A CODE GENERATION APPROACH TO SUPPORT DYNAMIC WORKFLOW MANAGEMENT

By

XIAOLI LIU

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2001

To My Family

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Herman Lam, for giving me an opportunity to work on this challenging topic and for providing continuous guidance throughout the course of this work and thesis writing.

I wish to thank Dr. Stanley Su as well for his guidance and support during this work. Thanks are also due to Dr. Joachim Hammer for agreeing to be on my committee.

I thank Sharon Grant for making the Database Center a great research place, and I also thank all my friends at the University of Florida for their support and encouragement.

I thank my parents and my brother for their constant love and support. Thanks also go to my husband, Haifei Li, and my daughter, Joy Li.

TABLE OF CONTENTS

page

LIST OF FIGURES

Figure Page

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

A CODE GENERATION APPROACH
TO SUPPORT
DYNAMIC WORKFLOW MANAGEMENT

By

Xiaoli Liu

December 2001

Chairman: Dr. Herman Lam
Major Department: Computer and Information Science and Engineering

Inter-enterprise workflow technology is a key service needed to coordinate the activities of different organizations in a virtual enterprise. A Dynamic Workflow Model (DWM) has been proposed to support dynamic management of inter-enterprise business processes. The Dynamic Workflow Model (DWM) has the following properties: active, flexible, adaptive, and customizable. This thesis presents a code generation approach to support the adaptive property of the DWM, the ability to dynamically change the activities, control flow, and data flow of an executing workflow instance.

A Code Generator has been designed and implemented. Based on the specification of a workflow model, the Code Generator generates (1) a set of run-time workflow structures and (2) activity code. The *run-time workflow structures* are used by the Workflow Engine to schedule and execute a workflow instance. Changes can be made at run-time to the control and data flows of an executing workflow instance by

adding, modifying, or deleting run-time structures. The *activity code* is generated based on the activity specifications of a workflow model. During activity code generation, activity specifications are translated and compiled into Java classes which, when executed, efficiently request the binding of e-service requests to service providers for execution. If the specification of an activity is changed, the code for the modified activity is re-generated and re-loaded using Java's class reloading capability. In this manner, adaptability is maintained without sacrificing performance, resulting in a lightweight and efficient Workflow Engine.

CHAPTER 1
INTRODUCTION

Workflow technology and workflow management systems (WfMSs) enable organizations to model their business processes and control their execution. This allows organizations to reduce processing time, allocate resources efficiently, and shorten product time to market. However, traditional workflow management systems currently available on the market usually work within a single organization and support a static workflow model.

The rapid growth of the Internet and Web technologies has brought about significant changes in the way business organizations operate and altered the nature of business competition. To stay competitive in the rapidly changing and expanding marketplace, it is often necessary for business organizations to form virtual enterprises with other businesses to achieve common business goals. In order to work in a virtual enterprise, characterized by its dynamic nature, the concept of dynamic workflow management should be introduced to workflow management systems to handle inter-enterprise business processes.

Today, there are many workflow management systems available on the market (e.g., Vitria's Business Ware and IBM's MQSeries [GEO95, VIT99, MQW00]). There are also several research projects which have explored the concept of dynamic workflow to support workflow management across enterprise boundaries [ELL95, CAS96, REI98, MUL99]. Recently, several research projects aim to solve workflow management

problems in the virtual enterprise environment [LAZ00, ALO99, CRO00, GRE99]. We will survey these works in Chapter 2.

Currently, there is an on-going project at the Database Systems Research and Development Center at the University of Florida to build an information infrastructure for supporting Internet-based Scalable E-business Enterprises (ISEE) [SU01]. A key ISEE service is the ISEE workflow service which incorporates "dynamic" properties into the workflow management system (WfMS). The work presented here is a part of that project. A Dynamic Workflow Model (DWM) has been proposed [MEN01] to model business processes.

The Dynamic Workflow Model is an extension of the Workflow Process Definition Language (WPDL) [WFM99a] by including business events and rules to capture the active properties of business processes. By integrating business processes with business events and rules, the enactment of a business process can post events to trigger business rules, which may in turn cause the enactment of other business processes. The processing of rules enables the dynamic changes made to the business processes. Connectors are introduced to represent control information (i.e., Split, Join) extracted from activity definition so that each activity definition is encapsulated and reusable. All the sharable tasks performed by people or automated systems in a virtual enterprise are regarded as Web services, or e-services. E-service requests are specified in each activity definition within a process model according to standardized e-service templates and are bound to the proper service providers at run-time with the help of the Broker Server. Therefore, the process models defined are independent of the service providers coming and going in a virtual enterprise. Dynamic properties are achieved by integrating

business processes with events and rules, and the dynamic binding of e-services with service providers.

The focus of this thesis is the design and implementation of build-time tools to support a dynamic workflow management system. Using a "code generation" approach, the build-time tools generate (1) run-time workflow structures and (2) activity code, based on the specification of a workflow model. The *run-time workflow structures* are used by the Workflow Engine to schedule and execute a workflow instance. Dynamic changes to the control and data flows of an executing workflow instance can be made at run-time by changing these run-time structures by invoking methods via the API of the Workflow Engine.

The generated *activity code* is in the form of Java code. The Workflow Engine, when scheduling the activities for a process instance, simply loads the Java activity classes from a Run-time Repository and executes the code directly to perform specified tasks. This code generation approach results in a lightweight and efficient Workflow Engine. It is lightweight because the Workflow Engine does not need the logic to interpret the activity specification in order to determine how to bind the e-service requests and calling the bound service provider to perform the e-services. Taking advantage of the performance of compiled classes, the Workflow Engine is efficient. Finally, changes to an activity (e.g., e-service requests, performer, performer constraints, etc.) can be made using a Process Definition Tool. The code for the modified activity is re-generated and re-loaded using Java's class reloading capability. In this manner, any change made to an activity specification will be reflected in the execution of the activity

as long as the re-generated code is placed in the Run-Time Repository *before* the execution of the activity. Thus, flexibility is maintained without sacrificing performance.

The organization of this thesis is as follows. Chapter 2 presents a survey of research works related to dynamic workflow management and workflow management in a virtual enterprise environment. Chapter 3 describes the global architecture of the ISEE information infrastructure and the dynamic WfMS. The modeling constructs of the Dynamic Workflow Model are presented in Chapter 4. Chapters 5 and 6 focus on the details of the run-time workflow structures generation and the activity code generation, respectively. Chapter 7 provides a summary and conclusion of the thesis.

CHAPTER 2
SURVEY OF RELATED WORK

Inter-enterprise workflow technology is a key e-service to coordinate the activities of different organizations in a virtual enterprise. In the highly competitive and rapidly changing and expanding global marketplace, business organizations often find the need to form virtual alliances with other businesses to achieve various business goals. Unlike a "real" enterprise when business processes and policies may be relatively static, the processes and policies of a virtual enterprise are much more dynamic. This chapter surveys related research focusing on bringing dynamism to workflow management and projects focusing on workflow management in virtual enterprises.

2.1 Process Modeling and Process Enactment

Process modeling and process enactment are the two essential parts of any workflow management system. The purpose of process modeling is to produce an abstract representation of a process model upon which the workflow specification is based. There are basically two kinds of process modeling methodologies, namely communication-based, and activity-based [GEO95, SHE96, WIN87]. Communication-based methodologies stem from "The Winograd/Flores Conversation for Action Model" [WIN87] in which the progress of a coordination is represented as a finite-state machine. In these methodologies, an action in a workflow is represented by the communication between a customer and a performer. The resulting organizational process reveals a social network in which a group of people, assuming various roles, fulfills an organizational process.

Activity-based methodology is most frequently adopted by workflow management systems because it decomposes a process into tasks that are ordered according to dependencies among them. This kind of process model can be easily turned into a workflow specification. Most commercial workflow management systems, including IBM's MQ Workflow [MQW00, LEY94], adopt this activity-based methodology in its process modeling. Our project also uses activity-based methodology to model processes to facilitate the dynamic workflow model specification. Workflow specification languages are used to describe process models. To capture the dependencies among activities, workflow specification languages often use rules, constraints, and/or graphical constructs.

During process enactment, the workflow engine is responsible for interpreting the process definition, creating the instance of the process, and managing its execution. A workflow management system may consist of one or more workflow engines. Most existing commercial workflow systems are centralized in the sense that a single workflow engine handles the execution of one process instance. For example, MQ Workflow [MQW00] adopts the centralized control of one workflow instance. It employs a three-tier client-server architecture and involves external applications that perform geographically distributed tasks on different platforms.

2.2 Dynamic Workflow Projects

There have been several projects that focused on various aspects of dynamic workflow management. Most of them deal with the dynamic changes of process models used in transitional workflow systems. In Ellis et al. [ELL95], an approach to provide dynamic changes to a process structure is introduced. The dynamic changes are accomplished by replacing a given sub-workflow with another completely specified sub-

workflow. This approach makes use of Petri-net formalism to analyze structural changes. Reichert and Dadam [REI98] presents a formal foundation for supporting dynamic structural changes of running workflow instances. Based upon a formal workflow model (ADEPT), a complete and minimal set of change operations (ADEPT_flex) is defined to support users in modifying the structure of a running workflow instance, while maintaining its (structural) correctness and consistency. The change operations include inserting tasks as well as task blocks into a workflow graph, deleting tasks, skipping tasks, serializing tasks that were previously allowed to run in parallel, and dynamically iterating and dynamically rolling-back a workflow. Muller and Rahm [MUL99] describe a rule-based approach for the detection of semantic exceptions and for dynamic workflow modifications, with a focus on medical workflow scenarios. In this approach, rules are used to detect semantic exceptions and to determine which activities need to be skipped or added.

The remainder of this section describes two major projects related to dynamic workflow management: EvE and RWS.

2.2.1 The EvE Project

EvE [GEP98] is an EVent Engine implementing event-driven distributed workflow execution. This project was carried out at the University of Zurich. The event engine performs event registration, detection, and management, as well as event notification to distributed, reactive software components. In the EvE project, events and Event-Condition-Action (ECA) rules are used as the fundamental concept for defining and enforcing workflow logic, and processing entities are used to represent the software applications that enact the workflow by reacting to events and generating new events. A broker/services model is used to describe the software and process architecture. In this

model, brokers are interacting, reactive components representing the processing entities. Broker behavior is defined by ECA rules specifying their reactions to events. The execution of a workflow starts when some event occurs. The event engine server then performs event detection and rule execution. While executing the rule, task assignment determines the brokers to be notified. These brokers then react according to their ECA rules. Brokers may generate new events, which are again processed by the event engine until the execution of the workflow is complete. The event-based EvE eases integration and coordination of the processing entities. Due to this infrastructure, dynamic changes to the workflow can be achieved by either changing or introducing new ECA rules.

2.2.2 The RWS Project

Reliable Workflow Systems (RWS) [SHR98, WHE98] is implemented at the University of Newcastle in joint collaboration with Nortel technology. There are two main components in the system architecture: the workflow repository service and the workflow execution service.

The workflow repository service stores workflow schemas and provides operations of initializing, modifying, and inspecting schemas. A scripting language has been designed for defining the workflow schemas represented in terms of tasks and dependencies.

The workflow execution service coordinates the execution of a workflow instance. It records inter-task dependencies of a schema and ensures that tasks are scheduled to run respecting their dependencies. The dependency information is maintained and managed by task controllers. Distributed task controller objects interacting and communicating directly with each other implement the workflow logic. Dynamic changes to workflow are done by changing the inter-task dependencies, either

in the workflow schema, or directly in the workflow instance, in this case the task controller.

2.3 Virtual Enterprise and Workflow Technology

The expansion of the Internet and the proliferation of inexpensive computing power provide the basic hardware infrastructure for B2B electronic business in virtual enterprise. However, the corresponding software infrastructure is still missing. Fortunately, there are some ongoing projects to address the limitation. Among them, WISE and CrossFlow are presented here.

2.3.1 The WISE Project

Workflow based Internet Services (WISE) [ALO99, LAZ00] is a project funded by the Swiss National Science Foundation. Its objective is to design, build, and test a commercially viable infrastructure for business to business e-commerce in the form of a working system capable of defining, enacting, and monitoring virtual enterprise business processes. The infrastructure provides an Internet workflow engine acting as the underlying distributed operating system controlling the execution of business processes, a process modeling tool for defining and monitoring the processes, and a catalogue tool for virtual enterprise services. There are four components in the WISE architecture, namely, process definition, process enactment, process monitoring, and process coordination.

- The process definition component defines virtual business processes with available services offered by different companies in the virtual enterprise.
- The process enactment component controls the execution of the process by invoking the corresponding services of the virtual enterprise.
- The process monitoring component keeps track of the progress made in the execution of the virtual enterprise business process.
- The process coordination component supports multimedia conferencing and browsing of relevant information between all participants in the virtual enterprise.

In WISE, different companies join their services to form a virtual enterprise, which provides a business process that can be executed over the Internet. The workflow system provided in WISE, however, does not have the facilities to capture and manage the dynamic properties of business processes.

2.3.2 The CrossFlow Project

CrossFlow [LUD99, CRO00] is a European research project for supporting cross-organizational workflow management in virtual enterprises. In this project, IBM's MQSeries Workflow is used. The goal of this project is to develop and implement the mechanism for connecting WfMS and other WfMS-like systems of different organizations in dynamically formed virtual organizations. Virtual organizations are dynamically formed by contract-based matchmaking between service providers and consumers. In CrossFlow, there is a *trader* known to consumers for service providers to post services. If a process in a WfMS requires some services from another organization, the WfMS sends a request to the *search manager* to look for suitable service providers. After the service provider is found and a contract is made between the consumer and the provider, the service can be initiated and a process in the provider WfMS can be activated on behalf of the consumer. Different workflow systems are linked through *proxy gateways*, which are dynamically generated modules at organizational boundaries. A flexible change control mechanism is also introduced in CrossFlow to react to potential problems during a workflow execution.

The ISEE dynamic workflow management system (WfMS) described in this thesis supports dynamic workflow management. By introducing events, triggers, rules, and constraint-based dynamic service binding, our process model can capture more dynamic properties, such as active, flexible, adaptive and customizable. The integration

of constraint-based dynamic service binding, event, trigger, and rule mechanism with business processes provides a comprehensive solution to inter-enterprise workflow management. The code generation approach described in this thesis supports the dynamic properties in the ISEE WfMS.

