

Previously mentioned project module *EFProjectApplicationClient* provides the above functionality. In this thesis, EventFlow Editor is implemented to support graphical representation and modeling of the workflows by the user. Generating first order predicates from graphical representation of the workflow is provided by this editor also. This is done while saving the designed workflow to the system or the disk by user. An open source Java graph library JGraph is used for graph visualization functionalities and drawing utilities. It is a powerful, easy-to-use, feature-rich and standards-compliant open source graph component available for Java.

Figure 4.5 shows EventFlow Editor Screenshot. Also **Figure 4.6** shows the modeling elements used in this implementation. The clock at the *AND-join* element means that all the activities before that must be finished, in other words, this is a synchronization point for the activities connected to that node in the graph. The question mark at the *XOR-split* means that this is a decision point to select the next activity with respect to input condition given.

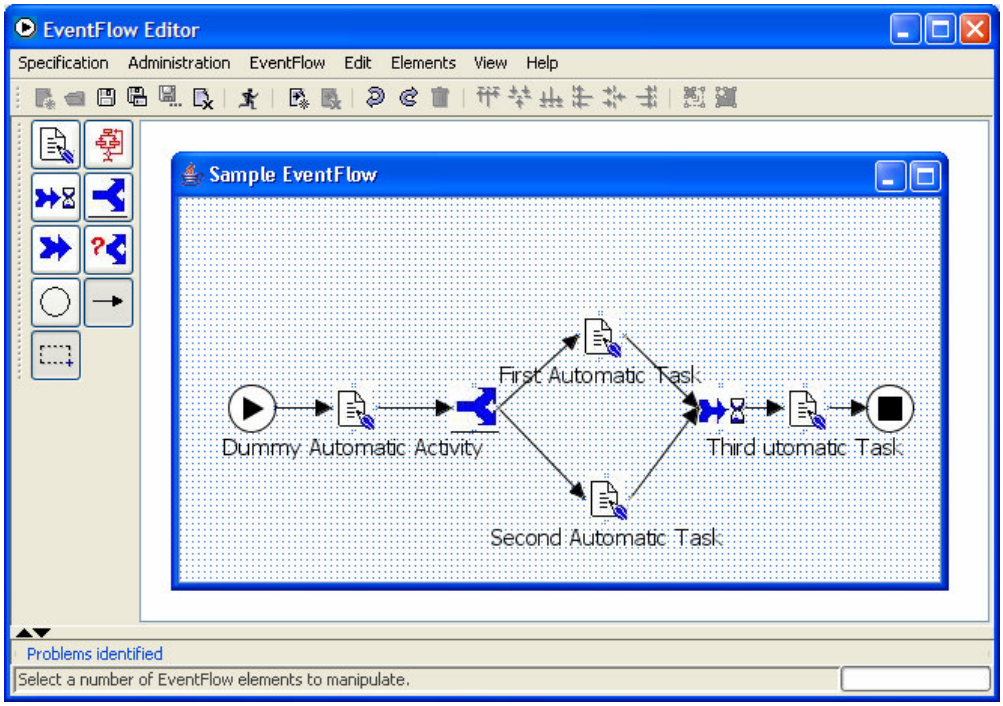


Figure 4.5 EventFlow Editor Screenshot

Each workflow specification has a starting activity and an ending activity (the circles with triangle and square). These are automatic activities that are executed by the workflow manager. The other activities are designed by the user and can be executed by the workflow manager and the agents interactively.

By using the dialogbox, reached through the activity node popup menu item "*Edit Activity Details*", one can assign agents or group of agents to a selected activity node by choosing them from the list of agents that are previously defined in the database, and also select an operation to be executed by the agents qualified when the activity is activated by any of the agents for that node. Operation description label is shown at the bottom of the activity node on the graph definition.

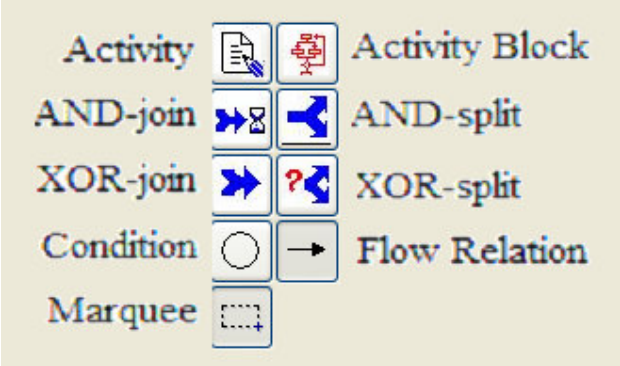


Figure 4.6 Modeling Elements of EventFlow

Although, in the formalization part of this thesis, there aren't any axioms to describe the loop structures and compensation operations, one can easily add these abilities to his model by using simply "*XOR*" nodes.

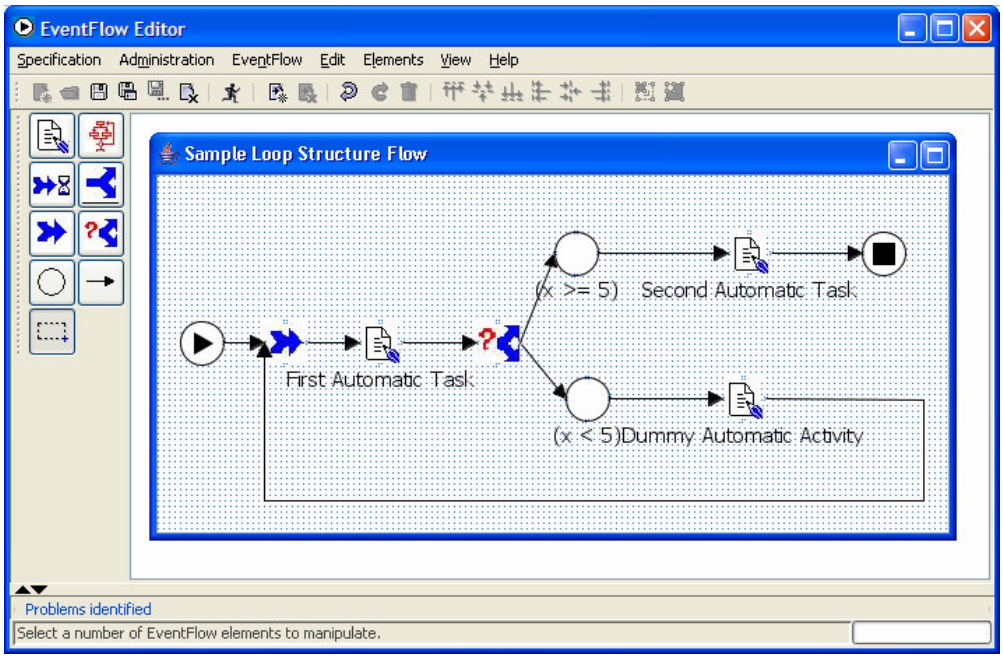


Figure 4.7 Loop implementation

Figure 4.7 shows a possible loop implementation in the EventFlow architecture. Here, the first activity will be done till the value of ***x*** becomes more than 5. Also, the activity after the condition "*x*<5" will be an automatic dummy operation; it

will just reinitiate the first activity. This is needed because it is not permitted to directly connect join and split nodes.

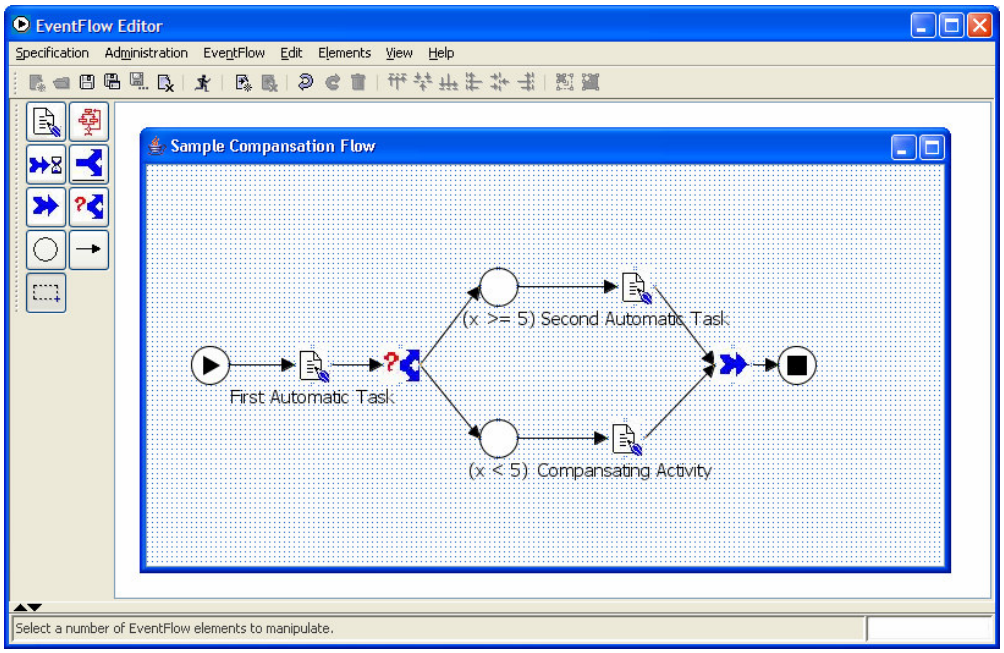


Figure 4.8 Compansation alternative

Figure 4.8 shows the specification of compensation ability for an activity. Here, let us say that if the value of ***x*** is greater then or equal to 5, then it is needed to rollback all the changes done at the application level till that point. This can be thought as an exceptional state for the workflow with respect to the application that uses the given workflow for its applicational purposes. In short, this alternative cannot give any information to the workflow manager about the exceptional state of the workflow. So, it just continues to run the executing instance of the workflow. Thus, from the workflow manager point of view, the workflow instance will end normally. But, by implementing new activity to compensate the exceptional state of the running workflow instance, and by defining the condition that causes this exception properly, an application developer can easily do the necessary operation at the application level such as sending an e-mail indicating the exceptional state of the workflow to any related agent like an application admin, or deleting and/or updating some data from the

application database to rollback the changes done by the agents previously assigned to the activities and executed them in this workflow instance.

After modeling a business process, one can save this as a template in the database. When this is done the editor converts the graphical representation into its first order predicate form and also generates the EC based axioms, and then stores these in the database for future use. For object serialization purposes XStream, a simple library to serialize objects to XML and back again, is used. Database part will be explained in the next section. After saving the workflow specification, the user can start a new instance or reopen it for editing purposes.

EventFlow Editor also provides an administrative user with the ability to define new operations and agents on the system. Details are described in a brief user manual for EventFlow Editor presented in **Appendix D** part of this thesis.

4.2.2. Run-time Process Control Functionality

At run-time, the process definition is interpreted by the software which is responsible for creating and controlling operational instances of the process, scheduling the various activities within the process and invoking the appropriate human and IT application resources. These run-time process control functions act as a linkage between the process as modelled within the process definition and the process as it is seen in the real world, reflected in the runtime interactions of users and IT application tools. The core component is the basic workflow management control software (or "engine"), responsible for process creation and deletion, control of the activity scheduling within an operational process and interaction with application tools or human resources.

Above functionalities are provided by the *EFProjectEJB* module of the EventFlow system. This module implements a simple logic to do required functionality and make a bridge between the external components such as the database and client applications.

The EC axioms are directly implemented in Prolog and compiled and loaded to the underlying Prolog interpreter. In this part, ***XSB Prolog***, a research-oriented Logic Programming system for Unix and Windows/DOS-based systems, representing a semantically enriched functional superset of Prolog and offering among other things evaluation through full SLG resolution – a table-oriented resolution method, is used for the evaluation of the axioms and also it is used as an event database for the system. ***Interprolog*** library is used to be able to communicate with ***XSB Prolog***. InterProlog is an open source Java front-end and functional enhancement for standard Prologs, running on Windows, Linux and Mac OS X. It consists of a Java application front-end that communicates with a Prolog system running either as a subprocess, using standard console redirection and TCP/IP sockets, or as a dynamic loadable library, using the Java Native Interface. It provides Java with the ability to call any Prolog goal through a PrologEngine object, and for Prolog to invoke any Java method through a javaMessage predicate, while passing virtually any Java objects and Prolog terms between both languages with a single instruction.

