A PROTOTYPE WORKFLOW ENGINE PARTIALLY SUPPORTING YAWL (YET ANOTHER WORKFLOW LANGUAGE)

YI CHEN

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

NOVEMBER 2004

Your file Votre référence
ISBN: 0-494-04442-X
Our file Notre référence
ISBN: 0-494-04442-X

Abstract

A Prototype Workflow Engine Partially Supporting YAWL (Yet Another Workflow Language)

Workflow has become popular and interesting by removing control flow dependence from business software systems, just as DBMS (DataBase Management Systems) has become a separate domain by removing data dependence from business software systems. YAWL (Yet Another Workflow Language) is a completely new language with its own semantics and it is specially designed for workflow specifications that provide direct support for the workflow patterns identified.

When this thesis was first conceived, no implementation based on YAWL was available. This inspired the development of this thesis.

In this thesis, as a starting point, an XML workflow schema employing seven workflow patterns of YAWL was designed for users to define the workflow specification. Based on this, a prototype workflow engine supporting these seven patterns of YAWL was designed and implemented to parse and interpret the workflow in the control flow specification (the execution order) of YAWL, which is described in the XML workflow document conforming to an XML workflow schema. At runtime, the engine handles the execution order of the workflow.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Dr. Gregory Butler, for giving me an opportunity to work on this challenging topic and for providing continuous guidance, advice, and support throughout the course of this research work.

Thanks to all my friends, who have helped me grow in these years, reminding me of what is really important in my life.

Specially, I want to say thanks to Michael for his great help in difficult times.

Last but not least, I would also like to thank my family, Mom, Dad and Sister. Thank you for listening to me when I needed it. Without their love, support and constant encouragement this work would not have been possible. Although you are far away, you are always in my mind.

Contents

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interface
BPR	Business Process Re-engineering
CEO	Chief Executive Officer
DBMS	Database Management System
DOM	Document Object Model
EPS	Event-controlled Chains
HR	Human Resource
IT	Information Technology
JAXB	Java APIs for XML Binding
SAX	Simple API for XML
WFMC	Workflow Management Coalition
WFMS	Workflow Management System
WSDL	Web Services Description Language
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformations
YAWL	Yet Another Workflow Language

Chapter 1 Introduction

1.1 The Problems

Workflow is "the computerized facilitation or automation of a business process, in whole or part" [29]. In recent years, Workflow Management Systems (WFMS) has become popular and interesting by removing control flow dependence from business software systems, just as DBMS (Data Base Management Systems) has become a separate domain by removing data dependence from business software systems.

The main concerns of workflow are the assessment, analysis, modeling, definition and subsequent operational implementation of the core business process of an organization (or other business entity). The workflow engine provides a runtime execution environment for a workflow instance. However, in the workflow domain, the languages supported by various workflow products differ significantly, and there is no universal standard to follow as a formal basis for the workflow specification, and furthermore, contemporary workflow systems and theoretical models such as Petri nets have problems supporting essential workflow patterns. YAWL workflow research group [2] of Queensland University of Technology identified twenty workflow patterns [30] and developed a new workflow language YAWL by taking Petri nets as a starting point and introducing mechanisms that provided direct support for the workflow patterns identified. The primary advantages of YAWL are that it supports all of the workflow patterns and has a formal semantics [2] as well as a graphical representation.

When this thesis was first conceived, no implementation of YAWL was available. This situation inspired us to develop a prototype workflow engine partially supporting YAWL.

1.2 My Work

The scope of this thesis is as follows: An XML workflow schema employing seven patterns of YAWL is designed for users to define the workflow specifications. These seven workflow patterns of YAWL are five basic control flow patterns: the Sequence pattern, the Parallel Split pattern, the Synchronization pattern, the Exclusive Choice pattern, the Simple Merge pattern, as well as two advanced patterns: the Multiple Choice pattern and the Synchronizing Merge pattern. Based on this, a prototype workflow engine supporting these seven patterns of YAWL has been designed and implemented to parse and interpret the workflow specifications in the control flow perspective (the execution order) of YAWL. The workflow specification has been described in the XML workflow document conforming to the XML workflow schema. At runtime, the engine handles the execution order of the workflow. That is why this thesis is titled: "A prototype workflow engine partially supporting YAWL." In future, the system can be expanded by incorporating other patterns and data perspectives of YAWL into the system.

1.3 Contributions

In this thesis, an XML workflow schema employing the seven workflow patterns of YAWL has been designed as a recursive hierarchy structure. In this way the workflow specifications can be described in an XML workflow document conforming to the XML workflow schema. A prototype workflow engine incorporating the seven workflow patterns of YAWL has been designed and implemented with Java and JAXB to parse and interpret the workflow instances or sample in the control flow perspective (the execution

order) of YAWL, where each workflow instance or samples is described in an XML document conforming to the XML workflow schema.

1.4 Organization of the Thesis

After a brief introduction of this thesis in Chapter 1, workflow, workflow language and YAWL are introduced in Chapter 2, which also covers the related technologies of Java, XML and JAXB. Chapter 3 describes the design of the XML workflow schema in a recursive hierarchy structure to define the workflow specification. And it focuses on the design, implementation, as well as the test cases of the prototype workflow engine. Chapter 4 concludes the thesis.

As the YAWL group developed a workflow management system supporting YAWL while this thesis was approaching its final version (The latest version was released in July, 2004.), a comparison between the two systems has been made in Chapter 3.

Chapter 2 Background

2.1 Definition

"Workflow is concerned with the automation of procedures where documents, information or tasks are passed between participants according to a defined set of rules to achieve, or contribute to, an overall business goal. Workflow is defined by the Workflow Management Coalition (WFMC) as: the computerized facilitation or automation of a business process, in whole or part" [29]. Workflow is concerned with the assessment, analysis, modeling, definition and subsequent operational implementation of the core business processes of an organization (or other business entity), and all of the above activities often associate with Business Process Re-engineering (BPR). Workflow technology often offers an appropriate solution for all BPR activities as it highlights the business procedure logic. Despite that, not all BPR activities result in workflow implementations. Conversely, not all workflow implementations necessarily form part of a BPR exercise, for example, implementations to automate an existing business procedure.

"Workflow Management System (WFMS) is a system that completely defines, manages and executes 'workflows' through the execution of software whose order of execution is driven by a computer representation of the workflow logic" [29]. The main function of a WFMS is to provide procedural automation of a business process, by managing of the sequence of work activities and the invoking of appropriate human or IT resources associated with the various activity steps.

A *business process* is a process coordinating sets of activities inherently dependent on the company's organizational structure in order to achieve business

objectives (for example, an ordering process for an online shopping company). An *Activity or Task* is a logical work item or step within a process (for example, a "register" activity in a trip booking process). A *Work Item* can be an atomic task, a composite task that the user utilizes at a specific point in time during the workflow process. An *instance* describes the course of execution of a process or an activity. *Control flow* is the order of execution of activities and processes.

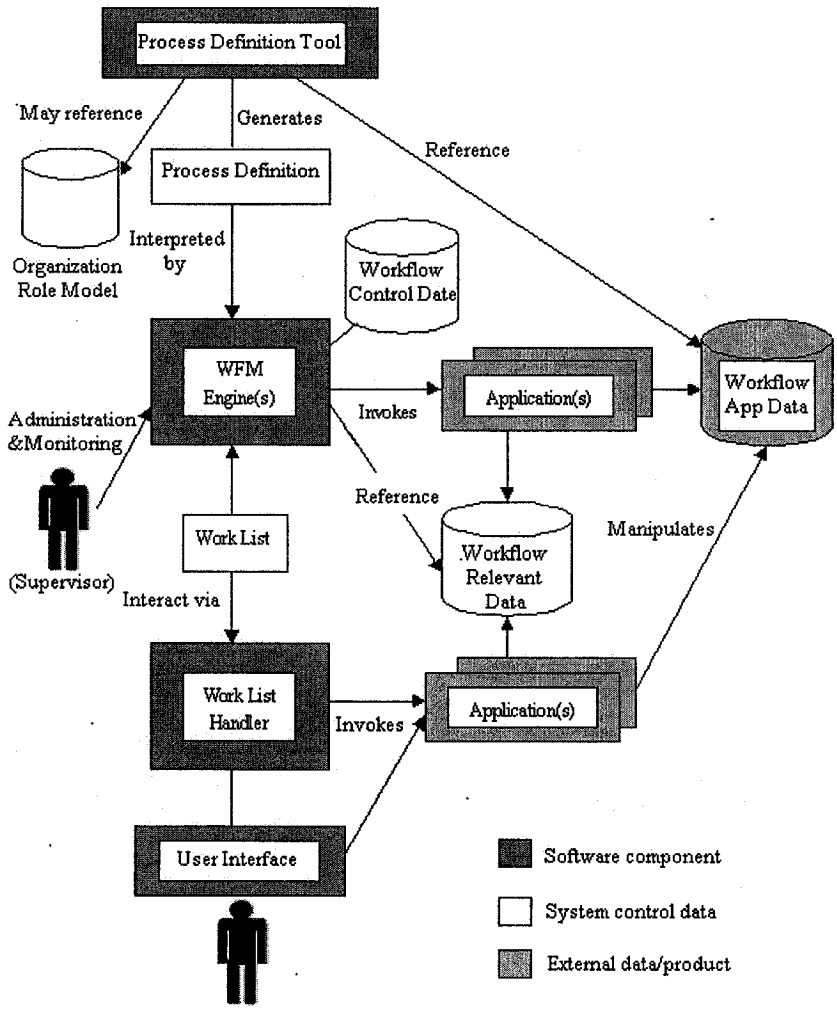

Figure 2-1 General Workflow Product Architecture [29]

Figure 2-1 shows the general workflow product architecture [29] derived from WFMC. The following gives a brief introduction to this general workflow product architecture.

The *Process Definition Tool* is either the component or software needed for users to specify the workflow specification. The *process definition* consists of a set of tasks as well as the specification of their relationships. It also specifies the execution sequence of the process and other related information for each individual task. For example, the start condition, the end condition and the rules for navigating the various activities within the process.

An Application is a software component that interacts with the workflow engine, dealing with certain parts of jobs that are required to support the correspondent task in the process.

Workflow Relevant Data may be manipulated by workflow applications and by the workflow engine. They may affect the choice of the next activity to be chosen (for example, decision data).

Workflow Control Data are the data that are managed by the *Workflow Engine*. These data are the state information of every process instance or every activity instance. The external applications can not access them. For example, the state information can be starting and terminal status of each task in the process.

"A *Workflow Engine* is a software service or engine that provides the run time execution environment for a process instance. Its functions include interpretation of the process definition, creation of process instances and management of their execution,

navigation between activities and the creation of appropriate work items for their processing, supervisory and management functions, etc." [29].

A *Worklist* is a list of activities associated with a given workflow participant. Through the interface information stored in this list, users can view their assigned activities and invoke the execution of the corresponding task applications.

A *Worklist Handler* is a software component that coordinates the interaction between the user and the workflow list. Through the *Worklist handler*, the user can start his assigned task, and the notification of the completion of the task can be passed between the user and the engine.

A *Process Role* abstracts participants to a collection of workflow activities (for example, an accountant who does accounting jobs). "*Organization model* is a model, which represents organizational entities and their relationships. It may also incorporate a variety of attributes associated with the entities, such as skills or role" [29]. The performance of a workflow instance may be administrated and monitored by a supervisor in the organization.

2.2 Why WFMS?

Three basic benefits can be derived from the use of WFMS. The first benefit is derived from its flexibility in changing the model of the underlying business processes, since it separates the control flow specification from the specification of the logic of the application functions, which comprise the algorithmic aspects of the application. Based on this separation, the modification of the model of processes will not influence the associated application implementation. The second benefit is derived from the integration capacity of the different activity implementations, and also, the cross-enterprise business

processes. The third benefit is derived from the improvement in efficiency. By using WFMS, the automation of many business processes results in the elimination of many unnecessary steps.

2.3 Petri Nets

Since the time Petri nets were introduced by C.A.Petri [1] in the sixties, their importance has grown. They have been widely used as a mathematical tool in a variety of ways such as the modeling and analysis of business processes, protocols etc.

A Petri net is a graphical and mathematical modeling tool. It consists of *places* (circles in Figure 2-2), *transitions* (squares in Figure 2-2), and *arcs* (arrows in Figure 2-2) that connect them. *Input arcs* connect places with transitions, while *output arcs* start at a transition and end at a place. *Places* correspond to conditions and each may contain *tokens*. *Tokens* represent objects (humans, goods, machines), information, conditions or states of objects. *Transitions* represent events, transformations or transportations. The states of a process are modelled by tokens in places, while the state transitions leading from one state to another are modelled by transitions. Transitions are only allowed to fire if they are enabled, which means that all the preconditions for the activity must be fulfilled (there are enough tokens available in the input places). When the transition fires, it removes tokens from its input places and adds some or all of them to its output places. The number of tokens removed or added depends on the cardinality of each arc. The interactive firing of transitions in subsequent markings is called the token game.

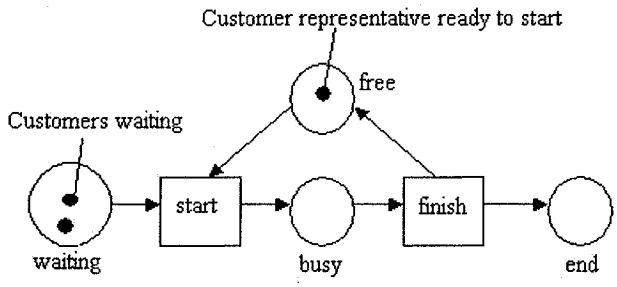

Figure 2-2 A Process Sample Using Petri Nets [1]

The widespread usage of Petri nets can be attributed to their formal semantics [1] and the simple set of graphical notations to represent the workflow.

We can distinguish two main types of Petri net, the classical Petri net and the high-level Petri net.

2.3.1 Classical Petri Net

The classical Petri net is the original basic version that has been in circulation for more than four decades. Classical Petri nets are simply composed of places (denoted by circles) and transitions (denoted by squares) connected through directed arcs. Figure 2-2 shows an example focusing on a simple business scenario for a call center customer service using the classical Petri nets. After a customer connects with the line, he will be in the waiting status. If the customer representative is free, he will provide a service for the client. He then remains in the busy state until the conversation is over. The customer disconnects his line and the service is over. At this moment, the customer representative becomes free, and he is available to talk with another client.

2.3.2 High Level Petri Net

The high level Petri net [1] extends the classical version of Petri nets. With an endowment of extra capabilities, it is able to accommodate more complex and advanced modeling requirements. In the classical Petri net, a token often represents an object

having all kinds of attributes. By extending the classical Petri net it through the color features, the high level Petri net enlarges the representation capabilities of Petri nets. This extension specifies that each token has a value (color) which refers to specific features of the object modelled by the token. For example, if the designer adds a color to the token for the customer representative as outlined in Figure 2-2, the color can be defined to represent the customer representative's name and his identity number.

The extension with hierarchy is done through a hierarchy construct called a subnet, which is well-suited to structure sizable processes. The hierarchy extension helps reduce the size and complexity of the Petri net needed for precise modeling of business processes. This allows simplified representations of complex and detailed process descriptions at the high levels of the hierarchy and detailed stored information at the lower levels of the structure, making it easier to express the functionalities. For example, the place with a colored background can represent a mapping from one task to a composite task, which contains tasks on a lower level of the hierarchy. Thus, Petri nets become more flexible, easy-to-use and handy when dealing with large and complicated business processes.

2.4 Workflow Languages and YAWL

First, this section provides an overview of the workflow patterns consolidated by the YAWL group. Then we will look into some of the popular workflow products and analyze them by using the patterns suggested by the YAWL group.

2.4.1 Workflow Patterns

Here we introduce the twenty workflow patterns introduced by the YAWL group [2]. Based on the analysis of many workflow projects, the YAWL group discovered that

many workflow requirements recur frequently in existing workflow languages. However, they were not implemented in the current versions of workflow products. This study inspired the YAWL group to look for a more structured approach to identify the specification of control flow in workflow languages. Since 1999, they have categorized a complete set of patterns [2], which shares common business requirements from specific workflow languages. These patterns are classified into six categories: basic control flow patterns, advanced branching and synchronization patterns, structural patterns, patterns involving multiple instances, state based patterns and cancellation patterns [2].

Basic control flow patterns: These patterns include the most common or basic flows that are employed for modeling in most workflow languages. They include:

1. ***seq (Sequence)*** is defined as an ordered series of activities, in which only after the completion of an activity, the other activity can start. The WFMC (Workflow Management Coalition) [29] defines this behavior as, "Sequential Routing." The order of the Sequence is determined by the direction of the sequence flow arrowheads. One can think of this as the passage of a conceptual "token" along a sequence flow from the source object to the target object.
2. ***par-spl (Parallel Split)*** is "a point in the workflow process, during which a single thread of control can split into multiple threads, thus activities can be processed simultaneously or in any order" [30]. A single path through the process is split into two or more paths, so that two or more activities are performed concurrently.
3. ***synch (Synchronization)*** is "a point in the workflow process where multiple parallel tasks converge into one single thread of control, thus synchronizing multiple threads. It is an assumption of this pattern that each incoming branch of a synchronizer is executed

only once" [30]. A combination of paths that were generated by a Parallel Split pattern, results in a Synchronization pattern. The "synchronization" of the parallel paths means that the entire set of activities within the flows must be completed before the process can continue.

4. ***ex-ch (Exclusive Choice)*** is a procedure in the workflow process where only one of several alternative paths is executed due to the prior decision. It is defined as being a location in a process where the flow is "split" into two or more exclusive alternative paths where only one of those alternative paths may be chosen for the process to continue.

5. ***simple-m (Simple Merge)*** is a mechanism in the workflow process where multiple parallel tasks converge without synchronization. It is similar to the Synchronization pattern, except that it does not wait for all the activities to be completed. As soon as the fastest task among the incoming branches is complete, the process is ready for the next step, and it will ignore and cancel the execution of activities from other branches.

Advanced branching and synchronization patterns: These patterns expand the basic patterns, thus accounting for more complex splitting and joining methods. This category is composed of:

6. ***m-choice (Multiple Choice)*** is a location in the workflow process where a part or all of the paths can be selected according to the runtime environment. It differs from the Exclusive Choice pattern in that the Multiple Choice pattern allows one, some or all of the alternative paths to be chosen at performance-time. For example, supplier1 and supplier2 are both in the supplier list. After executing the select_supplier activity, either

the activity contact_supplier1 or the activity contact_supplier2 is executed. However, it is also possible that both suppliers are chosen to be executed.

7. ***sync-m (Synchronizing Merge)*** may have a synchronization point if there are several running threads of incoming branches. After all the threads are complete, the next activity can be triggered. There will be no synchronization point, if only one thread of the incoming branches has been started. A branch that has already been activated cannot be executed again while the merge is still waiting for other branches to finish being executed. For example, extending the sample in Multiple Choice Patterns, after either or both contact_supplier1 and contact_supplier2 have been processed (depending on whether these activities are executed at all), the submit_order will be executed (only once).

8. ***multi-m (Multiple Merge)*** is a point in the workflow process where the activity following the merge point can be triggered every time each of the running threads of incoming branches have been completed. Multiple instantiations of the activity are likely to occur. Thus, if there are multiple paths converging on an activity without any token control, it is possible that the activity will be instantiated once, when a token arrives, for each of the paths that are merged. Furthermore, the tokens will all continue independently throughout the remainder of the process (if any). Thus, the main difference between the Multiple Merge pattern and the Synchronization or Simple Merge patterns is that, for the latter two patterns, the target activity will be instantiated only once. For example, in a booking system the activity book_hotel and the activity rent_car are executed concurrently and both of them are followed by the activity pay.

9. ***disc (Discriminator)*** is a point where it waits for one or several of the incoming branches to be completed before activating the subsequent activity. If the discriminator needs some of the incoming braches to be completed, it will only wait for the completion of these expected branches. As soon as these branches are completed, it will trigger the subsequent activity, and the execution of other branches will be cancelled. For example, in order to reduce the waiting time in customer service, a waiting request is sent to all the agents, and the first available will be responsible for the customer, thus canceling all the other agents in this case.

Structural patterns: These patterns are identified to allow for the block structure, which describes the entry and exit points clearly. The following two patterns in this group cover such behavior as looping and the independence of separate process paths.

10. ***arb-c (Arbitrary Cycles)*** can be regarded as an unstructured loop, where activities can be executed repeatedly. The looping segment of the process may allow more than one entry or exit point. This pattern is important to get an idea of valid, but complex, looping situations in a single diagram. For example, in a telecommunication company, the designer creates a new chip whose parameters are tested by the testing engineer. If the result matches the industrial requirements, this design can be introduced into the market. Otherwise, the testing engineer will issue a feedback report to the designer and the testing process is executed repeatedly until the tester obtains satisfactory results.

11. ***impl-t (Implicit Termination)*** occurs when "a given process is terminated if there is nothing to do. In other words, there are no active activities in the workflow and no other activity can be made active (and at the same time the workflow is not in deadlock)" [30].

Patterns involving multiple instances: These are patterns wherein sometimes activities within the context of the process need to be enacted several times during the course of a single process execution. For instance, multiple witness statements need to be handled during the process of an insurance claim. The following four patterns describe how multiple instances or copies of activities are created.

12. ***mi-no-s (Multiple Instances Without Synchronization)*** involves initialization of the multiple instances of an activity without any synchronization point. During the execution of a thread, it can create many independent instances of an activity and the execution of other threads will not depend on these instances either.

13. ***mi-dt (Multiple Instances With a Priori Design Time Knowledge)*** is a point in the workflow process where multiple instances of an activity can be created, and the number of these instances should be decided at the design time. The model designer knows how many times the activity should be performed, and this number is defined in the process model. After all the copies of the activity have been completed, other activities need to be started. For example, during the show time of TV programs, the sponsors' commercials are displayed regularly.

