University of Alberta

Library Release Form

Name of Author: Huaxin Zhang

Title of Thesis: Rule-based Application Integration using XML

Degree: Master of Science

Year this Degree Granted: 2001

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

_____

Huaxin Zhang
Department of Computing Science
University of Alberta
Edmonton, Alberta
Canada T6G 2E8.

Date: _____

University of Alberta

Rule-based Application Integration using XML

by

Huaxin Zhang

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements of the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta

Fall 2001

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Rule-based Application Integration using XML** submitted by **Huaxin Zhang** in partial fulfillment of the requirements for the degree of **Master of Science**.

_____

Supervisor: Eleni Stroulia

_____

Davood Rafiei

_____

Witold Pedrycz (Electrical Engineering)

Date: _____

To my parents, Canchun, and all those beloved.

ABSTRACT

Application interoperability is a challenging problem because different applications assume different data types and control-of-processing models. Even when related applications are aggregated in so-called portals, it is up to the user of the portal to access the different applications, collect information, and then combine this information to access yet other applications.

Babel is an information-mediation architecture for integrating multiple heterogeneous applications, by wrapping them and by specifying the logic of their interoperation in XML. The mediation system consists of a design time environment and a runtime engine as an integrated mediation system. The design time environment presents an easy to use GUI interface that automates the generation of mediation rules following an event-condition-action paradigm. The runtime engine provides a virtual machine for enacting the mediation rules as well as providing a workflow-based processing model based on the mediation rules.

Acknowledgements

I would like to express my deep gratitude towards my supervisor, Dr. Eleni Stroulia, for her supervision, encouragement and motivation through the course of this work. Without her vision, guidance, and insightful comments, this work would not have been possible.

I am also grateful to members of CelLEST project Roland Penner, Paul Iglinski, Rohit Kapoor, Maurine Hatch, Mohammad El-Ramly, Vijayan Menon, for their patient help, dedicated cooperation and timely suggestions throughout the tenure of this project.

Finally, I would like to thank Canchun, my parents for their selfless support and continuous encouragement.

Huaxin Zhang
September, 2001.
Edmonton, Canada.

Table of Contents

CHAPTER 1 INTRODUCTION AND MOTIVATION 1

1.1 THESIS OBJECTIVES 3

1.2 THESIS OVERVIEW 5

CHAPTER 2 BACKGROUND AND RELATED WORK 7

2.1 MEDIATOR ARCHITECTURES 7

2.2 COMMUNICATION MECHANISMS 9

2.3 MEDIATION LANGUAGES 12

2.4 MEDIATION LOGIC 13

CHAPTER 3 THE BABEL APPROACH TO APPLICATION INTEGRATION 15

3.1 DATA MODELING 16

3.2 COORDINATION LOGIC 19

3.2.1 Even-Condition-Action Rules 19

3.2.2 Workflow Sessions 22

CHAPTER 4 THE BABEL RUN-TIME ENVIRONMENT 25

4.1 MEDIATION RUN-TIME PROCEDURE 25

4.2 INFORMATION REPOSITORY 26

4.2.1 The Need for a Database 27

4.2.2 Relational vs. Object-Oriented Database Management 29

4.2.3 Relational Database Solutions 31

4.2.4 Benchmarking 37

4.3 PARALLEL PROCESSING AND WORKLOAD BALANCE 39

CHAPTER 5 BABEL WRAPPERS AND EXTENSIONS 1

5.1 THE WRAPPER INTERFACE 1

5.2 The Wrapper Architecture 2

5.3 The Babel Extension Service 4

CHAPTER 6 IMPLEMENTATION AND PERFORMANCE 6

6.1 Implementation 6

6.2 Performance Evaluation 14

CHAPTER 7 CONCLUSIONS 19

7.1 Feature of Babel Mediator 19

7.2 Contribution of Babel 20

7.2.1 Contribution of Babel in the CelLEST project 21

7.2.2 Contribution of Babel as general mediator approach 22

7.3 Concluding thoughts and future work 24

REFERENCES 27

APPENDIX 33

Data Modeling for *Task* (DTD source code) 33

Guidelines for Email Book Query Demonstration 34

Rule 1 source code 34

Rule 2 source code 37

Session Example 39

Email Query Task 44

Hollis Book Search Request Task 45

Hollis Book Search Result Task 46

Email Answer Task 48

Babel Runtime /peripheral initialization script 50

Index of Figures

Figure 1: Architecture of the Babel mediator. 15

Figure 2: Message queuing for task events. 27

Figure 3: Sample tree structured data for predefined DTD. 32

Figure 4: Working Thread Pool vs. Jobs Pool paradigm. 41

Figure 5: Barrier 43

Figure 6: The Wrapper Architecture. 3

Figure 7: The Rule Wizard and Session Builder as plug-ins. 8

Figure 8: Rule Description. 8

Figure 9: Event/History Selection. 8

Figure 10: Join Operation. 9

Figure 11: Constructing a Condition. 9

Figure 12: Action Template Selection. 9

Figure 13: Defining an Action. 9

Figure 14: Rule Previewing. 10

Figure 15: Saving the Rule. 10

Figure 16: Importing the rules. 11

Figure 17: Transition building. 11

Figure 18: The Babel mediator and the wrappers initialized. 12

Figure 19: The mediation procedure.. 13

Figure 20: Average processing time (t_s) vs. time between message requests. 15

Figure 21: Average processing time (t_s) vs. Number of rules $r_{n.}$ 17

Figure 22: Average recovery time vs. Number of simultaneous messages. 18

Chapter 1 Introduction and Motivation

With the fast thriving modern IT industry today, information processing and exchanging plays a more critical role than ever before. While users have easy access to huge amounts of data, decision-making based on the data is difficult. As the scope of the information systems grows, data sources have problems of increasing heterogeneity [WIE00]. Information comes from legacy systems, modern software systems, various databases, or from web-based applications. These data sources use different protocols to communicate with their users, and the data they provide are syntactically and semantically different. The autonomy of the available applications that can provide data and the heterogeneity of the available data present a great challenge for information integration, coordination, and inter-operation.

A lot of research has focused on the integration of heterogeneous information sources, helping to provide users a uniform, transparent, and efficient environment for querying, browsing, and maintaining data [HAAS99] [WIE97][WIE98]. Meanwhile, the problems of application integration have become much more challenging, in that not only users but also software systems need integrated access to information, which is stored in databases, object repositories, knowledge bases, file systems, and document-retrieval systems, as well as legacy systems and web applications. This is due to several reasons. First, information available within a particular application can be used in novel ways by new consumers, whether they may be new application systems or simply new users of the existing application [AEAIILA]. Second, large organizations, that own multiple

information systems, find themselves following outdated business procedures just because their applications that support these processes do not interact. To further automate their processes so that they meet the evolved business requirements, their application systems need to interact in new ways and exchange information with each other in the context of new workflow [SUND99]. Finally, organizations need to make information and services available beyond their boundaries, to customers and partners for example, in order to provide more value-added services and to maintain or increase their competitiveness. Therefore, information exchanging and processing between heterogeneous components within or across systems becomes increasingly important.

However, different applications are developed in different programming languages and assuming different data formats (syntactic), producing different data content, following different business rules (semantic), and operating on different hardware and software systems. Application information integration becomes difficult because of such heterogeneity of information services provided by different applications. Moreover, application integration is different from information service integration in that applications have different application programming interfaces (API) and each API receives data in various protocols and formats. Information flow is bi-directional and interfaces as input of applications require control and information data in various protocols, calling conventions, data formats, and semantic restrictions. Changing the logic of existing applications internally is usually not an option, since more than 60% of existing code are legacy applications [CARR95], and changing the source code or using white box migration for these systems is too risk ridden and costly. A more appropriate solution that we adopt is to apply “black-box migration methods”, that is to wrap the

applications in order to expose their desirable behaviors through a canonical API, and to adopt an information-mediation approach to integrate the data provided to (and by) the wrapped applications.

1.1 Thesis Objectives

To address the heterogeneity and distributed nature of applications integration, we need a mediation architecture that satisfies the following requirements:

Transparency: Heterogeneity between components should be transparent to the communication between the components with mediator. The integrated architecture provides an environment for communicating in a canonical way regardless of the data schema or protocol that the underlying components are using. Coordination logic can therefore operate on top of the canonical communication mechanism and need not interfere with the modules that parse, validate, and understand data to and from the heterogeneous components. For example, users need not take the heterogeneous nature of the components into consideration when writing mediation rules for integrating these components.

Scalability: The mediator engine should be capable of interacting with a large number of applications and the average service time (T_s) for the mediator should scale well with the number of involving components and business rules. The engine should be capable of recovering from sudden information flooding within reasonable time, and should have efficient service time under heavy communication load.

Extendibility: Adding components to an existing integration should have no side effect on the already integrated components. The complexity of adding a new component

should have no relationship with the current size of the existing system. The work of adding new components should be reasonably easy to accomplish. Rules for integrating can be defined and changed incrementally, and rule enactment should be fault-soft in that an error occurring during one rule enactment should not influence the enactment of other rules.

Semantic Richness: Having the capability of bridging heterogeneity is not the whole story for information mediation. Mediators are intended to provide extra value beyond the value provided by the original information sources. [WIE95a] [WIE97] therefore, information mediation for system integration should have the capability of exchanging information between applications with certain degree of automation, and information processing in the process of exchanging and interoperation.

Workflow-based Management: We are currently witnessing the desire to automate business processes across the enterprises and across the existing boundaries in modern enterprise application integration [AEIILA]. Workflow management is the automated coordination, control and communication for execution of multiple tasks [SHE96]. It is ideal to incorporate business rules into coordinated process descriptions that can be executed automatically [CHI98].

1.2 Thesis Overview

This thesis presents the design and implementation of the Babel mediation architecture, an architecture for information mediation and application integration. This approach meets all the requirements listed above and fits well into a realistic application integration domain. The architecture provides an integrated design time support for data modeling, interface auto generating, coordination logic auto generating, and automatic registration for both rules and application interfaces. The runtime environment of the architecture provides a mediator engine that runs rule enactment as low-level coordination at the basis and workflow-based coordination at a higher level.

This thesis argues that the mediation architecture can effectively address the problems observed with other mediation solutions, such as [CHAWA94][MIX][GM01]. These problems are incremental data integration [GM01], intertwined information access and integration [CHAWA94], and most important: non-reusable mediation rules. First, the data schema does not need to change when new applications need to be integrated after the initial system is built, and applications can be incrementally integrated into the mediation system by simply registering application specific rules within the design time. Second, the design time and the runtime activities for the mediation system are loosely coupled. The runtime environment is ignorant about how many applications are being integrated and what data schema each application is using. All these parts of the work are done within the design time environment and can evolve as the runtime is executing. Third, the mediation business rules are highly re-usable knowledge components that can

be plugged into the mediator engine, where they form the foundation of the workflow-based mediation.

As a concrete practical example, this thesis introduces a case study of mediation-based integration on several legacy systems as aggressive software maintenance. A Library query system and PINE email system are chosen as objective for application integration. They are both legacy systems with unfriendly user interfaces (both support character screens only) and they both mandate over-complicated yet unnecessary procedures for a simple task (must login even for querying about the keyword of a book). The mediation goal for this example is to simplify the access to both systems while implementing new business logic over them so that a user can query about books with a particular keyword in their title by sending an email to a help desk. The mediator can find if that user has queried about the same book keyword before. If the user has queried before, the mediator may respond to the user with an email message containing the query result from a book dealer. Otherwise the mediator may query the book against the digital library and send the query result back to the user in email.

Chapter 6 evaluates the Babel mediator. The design time procedure and runtime simulation indicates that the Babel mediator provides the architecture to automatically build the rules and the workflow logic and to effectively enact them at runtime. In concert with a parallel project, Mathaino [K2001], which enables users to construct wrappers for legacy applications by demonstrating the task that the wrapped application should accomplish, the whole mediation work can be easily achieved by common users.

Chapter 2 Background and Related work

Research efforts have begun a long time ago to improve the integration of heterogeneous databases. Early on, the research focused on providing integrated data schemas or integrated views for heterogeneous databases [BAT86], developing data interoperation and data integration techniques [BER89][BOT86], or creating new query languages to query distributed heterogeneous databases [LEVY96]. However, these efforts are mainly to support integrated queries for heterogeneous databases, which turn to be passive and lack of interoperability.

The information-mediation problem came to the forefront as new information systems for interoperation and coordination for distributed heterogeneous databases. Layered architectures are proposed to meet the requirement for homogenizing heterogeneous systems [KAR95][WIE92][NAV89], data modeling approaches for coordination are developed [MOLINA494], new languages are invented for mediator construction (Telos [MYL90] in COOPWARE, MSL/LOREL [GOLDMAN99] in TSIMMIS), software engineering methodologies for mediation systems [WIE95b] are adopted. Comparison between the Babel approaches with other existing mediators in the above mentioned aspects are discussed respectively in the following sections.

2.1 Mediator Architectures

Traditional methodology embraced the client-server paradigm for exchanging and manipulating heterogeneous data. The client-server architecture is intrinsically a two-

tiered architecture [UMA99] and is less scalable and reusable than a 3-tiered architecture. Changes made in one client's needs may require changes at the server, and in turn, impact an unpredictable number of other clients. In application-system integration, applications can play both the server and the client roles. Data-flow can be bi-directional rather than unidirectional. Conversation between each possible pair of application forms an arbitrarily connected network and squared growth to the number of applications greatly complicates the architecture. More than 80 percent of most IS department budgets is spent on doing application integration based on an unsuitable architecture [WIE98], and this approach greatly increases maintenance cost [WIE95a]. Another disadvantage of client server architecture is the lack of support for asynchronous operations and synchronization mechanism when interoperation involves multiple applications must abide to some transactional operation requirements.

The Babel mediator employs the well-known mediator architecture [WIE92], in which another layer—the mediator layer is inserted between the client and server. This architecture is widely adopted among many information mediation projects (TSIMMIS, SAP, MIX/CQ and COOPWARE [GM01]). Just like these information mediators are based on the integrated view provided by underlying wrappers [CAREY94], the Babel mediator is situated on top of wrappers being built in the CelLEST project [STROULIA00]. The wrappers for Babel mediators are capable of exposing object oriented task-based APIs for interaction with the underlying applications (see Chapter 5), and the Babel mediator could manipulate the applications through the API provided by the wrappers.

One of the advantages of this architecture is that it separates mediation logic from data-communication mechanism, making the whole system more extendable and scalable. Business rules are pluggable into the mediator, instead of being hard coded into the applications or wrappers.

The major difference from other implementations of the same architecture is the separation of the design time and the runtime environments. The Babel design time environment supports defining the application interfaces (wrappers) involved in the integrated application, the coordination logic among them and the workflow process definition. The runtime environment provides a virtual machine for carrying out the defined coordination logic based on that data modeling and a workflow engine based on that coordination logic. This separation makes the mediator runtime isolated from application domain and highly re-usable. Another advantage is that the design time is able to evolve (i.e. change data model and coordination rules) without impacting the runtime execution of the integrated application.

Both the runtime and design time environments are internally layered sub-architectures. The design time environment is layered into fine-grained controlling unit and global behavioral workflow definition, allowing for continuous information coordination based on more complex logic while enhancing the re-usability of business rules as software components. The runtime environment consists of three layers: the information repository, the rule enactment layer, and the workflow (based on rule) enactment layer (see more in Chapter 3).