In addition to the Prolog implementation part of the engine, there are some other parts implemented for administrative purposes, end user operations and sending relevant event occurrences to the event database (XSB) and getting worklist for any agent sending request to the engine. These parts are Java codes. For administrative purposes such as adding a new agent or activity (operation), saving or modifying workflow template, and also recording event log, there must be a way to keep all necessary data. The easiest way of doing that is to put all necessary data into a database. ***Apache Derby*** is used to store the necessary data. It is a relational database implemented entirely in Java.

In **Figure 4.9**, the tables created at the database and relations between them are shown. Also the create scripts are given in Appendix B.

The table ***RoleDetails*** keeps the role information of the agents. By using a role value, one can assign an activity to a group of agents. This can be done by the following *qualified* predicate.

qualified(agent(_, role1), act(act1, w1)).

This predicate indicates that any agent having role value ***role1*** is qualified for activity ***act1*** in a workflow instance with execution id ***w1.*** So, when an agent from *role1* logged on to the system and queries his worklist, he will see this activity in his worklist if it is in waiting state for the current workflow instance.

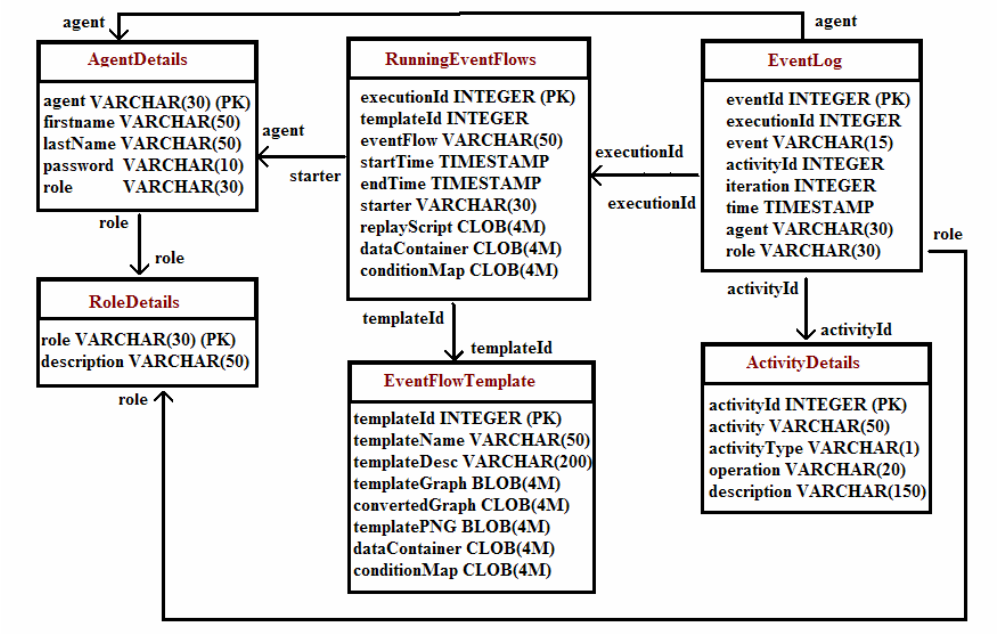


Figure 4.9 Database Tables

The table ***AgentDetails*** contains basic information about the defined agents. Validation of the agents and an agent list for activity assignment operation are maintained by using this table. For the automated activities, a static agent definition is put into this table. Values defined for that agent is "*eventFlow*", "*Event*", "*FLOW*", "*manager*" and "*manager*". The *efpManager* takes the worklist of this agent and performs the defined operation for each activity in that list.

The table ***ActivityDetails*** is used to store detailed information about the activities such as which operation will be executed when it is activated by the agent, or whether it is an automatic operation or not. If it is an automatic operation,

efpManager will execute the corresponding operation for that activity as soon as the activity is taken from the automatic task queue. Tasks are put into this queue by the EventFlow Manager by assigning these activities to the special agent *eventFlow*. Actually, the queue corresponds to the worklist of the agent *eventFlow*.

The table ***RunningEventFlows*** is used to store extra information about running instances of the workflow such as the corresponding template id, data container including the values assigned to the data for the running instance of the workflow and also conditions with the related activity values if the running workflow contains XOR node. Although, some value about workflow is shown to the user in his worklist, these are not kept in the deductive database provided by Prolog interpreter. So it is needed to store these data somewhere else. This table is used for this purpose.

The table ***EventLog*** is used to keep events occurred in the system. When an event occurs, it is also recorded in that table. If the system crashes the system state can be recovered by using the data recorded in this table. Also, it can be used for reporting facilities.

The table ***EventFlowTemplate*** is used to store template workflow definitions for future use. The field ***templateGraph*** contains the serialized version of the graphical representation of the specification, ***convertedGraph*** field contains the rules obtained by converting graph to first order predicate form and ***templatePNG*** contains the visual image of the workflow template.

The Primary Keys, that are Integer type, are generated by the database automatically by using the auto generation utility of *Apache Derby*.

4.2.3. Run-time Activity Interactions

Individual activities within a workflow process are typically concerned with human operations, often realised in conjunction with the use of a particular IT tool (for example, form filling), or with information processing operations requiring a

particular application program to operate on some defined information (for example, updating an orders database with a new record). Interaction with the process control software is necessary to transfer control between activities, to ascertain the operational status of processes, to invoke application tools and pass the appropriate data, etc.

This part includes the implementation of an application built on the developed framework. It also provides interfaces to indicate that the activity is checked out, completed or released by an agent. There is also Worklist (or inbox) implementation for the agents. This will be used to see *waiting* activities and select one of them and do the appropriate operation.

In this thesis, JavaServer Faces technology is used for the implementation of a sample application (described in **Chapter 5)** and for the implementation of worklists. JSF technology simplifies building user interfaces for JavaServer applications. Developers of various skill levels can quickly build web applications by: assembling reusable user interface components in a page; connecting these components to an application data source; and wiring client-generated events to server-side event handlers. For these reasons, JSF is selected for the implementation of the case study in **Chapter 5** of this thesis.

CHAPTER 5

AN EXAMPLE EVENTFLOW APPLICATION

The illustration of the use of the given formalization and the developed system will be done by an example application. It demonstrates an application of purchasing items over the Internet by authenticated customers and processing of the orders.

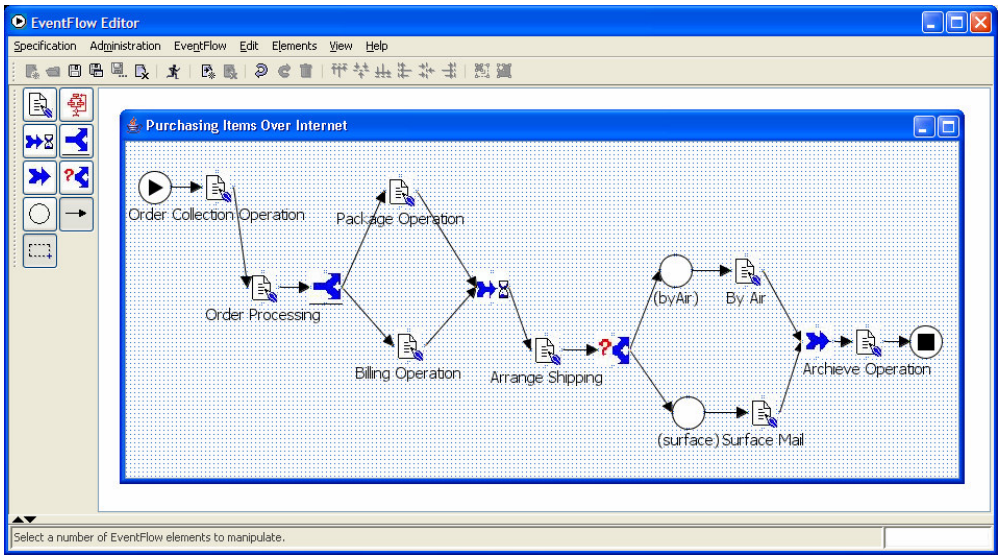


Figure 5.1 An Example EventFLow Specification

5.1. Workflow for Order Processing

A workflow (EventFlow) specification used for order processing is given in **Figure 5.1**. Instead of using activity names, primary keys generated automatically by the *Apache Derby* database system, are used for the activities. Thus, it

guarantees the uniqueness of the activities on the system. **Table 5.1** gives the values recorded in the database for the activities in the example workflow graph.

Table 5.1 ActivityDetails table content for example specification

ActivityId	*Activity*	*ActivityType*	*Operation*	*Description*
2	order_collection	A	orderCollection	Order Collection Operation
3	order_processing	U	operation	Order Processing Operation
4	package	U	operation	Package Operation
5	billing	A	billing	Billing Operation
6	arrange_shipping	U	operation	Arrange Shipping
7	byAir	U	operation	By Air
8	surfaceMail	U	operation	Surface Mail
9	archieve	U	operation	Archieve Operation
10	final_activitiy	A	finalAct	Final Activity
11	initial_activity	A	initialAct	Initial Activity

Activity with id "2" is an order collection operation, and done automatically by the eventflow manager to collect the ordered items from the sites. The value ***"A"*** indicates that the activity is an automatic one which will be executed by the automatic task invocator (e.g., *efpManager* in this framework). Activity with id "3" processes the order by updating the inventory. Activities "4" and "5" then start concurrently. Activity "4" removes the product from the warehouse and packages the item. Activity "5" performs the billing function automatically. After both activities are completed, activity "6" arranges shipping by initiating either activity "7" or activity "8" according to the selection done by the customer. Finally when the delivery is successful, the database is updated to indicate that the order has been fulfilled. In order to model and manage the execution of this workflow in the presented framework first the workflow graph specification is translated into first order logic using the predicates shown in **Table 3.1**. Thus the example workflow is translated into the following:

initial_activity(act(11,EID)).

sequential(act(11,EID), act(2,EID)).

sequential(act(2,EID), act(3,EID)).

and_split(act(3,EID),[act(4,EID), act(5,EID)]).

and_join([act(4,EID), act(5,EID)],act(6,EID)).

sequential(act(9,EID), act(10,EID)).

xor_split(act(6,EID),[(act(7,EID),selected(act(7,EID),EID)),
(act(8,EID),selected(act(8,EID),EID))]).

xor_join([act(7,EID), act(8,EID)],act(9,EID)).

final_activity(act(10,EID)).

The list of qualified agents is given as follows:

qualified(agent(eventFlow,manager),act(11,EID)).

qualified(agent(eventFlow,manager),act(2,EID)).

qualified(agent(agent1,role),act(3,EID)).

qualified(agent(agent6,role),act(3,EID)).

qualified(agent(agent2,role),act(4,EID)).

qualified(agent(eventFlow,manager),act(5,EID)).

qualified(agent(agent3,role),act(6,EID)).

qualified(agent(agent4,role),act(7,EID)).

qualified(agent(agent5,role),act(8,EID)).

qualified(agent(_,role),act(9,EID)).

qualified(agent(eventFlow,manager),act(10,EID)).

In original paper [19], agents do not have a role. By adding that value to the agent definition, it becomes to be possible to assign an activity to a group of agents without giving all agents' names. Let us say that, it is needed to assign an activity to 10 agents with having same role value. There is two possible ways to do that definition:

- Assign activity to all agents one by one. In that case there will be 10 *qualified* predicate and search space is increased

- Assign activity to common role value. (i.e., use the predicate *agent(_, role)*) Then there will be only one *qualified* predicate in search space for the given group of agents

In this example it is assumed that all activities, except the activities that are qualified to the agent *eventFlow,* are considered as non-automatic activities. These activities need human interference, thus their termination needs some external event such as waiting for the user to enter some data. For instance, activity *package* needs the operator to input data that the packaging is finished. The actual shipment of the package (by air or surface mail) is done by a person, thus the completion of this activity must be recorded by an input and this is considered as an external event.