CHAPTER 3
ARCHITECTURE OF THE ISEE WfMS

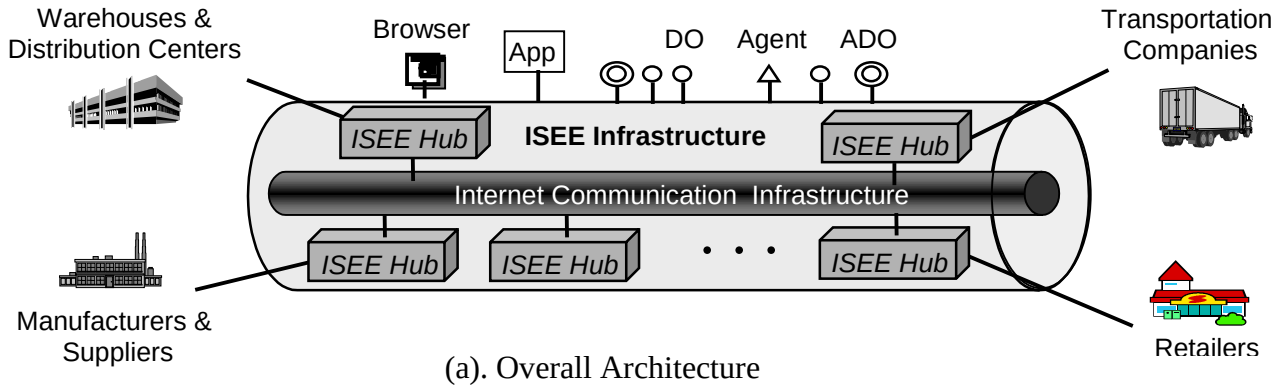
A research project on advanced technologies to support Internet-based Scalable E-business Enterprises (ISEE) is in progress at the Database Systems Research and Development Center of the University of Florida. Its purpose is to build an advanced information infrastructure capable of providing ISEE services for collaborative e-business [SU01]. One of the key services provided by the ISEE information infrastructure is the ISEE Workflow Management System (WfMS) which implements the dynamic workflow service.

This chapter describes the overall ISEE architecture, and the overall architecture of the ISEE WfMS. Section 3.1 describes the ISEE architecture, followed by an introduction of the overall architecture of the ISEE WfMS in Section 3.2. Section 3.3 presents the run-time architecture of the ISEE WfMS, while Section 3.4 describes the build-time architecture of the ISEE WfMS.

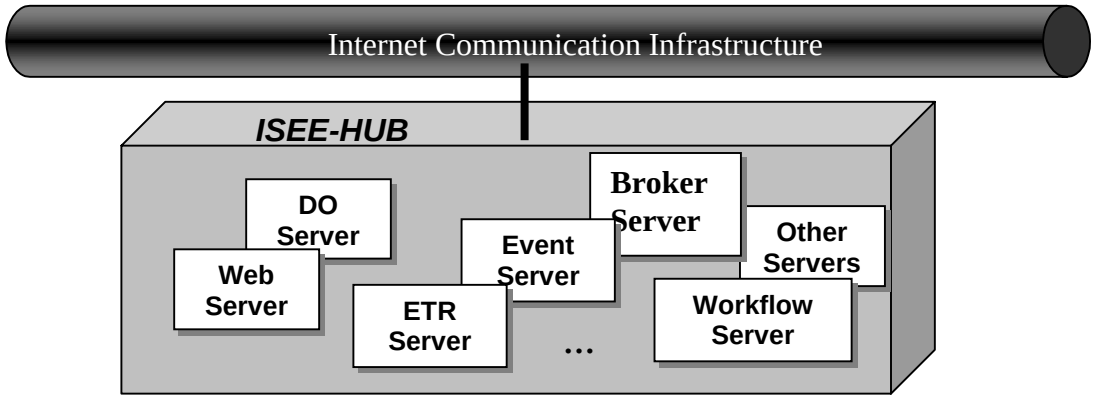
3.1 ISEE Architecture

With the Internet, World Wide Web (WWW), and other existing Information Technologies (IT), a user today can connect to Web servers, access their back-end database systems, and transact business. The emerging distributed object technology further enhances the information technology by allowing distributed applications to be invoked in a structured manner, in forms of communicating Distributed Objects (DOs). DOs can be used to encapsulate existing application systems and to develop new

applications. Their services are accessible by users through Web browsers. Thus, the current (Web + existing IT) paradigm provides a basic infrastructure over the Internet to interconnect enterprises and allow data and application systems to be shared among customers, suppliers, and business partners.



(a). Overall Architecture



(b). An ISEE Hub

Figure 3.1 ISEE Infrastructure

In [SU01], an information infrastructure to support an ISEE was proposed to enable collaborative e-business among multiple organizations to conduct business over the Internet. In an ISEE, the basic infrastructure is enhanced to, not only support the sharing of data and application systems over the Internet, but also provide ISEE services necessary for collaborative e-business and other general distributed applications. That is,

ISEE Infrastructure = Web + Existing IT + ISEE Services for Collaborative e-business.

The proposed infrastructure and its relationship with existing application systems, distributed objects, agents, active distributed objects, Web browsers, and Web servers in a virtual enterprise environment are shown in Figure 3.1. In Figure 3.1(a), ISEE hubs are replicated and deployed at the sites of participating ISEE members, much the same way the Web servers are deployed now to support Web services. As shown in Figure 3.1(b), each ISEE hub consists of a number of ISEE servers such as an Event Server, an Event-Trigger-Rule (ETR) Server, a Workflow Server, a Broker Server, a DO Server, a Web Server, and other general purpose servers. Internet users can still use the Internet and Web services as they are but the ISEE members can have access to the additional services.

The ISEE servers in an ISEE hub work together to provide a wide range of collaborative ISEE services to connected members. We have been developing and integrating a number of what we regard as the core information infrastructure technologies and services. They include active distributed object modeling, business event management, business rule management, messaging infrastructure for inter-enterprise communication, cost-benefit analysis and evaluation, trust management and access control, and constraint satisfaction processing. Other services needed to support collaborative e-business, such as constraint-based brokering, supplier selection, automated negotiation, dynamic workflow modeling and control, can be built upon these core technologies. The implementations of all these services in forms of servers are intended to complement the services by the Internet and Web.

The dynamic workflow service is a key ISEE service for managing inter-enterprise processes, and is the focus of this thesis. In a virtual enterprise, business processes such as an e-supply-chain, can cross company boundaries. Thus, workflow systems are required to manage business processes across enterprise boundaries. Unlike a "real" enterprise when business processes and policies may be relatively static, the processes and policies of a virtual enterprise are much more dynamic. In chapter 4, a Dynamic Workflow Model is presented to support the management of inter-enterprise business processes.

3.2 ISEE WfMS

The ISEE WfMS is the implementation of the ISEE dynamic workflow service. The architecture of the ISEE WfMS is shown in Figure 3.2. It involves several servers including a centralized Broker Server; and several servers in an ISEE hub: an Event Server, an ETR Server, and a Workflow Server.

The constraint-based Broker Server provides information about service providers. It also performs dynamic service binding to bind a requested e-service specified in an activity of an inter-enterprise process (workflow) to a specific service provider at run-time. In order to communicate with the Broker Server, a Broker Proxy is installed in each ISEE hub. All sharable tasks, which can be performed by different organizations, are considered as e-services. An e-service adapter exists in each organization to wrap its e-services so that the organization's e-services can be invoked in the execution of an inter-enterprise business process. The Event Server manages events coming to and leaving an ISEE hub. The ETR Server takes care of event, trigger and rule processing.

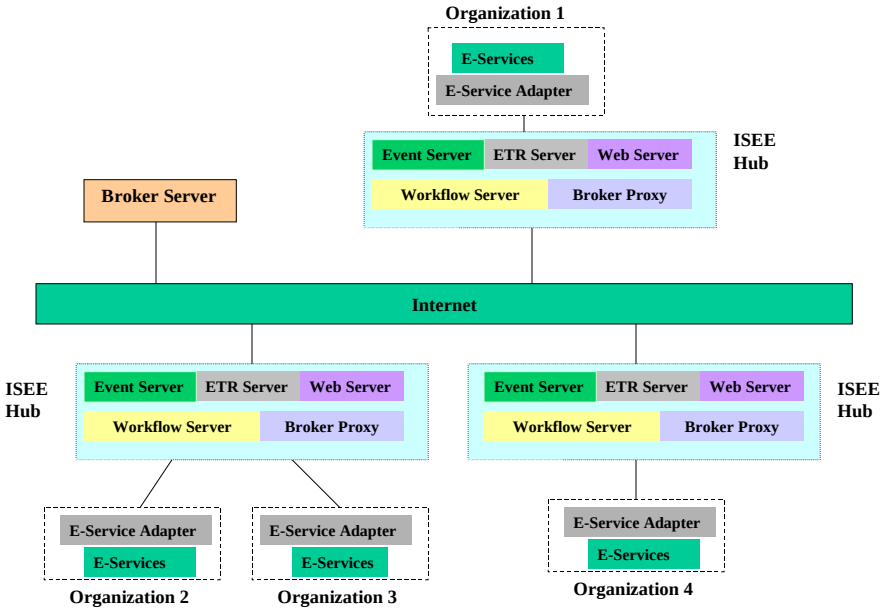


Figure 3.2 Overall Architecture of ISEE WfMS

The Workflow Server is the key component of the ISEE WfMS. The implementation of the Workflow Server is based on workflow technology which enables the modeling of business processes and automates the control of their execution. Before we continue, let us define some terms. A *workflow model* (also called *process model*) is used to model a business process and consists of a network of activities and their control/data flow of these activities. A workflow model can be instantiated as a *workflow instance* (also called *process instance*) and executed in the Workflow Server. The Workflow Server is composed of a set of build-time tools and the Workflow Engine. The build-time tools include the Process Definition Tool (PDT) and the Code Generator. The PDT is a Graphical User Interface (GUI) used to define and edit workflow models (note these are also called process models). The Code Generator is used to generate run-time

workflow structures and activity codes according to workflow model specifications defined with PDT. The Workflow Engine schedules and executes workflow instances according to the workflow model specifications.

3.3 Run-Time Architecture of ISEE WfMS

The run-time architecture of the ISEE WfMS is shown in Figure 3.3. The main components involved in the architecture are:

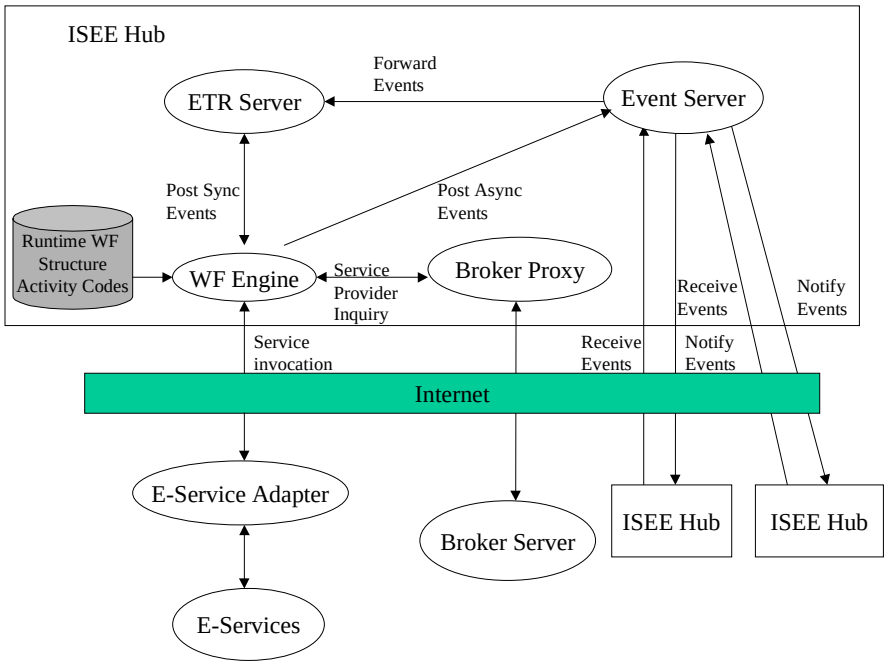


Figure 3.3 The Run-Time Architecture of the ISEE WfMS

1. Workflow Engine: The Workflow Engine manages the execution of a business process until completion by assigning activities to each specified performer at the specified time. During the execution, the Workflow Engine makes use of other collaborative ISEE services provided by other servers, namely, the Event Server, the ETR Server, and the Broker Server to achieve the dynamic properties (i.e., the active, flexible, and adaptive properties) of the WfMS. While executing a workflow instance, the Workflow Engine may post asynchronous events to the Event Server,

which then notifies the subscribers of these events. It may also post synchronous events to the ETR Server to trigger rules attached to the events. The Workflow Engine also contacts the Broker Server through the Broker Proxy to obtain service provider information for e-service requests specified in the activities of a process model. After a qualified service provider is selected, the Workflow Engine invokes the specified service through an e-service adapter.

2. Event Server: The Event Server provides the publish-subscribe model of an event service for asynchronous events. At the Event Server, event providers publish available events, while event consumers, as participants of the virtual enterprise, can subscribe or unsubscribe to events of their interest. When an event is posted, the Event Server will notify all subscribers to that event. The Workflow Engine, as one of the event providers, may post asynchronous events to the Event Server. The Event Server is also connected with the Event Servers in other ISEE hubs to notify them of events and receive event notifications from them.

3. ETR Server: The ETR Server and the Event Server provide the "active" property of the Dynamic Workflow Model. At run-time, the events specified in a Dynamic Workflow Model will be posted to the Event Server and the ETR Server to trigger rules in the ETR Server. The ETR Server is a subscriber to an Event Server. Thus, the ETR Server may receive event notifications from the Event Server (for asynchronous workflow events). Also, the ETR Server may receive events directly from the Workflow Engine (for synchronous workflow events). The posted events will activate all triggers associated with that event. The event history specified in a trigger will be processed and appropriate rules will be executed.