2.2 Communication Mechanisms

Research in distributed heterogeneous or federated database has identified representation and semantic agreement as necessary for semantic interoperability [HEI95]. Different applications model their data (internal and external) in various ways. When it comes to integration, the mediator should be able to break down the representation differences and bridge the semantic content. Traditional information mediation works directly on raw data generated from application. Typically, the block of data being sent across is not self-describing. The key to what a block of data means—where fields begin and end and how information is formatted—is incorporated into code that is responsible for parsing the block of data and validating the information contained therein based on some autonomous predefined data format agreement. Parsing and validation involve a lot of error-prone coding and the code is completely ad hoc and largely non-reusable [SUND99]. Some research work resorts to meta-data in defining the data flow [LAN98], but there is still ad hoc parsing and validating.

In order to facilitate data exchange, Object Oriented integration frameworks such as CORBA and DCOM have been proposed. However, their interoperability occurs at low-level data representation and behavioral agreements and not at the semantic level. This is insufficient for building a ubiquitous integration framework solution, because the underlying semantic of the data content is not explicit, and the framework cannot make safe assumptions of the data content it gets. Other methods for exchanging information between systems like RMI and RPC have much in common with the CORBA and DCOM methods [SUND99]. Little can be done to depict the semantic of data being sent except revolving around passing information that has been reduced to a block of data.

Therefore, the common data model should not only be more flexible than the commonly used for database management systems, it should also be able to describe them in terms of semantic content in a consistent way. XML, the eXtensible Markup Language [XML] provides a possible answer to these requirements.

- XML can be used as a meta-language for data content self-description. By using its markup tags, data content can be carried with semantics. XML is a meta-language to define both the syntax and the semantics of the data.
- XML data are semi-structured so that it gracefully handles missing information or related information of widely differing structures [MOLINA494].
- Parsing and re-generating data in XML format is standardized and has wide support through out the world. Modules for data handling are highly reusable and development for those modules is highly efficient and standardized.
- XML has its own data validating mechanism. By agreeing on Data Type Definition (DTD) [DTD], validating can be accomplished easily without ad hoc programming, yet that DTD does not rigidify flexibility. DTD and XML Schema assure syntactical uniform-ness across data agencies.
- XML supports strong type definition through XML-Schema [XSCHEMA]. This provides object-oriented data modeling for its data (inheritance, sub-typing, etc).

All these attributes make XML an ideal candidate for data modeling in the Babel mediator architecture. Application specific data from heterogeneous applications are interpreted by wrappers and transformed into XML data validated against a canonical DTD characterizing the semantics of the overall integrated application. By using XML in

conjunction with globally defined DTD, data can be sent with various semantic contents in a consistent data format.

2.3 Mediation Languages

Aside from the data modeling, there must be a mediation language for querying, manipulating, and transforming data. The choice of mediation language naturally follows the mediation data modeling, such as MSL/LOREL used in TSIMMIS [MOLINA494], and Telos used in COOPWARE [MYL96]. Rather than building a proprietary mediator language, Babel mediator chooses eXtensible Stylesheet Language Transformation (XSLT) as the coordination-level language. XSLT is W3C standard for transforming XML documents. Its element locating subset – XPATH -- is another W3C standard for inter-intra XML document queries. It is different from SQL language in that it is designed for XML language. It can address arbitrary locations inside tree-structured XML data and perform certain functional transformation on those query results. It is worth mentioning here that XSLT, like LISP, is a functional programming language [KAY99]. Functional programming language has structural features like high order functions and lazy evaluation, and those features attribute to better software modularity and software reuse [HUGHES89]. Therefore XSLT programming language is suitable for mediator implementation where knowledge-based programming is crucial and highly modularized software control is needed for that knowledge-based programming [WIE98]. The Babel mediator rules are template-based in that each task instance received by the mediator runtime will be checked against all templates for the registered rules. The template concept also fits with XSLT programming seamlessly.

2.4 Mediation Logic

Active information integration architectures like TSIMMIS, SAP, MIX/CQ and COOPWARE [GM01] all provide the functionality of value-adding mediator—exploits encoded knowledge about some sets or subsets of data to create information for a higher layer of applications. However, they all have their limitations: TSIMMIS aims for managing heterogeneous data sources, and lacks of active support for data coordination. MIX/CQ is built as a successor to TSIMMIS by providing a query interface based on XML (XMAS) and UI (BBQ) for heterogeneous data sources query mainly. SAP is basically a tightly coupled information service framework that only works on top of a homogenized environment [CUR96], and therefore is not suitable for general distributed autonomous data sources mediation.

COOPWARE is the most similar to the Babel mediator in that it has the same event-condition-action rule logic and architecture for mediation. However, it has several disadvantages when compared to Babel. It lacks support for its data modeling and component interfacing; all the design time work must be done manually, where in Babel, the design time environment simplifies this effort substantially. Second, it lacks a rule definition mechanism. Users need to learn detailed domain knowledge and rule-syntax knowledge to define rules. This prohibits COOPWARE from being an easy-to-use mediation system. Finally, it lacks a workflow enactment mechanism. As discussed in Chapter 1, without the capability for continuous processing and rule-based interoperation, COOPWARE is still not capable enough for modern mediator system for application integration. The objective of the Babel mediator is to provide solutions to all the above problems.

Chapter 3 The Babel Approach to Application Integration

Figure 1 diagrammatically depicts the Babel mediator architecture. The main components of the Babel mediator are its design time environment and its runtime environment. These two environments are loosely coupled.

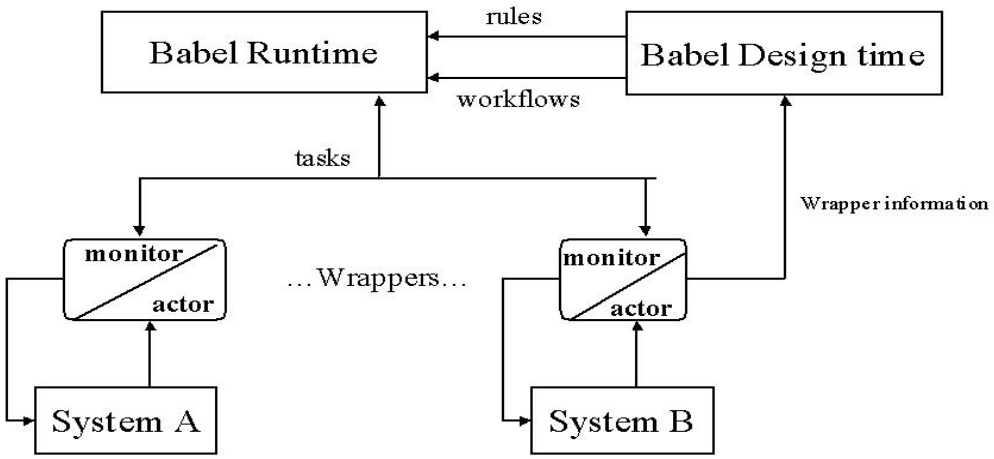

Figure 1: Architecture of the Babel mediator.

The design time environment provides to the runtime environment the definition of the coordination logic among the underlying wrapped applications, i.e., the rules and the workflow definitions. The rules are defined using a novel coordination language (as discussed in Chapter 2). The design time also extends to wrapper generation. Wrappers can be manually generated by wrapping the specified task based API (see Chapter 2) on top of existing software applications, or by using the CelLEST legacy system re-engineering tools (see Chapter 5) for certain legacy systems. Changes made to rules,

workflow definitions or data modeling take place only inside design time environment and have no global impact to the runtime meditation system.

The Babel runtime environment provides a mediation solution in a 3-layered architecture: the runtime Mediator is at the top level, and the underlying applications are at the bottom level. In-between is the wrapper middle-ware. Data exist in their raw form in the lowest layer. Both the mediator and the wrappers provide an interface for receiving and sending data that conform to a canonical data model, and interact with each other using this predefined data model. Each wrapper acts both as a monitor and as an actor for the system it represents and encapsulates. The activities of the system applications are recorded to the mediator through the applications wrappers and instructions are sent from the mediator to the wrappers that drive the applications accordingly.

3.1 Data modeling

As discussed earlier, data from the wrappers are transformed into a canonical data model and that only that data model is accepted as standard means for conversation within the mediation system. This enables external applications to drive the underlying wrapped applications and to import and export from them application domain-specific data. For the mediator, this data modeling makes the foundation for coordination logic. Therefore, Babel needs its data-modeling language to have the following features:

1. It has to enable the modeling of real-world objects easily,
2. It has to provide a functioning interface to the applications, and
3. It has to be syntactically succinct so as not to complicate coordination logic defined on the data.

The Babel mediator monitors the execution of every application through the wrapper encapsulating this application and drives these applications by sending information to their wrappers. Therefore, there are two types of information flowing between wrapper and mediator: One is of the type “event”, which is information gathered from the wrapper about the underlying application. The other is of type “action”, which is sent from mediator to the wrapper instructing it on how to execute the application. These two types share a lot of similarity, as they are both descriptions of certain activities with the underlying system applications. We view system applications as components that intake certain parameters, the component act upon the parameters and return the enactment result. Therefore, both types of information have parameter information (input) and running results (output). The only difference is that events are successful (accomplished) enactment of such activity and actions are yet to be fulfilled and therefore, actions must have one more return value showing whether the enactment has succeeded.

Babel adopts the *Task* data model for its events and actions [SS00]. *Task*s are abstraction of any procedures involving the underlying application. For event type, a task instance is the history description of such procedure. For action type, a task instance is the procedure to take place. By treating both event and action as task, Babel mediator could have a uniform view of the data from and to the wrapper, and the Babel coordination logic can be applied on both types of data to integrate action with events. A task is consisted of inputs, outputs, and its meta-data, like task type, session Id (see 3.2.2) etc.

An important feature of the Task data model is its domain specific objects information that participates in the execution of that activity of the underlying application. Both inputs and outputs contain aspects of knowledge of the real world domain in an object-

oriented way. For example, if there is a wrapper capable of querying books from a digital library using the keyword of the book, the task instance from the wrapper can be modeled in the following way:

```
{Book Query
  {input {book @keyword}
  {output {book1 @author @title @subject}
  {book2 @author @title @subject}
  {book3 @author @title @subject}
 }
}
```

This task has an input field as the "keyword" aspect of a book, and three book instances returned as output of the task. Each book instance in the output has three aspects, namely "author", "title", and "subject". These book instances are treated as equal book object entities in the output field and they are distinguished to the mediator by their object IDs.

The above task structure lacks generality. In order to depict objects in the input and output field in a uniform way, the task data model uses the "*information*"-structured data as input or output. A piece of "*information"* is an aspect of the application's knowledge of the involved objects. An *information* piece includes the object type, aspect path of the object, value, and object Id. The book1 in the output field will be depicted using three *information* data structures. For example, the author's *information* is structured as below:

```
(object_type = Book,
    relation_to_object="author",
    value="…",
    object_Id="…").
```

Each task also has some registration data. Among them, the "task-type" and "task-Id" attributes go with each task instances during runtime. "Task-Id" is globally unique and

each “task-type” corresponds to one wrapper. The “task-type” tells the mediator where this task comes from, and where should this task be forwarded as action. “Task-Id” helps mediator to track tasks in one session (see more in 3.2.2).

3.2 Coordination logic

Coordination logic is the heart of the Babel mediator. It is the specification of the mediator’s behavior upon receiving the incoming task data. Babel mediator has coordination logic occurring at two levels: the rule (3.2.1) at lower level and the session at higher level (3.2.2).

3.2.1 Even-Condition-Action Rules

Rule is the atomic element for defining mediator behavior. A rule is an *event-condition-action* triple that specifies the coordination in response to certain incoming task data (*Event*). This structure is consistent with active database convention, making the mediation event-based rather than polling-based and it enables mediation between a variety of applications (including legacy system applications).

The rule model consists of meta-data, which contains description of the rule, and rule type, rule Id, the *Event* part that specifies the task type for its invocation, and the core part. The core part contains (optionally) *Condition* of the rule and (optionally) the *Action* of the rule if condition is met. The *Condition* may be set to “true”, which means the *Action* of the rule must be taken. Rules without *Action* are only used in sessions (see later) to update session’s state.

Condition is a composite logic unit based on *Constraints*. *Constraint* is an equation based on the task event data and task information repository. It is a basic logic element that can be found in both *Condition* and *Action* part of the rule. Elements in the task event or history tasks (operand) can be further specified using *Constraints*.

Action is task template data models with specified data. Data could come from rule programmer's direct input, or come from task event or history tasks.

The following is the EBNF grammar for the Rule:

```
Rule            :: Event Condition? Action?
Event           :: task-type
Condition       :: Constraint (connectivity Constraint)*
Connectivity    :: and | or | not
Constraint      :: Operand comparison Operand
Operand         :: Functor | constant
Functor         :: (function)? Value
Value           :: aspected-knowledge (Restriction)*
Restriction     :: comparison Operand
Comparison      :: > | < | = | >= | <= |!=
Action          :: task-type (task-Id)? Input Output
Input/Output    :: Value
```

As a concrete example, consider the following scenario: a rule programmer want to specify all "email query" task instances from the history-task repository that have the same user name and email address as the event "email query" task. In doing this, the rule programmer could cast a "join" operation on the "user name" and "email address" input elements of the history "email query" task and the event "email query" task. In the "join" operation, the rule programmer chooses "equals". After specifying that task, the rule programmer can go on to specify further constraints using that task element. For example, the rule programmer could use that task as operand, and choose "count"

function on that operand, and the value of the function is greater than "1". By doing that, the constraint becomes equivalent to: "if the user has queried using email before".

The following example shows the rule layout for "if the user has queried using the keyword before, send the user an email containing book information from www.amazon.com

```
    Event: Email book query task
Condition: count(history_task
         (task_type="email book query" and
          EmailAddress=event_task.EmailAddress and
          BookKeyword=event_task.BookKeyword)) >= 1
   Action: Task
         (type="Email Answer"
          EmailAddress=event_task.EmailAddress
          Content=lookupatAmazon(event_task.BookKeyword))
```

It is worth noting that Babel has internal functionality named *Extension* that works like wrapper but no mediation is needed. The mechanism for looking up book at www.amazon.com is one of such build-in mechanisms and that mechanism can be built by hooking up to SAXON (see Chapter 6) mainframe.

Rules are incarnated as stored XSLT programs. As discussed in Chapter 2, XSLT is a functional programming language and its activation is based on templates. Each rule has its associated task as objective, and Babel uses that task-type as template for the rules. These programs can be automatically specified using the Babel design time environment, and they in turn, enable automatic coordination of the Babel mediator.

To facilitate rule registration in a distributed environment, rules are modeled using a DTD specification as well. The runtime the Babel mediator takes as input tasks as well as rules (or session, see 3.2.2) remotely. It is the parsing unit that infers the data type from data content and distinguishes rules from tasks.

3.2.2 Workflow Sessions

Coordination based solely on event-action-condition rules is not enough for Babel mediator to process data in a continuous, automated processing way. Yet that capability is required in most of today's E-commerce environment (see Chapter 1). Workflow management systems provide the automated coordination, control and communication of work needed for multiple task execution [SHE94], but as to the best of our knowledge, no existing workflow model applies a similar event-based mediation approach yet.

In order to incorporate workflow-like processing capability in the Babel mediator, Babel mediator incorporates *Session* management for such continuous automated mediation. A *session* is a light-weighted workflow process, which manages multiple tasks using certain control flow logic (sequential, branch and loop). WFMC specifies nine valid runtime states for workflow process instance [WFMC]. Babel Session as a limited workflow process chooses to incorporate the following states for its instances:

- *Not Started:* This is the initial state of every session. A session needs the occurrence of a particular task event to start.
- *Running:* After a session has started (a task event has passed the guard of one of the initial rules), the session will run a special activity (a new task) and move on to the next state.
- *NotRunning:* After a session has transitioned to its new state, and before one of the current rules' guard is passed.
- *Terminated:* The session has completed.