When the activities are finished, the agent will inform the system by using its worklist. After finishing the required operation, it must select the activity from his worklist and finalize that activity. When he submits the completion, the manager fires an *end* event on the system, so, the flow continues with the next available action in the specification. As seen above, the automatic tasks (i.e. the activity does not need an human interference to complete its job) are assigned to predefined agent *agent(eventFlow, manager)*, so these activities are put into the automatic task invocator module *efpManager* execution queue when activated.

The workflow is initiated by an external event which is the submission of an order request form. Every time this event is entered to the system a new workflow instance is started. The following rule is used to specify the initialization of a workflow instance:

starts(Ev, Wno) ←
ext_event(Ev),
Ev = runWF(CID),
Wno = CID.

Here occurrence of the external event *runWF* starts the current instance of the workflow. When creating an instance of a workflow, it is assigned to a new execution id ***EID***, and this is unique for the system. All event occurrence times are recorded in not only the deductive but also the application database with the clause *happened*.

The workflow specification for the given specification is now complete. The external events to initiate the workflow instances, to end varying activities will be input to the system at various points in time. Thus, given a set of predicates for a workflow graph specification, external events and qualified agents, the axioms that are presented in [19] can be used to answer queries such as finding out the system state at a specific time, or the period of time for which a certain property holds. By querying the history of events the actual order and occurrence times of all activities can be derived.

Because of the characteristic of the Prolog system, if an agent tries to get his worklist, the system searches all history of the events to find all activities with status *waiting* for the current agent. This is a very time consuming operation for big systems with very big history database. So some more rules are added to retract unused/finished workflow instance events from the history. This rule is executed automatically by the manager when *final_activity* is reached. Then from the history database of Prolog system, one can only search for the active workflow instances' events. But, we have also an application database and all events logged there, all kind of reports can be taken from this log. This log can be used to create any snapshot of the deductive database at any time, because all events are recorded there, as they occurred.

5.2. Sample Run

Customers can order any item from the site by using "Product List" page containing the available items and the current basket of the customer shown in **Figure 5.2** after signing into a system. Available products are shown to the customer. After picking up the items for purchasing, by approving the basket for

the order operation, the given order will be recorded into application database and this operation will cause to start a new instance of the workflow shown in **Figure 5.1**. In this sample run, *customer1* and *customer2* will order some items from the site. Different delivery option will be selected by the customers (i.e., the items ordered by one of the customer will be delivered via surface mail while the other by air.) Orders will be handled by the related agents concurrently.

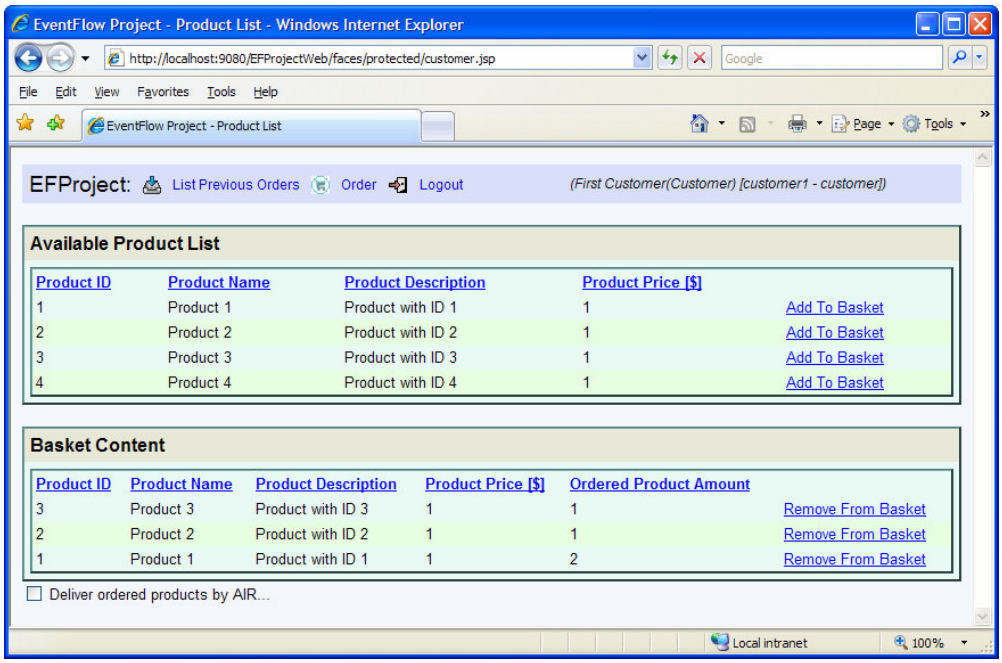


Figure 5.2 Product List page shown to the customer

Figure 5.2 and **Figure 5.3** show orders given by the customers. *Customer2* wants his items to be delivered by air, indicated by the checkbox shon on the page.

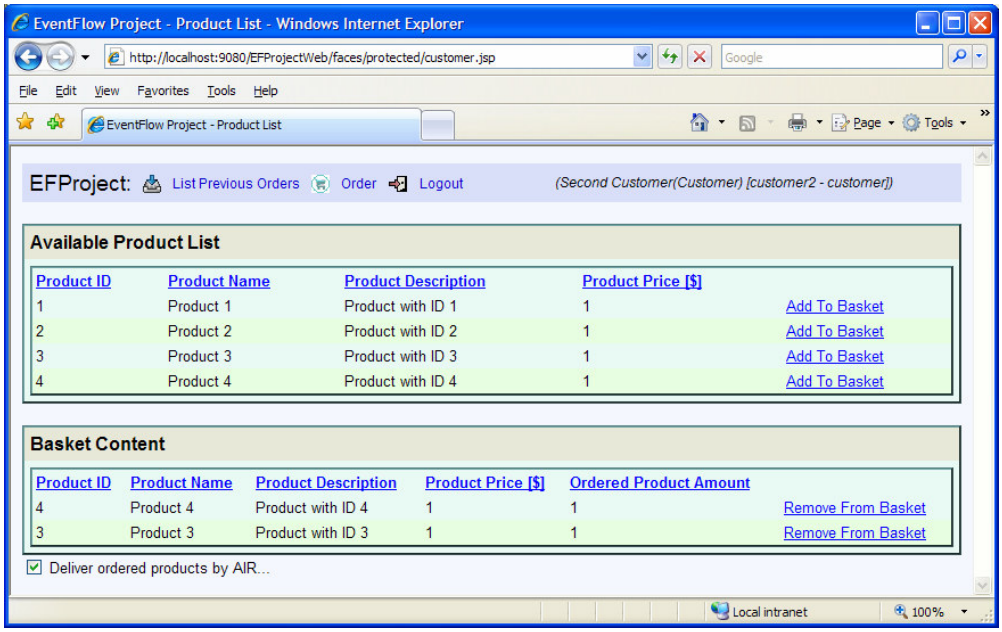


Figure 5.3 Order given by *customer2*

Also the customer can check his previous orders by the page shown in **Figure 5.4** and **Figure 5.5.** This page shows the current state of the order and the other details related with that order.

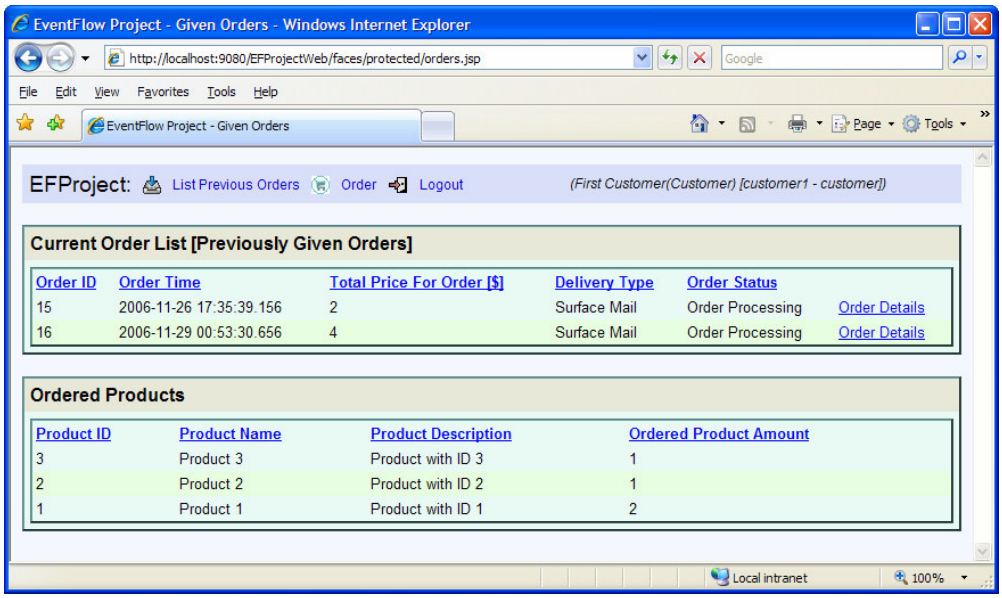


Figure 5.4 Previous Orders List page

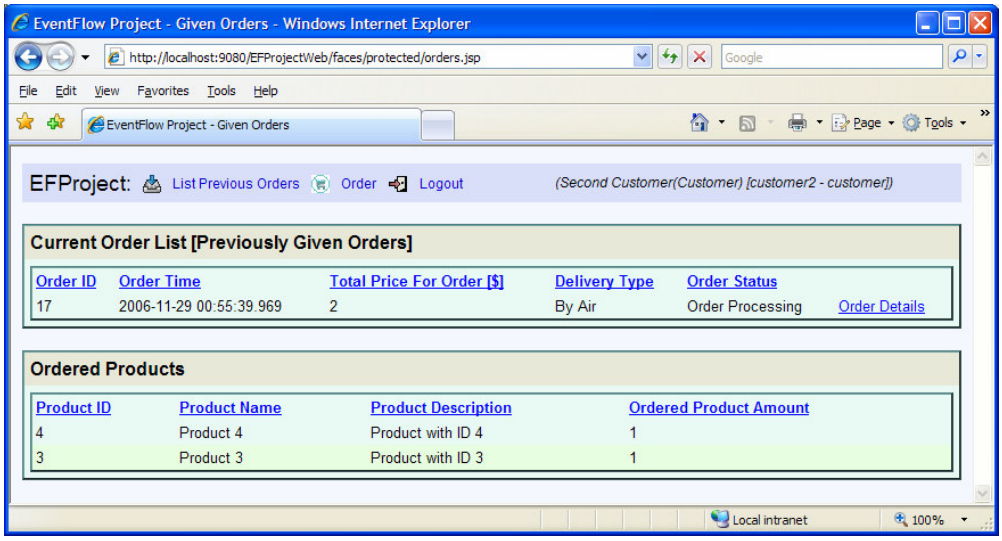


Figure 5.5 Details of the Order given by *customer2*

After starting the workflow, activities assigned to the system agent (i.e. eventFlow) will do the automated tasks. These are namely *initial_activity* and *order_collection* activities. After executing the *order_collection* activity by the system agent, the activity *order_processing* will be shown at the worklists of the qualified agents a*gent1* and *agent6* respectively, and these agents will see that activity in their inbox when they are query the system by logging into system or refreshing the inbox page. The first agent checking out the activity will be assigned to that activity by the system. Here the agent *agent1* will do that activity.