14. ***mr-rt (Multiple Instances With a Priori Runtime Knowledge)*** is different from the *Multiple Instances With a Priori Design Time Knowledge* pattern because the number of these instances can not be known at design time nor can it be set before runtime. It can be known according to the features of the environments, and it is known at some stage during runtime. After all the threads have been executed, the next activity will be started. For example, when doing online shopping, the customer may order many goods. During the processing, the activity check_availability should be executed for every individual

purchase. Only after the availability of each item has been checked, can the shipping activity be triggered.

15. ***mi-no (Multiple Instances Without a Priori Runtime Knowledge)*** demonstrates that "multiple instances of an activity are enabled many times. The number of instances of a given activity for a given case is not known during design time, nor is it known at any stage during runtime, before the instances of that activity have to be created. Once all instances are completed some other activities needs to be started" [30]. The exact number of copies of an activity is actually determined during the performance of these copies, making it different from the previous two patterns. Within a loop, the given instance must be coupled with another instance, which determines if another copy of the given instance is required. If another instance is deemed necessary, then the process continues and the next instance is started, when all the copies of the given instance have been completed. For example, in order to collect the matured grapes in a 200-acre farm to make wine, the winegrower needs to employ some temporary workers. However, it is difficult to figure out the quantity of grapes that can be collected per day, consequently the exact number of days in the harvest season is unknown in advance. After each day, the winegrower can determine whether another work day of the collection instance is needed by calculating the size of the rest of the field. The harvest process is completed after all the grapes in the field have been collected.

State-based patterns: Typical workflow systems do not concentrate on states, rather they concentrate more on flow-type variables such as activities and events. These inherent characteristics limit the expressive power of the workflow language. Hence, state based patterns were deemed necessary. The following three patterns in this group

define how the behavior of a business process may sometimes be affected by factors outside the direct control of the process engine.

16. ***def-c (Deferred Choice)*** is a situation in the workflow process where only one path out of all the possible paths is selected. The selection depends on the environment feature at runtime. This pattern is similar to the Exclusive Choice pattern except that the basis for determining the path that will be taken is different. The Exclusive Choice pattern is based on the evaluation of process data, while the Deferred Choice pattern is based on an event that occurs during the process. When the event is triggered, all the other alternative paths are ignored. For example, all the registered people can be qualified competitors in the auction process. In each round of the bidding, a competitor who submits a price quickly will have priority. Those competitors who react slower will be ignored. In this case, the path of choosing a buyer is based on the bidding event.

17. ***int-par (Interleaved Parallel Routing)*** describes that the set of activities can be executed in an arbitrary order. However, each time, only one activity can be executed. The execution order of the activity from different sets is known at runtime. Irrespective of the order, all the activities in the pattern must be performed sequentially, due to the sharing or updating of the same resources. For example, the MBA program in business schools requires prospective students to take two tests, TOEFL and GMAT, which can be taken in any order but not at the same time.

18. ***milest (Milestone)*** are points within a process where it is important to know whether a specific event has occurred or a condition has been met. An activity is only enabled if a non-expired milestone has been reached. The process model must be able to identify and

react to the milestone. For example, any qualified student can register himself as the candidate for the position in the new student union until the deadline for registration.

Cancellation patterns: These patterns illustrate how the completion of one activity may cause the cancellation of an activity or group of activities. Somewhere during the course of the workflow process, it is necessary to cancel one or more activities. Cancellation of the whole case might even ensue. To respond to these needs the category of cancellation patterns is defined, which includes:

19. ***can-a (Cancel Activity)***, can cancel the corresponding execution task. This pattern describes how, given two competing activities, when one of the activities ends, this activity signals that the other activity should stop processing. Thus, a mechanism for signaling the cancellation and a mechanism for interrupting activities based on this signal are both required. For example, if the customer cancels his booking of the flight before the deadline, the system will not wait for the payment anymore.

20. ***can-c (Cancel Case)*** can remove the whole process instance. This pattern is an extension of the Cancel Activity pattern. For example, if the conference holder cancels the scheduled meeting, the preparation process for the meeting will be disabled.

2.4.2 Background on Workflow Management Systems

The above identified workflow patterns provide a structured approach to construct the control flow perspective in workflow specifications. In order to collect workflow patterns corresponding to the control flow dependency in existing workflow languages, the YAWL group evaluated 15 workflow systems: COSA (Ley GmbH [23]), Visual Workflow (Filenet [10]), Forté Conductor (SUN [11]), Lotus Domino Workflow (IBM/Lotus [20]), Meteor (UGA/LSDIS [22]), Mobile (UEN [12]), MQSeries/ Workflow

(IBM [15]), Staffware (Staffware PLC [25]), Verve Workflow (Versata [28]), I-Flow (Fujitsu [12]), InConcert (TIBCO [27]), Changengine (HP [14]), SAP R/3Workflow (SAP [21]), and Eastman (Eastman [24]). These commercial systems use various workflow languages. The ensuing paragraphs give a brief introduction of workflow systems that the YAWL group evaluated [4].

1. *FlowMark*, one of the first workflow products distinct from document management and imaging services, was renamed *MQSeries/Workflow*. The workflow model mainly consists of activities linked by transitions. The workflow engine of *MQSeries/Workflow* has unique execution semantics, thus allowing every activity that has more than one incoming transition to act as a synchronizing OR-JOIN (*synchronized merge*). This is because, it propagates a False Token for every transition with a condition evaluating to False.

2. *Verve*, an embeddable workflow engine, is very powerful and supports multiple instances and dynamic modifications of running instances along with extra routing constructs, such as a synchronizer and discriminator. The *Verve* workflow model also consists of activities connected by transitions, each having an associated transition condition.

3. *I-Flow* is web-centric and built over a Java or CORBA based engine specifically for independent software vendors and system integrators. Decomposition as well as asynchronous sub-process invocation is supported though it does not allow multiple instances. The workflow model in *I-Flow* consists of activities and a set of routing constructs connected by transitions. Routing constructs include Conditional Node XOR-SPLIT (*exclusive choice*), OR-NODE (*Merge*), AND-NODE (*synchroniser*). AND-

SPLIT can be modeled implicitly by providing an activity with more than one outgoing transition.

4. Workflows are modeled as Event-controlled Process Chains (EPS) by the *SAP R/3 Workflow*. The control flow perspective of EPS consists of a set of functions (tasks), events and connectors (AND, XOR, OR).

5. *Changengine,* a workflow offered by HP Inc., is designed for high performance. The dynamic modeling of arbitrary loops is also supported. Workflow models in Changengine consist of a set of work nodes and routers linked by arcs. The work node can have only one incoming and one outgoing arc.

6. Sun Microsystem's *Forté Conductor* is a workflow engine, based on experimental work. The workflow model that it uses comprises a set of tasks connected with transitions, with each transition associated with a transition condition. Each activity has a trigger, if it has more than one incoming transition. The triggers determine the semantics of that activity and are flexible enough for easy specification of OR-JOIN, AND-JOIN and any type of N-out-of-M join.

7. *Visual WorkFlow*, one of the market leaders in the workflow industry, is part of FileNet's Panagon suite that also includes document management and imaging servers. The highly structural workflow modeling language of *Visual WorkFlow* consists of tasks and routing elements such as Branch (XOR-SPLIT), While (structured loop), Static Split (AND-SPLIT), Rendezvous (AND-JOIN), and Release.

2.4.3 Limitation of Existing Workflow Languages

YAWL group [2] analyzed the majority of the workflow products discussed in the above section. Table 2-1 gives an evaluation result of some of the workflow products

whose workflow languages use workflow patterns. A (+) sign indicates that the product supports the corresponding pattern, while a (-) sign in an appropriate row of the table indicates that the product does not support the corresponding pattern.

pattern	product						
	MQSeries	Forté	Verve	Vis WF	Changeng	I-Flow	SAP/R3
1(seq)	+	+	+	+	+	+	+
2(par-spl)	+	+	+	+	+	+	+
3(synch)	+	+	+	+	+	+	+
4(ex-ch)	+	+	+	+	+	+	+
5(simple-m)	+	+	+	+	+	+	+
6(m-choice)	+	+	+	+	+	+	+
7(sync-m)	+	-	-	-	-	-	-
8(multi-m)	-	+	+	-	-	-	-
9(disc)	-	+	+	-	+	-	+
10(arb-c)	-	+	+	+/-	+	+	-
11(impl-t)	+	-	-	-	-	-	-
12(mi-no-s)	-	+	+	+	-	+	-
13(mi-dt)	+	+	+	+	+	+	+
14(mi-rt)	-	-	-	-	-	-	+/-
15(mi-no)	-	-	-	-	-	-	-
16(def-c)	-	-	-	-	-	-	-
17(int-par)	-	-	-	-	-	-	-
18(milest)	-	-	-	-	-	-	-
19(can-a)	-	-	-	-	-	-	+
20(can-c)	-	+	+	-	+	-	+

Table 2-1 Evaluation Form for Seven Workflow Products [2]

As we can see in Table 2-1, *MQSeries* supports a straightforward Pattern 7 (sync-m). However, other workflow products do not support this pattern because they can not identify either the synchronization point or the merge point in the flow. For Pattern 8 (multi-m), *I-flow* and *Changengine* can not construct or initialize more than one active instance of an activity. *Visual WF* and *SAP/R3* even can not connect a merge construct with a parallel split construct. Only *Verve* and *Forté* are able to realize this pattern

directly. A similar situation occurs through the implementation of Pattern 9 (disc). Only *Verve* and *Forté* provide a specific solution for this pattern. Most of the workflow engines support Pattern 10 (arb-c). However, *MQSeries, Visual WorkFlow* and *SAP R/3* cannot support it directly, because they need to convert the arbitrary cycles into special cycles or loops construct. The semantics of Pattern 11 (impl-t) can only be implemented in *MQSeries*. Other workflow engines require the whole process to end when any end node is reached, and any current activities that happen to be running at that time will be ignored. In the case of Pattern 12 (mi-no-s), only *MQSeries, Changeng* and *SAP R/3* cannot construct multiple instances in a composite task that will be executed in parallel and independent of other threads. Pattern 13 (mi-dt) is supported by all workflow products in Table 2-1. Pattern 14 (mi-rt) can only be supported by *SAP R/3*'s strategy "Table-driven Dynamic Parallel Processing," which is similar to the bundle concept. It offers a bundle construct that is responsible for instantiating a given number of instances of an activity. As soon as these instances in a bundle are all completed, the next activity can be triggered. Unfortunately, none of the workflow products in the above evaluation table are able to construct direct constructs for Pattern 15 (mi-no) and all of the three State-based patterns, such as Pattern 16 (def-c), Pattern 17 (int-par) and Pattern 18 (milest). The cancellation of the activity function is implemented in many of the workflow management systems. However, only *SAP R/3* provides a solution to model Pattern 19 (can-a) in a direct and graphical manner. Others just offer APIs for the withdrawal function by removing the corresponding entry from the database. In the case of Pattern 20 (can-c), *MQSeries*, *Visual Workflow* and *I Flow* still use APIs to cancel cases such as the problem mentioned in Pattern 19.

According to these evaluation results as surveyed by the YAWL group, it is concluded that the workflow patterns have not been fully supported in the existing workflow products and a more structured approach to the issue of the specification of control flow dependencies needs to be established.

2.4.4 Definition of YAWL

Based on the analysis of existing workflow products, the YAWL group identified twenty workflow patterns and developed a new workflow language called YAWL, which directly supports all of the workflow patterns. Moreover, it has a formal semantics [2] and provides a graphical representation. The modeling elements of YAWL are displayed in Figure 2-3.

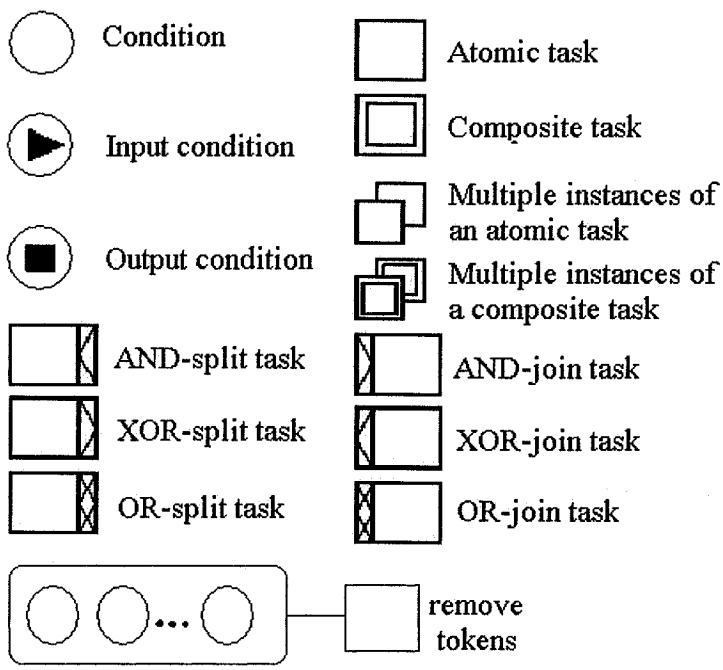

Figure 2-3 Symbols Used in YAWL [2]

A workflow specification in YAWL is a collection of process definitions organized in a hierarchy. In other words, the workflow in YAWL has a tree structure. Tasks can be of two types: either atomic tasks or composite tasks. Each composite task

refers to a process definition at a lower level in the hierarchy. Atomic tasks form the leaves of the tree-like structure. The top level workflow forms the root of the tree-like structure [2]. Each process definition is constructed by tasks and *Conditions* with an *Input condition* (a start place) and an *Output condition* (an end place). Since sequential tasks can connect with each other directly in YAWL, the *Conditions* in-between can be implicit. AND-split task, AND-join task, XOR-split task, OR-split task, OR-join task and XOR-join task correspond to Pattern 2 (par-spl), Pattern 3 (synch), Pattern 4 (ex-ch), Pattern 6 (m-choice), Pattern 7 (sync-m) and Pattern 9 (disc), respectively, in Section 2.4.1.

Moreover, each task (either composite or atomic) can have multiple instances as indicated in Figure 2-3. By extending Petri nets, YAWL defines the configuration of multiple instances with four parameters as [lower bound, upper bound, threshold and static or dynamic]. It is possible to specify a lower bound and an upper bound for the number of instances created after initiating the task. Moreover, it is possible to indicate that the task terminates the moment a certain threshold of instances has been completed. The moment this threshold has been reached, all running instances are terminated and the task is completed. If no threshold is specified, the task is completed, once all the instances have been completed. Finally, there is a fourth parameter indicating whether the number of instances is fixed, after creating the initial instances. The value of the parameter is "static", if, after creation, no instances can be added, and "dynamic" if it is possible to add additional instances while there are still instances being processed.

Finally, the specification of YAWL introduces a notation to remove tokens from places, irrespective of the number of tokens. Figure 2-3 shows this configuration denoted

by dashed rounded rectangles and lines. The enabling of the task does not depend on the tokens within the dashed area. However, the moment the task is executed, all of the tokens in this area are removed. Clearly, this extension is useful in regards of the cancellation patterns [3].

2.4.5 A Workflow Sample Represented by YAWL's Workflow Patterns

Based on the above definition of YAWL, we will discuss a simple workflow sample modeled through YAWL's workflow patterns.

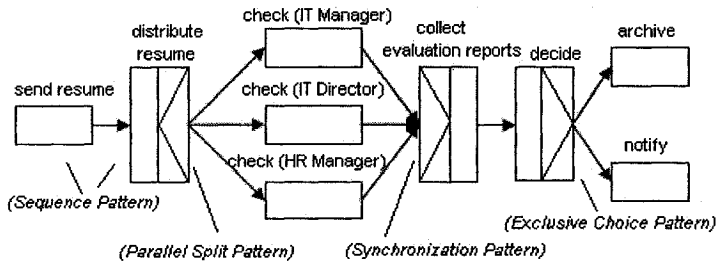

Figure 2-4 Human Resource Recruitment Process Sample

Figure 2-4 presents a scenario of a general human resource (HR) recruitment process for the IT department. The modeling of this process, as represented by YAWL, combines the Sequence pattern, the Parallel Split pattern, the Synchronization pattern and the Exclusive Choice pattern. At the beginning, the receptionist sends the resume to the HR manager. Sequentially, the HR manager simultaneously sends three copies of the resume to the IT manager, the IT director and himself, who are fully responsible for this particular recruitment. Consequently, three checking interviewee tasks will be triggered in parallel (Parallel Split pattern). These three leaders start to check the resume individually and make their evaluation reports. Then the HR manager collects all of these evaluation reports (Synchronization pattern). Based on these reports, he will decide whether to hire the interviewee or not (Exclusive Choice pattern). If the interviewee is

accepted, the HR manager will notify the interviewee. Otherwise, the manager will ask the receptionist to archive the recruitment information.

2.4.6 Advantages of YAWL

High-level Petri nets are also used to model and analyze business processes; however, serious limitations in high-level Petri nets have been discovered when it comes to: (1) *patterns involving multiple instances*, (2) *advanced branching and synchronization patterns*, and (3) *cancellation patterns*. High-level Petri nets are able to express such routing patterns. However, the modeling effort is considerable, and although the patterns are needed frequently, the burden of keeping track of things is left to the workflow designer [2]. The syntax of YAWL gives a better expression to the solution of these problems. The following sample workflow scenarios in YAWL depict how this language meets the corresponding requirements in three different business cases.

Case 1: As an example of how YAWL treats *patterns involving multiple instances*, Figure 2-5 shows the general process for publishing a magazine. At the beginning, the journalists provide reports from various sources without limitation. The interviews are depicted as multiple instances in this modeling diagram, in which the upper bound and threshold are all defined as "infinite". The keyword "dynamic" means that the creation of new interview instances is allowed before the deadline. When the deadline for the acceptance of reports comes, the editors can begin to check all the reports for approval. After they have finished their duties of selection and modification, the layout needs to be done. Then the final version of paper will be sent to the press.

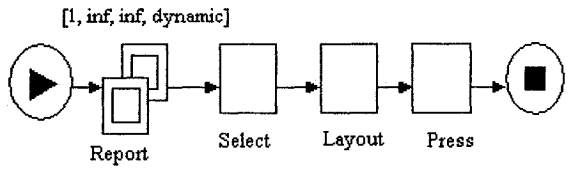

Figure 2-5 Publishing of a Magazine Process Consisting of a Multi-Instance Pattern

Case 2: As an example of how YAWL deals with *advanced branching and synchronization patterns*, Figure 2-6 shows the general order process for the purchasing department to order the necessary goods. When goods need to be ordered, the purchasing department will send mail to all the suppliers. However, it may not send orders to all of them every time. Alternatively, the purchasing department can ask the suppliers about the price list. Therefore, the send-order task is an Or Split Task. The order action should be executed only after the department receives all replies from the chosen suppliers. Here we define the order task as an Or Join Task instead of using the And Join Task to model it. The difference between the Or Join task and the And Join task is that the Or Join Task in the Synchronizing Merge pattern can decide which parts of the incoming branches are required to wait, instead of the whole set.

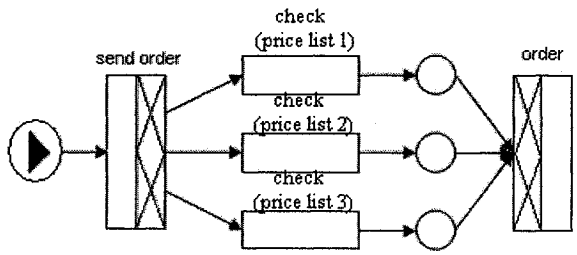

Figure 2-6 Order Process Consisting of an Advanced Synchronization Pattern

Case 3: As an example of how YAWL treats *Cancellation patterns*, Figure 2-7 shows the general process used to reimburse business trip expenses. The accountant sends the application form to the manager, director and CEO. He will then wait for their

signatures to confirm the application. Finally, the accountant will reimburse the expenses once he obtains the first commitment from any one of these three leaders. In the diagram, we use an And Split Task to represent the send requirement task, and we use the execution of the reimburse task as a trigger to cancel the other two commitment tasks.

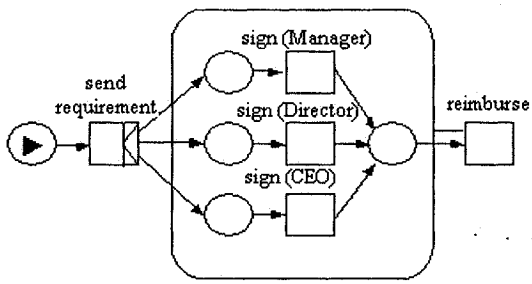

Figure 2-7 Reimbursing Business Trip Expenses Process Consisting of Cancellation Pattern

2.5 Choice of Technologies (DOM, SAX and JAXB)

Interaction with general XML driven software systems requires an XML processor, which transfers from the XML document to the application. We were faced with the following three choices for the XML processor: DOM, SAX and JAXB. After briefly previewing each, we discuss why we chose JAXB.

2.5.1 SAX

SAX or "Simple API for XML" analyzes an XML stream as it passes by. A SAX processor, while parsing, will produce events like Start document, Start element (samples), Characters (white space), etc. The SAX parser then transmits events to an event-handler, whose data are then dealt with by the application. Isolation of such individual events by the SAX API enables the developer to take any relevant action on them [7]. The major limitation of SAX is that it is not possible to navigate backwards in

the document. After firing an event, the parser forgets about it. The application must explicitly buffer those events it is interested in.

2.5.2 DOM

The Document Object Model (DOM) is the foundation of XML. XML documents have a hierarchy of informational units called nodes. DOM is a way of describing these nodes and the relationships between them. By using the DOM API, applications can read and manipulate XML data. The DOM API defines the objects that are present in an XML document and the methods and properties that are used to access and manipulate them. To do this, the whole document has to be loaded first and the related hierarchy has to be built [6]. Its interfaces contain the different types of information that can be found in an XML document, such as the elements and text. It also includes the methods and properties necessary to work with these objects.

2.5.3 JAXB

"The Java APIs for XML Binding, JAXB, establishes a correspondence between the XML schemas and the Java classes and then utilizes the resulting mapping to convert XML documents to the different Java objects" [26].