The expressiveness of a workflow depends heavily on its flexibility. A workflow should provide versatility in its control flow logic. There are several flow control formalisms

being applied in workflow management systems (WFMS), such as Petri-Net [AALST96] and State Chart [WW97]. Since Babel is event-based mediator, and its control flow relies on event as triggers, it adopts the *Transition Vector* as its internal data structure for control flow logic. A *Transition Vector* is a vector holding all transitions involved in a workflow process definition. Each transition is a "from-state, rule, to-state" triple, the following is the EBNF grammar for session definition.

```
Session              :: meta-data, Transition-Vector
meta-data            :: session-id, transition numbers
Transition-Vector    :: (Transition)
Transition           :: (from-state, Rule, to-state)
```

A session consists several transitions, and one or more of which with "from-state" as "0". These transitions are starting-up transitions and they decide when to instantiated a session instance. Once the mediator receives a task event, it will be checked the task event against all these starting-up transitions of the sessions already registered. If one of the rule conditions in these transitions is passed, a session instance will be spawn and its state will go from "0" to the state specified in that transition. If there are multiple transitions, the checking will follow the order the transitions are located in the transition vector. After a session instance is created, it will be entering its not-running state, until another task event comes in. If that session instance has some transitions with the session's current state and rules associated with that task, it will enter state "running" and check whether those rules' condition have met or not. Those transitions with "to-state" as "-1" are terminating transitions. Once a session instance enters the state "–1" by passing one of those terminating transitions, it will expire it self. If the transition fails (the rule of the transition not passed), the session will remain at its original state.

In order to avoid ambiguity in process definition, *Transition Vector* will not allow two transitions with the same "From" state and the same "Rule". For semantic reason, a transition vector must have at least one starting-up transition and one terminating transition.

In order to track multiple tasks taking place in a session instance, task data model has an optional data field that records the task's "task-Id". Wrappers get the action task from mediator, and record its task-id. When wrappers are required to return the result as task event in response to that action, it should set the task event Id to the task-Id it formerly recorded. When Babel applies rules on the task events, the action tasks will also have the same task-Id as the incoming task event. Therefore, it is possible to keep tract of all relevant tasks in fulfilling a multiple tasks session. For example, if the session contains two rules. One rule defines the logic: "whenever receives an email from user querying about a book, send that query to library search wrapper". The other rule defines "when receives an library search task with search result, find out the corresponding email query task in the information repository, and send the search result back to the email query task sender". With the task-Id tracing the whole procedure, Babel is able to locate the email query task in its history task repository by using the task-Id from the library search task, and therefore, it could email the query result back to the user who asked about the book.

Chapter 4 The Babel Run-time Environment

This chapter discusses the runtime enactment of the rules and sessions defined over the wrapped applications by the runtime Babel mediator.

4.1 Mediation Run-time Procedure

At runtime, the Babel mediator receives rules and sessions to be registered and enacted and new tasks events. Whenever Babel server detects an incoming stream of XML data, it parses it, infers its content type (DTD), validates it (see if its content goes with the DTD), and classifies it into either control data or *Task* data. If the input is a *rule* definition, the Babel mediator stores it in its rule repository, and makes an association of all the tasks with the "task-type" specified in the rule with it, so that instances of these tasks will activate this rule. If the input is a session definition, Babel will register that session with its Session Manager. If the input is a *task* instance, the Babel mediator will put that task into its internal message queue and get ready for next input.

During the same time, Babel has a thread running to process the tasks from the message queue. It pops up a task instance out of its message queue First-Come-First-Serve. Then it retrieves the "task type" attribute out of the task instance. Next the task instances is first examined by Session Manager to see if the triggering condition from state 0 of a defined session has been triggered. If it has, a session instance is created. Then all the session instances are checked against this task for possible session state update. Second, all those registered rules that may apply on that kind of task are identified. Each task-rule pair

defines a processing job to be accomplished. All those jobs, together with the session instance processing, are accomplished in parallel using Babel's worker threads from its threads pool (see 3.4). After all those rules and sessions processing completed, Babel dispatches all the new tasks generated by the rules or sessions to their wrappers, and update its information repository to include the newly added task. Below is the pseudo-code for the whole process:

```
Front End:
     While(true)
       {
            data = Wait_for_data();
            If( data.DTD == rule)
                 Register_as_rule (data);
            Else If (data.DTD == session)
                 Register_as_session(data);
            Else If(data.DTD == task)
                 Enqueue_task (data);
       }
Back End:
       While(true)
       {
            task = Take_Task_From_Queue();
            For_each_session
             {check_and_run(task, session)};
            For_each_rule
             {check_and_run(task, rule);}
       }
```

4.2 Information Repository

The Babel runtime mediator is designed for processing asynchronously multiple tasks. Wrappers send information to the mediator and return immediately. Babel may encounter burst of incoming tasks in a short time and fail to respond to later wrapper requests promptly enough. Therefore, the Babel mediator front-end receives incoming tasks, saves them in a temporary buffer, and lets the back-end process them with associated rules and

workflow sessions. Babel processes incoming task events on First Come First Service (FCFS) base. The following sequence diagram depicts Babel runtime:

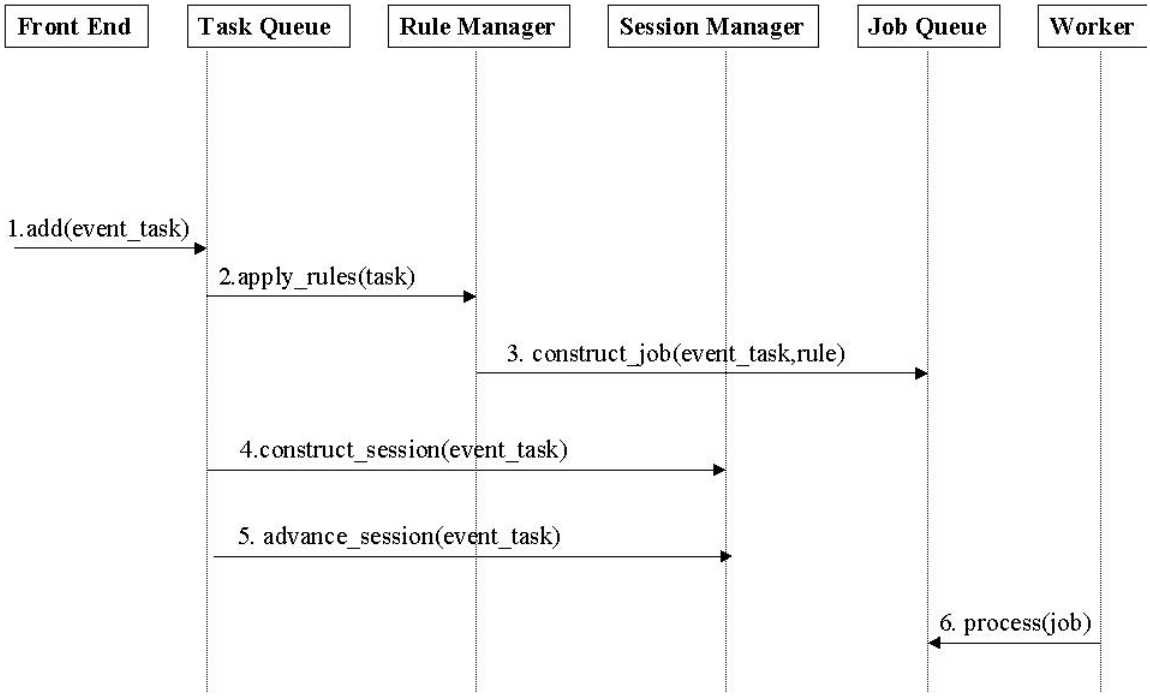

Figure 2: Babel runtime sequence diagram

Each task is processed within the context of history tasks. For example, when a "borrow book" task has triggered a rule like "if the borrower has borrowed the book for more than 3 times in the past, urge him to buy it at Amazon", Babel should process the current task in context of all past tasks with the task type of "borrow book". When the information repository of history events becomes large as events accumulate, scalability and performance becomes an issue.

4.2.1 The Need for a Database

There are several reasons that make database support necessary for the Babel mediator information repository. XML applications have not yet reached a maturity level that most XML developers are concerned about scalability issues -- managing very large collections, version control, concurrency, and so on. Babel also faces the same problems when dealing with large history data as context for incoming task data processing.

Most of current XML parsers and XSLT implementations are based on Java, whose object storage in memory and running time performance are not as efficient as that of C++. According to our experience in developing Babel, Java DOM implementation of XML document in memory is 10 times as big as the original document. Java XSLT implementation of DOM is even larger. On the average, the SAXON implementation of XML DOM is 20 times as big as the XML file on disk. Babel history task repository becomes unmanageable when the number of tasks in the information repository exceeds 1000. The plain task data amount only to 1 Megabytes as in a persistent XML file, but the Babel runtime needs more than 20 Megabytes to hold that task instance in memory, and that exceeds the 16M Java default maximum heap size (see JDK specification at http://java.sun.com/products/jdk/1.1/docs/tooldocs/solaris/java.html).

There is no capability to recover (rollback) a task collection process once the system crashes because of resource problems and hence the whole data repository becomes corrupted. However, database research has already addressed these problems, including scalability, parallel architectures, query optimization, and other technologies that will become important as the XML data becomes huge as unmanageable by common XML application.

In database management data elements are integrated and shared among different files; it is program that controls the structure of a database and that access to data. Besides the above mentioned, databases also offer reduced data redundancy, improved data integrity, more program independence, increased user productivity and increased security [RG99]. Taking all the above into consideration, it would be desirable for the Babel runtime mediator to make use of DBMS for its information repository.

4.2.2 Relational vs. Object-Oriented Database Management

Since the information within Babel mediation is based on XML data, a tree structured graph with indefinite depth of branches, there are certain requirements on the mechanism that would be used to save this XML data into a database. If a RDBMS were used, the problem of saving object data into the row and column grid of relational storage should be addressed. SQL has several shortcomings for addressing this problem. Most notably is the inability to work with variable depth hierarchies, or more generally, graphs. Storage of such data in table of columns and rows seems to be an awkward problem. Querying of such tree-structured data is also problematic, though recursive queries have been in the standard for years. Major RDBMS vendors provide SQL extension as workaround. For example, Oracle has its "CONNECT BY" option, which made querying of tree structured data easier. However it does not allow “Join” operation and “nested query” to be associated with this option. Although database researchers have workarounds with it [CELKO] [KM00], the relational database based query solution on tree-structured data still look like an ad hoc work-around rather than a general solution.

The difficulty of mapping tree-structured data into RDBMS suggests a shift to Object Database Management System (ODBMS) technology, or Object-Relational Database management System (ORDBMS). XML looks more fitted into hierarchical object world, which ODBMS has a direct mapping to. However, ODBMS has not as successful as RDBMS. It lacks of the integrated theoretically foundation and tends to have weaker security support. Many query procedures must be coded ad hoc and do not have a general query language solution. There is no widely accepted query language standard for OO-DBMS, while there are readily various SQL extensions or SQL plus workarounds to query XML data in RDBMS. Besides, OO-DBMS query optimizers are inferior to those for RDBMS. Generally speaking, OO-DBMS has less commercial successful cases and it continues to be a possibly useful but not reliable alternative.

A novel database paradigm like Object Relational Database Management System (ORDBMS) has emerged as a combination of RDBMS and OO-DBMS solutions. However, ORDBMS is intrinsically RDBMS with object extensions that are intended to increase the RDBMS ’s power and ease of use. They are not substituting OODBMS in that they are based on RDBMS, their internal structure and functionality is different from that of OODBMS. SQL standard of ORDBMS is in a state of flux. Many features of ORDBMS are a double sword and should be uses with great care. [DATE98]

The development of XML database becomes a dilemma: either trade scalability, efficiency, and robustness of RDBMS for the easiness of developing of ODBMS, or vice versa. However, database researchers propose that SQL's inability to work with tree-or graph-structured data is a "common myth" [CELKO]. In reality, the physical organization of data in a database (physical model) is separate from the logical model used for data

access (trees, networks, objects, or rows and columns). Either model of Relational or ODB can be implemented in the same way. For example: UniSQL uses pointers, objects, and SQL. Products such as Oracle 8i, Informix Internet Foundation.2000, and IBM DB2 Universal Database offer multiple alternatives for storing XML documents. Provided there are RDBMS schema solutions for XML data, Babel mediator accepts RDBMS solution for its scalability efficiency and robustness.

4.2.3 Relational Database Solutions

A lot of research work has been done on saving and querying tree or graph structured data in RDBMS. This section discusses several of them.

4.2.3.1 A simple solution

Algorithms have been developed to save tree-structured data into a RDBMS that supports standard SQL2. The algorithm works as follows:

Every node in the tree is treated as being equally important and is saves as a record in a table. The algorithm first adds every record in the table with two link values, and each node has its own label (A, B, C etc.). Then the algorithm performs a pre-order iteration within the tree, numbering each out-going and in-coming link in this iteration. When comes to a leaf node, this algorithm labels its outgoing and incoming link as a circle. Therefore, each node has one outgoing link number and one in-coming link number. After all the links are numbered, this algorithm saves the pair of link number of each node in the table. The node's first link number is this node's outgoing number, and the node's second link number is the incoming link number.

Suppose that the data to be stored follow the DTD shown below:

```
A:= (B*, C)
B :=(D*, E, F*)
C:= (F*, D)
```

This DTD requires A has zero to many number of B as children, followed by a C. B has zero to many numbers of D, followed by a E, followed by zero to many F. C has zero to many F, followed by a D.

The following picture shows an instance document tree of this DTD:

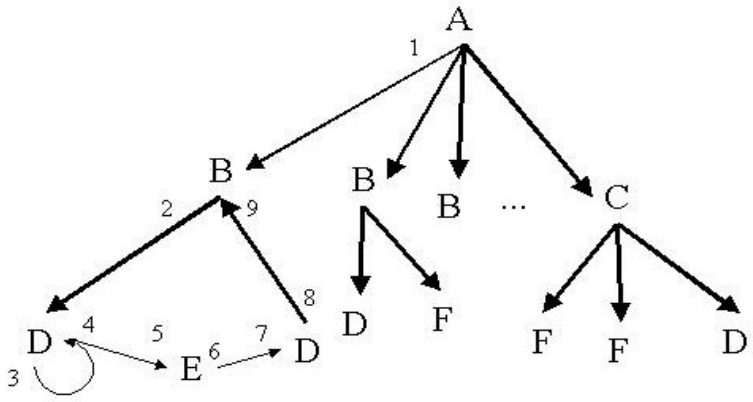

Figure 3: Sample tree structured data for predefined DTD.

Starting from the root node A, the algorithm goes to the left most B node. The algorithm numbers the outgoing link of node A as 1. Then it goes on to node D, labeling the outgoing link of B as 2. Node D is a leaf, and its circular link is assigned link number 3 and 4. After node E and node D are numbered, the algorithm goes back to node B with the link number of 9. And it finishes with the following table:

A	1	26
B	2	9
D	3	4
E	5	6
D	7	8
B	10	15

D	11	12
F	13	14
B	16	17
C	18	25
F	19	20
F	21	22
D	23	24

Table 1: Link for tree data.

Once the table is completed, structure of the data set can be easily deducted from left and right link value of each node. For example, if we want all the nodes that are leaves, we simple retrieve all the nodes that meet the following equation:

```
Right_Link –Left_Link == 1
```

If we want the parent nodes of node A, we can specify the nodes using the following query:

```
Select all nodes that node.Right_Link > A.Right_link
                    && node.Left_link < A.Left_Link
```

If it cannot be assumed that all the nodes in the tree are of the same type, we can take each type of elements into a separate table, filling their node information in those tables.