4. Broker Server: The Broker Server allows e-service providers to register and publish their e-services in terms of descriptive attributes and constraints through GUIs. It manages the registered e-service information and processes inquiries to service provider information. The e-service requests are also specified in terms of descriptive attributes and constraints. The Broker Server makes use of a Constraint Satisfaction Processing (CSP) server as its key component. The CSP is used by the Broker Server to dynamically bind e-service requests to e-service specifications and their providers according to the e-service requests specifications and their constraints.

5. Broker Proxy: The Broker Proxy in each ISEE hub maintains communication with the remote Broker Server. The Workflow Engine contacts the Broker Server through the Broker Proxy.

3.4 Build-Time Architecture of the ISEE WfMS

The build-time architecture discussed in this thesis is designed to support a code generation approach to realize a dynamic workflow management system. The main

components involved in the build-time architecture of the ISEE WfMS are outlined below.

1. MetaData Manager: The MetaData Manager is a server which provides a persistent repository for storing meta-information. The specifications for process model, data classes, event, rule and e-service request constraints are captured using GUI editors and stored in the MedaData Manager. Components of the ISEE WfMS make references to the MetaData Manager to access the specifications. The MetaData Manager is built on top of a Persistent Object Manager (POM) [SHE01]. It was developed for IKnet [LEE00, SU00] and is extended in our project to support workflow concepts.

2. GUI Editors: Three GUI editors are involved in defining a process model, namely, the Process Definition Tool (PDT), the Knowledge Profile Manager (KPM), and the Constraint Definition Tool (CDT). The Process Definition Tool is used to specify various parts of a process model, including activities, transitions, connectors, etc.. Knowledge Profile Manager is invoked from the PDT to define events, triggers, and rules related to a process model. The definitions of events generated by the process models are sent to both the ETR Server and the Event Server to perform installation operations. Also, the Constraint Definition Tool is invoked from the PDT to define constraints associated with the e-service requests specified in the process models.

3. Code Generator: The focus of this thesis is the Code Generator component used to support the dynamic workflow management system. The Code Generator translates the process model specifications into run-time workflow structures. The run-time workflow structures generation separates the run-time workflow structures from the process model specifications and supports the dynamic changes to a business process. The Code Generator also generates the code to implement the activities specified in a process model. Activity code generation results in an efficient and lightweight Workflow Engine and also supports dynamic changes to a business process. Specifically, activity specifications are translated and compiled into Java classes. These Java classes, when executed, perform tasks specified in the activities and may invoke the Broker Server for dynamic binding of the e-service requests with appropriate e-services and their providers. The compiled activity codes benefit the performance of the Workflow Engine. When the Workflow Engine is going to execute a workflow instance, it will read in the run-time workflow structures generated for the appropriate process model. According to the run-time workflow structures, activities are scheduled and executed by loading the corresponding activity code already generated. The Code Generator generates run-time workflow structures and activity code to support a combination of “interpretative” and “code generation” approach to realize dynamic workflow management.

3.4.1 Generation of Run-time Workflow Structures

Figure 3.4 illustrates how the main components are related to each other during the generation of run-time workflow structures.

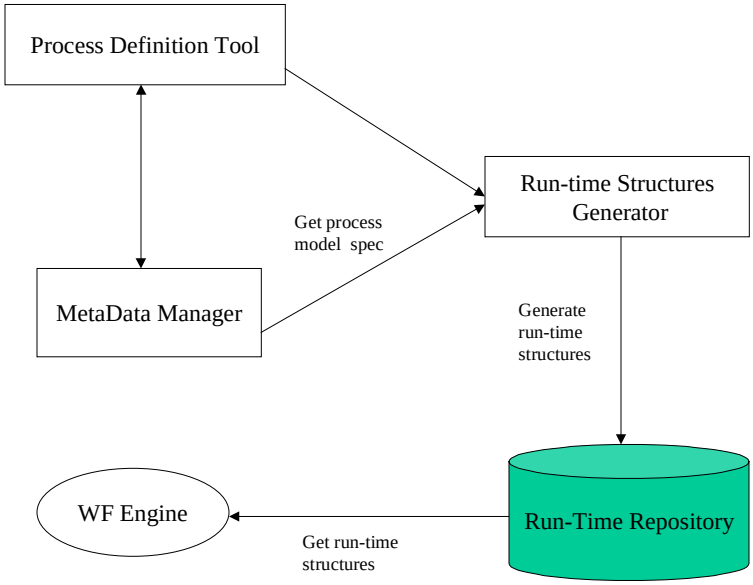


Figure 3.4 Run-time Workflow Structures Generation

The Process Definition Tool is used to define new process models and edit existing ones. The process model specification is saved in the MetaData Manager. The Run-time Structures Generator is invoked to generate run-time workflow structures according to the process model specification. The generated run-time workflow structures will be written to a serialization file and stored into the run-time repository to be used by the Workflow Engine to execute the workflow instance. The process model specification is separated from the workflow execution environment by run-time workflow structures generation. Dynamic changes to a business process during process

enactment can be accomplished by modifying the corresponding run-time structures of the process model.

3.4.2 Generation of Activity Code

Figure 3.5 shows how the main components are related to each other during the generation of activity code.

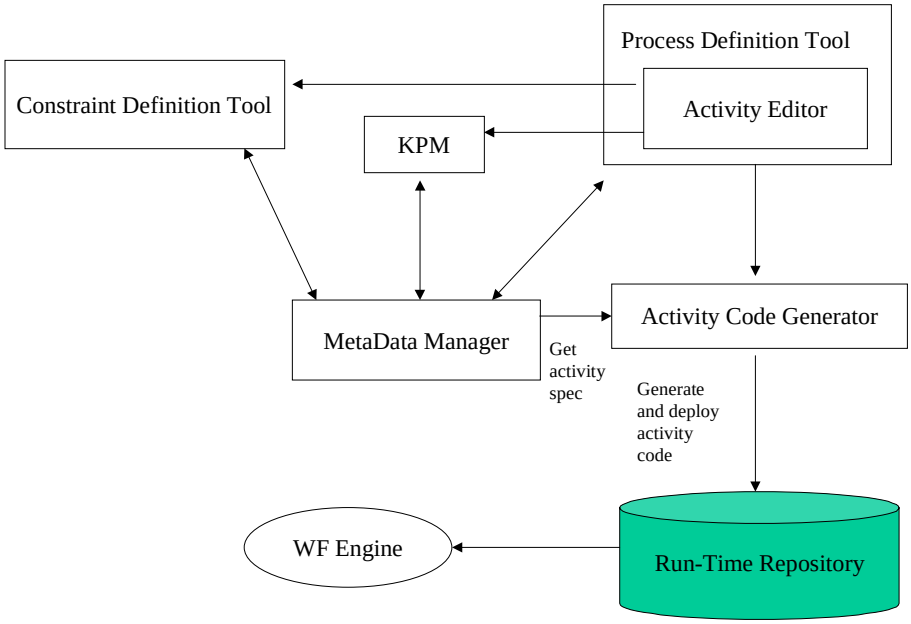


Figure 3.5 Activity Code Generation

The Activity Editor is a component of the Process Definition Tool. It is used to define new activities and edit existing activities in a process model. While defining an activity, the Activity Editor may invoke KPM to define external events in the activity, or it may invoke the Constraint Definition Tool to define e-service request constraints, depending on the type of tasks the activity is going to perform. Activity, event, and e-service request constraint specifications are stored in the MetaData Manager. The

Activity Code Generator is invoked by the Process Definition Tool to generate the Java code according to the activity specifications and to compile them. Finally, the compiled Java class files are placed into a run-time repository ready for the Workflow Engine to use.

3.4.3 Summary

The chapters that follow will present the details of the code generation approach to support the dynamic ISEE workflow management system. Chapter 4 presents the Dynamic Workflow Model specification. Chapter 5 focuses upon run-time workflow structures generation, and Chapter 6 focuses upon the code generation for activities.

CHAPTER 4
DYNAMIC WORKFLOW MODEL

The rapid growth of the Internet has made it necessary for different organizations to form virtual enterprises to do business together in order to stay competitive in the market. In order to work in a virtual enterprise environment, workflow systems have to be able to manage business processes across enterprise boundaries. As the processes and policies of a virtual enterprise are much more fluid, the inter-enterprise workflow system is required to have the capability of accommodating the changing business policies and strategies of participating organizations, handling expected and unexpected situations, and support dynamic changes to business processes.

In [MEN01], a Dynamic Workflow Model is proposed to model inter-enterprise business processes. This chapter describes the modeling constructs of the Dynamic Workflow Model. Section 4.1 summarizes WfMC's Workflow Process Definition Language (WPDL) [WFM99a] upon which the Dynamic Workflow Model is based. Section 4.2 presents the specifications for different entities in the Dynamic Workflow Model. Section 4.3 is a summary of dynamic properties provided by the Dynamic Workflow Model.

4.1 WfMC's WPDL

The Workflow Management Coalition (WfMC) [WFM99a, WFM99b] was founded in August 1993. It has been established to identify functional areas and develop appropriate specifications for implementation in workflow products. The Coalition's

mission is to promote the use of workflow technology through the development of standards for software terminology, interoperability, and connectivity between workflow products.

The Workflow Process Definition Language (WPDL) is one of the major contributions of WfMC. It is a language for describing workflow models in a structured and standardized manner for the interchange of process definitions. With WPDL, a set of activities can be defined with conditioned transitions to enable conditioned branching from activities to activities. In WPDL, the activity information include the specifications of Join and Split constructs and their constraints (AND, OR, and XOR). These constructs help define the structural relationships and constraints among activities. In a process model defined with WPDL, the data flows are implicitly defined by making use of global variables, which are called “workflow reference data”. This makes the data relationship between activities unclear. Also a process model defined with WPDL is static.

The Dynamic Workflow Model (DWM) specification is an extension to WfMC’s WPDL [WFM99a] to support the dynamic ISEE WfMS. Extensions and modifications to the WPDL include more modeling constructs such as connectors, events, triggers, rules, data flow, and e-service requests.

Graphically, a Dynamic Workflow Model can represent a business process using a dynamic process model diagram. An example process model diagram, named “Order Processing” is shown in Figure 4.1. This figure will be used throughout the remainder of the chapter to illustrate modeling constructs in the Dynamic Workflow Model.

4.2 Dynamic Workflow Model Specification

In order to support the dynamic nature of business processes in a virtual enterprise, the following extensions and modifications are made to the WPDL to form the DWM specification.

1. Introduction of Connectors: The specification of Join and Split constructs and their constraints (AND, OR, and XOR) is embedded in the specifications of activities in WPDL. The specification of this control information is extracted and represented by a new modeling construct named *Connector*. The introduction of connectors separates control flow information from the activity specification so that any change made in an activity will not affect the control flow and vice versa. This separation facilitates the dynamic changes to a business process.

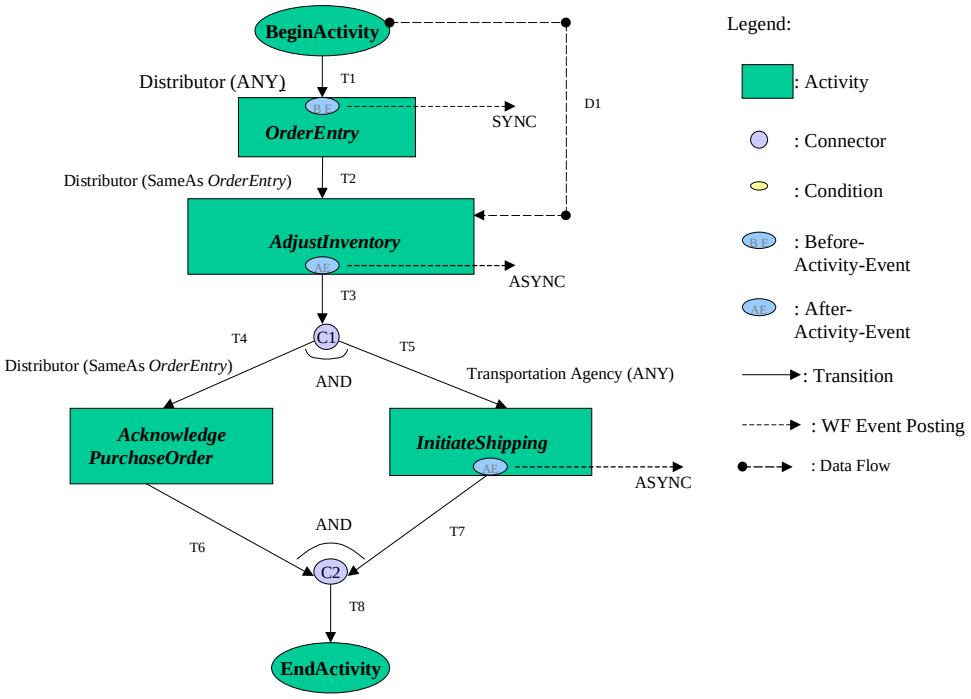


Figure 4.1 Process Model *OrderProcessing* for Distributors in the Supply Chain Community

2. Encapsulation of the activity specification: The specifications of input and output parameters are added to the WPDL's activity specifications. An activity can only reference the data passed to it by the input parameters. Secondly, the only data

outputted by an activity is through its output parameters. Thus, the activity specification is encapsulated.

3. Inclusion of e-service requests in activity specification: In an activity specified within a process model for an inter-enterprise workflow, there may exist tasks which can be performed by different business organizations. The business organizations which actually provide the requested e-services (we call these organizations service providers) can be dynamically bound to the e-service requests at run-time by the Broker Server according to constraints associated with e-service requests.

4. Introduction of Data Flows: In WPDL, data transfer among activities is through the workflow relevant data, which can be accessed by all workflow process definitions (and associated activities and transitions). In DWM, Data Flows are used to define inter-activity parameter mapping information. The introduction of Data Flows is used to eliminate any possible ambiguity in data relationship between activities.