The concept of data binding is initially used by JAXB to establish the correspondence between the XML templates and the API templates on either side. The JAXB compiler loads an XML schema, and it generates Java classes and interfaces based on the structure of that schema [26]. After the above process, Java applications can use JAXB API to read the XML document, which is written according to the constraints in the source schema. The JAXB API will convert elements in the XML document to Java objects. After it has validated the source XML documents, it will generate a content tree

of Java objects instantiated from the generated JAXB classes. This content tree represents the structure and content of the source XML documents. By using the JAXB API, Java applications can actually change, modify or even create new data objects and then store them as XML documents by means of the interface generated by the JAXB compiler [17].

2.5.4 Why Did We Choose JAXB?

By using JAXB, Java applications can access and modify XML documents easily without having to deal with the complexities of SAX or DOM. Even if developers don't know much about the XML syntax, they can still use the JAXB to compile an XML schema and employ classes generated by JAXB. In this project, the system can be implemented by using SAX, DOM or JAXB. However, JAXB is much more efficient than either SAX or DOM. It can work together with any XML technology that creates or uses SAX events. Specifically, it can communicate with XSLT, DOM, DOMJ, and the XML-aware database, and numerous other existing libraries [26]. That is why we chose JAXB to implement our system.

Chapter 3 The Prototype Workflow Engine

When YAWL was initially designed, there was no implementation available. This inspired us to start developing a prototype workflow engine partially supporting YAWL. In this chapter, we introduce the design of the XML workflow schema, as well as the design, the implementation and the test of the prototype workflow engine. In the last part, we introduce the related work in the YAWL group's project.

3.1 Overview of the Design

Firstly, an XML workflow schema has been designed as a recursive hierarchy structure, which employs YAWL's seven patterns: Sequence pattern, Parallel Split pattern, Synchronization pattern, Simple Merge pattern, Exclusive Choice pattern, Multiple Choice pattern and Synchronizing Merge pattern. Based on the XML workflow schema, users can define the workflow specification in an XML workflow document conforming to the XML workflow schema. This XML document is used as the input file of our system.

Secondly, a workflow engine supporting these seven workflow patterns has been designed and implemented to parse and interpret the XML workflow document. Its functions include: parsing the input file (the XML workflow document conforming to the XML workflow schema), interpreting the workflow specification in the XML document and managing the execution order of tasks. For the tasks without human interaction, when the engine enables the task, it directly invokes the corresponding application function of the task. For the tasks that require human interaction, a worklist handler provides a user interface for the task participant to view and operate the assigned task.

The operations include starting the execution of the application function corresponding to the assigned task and notifying the engine about the completion of the execution.

Finally, the log information of each task will be recorded as a demonstration of the execution of the workflow sample.

This prototype focuses on the control flow perspective (the execution order) of workflow. It runs on Windows XP platforms, and it is implemented with Eclipse version 2.1.3 and Java Runtime Environment (JRE) in version 1.4.0.

3.2 Concept Design

Our design of this system in Figure 3-1 is based on the general workflow product architecture in Figure 2-1.

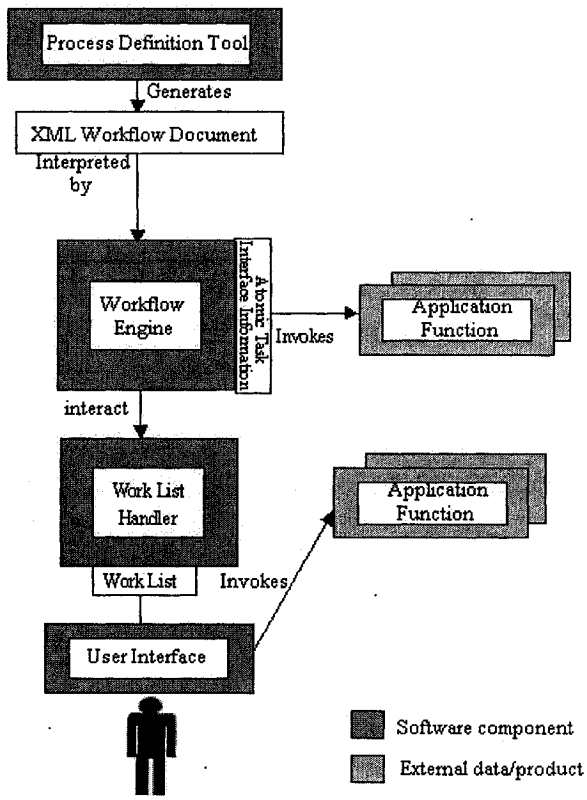

Figure 3-1 Architecture for the Prototype Workflow Engine

The main components in Figure 2-1 include a process definition tool (it is used for the process designer to define the workflow), a worklist handler (it is used to manage the human interaction of the workflow), and the core of the system, which is the workflow engine (its functions include interpretation of the process definition, creation of the process instances and management of their execution, including start / stop / suspend / resume, etc., supervisory and management functions, etc.). Based on the workflow specification, the engine will directly invoke the application corresponding to the scheduled task without human participants, and it passes the task to the worklist handler when the task requires human interaction. In the architecture of the industrial length of workflow product, the relevant data are stored in the specific database (these data include workflow relevant data, workflow control data and a role model, etc.). At runtime, the system can access this information in the databases.

The design of our system is based on the general workflow product architecture. We use XML Spy 5.0 (an XML editing tool) as the process definition tool to define the workflow specification. The workflow engine in our design supports seven patterns of YAWL, it can parse and interpret the workflow specification in the control flow perspective (the execution order) of YAWL, and it manages the execution order of an instance of workflow at runtime (the resuming and suspending the workflow instance is beyond the scope of our system). The monitor and administration function is not implemented in our system. For tasks without a human participant, the engine will also invoke the execution of application functions. For tasks that require human interaction, our engine will ask the worklist handler to provide a user interface for users to view and operate the assigned task. Our prototype focuses on the control perspective of workflow,

and no database is used in our system. The workflow specification contains the specification of workflow relevant data and workflow control data.

In our test cases, each time the engine enables a task, a new thread instance will be constructed to execute the task's corresponding application function. In order to show that the execution order conforms to the workflow specification in YAWL, each application function should print its starting time and end time. As the task of each application function is domain specific, and this thesis focuses on the control flow perspective of a workflow, we use one print log function to stand for all the different application functions, and this print log function is a Java function. However, users in specific domains can program their application libraries when they develop their workflow management systems based on this prototype, and they can specify which application function should be invoked for each atomic task.

With the interface information specified in the atomic tasks in the XML workflow document, the engine can invoke the corresponding application functions. This interface information can be regarded as the interface between the workflow engine and the application functions.

The design of this workflow engine is based on the workflow engine in the general workflow product architecture. However, this thesis focuses on the control flow perspective of workflows. In future, our prototype can be expanded by incorporating data perspectives into the system. The administration and monitor functions of the workflow engine in the general workflow product architecture are not within the scope of this prototype. Future research will be needed to address these problems.

3.3 Detail Design

In this section, we introduce the detail design of our system. Our design includes the design of the XML workflow schema and the design of the prototype workflow engine.

3.3.1 The Design of the XML Workflow Schema

When YAWL was first designed, no implementation was available. This inspired us to propose an XML workflow schema, which employs YAWL's seven workflow patterns to support the control flow specification (the execution order) in workflows. This schema defines the XML workflow document's structure, in which we employ the following patterns: Sequence pattern, Parallel Split pattern, Synchronization pattern, Simple Merge pattern, Exclusive Choice pattern, Multiple Choice pattern and Synchronizing Merge pattern. Moreover, we group the Parallel Split pattern and the Synchronization pattern together, group the Multiple Choice pattern and the Synchronizing Merge pattern together and finally group the Multiple Choice pattern and the Simple Merge pattern together (see Section 3.3.1.1). A prototype workflow engine has been designed to implement the semantics of these patterns. This section gives an overview of the XML workflow schema as well as its recursive hierarchy structure, and it describes how YAWL's seven workflow patterns are employed and how the grouping approach is designed.

3.3.1.1 Overview of the Control Flow Specification in the XML Workflow Schema

The XML workflow schema defines workflow patterns by specifying their elements, their attributes and their structure (the sequence, relationship and occurrence of the elements). It also defines composite tasks, each of which consists of a single

workflow pattern or a combination of several workflow patterns. It finally defines workflow in a recursive hierarchy with workflow patterns and composite tasks as building blocks.

Since business processes can be recursively broken down into finer levels of detail, the XML workflow schema is designed to be constructed with workflow patterns recursively in a hierarchy structure. The following paragraph defines the building blocks of this hierarchy structure.

Each workflow pattern realizes one specific complex function, which can not be recognized by an individual task. Each workflow pattern consists of a group of related tasks cooperating together to fulfill its complex function by specifying the execution order and the relationship of tasks within its scope. A task can be either an atomic task or a composite task. In the XML workflow schema, the composite task is defined as: the input condition (a starting point), a single workflow pattern or combination of several workflow patterns and the output condition (a terminal point). The following paragraph describes how this hierarchy is constructed recursively.

Using a top-down approach, the process at the top level can be regarded as a composite task, and it includes one workflow pattern consisting of tasks making up the second level of the hierarchy, with the atomic tasks as the leaf nodes, and the composite tasks as parent nodes. For all composite tasks at the second level, each composite task also includes one workflow pattern consisting of tasks making up the third level of the hierarchy with the atomic tasks as leaf nodes and composite tasks as parent nodes. We continue the above step for each composite task at newly generated levels of the

hierarchy until there are no more composite tasks at a certain level. In this way, the XML workflow schema hierarchy is constructed recursively.

According to the definition of YAWL, a combination of paths that are generated by a Parallel Split pattern, results in a Synchronization pattern. "Synchronization pattern is a point in the workflow process where multiple parallel activities converge into one single thread of control, thus synchronizing multiple threads" [30]. Therefore, we group the Parallel Split pattern and the Synchronization pattern together. In the XML schema, we briefly call this group "parsyn".

According to the definition of YAWL, the Multiple Choice pattern is a point where one, some, or all of the alternative activities are chosen at runtime. "The Synchronizing Merge pattern is a point in the workflow process where multiple paths reconverge into one single thread. If more than one path is taken, synchronization of the active threads needs to take place. If only one path is taken, no synchronization occurs" [30]. Therefore, we also group the Multiple Choice pattern and the Synchronizing Merge pattern together. In the XML schema, we briefly call this group "mulsyn".

In YAWL, the Multiple Choice pattern is a point where one, some or all of the alternative activities are chosen at runtime time, then those selected activities are executed in parallel. The Simple Merge pattern is a point where multiple activated branches reconverge without synchronization. Therefore, we group the Multiple Choice pattern and the Simple Merge pattern together. In the XML schema, we briefly call this group "mulsim".

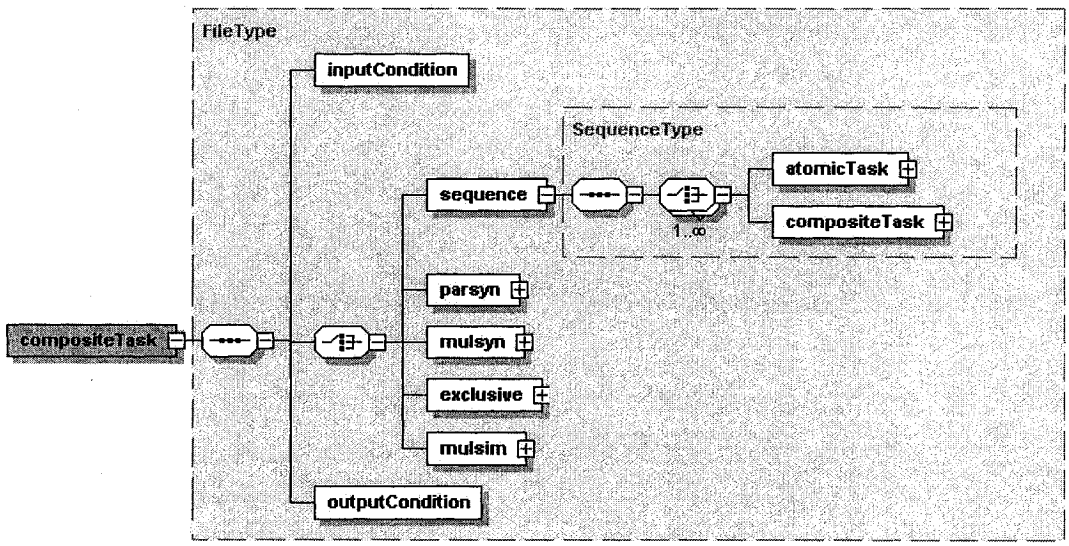

Figure 3-2 Control Flow Specification in the XML Workflow Schema

Figure 3-2 gives an overview of the recursive hierarchy structure of the XML workflow schema. The "FileType" represents the composite task. Each composite task consists of an input condition, a single workflow pattern, or a combination of workflow patterns, and an output condition. Each workflow pattern consists of a group of related tasks cooperating together to fulfill its complex function. A task can be an atomic task or a composite task. If the components of the patterns consist of composite tasks, the composite tasks can be extended recursively.

To explain the recursive hierarchy structure, we again present the human resources recruitment process in the following diagram.

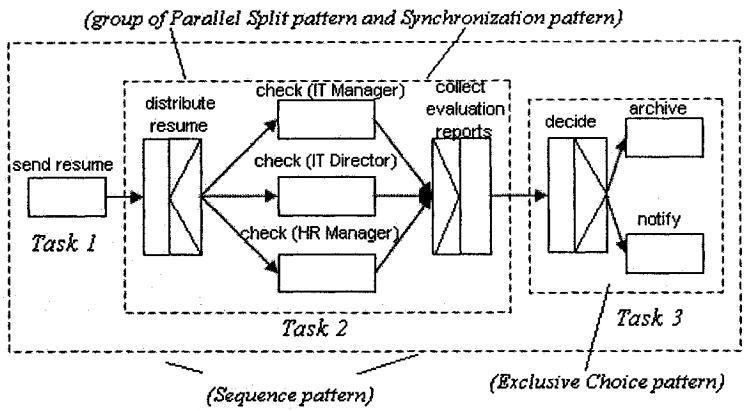

Figure 3-3 A Process Sample Modeled in the Recursive Hierarchy Structure

Figure 3-3 shows the workflow for this process modeled as a three level hierarchy. The top level is the process consisting of the Sequence pattern. The Sequence pattern consists of:

- Task 1 involves sending a resume from the receptionist to the human resource manager;
- Task 2 involves evaluating the interviewee by the human resources manager, the IT manager and the IT director;
- Task 3 involves the processing of the recruitment result;

At the second level, Task 1 is an atomic task, and Task 2 and Task 3 are composite tasks.

The composite task, Task 2, again consists of the group of the Parallel Split pattern and the Synchronization pattern. Tasks that make up this pattern are at the third level.

- Task 2.1 (the and split task) involves distribution of the resume from the human resource manager to the IT manager, the IT director and the human resources manager;

- Task 2.2 (the atomic task) involves checking the interviewee by the IT manager;
- Task 2.3 (the atomic task) involves checking the interviewee by the IT director;
- Task 2.4 (the atomic task) involves checking the interviewee by the human resources manager;
- Task 2.5 (the and join task) involves collecting the evaluation reports from the IT manager, the IT director and the human resources manager;

The composite task, Task 3, again consists of the Exclusive Choice pattern. Tasks that make up this pattern are indicated at the third level.

- Task 3.1 (the xor split task) involves a decision by the HR manager;
- Task 3.2 (the atomic task) involves archiving if the interviewee is denied;
- Task 3.3 (the atomic task) involves sending a letter to notify the interviewee if he or she has been accepted.

In the above example, Task 2 is a composite task consisting of the group of the Parallel Split pattern and the Synchronization pattern. Task 3 is a composite task consisting of the Exclusive Choice pattern. Task 2 and Task 3 are connected by a Sequence pattern. Because there are no more composite tasks at the third level, the recursive analysis method can cease at this level. In this way, the control flow perspective of this process is specified by patterns in the recursive hierarchy structure.

3.3.1.2 Sequence Pattern

As described in the following schema diagram, the Sequence pattern consists of atomic tasks or composite tasks (the "FileType" represents the composite task in Figure 3-4). The number of tasks should be more than or equal to one. If the task is a composite task, it will consist of either a single pattern or a combination of workflow patterns. As

shown in Figure 3-4, the Sequence pattern is within the recursive hierarchy structure (See the description of the recursive hierarchy structure for the XML schema in Section 3.3.1.1.).

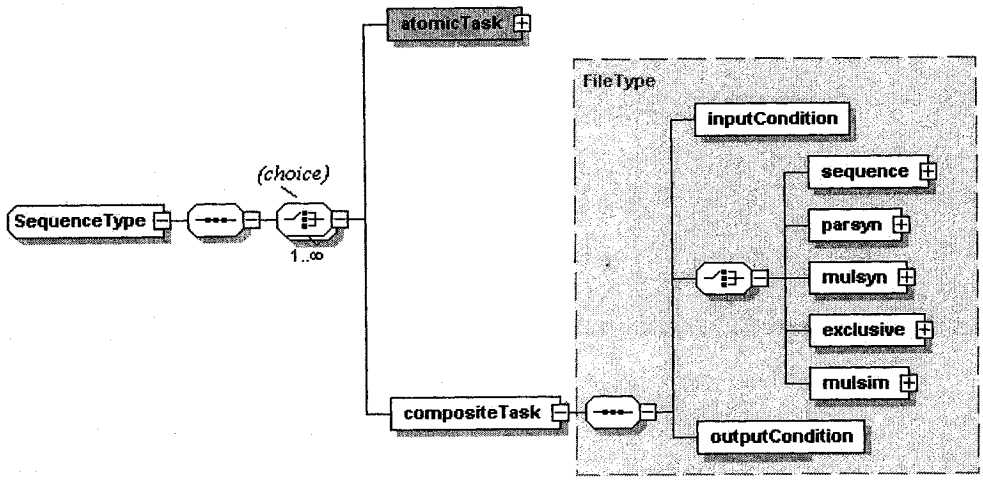

Figure 3-4 Sequence Pattern

3.3.1.3 Group of the Parallel Split Pattern and the Synchronization Pattern

The following diagram in Figure 3-5 specifies the group involving the Parallel Split pattern and the Synchronization pattern.

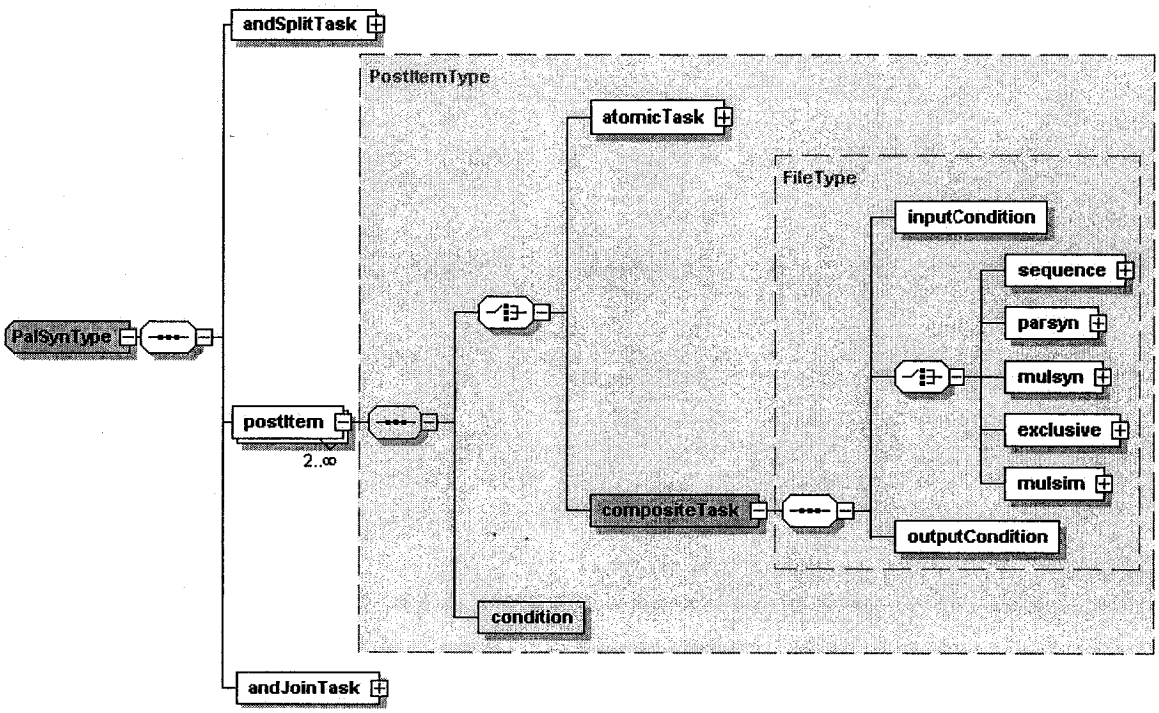

Figure 3-5 Group of the Parallel Split Pattern and the Synchronization Pattern

According to the specification in YAWL, a combination of paths that are generated by a Parallel Split pattern, results in a Synchronization pattern. The YAWL's Synchronization pattern is "a point in the workflow process where multiple parallel activities converge into one single thread of control, thus synchronizing multiple threads" [30]. Hence, we group the Parallel Split pattern and the Synchronization pattern together. This group is called "ParSynType" in the XML workflow schema. In the above diagram, the andSplitTask, the parallel tasks (either atomic tasks or composite tasks) in the postItem and the andJoinTask are executed sequentially at runtime. The andSplitTask represents the first triggered activity. After its execution, all the remaining parallel tasks will be triggered. Thus, the atomicTask or the compositeTask in the incoming branches (postItem) can be processed in any order, or they can be processed simultaneously. There

is a synchronization point, where the andJoinTask waits until all the tasks in the postItem have finished their executions.

3.3.1.4 Group of the Multiple Choice Pattern and the Synchronizing Merge Pattern

The following diagram in Figure 3-6 specifies the group of the Multiple Choice pattern and the Synchronizing Merge pattern.