4.2.3.2 Commercial RDBMS Approach

Commercial RDBMS products such as Oracle, DB2, and SQL-Server implement their own SQL extension to support tree-structured data storage. For example, Oracle has its own tree structure extensions (CONNECT BY ... PRIOR). The data-link approach for saving tree-structured data can be further supported. Similarly, each object (node) in the tree can be represented as a row in an SQL database. Each node must have a primary key

and a reference key. In most scenarios, node has already primary key, so the space overhead is just the reference key (one integer field).

For the similar tree structure shown above, we have the following database setup to keep the structural information.

Node	Reference
A1	NULL
B1	A1
B2	A1
B3	A1
C1	A1
D1	B1
D2	B2
D3	B2
D4	C1
E1	B1
F1	B2
F2	C1
F3	C1

Table 2: RDBMS link table.

Queries can be made with a visual effect on the above structured data by using the RDBMS vendor specific SQL extensions. For example, by using Oracle's "CONNECT BY" operation and its padding support, users can issue the following query to obtain the corresponding query result.

```
column padded_name format a30
select
 lpad(' ', (level - 1) * 2) || name as padded_name,
 slave_id,
 supervisor_id,
 level
from corporate_slaves
connect by prior slave_id = supervisor_id
start with slave_id = 1;

PADDED_NAME                SUPERVISOR_ID  LEVEL
```

```
--------------          --------------- ----------
A1                                  1
   B1                          A1    2
   B2                          A1    2
   B3                          A1    2
   C1                          A1    2
     D1                          B1    3
     E1                          B1    3
     D2                          B2    3
     D3                          B2    3
     F1                          B2    3
     F2                          C1    3
     F3                          C1    3
     D4                          C1    3
13 rows selected.
```

IBM DB2 and SQL-Server have similar connection features in their products. These features facilitate tree-structured data query and data transformation/reconstruction.

4.2.3.3 A Combined RDMS Solution

There are two approaches to store and query XML data for RDBMS, according to the granite of XML data being decomposed into the SQL table. The first approach is to save XML document as a large data chunk in RDBMS table, leaving meta-data of the documents to be accessible by SQL queries. The whole XML document is retrieved as a RDBMS query result and XML API is used to further query at an intra-document level. In the case of Babel mediator, each task instance data can be saved in the table as a document, leaving the keywords, task-type, and task-Id of the tasks as key word for the document. Each task instance corresponds to a row in the table. Suppose there is a query based on all those tasks with the type of "Email Query" and happened between Nov 30th, 2000 and Dec 31st 2001, RDBMS will then select all suitable tasks by querying their meta-data, and reconstruct a large XML document based on the query result. Then user

could further specify query operations on the assembled XML data pieces by using XML API.

The second approach is to decompose each task XML data into fine-grained elements and have each element correspond to an element in the RDBMS table, and totally rely on SQL to manipulate elements at the intra-document level. XML data to relational data mapping has been introduced earlier in this chapter. This approach enables users to speed up queries by using indexes on columns. Users can map commonly searched elements into columns, place indexes on those columns, and perform document section searches. In the second approach, XML data reconstruction from decomposed data elements in the table at runtime can be very expensive operation. This approach is only useful when XML document query relies mainly on SQL, and is not as efficient as the coarse-grained approach for Babel mediator that uses XML API for most internal XML document processing.

Several major companies have their solutions in various ways, besides the capability of SQL to process nested structures. Informix takes the data-blade approach for XML, rather than trying to decompose XML into relational form. Microsoft SQL Server 7.5 will do on-the-fly transforms between SQL and XML. Major companies have adopted both strategies as an integrated solution to address the problem. Informix, Oracle, IBM, and Microsoft have developed solutions for their SQL servers that treat a document as a column, or decompose it and store meta-data that describes the document structure. They have also extended their servers to include XML parsers and extend their SQL dialects to support XML processing. For instance, Microsoft has its various XML support like “FOR

XML EXPICIT", "OPENXML", and "Update grams", and Oracle add support for XPath expressions in SQL queries.

While Babel was being developed, there was no XSLT API for querying and constructing query results from RDBMS systems. Oracle announced its future Oracle 9i would be able to use XPath expressions, supporting XML API (XPATH) query over non-XML-native DBMS. It can be foreseen that more RDBMS vendors will provide better coarse-grained XML data support, and there will be more choices between XML querying functionality provided by RDBMS and standard XML API.

4.2.4 Benchmarking

There are several solutions for storing the task XML document using the coarse-grained storage. Most RDBMS provide users two options to save XML document:

1. Using XML data in database, parsing at query time.
2. Saving serialized DOM in database, no parsing at query time.

The advantage for the first option is that XML text data is kept in its original form in database. Applications based on third party XML parsers also have access to this data. The drawback is the XML parsing overhead at runtime. The second option saves XML parsing at query time, making queries faster. However, XML Document Object Management (DOM) serialization may take considerable much more space than original XML file. Moreover, since the DOM implementation is parser specific, DOM created using one XML parser may not be used by applications based on third party XML parser implementation.

Each category has two possible alternative implementations based on whether the DOM saved in database is raw TEXT string or the DOM saved to database as binary large object (BLOB). Saving data as BLOB has the advantage of being robust in handling all kind of data may occur in the XML file.

When converting the XML DOM byte stream to long TEXT String, some encoding mechanism (base64) is necessary to make arbitrary byte array to string data that is acceptable to RDBMS table. Suppose the XML data has an unparsed entity of image or video data in it that contains characters out of the range of XML allowed characters (0x00 to 0x1F, other than CR, LF, TAB). If that XML file is to be saved as a BLOB structure outside of the table, there will be no need for encoding. Also some RDBMS have restrictions on the length of long text in table space, and some naive RDBMS implementations have no support for long text strings. These factors inhibit the use of long string as XML data representation.

However, the BLOB approach requires keeping pointers pointed to the BLOB structure and saving those pointers inside the database table, rather than saving XML data directly inside the database table. There will be considerable overhead to fetch the data block from the pointer in order to query inside that data block. This overhead can be quite expensive at runtime.

Simulation is performed for each approach to retrieve different amount of task instances from databases holding different amount of history task data. The simulation is based on the “Email Query” task event data, and is performed on Windows NT 4.0 platform, Intel CPU Celeron with 500MH, DRAM 128Megabytes, using Microsoft ACCESS.

Method of storing data	10	20	50	100
1. XML text	140	270	631	1492
2. XML BLOB	981	1692	4216	8863
3. DOM text	380	730	2392	4426
4. DOM BLOB	1304	2739	6300	11034

Table 3: Time (in milliseconds) required for querying a number of task data using different approaches

Table 3 shows the average time for querying 10,20,50, and 100 tasks data from database when the XML data is stored using different strategies. These approaches are

1. Storing XML data as text in the table.

2. Storing XML data in its raw data form as BLOB outside the table,

3. Storing XML data as parsed DOM object in the table (after some encoding), and

4. Storing XML data as DOM object as BLOB outside the table.

The simulation shows that storing data in as unparsed XML string as LONG_TEXT in a table has the best performance. Storing XML data as raw text in BLOB incurs the overhead of extra database operations, both DOM approaches have overhead of object serialization and encoding/decoding. It can be inferred that object serialization and BLOB operation are much more expensive operations than XML data parsing.

4.3 Parallel Processing and Workload Balance

As mentioned before in this chapter, each task event may trigger multiple rules or sessions, and the Babel mediator processes these rules and sessions in parallel -- each rule

or session is applied on the task event by an independent thread. Since session processing is based on rules, discussion is focused on the parallel application of multiple rules to tasks. The principles discussed here also apply on session processing.

Many server-side applications restrict the amount of threads that can be running at the same time, for fear that tremendous thread running on the server will exhaust all resources and none of them would have access to the CPU or memory resources it needs to run. Meanwhile, it is required to assign dynamically balanced workload to threads. The jobs are not evenly distributed, in terms of the number of jobs that each thread actually processes, but they are distributed so that most threads are kept working instead of finishing far earlier than other threads and stay idle afterwards.

As a solution to the above stated problems, Babel adopts the *Working-Thread Pool* model. When the Babel mediator runtime initiates, it creates a fixed amount of threads, and groups them in a Working Thread Pool. This approach shortens mediation processing latency because creating a thread and starting it off at mediation time is an expensive operation. Meanwhile, keeping a fixed number of threads ensures the stability of the system because threads will not grow indefinitely to exhaust system resources. The Job Pool is a data structure that takes as input jobs. The Rule Manager dynamically puts task-rule pair jobs into this Job Pool, while the threads (workers) in the Thread Pool randomly pick up those jobs and execute them. Worker threads are continuously trying to get a job from the Job Pool. A thread either starts working on the job on successful job retrieval, or suspends itself until new jobs become available. This Working Threads Pool (WTP) paradigm also solves the problem of workload balance between the threads: the jobs in

the Job Pool are dynamically assigned to the worker threads, ensuring an efficient workload balance.

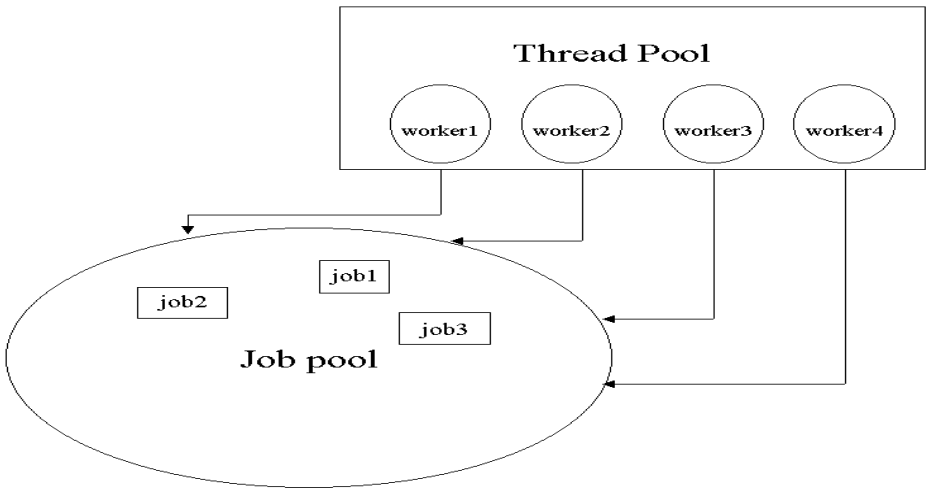

Figure 4: Working Thread Pool vs. Jobs Pool paradigm.

When adopting the WTP paradigm, one has to address the following issues:

- Concurrent Access: the jobs, i.e., that "task-rule" pairs, are put into the Job Pool where they will be picked up by worker threads in a random order. Because multiple threads may be competing for the same job in the Job Pool simultaneously, the Job Pool data structure must be kept thread safe. There should be only one successful worker thread for each job and all other threads must yield and request for the other jobs.
- Internal State Transition of the Worker Threads: When the Job Pool becomes empty, the worker threads should not be busy waiting for new jobs to come. This kind of busy-waiting would cost a lot of CPU time and memory resources [WA99]. Rather, they should suspend themselves, and let the Job Pool notify them when new jobs become available.

- Synchronization using Barrier: Each rule is applied on the current task event in the context of the history of previous tasks that have occurred. Before all rules are applied on the current task event, no task events can be taken out of the event queue and processed by the mediator. Only after all the workers have finished working on the current task event, could the Babel mediator begin to dispatch the new tasks to the wrappers and update the information repository. In order to synchronize the workers with the main Babel thread, the mediator applies barriers [WA99] on all the workers in the thread pool. The Barrier construct is used in parallel computing where all participating threads must arrive to one stage before they can go on to the next stage. When the input task invokes a number of rules, the mediator sets up a counter with the value of the number of triggered rules. Each thread worker will notify the mediator after applying the rule on the task, and the counter will decrease by one. The mediator could take a task event out of the event queue only when the counter is set to zero. That way, the mediator can make sure that all rules have been applied on the task before it goes on to process the next task event in the event queue (see Figure 5).

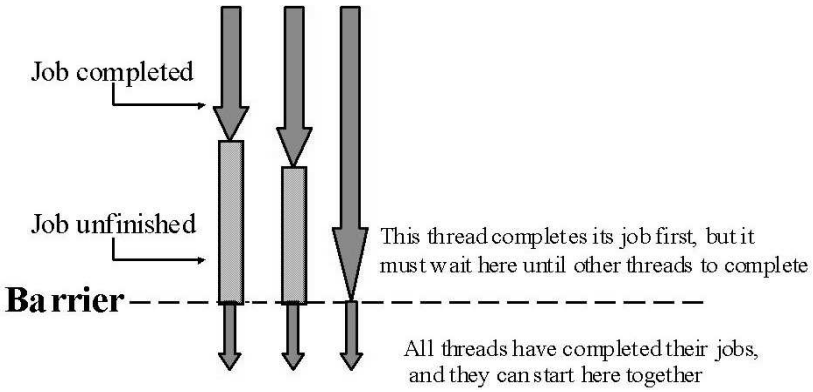

Figure 5: Barrier

Chapter 5 Babel Wrappers and Extensions

The Babel runtime mediator relies on wrappers for enacting the action tasks as instructions to the underlying applications as well as extracting information from the applications. The wrappers provide canonical communication interface for the applications. This chapter discusses the wrapper infrastructure and inner wrapper (Babel *Extension*) implementation.

5.1 The Wrapper Interface

The Babel mediator runtime interacts with its environment (the wrappers) by sending and receiving task XML data. This interface mainly uses Java Remote Method Invocation (RMI) as calling convention. RMI is a distributed programming mechanism that enables functionality implemented on one machine could be remotely invoked on other machines through network. RMI can be used to bridge existing systems using the standard Java native method interface (JNI). RMI can also connect to existing relational database using the standard JDBC package. The RMI/JNI and RMI/JDBC combinations provide solution for most of communication problem for today's existing servers. Besides, RMI as Java remote procedure call (RPC) implementation has the following advantages over traditional RPC systems:

1) RMI is completely based on Java, which means this architecture is highly portable.

2) Unlike traditional RPC, RMI is an Object-Oriented API and takes as input parameters as objects. In existing RPC systems, both client and server should define mechanisms for marshalling and decomposition for complex data structure as parameter.
3) RMI hides implementation details from client, and that offers better transparency. For example, an interface is defined for a particular service. The client invokes this service by calling that remote interface at server side. Suppose that the service is updated after the client has been deployed, only the implementation on the server's side needs updating and the client system can be left untouched. This is because the RMI client requests the service by dynamically downloading the server implementation proxy and lets the proxy call the real implemented service from the server. When the services change, the server will start returning a different implementation proxy of that interface. This gives system maximal flexibility, since changing implementation does not requires client users to download and install newer version on their side. Therefore, this flexibility helps Babel mediating in a highly distributed environment.

5.2 The Wrapper Architecture

Babel wrappers have the following architecture:

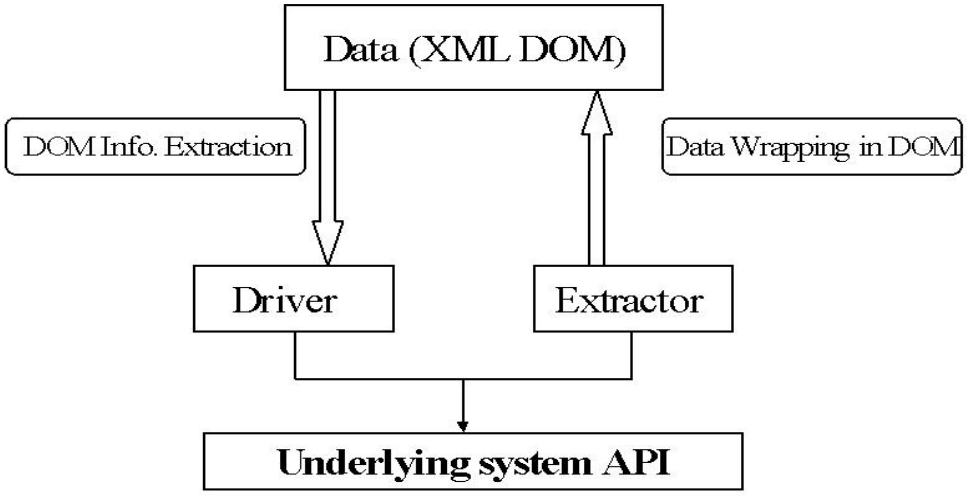

Figure 6: The Wrapper Architecture.