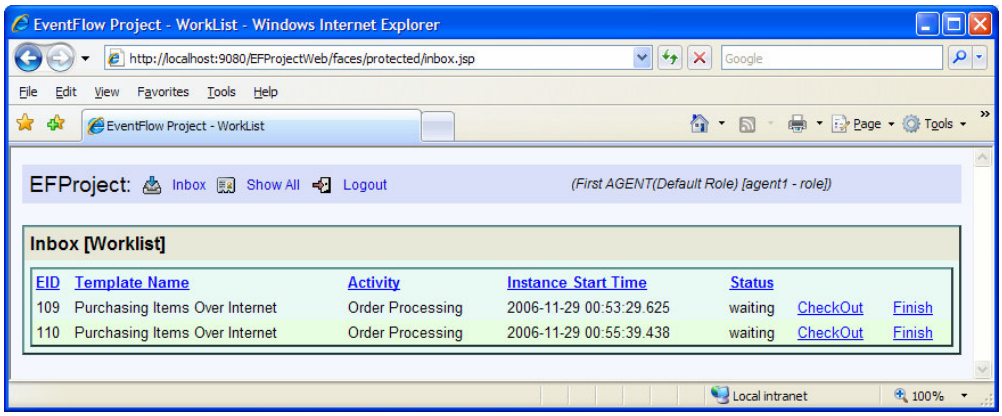


Figure 5.6 Worklist of agent Agent1

Figure 5.6 shows the worklist of the agent *agent1*, and there are more than one instance of an "Purchasing Items Over Internet". This shows that the concurrent instances of any workflow can run on the system. To do the corresponding operation with the activity *agent1* will use the "CheckOut" link. This link will redirect the agent to the coreesponding operation page in **Figure 5.7.**

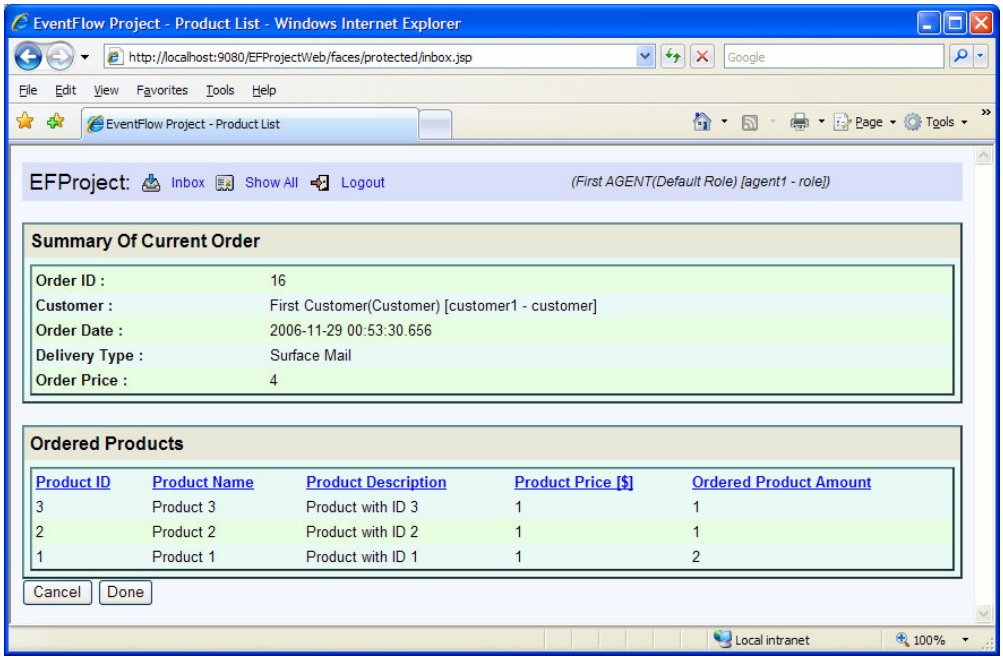


Figure 5.7 Activity page containing details of the related order

agent1 will finish that activity by clicking the button "Done". This will end that activity and the next activities will be scheduled by the workflow manager. If the button "Cancel" is clicked by the agent, the acivity will return to the *waiting* state again.

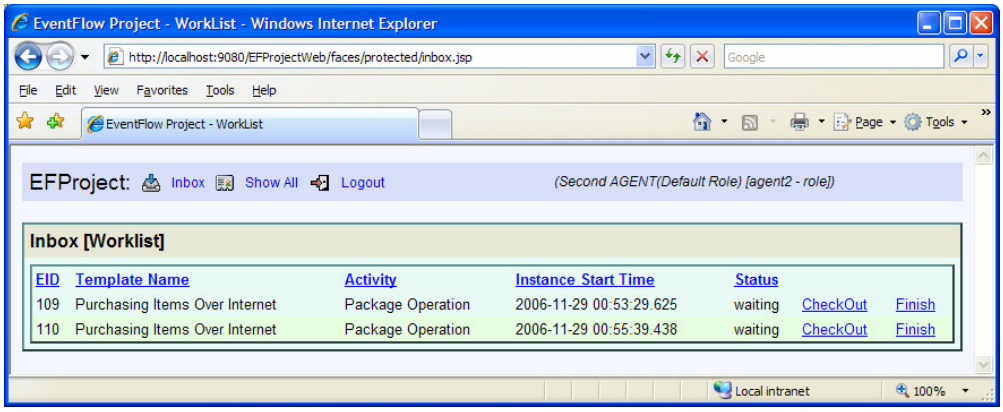


Figure 5.8 Worklist of an agent2

After the activity "Order Processing" is finished by an agent *agent1*, both activities "Package Operation" and "Billing" are put into the waiting state for the corresponding qualified agents. As stated before, activity "Billing" is an automated task so it is executed by the eventFlow manager as soon as it gets that activity from its execution queue. The other activity "Packaging" is waiting in the worklist of the agent *agent2.* When *agent2* finishes packaging, the next activity will be scheduled by the manager.

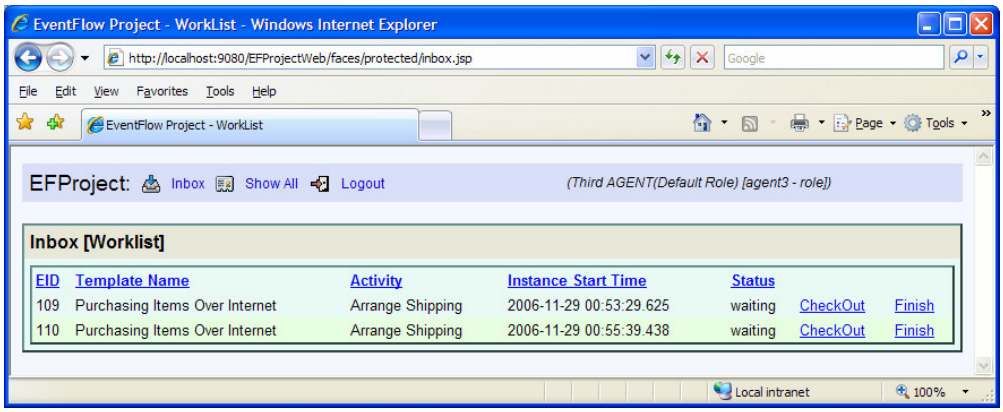


Figure 5.9 Worklist of an agent3

The next activity is "Arrange Shipping". This activity is assigned to the agent *agent3* by the manager. After that activity the manager will select one of the branches according to the evaluation of the conditions. Because the *customer1* did

not select the “Delivery by Air” option, the next scheduled activity will be the activity “Surface Mail”, and it is *agent5*’s responsibility to do that activity, because this agent is the only agent qualified to do that activity. For *customer2*’s order, “By Air” activity will be assigned to the qualified agent *agent4*. By using **EID** values displayed in the worklists, the correctness of the assignment can be checked.

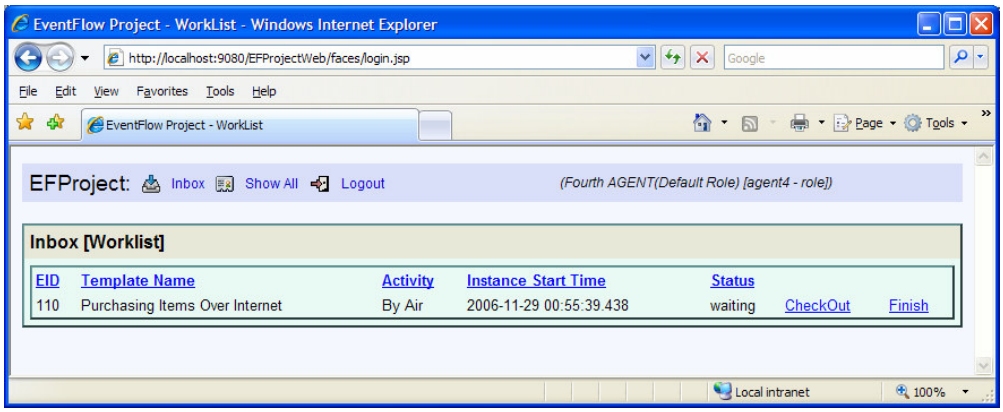


Figure 5.10 Worklist of an agent4

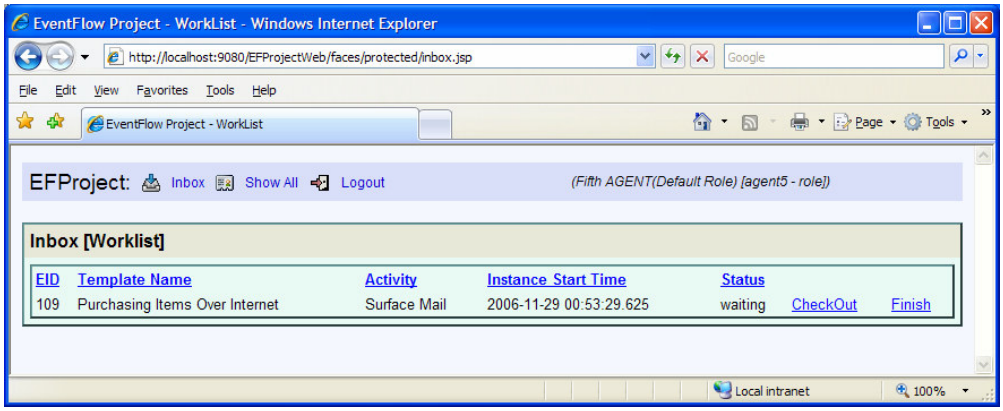


Figure 5.11 Worklist of an agent5

After finishing “Surface Mail”, the activity “Archieve Operation” will be scheduled to be done by the qualified agents. Here, the role *role* is qualified to that activity. Thus, the activity will be shown each agent that has the role *role*. One of the agents having that role value can checkout and do the corresponding operation

to complete that task. When this activity is finished by any of the qualified agent, the workflow instance will be completed. **Figure 5.12** shows the worklist of an *agent6* having *role* as his role value.

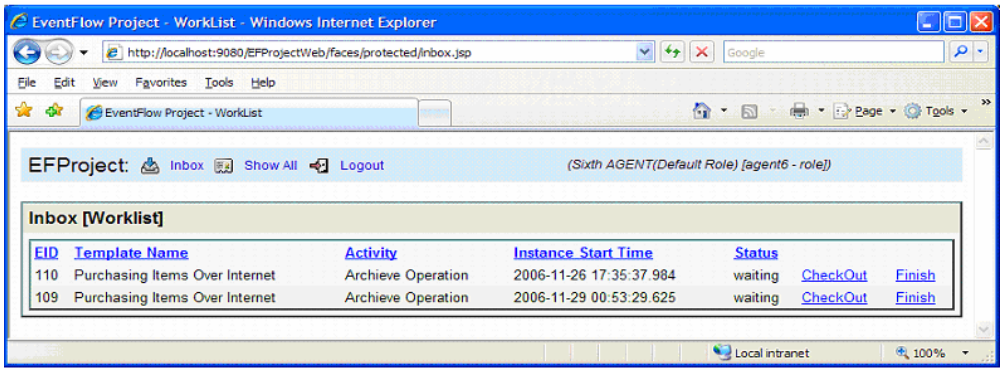


Figure 5.12 Worklist of an agent6

CHAPTER 6

CONCLUSION

This thesis demonstrates the use of the event calculus to describe the specification and execution of activities in a workflow. The main axioms of the event calculus are integrated with a set of activity execution dependency rules and a set of agent assignment rules for the formalization of workflow systems. It is shown that major types of activity routings in a workflow (namely sequential, concurrent and conditional) can be expressed in a declarative way. It is also illustrated that agent assignments and concurrent workflow instances can be modeled within the framework of the event calculus. An implemented architecture of a workflow management system is presented as a proof of concept application of this logic-based approach. For a quick simulation of a workflow, the user needs merely to specify the activities in the control flow graph and the external events and their possible effects on the underlying database. The rest of the workflow management is done by the rules presented in this thesis.