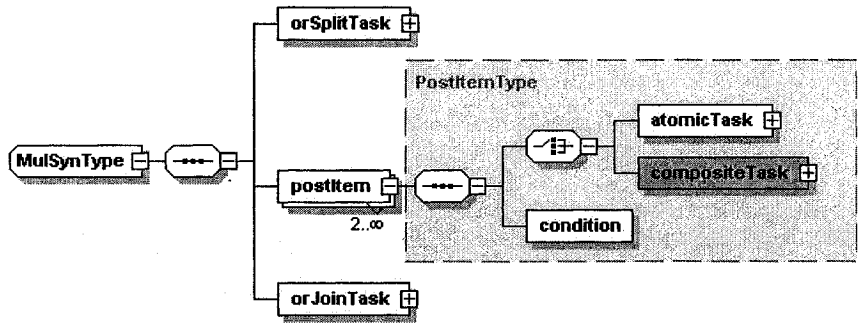

Figure 3-6 Group of the Multiple Choice Pattern and the Synchronizing Merge Pattern

According to the definition of YAWL, the Multiple Choice pattern is a point where one, some, or all of the alternative activities are chosen at runtime. "Synchronizing Merge pattern is a point in the workflow process where multiple paths converge into one single thread. If more than one path is taken, synchronization of the active threads needs to take place. If only one path is taken, the alternative branches should reconverge without synchronization" [30]. Therefore, we group the Multiple Choice pattern and the Synchronizing Merge pattern together. This group is called the "MulSynType" in the XML workflow schema. The schema specifies that there are at least two incoming branches (postItem). Based on the result of the execution of the orSplitTask, only a subset of the parallel child tasks in the incoming branches will be selected for execution. The orJoinTask represents the point where it will not start until all the selected tasks have been executed.

3.3.1.5 Exclusive Choice Pattern

The Exclusive Choice pattern is called the "ExclusiveType" in the XML workflow schema. It consists of xorSplitTask and other alternative tasks. The number of those alternative tasks must be more than one. The schema specifies that the xorSplitTask must be executed first to select only one task out of many. The selected task needs to be processed afterwards at runtime.

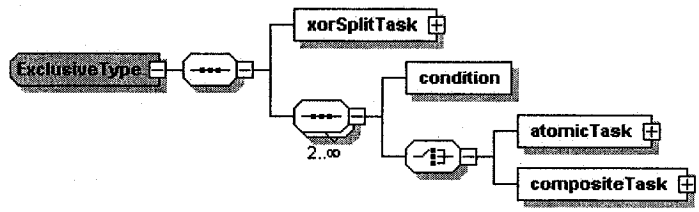

Figure 3-7 Exclusive Choice Pattern

3.3.1.6 Group of the Multiple Choice Pattern and the Simple Merge Pattern

The following diagram in Figure 3-8 specifies the group of the Multiple Choice pattern and the Simple Merge pattern.

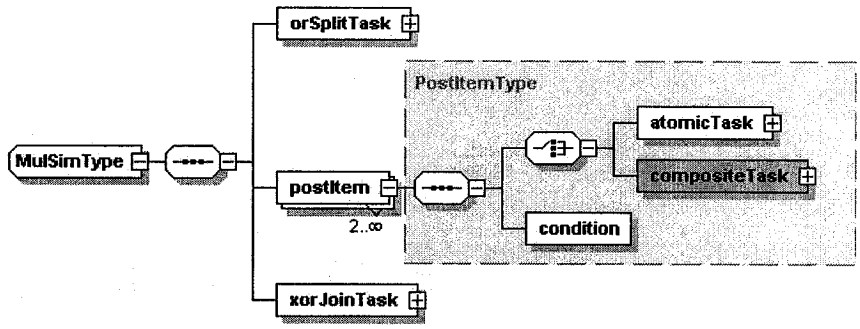

Figure 3-8 Group of the Multiple Choice Pattern and the Simple Merge Pattern

In YAWL, the Multiple Choice pattern is a point where one, some or all of the alternative activities are chosen at runtime. Simple Merge is a point where multiple activated tasks in parallel reconverge without synchronization. Therefore, we group the Multiple Choice pattern and the Simple Merge pattern together. In the XML schema, this

group is called the "MulSimType". At first, the orSplitTask is responsible for choosing several execution paths from many alternatives. Based on the result of the execution of the orSplitTask, a subset of the parallel tasks will be selected for execution. However, the xorJoinTask will not wait for all the enabled tasks to finish their jobs. When the enabled task that is completed first, other parallel tasks will be ignored. The task that is completed first can trigger the xorJoinTask to work, and the execution of the other tasks is canceled.

3.3.1.7 Interface Information in the XML Workflow Schema

Figure 3-9 shows an overview of the XML workflow schema for a general workflow process. We use the root element "process" to represent a workflow containing the control flow specification and the specification of workflow relevant data. In former sections, we have already outlined how we employ YAWL's seven workflow patterns in the recursive hierarchy structure for the modeling of the control flow specification in workflow. In this section, we introduce the specification of workflow relevant data in Figure 3-9.

relevantData: This is an optional element for modeling the workflow relevant data. It is only specified if workflow relevant data is accessed or processed by different tasks in the process, and it includes a *variable* element list.

variable: This element is used to specify each workflow relevant data. The *initialValue* element is used to define the initial value of the *variable*. The *name* element specifies the *variable*'s name. The *value* element retains the *variable*'s value at runtime.

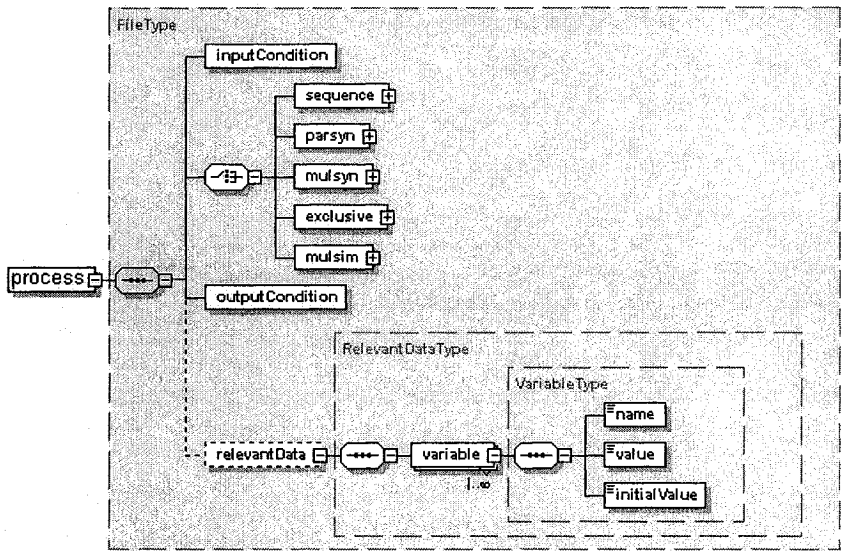

Figure 3-9 Workflow Relevant Data

3.3.1.8 Interface Information for the Atomic Task and the Application Function

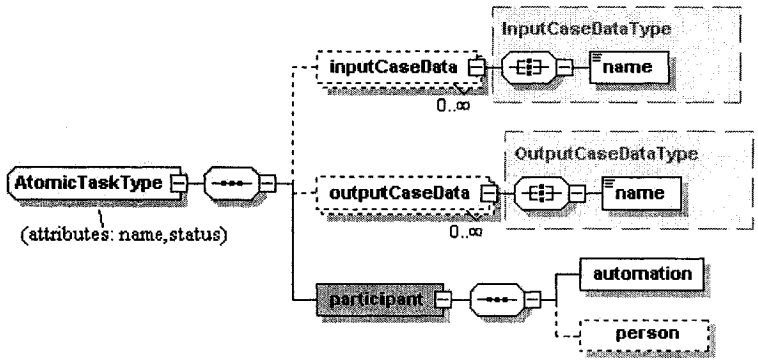

Figure 3-10 Interface Information for the Atomic Task and the Application Function

In Figure 3-10, each atomic task contains two attributes: the *name* and the *status*. The *name* attribute represents the name of the task. The *status* attribute represents the control state information of the task, which is the workflow control data. The state information can be "enabled" (this means that the task can be selected by the engine for future execution), "disabled" (this means that the engine disables the execution of the task), "start" (this means that the task is started for execution) and "end" (this means that the task is over).

In our design, we specify that a task can be either an automatic task without human interaction or a task associated with a human participant. For the automatic task, we specify that the *participant* element of the automatic task contains only the name of the task's corresponding application function (*automation*). (The *automation* element defines the name of the application function, which will be invoked when the task is enabled at runtime.) For the task associated with a human participant, the *participant* element contains the name of the application function (*automation*) and a human participant (*person*). Here the *person* element is specified by the human participant's role (for example, a manager or a staff member). The engine can recognize the type of each atomic task by checking the *participant*. If it is an automatic task, the engine can directly invoke the corresponding application function to execute. If it is a task that needs human interaction, the engine will pass the task to the worklist handler, which will add the interface information of this task into the corresponding user's worklist. In our implementation, the worklist handler will show a user interface with the interface information in the worklist. Through the user interface, the user can invoke the execution of the corresponding application function and notify the engine of the completion of the application function (see the description of WorklistHandler and Worklist in Section 3.3.2.4).

The information in the atomic task can be regarded as the interface information between the engine and the task's corresponding application function. During the execution of the corresponding application function, it may need input parameters and output parameters. These data are all defined in the specification of the atomic task as shown in Figure 3-10.

inputCaseData: This is an optional element. If the application function needs input parameters for its execution, each input parameter can be specified with its names in this element. According to this specification, the application function can get its input parameters from the *variable* list defined in Figure 3-9.

outputCaseData: It is an optional element for the atomic task to specify the output parameters returned by the corresponding application function.

3.3.2 The Design of the Prototype Workflow Engine

In this section, we introduce our design of the prototype workflow engine.

3.3.2.1 Architecture Diagram of the Prototype Workflow Engine

The workflow engine in this thesis can be divided into five subsystems: Generator, Parser, ParseTree, Visitor and ThreadGroupManager as shown in Figure 3-11.

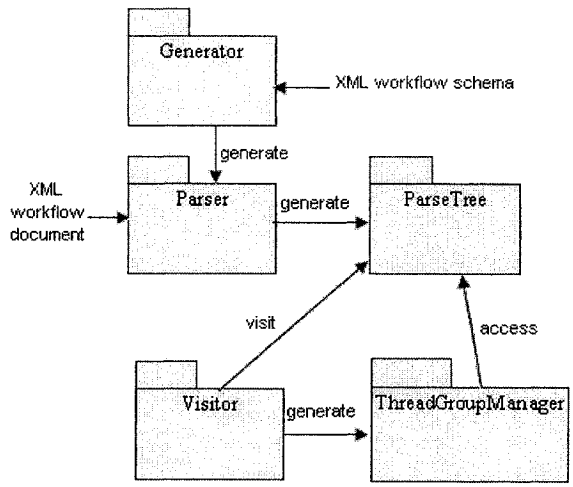

Figure 3-11 UML Architecture Diagram for the Workflow Engine

The Generator subsystem is used to create a correspondence between the XML workflow schema and Java classes. It reads an XML workflow schema, and it generates Java classes and interfaces based on the structure of that schema [26]. The Parser

subsystem reads the XML workflow document conforming to the XML workflow schema. The Parser subsystem will convert elements in the XML document to Java objects. After it has validated the XML workflow documents, it will generate a content tree of Java objects instantiated from the generated JAXB classes. This content tree represents the structure and content of the XML documents. In our design, the Parser subsystem can only load the XML workflow document conforming to the XML workflow schema, and then convert it to Java objects that are instances of the classes that were created by the Generator. We use JAXB technology to implement the Generator subsystem and the Parser subsystem. The JAXB schema complier compiles the XML workflow schema and generates corresponding Java classes. Then the JAXB API acts as a parser to transform the XML workflow document into the corresponding Java objects in a tree structure.

The ParseTree subsystem has a composite tree structure based on the Composite design pattern [13]. As mentioned above, it is generated into an inheritance tree structure by the JAXB API. A further illustration of this package will be presented in Section 3.3.2.2.

The Visitor subsystem is implemented to access the object nodes in the ParseTree. During its visiting, the visitor constructs corresponding threads that are managed by the ThreadGroupManager subsystem. This design is based on the Visitor design pattern [13]. We will explain the details of this pattern in Section 3.3.2.3.

The ThreadGroupManager subsystem is responsible for the management of the execution order of tasks in each workflow scenario. It consists of controller threads and broker threads. After parsing and interpreting the XML workflow document, controller

threads are then constructed. Since those controllers implement the specification of workflow patterns, they are responsible for managing the execution order of other threads. Each atomic task defined in the workflow specification is related to its own broker thread. Based on the interface information specified in the atomic task in the workflow specification, the broker can take care of the execution of the enabled task's corresponding application function. Details of our design of this subsystem will be illustrated in Sections 3.3.2.4.

3.3.2.2 ParseTree Subsystem

At runtime, the Parser (JAXB API) automatically converts the XML workflow document conforming to the XML workflow schema into Java objects, and it generates a Java object tree, which represents the structure and content of the XML workflow document. This Java object tree makes up the ParseTree subsystem. Since the XML document is written according to the constraints in the XML workflow schema and the XML workflow schema has already been specified in a recursive hierarchy structure, which is an implementation of the Composite design pattern [13], here we use the Composite design pattern to describe the structure of the ParseTree subsystem.

According to the definition of the Composite design pattern, "the Composite design pattern allows various individual tasks to be composed into composite tasks. In this way, the primitive element can be alternatively composed and so on recursively" [13]. By using the Composite design pattern, individual tasks and composite tasks can be handled uniformly, therefore, people do not need to care whether they are dealing with individual or composite tasks. Moreover, newly defined process and leaf subclasses can be manipulated conveniently through the existing structure and codes.

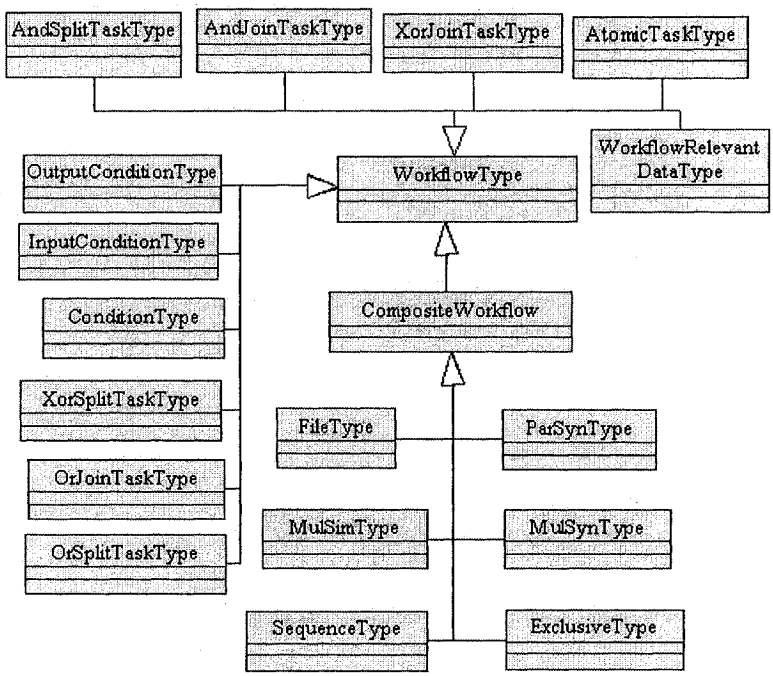

Figure 3-12 Composite Design Pattern in the Workflow Engine

In the ParseTree subsystem, the WorkflowType abstract class is defined as a component to provide the interface for all the classes, as shown in the composition in Figure 3-12. Individual tasks, the input condition and the output condition are declared as primitive classes, which have no children and have no implementation behavior. We add a composite class called CompositeWorkflow as a container. The Sequence pattern, the Exclusive Choice pattern, and the other three groups of workflow patterns as well as the composite task are defined to inherit the CompositeWorkflow class, which have child components and which specify the related behavior among their children. Through the declaration of this structure with the composite design pattern, we can get an explicit hierarchy tree structure. This pattern also makes it easier for us to add other workflow patterns in the future.

After the JAXB parser has parsed the XML workflow document, a parse tree is constructed as a composite structure in the memory. Each node provides interfaces to

access its elements. In order to avoid race conditions, all concurrent accesses are synchronized.

3.3.2.3 Visitor Subsystem

We use the Visitor design pattern [13] to construct this subsystem as shown in Figure 3-13.

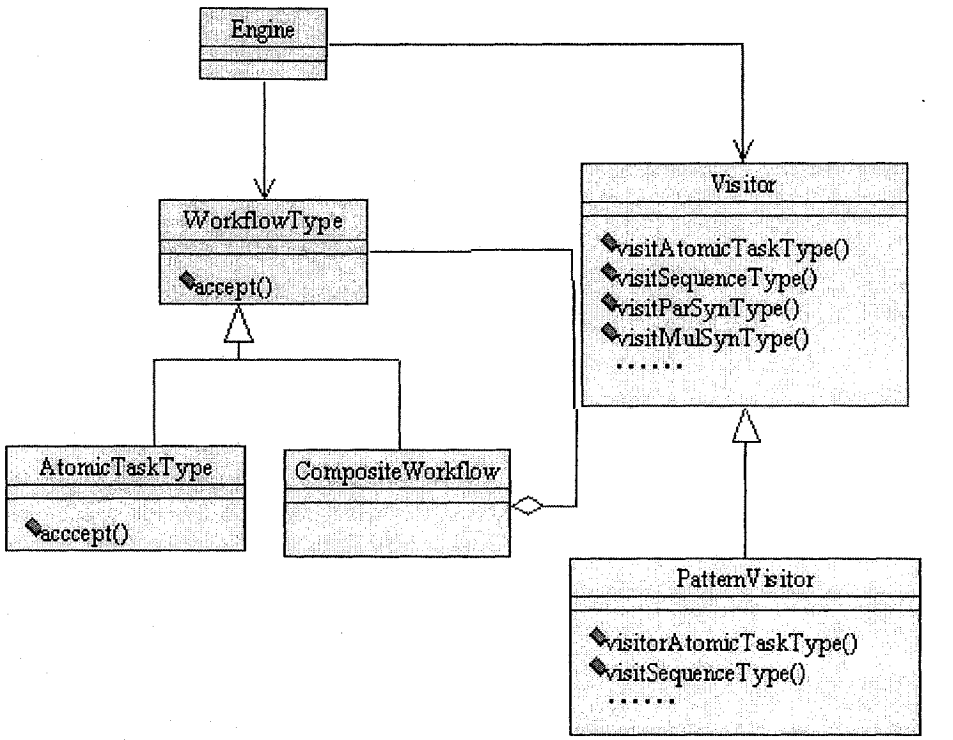

Figure 3-13 Visitor Design Pattern in the Workflow Engine

For the sake of the operation performed on the composite structured tree, the nodes representing composite tasks, patterns, conditions and individual tasks should be treated differently. With Visitor design pattern, we declare PatternVisitor as a concrete visitor class to inherit the abstract Visitor class. PatternVisitor is responsible for visiting all the nodes of the generated ParseTree, and handles the operation on the node elements. In Figure 3-13, we define a Visitor abstract class to provide the interface for all concrete

visitors and also define operations for each concrete element in our composite tree. This abstract class can decide which concrete elements are going to be visited later. The concrete visitor, which is defined as PatternVisitor in this thesis, overrides and implements all the operations in Visitor class for the corresponding concrete element classes in the ParseTree. Each concrete class in the parse tree structure implements an *accept* operation where the related method in the PatternVisitor is executed. When the PatternVisitor visits the ParseTree, the visitor constructs the corresponding threads making up the ThreadGroupManager subsystem. We will explain the generation course of these threads in the next section.

3.3.2.4 ThreadGroupManager Subsystem

The ThreadGroupManager subsystem in Figure 3-14 is made up of controller threads and broker threads. The controller is only responsible for managing the execution order of the workflow. Every time the controller decides it is the time for one or some tasks to be executed, it will ask its task broker or their task brokers to enable the execution.