This architecture consists of three layers:

1) the XML data interface layer,
2) the driver and Extractor layer, and
3) the API to the underlying applications.

The XML data interface layer retrieves task data from mediator and parses it. It is also responsible for getting data from underlying application, formatting the data into task structure, and sending data to the mediator. This layer shares common functionality with most wrappers and contains general re-usable modules.

Driver and *Extractor* are wrapper specific modules. The *Driver* is the module that takes as input information provided from the Babel mediator to drive the underlying application. For instance, the email wrapper gets task data from mediator, extracts the email address and user name meta-data, and makes the system call to send the email. On the other end, information feedback from the applications will be detected and sent to Babel by *Extractor.* For example, the email wrapper will monitor the system for any

email with the subject "Email Book Query". Once the email is found in the mailbox, the email content together with the email address and email user name will be gathered by the *Extractor*. This information will then go through the XML data interface layer, and finally be dispatched to the Babel mediator in a *task* structure.

5.3 The Babel Extension Service

Babel wrappers are heavy-weighted components that monitor and drive the underlying system. Babel also has its internal wrapper (termed as Babel *extension*) service as light-weighted components. These extensions are co-located within the Babel mediator runtime, and there is no communication overhead involved for interacting with those extensions, and communication with these extensions is synchronized rather than asynchronous as with wrappers. For example, a rule can be defined as following:

"*when a user has query about a book, first query the book using www.amazon.com, and return the query result in email back to the user*."

Instead of asking an independent wrapper to query the book at www.amazon.com, the mediator could fetch the result by calling its extension directly, block until the result becomes available, and send it right back to the user. The Babel mediator provides a framework for building such extension services as hooks to this framework. The process of developing an extension involves the following steps:

1) Define the information processing logic, specify the procedure and implement the functionality of such procedure.

2) Hook the module into main processing framework. Since Babel is based on SAXON, the hooking should follow the interface specification provided by SAXON mainframe.
3) Make corresponding changes to design time environment to allow users specify the processing while designing rules.

Chapter 6 Implementation and Performance

This chapter discusses in detail the status of the Babel implementation and a series of performance-evaluation experiments based on the current implementation.

6.1 Implementation

Both the design time and the runtime environments of Babel are implemented using Java (JDK1.3). The XML parsing mechanism is provided by JAXP 1.1 [JAXP], and the XSLT processing engine is based on SAXON 6.2.2 [SAXON]. Parallel processing inside the mediator message processing is implemented using Java threads. Synchronization and barriers are implemented using the Java thread API. The mediator information repository is currently based on a database implementation layer that supports flat files, Object-Oriented, or Relational databases. All these repository implementations can be plugged into the mediator-processing engine. This is achieved by layering the mediator coordination unit on top of the repository unit. The prototype implementation includes DBMS support like Oracle and ACCESS, and flat XML file support. Currently XSLT has little built-in support from RDBMS, and the Babel integrated system can only make use of flat XML file. However, it is foreseeable in the near future that RDBMS vendors will provide better XSLT support for XML data storage (for example, Oracle 9.0 will support XSLT query), and by then the Babel mediator will have better scalability.

The Babel mediator has its internal message-queuing sub-module so that the wrappers interact with the mediator asynchronously by sending a message to the Babel mediator

and the message sending thread returns immediately. Wrappers are recommended to have asynchronous calling interface, but that is not mandatory because the Babel mediator dispatches messages in separate thread running parallel to coordination processing. All messaging takes place at the communication level, which can be RMI, Servlet, or CORBA. RMI is chosen for the Babel mediator prototype because RMI is pure Java and that makes the mediation subsystem totally platform independent and portable. Another reason is that RMI is Java (OO) RPC for its distributed environment, and requires lighter environment setting than Java Servlet does.

Wrappers are built using Mathaino [KAP01]. Mathaino is part of the CelLEST [STROULIA00] project, which aims at re-engineering legacy systems and providing modern interface to these legacy systems semi-automatically. Mathaino can generate Object Oriented task based API for driving and probing the underlying legacy system, based on tracing information collected from other CelLEST project modules. Mathaino is also developed using JDK 1.3, and forms an extended design time environment with the Babel mediator. During runtime, the wrappers interact with legacy system via a BlackSmith proxy, which is a driver [BLACK] for driving and monitoring the legacy system.

The Babel design time environment is implemented as an independent Java (Swing) application. The processes of rule definition and session building are supported through visual-programming interfaces, the Rule Wizard and the Session Builder. Both the Rule Wizard and Session Builder are plug-ins to the CelLEST project mainframe.

Figure 7: The Rule Wizard and Session Builder as plug-ins.

Through the Rule Wizard, rules can be defined in five simple steps:

1. Describe the rule
2. Select event type, select all relevant history tasks
3. Make Condition
4. Select Action template
5. Make Action (optional)

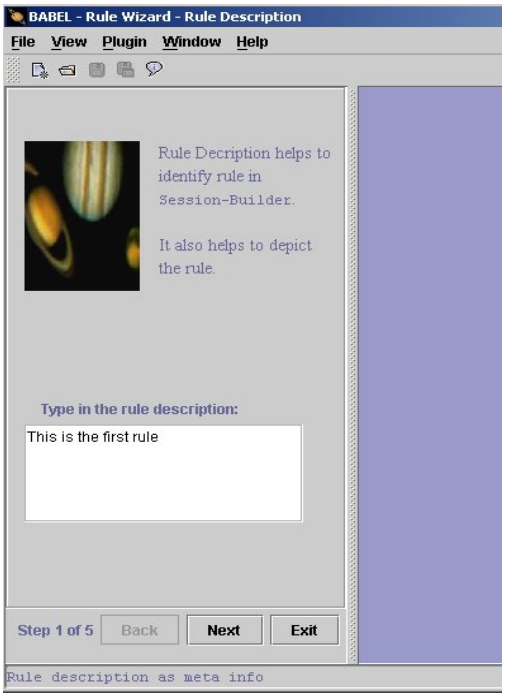

Figure 8: Rule Description.

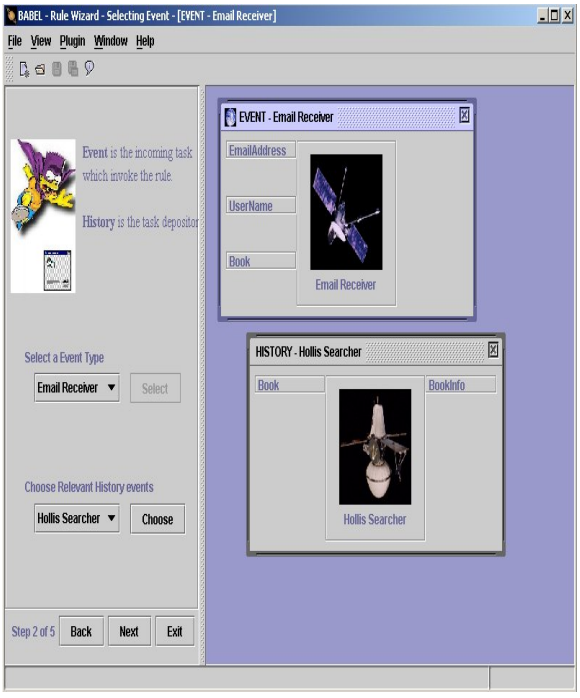

Figure 9: Event/History Selection.

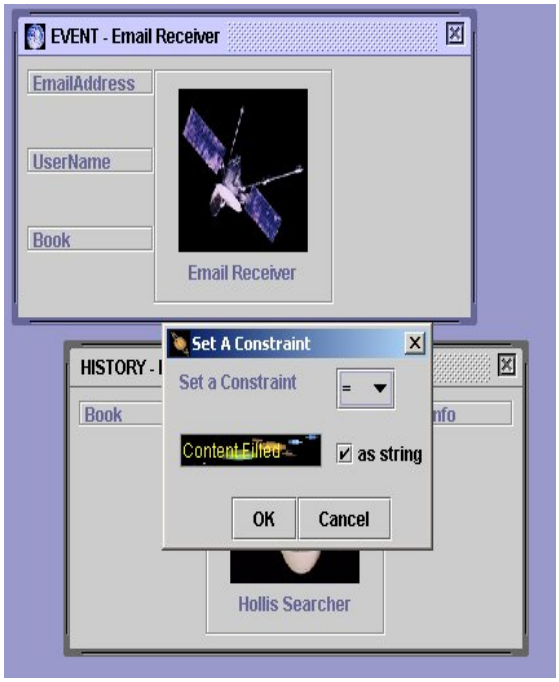

Figure 10: Join Operation.

Figure 11: Constructing a Condition.

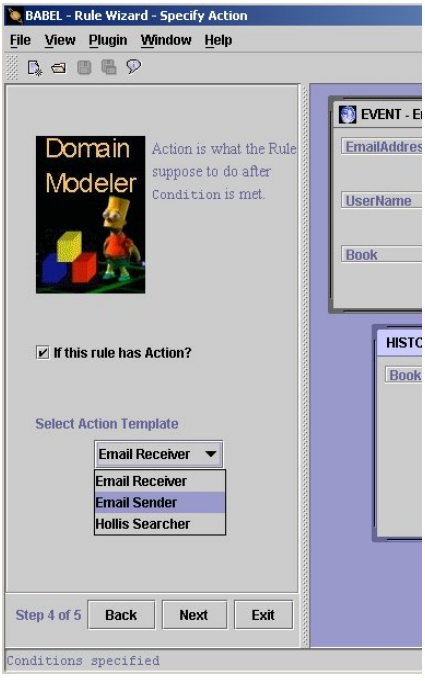

Figure 12: Action Template Selection.

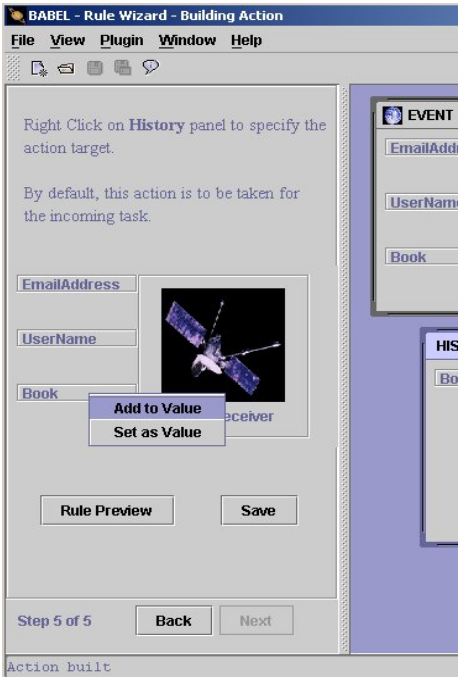

Figure 13: Defining an Action.

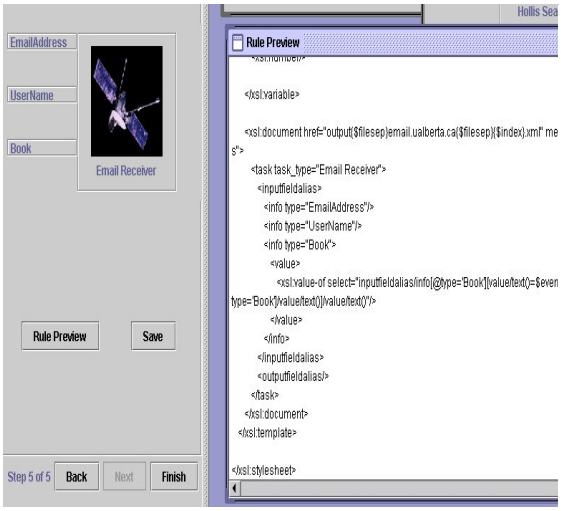

Figure 14: Rule Previewing.

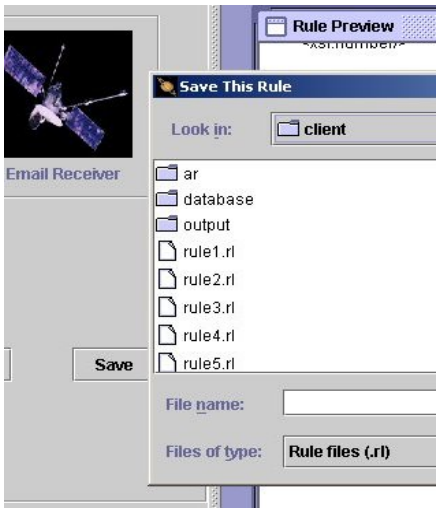

Figure 15: Saving the Rule.

All rule logic can be accomplished by drag-and-drop or dialog based operations on the Rule Wizard, and there is no need for XSLT knowledge required. For example, Figure 8 ~ Figure 15 show the procedures for defining the Condition “if the user has queried about the same book keyword before”. All the rule programmer needs to do is to drag & drop the “user name” item of the task event to the corresponding item on the “Email Book Keyword Query” history task to make a “join” operation. A dialog will pop up asking for relational logic of the” join”. The rule programmer selects “=” relation, and drag-and-drops the history task to the upper column in the condition maker panel, and selects “count” operation on that column. Then the rule programmer selects relation of “>=” and “1” as operand, meaning that kind of task counts more than 1. Figure 14 shows the preview of the XSLT program generated as rule.

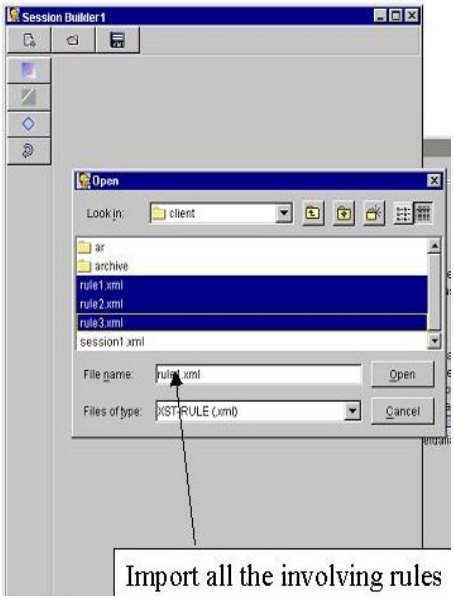

Figure 16: Importing the rules.

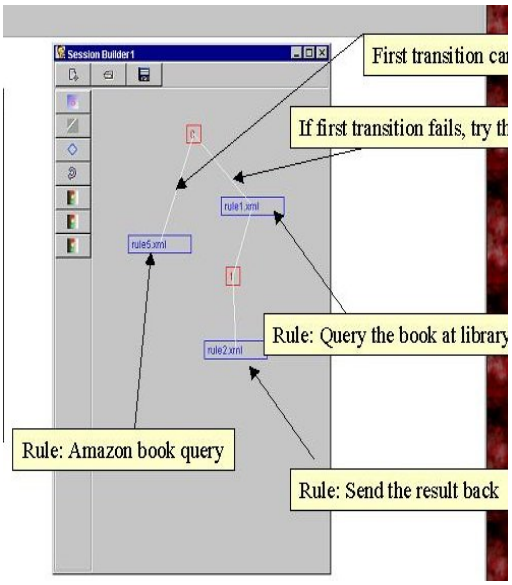

Figure 17: Transition building.