The proposed logic-based approach can be used as a quick tool in prototyping applications and/or simulations of workflows. Due to its additional temporal dimension, it provides facilities for querying the history of all activities, thus providing opportunities to analyze the execution of the workflows. It can be used as an easy tool to simulate and verify the execution of a prototype workflow system. The workflow might be executed with different number of agents and assignments. The behavior of the workflow can be analyzed by querying the history of events and the snapshots of the workflow state at different times.

A graphical tool (EventFlow Editor) is developed for the designed architecture to provide the user with the facility of drawing the control-flow graph of the workflow. And this application is also used to map the generated graph into a set of atomic formulas automatically. A sample application is implemented by using the developed architecture, to show the usability of the framework and the correctness of the axioms stated for the implementation of workflow systems by using the EC. Also main functionalities such as running concurrent instances of the same workflow specification are provided by the system.

In this thesis the workflows that do not terminate successfully are not considered. Some of the activities can abort and therefore they need to be compensated or some kind of exception handling mechanism must be applied. As a future work, the set of execution dependency rules can be extended to cover such control flows. These extensions do not require substantial changes to the proposed architecture. Broadly speaking, what needs to be done is to define additional scheduling rules to the set of axioms AxS, so that when an activity does not end, the execution is diverted to another route of activities, which will be used either to abort the workflow or compensate the failed activity.

REFERENCES

1. Adam, N.R., Atluri, V. & Huang, W.K. (1998). Modeling and analysis of workflows using Petri Nets. In: *Journal of Intelligent Information Systems 10 (2).* (pp. 131-158)

2. Apache Derby, An Open Source Relational Database Implementation in Java. (Last Visited:2006) http://db.apache.org/derby/

3. Attie, P.C., Singh, M.P., Sheth, A. & Rusinkiewicz M. (1993) Specifying and Enforcing Intertask Dependencies. In: *Proceedings of the 19th Conference on Very Large Databases, (Los Altos CA), Dublin.*

4. Baral, C., Lobo, J. & Trajcevski, G. (2001). Formalizing and reasoning about the requirements specifications of workflow systems. *International Journal of Intelligent Information Systems* 10 (4) (pp. 483–507).

5. Bettini, C., Wang, X., & Jajodia, S. (2002) Temporal reasoning in workflow systems. *Distributed and Parallel Databases* 11 (3) (pp. 269–306).

6. Davulcu, H., Kifer, M., Ramakrishnan, C.R. & Ramakrishnan, I.V. (1998) Logic based modeling and analysis of workflows. In: *Proceedings of ACM Symposium on Principles of Database Systems*, ACM Press, Seattle, Washington. (pp. 25–33).

7. Dayal, U., Hsu, M. & Ladin, R. (1990). Organizing long running activities with triggers and transactions. In: *Proceedings of the International Conference on Management of Data, Atlantic City, NJ.*

8. Ebbers, M., Kasselman, A., Mitchell, A. & Orr, B. (1999). Image And Workflow Library: MQSeries Workflow Concepts, Installation and Administration. *Redbook. SG24-5375-00.* IBM. USA

9. Enhydra Shark Open Source Workflow (Last Visited:2006) http://www.enhydra.org/workflow/shark/index.html

10. Fernandez, A.A., Williams, M.H. & Paton, N.W. (1997). A logic-based integration of active and deductive databases. *New Generation Computing* 15 (pp. 205–244).

11. FileNet. (1997). Visual WorkFlo Design Guide. FileNet Corporation, Costa Mesa, CA, USA.

12. FileNet. (1999) Panagon Visual WorkFlo Architecture. FileNet Corporation, Costa Mesa, CA, USA.

13. Georgakopoulos, D., Hornick, M. & Sheth, A. (1995). An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. In: *Distributed and Parallel Database,* 3. Kluwer Academic Publishers, Boston. (pp. 119-153).

14. Hollingsworth, D. (1995). The workflow reference model. *Workflow Management Coalition*. (Last Visited: 2006) http://www.wfmc.org/standards/docs/tc003v11.pdf

15. Hull, R., Llirbat, F., Simon, E., Su, J., Dong, G., Kumar, B. & Zhou, G. (1999). Declarative workflows that support easy modification and dynamic browsing. In: Georgakopoulos, G., Prinz, W. & Wolf, A.L. (Eds.) *Proceedings of the International Joint on Work Activities Coordination and Collabaration (WACC'99)San Francisco* (pp. 69-78)

16. InterProlog, A Java front-end and enhancement for Prolog, Declarative. (Last Visited: 2006) http://www.declarativa.com/interprolog/

17. JGraph, An Open Source Java Graph Library. (Last Visited: 2006) http://www.jgraph.com/

18. Keen, M., Cavell, J., Hill, S., Kee, C.K., Neave, W., Rumph, B. & Tran, H. (2004). BPEL4WS Business Process with WebSphere Business Integration: Understanding, Modeling, Migrating. *Redbook. SG24-6381-00.* IBM. USA.

19. Kesim-Çiçekli, N. & Çiçekli, I. (2006). Formalizing the specification and execution of workflows using the Event Claculus. In: *Information Sciences, Volume 176, Issue 15.* (pp. 2227-2267)

20. Kesim-Çiçekli, N. & Yıldırım Y. (2000). Formalizing workflows using the event calculus. In: Ibrahim, M., Kung, J. & Revell, N. (Eds.), *The 11th International Workshop on Database and Expert Systems Applications* (DEXA_00), LNCS, vol. 1873, Springer-Verlag, Berlin. (pp. 222–231).

21. Kowalski, R.A. & Sergot, M.J. (1986). A logic-based calculus of events. *New Generation Computing*, *4,* 67–95.

22. Kowalski , R.A. (1992). Database updates in the event calculus. *Journal of Logic Programming 12, (1-2),* 121–146.

23. Nielsen, S.P., Serpola, S., Collins, F., Morrison, D. & Strobl, R. (1999). Lotus Domino Workflow 5.0: A Developer's Handbook. *Redbook SG24-5331-01*. IBM. USA.

24. NovaManage Integrated document management and workflow for highly regulated industries. (Last Visited: 2006) http://www.cimage.com/products/novamanage/index.htm

25. Muth, P., Wodtke, D., Weissenfels, J., Weikum, G., & Kotz Dittrich, A. (1998). Enterprise-wide Workflow Management based on State and Activity Charts. In Dogac, A., Kalinichenko, L., Ozsu, M.T., & Sheth A. (Eds.), *Workflow Management Systems and Interoperability*, (pp. 281-303). NATO Advanced Study Institute, Springer-Verlag.

26. Perez, C.E. (Last Updated : 2006) Open Source Workflow Engines written in Java. http://www.manageability.org/blog/stuff/workflow_in_java

27. SAP. (1997). WF SAP Business Workflow. *SAP AG.* Walldorf, Germany.

28. Senkul, P., Kifer, M. & Toroslu, İ.H. (2002) A logical framework for scheduling workflows under resource allocation constraints. In: *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB 2002)*. Hong Kong, China. (pp. 694–705).

29. Software-Ley GmbH. (1999). COSA 3.0 User Manuel. Pullheim, Germany.

30. XStream, An Open Source Object Serialization Library. (Last Visited: 2006) http://xstream.codehaus.org/

31. XSB, An Open Source Prolog Interpreter. (Last Visited: 2006) http://xsb.sourceforge.net/

32. Van Der Aalst, W.M.P., Ter Hofstede, A.H.M., Kiepuszewski, B. & Barros, A.P. (2000). Workflow patterns. In: Etzion O. & Scheuermann, P. (Eds.). *Proc. CoopIS 2000, LNCS 1901*. Springer. (Last Visited: 2006) http://is.tm.tue.nl/staff/wvdaalst/publications/p159.pdf

33. Van Der Aalst, W.M.P. (1998). The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers 1 (8).* (Last Visited: 2006) http://is.tm.tue.nl/staff/wvdaalst/publications/p53.pdf

34. Van Der Aalst, W.M.P. & Ter Hofstede, A.H.M. (2003). YAWL: Yet Another Workflow Language. *QUT Technical report, FIT-TR-2003-04*. Queensland University of Technology, Brisbane. (Last Visited: 2006) http://is.tm.tue.nl/staff/wvdaalst/publications/p198.pdf

35. Van Der Aalst, W.M.P., Aldred, L., Dumas, M. & Ter Hofstede, A.H.M. (2004). Design and Implementation of the YAWL System. In : Persson, A.

& Stirna, J. (Eds.), *Advanced Information Systems Engineering, Proceedings of the 16th International Conference on Advanced Information Systems Engineering (CAiSE'04), volume 3084 of Lecture Notes in Computer Science.* Springer-Verlag, Berlin. (pp 142-159) (Last Visited: 2006) http://is.tm.tue.nl/staff/wvdaalst/publications/p224.pdf

36. Van Der Aalst, W.M.P. (1996). Three Good Reasons for Using a Petri-net-based Workflow Management System. *Proceedings of the International Working Conference on Information and Process Integration in Enterprises (IPIC'96)* (Last Visited: 2006) http://is.tm.tue.nl/staff/wvdaalst/publications/p52.pdf

37. WfMOpen : Open Source Workflow Project (Last visited: 2006) http://wfmopen.sourceforge.net/

38. YAWL: Yet Another Workflow Language (Last visited: 2006) http://www.yawl.fit.qut.edu.au/

APPENDICES

APPENDIX A. ORIGINAL PREDICATES

In this section, the predicates (axioms) used for the implementation of EventFlow are given in [19] are listed.

Axioms for Happens:

happens(start(Act, Ag, W), T) ← *happens(assign(Ag, Act, W), T).* (AxH1)

happens(release(Ag, Act, W), T) ← *happens(end(Act, Ag, W), T).* (AxH2)

happens(end(Act, Ag, W), T) ←
happens(start(Act, Ag, W), Ts), fixed_activity(Act),
qualified(Ag, Act, Td), T = Ts + Td. (AxH3)

happens(end(Act, Ag, W), T) ←
happens(start(Act, Ag, W), Ts), varying_activity(Act),
end_event(Act, ExtEvent), happens(ExtEvent, Te),
qualified(Ag, Act, Td), Tf = Ts + Td, max([Te, Tf], T). (AxH4)

happens(assign(Ag, Act, W), T) ←
happens(release(Ag, _, _), T),
holds_at(waiting(Act, Ag, W, T1), T), holds_at(idle(Ag), T),
not waiting_longer(Act, Ag, T1, T), not better_agent(Ag, Act, T). (AxH5)

happens(assign(Ag, Act, W), T) ←
initiates(_, waiting(Act, Ag, W, T)), holds_at(waiting(Act, Ag, W, T), T),
holds_at(idle(Ag), T), not better_agent(Ag, Act, T). (AxH6)

happens(free_agent(Ag), 0) ← *agent(Ag).* (AxH7)

waiting_longer(Act, Ag, T1, T) ←
holds_at(waiting(Act2, Ag, W, T2), T),
Act ≠ Act2, T2 < T1.

better_agent(Ag1, Act, T) ←

qualified(Ag1, Act, C1), qualified(Ag2, Act, C2),

C2 < C1, holds_at(idle(Ag2), T).