5. Introduction of events, triggers, and rules: Events, triggers and rules are introduced to the process model specification as an important part of the DWM. Activities specified in a DWM can post events. Three different types of events can be posted, namely, Before-Activity-Event, After-Activity-Event, and External Events. A Before-Activity-Event can be posted before an activity is started. An After-Activity-Event can be posted after an activity is completed. External events are published "user defined" events posted inside an activity. When an activity is activated during the execution of a business process instance, their events are posted to automatically trigger the processing of some business rules. These rules are Event Condition Action (ECA) or Event Condition Alternative Action (ECAA) rules. The processing of these rules may cause some actions taken to enforce some business policies, regulations, or constraints. It may also cause the execution flow of the process instance to be modified, for example, by skipping some activities. With events, triggers, and rules, different users can attach different sets of rules to the same process model. The execution of different workflow instances can trigger different sets of rules. Therefore, one process model can be tailored to fit different organizations' needs.

4.2.1 Process Model Specification

A process model is specified in terms of its modeling constructs. In DWM, there are eight different modeling constructs including activity, connector, transition, block, subflow, data flow, event, and data class. The syntax of a process model specification language is shown in Figure 4.2.

The pair of brackets, "[]", indicates the clauses surrounded are optional. The **CREATED**, **DESCRIPTION**, **CLASSIFICATION**, **PRIORITY**, **time estimation** and **DOCUMENTATION** clauses specify the descriptive attributes of a process model.

Activity list contains a set of activities specified in a process model. Activities are the most important components in a process model. They are used to define tasks that need to be fulfilled for a business process. There are two special types of activities in DWM: BeginActivity which indicates the entry point of a process model or a block, and EndActivity which indicates the end point of a process model or a block. Each process model or block can have only one BeginActivity and one EndActivity.

```
PROCESS         <process id>
        [CREATED        <creation date>]
        [DESCRIPTION <description>]
        [CLASSIFICATION <classification>]
        [PRIORITY <priority>]
        [<time estimation>]
        [DOCUMENTATION <documentation>]
        <activity list>
        [<connector list>]
        <transition list>
        [<block list>]
        [<subflow list>]
        [<data flow list>]
        [<event list>]
        [<data class list>]
END PROCESS
```

Figure 4.2 Process Model Specification Language

Connector list contains a list of connectors specified in a process model. **Transition list** is a list of transitions. **Block list** is a list of blocks in a process model. **Subflow list** lists all the subflows referring to already defined process models. **Data flow list** specifies the explicit data flows defined. **Event list** has all the events while **Data class list** has all the data classes contained in a process model.

In the example process model named *OrderProcessing* (Figure 4.1), several organizations form an ISEE virtual enterprise for supply chain management: a retailer, a distributor, a manufacturer, and a transportation agency. This process model can be used by distributors to process orders issued by retailers. The distributor first gets an order from a retailer. Assuming that the ordered quantity can be satisfied, it then adjusts product quantity in the inventory. The distributor goes on to acknowledge the order and asks the transportation agency to ship the order to the retailer.

In *OrderProcessing*, the activity list contains four activities: *OrderEntry*, *AdjustInventory*, *AcknowledgePurchaseOrder*, and *InitiateShipping*. The connector list contains two connectors, namely *C1* and *C2*. The transition list contains eight transitions from *T1* to *T8*. The data flow list has one data flow *D1* from BeginActivity to the activity *AdjustInventory*. This process model does not contain any block or subflow. The external event and data class information is not shown in this example.

4.2.2 Activity Specification

Activities are the basic elements in a process model. Figure 4.3 shows the overall syntax of activity specification. The explanation of each clause is provided below.

```
ACTIVITY <activity id>
        [DESCRIPTION <description>]
        [PERFORMER <participant assignment>]
        IN_PARAMETER <input parameter list>
        OUT_PARAMETER <output parameter list>
        [WF_EVENTS <workflow event list>]
        [ACTIVITY_VAR <variable list>]
        IMPLEMENTATION <activity body>
END ACTIVITY
```

Figure 4.3 Activity Specification Language

The **activity id** is a unique identifier (or activity name) for the activity in a process model. The **DESCRIPTION** clause contains a text string describing what the activity does.

The **PERFORMER** clause specifies the *business type* of the business organization whose e-services are going to be requested in the activity body (e.g., retailer as a business type). The details of the **PERFORMER** clause is as follows:

PERFORMER <business type name> (<performer selection constraint>)

There are four types of performer selection constraints which can be specified to select a proper performer (i.e., an e-service provider), namely, *constant*, *any*, *same_as*, and *variable*:

- CONSTANT: The performer is a particular organization specified by the name of the organization. For example:
  PERFORMER *Distributor* (CONSTANT *IBM*) specifies the service provider is IBM.

- ANY: The performer can be any suitable organization of the specified business type. With this type of performer, the e-service requests specified in the activity body can be dynamically bound to a proper organization at run-time through the interaction between the Workflow Engine and the Broker Server. An example is given below:
  PERFORMER *Distributor* (ANY) specifies the service provider can be any distributor.

- SAME_AS: The performer of the activity being defined should be the same performer that performed another activity. In the example shown below, the performer of this activity should be the same performer which performed Activity1.
  PERFORMER *Distributor* (SAME_AS *Activity1*)

- VARIABLE: The performer of the activity being defined is specified by the output parameter of another activity. In the following example, the performer of the current activity is computed by *Activity2* and represented as the output parameter *bestManufacturer*.
  PERFORMER *Manufacturer* (VARIABLE *Activity2.bestManufacturer*)

The performer selection constraints categorized as *same_as* and *variable* restrict the selection of the performer for the activity being defined to performers (i.e. e-service providers) specified in other activities.

The **IN_PARAMETER** clause specifies the input data that will be used in the *activity body*. The **OUT_PARAMETER** clause specifies the data that will be produced by the activity.

The **WF_EVENT** clause specifies the workflow events that this activity posts. The workflow events can be synchronous or asynchronous Before-Activity-Events, synchronous or asynchronous After-Activity-Events. These workflow events are automatically generated.

The **ACTIVITY_VAR** clause defines a list of variables that can be used inside the *activity body*.

In the **IMPLEMENTATION** clause, the **activity body** specifies a set of task items that need to be done inside the activity. Three types of task items can be defined inside the activity body: e-service request, inline code, and event posting.

E-service request: In a process model defined for an inter-organizational workflow, the e-service requests are the most important task items in an activity. There may be several e-service requests in one activity. In DWM, the e-service requests specified within one activity have to be bound to one service provider. The syntax of the e-service request in an activity-body is shown in Figure 4.4.

The **service name** is the name of the e-service and the **operation name** is the actual operation performed in the e-service. The **in_attributes mapping** in the **INPUT** clause specifies the way to map the activity data (input parameters of the activity and

activity variables) to the input attributes of the e-service request. The **out_attributes mapping** in the **OUTPUT** clause specifies the way to map the output attributes of the e-service request to the activity data (output parameters of the activity and activity variables). The **CONSTRAINT** clause specifies constraints in selecting suitable e-services.

```
E-SERVICE REQUEST<service name>.<operation name>
        INPUT <in_attributes mapping>
        [OUTPUT <out_attributes mapping>]
        CONSTRAINT <constraint definition>
END_SERVICE REQUEST
```

Figure 4.4 E-Service Request Specification Language

In-line code: Programming code can be a task item to compute or make a variable assignment. The Java programming language syntax is adopted for the in-line code.

External event posting: Task items in the **activity body** can also be posting of events. These events are considered as external events as they are defined outside of the activity. The event can be either synchronous or asynchronous. Synchronous events will be posted to the Event-Trigger-Rule (ETR) Server to trigger rules and asynchronous events posted to an Event Server.

At run-time, when an activity is scheduled, it will perform tasks in the task list in the sequence task items are specified. The activity either reaches a finished status or time-out status.

In Figure 4.1, *OrderEntry*, *AdjustInventory*, *AcknowledgePurchaseOrder*, and *InitiateShipping* are activities. Workflow events such as a synchronous Before-Activity-Event is posted before activity *OrderEntry*, and an asynchronous After-Activity-Event is

posted after activity *AdjustInventory* and *InitiateShipping* respectively. The performer for activity *OrderEntry* is Distributor (ANY) meaning any distributor can be the service provider. The performer for *AdjustInventory* is Distributor (Same_As *OrderEntry*) meaning service provider for activity *AdjustInventory* has to be the same as activity *OrderEntry*'s.

There are two special types of activities: BeginActivity and EndActivity. These special activities do not have any tasks.

- BeginActivity: Figure 4.5 gives the syntax of the BeginActivity specification language. BeginActivity is the entry point of a process model or a block indicated by the **process/block id**. The **DESCRIPTION** clause contains some text string about the BeginActivity. The **IN_PARAMETERS** clause contains the input data for a process model or a block. The **LOOP** clause is also optional. It is for BeginActivity specified in a block, indicating whether the loop is DO …WHILE or REPEAT … UNTIL.

```
BEGINACTIVITY          <process/block id>
        [DESCRIPTION <description>]
        IN_PARAMETERS <input parameters>
        [LOOP <loop control information>]
END BEGINACTIVITY
```

Figure 4.5 BeginActivity Specification Language

- EndActivity: The syntax for the EndActivity Specification Language is shown in Figure 4.6. EndActivity is the exit point of a process or block indicated by **process/block id**. The **DESCRIPTION** clause contains information about the EndActivity. The **OUT_PARAMETERS** clause contains the output data of the entire process or block.

```
ENDACTIVITY  <process/block id>
        [DESCRIPTION <description>]
        OUT_PARAMETERS <output parameters>
END ENDACTIVITY
```

Figure 4.6 EndActivity Specification Language

4.2.3 Connector Specification

In DWM, connectors are control flow constructs modeling the aggregation properties such as Join and Split and their constraints (OR, AND, and XOR). Figure 4.7 gives the syntax of Connector specification.

```
CONNECTOR    <connector id>
        [DESCRIPTION <description>]
        AGGREGATION < aggregation information>
END CONNECTOR
```

Figure 4.7 Connector Specification Language

The **connector id** is a unique identifier for the connector specified in a process model. The **DESCRIPTION** clause describes the purpose of the connector. The **AGGREGATION** clause contains the aggregation information of the connector, such as JOIN or SPLIT characterization.

In Figure 4.1, *C1* and *C2* are connectors. The aggregation information for *C1* is AND SPLIT, and the aggregation information for *C2* is AND JOIN.

4.2.4 Transition Specification

The transition construct connects other entities specified in a process model, such as activities, subflows, connectors, blocks, BeginActivity, and EndActivity. In DWM, transitions and connectors together form the control flow of a business process model. Figure 4.8 shows the Transition specification language.

The **transition id** is a unique identifier for the transition specified in a process model. The **DESCRIPTION** clause contains information about what the transition does. The **FROM/TO** clause specifies the source entity and the destination entity of the transition. Activities, subflows, connectors and blocks can serve as source or destination

entity. BeginActivity can only be the source entity and EndActivity can only be the destination entity. The optional **CONDITION** clause defines the condition attached to the transition. At run-time, transitions are enabled, or disabled, according to the evaluation of their conditions. **The DATA-MAPPING** clause specifies the parameter mapping between two activities. This is also used to define the implicit data flow in DWM.

```
TRANSITION    <transition id>
        [DESCRIPTION <description>]
        FROM <entity id> TO <entity id>
        [CONDITION <transition condition >]
        [DATA_MAPPING <inter-activity parameter mapping >]
END TRANSITION
```

Figure 4.8 Transition Specification Language

In Figure 4.1, there are eight transitions from *T1* to *T8*. Each of them has a source entity and a target entity. Some of them have conditions and data mappings.

4.2.5 Block Specification

The block specification is very similar to the process specification. The differences between these two are as follows.

- A block is identified by its block id.
- A block can be specified as a loop indicated by its BeginActivity. This kind of block executes a set of activities defined in it repeatedly and stops executions based on the evaluation of the condition associated with the loop. BeginActivity in a process model can not specify a loop.
- A block can also be an inline block used to group many activities together in a complex process model.

4.2.6 Subflow Specification

In defining a complex process model, a subflow is often used to refer to an existing process model to simplify the specification of the process model being defined. Figure 4.9 shows the syntax of subflow specification.

```
SUBFLOW        <subflow id> <referenced process id>
        [DESCRIPTION <description>]
        [IN_PARAMETER <input parameter list>]
        [OUT_PARAMETER <output parameter list>]
END SUBFLOW
```

Figure 4.9 Subflow Specification Language

The **subflow id** is a unique identifier for the subflow defined in the process model. The **reference process id** is the unique identifier for the referenced process model, which is previously defined. The **DESCRIPTION** clause contains a text string about the subflow. The **IN_PARAMETER** clause specifies the input parameters and the **OUT_PARAMETER** clause specifies the output parameters of the subflow. Both clauses are optional as the referenced process model has everything defined.

4.2.7 Data Flow Specification

A data flow is used to explicitly specify data passed from one activity to another if the two activities are not directly connected to each other with transitions. Figure 4.10 gives the syntax of data flow specification.

```
DATAFLOW                <data flow id>
        [DESCRIPTION <description>]
        FROM <activity id> TO <activity id>
        <parameter mapping info>
END DATAFLOW
```

Figure 4.10 Data Flow Specification Language

The **data flow id** specifies the unique identifier of the data flow. The **DESCRIPTION** clause describes the purpose of the data flow, while the **FROM/TO** clause indicates the data passed from the source activity to the target activity. The **Parameter mapping info** specifies how to map the source activity's output parameters to the target activity's input parameters.

In Figure 4.1, there is one data flow identified as *D1*. Its source entity is BeginActivity and its target entity is activity *AdjInventory*.

4.2.8 External Event Specification

Inside the activity body, external events can be posted. These events are already defined and are imported to the process model prior to its definition. While defining a process model with DWM, **event id** is needed in order to get appropriate events from the MetaData Manager to be posted in an activity. Figure 4.11 shows the syntax of event specification.

```
EVENT <event id>
        [DESCRIPTION <description>]
END EVENT
```

Figure 4.11 Event Specification Language

4.2.9 Data Class Specification:

The data classes define data types used to specify data flows between activities specified in a process model. The specification is given in Figure 4.12.

```
DATACLASS     <data class id>
        [DESCRIPTION <description>]
END DATACLASS
```

Figure 4.12 Data Class Specification Language

4.3 Dynamic Properties of the DWM

The Dynamic Workflow Model was briefly described in this chapter. For additional details, please refer to [MEN01]. The DWM extended WPDL to provide the following dynamic properties needed to support the requirements of inter-enterprise business processes:

- Active: As business events and rules are integrated with business processes, a business process is active in that its enactment may post synchronous and/or asynchronous events to trigger the business rules. Synchronous events posted may trigger rules, which may cause dynamic changes to the business process. These rules can be defined by the process model designer or the organization, which initiates the process instance. Asynchronous events can be used for notifications of the processing milestones of an enacted business process. Organizations interested in these events can subscribe to them and define corresponding business rules. The processing of these rules may cause enactment of other business processes.