Workflow sessions are defined similarly using the Session Builder based on the rules already defined. In order to define a session, programmers should first import all component rules for this session. Then they can define initial state and state transition. Rules with *Condition* only (no *Action*) can be used for session state update purpose here. Figure 17 shows the configuration of the workflow definition: if the user has queried the book before, send him a promotion letter from Amazon bookstore on that book, other wise search it and forward the result back to the user.

Rule and session registration uses the same RMI interface for task events. Babel has a client side tool to register rules or sessions for the users. The mediator, upon the receiving of data, will distinguish it as rule, or as session, or as normal event messages. There is no need to register wrappers with mediator during runtime. The wrapper has its name specified with the design time and each *Action* will have a wrapper's name with it. Each

wrapper will register its existence through RMI registry once it is started. The mediator runtime will simply call the wrapper by its name from RMI naming service. If that wrapper is not available, the mediator will just ignore the message dispatch until that service becomes available.

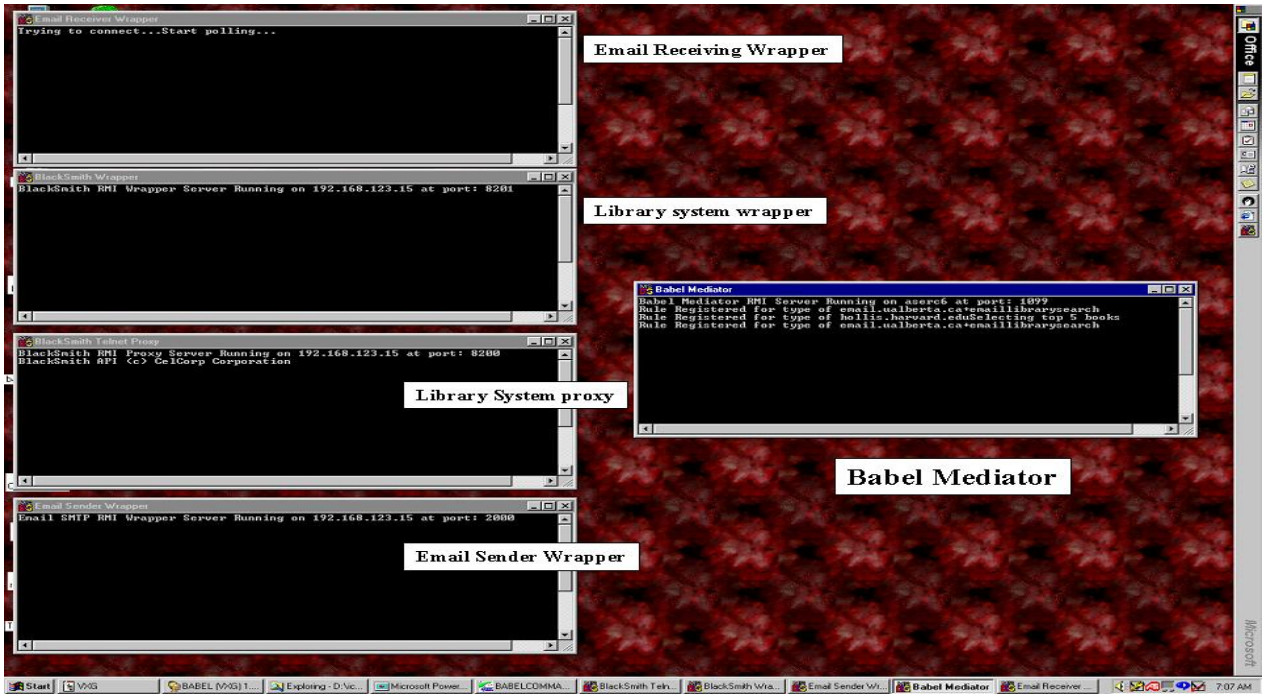

Figure 18: The Babel mediator and the wrappers initialized.

Figures 17 and 18 show the whole enactment of the above session logic: the mediator is initialized and ready for messages. The POP3 wrapper is busy polling email box for messages with certain subjects. The SMPT wrapper is waiting for sending messages. The library search wrappers and the legacy system driver proxy are started. Then Babel users can use the mediator client tool to register the session definition. The mediator will keep that session in its Session Manager.

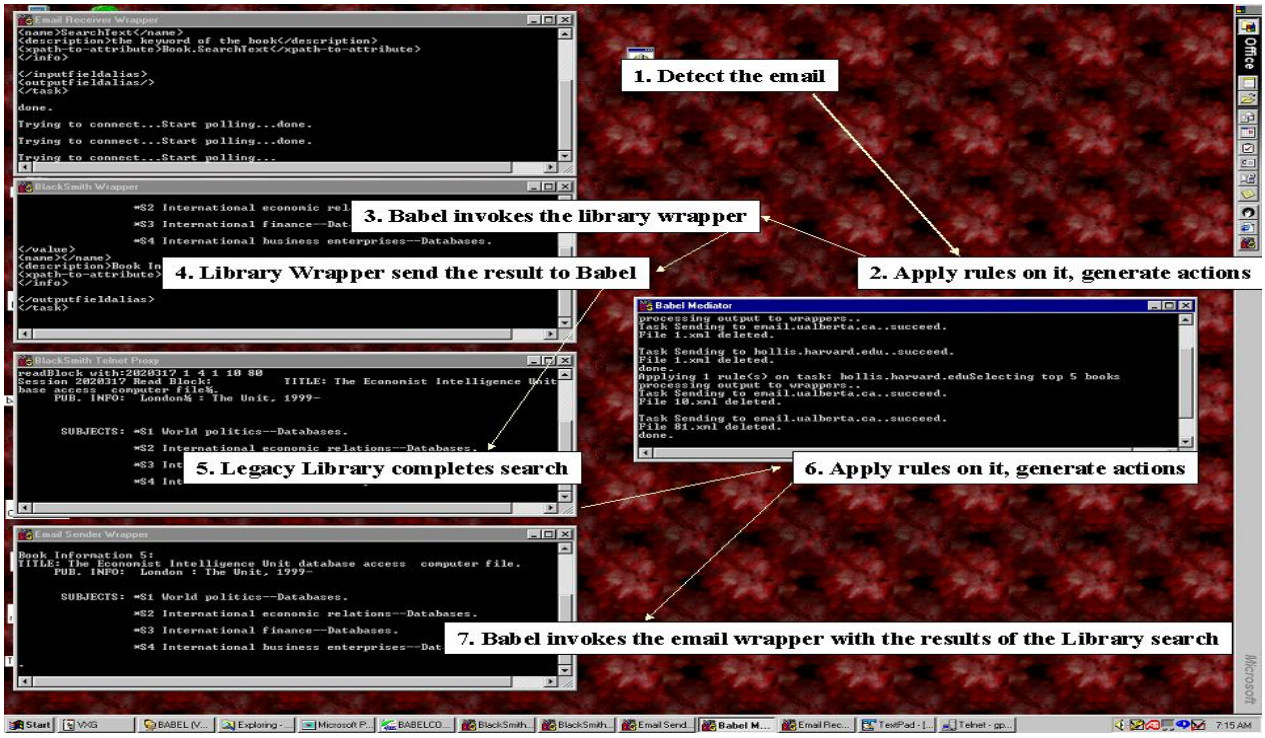

Figure 19: The mediation procedure.

Once an email with the subject of "Search Book Keyword" is traced down from the POP3 wrapper and this event is received to mediator, the mediator will spawn a new session instance for that event. The session will check for the first possible transition: "If the user has queried the book before", and that condition is not met. Then the mediator will check for the second transition. The rule for second transition is "search the book using the legacy system wrapper", while the *Condition* for the rule is set to "true". Then search action will be forwarded to the library search wrapper and the session will go from state 0 to state 1. When the result task arrives to the mediator, the Session Manager retrieves the event type from the task event, and searches for sessions with that event type for its current session state. The only rule to be found there is "forward the result back to the mediator". Therefore, the mediator will find the "email query" task instance from its

information repository that has the same session ID, check the user name and email address, and generate an *Action* of "email response" to be dispatched to the SMTP wrapper. The session transits from state 1 to state –1 and successfully completes. When the mediator receives a second "email query" task event, the same will occur except that the session will choose the first transition and complete right after the Amazon promotion letter action is generated.

6.2 Performance Evaluation

This section talks about the efficiency and scalability of the Babel mediator through its performance evaluation, and concludes its feasibility through the profiling. The Babel mediator runtime processing involves the following four steps:

1) Receive the task message, and put it into message queue
2) Process the "task– rule" pairs in parallel
3) Update information repository, and
4) Dispatch new task messages.

Step 1 is message queuing, step 2 is mediator coordination at message processing level, step 3 is considered as mediator update its state (current task message saved into task message repository), these are all inside mediation execution domain. However, wrappers can be either asynchronous or synchronous, and task message dispatching involves object serialization, marshalling (RMI), network latency (TCP/IP), and wrapper processing time. All these factors are irrelevant to information mediation and therefore should be excluded from counting into mediator performance evaluation. Therefore, the simulation

restricts to only the first three steps. Our simulations were performed on sun4u SPARC-SUNW, Ultra-30, under SunOS 5.6 operating system.

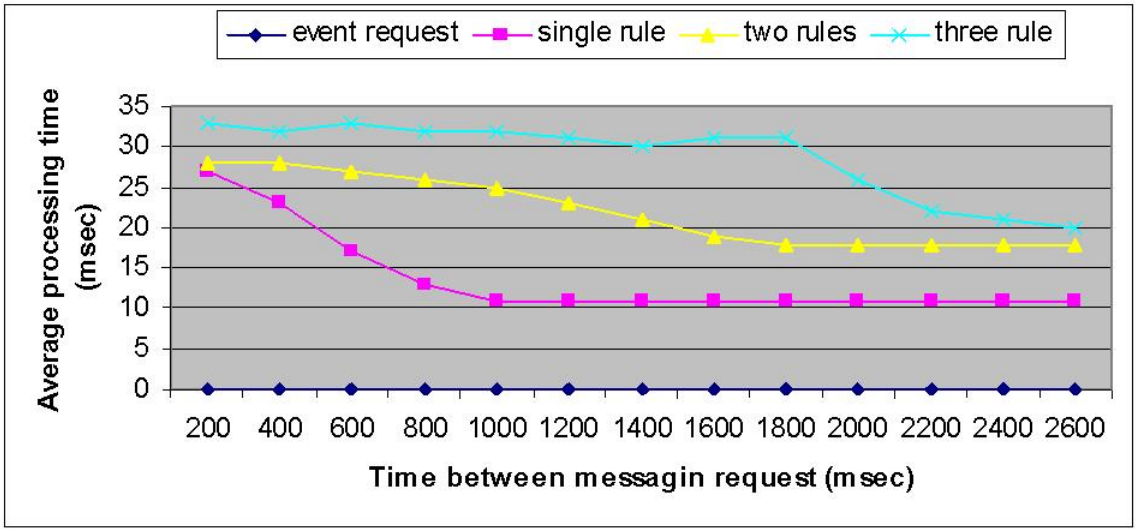

Figure 20: Average processing time (t_s) vs. time between message requests.

Figure 20 describes the experiment on the efficiency of mediation. A wrapper sends messages to the mediator periodically, with constant time t_p between subsequent messages. Timing starts when the message is received to the mediator. The mediator will then de-serialize the object received from RMI interface, re-organize the data into XML raw document, parse it (validate it), and save it in the message queue. Meanwhile that message will be popped out at the other end of the line and processed by the Rule Manager and the Session Manager. The Manager (in parallel) will find all relevant rules and sessions for that message event and process the message using the rule in the context of information repository, or update session state according to the triggering condition of rules. After processing, the message will be saved into repository and new messages based on business rules will be sending to output message queues, where a separate

thread takes care for dispatch them to registered wrappers. All activities except the outgoing message dispatching involve the mediation procedure and are recorded for mediator performance profiling. In the following simulation, costs in various aspects will be estimated.

For each run event-messages are sent between a constantly time period t_p. Suppose an event-message is received at t_r and that message is processed and the new messages (if there are) are placed in the message dispatching queues of each wrapper at t_f, then the mediation processing time $t_s = t_f - t_r$. The t_s vs. t_p metric evaluates the mediation efficiency regardless of wrapper performance and communication issues. Different t_p at multiple of 200 milliseconds are chosen for each run to evaluate the mediation performance. Figure 20 shows a negligible service time for pure event message (no rules are triggered), which means the message queuing, parsing, validating and information repository updating costs are trivial. The other three lines are the average processing time of using 1 rule, 2 rules, 3 rules on the incoming task data respectively. We understand using different rules would yield different average processing time, and we averaged our simulation result data using different rules. We based our simulation result on 100 test runs, and the standard deviation ranges from *2.1—3.7*. This simulation shows that the average processing time comes down when the time gap increases, and at certain time gap, the average processing time becomes constant. Babel mediator has a shorter time gap for that constant average processing time than [MYL96].

Figure 21: Average processing time (t_s) vs. Number of rules r_n.

Figure 21 shows the mediator's efficiency and scalability in terms of multiple rules processing. A moderate number of rules have been defined related with certain event. After the mediator receives this type of event-message, the Rule Manager will retrieve a number of rules r_n already registered for this event and starts processing those rules against the current task event in parallel. In this simulation, Babel is registered with 1 to 10 rules and average mediator processing time T_s (measured in the same way as first simulation) has been recorded respectively. Standard deviation ranges from *3.2 – 5.8*. As seen from the Figure 21, the mediator shows very moderate increase in T_s with the increase of the number of triggered rules. Processing of 10 rules takes 119.6 milliseconds while one rule takes 28.4 milliseconds. Therefore, the parallel rule-processing model scales well with large number of concurrently triggered rules.

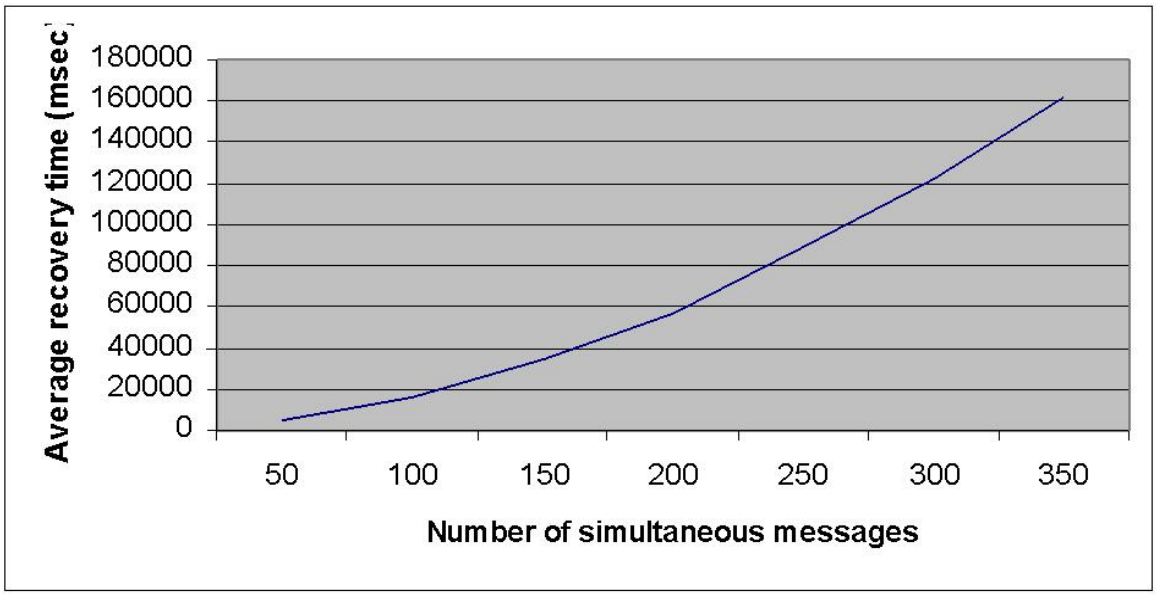

Figure 22: Average recovery time vs. Number of simultaneous messages.