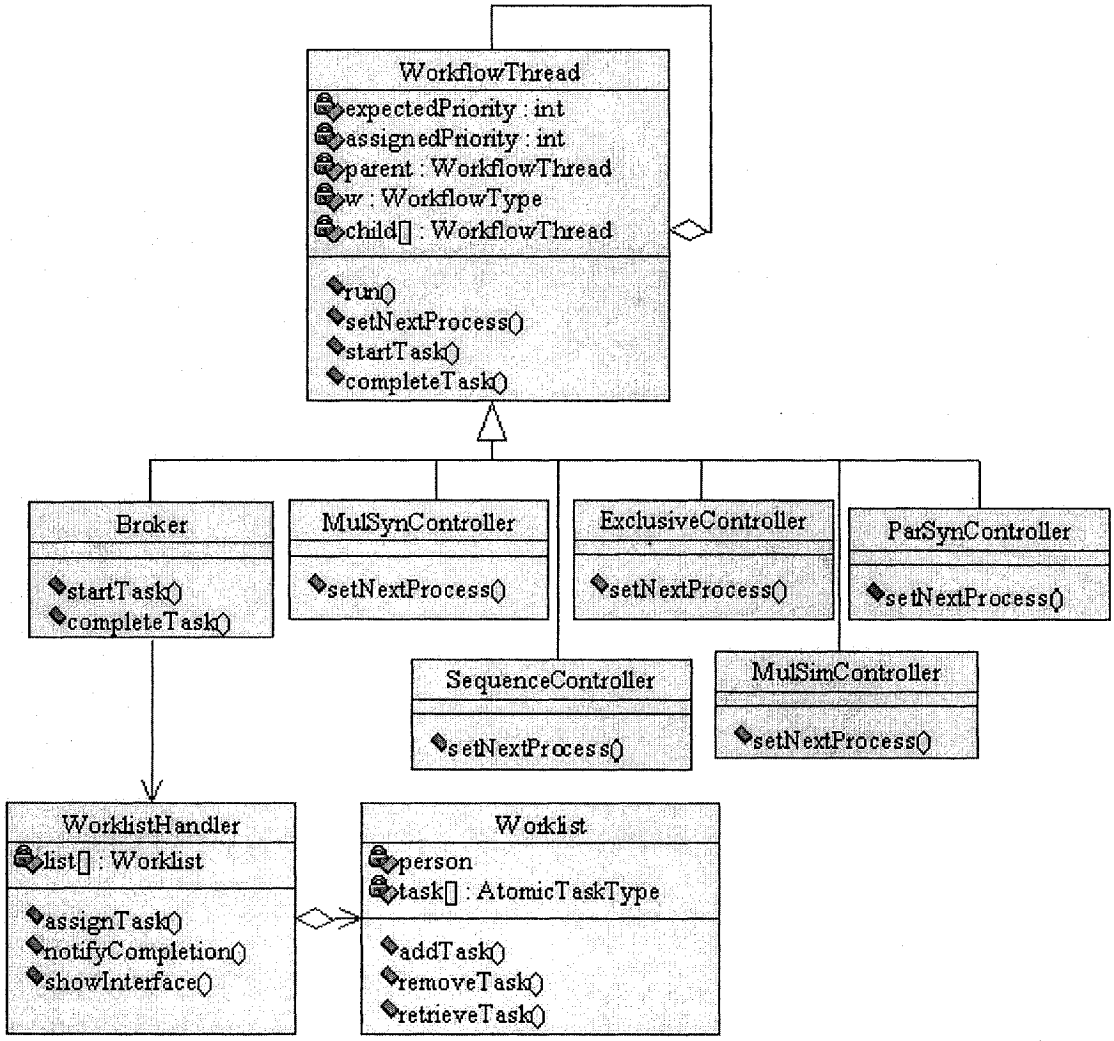

Figure 3-14 Class Diagram for the ThreadGroupManager Subsystem

When the pattern nodes and atomic task nodes in the ParseTree are accessed, the visit method will construct corresponding threads. For example, when the Sequence pattern node is visited, a controller thread called "SequenceController" is constructed. When the AtomicTask type node is visited, a broker thread whose name is the same as the name of the atomic task is constructed. Each broker is related to an atomic task in the workflow specification. The semantics of the Sequence pattern, the Exclusive Choice pattern and the other three grouped workflow patterns such as the group of the Parallel Split pattern and the Synchronization pattern, the group of the Multiple Choice pattern

and the Synchronizing Merge pattern and the group of the Multiple Choice pattern and the Simple Merge pattern are implemented by their corresponding controller threads. The controllers are defined as parent nodes in a recursive hierarchy thread structure. Their child threads can be either a broker or a controller. If this child relates to an atomic task in the workflow specification, it is a broker thread. If it is related to a composite task in the workflow specification, the child thread is a controller, which can also coordinate the execution of its child threads within the composite task. In the following paragraph, we will introduce the details of the ThreadGroupManager subsystem.

1. Broker thread: The broker thread in Figure 3-14 acts as intermediary between our workflow engine and the tasks executed at runtime. In the engine, the controller is only responsible for managing the execution order of the workflow. Every time the controller decides it is the time for one or some tasks to be executed, it will ask its task broker or their task brokers to enable the execution. The *startTask()* operation in the broker is responsible for enabling the execution of the corresponding task based on the interface information of the atomic task in the workflow specification. Once the execution is over, the broker will ask its controller to deal with other tasks waiting for execution. The *completTask()* in the broker thread is responsible for this interaction.

In the workflow specification, the task can be modeled as either automatic tasks without human interaction or tasks associated with human participants. If the task is specified as an automatic task without human interaction, the broker will invoke the corresponding application function directly for the execution of the task. In our test cases, each time the engine enables a task, a new thread instance will be constructed to execute the task's corresponding application function. In order to show that the execution order

conforms to the workflow specification in YAWL, each application function should print its starting time and end time. If the task is modeled as a task associated with a human participant, the broker will pass the task interface information to the WorklistHandler, which provides a user interface for the task participant to invoke the execution of the corresponding application function that requires human interaction.

2. WorklistHandler: The WorklistHandler class in Figure 3-14 provides a user interface for the user to view and operate his assigned task. Through this interface, the task participant can invoke the execution of the corresponding application function that requires human interaction, and the user can notify the engine of the completion of the task once the application function is over. The *assignTask()* method is called by the broker, when the task is enabled. This method is responsible for adding the interface information of the assigned task into the Worklist and showing a user interface with the information in the worklist. Through this interface, the participant can view and start the execution of the application function corresponding to the assigned task. The *notifyCompletion()* method is called when the participant notifies the completion of the task through the user interface. This method will ask Worklist to remove the corresponding task information and notify the completion to the task broker, which is responsible for the enabling of the task.

3. Worklist: Each task participant *(person)* has a Worklist. At runtime, the WorklistHandler passes the interface information of the assigned task to the corresponding Worklist. The *addTask()* method is called by the WorklistHandler to add interface information of the assigned task into the Worklist. (The list *task[]* in the Worklist is used to store the interface information of each assigned task.) The

removeTask() method is called by the WorklistHandler to remove the corresponding task information from the Worklist when the participant notifies the completion of the application function. The *retrieveTask()* method is called by the user interface for the task participant to view all his assigned tasks in the Worklist.

4. Controller threads: In Figure 3-14, the SequenceController, ParSynController MulSynController, MulSimController, and ExclusiveController are defined respectively as controller threads to implement the specification of the Sequence pattern, the group of the Parallel Split pattern and the Synchronization pattern, the group of the Multiple Choice pattern and the Synchronizing Merge pattern, the group of the Multiple Choice pattern and the Simple Merge pattern, as well as the Exclusive Choice pattern. For instance, the MulSynController class is responsible for the management of all its child threads. The controller threads schedule the processing by defining a logical priority policy for each child thread. According to the semantics of different workflow patterns, the logical priority of each child thread is assigned to the child thread, and it is kept in its child's *assignedPriority* attributes. The controller selects one child thread or some child threads to be executed by analyzing the *assignedPriority* at runtime. If the assigned execution order of the child threads matches the execution order expected by the controller at runtime, the corresponding child threads can be triggered. The expected execution order is kept in the *expectedPriority* attribute and is managed by controllers. In order to coordinate the execution of their child threads, the controllers update the value of the *expectedPriority* attribute. Each time the child thread notifies its completion, then the controller schedules next child thread by finding child threads whose *assignedPriority* equals the *expectedPriority*, and so on, until all its child thread have been dealt with.

In the case of a Sequence pattern, the execution order of each child thread ranges from 1 to n (the total number of child threads under the control of the SequenceController thread). Generally, if the value of the *assignedPriority* of the child thread is equal to the *expectedPriority* of the controller, this thread can be executed. After the child thread has been executed, the controller adds *expectedPriority* by 1. Then it will schedule the other child threads whose *assignedPriority* is equal to the *expectedPriority* to run. In this way, the controller keeps scheduling its child threads until there are no more child threads whose *assignedPriority* equals the *expectedPriority* value, which means that all the child threads have been executed. For example, when the SequenceController thread's *expectedPriority* becomes one, only the child thread whose *assignedPriority* is equal to one, can qualify to be triggered. When the corresponding child finishes its execution, it will call its parent to add *expectedPriority* by 1. Then the controller sets its *expectedPriority* to 2 and starts the corresponding child thread whose *assignedPriority* is equal to two. (If this child thread is not a broker but a controller, which is related to a composite task in the workflow specification, its execution is completed once all its children have been executed. Then it will notify its parent controller to add the *expectedPriority* by 1 and schedule other child threads' whose *assignedPriority* is equal to the *expectedPriority* to run. In this way, the controller coordinates its children's execution recursively.)

In order to handle the group of the Parallel Split pattern and the Synchronization pattern, only three levels of priority are specified. The child thread corresponding to the andSplitTask in the workflow specification will be invoked first by the ParSynController, because the value of *assignedPriority* in the andSplitTask thread is 1, and it is equal to

the *expectedPriority* of the controller (the initial value of the *expectedPriority* is equal to 1). After its execution, the controller adds the *expectedPriority* by 1. All the parallel child threads will be triggered, because they have the same *assignedPriority* value of 2 which is equal to the value of controller's *expectedPriority*. When all the parallel threads have completed their execution, the controller adds the *expectedPriority* by 1. The child thread corresponding to the andJoinTask is started, because its *assignedPriority* is 3 which is equal to the value of the controller's *expectedPriority*. Thus, multiple parallel threads having the same priority can run concurrently. Based on this strategy, when child threads finish their tasks, they will call their controller to add the *expectedPriority* by 1 and then start the next thread, whose *assignedPriority* is equal to the *expectedPriority*.

Moreover, for the implementation of the group of the Multiple Choice pattern and the Synchronizing Merge pattern, the OrSplitTask broker thread is first invoked. The orSplitTask is responsible for making multiple choices of the next parallel child threads (its *assignPriority* is 1). After its execution, these selected parallel threads can begin to work (their *assignPriority* are 2). After all those parallel threads complete, the last step is the OrJoinTask broker thread (its *assignPriority* is 3).

For the implementation of the group of the Multiple Choice pattern and the Simple Merge pattern, the first step is to start the OrSplitTask Broker (its *assignPriority* is 1), after whose execution, the selected parallel threads are triggered (their *assignPriority* is 2). The last step is the XorJoinTask Broker (its *assignPriority* is 3), which is triggered by the completion of the fastest parallel thread. According to the specification of the Simple Merge pattern, the controller will kill other parallel threads.

For the implementation of the Exclusive Choice pattern, after execution of the XorSplitTask Broker (its *assignPriority* is 1), only one child thread is selected, which is then executed (its *assignPriority* is 2).

Based on this logical priority policy, each controller has its own strategy to handle the execution order of its child threads. The above strategies are defined in the synchronized method *setNextProcess()* in different controller WorkflowThread classes. This method can invoke the execution of the child threads whose *assignedPriority* value is equal to the value of the *expectedPriority*. According to the concept of polymorphism in Java, "when a request is made through a super class reference to use a method, Java chooses the correct overridden method polymorphically in the appropriate subclass associated with the object" [9]. In Figure 3-14, each child thread declares a *parent* attribute, which type is declared as the abstract super class "WorkflowThread". At runtime, each child thread's *parent* is referred to a corresponding controller. The *setNextProcess()* is defined as an abstract method in the abstract super class. Each controller thread class overrides this method to implement its own strategy. During the execution, when each child thread has finished its task, it sends a request to its *parent* to use the *setNextProcess()* method and Java chooses the overridden method in the corresponding controller in accordance with the property of polymorphism.

5. A sequence diagram for the group of the Parallel Split pattern and the Synchronization pattern: The UML sequence diagram in Figure 3-15 shows how the ParSynController implements the group of the Parallel Split pattern and the Synchronization pattern to manage the execution order of its child threads at runtime.

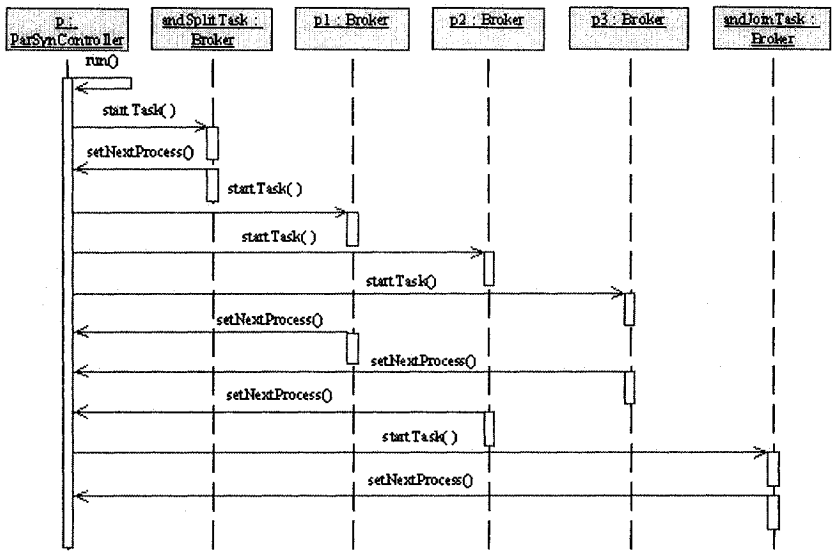

Figure 3-15 Sequence Diagram for the Group of the Parallel Split Pattern and the Synchronization Pattern

In Figure 3-15, the ParSynController 'p' coordinates the execution of all its child threads: such as the Broker thread ('andSplitTask') whose *assignedPriority* value is 1, three parallel Broker threads ('p1', 'p2', 'p3') whose *assignedPriority* value is 2, and the Broker thread ('andJoinTask') whose *assignedPriority* value is 3. The initial value of the *expectedPriority* is 1. After the *run()* method of the controller 'p' is invoked, it will call the *setNextProcess()* method to select the first step for the execution. The controller will start the 'andSplitTask', because its *assignedPriority* value is equal to the value of the *expectedPriority*. After 'andSplitTask' has entered the end state, it has to notify the controller to set the next process, in which the value of *expectedPriority* is increased by 1 and then the controller schedules the next task or tasks whose *assignedPriority* value is equal to the *expectedPriority* value. Thus, the controller can recognize that it is the time for all the parallel child threads to start and consequently, 'p1', 'p2' and 'p3' are invoked. After their completion, these threads also have to call the controller's *setNextProcess()*

method, in which the value of the *expectedPriority* is increased by 1 and then the controller starts the next task or tasks whose *assignedPriority* value is equal to the *expectedPriority* value. The controller invokes the last thread 'andJoinTask', because its *assignedPriority* value is 3, which is equal to the value of the *expectedPriority*. The *setNextProcess()* will be called again once the 'andJoinTask' has been completed. Thus the controller can make sure that all those child threads under its coordination finish, because there are no more threads whose *assignedPriority* is equal to the *expectedPriority* value.

6. Java synchronization technology is used in our implementation: we use a controller (parent) thread to schedule its child threads. These children should call back their controller (parent) to check and modify the execution order information once their execution status is over. As we have indicated above, in the former paragraphs, those order information (for example, *expectedPriority*) is used for the controller to analyze and decide the execution order of its child threads at runtime. In order to avoid two or more concurrent threads accessing the same data in the controller at the same time, we use the Java synchronization technology in the implementation of the ThreadGroupManage subsystem. When child threads acquire a resource in their controller, their data accessing is synchronized (see Figure 3-16).

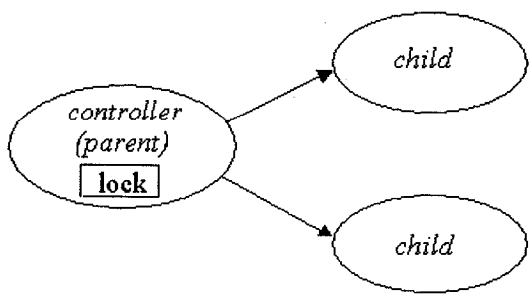

Figure 3-16 Java Synchronization Technology in Our Implementation

In Java, the keyword *synchronized* [9] is used to identify a block of code or a method as a critical section. (A critical section is a segment of code in which shared variables are modified.) When a child thread begins executing a *synchronized* method on the data element in the controller, the synchronization makes it impossible for other threads to invoke the data object, until the object is unlocked. The lock is released only when the method has been executed or if the thread executing the method invokes the wait method in Java. In this way, the execution order information in the controller is avoided to be in an inconsistent state. There will never be a situation where two child threads attempt to acquire a resource directly at the same time.

3.4 Test of the Prototype Workflow Engine

We have employed the Sequence pattern, the Parallel Split pattern, the Synchronization pattern, the Exclusive Choice pattern, the Simple Merge pattern, the Multiple Choice pattern and the Synchronizing Merge pattern in the XML workflow schema. Based on the specification of this XML schema, the user can define the workflow specification in the XML workflow document, which is the input file of our system. The purpose of our testing is to test that our workflow engine can parse the workflow specification in the XML workflow document, interpret the control flow specification, and manage the execution order of the workflow correctly.

3.4.1 What do We Test?

First of all, we test five workflow samples, only including each of the Sequence pattern, the Exclusive Choice pattern, the group of the Parallel Split pattern and the Synchronization pattern, the group of the Multiple Choice pattern and the Simple Merge pattern, as well as the group of the Multiple Choice pattern and the Synchronizing Merge

pattern, respectively, to demonstrate the support of our engine. Furthermore, we use an XML workflow document containing the combination of all seven workflow patterns as the input file of our system. This testing is to test that our engine can deal with the workflows consisting of these seven workflow patterns in the recursive hierarchy structure. The last testing case is to test whether the engine can deal with a workflow that needs human interaction. When the engine enables a task that requires human interaction, the worklist handler will add the interface information of the task into the corresponding user's work list. Then the handler will provide a user interface with the information in the worklist. Through the user interface, the user can view his assigned tasks and operate the execution of the corresponding application functions. The operations include starting the execution of the corresponding application function for the task and notifying its completion to the engine once the application function is over.

3.4.2 How do We Simulate Application Functions?

In our testing, we use the XML workflow document conforming to the XML workflow schema as the input file of our system (XML SPY 5.0 is used as the XML editing tool to specify the XML workflow documents). Each time the engine enables an atomic task, a new thread instance will be constructed to execute the task's corresponding application function.

In order to show that the execution order conforms to the workflow specification in YAWL, the application function should print its starting time and end time. As the task of each application function is domain specific, and this thesis focuses on the control flow perspective of workflow, we use one print log function to stand for all of the different application functions. Therefore, in the XML workflow document of every test

case, the name of the corresponding application functions are specified as "print log", which is the names of the print log function. However, users in a specific domain can program their application libraries, when they develop the actual workflow management system based on this prototype, and they can specify the name of the corresponding application functions in the "automation" element of each atomic task.

3.4.3 How do We Simulate the Selection Course for the OrSplitTask?

According to YAWL's specification, the orSplitTask has to select parts or the whole of the parallel tasks for the next execution. Since the selection course for the orSplitTask is domain specific, we implement a random function called "or split random function" to simulate this selection course in our tests. The name of this selection function is specified as "or split random function" in the "automation" element of the orSplitTask. When the engine interprets the orSplitTask, a new thread instance will be constructed to execute this selection function. If the orSplitTask is modeled as an automatic task and its selection function is the "or split random function", this function will randomly select parts or whole of the parallel tasks for the next execution. However, the developers can implement their domain specific selection functions to replace this random function, while developing a workflow management system based on this prototype. If the orSplitTask is a task that requires human interaction, this function will provide a user interface for the user to select the parallel tasks for future execution. In this case, the user can make the decision.

3.4.4 How do We Simulate the Selection Course for XorSplitTask?

According to YAWL's specification, the xorSplitTask has to select only one of the alternative tasks for the next execution. By using a similar approach, we implement

another random function called "xor split random function" to simulate this selection course. In our test, if the xorSplitTask is modeled as an automatic task, the random function "xor split random function" will randomly select only one of the alternative tasks for the next execution. If the xorSplitTask is a task associated with a human participant, this function will also provide a user interface for the user to make a selection. In this case, the user makes the decision.

The selection result of these two random functions will be demonstrated in the log file, and they will print their starting time and end time. We format the time information as hour: minute: second: millisecond.

3.4.5 Test of the Sequence Pattern

Objectives: In this case, we test the Sequence pattern. The control flow in the workflow specification is specified only through the Sequence pattern. The engine should parse and interpret the workflow specification in the XML workflow document correctly. According to the specification of the Sequence pattern, those tasks modeled within the Sequence pattern should be executed in a sequential order. The log file is used as an output of this test case to demonstrate the execution order of each task.

Testing Sample: We specify the workflow specification consisting of a single Sequence pattern in an XML workflow document as the input file of our system. The process is called "SequenceSample". Three atomic tasks called "Task A", "Task B" and "Task C" are linked as a Sequence pattern in the workflow specification. These tasks are all automatic tasks without human interaction. Their corresponding application functions are all modeled as "print log". The input file of this test case is in Appendix A.

Results: The following log file in Figure 3-17 demonstrates the execution order of each task in this testing. The log information shows that "Task B" is enabled after the completion of "Task A", and "Task C" follows "Task B" (sequence pattern). This result shows that the engine interprets the Sequence pattern and manages the execution order correctly.

```
Process (SequenceSample) starts to process at 15:26:44:301
Sequence pattern starts to process
Sequence pattern is processing
     AtomicTask (Task A) starts to process at 15:26:44:312
     AtomicTask (Task A) is over at 15:26:44:429
     AtomicTask (Task B) starts to process at 15:26:44:595
     AtomicTask (Task B) is over at 15:26:44:716
     AtomicTask (Task C) starts to process at 15:26:44:772
     AtomicTask (Task C) is over at 15:26:44:923
Sequence pattern is over
Process (SequenceSample) is over at 15:26:45:26
```

Figure 3-17 Log File of the Test for the Sequence Pattern

3.4.6 Test of Group of the Parallel Split Pattern and the Synchronization Pattern

Objectives: In this case, we test the group of the Parallel Split pattern and the Synchronization pattern. The input file of this test is an XML workflow document, whose control flow is specified only through the group of these two patterns. During our test, the engine should parse and interpret the specification in the XML document correctly. According to the specification of the group of the Parallel Split pattern and the Synchronization pattern, each of the parallel tasks defined in the workflow specification should be executed in parallel and their execution should be synchronized at runtime. The log file is used as an output of this test case to demonstrate the execution order of each task.

Testing Sample: The XML workflow document in Appendix B is used as an input file for this testing. In this XML document, the control flow perspective is specified only with the group of the Parallel Split pattern and the Synchronization pattern. The process is called "ParSynSample". "Task A" is defined as andSplitTask. "Task B", "Task C' and

"Task D" are defined as parallel tasks in the workflow specification. "Task E" is defined as andJoinTask. These tasks are all automatic tasks, which do not require human interaction. Their corresponding application functions are all modeled as "print log".