- Flexible: As e-services are integrated with business processes, a business process is flexible in the sense that the actual service provider is not bound until the enactment of the process. The constraint-based dynamic service binding is provided by the Broker Server. At run-time, e-service requests specified in activities are sent to the Broker Server. The Broker Server is responsible for finding the suitable service providers for the e-service requests based on the constraints specified in them. With dynamic service binding, the enactment of a process is not affected by the availability of service providers providing particular e-services.

- Adaptive: Process models defined in the dynamic workflow model are adaptive in the sense that they can be modified to adapt to the changing virtual enterprise environment. During the process enactment, dynamic changes to the business processes can be made by modifying the appropriate run-time workflow structures. These modifications may be caused by the processing of business rules triggered by synchronous workflow events, or by users who monitor the processing of the process instance.

- Customizable: In a virtual enterprise, inter-organizational process models can be designed for participating business organizations to enact. In each enactment of a process model, the organization which initiates the process model may need to customize the process model to suit its business policies, constraints or regulations. To achieve the customization, the organization could define its own set of business rules to the process model to dynamically change the process during each enactment.

For conducting collaborative e-business over the Internet, the integration of events and rules into a process and the dynamic binding of services are especially important because collaborative e-business operates across enterprise boundaries. Different business rules and policies may apply in different enterprises. And, during the lifetime of the virtual enterprise, these rules may change more frequently than the actual process. The ETR service allows business rules to be specified and managed separately from processes, but inter-operated with them through events and triggers. When a virtual enterprise rules or policies change, the rules in the ETR Server can be changed dynamically without having to change the definition of the process.

CHAPTER 5
RUN-TIME WORKFLOW STRUCTURES GENERATION

The build-time architecture of the ISEE WfMS, introduced in Chapter 3, is designed to support a "code generation" approach to realize a dynamic workflow management system. Based on the specification of a workflow model defined using the Process Definition Tool and stored in the MetaData Manager, run-time workflow structures and activity code are generated. *Run-time workflow structures* are used by the Workflow Engine to schedule and execute a workflow instance. The generated *activity code* is directly executed to efficiently request the binding of e-service requests to service provider for execution. Changes to the control and data flows of an executing workflow instance can be made dynamically at run-time by adding, modifying, or deleting run-time structures. Changes to an activity (e.g., e-service requests, performer, performer constraints, etc.) can be made using the Process Definition Tool. The code for the modified activity is re-generated and re-loaded using Java's class reloading capability. Thus flexibility is maintained without sacrificing performance.

In Chapter 6, the details of activity code generation will be provided. This chapter gives a detailed description of the generation of run-time workflow structures. Section 5.1 discusses the main components involved in the run-time workflow structures generation, followed by an explanation of the generated run-time workflow structures. Section 5.3 presents the details of the implementation, and Section 5.4 gives a summary of this chapter.

5.1 Run-Time Workflow Structures Generation

The main components involved in the run-time workflow structures generation and their interactions are shown in Figure 5.1.

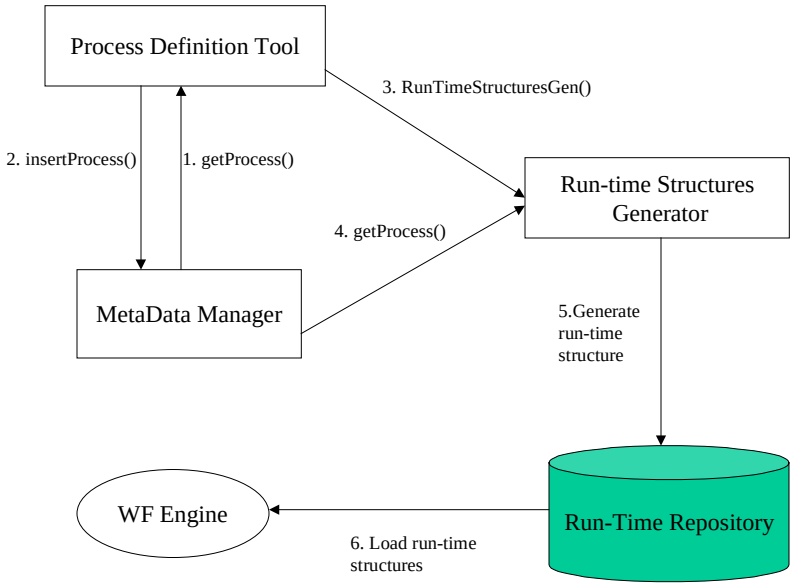


Figure 5.1 Run-Time Workflow Structures Generation

Steps 1 and 2: The Process Definition Tool is used to define new process models and edit existing process models. In order to edit an existing process model, the Process Definition Tool sends a request to the MetaData Manager to retrieve the process specification for modification (Step 1). A newly defined process or a modified process specification is sent to the MetaData Manager for persistent storage (Step 2).

Steps 3, 4 and 5: To generate code for a process model, the Process Definition Tool invokes the Run-Time Structures Generator to generate the run-time workflow structures for the process model (Step 3). A reference to that process model is passed to

the Run-Time Structures Generator. Also passed to the Generator is the path of the directories to store the serialization files for the generated run-time workflow structures. Using the reference, the Generator retrieves the corresponding process specification from the MetaData Manager (Step 4). Based upon the process specification, the run-time structures are generated and written into a serialization file (Step 5). The serialization file is stored in the run-time repository ready for the Workflow Engine to use.

Step 6: When a process instance is enacted, the Workflow Engine loads the run-time structures generated for the corresponding process model and schedules the activities in it according to the run-time structures (Step 6).

By using run-time structures for the execution of an instance of a process model, dynamic changes to a business process can be achieved by changing (i.e., adding/deleting/modifying) the structures at run-time. Changes to the structures can be made by invoking methods via the API of the Workflow Engine (e.g., from an application program or from the action of a rule triggered by a workflow event.).

5.2 Generated Run-Time Workflow Structures

There are basically three types of run-time structures generated: entity structures, control flow structures, and data flow structures.

Entity structures provide the Workflow Engine with the necessary information to schedule and execute the following entities in a workflow model: activities, BeginActivity, EndActivity, blocks, and subflows. The contents for the run-time structures for these entities correspond to their specifications as presented in Chapter 4. In other words, these structures contain the meta-data for these entities needed to execute a process instance by the Workflow Engine.

Control flow structures capture the control dependencies among entities of a process model. A control flow structure is generated for each of the following entities of a process model: activity, subflow, block, connector, and EndActivity. During the execution of a process instance, the Workflow Engine determines whether the next entity can be scheduled according to its corresponding control flow structure.

Data flow structures capture data dependencies in a process model specification. In particular, a data flow structure specifies the parameter mapping information between two entities in a process model.

5.2.1 Entity Structures Generation

The run-time entity structures generation for activities, BeginActivity, blocks, subflows and EndActivity is similar and comparatively easy. The discussion here will be focused on the generation of the run-time structures for activities and how the Workflow Engine uses them.

The run-time structures generated for activities are intended to provide necessary information to the Workflow Engine for the purpose of scheduling the activities and executing the generated activities' code (to be described in detail in Chapter 6). As shown in Figure 5.2, the run-time structure generated for an activity consists of the following: activity id, input and output parameters, performer, performer selection constraint and e-service requests and their constraints. The information for this structure can be obtained in a straightforward manner from the specification of the activity (stored in the MetaData Manager).

The activity id provides the identifier to facilitate the Workflow Engine to schedule and execute the appropriate activity in a process model. The input and output parameters are included so that when an activity is scheduled, the Workflow Engine can

obtain the right values to initialize its input parameters and perform the data mapping accurately during the process enactment.

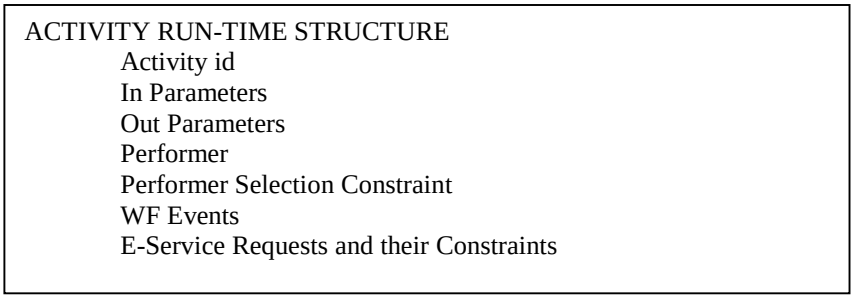


Figure 5.2 Activity Run-time Structure

From the performer and performer selection constraint, the Workflow Engine can tell whether there is a need for dynamic service binding. If the performer selection constraint is “ANY”, a call to a Broker Server to perform dynamic service binding is needed. In this case, the Workflow Engine will load the corresponding activity code with the following information: a vector of values for the initialization of the input parameters, the process model name, the business type obtained from the performer information, and the hash table with e-service request names and constraints. The loaded activity code will contact the Broker Server for the dynamic service binding, after which it will perform the tasks specified in the activity and return the result.

If the performer selection constraint is not “ANY”, there is no need to call a Broker Server for dynamic service binding. The Workflow Engine will obtain the service provider’s URL according to the performer selection constraint, and pass it to the appropriate activity code. The activity code uses this service provider’s URL to invoke the services to perform its e-service requests.

Information about workflow events specified in the activity is also contained in the activity run-time structure so that the Workflow Engine knows what types of events can be posted. E-Service request constraints in the run-time structures for activities are stored in a hash table with e-service request names as the keys and e-service request constraints as the corresponding values.

For example, the specification for an activity named "*InitiateShipping*" in the process model "Order Processing" (from Figure 4.1 in Chapter 4) is repeated in Figure 5.3. The corresponding run-time structure that is generated is shown in Figure 5.4.

```
ACTIVITY  InitiateShipping
        DESCRIPTION "Initiate Shipping: ship the order to the retailer."
        IN_PARAMETERS UserInfo user_info, ProdDetailDesc
                        prod_detail_desc, int quantity, String delivery_date

        OUT_PARAMETERS  boolean shipping_status
        PERFORMER TransportationAgency (ANY)
        WF_EVENTS   asyn_after
        IMPLEMENTATION
            ESERVICE ShipOrder.InitiateShipping
                INPUT user_info, proc_detail_desc, quantity, delivery_date
                OUTPUT shipping_status
                CONSTRAINT
            END_ESERVICE
        END_IMPLEMENTATION
END_ACTIVITY
```

Figure 5.3 Specification for Activity *InitiateShipping* in Process Model "Order Processing"

5.2.2 Control Flow Structures Generation

Control flow structures capture the control dependencies among entities of a process model. A control flow structure is generated for each of the following entities of a process model: activity, subflow, block, connector, and EndActivity. During the

execution of a process instance, the Workflow Engine determines whether the next entity can be scheduled according to its corresponding control flow structure.

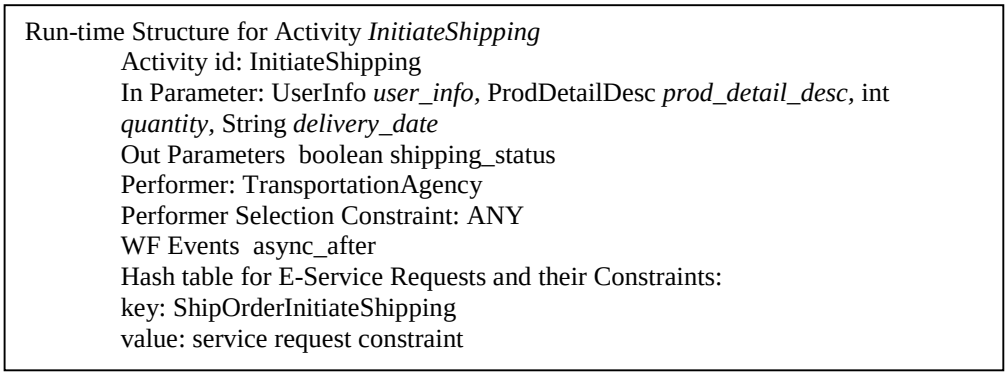


Figure 5.4. Run-time Structure for Activity *InitiateShipping*

As shown in Figure 5.5, a control flow structure consists of three attributes: entity name, aggregation property, and transition. Entity name is the name of the entity corresponding to the control flow structure. If the entity of the control flow structure is a JOIN connector, the aggregation property of the control flow structure is the same as the aggregation property of the JOIN connector (i.e., AND, OR or XOR). Otherwise the aggregation property is SIMPLE. A control flow structure may contain one or more transitions. If the entity of the control flow structure is a JOIN connector, there are multiple transitions in this control flow structure. For all the other entities, there is only one transition in the control flow structure.

Figure 5.5. Control Flow Structure

The generation of the control flow structures is much more involved than the generation of entity structures. First, a list of transitions specified in a process model is obtained and the target entities are determined. For each target entity, a control flow structure is generated. The entity name of the control flow structure is obtained by getting the target entity's name. The control flow structure's aggregation type is set according to that of the target entity. For example, in Figure 4.1, in the control flow structure for JOIN connector *C2*, the entity name is C2 and the aggregation property is AND. In the control flow structure for activity *OrderEntry*, the entity name is OrderEntry and the aggregation property is SIMPLE.

For the transition attribute of a control structure, a hash table is generated. There are two situations to be considered. If the target entity of the transition is not a JOIN connector, the generated hash table contains only one entry, with the name of the source entity of the transition as the hash key, and the transition name with its condition as the value.