Another important metric to evaluate is the mediator's capability of handling message flooding – large amount of messages sent to the mediator simultaneously. In order to test this capability, we start a lot of wrapper instances on the same machine as the mediator is running, and bombard the mediator with simultaneous messages. The time $t_{r,}$ as needed for the mediator to process various number n_r of simultaneous messages, is recorded. The simulation shows the mediator is able to recover within very short period of time (5.317 sec) for n_r =50 (standard deviation is from 0.49—0.83 sec). The mediation slows down as the number of message increase, and reach 161.909 sec at 350 simultaneous messages, and does not crash under a very heavy load. However, this performance impact is attributed to the constant information repository updating. When the information repository increases, it takes much longer to update the flat XML file based database.

Chapter 7 Conclusions

7.1 Feature of Babel Mediator

In Chapter 1, we listed the requirements of an information mediator for application integration, and Babel aims to fulfil these requirements throughout its design and implementation.

Transparency is provided to the designers of the coordination logic so that when designing rules, they do not need to think of the heterogeneity of the underlying applications.

Scalability is proven by the performance tests of Babel mediator runtime (Chapter 6) that Babel mediator scales well with multiple rules, has fast response to messages coming subsequently in short time gap, and recovers fast from large number of simultaneous messages.

Extendibility is achieved by using wrappers and separate design-time from runtime. Adding a new application to Babel mediation system is a three-step-procedure:

1. Make the corresponding wrapper for that application, either manually or automatically using Mathaino.
2. Update corresponding information of the wrapper at the design-time environment through an XML configuration file, and
3. Leave a proxy at the mediator's runtime environment so that mediator can invoke the wrapper at runtime.

Control logic of the interactions of the wrapper and the mediator is specified in Rules introduced in Chapter 3. Information on how this new application can provide functionality for other applications can be found in the documentation of the wrapper, and the rule designer should be familiar with all the wrappers involved in the rule.

It is worth noting that the complexity of the whole Babel mediation system does not simply goes linear to the number of wrappers it has. Suppose there are 1000 wrappers. Not every rule will deal with all of them. Usually a rule will only involve one or two wrappers, and the rule designer need only understand the behavior of these wrappers and forget all the rest. The rest wrappers will not have any impact on this rule. Even if there are a large number of rules available there, they are independent of each, and they may each cover their own wrappers. Control logic never intermingles with how many wrappers are there, or what other rules are there.

Wrappers can be located on any machine, and Babel mediator interacts with them in a distributed environment. This is necessary for scalability that some applications, like PINE email reader, are not necessarily located on the same machine of Babel mediator.

Semantics Richness is a major concern of Babel design-time environment. Rules can reflect on history, use the standard functions in XSLT, specify complex *Condition*s, and choose from a variety of *Action* templates.

Workflow: By having the Session as light-weighted workflow-based processing model, Babel mediator is capable of coordinating the enactment of multiple rules and further automating the business processes across multiple applications.

7.2 Contribution of Babel

Babel's contributions should be examined in two contexts: a component of the CelLEST, the legacy system re-engineering project in the context of which it was developed, and as a general mediator for application integration.

7.2.1 Contribution of Babel in the CelLEST project

Babel is developed in the context of the CelLEST project [STROULIA00]. The CelLEST project aims at legacy system re-engineering, including legacy system analysis, legacy system migration and automatic wrapper generating [KAP01], and legacy system integration. The Babel mediator aims to fulfill the last role. After the reverse- and re-engineering process of the Lendi and Mathaino systems [STROULIA00], the interaction between the legacy information system (LIS) and its users in the context of performing specific tasks of interest is fully analyzed and wrappers for the tasks in question can be automatically generated. These wrappers act as middle-ware between the user and the LIS, preserve the original functionality of the LIS while providing modern navigation user interface and Object-Oriented task-based API [KAP01].

Babel steps in after the wrappers for the LIS task of interest have being built. The major goal of Babel as a CelLEST component is to provide another level of aggressive maintenance to the overall legacy system re-engineering project. The wrappers produced by Mathaino are data agencies that are independent of each other; they simply provide access to existing services through XML-enabled task interfaces. Business rules and workflow processing based interaction among these wrappers could potentially be accomplished by writing ad hoc controlling programs, and those controlling programs are not only time consuming to make but also largely non-reusable [MSXML].

Babel was designed as a software integration solution based on a view of system integration as coordinated information production and consumption [ZS01]. The objective was to build a mediator one level above the LIS wrappers, to coordinate information flows and apply high-level business rules on those wrappers. Decision making that would normally be performed manually by users based on the data received from the LIS wrappers and interactions with the wrappers based on these decisions can now be performed automatically by the Babel system. This can be accomplished by specifying the business rules guiding the decision-making and the interaction process in the Babel design time environment and having the runtime mediator enact them in coordination with the LIS wrappers. Since the business rules are high level descriptive procedure automated by using easy-to-use Babel design time, the programming effort dedicated to improving the LIS system capability can greatly saved.

Business rules defined on the PINE legacy email reader [KAP01] and the Hollis library system [HOLLIS] have been implemented using the Babel system. Other examples are the integration of the PINE legacy email reader and the Amazon search wrapper (for details of the integration and other integration examples please refer to the Appendix). Through these examples we have collected sufficient evidences that Babel presents is an aggressive maintenance tool further enhancing the capabilities of the CelLEST toolkit in legacy system migration.

7.2.2 Contribution of Babel as general mediator approach

Babel's contribution as general information-mediation architecture is twofold:

As an integration framework, it enables the conversation between existing applications, the enactment of business rules controlling their underlying services, and the management of workflow sessions among these services. The Babel mediator uses XML for its data flow, and bases its business-rule processing on the XML processing standard language -- XSLT. It is a performance oriented, robust environment. It also provides a workflow-modeling language compatible with Workflow Management Model defined at WFMC.

Traditional mediation systems use mediator specific language for their data interoperation, and the data format definitions and semantics layout are non-interoperable with other mediation systems [MS XML]. XML comes to the rescue naturally and its makes mediator to application conversation easier. By using XSLT as embodiment for the Babel business control logic, the Babel mediator makes use of a ready-made mediator engine implementation, and lets users focus on data processing based on business logic, rather than business logic enactment. XSLT processing engines are widely available (XT, SAXON, Xalan, etc), and the functioning programming feature of XSLT enables this language for AI-featured tasks [KAY99].

For the time being, none of the existing workflow models provides an abstraction mechanism as powerful as event-based information mediator, though workflow models share a lot of commonality with Babel's information mediator approach.

The second important contribution of Babel is its visual design time environment for describing rules and workflow sessions. This visual design time environment provides a powerful abstraction in which to develop the rules and workflow sessions. It saves users

from intensive learning of the business rule definition or workflow process definition logic by providing the auto-generation of XSLT business rules based on that abstraction.

7.3 Concluding thoughts and future work

This thesis develops an information-mediation architecture for application integration. The solution is based on existing mediation architecture and state of art mediation technology (XML). The Babel mediator's attempt to separate design time from runtime helps to better deploy the mediator framework in a highly distributed and heterogeneous environment. We hope by this integrating wrapper construction and business rule logic definition and restricting them within scope of the design time environment, evolution of business rules and adding newer applications will have little impact on existing running runtime environment. Babel data modeling is based on a highly flexible object-oriented task-based language, helping Babel to get information from the applications as well as to drive them through their existing API. We choose the coordination language as incremental, template based knowledge description language (XSLT). Babel's *Event-Condition-Action* rule definition helps active information processing. The whole system implementation is based on public standard software packages (JAXP, SAXON), and various approaches (parallel computing, message queuing, asynchronous message processing) have been applied for enhancing the overall performance of the mediator. The system has various auxiliary tools (Rule Wizard, Session Builder, and registry tool) to increase the ease of use of the mediator. These tools allow users to define business rules at a high level of abstraction without having to learn any complex syntax.

The major shortcoming of the Babel mediator is its history information repository. The performance of the Babel mediator will be considerably impacted when the information repository becomes large. This problem is mainly implementation specific rather than intrinsic to the Babel mediator. Using flat XML file as the information repository adds cost to parsing XML file, update it, and serialize it to flat file. Each event processing will go over the whole procedure, because Babel's XSLT processing engine (SAXON) relies on persistent XML document. However, this incapability can be soon resolved as database applications supporting XSLT becomes available. OO-DBMS systems provide easier object oriented support in storing XML data, but their scalability and robustness prevent them from being successful information repository candidates. Relational DBMS systems have had long-running success in industry and scale well with large amount of data. Storing object oriented task-based events into relational schema (tables of rows and columns) has been studied widely and various approaches have been proposed to deal with saving objects in RDMBS [KM00] [CELKO]. Since all major RDBMS vendors (DB2, MS-SQL, Oracle, Sybase) have developed XML support for their products, XPATH and XSLT support will shortly become available as W3C standards for locating elements and transforming in XML documents.

Our work with Babel has raised several issues for further research. First, the design time environment needs to be extended to further assist rule definition. It is possible that users will have business rules that Rule Wizard could not model at this time. Providing a design tool for the automation of rules with richness and variety in semantic can be difficult and this requires better understanding or the domain in which the rules are to be applied. We might also try different query standards other than XSLT as supplement to

realize the business rules. The second issue is that the mediation system needs a better transactional support for Babel's workflow processing. That includes a better process definition model that creates workflow definition with more richness and better conforms to the WFMS model. Furthermore, we would like to enhance the workflow interoperability by having multiple mediators working as coordinated workflow engines. That workflow interoperability gain Babel mediation better load balance and fault tolerance in that one mediator's crash will not bring the whole mediation system down.

Information mediation in application integration is a fairly new research area. Through this project, we believe Babel's rule-oriented, XML-powered mediation approach provides a transparent, efficient, easy to use, and extensible architecture for distributed heterogeneous application integration.

References

[AALST96] W.M.P. van der Aalst, "Petri-net-based Workflow Management Software", *Proceedings of the NFS Workshop on Workflow and Process Automation in Information Systems*, Athens, Georgia, May 1996, pp. 114—118.

[AEAIILA] "Approaches to Enterprise Application Integration Involving Legacy Applications", *Introduction: Doing Business in the New Networked Economy*, available at http://www.attachmate.com/article/0, 1012, 3163_1, 00.html

[BAT86] C. Batini et al, "A comparative analysis of methodologies for database schema integration", *ACM Computing Surveys*, Vol. 18, No. 4, Dec.1986, pp.323-364.

[BER89] E. Bertino, "Integration of heterogeneous database applications though object-oriented interface", *Information systems*, 14(5):407-420, 1989.

[BLACK] Celcorp, *Blacksmith Apprentice: A Programmer Introduction*, Celcorp, 1998.

[BOT86] Y. Breitbart, P. Olson, and G. Thompson, "Database integration in a distributed heterogeneous database system", *Proceedings of the 2nd International Conference on Data Engineering*, Los Angeles, CA, Feb 1986, , pp. 301-310.

[CAREY94] M.J.Carey et al, "Towards heterogeneous multimedia information systems: The Garlic Approach", *Technical Report RJ 9911*, IBM Almaden Research Center 1994.

[CARR95] D. Carr, and R.J. Kizior, "The case for continued Cobol education", *IEEE Software*, Vol. 12, Issue 1, January 1995, pp.55-63.

[CELKO] J. Celko, "Joe Celko's SQL for Smarties : Advanced SQL Programming", *The Morgan Kaufmann Series in Data Management Systems*, Morgan Kaufmann Publishers, ISBN: 1558603239.

[CHAWA94] S. Chawathe, et al, “The TSIMMIS Project: Integration of Heterogeneous Information Sources”, *16th Meeting of the Information Processing Society of Japan*.

[CHI98] A. Chichoki, et al, “Workflow and Process Automation: Concepts and Technology”, Kluwer Academic Publishers, Boston, MA., 1998

[CUR96] Y. Curran, “Client/Server Development With SAP’s ABAP/4 Development Workbench 3.0”, Prentice Hall, July 1996. ISBN 0135253950.

[DATE98] CJ Date. “Don’t mix pointers and relations”, *3rd Annual Object/Relational Summit*, Washington dc, September 1998.

[DTD] Guide to the W3C XML Specification ("XMLspec") DTD, Version 2.1, *available from http://www.w3.org/XML/1998/06/xmlspec-report-v21.htm*

[ETZ93] O.Etzioni. “PARDES—a data-driven oriented active database model”, *ACM SIGMOD Record*, Mar 1993.

[GM01] A. Gal, J. Mylopoulos, “Supporting Distributed Autonomous Integration Services Using Coordination”, *International Journal of Cooperative Information Systems*, 2001.

[GOLDMAN99] R. Goldman, J. McHugh, and J. Widom. “From Semistructured Data to XML: Migrating the Lore Data Model and Query Language”, *Proceedings of the 2nd International Workshop on the Web and Databases (WebDB '99),* Philadelphia, Pennsylvania, June 1999.

[HAAS99] L. Haas, et al, “Transforming Heterogeneous Data with Database Middleware: Beyond Integration”, *IEEE Data Engineering Bulletin*, 1999.

[HEI95] S. Heiler, “Semantic Interoperability”, *ACM Computing Surveys*, June 1995.

[HOLLIS] Harvard digital library, *available from telnet://hollis.harvard.edu.*

[HUGHES89] J. Hughes, “Why Functional Programming Matters”, *Computer Journal* vol.32, April 1989.

[JAXP] JavaTM APIs for XML Processing (JAXP), *available from http://java.sun.com/xml/xml_jaxp.html*

[KAR95] K.Karlapalem, Q.Li and C.Shun. Hodfa, “An architecture framework for homogenizing heterogeneous legacy databases”, *SIGMOD RECORD*, Mar 1995.

[KAY99] M. Kay, “XSLT, Programmer’s Reference”, Wrox Press. ISBN 1-861003-12-9.

[KAP01] R. Kapoor, “Mathaino: Device Retargetable User Interface Migration Using XML”, Master’s thesis, University of Alberta, 2001.

[KM00] M. Klettke, H. Meyer, "XML and Object-Relational Database Systems - Enhancing Structural Mappings Based on Statistics". *International Workshop on the Web and Databases (WebDB),* Dallas, TX, May 2000.

[LAN98] T. Langer, “MeBro - A Framework for Metadata-Based Information Brokerage”, *I'MEDIAT'98, Electronic Proceedings of the First International Workshop on Practical Information Mediation and Brokering, and the Commerce of Information on the Internet*, Tokyo, Japan, September 14, 1998.

[LEVY96] A.Levy, A. Rajaraman, and J.Ordile, “Querying heterogeneous information sources using source description”, *Proceedings of the International Conference on VLDB*, Bombay, India, 1996, pp. 251—262.

[MAPPING] “Mapping objects to relational databases”, IBM developerWorks: Components, *available from http:www.106.ibm.com/developerworks/library/mapping-tordb/*

[MIX] R. Cover, “MIX - Mediation of Information Using XML”, August 05, 1999.

[MOLINA94] H. Garcia-Molina, et al, “The TSIMMIS Approach to Mediation: Data Models and Languages”, *Technical Report*, Stanford University, 1994.