Axioms for Scheduling:

follows(Act1, Act2, W, T) ←

sequential(Act1, Act2), happens(end(Act1, _, W, T)). (AxS1)

follows(Act1, Act2, W, T) ←

and_split(Act1, ActList), happens(end(Act1, _, W, T)),

member(Act2, ActList). (AxS2)

follows(Act1, Act2, W, T) ←

and_join(ActList, Act2),

findActEndTimePairs(ActList, W, ActEndTimePairs),

actWithMaxEndTime(ActEndTimePairs, Act1, T). (AxS3)

follows(Act1, Act2, W, T) ←

xor_split(Act1, ActCondPairs), happens(end(Act1, _, W), T1),

member((Act2, Cond2), ActCondPairs), initiates(Ev, Cond2),

happens(Ev, T2), max([T1, T2], T), holds_at(Cond2, T). (AxS4)

follows(Act1, Act2, W, T) ←

xor_join(ActList, Act2),

findOneActEndTimePair(ActList, W, Act1, T). (AxS5)

follows(Act1, InitAct, W, T) ←

serial(Act1, B), happens(end(Act1, _, W), T),

initial(B, InitAct), setIterationNo(InitAct, 1). (AxS6)

follows(FnlAct, Act2, W, T) ←

serial(B, Act2, Cond), final(B, FnlAct),

happens(end(FnlAct, _, W), T), not holds_at(Cond, T). (AxS7)

follows(FnlAct, InitAct, W, T) ←

initial(B, InitAct), final(B, FnlAct), serial(B, _, Cond),

happens(end(FnlAct, _, W), T), holds_at(Cond, T),

getIterationNo(FnlAct, I), J = I + 1, setIterationNo(InitAct, J). (AxS8)

setIterationNo(Act, N) ←*Act = act(_, b(_, _, IterationNo)), IterationNo = N.*

getIterationNo(Act, IterationNo) ←*Act = act(_, b(_, _, IterationNo)).*

findOneActEndTimePair(ActList, W, Act, EndTime) ←

member(Act, ActList),

happens(end(end(Act, _, W)), EndTime).

findActEndTimePairs(ActList, W, ActTimePairs) ←

findall((Act, EndTime),

(member(Act, ActList),happens(end(Act, _, W), EndTime)),

ActTimePairs),

length(ActList, ActListLen), length(ActTimePairs, ActTimePairsLen),

ActListLen = ActTimePairsLen.

actWithMaxEndTime([FirstPair|ActEndTimePairs], Act, EndTime) ←

actWithMaxEndTime(ActEndTimePairs, FirstPair, Act, EndTime).

actWithMaxEndTime([], (Act, EndTime), Act, EndTime).

actWithMaxEndTime([CurrPair|Rest], CurrMax, Act, EndTime) ←

CurrPair = (Act1, T1), CurrMax = (Act2, T2), T1 > T2,

actWithMaxEndTime(Rest, CurrPair, Act, EndTime).

actWithMaxEndTime([CurrPair|Rest], CurrMax, Act, EndTime) ←

CurrPair = (Act1, T1), CurrMax = (Act2, T2), T1 ≤ T2,

actWithMaxEndTime(Rest, CurrMax, Act, EndTime).

Axioms for Initiates/Terminates:

initiates(start(Act, Ag, W), active(Act, Ag, W)). (AxIT1)

initiates(end(Act, Ag, W), completed(Act, Ag, W)). (AxIT2)

terminates(end(Act, Ag, W), active(Act, Ag, W)). (AxIT3)

terminates(assign(Ag, _, _) , idle(Ag)). (AxIT4)

initiates(assign(Ag, Act, W), assigned(Ag, Act, W)). (AxIT5)

initiates(release(Ag, _, _), idle(Ag)). (AxIT6)

terminates(release(Ag, Act, W), assigned(Ag, Act, W)). (AxIT7)

initiates(release(Ag1, Act1, W), waiting(Act2, Ag2, W, T)) ←

follows(Act1, Act2, W, T),

qualified(Ag2, Act2, _). (AxIT8)

terminates(assign(_, Act, W), waiting(Act, _, W, _)). (AxIT9)

initiates(free_agent(Ag), idle(Ag)). (AxIT10)

initiates(Ev, waiting(Act, Ag, W, T)) ←

initial_activity(Act),
starts(Ev, W), happens(Ev, T),
setEID(Act, W), qualified(Ag, Act, _). (AxIT11)

APPENDIX B. CREATE SCRIPTS FOR TABLES

These scripts are prepared for Apache Derby database. It is possible to use these scripts for other databases by changing data type definitions and modifying keywords for the database that is planned to be used.

Scripts For System Tables :

```
CREATE TABLE RoleDetails
        (role           VARCHAR(30) NOT NULL,
         description    VARCHAR(50) NOT NULL,
                PRIMARY KEY (role));

CREATE TABLE AgentDetails
        (agent          VARCHAR(30) NOT NULL,
         firstName      VARCHAR(50) NOT NULL,
         lastName       VARCHAR(50) NOT NULL,
         password       VARCHAR(10) NOT NULL,
         role           VARCHAR(30)  NOT NULL,
                PRIMARY KEY (agent),
                FOREIGN KEY (role) REFRENCES RoleDetails (role));

CREATE TABLE ActivityDetails
        (activityId     INTEGER  NOT NULL GENERATED ALWAYS AS IDENTITY,
         activity       VARCHAR(50) NOT NULL,
         activityType   VARCHAR(1)    NOT NULL CHECK (activityType in ('A','U')),
         operation      VARCHAR(20) NOT NULL,
         description    VARCHAR(150) NOT NULL,
                PRIMARY KEY (activityId));
```

```
CREATE TABLE RunningEventFlows
        (executionId      INTEGER  NOT NULL GENERATED ALWAYS AS IDENTITY,
         templateId       INTEGER  NOT NULL,
         eventFlow        VARCHAR(50) NOT NULL,
         startTime        TIMESTAMP,
         endTime          TIMESTAMP,
         starter          VARCHAR(30),
         replayScript     CLOB(4M),
         dataContainer    CLOB(4M),
         conditionMap     CLOB(4M),
                PRIMARY KEY (executionId),
                FOREIGN KEY (starter)          REFERENCES AgentDetails (agent),
                FOREIGN KEY (templateId)       REFERENCES EventFlowTemplate
(templateId));

CREATE TABLE EventLog
        (eventId INTEGER  NOT NULL GENERATED ALWAYS AS IDENTITY,
         executionId      INTEGER       NOT NULL,
         event            VARCHAR(15)   NOT NULL,
         activityId       INTEGER       NOT NULL,
         iteration        INTEGER       NOT NULL DEFAULT 0,
         time             TIMESTAMP     NOT NULL,
         agent            VARCHAR(30),
         role             VARCHAR(30),
                PRIMARY KEY (eventId),
                FOREIGN KEY (executionId) REFERENCES RunningEventFlows
(executionId) ON DELETE CASCADE,
                FOREIGN KEY (activityId) REFERENCES  ActivityDetails (activityId) ON
DELETE CASCADE);

CREATE TABLE EventFlowTemplate
        (templateId       INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY,
         templateName     VARCHAR(50)  NOT NULL,
         templateDesc     VARCHAR(200),
         templateGraph    BLOB(4M),
         convertedGraph CLOB(4M),
         templatePNG      BLOB(4M),
         dataContainer    CLOB(4M),
```

```
    conditionMap    CLOB(4M),
        PRIMARY KEY (templateId));
```

Scripts For Sample Application Tables :

```
CREATE TABLE Products
    (productId          INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY,
     productName            VARCHAR(50)  NOT NULL,
     productDescription     VARCHAR(250) NOT NULL,
     productPrice           INTEGER,
        PRIMARY KEY (productId));

CREATE TABLE Orders
    (orderId          INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY,
     executingEFID    INTEGER NOT NULL,
     orderDate        TIMESTAMP NOT NULL,
     customer         VARCHAR(30),
     deliveryType     VARCHAR(20),
     status           VARCHAR(100),
     price            INTEGER,
        PRIMARY KEY (orderId),
        FOREIGN KEY (executingEFID)  REFERENCES RunningEventFlows
(executionId),
        FOREIGN KEY (customer)       REFERENCES AgentDetails (agent));

CREATE TABLE OrderedItems
    (orderId          INTEGER      NOT NULL,
     productId        INTEGER      NOT NULL,
     amount           INTEGER NOT NULL,
        PRIMARY KEY (orderId,productId),
        FOREIGN KEY (orderId) REFERENCES Orders (orderId),
        FOREIGN KEY (productId) REFERENCES Products (productId));
```

APPENDIX C. XSB IMPLEMENTATION OF FORMALIZATION PREDICATES

```
/*************************************************************/
/*                                                           */
/*                    EventFlow Formalization                */
/*              Formalizing the specification and execution of */
/*                 workflows using the Event Calculus        */
/*                                                           */
/*************************************************************/
/*************************************************************/
/*                                                           */
/*                      Workflow Management                  */
/*                                                           */
/*************************************************************/
/*********************     Workflow State    *****************/
/*
 *                Execution states of activities
 *       active(Act,Ag,W)    -> Act is being executed by Ag in W
 *                 (initiated by event -> start(Act,Ag,W))
 *       completed(Act,Ag,W) -> Act is completed by Ag in W
 *                 (initiated by event -> end(Act,Ag,W))
 *       waiting(Act,Ag,W,T) -> Act is in the worklist of Ag in W by timestamp T
 *                 (initiated by event -> start(Ag0,Act0,W))
 *
 *                      States of Agents
 *       idle(Ag)          -> Ag is idle
 *                 (initiated by event -> release(Ag,Act,W))
 *       assigned(Act,Ag,W) -> Ag is carrying out Act in W
 *                 (initiated by event -> assign(Ag,Act,W))
 */

/*********************  AxIT  **********************/
```

```
/*
 *          An activity becomes active in a workflow instance
 * when its starting event is recorded in the database. An event
 * recording the end of an activity sets up a completed state for
 * that activity, terminating its active state.
 */

        initiates(  start(Act, Ag, W) , active(Act, Ag, W)    ).          %AxIT1
        initiates(  end(Act, Ag, W)   , completed(Act, Ag, W) ).          %AxIT2
        terminates( end(Act, Ag, W)   , active(Act,Ag,W)      ).          %AxIT3

/*
 *          When an activity starts being executed by an agent, the
 * agent is not idle any more and it is assigned to that activity
 * until it finishes the activity. When the activity is finished,
 * the agent is released and it is ready to execute the next activity.
 */

        terminates( assign(Ag, _, _)   , idle(Ag)             ).          %AxIT4
        initiates(  assign(Ag, Act, W) , assigned(Ag, Act, W) ).          %AxIT5

/*
 *          When an agent finishes its task and it is released, it
 * becomes idle. If the worklist of the agent is empty, the agent remains
 * in the idle state. If there are one or more activities waiting
 * for that agent in the agent's worklist, the agent is assigned
 * to the next activity in its worklist.
 */

        initiates(  release(Ag, _, _)   , idle(Ag)             ).         %AxIT6
        terminates( release(Ag, Act, W) , assigned(Ag, Act, W) ).         %AxIT7

/*
 *          waiting property is used to represent the state of an activity
 * and to represent the worklists of agents.
 */
        initiates( release(Ag1, Act1, W), waiting(Act2, Ag2, W, T)) :-
                follows( Act1, Act2, W, T),
```

```
                qualified(Ag2, Act2).                                   %AxIT8
        terminates( assign(_, Act, W), waiting(Act, _, W, _)).          %AxIT9
        initiates( Ev, waiting( Act, Ag, W, T)) :-
                initial_activity(Act),
                starts(Ev, W),
                happened(Ev, T),
                setEID(Act, W),
                qualified(Ag, Act).                                     %AxIT10

/*********************************************************************/
        /*Selection done by Ag for given Act for instance W.*/

        initiates(select(Ag, Act, W), selected(Act, W)).
        terminates(end(Act, _, W),    selected(Act, W)).

/*********************************************************************/
/***************  Workflow Execution **********************/

/*      Rules for triggering Events             */
/*
 *      The execution of an activity can start only when an agent
 * is assigned to that activity. As soon as the agent is assigned,
 * the starting event of the activity is generated.
 */
        happens( start(Act, Ag, W), T) :-
                happened( assign(Ag, Act, W), T).                       %AxH1
/*
 *      Releasing Agents
 *      When an activity is completed, the ending event of the activity
 * is recorded and the agent that completed the activity is released.
 *
 *      An agent is released when the task is finished. [This rule can
 * be extended to model the case where the task is suspended and agent
 * can look at other tasks in the meanwhile. ]
 */
happens( release(Ag, Act, W), T) :-
                happened( end(Act, Ag, W), T).                          %AxH2
```

```
/*********   Execution Dependencies of Activities   ********/
/******       Sequential Activities      *******/

        follows(Act1, Act2, W, T) :-
                sequential(Act1, Act2),
                happened( end(Act1, _, W), T).                          %AxS1

/*****    AND-Split and AND-Join     *****/

        follows(Act1, Act2, W, T) :-
                and_split(Act1, ActList),
                happened( end(Act1, _, W), T),
                member(Act2, ActList).                                  %AxS2

        follows(Act1, Act2, W, T) :-
                and_join(ActList, Act2),
                findActEndTimePairs(ActList, W, ActEndTimePairs),
                actWithMaxEndTime(ActEndTimePairs, Act1, T).            %AxS3

        findActEndTimePairs(ActList, W, ActTimePairs) :-
                findall( (Act, EndTime),
               (member(Act, ActList), happened( end(Act, _, W), EndTime) ),
                        ActTimePairs),
                length(ActList, ActListLen),
                length(ActTimePairs, ActTimePairsLen),
                ActListLen = ActTimePairsLen.

        actWithMaxEndTime([FirstPair|ActEndTimePairs], Act, EndTime) :-
                actWithMaxEndTime(ActEndTimePairs, FirstPair, Act, EndTime).
        actWithMaxEndTime([], (Act, EndTime), Act, EndTime).
        actWithMaxEndTime([CurrPair|Rest], CurrMax, Act, EndTime) :-
                CurrPair = (Act1, T1),
                CurrMax  = (Act2, T2),
                T1 @> T2,
                actWithMaxEndTime(Rest, CurrPair, Act, EndTime).
        actWithMaxEndTime([CurrPair|Rest], CurrMax, Act, EndTime) :-
                CurrPair = (Act1, T1),
                CurrMax  = (Act2, T2),
```

```
                T1 @=< T2,
                actWithMaxEndTime(Rest, CurrMax, Act, EndTime).

/*****    XOR-Split and XOR-Join    *****/

        follows(Act1, Act2, W, T) :-
                xor_split(Act1, ActCondPairs),
                happened( end(Act1, _, W), T1),
                member( (Act2, Cond2), ActCondPairs),
                initiates(Ev, Cond2),
                happened(Ev, T2),
                max([T1,T2], T),
                holds_at(Cond2, T).                                    %AxS4

        follows(Act1, Act2, W, T) :-
                xor_join(ActList, Act2),
                findOneActEndTimePair(ActList, W, Act1, T).            %AxS5

        findOneActEndTimePair(ActList, W, Act, EndTime) :-
                member(Act, ActList),
                happened( end(Act, _, W), EndTime).
/*************************************************************************/
/*
 * HOLDS_AT - System states holding at a certain time EC Axioms
 */
        holds_at(P, T) :-
                happened(E, T1),
                T1 @=< T,
                initiates(E, P),
                not broken(P, T1, T).

        broken(P, T1, T2) :-
                happened(E, T), terminates(E, P),
                T1 @=< T, T  @=< T2.

        holds_for(P, T1, T2) :-
                happened(E1, T1),
                initiates(E1, P),
```

```
                happened(E2, T2),
                terminates(E2, P),
                not broken(P, T1, T2).

        holdsNow(Property) :-
                clock(Now),
                holds_at(Property, Now).

/*  To start workflow instances */
starts(Ev, Wno) :-
  ext_event(Ev),
  Ev = runWF(CID),
  Wno = CID.
/*********************  Dummy predicates  *****************/
and_split(dummy,_).
and_join(_,dummy).
xor_split(dummy,_).
xor_join(_,dummy).
sequantial(dummy,dummy).
happened(dummy,dummy).

/******************** Utility functions ******************/
min([H|L],M) :- min(L,H,M).
min([],M,M) :- !.
min([H|L],CM,M) :- H @< CM, !, min(L,H,M).
min([_|L],CM,M) :- min(L,CM,M).

max([H|L],M) :- max(L,H,M).
max([],M,M) :- !.
max([H|L],CM,M) :- H @> CM, !, max(L,H,M).
max([_|L],CM,M) :- max(L,CM,M).
```