If the target entity is a JOIN connector, there are multiple transitions pointing to the same target entity. Each transition has an entry in the hash table. Again, the source entity's name of each transition is the hash key and the corresponding transition name (and its condition) is the value.

The control flow structure is generated with the hash table, the entity name, and its aggregation type. If the aggregation type is AND, the entity of the control flow structure cannot be initiated until the conditions on all incoming transitions evaluate true. If the aggregation type is XOR, the entity of the control flow structure will be initiated

when the conditions of any one of the incoming transitions evaluates true. The XOR siblings will be added to each transition when the control flow structure is generated.

For example, in process model "Order Processing" (Figure 4.1), transition T6 and T7 point to the same entity: connector *C2*. The control flow structure generated for *C2* contains two transitions, namely, T6 and T7. The hash table in the control flow structure will look like Figure 5.6. The hash table has transitions' source entity names as keys and transitions and their conditions as values.

AcknowledgePurchaseOrder	T6 and T6 condition
InitiateShipping	T7 and T7 condition

Figure 5.6 Hash table for Transition in Control Flow Structure for *C2*

If the aggregation property for *C2* in Figure 4.1 is XOR. T6 will be considered as T7's sibling, and vice versa. At run-time, *C2* is initiated when the transition conditions of any one of the incoming transitions (T6 and T7) evaluates true and the transition is enabled while all its siblings are disabled. In this way, the initiation of *C2* is guaranteed to occur once.

In both cases, the type of the source entity of each transition is also checked. If the source entity is an XOR SPLIT connector, siblings of each transition are added when the control flow structure is generated. At run-time, the Workflow Engine evaluates conditions of each transition in the sequence specified, selects one to enable according to its evaluation, and disables all of its siblings.

5.2.3 Data Flow Structures Generation

The generated data flow structures are intended for the Workflow Engine to map data outputted by entities (such as activities, subflows, and blocks) which have

successfully completed to entities which need these data to proceed. Data flow structures inform the Workflow Engine where the data comes from, where it should go, and provide the mapping information between parameters from different entities. Data flow structures are generated from the data mapping information found in a Transition specification or from an explicit Data Flow specification.

For each transition specified in the process model, if the transition has a data mapping list defined, a data flow structure is generated. As shown in Figure 5.8, a data flow structure has a source entity, a target entity, and parameter mapping information. The source entity indicates where the data comes from. The target entity specifies where the data goes. The parameter mapping information specifies how the output parameters of the source entity are mapped to the input parameters of the target entity.

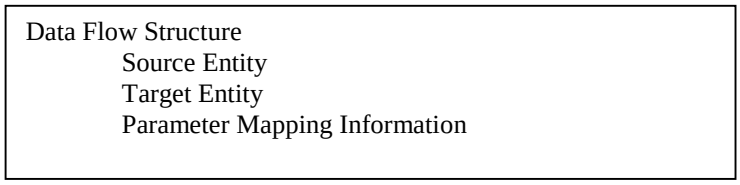


Figure 5.7 Data Flow Structure

The source entity of the data flow structure is not necessarily the same as the source entity of the corresponding transition. If the transition directly connects two entities such as activities, blocks or subflows, the source entity of the data flow structure generated is the same as that of the transition. However, if the transition connects a connector to an activity, a block or a subflow, the source entity of the generated data flow structure cannot be the connector as connectors have no input nor output parameters. In this case, the source entity is the preceding entity which has input and output parameters.

To illustrate, let's look at the process model "Order Processing" (Figure 4.1). In "Order Processing", the source entity of T3 is activity *AdjustInventory* and its target entity is connector *C1*. T3 does not have a data mapping list because its target entity *C1* has no input and output parameters. In this case, there is no data flow structure generated for T3. The target entities for T4 and T5 are activities. Both of them have a non-empty data-mapping list. A data flow structure is generated for T4 and T5. The source entity of the generated data flow structures is activity *AdjustInventory* instead of connector *C1*. The target entity of a data flow structure is the same as that of the transition which has a non-empty data mapping list.

The parameter mapping information of a generated data flow structure has to be obtained by analyzing the data mapping list associated with the transition. The output parameter names of the source entity and the input parameter names of the target entity are obtained by parsing each element packed in the data mapping list defined in the transition.

The generation of data flow structures based on Data Flow specifications of a process model is straightforward. The source entity and the target entity of this kind of data flow structure are the same as those of the Data Flow entities. The parameter mapping information can be obtained directly from the data mapping list defined in the Data Flow specifications.

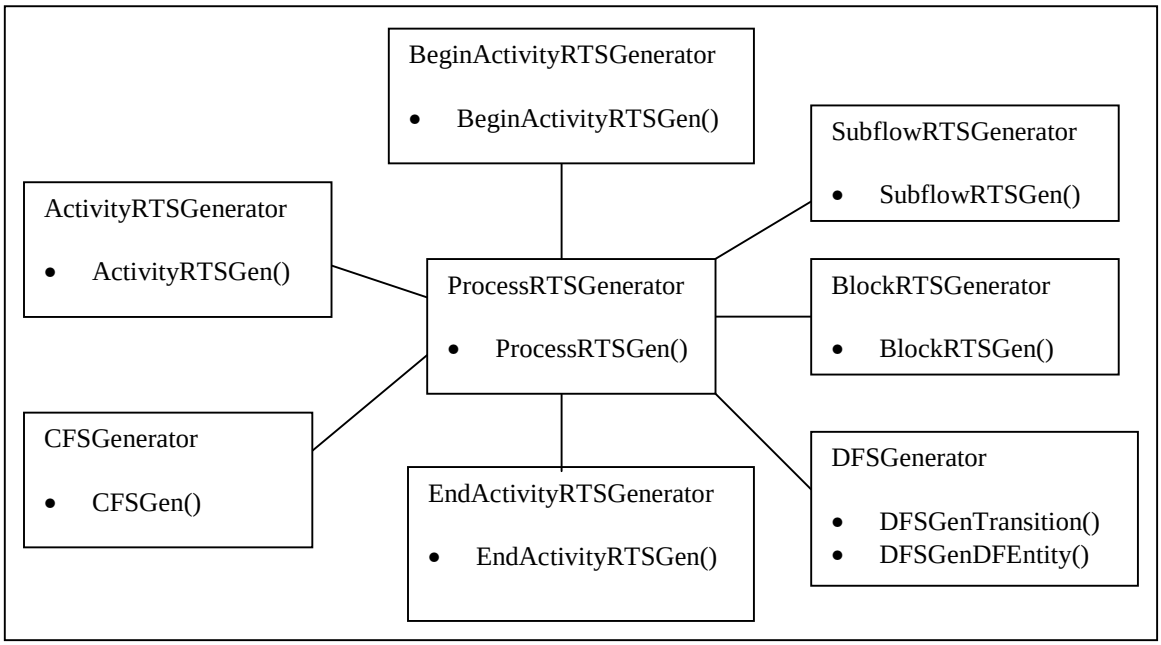
5.3 Implementation of the Run-time Structure Generator

Figure 5.9 is the class diagram of the Run-time Structures Generator, showing the key classes and methods that are involved in the run-time workflow structures generation.

5.3.1 ProcessRTSGenerator

The ProcessRTSGenerator class is the interface class which provides the API for an external component to use the Run-time Structures Generator. When ProcessRTSGenerator is instantiated, a reference to the MetaData Manager is passed to the constructor. This class has one public method:

- ProcessRTSGen(): This method is called to generate run-time workflow structures for a process model according to its specification. The process model specification is fetched from the MetaData Manager. It calls the methods of other classes to generate the structures.



Note: For readability of the diagram, the following abbreviations are used, RTS: Run-Time Structures, DFS: Data Flow Structures, and CFS: Control Flow Structures.

Figure 5.8 Class Diagram of Run-time Structures Generation

5.3.2 BeginActivityRTSGenerator

This class is used by the ProcessRTSGenerator to generate the run-time structures for the BeginActivity specified in the process model. A reference to the MetaData Manager is passed to this class. There is one public method in this class:

- BeginActivityRTSGen():This method is invoked by the ProcessRTSGenerator with the process name and the empty BeginActivity run-time structure as arguments. The BeginActivity specification is obtained from the MetaData Manager and the run-time structure is filled in with the input parameter information such as parameter name and parameter type.

5.3.3 ActivityRTSGenerator

This class is called by the ProcessRTSGenerator to generate the run-time structures for activities specified in the process model. There is one public method in this class:

- ActivityRTSGen(): This method is called by the ProcessRTSGenerator with the process name, the activity name and an empty run-time structure for the activity as arguments. The activity specification is obtained from the MetaData Manager. Based on the activity specification, the input parameter information and the output parameter information of the run-time structure are filled in. The performer information is stored in the MetaData Manager as a string. This string is parsed and the business type and the performer selection constraint are extracted put into the run-time structure. From the e-service requests specified in the activity, e-service request constraints can be obtained and put into a hash table with e-service request names as the keys and the constraints as the values. The performer information and the e-service request information are also filled in the run-time structure for the activity.

5.3.4 CFSGenerator

CFSGenerator stands for Control Flow Structures Generator. This class is called by the ProcessRTSGenerator to generate the run-time control flow structures based on transitions specified in the process model. There is one public method in this class:

- CFSGen(): CFSGen stands for Control Flow Structures Generation. This method is called by ProcessRTSGenerator with the process name, the transition name, and the empty control flow structure as arguments. If the target entity is a JOIN connector, all the transition names pointing to the same target entity are obtained from the aggregation property of the specification of the target entity. The specifications for

these transitions are fetched from the MetaData Manager. All the transitions are put into one hash table, with the source entity name as the keys and the transition names and their corresponding conditions as the values. This hash table is put in the control flow structure. If the target entity is not a JOIN connector, one hash table is created for each transition. Each hash table is a part of the corresponding control flow structure with one transition in it. The aggregation property and the entity name of the control flow structure are also put in the control flow structure.

5.3.5 DFSGenerator

DFSGenerator stands for Data Flow Structures Generator. This class is called by the ProcessRTSGenerator to generate the run-time data flow structures based on data flows specified in transitions and data flows specified in data flow entities in the process model. There are two public methods in this class:

- DFSGenTransition(): This method is called by the ProcessRTSGenerator with the process name, the transition name, and the empty data flow structure as arguments. This method is used to generate a data flow structure when the data flow information is implicitly specified in the associated transition. The data flow is specified in the data mapping list of the transition specification. The source entity of the implicit data flow structure is obtained by examining each element in the data mapping list. The parameter mapping information is also obtained by analyzing each element in the data mapping list. The target entity of the data flow structure is the same as the target entity of the transition.

- DFSGenDFEntity(): This method is called by the ProcessRTSGenerator with the process name, the data flow name, and the empty data flow structure as arguments. This method is used to generate a data flow structure when the data flow information is explicitly defined in a data flow specification. The data flow is specified according to the data flow entity specification. The source entity, the target entity and the parameter mapping information of the data flow structure are the same as those of the data flow entity specified in the process model. All of the information is filled in the empty data flow structure.

5.3.6 SubFlowRTSGenerator

This class is called by the ProcessRTSGenerator to generate the run-time subflow structures based on subflow specifications in the process model. There is one public method in this class:

- SubFlowRTSGen(): This method is called by the ProcessRTSGenerator with the process name, the subflow name, and the empty subflow run-time structure as

arguments. The referenced process identifier, the input and output parameter information are filled in the subflow run-time structure so that during a process enactment, the Workflow Engine could load in the referenced process model and execute it with given input parameters.

5.3.7 BlockRTSGenerator

This class is called by the ProcessRTSGenerator to generate the run-time block structures based on block specifications in the process model. There is one public method in this class:

- BlockRTSGen(): This method is called by the ProcessRTSGenerator with the process name, the block name, and the empty block run-time structure as arguments. The run-time structures for a block are generated in the same way as those generated for a process model. However, for the BeginActivity, it is necessary to check to see whether it is a loop block.

5.3.8 EndActivityRTSGenerator

This class is called by the ProcessRTSGenerator to generate the run-time EndActivity structures based on the EndActivity specifications in the process model. There is one public method in this class:

- EndActivityRTSGen(): This method is called by the ProcessRTSGenerator with the process name and the empty EndActivity run-time structure as arguments. The output parameter information is obtained from the EndActivity specification and filled in the run-time structure.

5.4 Dynamic Changes to The Business Process

As described in Chapter 4, the Dynamic Workflow Model (DWM) is dynamic in the following ways:

- Active: A DWM process is active in that its enactment may post synchronous and/or asynchronous events to trigger the business rules.

- Flexible: A DWM process is flexible in the sense that the actual service provider is not bound to an e-service request until the enactment of the process.

- Adaptive: A DWM process is adaptive in the sense that it can be modified at run-time to adapt to the changing business environment.

- Customizable: A DWM process is customizable in the sense that different organization can define its own set of business rules to associate with its instantiation of a process model.

The run-time workflow structures generation approach presented in this chapter supports the adaptive property of the DWM. The generated entity, control, and dataflow structures are used by the Workflow Engine to schedule and execute a workflow instance. Dynamic changes to the control and data flows of an executing workflow instance can be made at run-time by changing these run-time structures. For example, rules may be used to alter the execution flow of the running process instance. Such rules may be triggered by some workflow event to skip one or more activities under certain conditions. The alteration is accomplished by invoking methods via the API of the Workflow Engine to modify the run-time control flow structures and data flow structures of the running process instance. The Workflow Engine of the ISEE WfMS supports the following changes to a running process instance:

- Deletion/Addition of a Transition: The deletion/addition of a transition is accomplished by deleting/adding the corresponding control flow structure from/to the run-time workflow structures of the process instance.

- Deletion/Addition of a Data flow: The deletion/addition of a data flow is accomplished by deleting/adding the corresponding data flow structure from/to the run-time workflow structures of the process instance.

- Replace an Activity: To replace an activity with a new activity, the old activity's name is replaced with the new activity's name in the associated control flow structures and data flow structures. When the Workflow Engine is scheduling the activities, the new activity will be scheduled and its generated code (See Chapter 6) will be loaded for execution.

- Modify Transition Conditions: The Workflow Engine provides API to modify transition conditions directly.