Results: The log information in Figure 3-18 demonstrates the execution order of the automatic tasks in this testing.

```
Process (ParSynSample) starts to process at 16:5:52:443
Group of Parallel Split pattern and Synchronization pattern starts to process
Group of Parallel Split pattern and Synchronization pattern is processing
And split Task (Task A) starts to process at 16:5:52:443
      And split Task: (Task A) is over at 16:5:53:886
      AtomicTask (Task B) starts to process at 16:5:53:907
      AtomicTask (Task C) starts to process at 16:5:53:907
      AtomicTask (Task D) starts to process at 16:5:53:907
      AtomicTask (Task D) is over at 16:5:55:102
      AtomicTask (Task C) is over at 16:5:55:336
      AtomicTask (Task B) is over at 16:5:56:225
      And join Task (Task E) starts to process at 16:5:56:301
      And join Task: (Task E) is over at 16:5:57:277
Group of Parallel Split pattern and Synchronization pattern is over
Process (ParSynSample) is over at 16:5:57:280
```

Figure 3-18 Log File of the Test for the Group of the Parallel Split Pattern and the Synchronization Pattern

In the log file, we can see that after the execution of "Task A", all the parallel tasks are begun and they are executed simultaneously at runtime (Parallel Split pattern). "Task B", "Task C" and "Task D" are all synchronized during the execution of this workflow. The andJoinTask "Task E" is triggered after all the parallel tasks have been executed (Synchronization pattern). The testing log information demonstrates that the engine interprets the group of the Parallel Split pattern and the Synchronization pattern correctly and manages the execution order correctly.

3.4.7 Test of Group of the Multiple Choice Pattern and the Synchronizing Merge Pattern

Objectives: In this case, we test the group of the Multiple Choice pattern and the Synchronizing Merge pattern. The input file of this experiment is an XML workflow

document, whose control flow is specified only by the group of these two patterns. Our engine should parse and interpret the specification in the input file correctly. According to the specification of the group of the Multiple Choice pattern and the Synchronizing Merge pattern, after the orSplitTask has selected parts or the whole of the parallel tasks for future execution, only those selected tasks can be enabled. The engine can trigger the orJoinTask only when all of these selected parallel tasks have completed their execution.

Testing Sample and methods: The XML workflow document in Appendix C is used as an input file for this testing. The control flow perspective in this XML document is specified as the group of the Multiple Choice pattern and the Synchronizing Merge pattern. The process is called "MulSynSample". "Task A" is defined as an orSplitTask". "Task B", "Task C" and "Task D" are defined as parallel tasks. "Task E" is defined as an orJoinTask. All these tasks are automatic tasks without human participants. In this test case, the corresponding selection function for the orSplitTask "Task A" is specified as "or split random function" in the "automation" element. The corresponding application functions for other tasks are all modeled as "print log". In our demonstration, all of these application functions will print their starting time and end time. The selection result of the "or split random function" will also be demonstrated in the log file.

Results: The following log information in Figure 3-19 demonstrates the execution order of this test case. In the log file, we can see that during the execution of the orSplitTask, "Task A" selects "Task B" and "Task D" for their future execution. When "Task A" is over, these two parallel tasks are started by the engine (Multiple Choice pattern). Once these two tasks have been completed, the orJoinTask, "Task E", is triggered by the engine (Synchronizing Merge pattern). This process is completed after

"Task E" is over. From this log information, we can see that the engine synchronizes all the selected parallel tasks at runtime. These testing results indicate that the engine interprets the group of the Multiple Choice pattern and the Synchronizing Merge pattern and manages the execution order correctly.

```
Process (MulSynSample) starts to process at 21:46:5:87
Group of Multiple Choice and Synchronizing Merge pattern starts to process
Group of Multiple Choice and Synchronizing Merge pattern is processing
      Or split Task (Task A) starts to process at 21:46:5:87
ParallelTask (Task B)is selected
ParallelTask (Task C)is not selected
ParallelTask (Task D)is selected
      Or split Task (Task A) is over at 21:46:6:268
      AtomicTask (Task B) starts to process at 21:46:6:278
      AtomicTask (Task D) starts to process at 21:46:6:278
      AtomicTask (Task B) is over at 21:46:7:309
      AtomicTask (Task D) is over at 21:46:7:429
      Or join Task (Task E) starts to process at 21:46:7:515
      Or join Task: (Task E) is over at 21:46:8:331
Group of Multiple Choice and Synchronizing Merge pattern is over
Process (MulSynSample) is over at 21:46:8:370
```

Figure 3-19 Log File of the Test for the Group of the Multiple Choice Pattern and the Synchronizing Merge Pattern

3.4.8 Test of the Exclusive Choice Pattern

Objectives: In this case, we test the Exclusive Choice pattern. The input file of this test is an XML workflow document, whose control flow is specified only through the Exclusive Choice pattern. The engine should parse and interpret the specification of this input XML document correctly. According to the specification of the Exclusive Choice pattern, the xorSplitTask chooses only one of those alternative tasks in the workflow specification for the future execution. Then the selected task can be triggered.

Testing Sample: The XML workflow document in Appendix D is used as the input file for this testing. The process defined in this sample only consists of the Exclusive Choice pattern. The process is called ExclusiveSample. "Task A" is defined as an xorSplitTask. "Task B", "Task C" and "Task D" are defined as the alternative tasks, out of which only one will be chosen by the xorSplitTask at runtime. All these tasks are

automatic tasks without human interaction. In this test case, the corresponding selection function for the xorSplitTask "Task A" is specified as "xor split random function" in the "automation" element. The corresponding application functions for other tasks are all modeled as "print log". In our demonstration, all these application functions will print their starting time and end time. The selection result for the "xor split random function" will also be demonstrated in the log file.

Results: The following log file in Figure 3-20 demonstrates that the "Task A" thread only selects "Task D" for the future execution at runtime. Consequently, only "Task D" is invoked by the engine (Exclusive Choice pattern). After the completion of "Task D", this process is executed. This test result indicates that the workflow engine interprets the Exclusive Choice pattern and manages the execution order correctly.

```
Process (ExclusiveSample) starts to process at 22:24:46:805
Exclusive Choice pattern starts to process
Exclusive Choice pattern is processing
      Xor split Task (Task A) starts to process at 22:24:46:855
ExclusivedTask (Task B)is not selected
ExclusivedTask (Task C)is not selected
ExclusivedTask (Task D)is selected
      Xor split Task: (Task A) is over at 22:24:48:176
      AtomicTask (Task D) starts to process at 22:24:48:586
      AtomicTask (Task D) is over at 22:24:50:12
Exclusive Choice pattern is over
Process (ExclusiveSample) is over at 22:24:50:12
```

Figure 3-20 Log File of the Test for the Exclusive Choice Pattern

3.4.9 Test of Group of the Multiple Choice Pattern and the Simple Merge Pattern

Objectives: In this testing case, we test the group of the Multiple Choice pattern and the Simple Merge pattern. The input file of this testing is an XML workflow document, whose control flow is specified only through the group of these two patterns. The engine should parse and interpret the specification in the input file correctly. According to the specification of the group of the Multiple Choice pattern and the Simple Merge pattern, after the orSplitTask selects parts or whole of the parallel tasks for future execution, the corresponding tasks are triggered. However, the engine will not

synchronize all of these parallel tasks. It only waits for the first one to be accomplished. This completion will make the engine trigger the xorJoinTask, while the execution of the other started parallel tasks is ignored by killing the corresponding threads.

Testing Sample: The XML workflow document in Appendix E is used as an input file for this testing. The process defined in this sample only consists of the group of the Multiple Choice pattern and the Simple Merge pattern. The process is called "MulSimSample". "Task A" is defined as orSplitTask. "Task B", "Task C" and "Task D" are defined as alternative tasks. "Task E" is defined as the xorJoinTask. All these tasks are automatic tasks without human interaction. In this test case, the corresponding selection function for the orSplitTask "Task A" is specified as the "or split random function" in the "automation" element. The corresponding application functions for the other tasks are all modeled as "print log". In the demonstration, all these functions will print their starting time and end time. The selection result of the "or split random function" will also be demonstrated in the log file.

Results: The following log file in Figure 3-21 demonstrates that "Task A" selects "Task C" and "Task D" for future execution.

```
Process (MulSimSample) starts to process at 23:41:11:768
Gorup of Multiple Choice and Simple Merge pattern starts to process
Gorup of Multiple Choice and Simple Merge pattern is processing
      Or split Task (Task A) starts to process at 23:41:11:768
ParallelTask (Task B)is not selected
ParallelTask (Task C)is selected
ParallelTask (Task D)is selected
      Or split Task (Task A) is over at 23:41:12:309
AtomicTask (Task C) starts to process at 23:41:12:359
AtomicTask (Task D) starts to process at 23:41:12:359
      AtomicTask (Task D) is over at 23:41:13:101
      Xor join Task (Task E) starts to process at 23:41:13:170
      Xor join Task: (Task E) is over at 23:41:14:542
Gorup of Multiple Choice and Simple Merge pattern is over
Process (MulSimSample) is over at 23:41:14:582
```

Figure 3-21 Log File of the Test for the Group of the Multiple Choice Pattern and the Simple Merge Pattern

When "Task A" is over, "Task C" and "Task D" are started by the engine in parallel (Multiple Choice pattern). The log file shows that as soon as "Task D" finishes, "Task E" is triggered (Simple Merge pattern). The engine ignores the execution of "Task C" because "Task C" is slower than "Task D". The process is completed when "Task E" is over. This log information demonstrates that the engine interprets the group of the Multiple Choice pattern and the Simple Merge pattern and manages the execution order correctly.

3.4.10 Test of the Workflow Sample Consisting of All Seven Workflow Patterns

Objectives: In this test case, we test the combinations of all the seven workflow patterns in the recursive hierarchy structure. The input file of this testing is an XML workflow document, whose control flow is specified by the combinations of all the seven workflow patterns in the recursive hierarchy structure. The engine should parse the XML workflow document, interpret the specification in the input file, and manage the execution order correctly.

Testing Sample: The XML workflow document in Appendix F is used as the input file for this testing. The process defined in this sample consists of Sequence (Parallel Split and Synchronization (Multiple Choice and Synchronizing Merge (Exclusive Choice (Multiple Choice and Simple Merge)))) patterns in the recursive hierarchy structure. Here we use parentheses to represent the different levels of the hierarchy relationships among the patterns in the XML workflow document, the utter most parenthesis stands for the top level, and the inner most parenthesis stands for the lowest level.

The process is called "seq(psyn(msyn(ex(msim))))". On the top level, the process consists of "seq1", "seq2(psyn)" and "seq3" in the Sequence pattern. On the second level, the composite task "seq2(psyn)" consists of "seq2(psyn_andSplit)" as defined by andSplitTask, "seq2(psyn1)", "seq2(psyn2)" and "seq2(psyn3(msyn))" as defined by parallel tasks as well as "seq2(psyn_andJoin)" as defined by andJoinTask. They are within the group of the Parallel Split pattern and the Synchronization pattern. On the third level, the composite task "seq2(psyn3(msyn))" consists of "seq2(psyn3(msyn_orSplit))" as defined by orSplitTask, "seq2(psyn3(msyn3(ex)))", "seq2(psyn3(msyn1))", "seq2(psyn3(msyn2))" as defined by parallel tasks and "seq2(psyn3(msyn_orJoin))" as defined by orJoinTask. They are all within the group of the Multiple Choice pattern and the Synchronizing Merge pattern. On the fourth level, the composite task "seq2(psyn3(msyn3(ex)))" consists of "seq2(psyn3(msyn3(ex_xorSplit)))" as defined by xorSplitTask, "seq2(psyn3(msyn3(ex1)))" and "seq2(psyn3(msyn3(ex2(msim))))" as defined by alternative tasks. They are within the Exclusive Choice pattern. On the fifth level, the composite task "seq2(psyn3(msyn3(ex2(msim))))" consists of "seq2(psyn3(msyn3(ex2(msim_orSplit)))" as defined by orSplitTask, "seq2(psyn3(msyn3(ex2(msim1)))" and "seq2(psyn3(msyn3(ex2(msim2)))" as defined by parallel tasks, "seq2(psyn3(msyn3(ex2(msim_xorJoin)))" as defined by xorJoinTask. They are within the group of the Multiple Choice pattern and the Simple Merge pattern.

All of these tasks are automatic tasks without human interaction. In this test case, the corresponding selection functions for the orSplitTasks "seq2(psyn3(msyn_orSplit))" and "seq2(psyn3(msyn3(ex2(msim_orSplit)))" are specified as "or split random function" in their "automation" elements. The corresponding selection function for the

xorSplitTask "seq2(psyn3(msyn3(ex_xorSplit)))" is specified as "xor split random function". The corresponding application functions for other tasks are all modeled as "print log". In our demonstration, all of these functions will print their starting time and end time. The selection results of these random functions will also be demonstrated in the log file.

Results: The following log file in Figure 3-22 demonstrates that all of the tasks have been executed in the correct order at different levels.

```
Process seq(psyn(msyn(ex(msim))))starts to process at 0:49:44:348
Sequence pattern starts to process
Sequence pattern is processing
    AtomicTask (seq1) starts to process at 0:49:44:351
  AtomicTask (seq1) is over at 0:49:44:367
Task (seq2(psyn)) starts to process at 0:49:44:391
Group of Parallel Split pattern and Synchronization pattern starts to process
Group of Parallel Split pattern and Synchronization pattern is processing
    And split Task (seq2(psyn_andSplit)) starts to process at 0:49:44:441
    And split Task: (seq2(psyn_andSplit)) is over at 0:49:45:152
    AtomicTask (seq2(psyn1)) starts to process at 0:49:45:182
    AtomicTask (seq2(psyn2)) starts to process at 0:49:45:182
    AtomicTask (seq2(psyn1)) is over at 0:49:45:282
    AtomicTask (seq2(psyn2)) is over at 0:49:45:293
Task (seq2(psyn3(msyn))) starts to process at 00:49:45:503
Group of Multiple Choice pattern and Synchronizing Merge pattern starts to process
    Or split Task (seq2(psyn3(msyn_orSplit))) starts to process at 0:49:45:543
ParallelTask (seq2(psyn3(msyn1))) is selected
ParallelTask (seq2(psyn3(msyn2))) is selected
ParallelTask (seq2(psyn3(msyn3(ex)))) is selected
    Or split Task: (seq2(psyn3(msyn_orSplit))) is over at 0:49:45:559
Task (seq2(psyn3(msyn3(ex)))) starts to process at 0:49:46:254
    AtomicTask (seq2(psyn3(msyn1))) starts to process at 0:49:46:264
    AtomicTask (seq2(psyn3(msyn1))) is over at 0:49:46:302
Exclusive Choice pattern starts to process
    Xor split Task (seq2(psyn3(msyn3(ex_xorSplit)))) starts to process at0:49:46:474
ExclusivedTask (seq2(psyn3(msyn3(ex1)))) is not selected
ExclusivedTask (seq2(psyn3(msyn3(ex2(msim))))) is selected
Exclusive Choice pattern is processing
    AtomicTask (seq2(psyn3(msyn2))) starts to process at 0:49:46:504
    AtomicTask (seq2(psyn3(msyn2))) is over at 0:49:46:656
    Xor split Task: (seq2(psyn3(msyn3(ex_xorSplit)))) is over at 0:49:47:155
Task (seq2(psyn3(msyn3(ex2(msim))))) starts to process at 0:49:47:165
Group of Multiple Choice pattern and Simple Merge pattern starts to process
Group of Multiple Choice pattern and Simple Merge pattern is processing
  Or split Task (seq2(psyn3(msyn3(ex2(msim_orSplit))))) starts to process at 0:49:47:173
ParallelTask (seq2(psyn3(msyn3(ex2(msim1))))) is selected
ParallelTask (seq2(psyn3(msyn3(ex2(msim2))))) is selected
  Or split Task (seq2(psyn3(msyn3(ex2(msim_orSplit))))) is over at 0:49:47:187
  AtomicTask (seq2(psyn3(msyn3(ex2(msim1))))) starts to process at 0:49:47:202
  AtomicTask (seq2(psyn3(msyn3(ex2(msim2))))) starts to process at 0:49:47:222
  AtomicTask (seq2(psyn3(msyn3(ex2(msim1))))) is over at 0:49:47:402
  Xor join Task (seq2(psyn3(msyn3(ex2(msim_xorJoin))))) starts to process at 0:49:47:470
  Xor join Task (seq2(psyn3(msyn3(ex2(msim_xorJoin))))) is over at 0:49:47:476
Group of Multiple Choice pattern and Simple Merge pattern is over
Task (seq2(psyn3(msyn3(ex2(msim))))) is over at 0:49:47:495
Exclusive Choice pattern is over
Task (seq2(psyn3(msyn3(ex)))) is over at 0:49:48:517
    Or join Task (seq2(psyn3(msyn_orJoin))) starts to process at 0:49:48:948
    Or join Task: (seq2(psyn3(msyn_orJoin))) is over at 0:49:49:468
Group of Multiple Choice pattern and Synchronizing Merge pattern is over
Task (seq2(psyn3(msyn))) is over at 0:49:49:789
    And join Task (seq2(psyn_andJoin)) starts to process at 0:49:49:919
    And join Task: (seq2(psyn_andJoin)) is over at 0:49:50:320
Group of Parallel Split pattern and Synchronization pattern is over
Task (seq2(psyn)) is over at 0:49:50:550
    AtomicTask (seq3) starts to process at 0:49:50:550
    AtomicTask (seq3) is over at 0:49:50:550
Sequence pattern is over
Process Seq(Psyn(msyn(ex(msim)))) is over at 0:49:50:610
```

Figure 3-22 Log File of the Test for Combination of Workflow Patterns

On the top level, "seq1", "seq2(psyn)" and "seq3" are executed sequentially based on the specification of the Sequence pattern. On the second level in the composite task of "seq2(psyn)", the execution order of "seq2(psyn_andSplit)", "seq2(psyn1)", "seq2(psyn2)", "seq2(psyn3(msyn))" and "seq2(psyn_andJoin)" follows the specification in the group of the Parallel Split pattern and the Synchronization pattern. On the third level in the composite task of "seq2(psyn3(msyn))", "seq2(psyn3(msyn_orSplit))" enables the execution of "seq2(psyn3(msyn3(ex)))", "seq2(psyn3(msyn1))", and "seq2(psyn3(msyn2))". All these enabled tasks are synchronized by "seq2(psyn3(msyn_orJoin))" based on the specification of the group of the Multiple Choice pattern and the Synchronizing Merge pattern. On the fourth level in the composite task of "seq2(psyn3(msyn3(ex)))", "seq2(psyn3(msyn3(ex_xorSplit)))" only selects "seq2(psyn3(msyn3(ex2(msim))))" at runtime (Exclusive Choice pattern). On the fifth level in the composite task of "seq2(psyn3(msyn3(ex2(msim))))", "seq2(psyn3(msyn3(ex2(msim_orSplit)))" selects "seq2(psyn3(msyn3(ex2(msim1)))" and "seq2(psyn3(msyn3(ex2(msim2)))" for future execution (Multiple Choice pattern). At runtime, "seq2(psyn3(msyn3(ex2(msim1)))" is completed first, then it triggers "seq2(psyn3(msyn3(ex2(msim_xorJoin)))" (Simple Merge pattern). The execution result in the log file demonstrates that the engine interprets the combination of all the seven workflow patterns in the recursive hierarchy structure and manages the execution order correctly.

3.4.11 Test for Workflow Sample with Human Interaction

Objectives: The purpose of this test is to demonstrate that our engine can deal correctly with the workflow involving human interaction. The input file of this testing is

an XML workflow document, in which we specify tasks requiring human participants. The engine should parse and interpret correctly the workflow specification in the XML workflow document and manage the execution order of each task correctly. At runtime, when the engine enables a task that requires human interaction, it will ask the WorklistHandler to add the interface information of this assigned task to the user's worklist. Then the handler will provide a user interface with the information in the worklist. Through the user interface, the user can view and operate his assigned task. The operations include starting the execution of the application function corresponding to the assigned task and notifying completion to the engine once the application function is over.

Sample and methods: The XML workflow document in Appendix G is used as the input file for this testing case. We use the former human resource recruitment example in Section 3.3.1.1 as the input sample for this test case. The process is called "HRrecruitmentSample". The Sequence pattern is on the top level. It consists of an atomic "send resume" with the participant "Secretary" as well as two composite tasks, "ParSynTask" and "ExclusiveTask". The "ParSynTask" consists of the group of the Parallel Split pattern and the Synchronization pattern. "distribute resume" is defined as andSplitTask with the participant "HR Manager". "check (IT Manager)", "check (IT Director)" and "check (HR Manager)" are defined as parallel tasks, with participants "IT Manager", "IT Director" and "HR Manager", respectively. "collect evaluation reports" is defined as andJoinTask with the participant "HR manager". The composite task "ExclusiveTask" consists of an xorSplitTask, "decide" with the participant "HR Manager" and two alternative tasks, "archive" as well as "notify", with "Secretary" and

"Interviewee" as their participants, respectively. The corresponding selection function for the xorSplitTask "decide" is specified as the "xor split random function". (As indicated in Section 3.4.4, if the xorSplitTask is a task that needs human interaction, the "xor split random function" will provide a user interface for the user to make a selection.) The corresponding application functions for other tasks are all modeled as "print log".