APPENDIX D. BRIEF USER MANUAL FOR EVENTFLOW EDITOR

The EventFlow Editor is a graphical user interface to build a workflow model, save the designed workflow to disk or system, and convert it to the defined first order predicates. Also, it provides some administrative facilities to user such as defining new activity. The first time the EventFlow Editor is started, you will be presented with a blank canvas, with the instructions in the Status Bar asking you to open or create specification to begin. **Figure D.1** shows the EventFlow Editor.

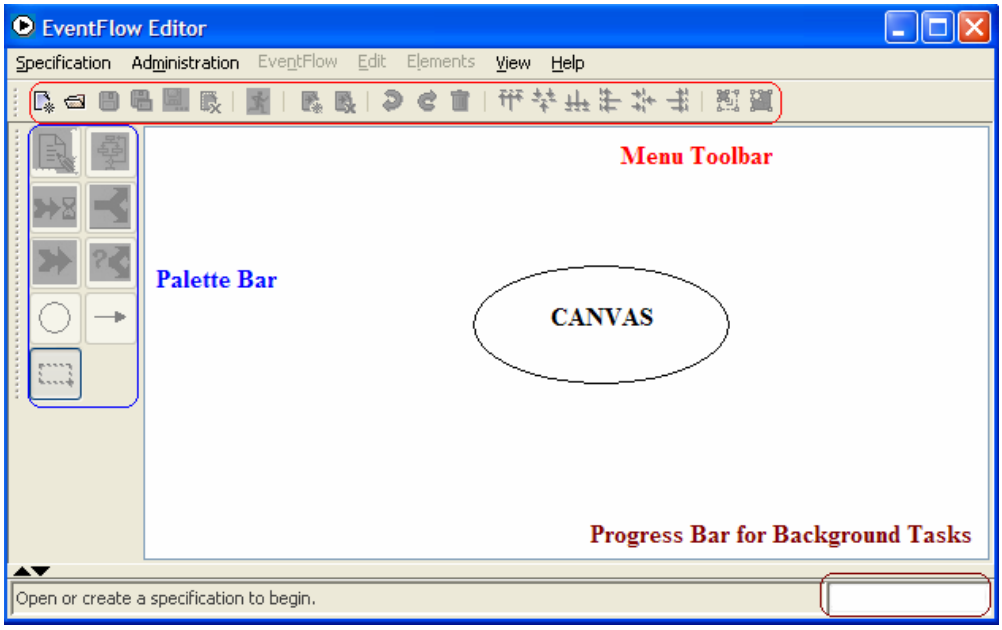


Figure D.1 The EventFlow Editor

Elements within the EventFlow Editor

Before giving the details of the design of a workflow by the editor, the elements within the editor are described briefly.

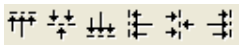
Menu Toolbar

The Menu Toolbar contains five groups of buttons to assist the user in maintaining his/her EventFlow design. The menu can be repositioned by dragging the left-hand anchor bar.

- ***Specification Maintenance Buttons:***  This group of buttons provides the user the standard options to create, save, open and close EventFlow specification.

- ***EventFlow Maintenance Buttons:***  The designed workflow diagrams are captured within EventFlow specification. Using this group of buttons, one can create a new specification, remove an existing specification or run an instance of specification on developed framework.

- ***Edit Options Buttons:***  This group of buttons provides the standard Undo and Redo options as well as the option to delete the currently selected objects.

- ***Alignment Options Buttons:***  These buttons can be used to assist with the alignment of objects within the currently opened specification, when multiple objects have been selected.

- ***Object Size Buttons:***  To increase or decrease the size of an object within the currently opened workflow model, these buttons can be used.

Palette Bar

The Palette Bar contains nine selector buttons that assist with creation, selection and positioning of objects within the specification. This menu can be repositioned

by dragging the left-hand anchor bar. It is also accessible by right-clicking anyempty place on a workflow model. Once an element is selected, it is possible to drop objects in the canvas by left-clicking the mouse button.

- ***Activity:***  Use this button to create an activity representing a single task to be performed by the agent(s).

- ***Activity Block:***  To create an activity block, a container for another EventFlow specification, this button can be used.

- ***AND-split:***  This button can be used to add an AND-split node to the EventFlow specification currently opened. The symbol on the button indicates that all activities connected to that node will be activated without chacking any condition.

- ***AND-join:***  The button indicates the synchronization point for the branches coming out from the AND-aplit node. The clock means that all incoming branches must be completed before continueing with the next activity.

- ***XOR-split:***  To add an XOR-split node to a workflow model, this button can be used. Question mark indicates a decision point to select next activity to continue with.

- ***XOR-join:***  To ad an XOR-join node, this button must be used. If one of the incoming branches is completed, then the next activity will be activated by the workflow manager.

- ***Condition:***  Adding condition for the branch connected to an XOR-split in a workflow model, this button must be used.

- ***Flow Relation:***  This button is used to create a relation between two nodes in a workflow model. These nodes must be connectable, otherwise the relation between them cannot be built.

- ***Marquee Selection:***  This button activates the Marquee Selector, which allows the user to select individual or multiple objects by clicking and dragging left mouse button.

Other Elements

The canvas is where the user is creating and editing his/her workflow model. The Background task progress bar shows work in progress for certain background tasks, like the saving of specification files.

Menu Items

This section provides a brief overview of the EventFlow Editor Menu Items located along the top of the EventFlow Editor.

- ***Specification Menu:*** The Specification Menu provides all the standard file options of Create, Print, Open, Save, Close and Exit. Also, It has two more options to run an opened workflow specification on developed system and open a workflow specification from the list of templates previously saved to the system database. **Figure D.2** shows the dialog reached by clicking "Open Specification From System" menu item. This dialog lists templates which are previously saved to system database. The user can select and open one of the template from the list by just clicking on it, and using "Done" button.

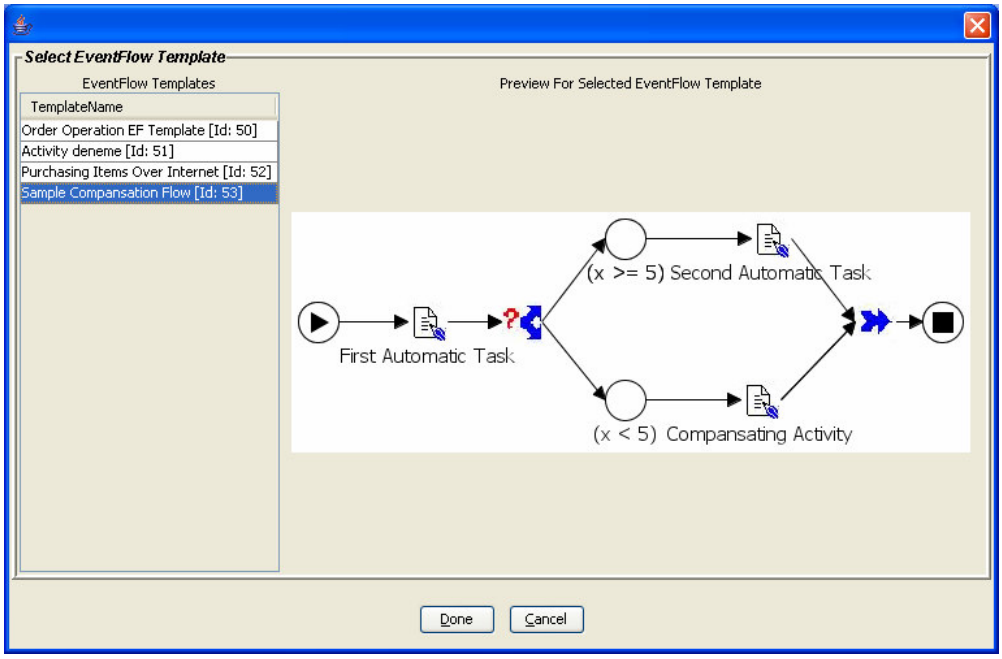


Figure D.2 Dialog to open template from the system

- ***Administration Menu:*** This menu provides the user to define new activity, agent and role definitions for the system. The dialogs opened with the menu items belong to that menu are shown in **Figure D.3, Figure D.4** and **Figure D.5**.