CHAPTER 6
ACTIVITY CODE GENERATION

As described, to support a "code generation" approach to realize a dynamic workflow management system, run-time workflow structures and activity code are generated. The generation and use of run-time workflow structures to schedule and execute a workflow instance were described in Chapter 5. This chapter describes the activity code generation in detail. The generated *activity code* is directly executed to efficiently request the binding of e-service requests to service provider for execution. Changes to an activity (e.g., e-service requests, performer, performer constraints, etc.) can be made using the Process Definition Tool. The code for the modified activity is re-generated and re-loaded using Java's class reloading capability. Thus, flexibility is maintained without sacrificing performance.

The organization of the remainder of this chapter is as follows. First, the interaction among the main components involved in the activity code generation will be presented in Section 6.1. Section 6.2 explains generated activity code. The details of the implementation of the Activity Code Generator are described in Section 6.3. Section 6.4 summarizes this chapter.

6.1 Activity Code Generation

Activities are the fundamental building blocks of a business process model. In the Dynamic Workflow Model, activity information is encapsulated by specifying an activity's input and output parameters. Tasks within an activity can be specified as e-

service requests. When the activity is executed, a Broker Server will be contacted to dynamically bind the e-service requests to appropriate e-services and service providers if necessary. Within the activity specification, there is no control information such as Split or Join constructs information. Thus, the code generated for an activity contains no control flow information. Also, since the BeginActivity (which is the entry point of a business process model) and EndActivity (which is the exit point of a business process model) do not contain any tasks (e.g., e-service requests), no code is generated for those special-purpose activities.

This section explains the step-by-step process of activity code generation. Figure 6.1 shows the various components involved in the activity code generation and their interaction.

Steps 1 and 2: The Activity Editor is used to define new activities and edit existing activities. In order to modify existing activities, the activity specification is obtained from the MetaData Manager (Step 1). The specification is displayed and can be modified in the Activity Editor. A newly defined activity specification or a modified activity specification is then saved in the MetaData Manager (Step 2).

Step 3: To generate code for an activity specification, the Activity Editor invokes the Activity Translator. The Activity Translator provides Application Program Interfaces (APIs) to generate codes for all the activities in a process model (allActivityCodeGen()) or for a list of given activities (i.e., a subset) defined in a process model (listCodeGen()). In the former case, the process model name is passed. In the latter case, the process model name and the list of activity names are passed to the Activity Translator. In both

cases, the path of the Run-Time Repository is passed to the Activity Translator to place the compiled activity classes in the predetermined directory.

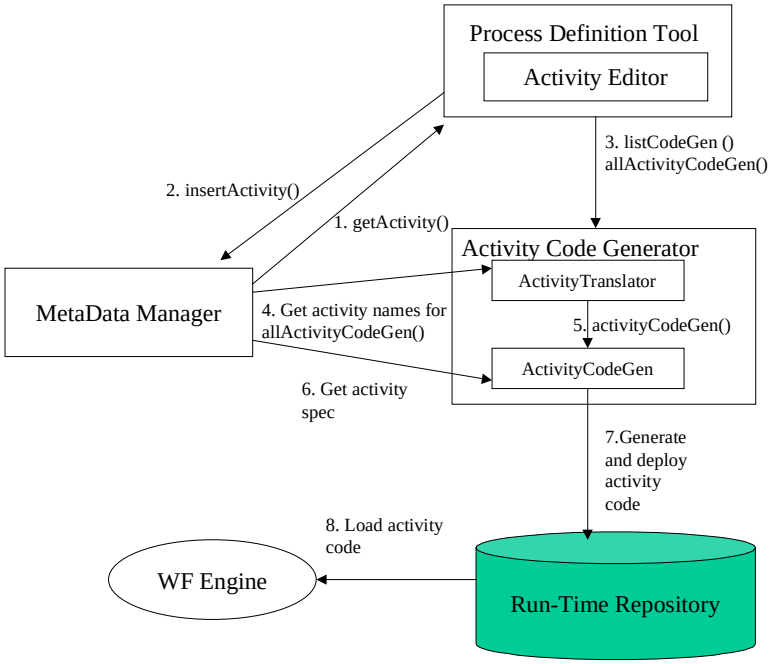


Figure 6.1 Activity Code Generation

Steps 4 and 5: The Activity Translator calls the ActivityCodeGen class to generate the actual code. In order to get information necessary for activity code generation, a reference to the MetaData Manager is made in the Activity Translator and passed to the ActivityCodeGen. If code for all activities in a process model needs to be generated, the Activity Translator first obtains the list of all activities specified in the process model from the Metadata Manger (Step 4) and then calls the ActivityCodeGen (Step 5) for each activity. In case of code generation for a list of activities, the Activity Translator invokes the ActivityCodeGen (Step 5) for each activity in the list.

Steps 6 and 7: The ActivityCodeGen class gets the activity specification from the MetaData Manager (Step 6) and generates the activity code in the form of a Java source file (.java files) and stores it in the specified directory (Step 7). The generated activity code is compiled by invoking the Javac compiler and the activity classes (.class files) are stored in the same directory.

Step 8: The Workflow Engine, when scheduling the activities for a process instance, loads the Java activity code from the Run-time Repository *as needed* to execute the corresponding activities. Note that in this manner, any change made to an activity specification (using the PDT) will be reflected in the execution of the activity as long as the re-generated code is placed in the Run-Time Repository *before* the execution of the activity.

6.2 Generated Activity Code

As described above, the activity code generated by the ActivityCodeGen class is in the form of a Java class (.java file). For each activity in a process model, an activity class is generated. The general structure of a generated activity class is shown in Figure 6.2. An example of the generated activity class for the activity *InitiateShipping* from "Order Processing" (Figure 4.1) is shown in Figure 6.3 and Figure 6.4.

Note that the different components specified in the activity specification are translated into member variables and methods of the generated activity class. This section presents a detailed description of the generated activity code.

```
Import statements
Class process_name_activity_name
{
        declarations of variables for activity input parameters;
        declarations of variables for activity output parameters;
        declarations of vector for outParameterValues;
        if (service requests are specified in the activity)
                declarations of service-related variables;
        }
        if event is specified in the activity{
                declarations of event-related variables;
        }
        activity_name(){
                try{  if (task item is service) {       do service-related work     }
                      create a vector for outParameterValues
                }
                catch (Exception e) { call the Exception Handler }
        }
        if (task items include service and the activity's performer is of type ANY)
        ActivityResult activate(Vector vActParam, String modelName, String businessType,
                                        Hashtable serviceConstraints) {
                        Try{    initialize activity's input parameters with values in vActParam
                                If (task items are services){
                                        Find the service provider and invoke services to
                                        obtain response
                                }
                                If (task items are events){
                                        Post sync events to ETR Server
                                        Or Post async events to Event Server
                                }
                                if(task items are inline codes){
                                        fetch inline codes from MetaData Manager
                                        and concat them to the code
                                }
                                Return ActivityResult with service provider's URL and
                                outParameterValues
                        } catch (Exception e){ call the Exception Handler }
                        return null;
        }
        if (task items include service and the activity's performer is of type Other Than ANY)
        ActivityResult activate(Vector vActParam, String modelName, String serviceProviderURL){
                Try{ initialize activity's input parameters with values in vActParam
                     If (task items are services){
                                Invoke services with the given service provider to get response
                     }
                     … the rest is the same as the previous one with Performer of type ANY
        }
}
```

Figure 6.2 General Structure of Generated Activity Class

```
public class OrderProcessing_Shipping
{       /* declare activity input, output, and activity vars */
        public UserInfo user_info;
        public ProdDetailDesc prod_detail_desc;
        public int quantity;
        public String delivery_date;
        public boolean shipping_status;

        /* declare service-related variables */
        public String XMLInput_ShipOrder;
        public String XMLOutput_ShipOrder;
        public ServiceClient sHandler;
        public Broker.BrokerProxyIntf brokerProxy = null;
        public RequestGen requestGenerator;

        /* declare the Vector containing the return value of the activity */
        Vector outParameterValues = null;

        /* constructor */
        public OrderProcessing_Shipping()
        {
                try {if (System.getSecurityManager() == null ) {
                                System.setSecurityManager( new RMISecurityManager() );
                        }
                        String proxyName = "//" + InetAddress.getLocalHost().getHostAddress() +
"/BrokerProxy";
                        sHandler = new ServiceClient();
                        brokerProxy = (Broker.BrokerProxyIntf)Naming.lookup(proxyName);
                        requestGenerator = new RequestGen();
                        outParameterValues = new Vector();
                } catch ( Exception ex ) {
                        System.out.println( "Activity Constructor: " + ex );
                }
}
```

Figure 6.3 Declarations and Constructor in Code Generated for the Activity *InitiateShipping*

6.2.1 Member Variables

The activity's input parameters, output parameters, activity variables, and other variables necessary to handle dynamic service binding and event posting are translated into member variables of the activity class.

The input and output parameters specified in the activity are stored in vectors of parameter objects which have both a parameter type and a parameter name. The input or output parameter can be of any primitive type or user-defined data type. In case of user-defined data types, import statements for the packages containing the data classes are generated. A vector is declared to contain values of the output parameters of the activity. For example, in Figure 6.3, the input parameters of the activity *InitiateShipping* are declared as UserInfo user_info, ProdDetailDesc prod_detail_desc, int quantity, and String delivery_date. Data classes for UserInfo and ProdDetailDesc are imported prior to the declaration. The output parameter is declared as boolean shipping_status. Vector outParameterValues is declared to contain the values for the output parameters of the activity *InitiateShipping*.

Besides these common declarations, some other declarations are made according to the type of task items specified in the activity.

- If a task item specified in the activity is an e-service request, some service-related variables have to be declared for later use. Two strings are declared, one for the XML input message to contain the e-service request's input data, the other for the XML output message to contain the result of the operation invoked in the e-service. An e-service request generation class variable is declared in order to generate XML input messages and extract results from the XML output messages. A service handler class variable is declared for the Workflow Engine to handle e-services. If the activity's performer is of type ANY, a Broker Proxy class variable is declared for the Workflow Engine to contact the Broker Proxy for dynamic binding of the e-service request with an appropriate e-service and service provider. For example, activity *InitiateShipping* has one e-service request as its only task. Since its performer is TransportationAgency ANY, dynamic service binding is needed. In Figure 6.3, String XMLInput_ShipOrder and String XMLOutput_ ShipOrder are declared for the e-service request's input and output respectively. Variables such as ServiceClient sHandler, Broker.BrokerProxyIntf brokerProxy, and RequestGen requestGenerator are declared for purposes described above.

- If the task item is inline code, no variables need to be declared. The inline code obtained from the activity specification is directly concatenated to the generated activity class.

```
public ActivityResult activate(Vector vActParam, String modelName, String businessType, Hashtable
serviceConstraints)
        {       try{    /* initialization */
                        Object serviceRetValue;
                        ActivityResult actResult = new ActivityResult();
                        user_info=(UserInfo)vActParam.elementAt(0);
                        prod_detail_desc=(ProdDetailDesc)vActParam.elementAt(1);
                        quantity=((Integer)vActParam.elementAt(2)).intValue();
                        delivery_date=(String)vActParam.elementAt(3);
                        /* generate e-service request constraint */
                        Constraint constr =
(Constraint)serviceConstraints.get("ShipOrder_InitiateShipping");
                        if(constr == null)
                                constr = new Constraint();
                        constr.setProduct("ShipOrder_InitiateShipping");
                        Tools.insertConstraint(user_info,constr);
                        Tools.insertConstraint(prod_detail_desc,constr);
                        Tools.insertConstraint("quantity",quantity, constr);
                        Tools.insertConstraint("delivery_date",delivery_date, constr);
                        /* Contact broker proxy */
                        String serviceProviderURL = brokerProxy.selectServiceProvider(businessType,
serviceConstraints);
                        /* for service request: generate XML request msg */
                        requestGenerator.createXMLMessage("InitiateShipping");
                        requestGenerator.addParam("prod_detail_desc",prod_detail_desc);
                        requestGenerator.addParam("delivery_date",new String(delivery_date));
                        requestGenerator.addParam("quantity",new Integer(quantity));
                        requestGenerator.addParam("user_info",user_info);
                        XMLInput_ShipOrder =requestGenerator.getXMLMessageString();
                        sHandler.setServiceURL(serviceProviderURL);
                        XMLOutput_ShipOrder =sHandler.call(XMLInput_ShipOrder);
                        /* get result from Xml message */
                        Hashtable xmlResult = requestGenerator.getResult(XMLOutput_ShipOrder,
null);
                        Enumeration enum = xmlResult.elements();
                        serviceRetValue = enum.nextElement();
                        shipping_status= ((Boolean)Tools.getPrimitiveValue("Boolean",
(String)serviceRetValue)).booleanValue();
                        actResult.addServiceProviderURL(serviceProviderURL);
                        outParameterValues.addElement(new Boolean(shipping_status));
                        actResult.addOutputValues(outParameterValues);
                        return actResult;
                }catch (Exception ex) { System.out.println("in Activity: " + "Shipping");}return null;}
}
```

Figure 6.4 Activate Method in Code Generated for the Activity *InitiateShipping*

6.2.2 The Activity Constructor

A constructor method is generated for each activity class. This method is used to contact different servers at run-time and to assign values to class variables declared

previously. If there is an e-service request specified and the performer of the activity is of type ANY, the Broker Proxy is contacted. The service handler and XML request generator are instantiated. If there is an event specified in the activity, the Event Server and the Event-Trigger-Rule (ETR) Server are contacted. A vector is created to contain values of output parameters of the activity. In Figure 6.3, in the constructor for activity *InitiateShipping*, a reference to the Broker Proxy will be obtained at run-time by executing "brokerProxy = (Broker.BrokerProxyIntf)Naming.lookup(proxyName)". The service handler is instantiated by executing "sHandler = new ServiceClient()", and the request generator is instantiated with "requestGenerator = new RequestGen()". The vector to contain output parameter values of the activity *InitiateShipping* is created with "outParameterValues = new Vector()".

6.2.3 The Activate Method

In each generated activity class, there is one activate() method, which performs the tasks specified in the activity body. If an e-service request is specified in the activity, according to the type of the activity's performer, two different kinds of activate methods are generated.