Results and demonstration: In the demonstration, the log information of the user's assigned task including the task assignment time as well as the starting time and the end time of the application function corresponding to the task is demonstrated in a table in the user interface. These corresponding functions are modeled as "print log". In the following parts, we display the screenshots of each participant's user interface at runtime. For example, the top window in Figure 3-23 (see next page) shows the time at which the task is assigned to the user. When the user starts the corresponding application function, the starting time is added into the interface of the middle window in Figure 3-23. The end time of the application function is demonstrated in the bottom window. The "START" button in the interface is for the user to invoke the execution of the application function corresponding to the assigned task. The "COMPLETE" button is for the user to notify the engine of the completion of the task when the application function is over. In this test, when the user sees the end time information of the application function demonstrated in the user interface, he can activate the "COMPLETE" button to notify the completion of the task to the system. The demonstration of the HR recruitment sample is as follows:

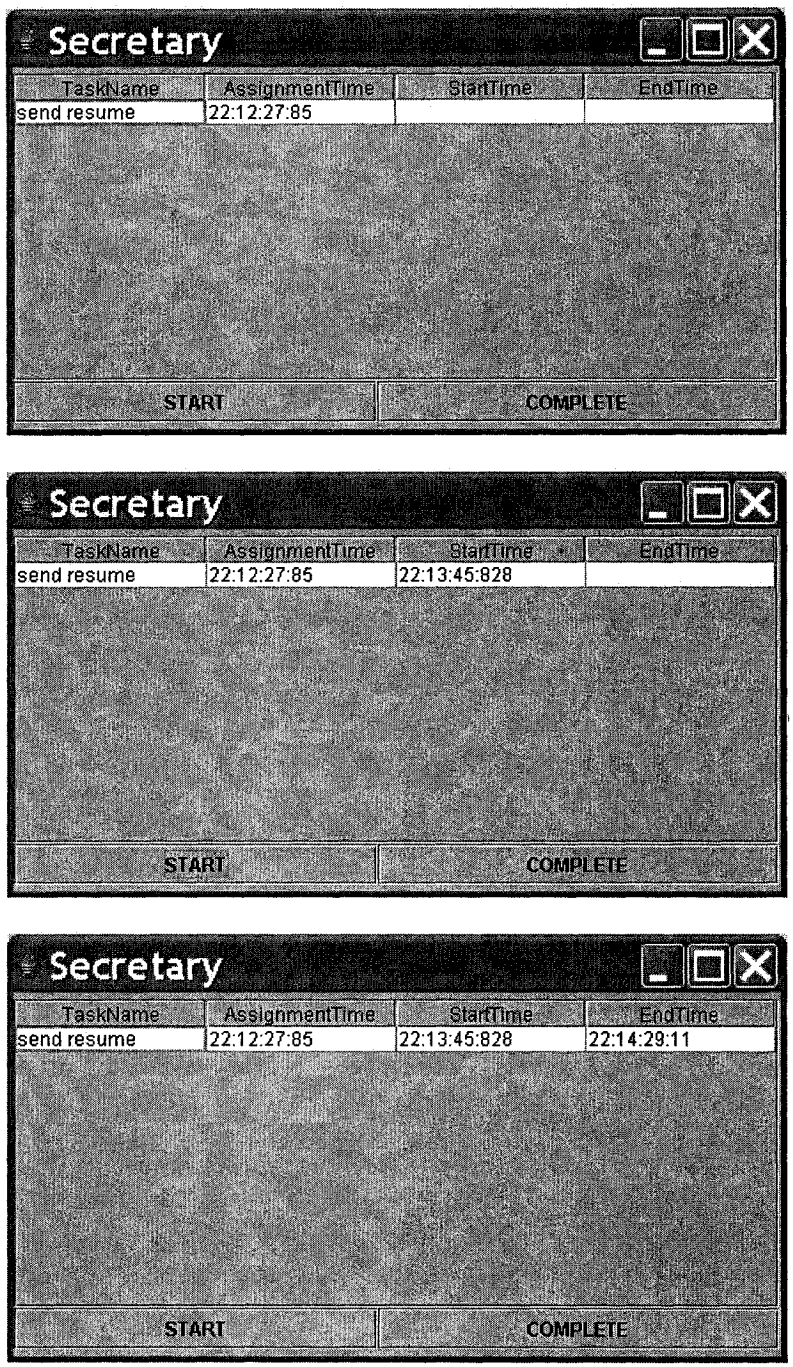

Figure 3-23 Log Information for Task "send resume"

At runtime, when the engine interprets the task "send reseume" in the workflow specification and enables this task, it will assign this task to the WorklistHandler. The handler will add the interface information of this task to the secretary's worklist, and then provide a user interface with the information in the worklist. The secretary can view this

assigned task through the user interface (see the top window in Figure 3-23). Then the user starts the execution of the corresponding application (see the middle window in Figure 3-23). The end time of the application function corresponding to this task is demonstrated in the bottom window in Figure 3-23.

Task "distribute resume" is assigned to the "HR Manager" as shown in Window 1 in Figure 3-24. Then the user starts the corresponding application function. When the application function for this task is over, the user notifies the completion of this task (see Window 3 in Figure 3-24).

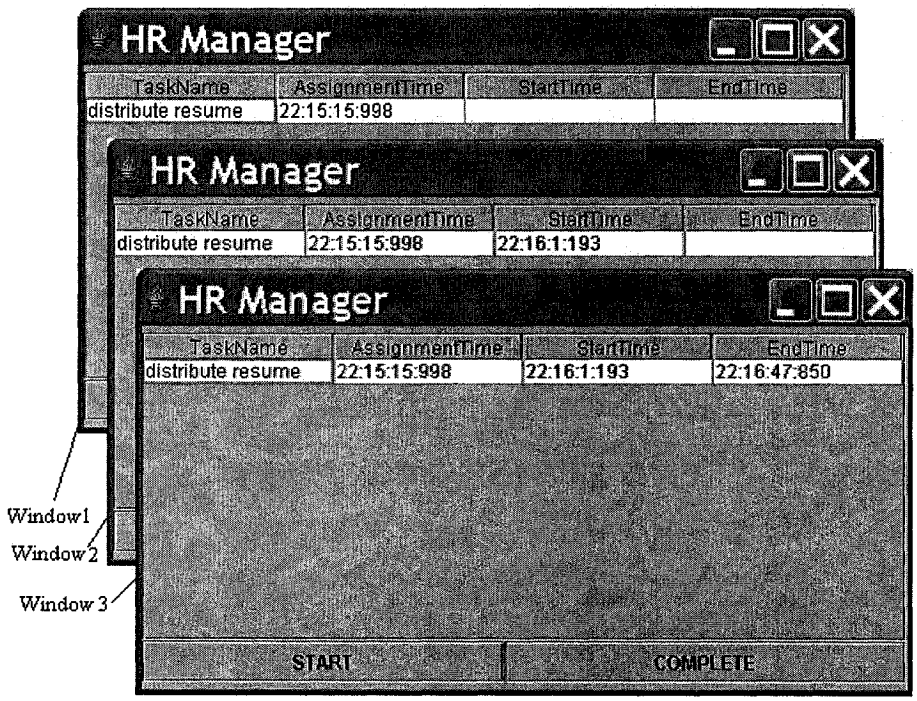

Figure 3-24 Log Information for Task "distribute resume"

Parallel tasks: "check (IT Manager)", "check (IT Director)" and "check (HR Manager)" are assigned to "IT Manager", "IT Director" and "HR Manager" after the execution of "distribute resume" task. These three participants begin their tasks, respectively, and execute their tasks in parallel (see Figure 3-25). The following three screenshots in Figure 3-25 are the user interfaces for "IT Director", "IT Manager" and

"HR Manager". They demonstrate the log information when their tasks have been completed.

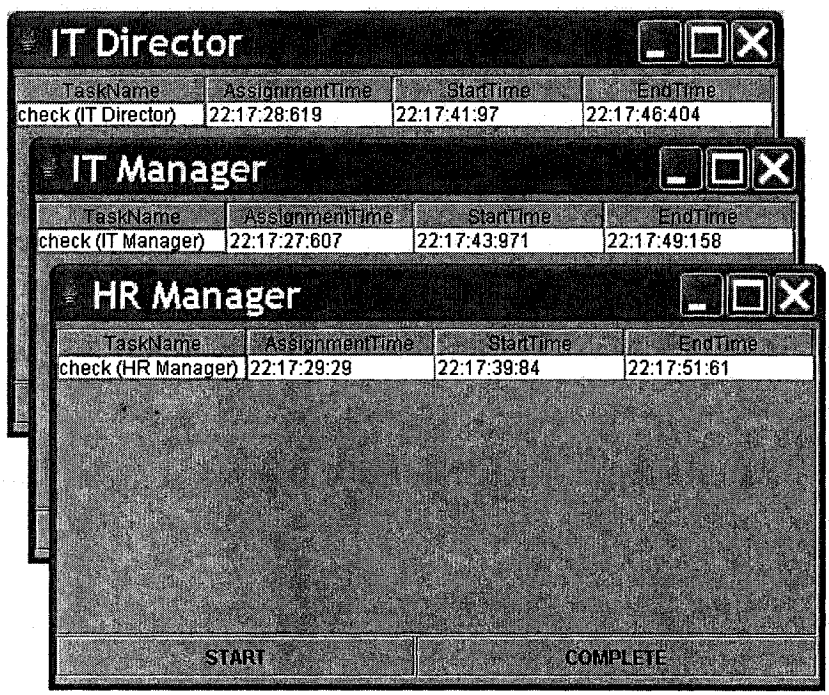

Figure 3-25 Log Information for the Three Parallel Tasks

According to the end time information of each of the parallel tasks in Figure 3-25 and the log information in Figure 3-26, it is clear that task "collect evaluation reports" has been enabled and assigned to "HR Manager" once the parallel tasks "check (IT Manager)", "check (IT Director)" and "check (HR Manager)" have been completed. This result demonstrates that these three parallel tasks have been synchronized by the engine.

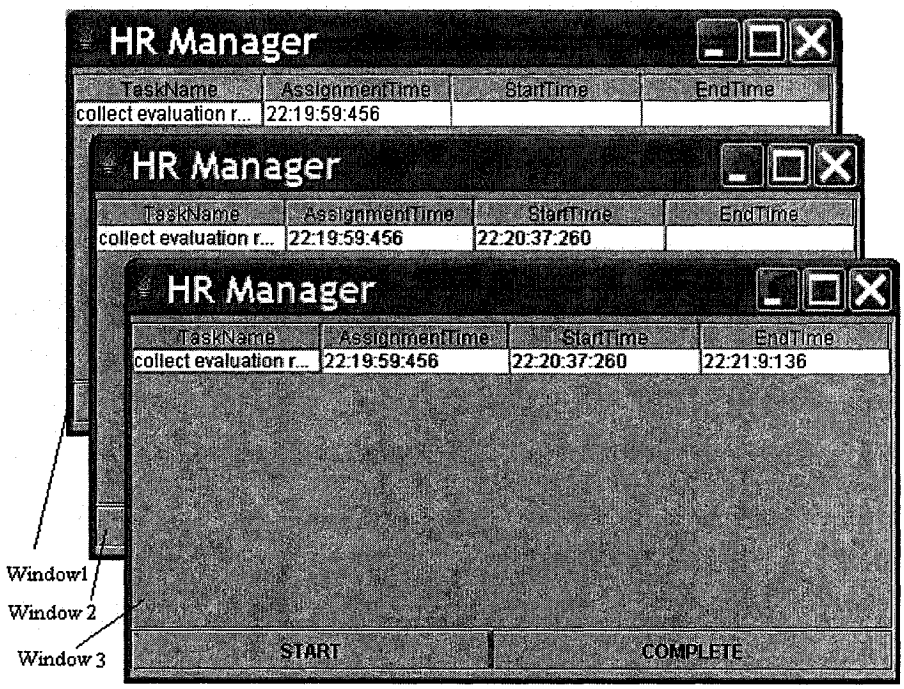

Figure 3-26 Log Information for Task "collect evaluation reports"

According to the workflow specification for this test sample, Task "decide" is for the "HR Manager" to select either task "archive" or task "notify" for future execution. Window 1 in Figure 3-27 shows that "decide" has been assigned to the "HR manager". Window 2 in Figure 3-27 shows that the user has started this assigned task. Since the selection function corresponding to the xor split task "decide" has been specified as the "xor split random function" in the workflow specification, this selection function is invoked when the user clicks the "START" button. The user interface provided by this selection function is shown in Window 3 in Figure 3-27. In this interface, two alternative tasks: "archive" and "notify" are shown so that the user can make a selection. The user can select a task for future execution by clicking the corresponding task name in the user interface. In this test case, the user selects the task "notify" for future execution.

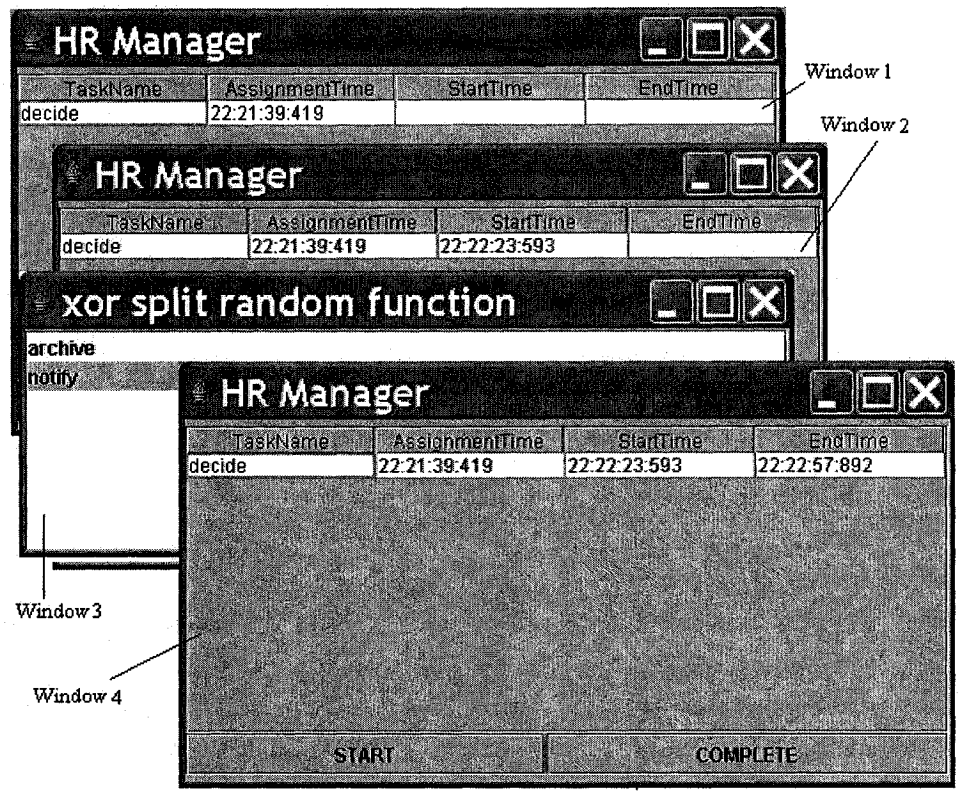

Figure 3-27 Log Information for Task "decide"

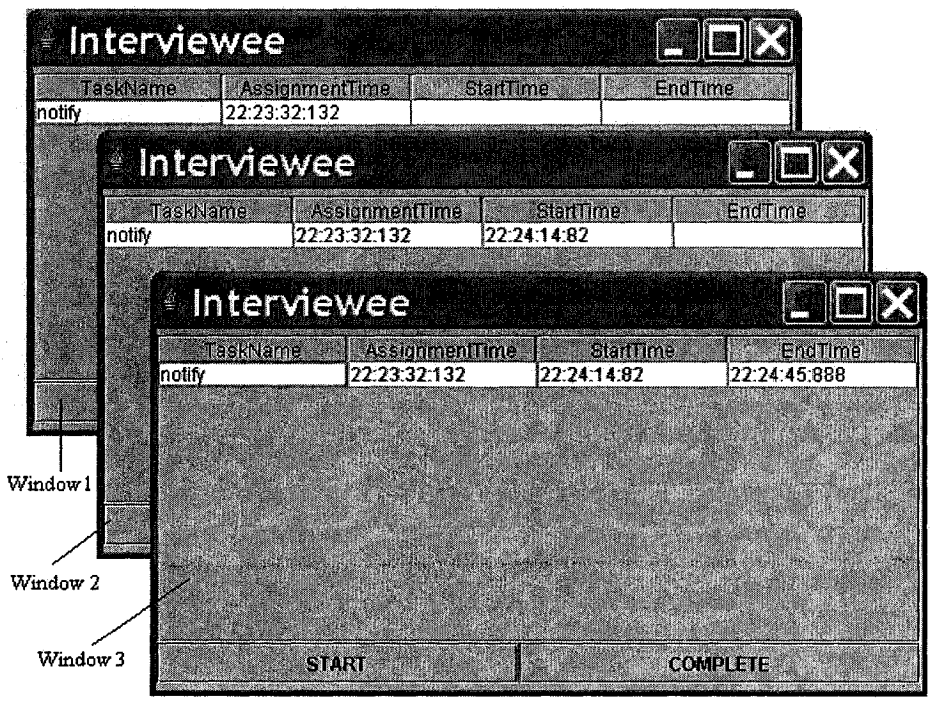

Figure 3-28 Log Information for Task "notify"

The above screenshots in Figure 3-28 demonstrate the assignment and the execution of the task "notify".

The result of this test demonstrates that the engine has interpreted the workflow specification correctly, and it has handled the execution order of each task based on the control flow specification in the XML workflow document. Through the user interface provided by the WorklistHandler, users can start the execution of the corresponding application function and users can notify its completion to the engine once the application function is over.

3.5 Related Work

In Section 2.4.1, we have introduced that the YAWL group categorized a collection of workflow patterns. So far, YAWL has identified 20 workflow patterns. YAWL was originally released in June, 2002. It provides direct support for these identified workflow patterns, which are associated with a graphical representation. There was no implementation available when YAWL was initially designed. (This inspired us to start developing a prototype partially supporting YAWL in May, 2003. For the sake of simplicity, we implemented a prototype workflow engine supporting only seven workflow patterns of YAWL.) Just as my thesis approaches completion, the YAWL group has been implementing a prototype supporting YAWL, and the latest version of the YAWL system was released in July, 2004.

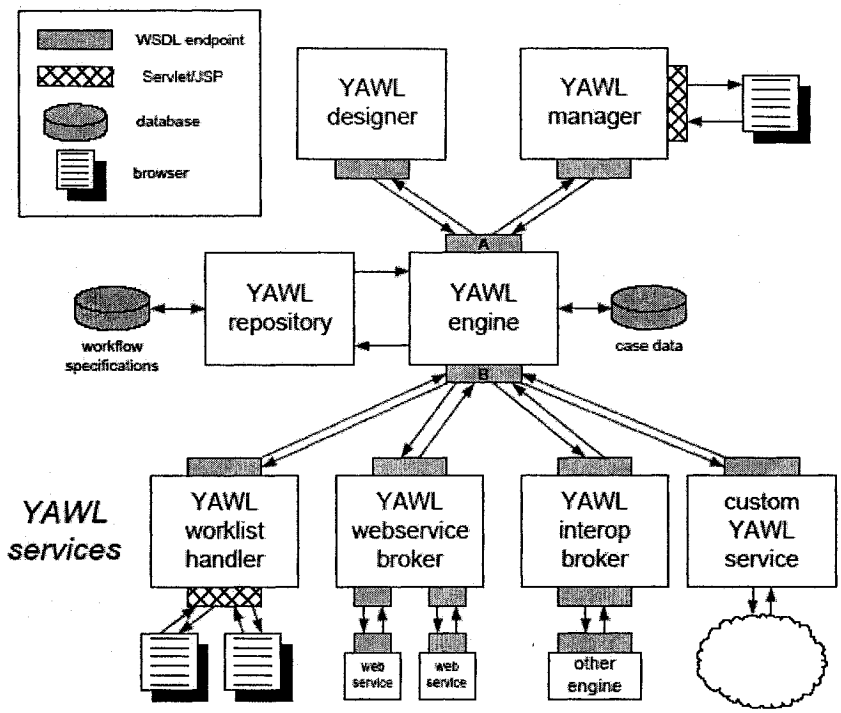

Figure 3-29 YAWL Architecture [3]

In the following paragraphs, we are going to introduce the design and the implementation of the YAWL system abstracted in Figure 3-29, and then we offer a brief comparison with our own system.

The YAWL group's proposed system architecture consists of *YAWL designer*, *YAWL manager*, *YAWL repository*, *YAWL engine* and *YAWL services*.

YAWL designer corresponds with the *Process Definition Tool* in the general workflow product architecture shown in Figure 2-1. It provides a graphic user interface by which the process designer can model the control flow specification of workflows. Moreover, it can validate the designer's specification and export it to the *YAWL engine*. In our prototype, the XML workflow document conforming to the XML workflow schema is used as the input file of the system. The XML editor tool, XML Spy5.0, is used

to specify the workflow specification in the XML document conforming to the XML workflow schema.

The *YAWL engine* is designed to verify the workflow specification, register tasks with specified services and keep the workflow specification in the *YAWL repository*. (The *YAWL repository* is in charge of the storage of all workflow specifications.) After the engine has assigned all these operations, it will instantiate the workflow specification, create a workflow instance and handle its execution. The current version of the *YAWL engine* implements the control flow specification of the workflow and it realizes all of the twenty workflow patterns in YAWL. The implementation of the prototype presented in this thesis is based on the specification of the workflow patterns of YAWL. Our system mainly focuses on the implementation of the control flow perspective of workflow. Currently, we employ seven workflow patterns. The design of our engine is based on the workflow engine in the general workflow product architecture in Figure 2-1. Its functions include: parsing the XML workflow document conforming to the XML workflow schema, interpreting the workflow specification in the XML workflow document, and managing the execution order of the workflow instance. Furthermore, in order to deal with the user interactions in the workflow, we model the tasks as either automatic tasks or tasks associated with human participants in the workflow specification. If the engine recognizes that the task is an automatic task, it invokes the corresponding application function directly. Otherwise, it will pass the task to a WorklistHandler to deal with the human interaction. However, the YAWL system deals with the user interaction in a different way, as the *YAWL engine* only deals with control flow but not explicitly with users. It does not discriminate between manual tasks and automatic tasks, instead it uses

services to take control of communication with the environment of YAWL systems. (These services will be introduced in the later parts of this section.) Another difference between our system and YAWL's is that the YAWL engine supports all work patterns while our engine only interprets seven workflow patterns.

The *YAWL manager* is a manual tool used to manage the execution of workflow instances. It provides functions to delete a workflow instance or a specified workflow specification. It can also provide state and relevant data information for the execution of workflow processes.

YAWL designer and *YAWL manager* interact with the *YAWL engine* through interface *A*, which captures the interactions between the *YAWL designer* and the *YAWL manager* on the one hand, and the *YAWL engine* on the other. Interface *A* is specified in WSDL (Web Services Description Language). The functions of the *YAWL designer* and the *YAWL manager* include the import and export of workflow specifications as well as administration and monitoring functions. However, the *YAWL manager* has not yet been implemented in the latest system to be released. In our prototype, the JAXB API in the workflow engine parses the workflow specification in the XML workflow document (introduced in section 3.3.2.1) directly. However, the manual administration function for monitoring the execution of the process instance has not been implemented in our prototype.