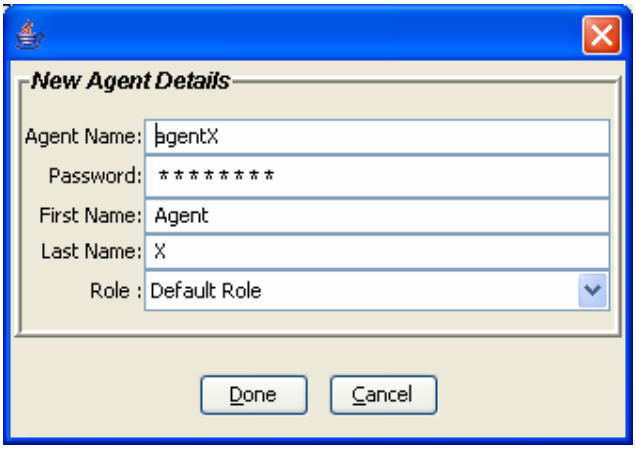


Figure D.3 Add new agent dialog

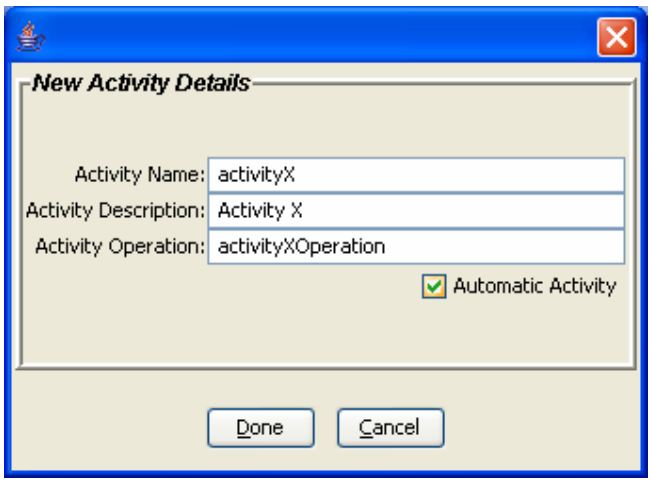


Figure D.4 Add new activity dialog

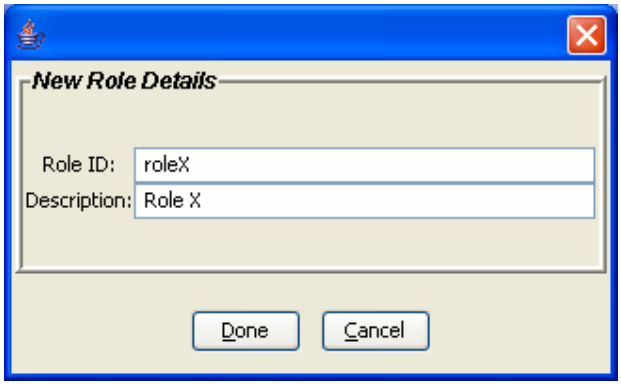


Figure D.5 Add new role dialog

- ***EventFlow Menu:*** The EventFlow Menu provides options to create, remove, rename, and resize the currently selected eventflow specification. It also provides options to export a workflow model to a PNG image file and for directly printing out it.

- ***Edit Menu:*** The Edit Menu provides the standard options of Undo, Redo, Cut, Copy, Paste and Delete objects within your specification.

- ***Elements Menu:*** The Elements Menu allows the user to align EventFlow elements within the current specification, modify their size.

Create an EventFlow specification

This next chapter will lead the user through the process of creating an eventFlow specification from beginning to end, through a series of brief lessons following a scenario. The scenario will be the one used in sample run (see Section 5.2) part of this thesis.

To create new EventFlow specification follow the steps listed below:

- Click  button, or use "*Specification → Create Specification*" menu item to create new empty EventFlow specification

- When the new specification is created, the title will be "*New EventFlow Specification [1]*". To change this title, use "*EventFlow → Rename EventFlow Specification*"

For the example specification, two different data must be declared for indicating the delivery type. These are namely "byAir" with the data type of *Boolean*, and "surface" with the same data type. Follow given steps:

- Right-click the Start element () end select "*Set Global Data*" from the displayed Popup-menu.

- A dialog will be shown. Fill the necessary values (Data name and Data Type) as shown in the **Figure D.6**. There are four different data types available. These are *Boolean, Integer, String* and *Complex.*

- Use "Add" button to add definition to the data container

- Use "Delete" button to remove unnecessary data definition from the data container

- After defining all necessary data, use "Done" button to commit the changes

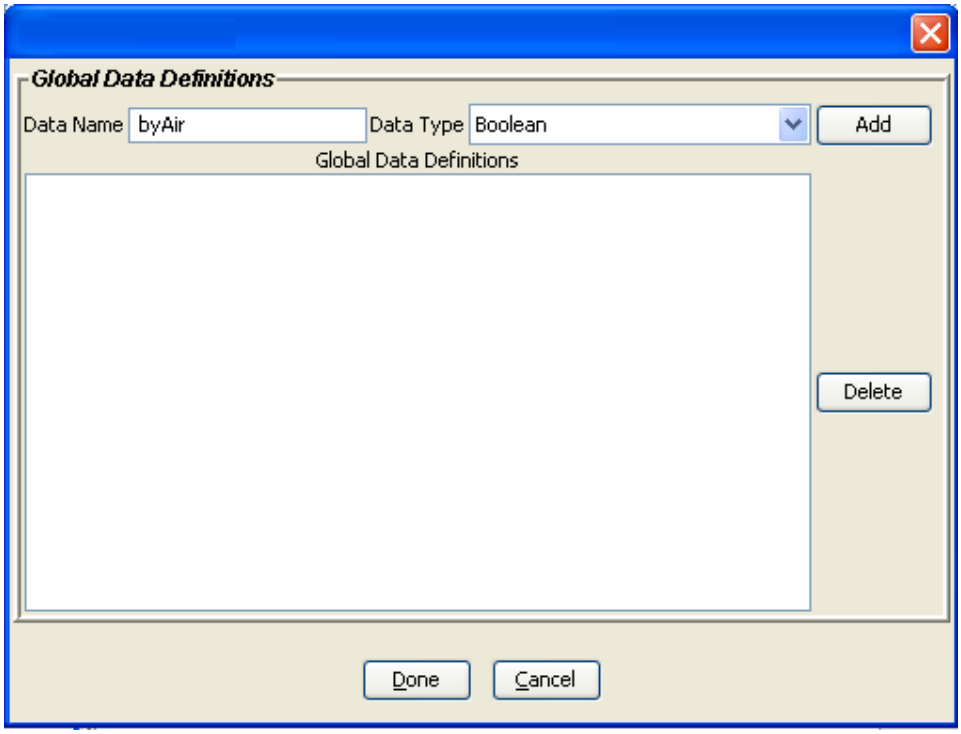


Figure D.6 Global Data Definitions Dialog

To add an activity to the current specification, follow these steps:

- Click  or the popup-menu item "*Activity*" shown when clicking the empty place in the canvas
- Position the mouse where the activity will be placed
- Left click the mouse to place the activity
- Right click the newly added activity, and select "*Edit Activity Details*" from popup menu, to select the task and agent(s) for that activity node. **Figure D.7** shows that dialog.

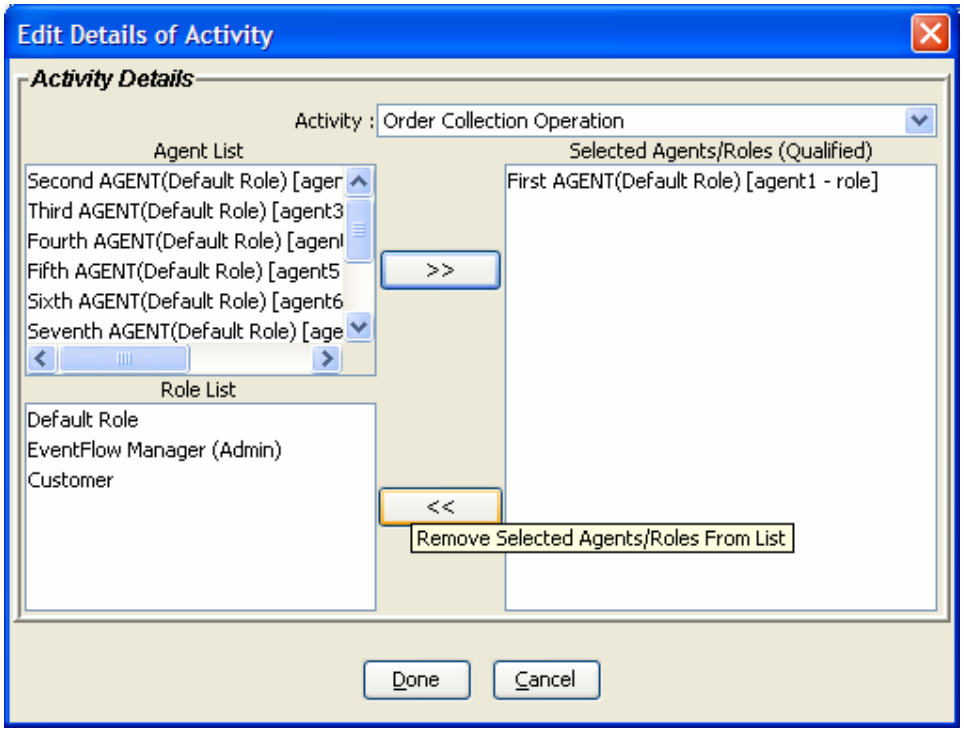


Figure D.7 Edit Activity Details Dialog

To add a condition to the current workflow model:

- Click  , or select "*Condition*" item from the popup menu to add new condition
- Place the mouse to the correct place on the canvas and left click to mouse
- Right click the new condition node to define Condition. Use the dialog shown in **Figure D.8**.

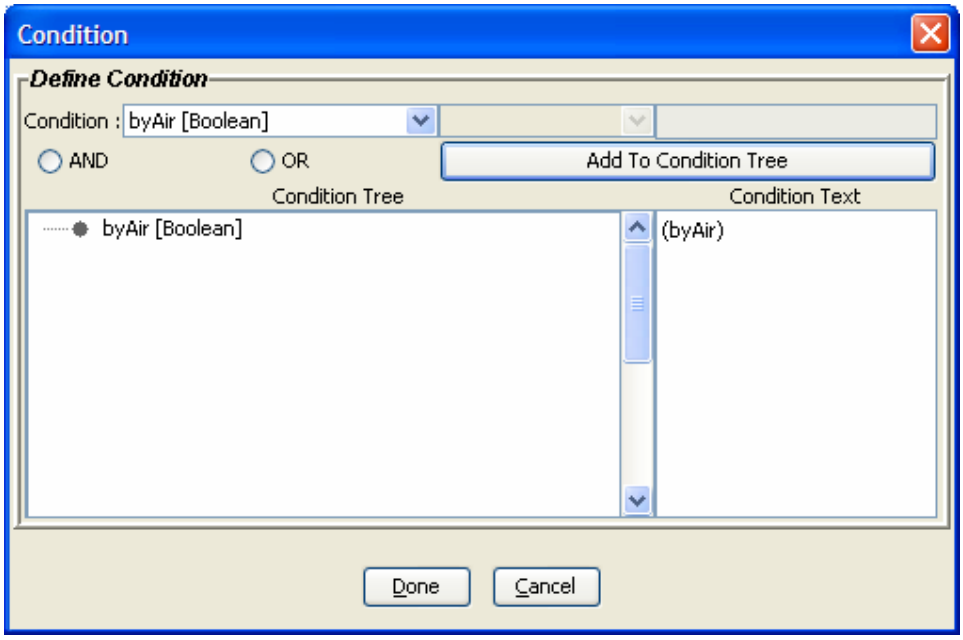


Figure D.8 Define Condition Dialog

To connect two nodes for the current workflow module use "*Flow relation*" like:

- Click  , or select "*Flow Relation*" from popup menu.
- Hold the left mouse button down and dram a line from the node selected to the one it will be connected.
- The editor will indicate which connection points are valid by drawing a blue box around suitable connectors as the mouse passes over them

There are some restrictions to connect two nodes. These are basicly:

- Condition nodes only accept an incoming flow from XOR-split.
- Each node can accept only one incoming flow except the join nodes. (XOR-join and AND-join)
- Each node can have only one outgoing flow except the split nodes (XOR-split and AND-split)

- XOR and AND nodes cannot connect directly.
- Split and join nodes are cannot connect directly

To save the EventFlow specification to the system database use  button. This will sabe the newly created workflow template to the system database with given title for future use.