If the activity's performer is of type ANY, then dynamic service binding is needed and the Broker Server needs to be contacted for the binding. The activate method generated has four arguments:

- A vector containing values for the activity's input parameters
- A string containing the process model's name
- A string containing the business type of the performer
- A hash table containing e-service request constraints specified in the activity

Based on the business type and e-service request constraints, the Broker Server dynamically binds the e-service requests with appropriate e-services and service

providers. After the binding, the Broker Server returns the selected service provider's URL to the activity. XML messages containing the input attributes of the e-service request are generated. These messages are sent to the service provider for processing. After processing, the service provider returns the outputs in the form of XML messages. Values for the output parameters are extracted from the XML output messages. The result, containing the selected service provider's URL and output parameters values, is returned at the end of the activate method.

For example, in the activate method generated for the activity *InitiateShipping* (Figure 6.4), we can see that the activate method executes as follows:

1. The input parameters for the activity is initialized with values passed by the Workflow Engine.

2. The e-service constraint is obtained from the e-service constraints hash table.

3. The Broker Proxy is invoked with the performer's business type and the e-service constraints. The Broker Proxy will communicate with the Broker Server for an appropriate service provider. The URL for the selected service provider will be returned to the activate method.

4. An XML message with the input attributes for the e-service request is generated. With this message, the service handler invokes the e-service requested for the XML message with the output.

5. The value extracted from the XML output message is assigned to the activity's output parameter. It is also added to the vector for output parameter values. The service provider's URL and the output parameter values are returned to the Workflow Engine.

If the activity's performer is NOT of type ANY, it will be of type CONSTANT, SAME_AS or VARIABLE. In this case, the generated activate method has three arguments:

- A vector containing values for the activity's input parameters
- A string of the process model's name
- A string with the service provider's URL

E-service requests specified in such an activity do not need dynamic service binding and the Broker Server does not need to get involved. The service provider can be obtained according to the activity's performer specification before the activate method is invoked. The service provider can be directly contacted with XML messages containing the e-service request's input data for processing. The result, with values for the activity's output parameters, is returned at the end of the activate method.

In both kinds of generated activate methods, *try-catch* statements are used to handle possible exceptions. The try block takes normal actions, while the catch block handles the exception and prints out information about the exception.

6.3 Implementation of Activity Code Generator

The class diagram of the Activity Code Generation is given in Figure 6.5. Descriptions about the key classes and methods involved in the generation of the activity code are presented.

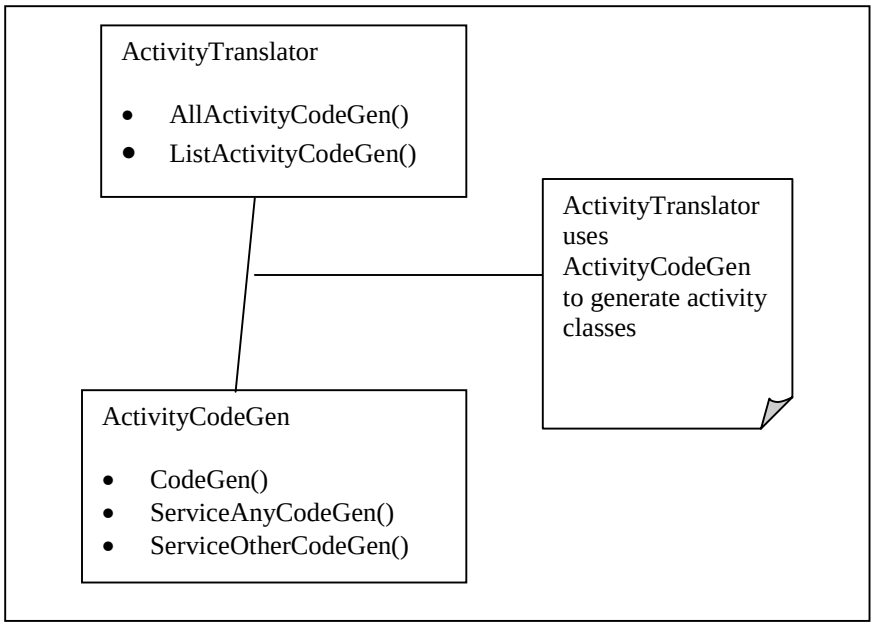


Figure 6.5 Class Diagram for Activity Code Generation

6.3.1 Activity Translator

The ActivityTranslator class is the interface class which provides the API for an external component to use the Activity Code Generator. When the Activity Translator is instantiated, the MetaData Manager is contacted and a reference to it is passed to the constructor. There are two public methods in the Activity Translator.

- AllActivityCodeGen(): This method is used to generate code for all the activities specified in a particular process model. The process model name and the directory path are passed as arguments to the method. With the process model name, a list of all the activities defined in it can be obtained from the MetaData Manager. For each activity, an activity class is generated and stored in the directory. The method CodeGen() in the ActivityCodeGen class is called to generate the actual code.

- ListActivityCodeGen(): This method is used to generate code for a list of activities in a process model. The process model name, the directory, and the list of the activity names packed into a vector are passed as arguments to this method. An activity code is generated for each activity in the vector. Again, the method CodeGen() in the ActivityCodeGen class is called to generate the actual code.

6.3.2 ActivityCodeGen

The actual activity code generation is done in this class. The process model name and a reference to the MetaData Manager are passed to the class when it is instantiated. There are mainly three public methods in the class:

- CodeGen(): Activity Translator calls this method to generate the actual code for each activity. The activity name and the directory to store the generated code are passed to this method as arguments. The activity specification is fetched from the MetaData Manager and the corresponding Java class is generated. For each input parameter, output parameter, activity variable, service-related variable, and output parameter values, the appropriate import statements, declarations and initializations are generated. According to the activity specification, the appropriate activity constructor and the activate method are also generated, using the CodeGen method.

- ServiceAnyCodeGen(): This method is called by CodeGen to generate code concerning e-service requests when the activity's performer is of type ANY. In this method, code to contact the Broker Server to do the dynamic binding of e-service requests with suitable e-services and service providers is generated.

- ServiceOtherCodeGen(): This method is called by CodeGen to generate code handling e-service requests when the activity's performer is of type other than ANY. Code to fulfill the e-service requests with a designated service provider is generated.

CHAPTER 7
SUMMARY AND CONCLUSIONS

In this thesis, a “code generation” approach to support a dynamic workflow management system has been presented. Based on the specification of a workflow model defined using the Process Definition Tool and stored in the MetaData Manager, run-time workflow structures and activity code are generated.

The run-time workflow structures generation approach presented in this thesis supports the adaptive property of the Dynamic Workflow Model. The generated workflow structures, consisting of entity, control, and dataflow structures, are used by the Workflow Engine to schedule and execute a workflow instance. Dynamic changes to the control and data flows of an executing workflow instance can be made at run-time by changing these run-time structures. For example, rules may be used to alter the execution flow of the running process instance. Such rules may be triggered by some workflow event to skip one or more activities under certain conditions. The alteration is accomplished by invoking methods via the API of the Workflow Engine to modify the run-time control flow structures and data flow structures of the running process instance.

The generated activity code is in the form of a Java source file (.java files) and is stored in the Run-time Repository in a specified directory. It is compiled by invoking the Javac compiler and the activity classes (.class files) are stored in the same directory. The Workflow Engine, when scheduling the activities for a process instance, simply loads the Java activity classes from the Run-time Repository to and executes the code directly to

perform specified tasks (e.g., contacts the Broker Server for the binding of e-service requests to service provider and the performance of the services).

This code generation approach results in a lightweight and efficient Workflow Engine. It is lightweight because the Workflow Engine does not need the logic to interpret the activity specification in order to determine how to bind the e-service requests (by contacting the Broker Server) and calling the bound service provider to perform the e-services. Taking advantage of the performance of compiled classes, the Workflow Engine is efficient. Finally, changes to an activity (e.g., e-service requests, performer, performer constraints, etc.) can be made using the Process Definition Tool. The code for the modified activity is re-generated and re-loaded using Java's class reloading capability. In this manner, any change made to an activity specification will be reflected in the execution of the activity as long as the re-generated code is placed in the Run-Time Repository *before* the execution of the activity. Thus, flexibility is maintained without sacrificing performance.

LIST OF REFERENCES

[ALO99] Alonso, G., Fiedler, U., Hagen, C., Lazcano, A., Schuldt, H., Weiler, N. "WISE – Business to Business E-Commerce," Proceedings of 9th International Workshop on Research Issues on Data Engineering: Information Technology for Virtual Enterprises, Sydney, Australia, 23 - 24 March 1999.

[CAS96] Casati, F., Grefen, P., Pernici, B., Pozzi, G., Sanchez, G. "WIDE Workflow Model and Architecture," Technical report, University of Twente, Netherland, 1996.

[CRO00] CrossFlow Consortium, "Flexible Change Control," CrossFlow deliverable: D8.a, http://www.crossflow.org, February 07, 2000.

[ELL95] Ellis, C., Keddara, K., Rozenberg, G., "Dynamic Change Within Workflow Systems," In Proc. Conference on Organizational Computing Systems (COOCS), Milpitas, CA 1995, 10-22.

[GEO95] Geogakopoulos, D., Hornick, M, Sheth, A. "An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure," Distributed and Parallel Database, 3, 119-153 1995.

[GEP98] Geppert, A., Tombros, D, "Event-based Distributed Workflow Execution with EVE," Technical Report, No. 96.05, University of Zurich, Switzerland, March 1998.

[GRE99] Grefen, P., Hoffner, Y., "CrossFlow – Cross-Organizational Workflow Support for Virtual Organizations," Proceedings of 9th International Workshop on Research Issues on Data Engineering: Information Technology for Virtual Enterprises, Sydney, Australia, 23-24 March, 1999.

[LAZ00] Lazcano, A., Alonso, G., Schuldt, H., Schuler, C., "The WISE Approach to Electronic Commerce," International Journal of Computer Systems Science & Engineering, special issue on Flexible Workflow Technology Driving the Networked Economy, 15(5), September 2000.

[LEE00] Lee, M. "Event and Rule Services for Achieving a Web-based Knowledge Network," Ph.D. Dissertation, Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, 2000.

[LEY94] Leymann, F. and Roller, D, “Business process management with flowmark,” Proceedings of the 39 the IEEE Computer Society International Conference, p:231-233, California, February 1994.

[LUD99] H. Ludwig, Y. Hoffner; “Contract-based Cross-Organisational Workflows: The CrossFlow Project,” Proceedings of the WACC Workshop on Cross-Organisational Workflow Management and Co-Ordination, San Francisco, CA, February 22, 1999.

[MEN01] Meng, Jie, “Achieving Dynamic Inter-organizational Workflow Management by Integrating Business Processes, Events and Rules,” Ph.D. Research Proposal, University of Florida, Gainesville, FL, 2001.

[MQW00] IBM, “MQSeries Workflow,” http://www.software.ibm.com/ts/mqseries/workflow/, August 10, 2000.

[MUL99] Muller, R., Rahm, E., “Rule-based Dynamic Modification of Workflows in a Medical Domain,” Proceedings of BTW99, Spinger, Berlin 1999: 429-448.

[RAN97] Ranno, F., Shrivastava, S.K., Wheater, S.M., "A System for Specifing and Coordinating the Execution of Reliable Distributed Applications," International Working Conference on Distributed Applications and Interoperable Systems (DAIS'97), Cottbus, Germany, September 30 - October 2, 1997.

[REI98] Reichert, M., Dadam, P., “Adept_flex-Supporting Dynamic Changes of Workflows Without Losing Control,” Journal of Intelligent Information Systems, Special issue on Workflow and Process Management, 10(2), March 1998.

[SHE96] Sheth, A., Georgakopoulos, D., Joosten, S., Rusinkiewicz, M., Scacchi, W., Wileden, J., Wolf, A., “Report from the NSF Workshop on Workflow and Process Automation in Information System,” Sigmod Record, 25(4), December 1996.

[SHE01] Shenoy, A., “A Persistent Object Manager For Java Applications,” M.S. Thesis, Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, 2001.

[SHR98] Shrivastava, S.K. and S.M. Wheater, “Architectural Support for Dynamic Reconfiguration of Large Scale Distributed Application,” Proceedings of the 4th International Conference on Configurable Distributed Systems (CDS), Annapolis, May 4-6, 1998.

[SU00] Su, S.Y. W., Lam, H., “IKnet: Scalable Infrastructure for Achieving Internet-based Knowledge Network,” invited paper, Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet, l’Aquila, Rome, Italy, July 31-Aug.6, 2000.

[SU01] Su, S.Y. W., Lam, H., Lee, M., Bai, S., Shen, M., "An Information Infrastructure and E-services for Supporting Internet-based Scalable E-business Enterprises," paper submitted to 5th International Enterprise Distributed Object Computing Conference, Seattle, USA, September 4-7, 2001.

[VIT99] VITIRA Corporation, “Vitria Business Ware,” http://www.vitria.com/products/businessware/, August 18, 2000.

[WFM99a] WfMC, “Interface1: Process Definition Interchange V 1.1 Final (WfMC-TC-1016-P)," http://www.wfmc.org, Sep. 18, 2001.

[WFM99b] WfMC, “Terminology and Glossary,” http://www.wfmc.org, August 18, 2000.

[WHE98] S.M. Wheater, S.K. Shrivastava and F. Ranno, “A CORBA Compliant Transactional Workflow System for Internet Applications,” Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware’98), The Lake District, England, September 15-18, 1998.

[WIN87] T.Winograd and R. Flores, “Understanding Computers and Cognition,” Addison-Wesley, 1987.

BIOGRAPHICAL SKETCH

Xiaoli Liu was born March 27, 1971, in Jinxiang, Shandong Province, China. She received a Bachelor of Arts in English from Qingdao University, Qingdao, in August, 1993. In 1996, she received a Master of Arts in English literature from Peking University, Beijing, China.

Liu joined the University of Florida in August 1997 to pursue a master's degree in the College of Journalism and Communications. In August 1999, she changed direction and joined the Department of Computer and Information Science and Engineering to pursue a Master of Science degree in computer science. Liu has worked as a teaching assistant at the Department of Computer and Information Science and Engineering while pursuing her degree. Her research interests include dynamic workflow management and e-business.