YAWL services construct the environment of a YAWL system. Inspired by the web service paradigm, end-users, applications, and organizations are all abstract services in YAWL. The YAWL services consist of the *YAWL worklist handler*, the *YAWL web services broker*, the *YAWL interoperability broker*, and the *custom YAWL services* which

make up the web service environment of the YAWL system. The *YAWL engine* interacts with *YAWL services* through interface *B*, which captures the interaction between the *YAWL services* and the *YAWL engine*. Interface *B* is also specified in WSDL. It identifies the corresponding services for each task and the operations within each service, and it contains XML message formats for input messages that contain the operations' input parameters and output messages that contain the operations' results. Since the workflow specification in this thesis focuses on the control flow perspective and the prototype is constructed in a single computer environment, the operational perspective of workflows is beyond the scope of this thesis.

The *YAWL worklist handler* communicates with the work list service [3], which is responsible for assigning the task to the corresponding user. This *worklist handler* is a service separated from the engine. If a given task is expected to register with a *worklist handler*, interface *B* is used to identify the corresponding worklist service for the task, and it also specifies the participant (role), who is asked to execute the task. However, in the latest released version of the YAWL system, the implemented *YAWL worklist handler* has not been separated from the engine. Since our prototype is constructed in a single computer environment, the worklist handler is a component of the workflow engine. During the execution, the engine will ask the worklist handler to add interface information of the enabled task to the user's worklist. When the task has been completed, this information will be removed from the worklist. The worklist handler shows an interface with the information in the worklist. Through this interface, the user can view his assigned tasks and operate the execution of the corresponding application functions.

The *YAWL web service broker* is responsible for connecting the engine to invoke external web services. If a task should be registered with the web service broker, interface *B* contains the information specifying the corresponding web service for the task, the operations within this service, the input messages that contain the operations' input parameters, and the output messages that contain the operations' results. The web service broker will use this information in interface *B* to interact with the corresponding web service. However, this service is still undergoing development in the YAWL project. In our prototype, the workflow engine runs in a single computer environment, while the web service has not been implemented yet.

The *YAWL interoperability broker* provides a service to enable remote workflow engines to communicate and work together. In the proposed YAWL system, different engines can be interconnected with each other through this broker. If a task is registered with the *YAWL interoperability broker*, interface *B* contains the identifier of the remote workflow engine to which the instance of this task will be delegated, the corresponding remote process to be instantiated by the remote engine and the input data and output data of the process. Since interface *B* specifies the location of the remote engine, the interoperability broker will use this information to invoke the remote *YAWL engine* in order to create a designated process instance. The broker can also receive the output data of the process instance and map it back to the local engine when the remote process instance is completed. However, the YAWL group has not implemented this service as yet. In our prototype, the interoperation with the remote workflow engine has not yet been considered.

The environment of YAWL system varies; for example, a mobile communication may be required to connect the engine with the environment. Thus, the YAWL group proposed the *YAWL custom service* to link the engine with entities in the different environment. However, this service has not been implemented in the YAWL system as yet.

In order to demonstrate the control flow features of the current YAWL system, the implemented prototype provides a Web browser interface for users to interact with the system. During the demonstration, the process definition designer first uses *YAWL designer* to model the workflow specification through a HTML front-end graphical interface. Then the definition is exported to the *YAWL engine* to invoke a process instance. During the interpretation of the workflow specification, the *YAWL engine* coordinates the execution order of the tasks in the workflow instance.

Initially, YAWL focuses exclusively on the control flow specification of the workflows, and there is no data perspective specified in YAWL [2]. Recently, the data perspective has been added in the latest version of the YAWL prototype. However, YAWL's existing definition still only supports the control flow perspective of the workflows [2], while other perspectives such as data, resources etc. have not been defined in it as yet. The current implementation of the YAWL system does not provide all of the components and functionality in the YAWL architecture, as outlined in Figure 3-29. However, the *YAWL engine* is fully developed. The functions of the *YAWL worklist handler* have been implemented but the handler is still embedded in the engine as a component of the system. The current release of the *YAWL designer* can only support the

control flow perspective. Other components and services in the YAWL architecture are still under development.

Chapter 4 Conclusion

In this thesis, we design and implement a prototype workflow engine supporting seven workflow patterns of YAWL, which are the five basic control flow patterns: the Sequence pattern, the Parallel Split pattern, the Synchronization pattern, the Exclusive Choice pattern and the Simple Merge pattern, as well as two other advanced patterns: the Multiple Choice pattern and the Synchronizing Merge pattern. As the starting point for our work, an XML workflow schema was designed for users to define the workflow specification. Based on this, the XML workflow document conforming to this XML schema is used as the input file of our system. At runtime, the engine parses the input file, interprets the workflow specification in the control flow perspective of YAWL and handles the execution order of the workflow.

Briefly, the main contribution of this thesis is that an XML workflow schema employing the seven workflow patterns of YAWL has been designed as a recursive hierarchy structure, and a prototype workflow engine incorporating the seven workflow patterns of YAWL, is designed and implemented with Java and JAXB to parse and interpret the workflow specification in the control flow perspective of YAWL.

As an academic project, our system has certain limitations. Our prototype workflow engine supports only YAWL's seven workflow patterns. However, there are totally 20 workflow patterns, as identified by the YAWL group. Other patterns should be added in the future. In this prototype, users have to use XML Spy 5.0 to define the workflow specification in the XML workflow document as the input file for the system. A graphical user interface must be designed and implemented to facilitate users to define the workflow specification. We also consider that the exception handling is an important

issue that needs to be implemented in every workflow product. However, since this thesis focuses on the workflow in the control flow specification of YAWL, the implementation of the exception handling will be the focus of future research.

Bibliography

[1] W.M.P. van der Aalst, The Application of Petri Nets to Workflow Management, Journal of Circuits, Systems and Computers, vol. 8(1), 21-66, pp.2-15, 1998.

[2] W.M.P. van der Aalst and A.H.M. ter Hofstede, YAWL: Yet Another Workflow Language, QUT Technical report, FIT-TR-2002-06, Queensland University of Technology, Brisbane, pp.3-13, 2002.

[3] W.M.P. van der Aalst, L. Aldred, M. Dumas, and A.H.M. ter Hofstede, Design and implementation of the YAWL system, QUT Technical Report, FIT-TR-2003-07, Queensland University of Technology, Brisbane, pp.3-11, 2003.

[4] W.M.P. van der Aalst, Advanced Workflow Patterns, 7th International Conference on Cooperative Information Systems (CoopIS 2000), volume 1901 of Lecture Notes in Computer Science, pages 18-29. Springer-Verlag, Berlin, 2000.

[5] G. Alonso, D. Agrawal A. El Abbadi, and C. Mohan, Functionality and Limitations of Current Workflow Management Systems, IEEE Expert, 12(5), pp.1-5, September-October 1997.

[6] N. Chase, Understanding DOM, IBM developerWorks Chase and Chase Inc., July 2003.

[7] N. Chase, Understanding SAX, IBM developerWorks Chase and Chase Inc., July 2003.

[8] P. Chrzastowski-Wachtel, Top-down Petri Net Based Approach to Dynamic Workflow Modeling (Work in Progress). University of New South Wales, Sydney, 2002.

[9] H.M.Deitel, P.J. Deitel, Java: How to program, Third Edition, Prentice Hall, p.411, pp.748-pp.753, 1999.

[10] FileNet Corporation. Visual WorkFlo Design Guide. Costa Mesa, CA, USA, 1997.

[11] Forté Software, Inc. Forté Conductor Process Development Guide. Oakland, CA, USA, 1998.

[12] Fujitsu Software Corporation. i-Flow Developers Guide. San Jose, CA, USA, 1999.

[13] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Pattern, First Edition Addison-Wesley, pp.163-166, pp.331-336, 1995.

[14] Hewlett-Packard Company. HP Changengine Process Design Guide. Palo Alto, CA,USA, 2000.

[15] IBM. IBM MQSeries Workflow - Getting Started With Buildtime. IBM Deutschland Entwicklung GmbH, Boeblingen, Germany, 1999.

[16] S. Jablonski and C. Bussler. Workflow Management: Modeling Concepts, Architecture, and Implementation. International Thomson Computer Press, London, UK, pp.12-20, 1996.

[17] K. Kawaguchi, The JAXB API, IBM developerWorks Chase and Chase Inc., January 2003.

[18] D. Lea, Concurrent Programming in Java, Second Edition, Addison-Wesley, p.40, p.75, March 2002.

[19] IBM. F. Leymann and D. Roller, Workflow-based applications, IBM Systems Journal, Volume 36, pp.1-3, November 1997.

[20] S.P. Nielsen, C. Easthope, P. Gosselink, K. Gutsze, and J. Roele. Using Lotus Domino Work-flow 2.0, Redbook SG24-5963-00. IBM, Poughkeepsie, USA, 2000.

[21] SAP. WF SAP Business Workflow. SAP AG, Walldorf, Germany, 1997.

[22] A. Sheth, K. Kochut, and J. Miller, Large Scale Distributed Information Systems (LSDIS) laboratory, METEOR project page, December 2001.

[23] Software-Ley GmbH, COSA 3.0 User Manual, Pullheim, Germany, 1999.

[24] Eastman Software, RouteBuilder Tool User's Guide, Eastman Software Inc., Billerica, MA, USA, 1998.

[25] Staffware plc, Staffware 2000 / GWD User Manual, Berkshire, United Kingdom, 2000.

[26] D. Steinberg and D. S. Thinking, Data binding with JAXB, IBM developerWorks Chase and Chase Inc., May 2003.

[27] Tibco Software Inc., TIB/InConcert Process Designer User's Guide, Palo Alto, CA, USA, 2000.

[28] Verve Inc., Verve Component Workflow Engine Concepts, San Francisco, CA, USA, 2000.

[29] Workflow Management Coalition, Workflow Management Coalition Terminology & Glossary, Document Number WFMC-TC-1011 Document Status - Issue 3.0, pp.8-13, pp.18-21, pp.39-41, February 1999.

[30] Workflow Patterns Home Page, updated on December 11, 2003.

[31] The Yawl Group homepage http://www.citi.qut.edu.au/yawl/index.jsp

Appendix A

An XML Workflow Document for the Sequence Pattern

```
<process id="1" name="SequenceSample">
        <inputCondition/>
        <sequence>
            <atomicTask id="1-1" name="Task A" status="null">
                <participant>
                    <automation>print log</automation>
                </participant>
            </atomicTask>
            <atomicTask id="1-2" name="Task B" status="null">
                <participant>
                    <automation>print log</automation>
                </participant>
            </atomicTask>
            <atomicTask id="1-3" name="Task C" status="null">
                <participant>
                    <automation>print log</automation>
                </participant>
            </atomicTask>
        </sequence>
        <outputCondition/>
</process>
```

Appendix B

An XML Workflow Document for Group of the Parallel Split Pattern and the Synchronization Pattern

```
<process id="1" name="ParSynSample">
        <inputCondition/>
        <parsyn>
            <andSplitTask id="1-1" name="Task A" status="null">
                <participant>
                    <automation>print log</automation>
                </participant>
            </andSplitTask>
            <postItem>
                <atomicTask id="1-2" name="Task B" status="null">
                    <participant>
                        <automation/>
                    </participant>
                </atomicTask>
                <condition/>
            </postItem>
            <postItem>
                <atomicTask id="1-3" name="Task C" status="null">
                    <participant>
                        <automation>print log</automation>
                    </participant>
                </atomicTask>
                <condition/>
        </postItem>
            <postItem>
                <atomicTask id="1-4" name="Task D" status="null">
                    <participant>
                        <automation>print log</automation>
                    </participant>
                </atomicTask>
                <condition/>
            </postItem>
            <andJoinTask id="1-5" name="Task E" status="null">
                <participant>
                    <automation>print log</automation>
                </participant>
            </andJoinTask>
        </parsyn>
        <outputCondition/>
</process>
```

Appendix C

An XML Workflow Document for Group of the Multiple Choice Pattern and the Synchronizing Merge Pattern

```
<process id="1" name="MulSynSample">
        <inputCondition/>
        <mulsyn>
            <orSplitTask id="1-1" name="Task A" status="null">
                <participant>
                    <automation>or split random function</automation>
                </participant>
            </orSplitTask>
            <postItem>
                <atomicTask id="1-2" name="Task B" status="null">
                    <participant>
                        <automation>print log</automation>
                    </participant>
                </atomicTask>
                <condition/>
            </postItem>
            <postItem>
                <atomicTask id="1-3" name="Task C" status="null">
                    <participant>
                        <automation>print log</automation>
                    </participant>
                </atomicTask>
                <condition/>
            </postItem>
            <postItem>
                <atomicTask id="1-4" name="Task D" status="null">
                    <participant>
                        <automation>print log</automation>
                    </participant>
                </atomicTask>
                <condition/>
            </postItem>
            <orJoinTask id="1-5" name="Task E" status="null">
                <participant>
                    <automation>print log</automation>
                </participant>
            </orJoinTask>
        </mulsyn>
        <outputCondition/>
</process>
```

Appendix D

An XML Workflow Document for the Exclusive Choice pattern

```
<process id="1" name="ExclusiveSample">
        <inputCondition/>
        <exclusive>
            <xorSplitTask id="1-1" name="Task A" status="null">
                <participant>
                    <automation>xor split random function</automation>
                </participant>
            </xorSplitTask>
            <condition/>
            <atomicTask id="1-2" name="Task B" status="null">
                <participant>
                    <automation>print log</automation>
                </participant>
            </atomicTask>
            <condition/>
            <atomicTask id="1-3" name="Task C" status="null">
                <participant>
                    <automation>print log</automation>
                </participant>
            </atomicTask>
            <condition/>
            <atomicTask id="1-4" name="Task D" status="null">
                <participant>
                    <automation>print log</automation>
                </participant>
            </atomicTask>
        </exclusive>
        <outputCondition/>
</process>
```

Appendix E

An XML Workflow Document for Group of the Multiple Choice Pattern and the Simple Merge Pattern

```
<process id="1" name="MulSimSample">
        <inputCondition/>
        <mulsim>
            <orSplitTask id="1-1" name="Task A" status="null">
                <participant>
                    <automation>or split random function</automation>
                </participant>
            </orSplitTask>
            <postItem>
                <atomicTask id="1-2" name="Task B" status="null">
                    <participant>
                        <automation>print log</automation>
                    </participant>
                </atomicTask>
                <condition/>
            </postItem>
            <postItem>
                <atomicTask id="1-3" name="Task C" status="null">
                    <participant>
                        <automation>print log</automation>
                    </participant>
                </atomicTask>
                <condition/>
            </postItem>
            <postItem>
                <atomicTask id="1-4" name="Task D" status="null">
                    <participant>
                        <automation>print log</automation>
                    </participant>
                </atomicTask>
                <condition/>
            </postItem>
            <xorJoinTask id="1-5" name="Task E" status="null">
                <participant>
                    <automation>print log</automation>
                </participant>
            </xorJoinTask>
        </mulsim>
        <outputCondition/>
</process>
```

Appendix F

An XML Workflow Document for Combinations of Seven Workflow Patterns in the Recursive Hierarchy Structure

```
<process id="1" name="seq(psyn(msyn(ex(msim))))" status="null">
        <inputCondition/>
        <sequence>
            <atomicTask id="2" name="seq1" status="null">
                <participant>
                    <automation>print log</automation>
                </participant>
            </atomicTask>
            <compositeTask id="3" name="seq2(psyn)" status="null">
                <inputCondition/>
                <parsyn>
                    <andSplitTask id="3" name="seq2(psyn_andSplit)" status="null">
                        <participant>
                            <automation>print log</automation>
                        </participant>
                    </andSplitTask>
                    <postItem>
                        <atomicTask id="4" name="seq2(psyn1)" status="null">
                            <participant>
                                <automation>print log</automation>
                            </participant>
                        </atomicTask>
                        <condition/>
                    </postItem>
                    <postItem>
                        <atomicTask id="5" name="seq2(psyn2)" status="null">
                            <participant>
                                <automation>print log</automation>
                            </participant>
                        </atomicTask>
                        <condition/>
                    </postItem>
                    <postItem>
                        <compositeTask id="6" name="seq2(psyn3(msyn))" status="null">
                            <inputCondition/>
                            <mulsyn>
                                <orSplitTask id="7" name="seq2(psyn3(msyn_orSplit))" status="null">
                                    <participant>
                                        <automation>or split random function</automation>
                                    </participant>
                                </orSplitTask>
                                <postItem>
                                    <atomicTask id="8" name="seq2(psyn3(msyn1))" status="null">
                                        <participant>
                                            <automation>print log</automation>
                                        </participant>
                                    </atomicTask>
                                    <condition/>
                                </postItem>
                                <postItem>
                                    <atomicTask id="9" name="seq2(psyn3(msyn2))" status="null">
                                        <participant>
                                            <automation>print log</automation>
                                        </participant>
                                    </atomicTask>
                                    <condition/>
                                </postItem>
                                <postItem>
                                    <compositeTask id="10" name="seq2(psyn3(msyn3(ex)))" status="null">
                                        <inputCondition/>
                                        <exclusive>
                                            <xorSplitTask id="11" name="seq2(psyn3(msyn3(ex_xorSplit)))"
status="null">
                                                <participant>
                                                    <automation>xor split random function</automation>
                                                </participant>
                                            </xorSplitTask>
                                            <condition/>
```

```
                                        <atomicTask id="12" name="seq2(psyn3(msyn3(ex1)))"
status="null">
                                            <participant>
                                                <automation>print log</automation>
                                            </participant>
                                        </atomicTask>
                                        <condition/>
                                        <compositeTask id="13" name="seq2(psyn3(msyn3(ex2(msim))))"
status="null">
                                            <inputCondition/>
                                            <mulsim>
                                             <orSplitTask id="14"
name="seq2(psyn3(msyn3(ex2(msim_orSplit))))" status="null">
                                                    <participant>
                                                        <automation>or split random function</automation>
                                                    </participant>
                                             </orSplitTask>
                                             <postItem>
                                                 <atomicTask id="15"
name="seq2(psyn3(msyn3(ex2(msim1))))" status="null">
                                                     <participant>
                                                         <automation>print log</automation>
                                                     </participant>
                                                 </atomicTask>
                                                 <condition/>
                                             </postItem>
                                             <postItem>
                                                 <atomicTask id="16"
name="seq2(psyn3(msyn3(ex2(msim2))))" status="null">
                                                     <participant>
                                                         <automation>print log</automation>
                                                     </participant>
                                                 </atomicTask>
                                                 <condition/>
                                             </postItem>
                                             <xorJoinTask id="17"
name="seq2(psyn3(msyn3(ex2(msim_xorJoin))))" status="null">
                                                 <participant>
                                                     <automation>print log</automation>
                                                 </participant>
                                             </xorJoinTask>
                                            </mulsim>
                                            <outputCondition/>
                                        </compositeTask>
                                    </exclusive>
                                    <outputCondition/>
                                </compositeTask>
                                <condition/>
                            </postItem>
                            <orJoinTask id="18" name="seq2(psyn3(msyn_orJoin))" status="null">
                                <participant>
                                    <automation>print log</automation>
                                </participant>
                            </orJoinTask>
                        </mulsyn>
                        <outputCondition/>
                    </compositeTask>
                    <condition/>
                </postItem>
                <andJoinTask id="19" name="seq2(psyn_andJoin)" status="null">
                    <participant>
                        <automation>print log</automation>
                    </participant>
                </andJoinTask>
            </parsyn>
            <outputCondition/>
        </compositeTask>
        <atomicTask id="20" name="seq3" status="null">
            <participant>
                <automation>print log</automation>
            </participant>
        </atomicTask>
    </sequence>
    <outputCondition/>
</process>
```

Appendix G

An XML workflow document for the HR recruitment process sample

```
<process id="0" name="HRrecuitmentSample">
        <inputCondition/>
        <sequence>
            <atomicTask id="1" name="send resume" status="null">
                <participant>
                    <automation>print log</automation>
                    <person>Secretary</person>
                </participant>
            </atomicTask>
            <compositeTask id="2" name="ParSynTask">
                <inputCondition/>
                <parsyn>
                    <andSplitTask id="3" name="distribute resume">
                        <participant>
                            <automation>print log</automation>
                            <person>HR Manager </person>
                        </participant>
                    </andSplitTask>
                    <postItem>
                        <atomicTask id="4" name="check (IT Manager)">
                            <participant>
                                <automation>print log</automation>
                                <person>IT Manager</person>
                            </participant>
                        </atomicTask>
                        <condition/>
                    </postItem>
                    <postItem>
                        <atomicTask id="5" name="check (IT Director)">
                            <participant>
                                <automation>print log</automation>
                                <person>IT Director</person>
                            </participant>
                        </atomicTask>
                        <condition/>
                    </postItem>
                    <postItem>
                        <atomicTask id="6" name="check (HR Manager)">
                            <participant>
                                <automation>print log</automation>
                                <person>HR Manger</person>
                            </participant>
                        </atomicTask>
                        <condition/>
                    </postItem>
                    <andJoinTask id="7" name="collect evaluation reports">
                        <participant>
                            <automation>print log</automation>
                            <person>HR Manager</person>
                        </participant>
                    </andJoinTask>
                </parsyn>
                <outputCondition/>
            </compositeTask>
            <compositeTask id="8" name="ExclusiveTask" status="null">
                <inputCondition/>
                <exclusive>
                    <xorSplitTask id="9" name="decide" status="null">
                        <participant>
                            <automation>xor split random function</automation>
                            <person>HR Manager</person>
                        </participant>
                    </xorSplitTask>
                    <condition/>
                    <atomicTask id="10" name="archive" status="null">
                        <participant>
                            <automation>print log</automation>
                            <person>Secretary</person>
                        </participant>
                    </atomicTask>
                    <condition/>
                    <atomicTask id="11" name="notify" status="null">
                        <participant>
```

```
                        <automation>print log</automation>
                        <person>Interviewee</person>
                    </participant>
                </atomicTask>
            </exclusive>
            <outputCondition/>
        </compositeTask>
    </sequence>
    <outputCondition/>
</process>
```