[MSXML] “XML Enables the Next Stage of the Digital Revolution by Eliminating Barriers Between Systems and Devices”, Microsoft PressPass, *available from http://www.microsoft.com/presspass/features/2000/jul00/07-10xml.asp*

[MYL90] J. Mylopoulos et al, “Telos: Representing Knowledge About Information Systems”, *Transactions on Information Systems* 8, 4 (1990), pp. 325--362.

[MYL96] J. Mylopoulos et al, “A Generic Integration Architecture for Cooperative Information Systems”, *Proceedings of the 1st IFCIS Intl. Conference on Cooperative Information Systems*, Brussels, Belgium, June 1996.

[NAV89] S.B. Natathe et al, “A federated architecture for heterogeneous information systems”, *Proceedings of Workshop on Heterogeneous Databases*, Northwestern University, Evanston, IL, December 1989.

[PHJ95] Y. Papakonstantinou, H. Garcia-Molina, J. Widom, “Object Exchange Across Heterogeneous Information Sources”, *Proceedings of ICDE*, 1995.

[RG99] R. Ramakrishnan and J. Gehrke, “Database Management Systems”, 2nd Edition, McGraw-Hill, August 1999, ISBN: 0-07-232206-3.

[SAXON] XSLT processor implementation, *available from http://users.iclway.co.uk/mhkay/saxon/*

[Shaw96] M. Shaw and D. Garlan, “Software Architecture, Perscpectivevs on an Emerging dicipling”, Prentice Hall, April 1996.

[SHE96] A. Sheth et al, “Reports from the NSF Workflow and Process Automation in Information Systems”, *Technical Report*, University of Georgia, UGA-CS-TR-96-003.

[SS00] Q. Situ and E. Stroulia, “Task-structure Based Mediation: The Travel-Planning Assistant Example”, *Proceedings of the 13th Canadian Conference on Artificial Intelligence (AI'2000),* 14-17 May, 2000, Montreal, Quebec, Canada.

[STROULIA00] E. Stroulia, et al, "Legacy Systems Migration in CelLEST”, *Proceedings of the 22nd International Conference on Software Engineering*, Limerick, Ireland (June 4-11, 2000).

[SUND99] T. Sundsted, “XML and JavaTM Technology Tackle Enterprise Application Integration”, Reprinted from Java World, June 1999, *available from http://developer.java.sun.com/developer/technicalArticles/Networking/XMLAndJava/*

[UMA99] A. Uma, "Introduction to Client/Server Fundamentals", *Object-Oriented Client/Server Internet Environments-The IT Infrastructure*, February 8, 1999, *available from http://www.networkcomputing.com/netdesign/1005part1.html*

[WA99] B. Wilkinson and M. Allen, "Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers", Prentice Hall, 1999.

[WFMC] "The Workflow Reference Model", Workflow Management Coalition, Document Number TC00-1003.

[WW97] D. Wodtke and G. Weikum, "A formal foundation for distributed workflow execution based on state charts", *6th International Conference on Database Theory (ICDT 97)*, 1997.

[WIE92] G. Wiederhold, "Mediators in the Architecture of Future Information Systems", *IEEE Computer*, March, 1992.

[WIE95a] G. Wiederhold, "Mediation in Information Systems", *ACM Computing Surveys*, Vol.27, No 2, June 1995.

[WIE95b] "Mediation and Software Maintenance", *Stanford CSD Technical Note*, STAN-CS-TN-95-26.

[WIE97] G. Wiederhold, "Value-added Mediation in Large-Scale Information Systems, Wiederhold", Gio in Robert Meersman and Leo Mark(ed): *Database Application Semantics,* Chapman and Hall, 1997, pp. 34-56.

[WIE98] G. Wiederhold, "Weaving Data into information", *DBLP*, Sept 1998.

[WIE00] G. Wiederhold, "Mediation Technology", *Technical Report*, Stanford University, 19 June 2000.

[XML] W3C, "Extensible Markup Language (XML) 1.0 (Second Edition)", W3C Recommendation, *available from: http://www.w3.org/TR/2000/REC-xml-20001006, October 2000. www.w3.org/XML/*

[XPATH] XML Path Language (XPath) Version 1.0, W3C Recommendation 16 November 1999, available from www.w3.org/TR/xpath

[XSCHEMA] XML Schema, W3C Recommendation, *available from www.w3.org/XML/Schema*

[XSLT] XSL Transformations (XSLT) Version 1.0, W3C Recommendation 16 November 1999, *available from www.w3.org/TR/xslt*

[ZS01] H. Zhang, and E. Stroulia, "Babel: Application Integration through XML specification of Rules", *23rd International Conference on Software Engineering (ICSE 2001)*, Toronto, Canada.

Appendix

Data Modeling for *Task* (DTD source code)

```
<!-- babel task dtd -->

<!-- root element -->
<!ELEMENT task (description, mediator-specific-information, inputfieldalias, outputfieldalias)>

<!ATTLIST task
    task_type CDATA #IMPLIED
>


<!-- this is the mediator-specific-information element -->
<!ELEMENT mediator-specific-information (project, bkrmiserverurl, bkrmiserverport, reconnectflag)>

<!ELEMENT project (#PCDATA)>       <!-- Wrapper   -->
<!ELEMENT bkrmiserverurl (#PCDATA)>    <!-- RMI server -->
<!ELEMENT bkrmiserverport (#PCDATA)>    <!-- RMI port  -->
<!ELEMENT reconnectflag  (#PCDATA)>     <!-- false, true -->
<!-- end of mediator-specific-information and its sub-element -->


<!ELEMENT inputfieldalias (info*)>
<!ELEMENT outputfieldalias (info*)>

<!-- this is the info element -->
<!ELEMENT info (name, description, xpath-to-attribute, value)>
<!ATTLIST info
    type CDATA #REQUIRED
>
<!ELEMENT name (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT xpath-to-attribute (#PCDATA)> <!-- needed by mathaino -->
<!ELEMENT value (#PCDATA)>
<!-- end of info and its sub-element -->
```

Guidelines for Email Book Query Demonstration

Rule One (This rule direct Email Search information to Hollis BlackSmith Wrapper)

1. set Incoming task type to "email ... email book search",
2. set input/info/type "Book"
3. set NewTask type to "hollis booksearch"
4. Drag the Incoming task/input/info to NewTask/input/info
5. Save it as "rule1.xml"

Rule Two (This rule get inquiry result from Hollis and find all previous email enquiries on that book, and send back results)

1. new a rule
2. reset the whole IncomingTask tree
3. set the incoming task type "hollis.searchTop5 books"
4. set the input/info/type to "Book", make the PCData as variable

5. set the Objective task type "email library search...."
6. set constraint ObjectiveTask/input/info[type=Book]/PCData == Variable0 (must "Book", select the variable!)
7. set the Objective!

8. set the New Task tasktype to email.answeer
9. clone input/info
10. set Objective Task's input/info[Username] info
11. set Objective Task's input/info[EmailAddress] info

12. Drag the Book
13. Drag the Email
14. Drag the User
15 Drag the IncomingTask/output/info (don't set constraint!!)

16. save it as rule2.xml

Rule 1 source code

```
<?xml version="1.0"?>
<!DOCTYPE RULE SYSTEM "rule.dtd">
<rule task_type = "Email Receiver"
   rule_id = "rule2001072714465449">
  <desc></desc>
  <xslrule><![CDATA[
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:saxon="http://icl.com/saxon" xmlns:ua="http://www.cs.ualberta.ca/Functions"
xmlns:url="http://www.cs.ualberta.ca/hxzhang.ca.ualberta.cs.babel.extension.URLElementFact
ory" version="1.0" saxon:trace="no" extension-element-prefixes="saxon">

  <xsl:output method="xml" omit-xml-declaration="no" indent="yes" encoding="iso-8859-1"
doctype-system="task.dtd"/>

  <xsl:strip-space elements="description * "/>

  <xsl:variable name="filesep" select="system-property('file.separator')"/>

  <xsl:variable name="event" select="/task"/>
  <xsl:variable name="history">database<xsl:value-of select="$filesep"/>data<xsl:value-of
select="$filesep"/>tasks.xml</xsl:variable>

  <xsl:template match="task">

   <xsl:variable name="condition">true</xsl:variable>
   <xsl:if test="$condition != 'true'">
    <xsl:message terminate="yes">Rule Condition not passed.</xsl:message>
   </xsl:if>
   <xsl:variable name="index">

    <xsl:number/>

   </xsl:variable>

   <xsl:document href="output{$filesep}hollis.harvard.edu{$filesep}{$index}.xml"
method="xml" indent="yes">
    <task task_type="Hollis Searcher">
     <inputfieldalias>
      <info type="Book">
       <value>
        <xsl:value-of select="$event/inputfieldalias/info[@type='Book']/value/text()"/>
       </value>
      </info>
     </inputfieldalias>
     <outputfieldalias>
```

```
        <info type="BookInfo"/>
      </outputfieldalias>
     </task>
   </xsl:document>
  </xsl:template>

</xsl:stylesheet>]]>
</xslrule>
</rule>
```

Rule 2 source code

```
<?xml version="1.0"?>
<!DOCTYPE RULE SYSTEM "rule.dtd">
<rule task_type = "Hollis Searcher"
   rule_id = "rule2001072714481891">
  <desc></desc>
  <xslrule><![CDATA[
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:saxon="http://icl.com/saxon" xmlns:ua="http://www.cs.ualberta.ca/Functions"
xmlns:url="http://www.cs.ualberta.ca/hxzhang.ca.ualberta.cs.babel.extension.URLElementFact
ory" version="1.0" saxon:trace="no" extension-element-prefixes="saxon">

  <xsl:output method="xml" omit-xml-declaration="no" indent="yes" encoding="iso-8859-1"
doctype-system="task.dtd"/>

  <xsl:strip-space elements="description * "/>

  <xsl:variable name="filesep" select="system-property('file.separator')"/>

  <xsl:variable name="event" select="/task"/>
  <xsl:variable name="history">database<xsl:value-of select="$filesep"/>data<xsl:value-of
select="$filesep"/>tasks.xml</xsl:variable>

  <xsl:template match="task">

   <xsl:variable name="condition">true</xsl:variable>
   <xsl:if test="$condition != 'true'">
     <xsl:message terminate="yes">Rule Condition not passed.</xsl:message>
   </xsl:if>
   <xsl:for-each select="document($history)/tasks/task[@task_type='Email
Receiver'][inputfieldalias/info[@type='Book']/value/text()=$event/inputfieldalias/info[@type='
Book']/value/text()]">

     <xsl:variable name="index">

      <xsl:number/>

     </xsl:variable>

     <xsl:document href="output{$filesep}email.ualberta.ca{$filesep}{$index}.xml"
method="xml" indent="yes">
      <task task_type="Email Sender">
```

```
        <inputfieldalias>
         <info type="EmailAddress">
           <value>
            <xsl:value-of select="inputfieldalias/info[@type='EmailAddress']/value/text()"/>
           </value>
         </info>
         <info type="UserName">
           <value>
            <xsl:value-of select="inputfieldalias/info[@type='UserName']/value/text()"/>
           </value>
         </info>
         <info type="Book">
           <value>
            <xsl:value-of
select="inputfieldalias/info[@type='Book'][value/text()=$event/inputfieldalias/info[@type='Bo
ok']/value/text()]/value/text()"/>
           </value>
         </info>
        </inputfieldalias>
        <outputfieldalias>
         <info type="Content">
           <value>
            <xsl:value-of select="$event/outputfieldalias/info[@type='BookInfo']/value/text()"/>
           </value>
         </info>
        </outputfieldalias>
      </task>
     </xsl:document>
   </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>]]>
</xslrule>
</rule>
```

Session Example

```
<?xml version="1.0"?>
<!DOCTYPE SESSION SYSTEM "session.dtd">
<session transitions = "3">
 <transition from = "0" to = "1">
<rule    rule_id="rule2001022314133704"    task_type="email.ualberta.ca+EmailLibrarySearch"
wrapper="email">
  <xslrule><![CDATA[
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 saxon:trace="no" xmlns:saxon="http://icl.com/saxon"
 extension-element-prefixes="saxon"
 xmlns:ua="http://www.cs.ualberta.ca/Functions"

xmlns:url="http://www.cs.ualberta.ca/hxzhang.ca.ualberta.cs.babel.extension.URLElementFact
ory">
<xsl:output method="xml" omit-xml-declaration="no" indent="yes"
 encoding="iso-8859-1"
 doctype-system="task.dtd"
/>
<xsl:strip-space elements="description * "/>
<xsl:variable name="filesep" select="system-property('file.separator')" />
<xsl:variable name = "history">data<xsl:value-of select ="$filesep"/>tasks.xml</xsl:variable>
<xsl:template match="/">
 <xsl:variable name="variable2" select="task/inputfieldalias/info[@type='BookTitle']/text()">
</xsl:variable>
 <xsl:variable name="variable3" select="task/inputfieldalias/info[@type='EmailAddress']">
</xsl:variable>
<xsl:variable name="trigger">1</xsl:variable>
<xsl:if test="$trigger!=1">
<xsl:message terminate="yes" >
Rule Not Triggered
</xsl:message>
</xsl:if>
<xsl:variable name = "index">
<xsl:number/>
</xsl:variable>
<saxon:output          file="output{$filesep}email{$filesep}{$index}.xml"          method="xml"
indent="yes">
<xsl:element name="task">
<xsl:element name="description">
```

```
</xsl:element>
<xsl:element name="mediator-specific-information">

</xsl:element>
<xsl:element name="inputfieldalias">
<xsl:copy-of select ="$variable3"/>

</xsl:element>
<xsl:element name="outputfieldalias">
<xsl:element name="info">
<xsl:attribute name = "type"><xsl:value-of select="'amazon lookup'"/></xsl:attribute>
<url:lookup  site="www.amazon.com"  proxy="/exec/obidos/search-handle-form/105-1975066-
9257502" xsl:extension-element-prefixes="url">$variable2</url:lookup>
<xsl:element name="description">

</xsl:element>
<xsl:element name="xpath-to-attribute">

</xsl:element>

</xsl:element>

</xsl:element>

</xsl:element>

</saxon:output>
</xsl:template>
</xsl:stylesheet>
]]>
</xslrule>
</rule></transition>
<transition from = "0" to = "-1">
<rule                    task_type="hollis.harvard.edu+booksearch"                    wrapper="email"
rule_id="rule2001022314133745" >
  <xslrule><![CDATA[
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 saxon:trace="no" xmlns:saxon="http://icl.com/saxon"
 extension-element-prefixes="saxon"
 xmlns:ua="http://www.cs.ualberta.ca/Functions">
<xsl:output method="xml" omit-xml-declaration="no" indent="yes"
 encoding="iso-8859-1"
 doctype-system="task.dtd"
/>
```

```
<xsl:strip-space elements="description * "/>
<xsl:variable name="filesep" select="system-property('file.separator')" />
<xsl:template match="/">
 <xsl:variable name="variable0" select="task/inputfieldalias/info[@type='UserName']">
</xsl:variable>
 <xsl:variable name="variable1" select="task/inputfieldalias/info[@type='EmailAddress']">
</xsl:variable>
 <xsl:variable name="variable2" select="task/inputfieldalias/info[@type='BookTitle']">
</xsl:variable>
<xsl:variable name="trigger">
1
</xsl:variable>
<xsl:if test="$trigger!=1">
<xsl:message terminate="yes" >
Rule Not Triggered
</xsl:message>
</xsl:if>
<xsl:variable name = "index">
<xsl:number/>
</xsl:variable>
<saxon:output file="output{$filesep}email{$filesep}{$index}.xml" method="xml"
indent="yes">
<xsl:element name="task">
<xsl:attribute name = "resource"><xsl:value-of select="'babel'"/></xsl:attribute>
<xsl:attribute name = "task_type"><xsl:value-of select="'email replay'"/></xsl:attribute>
<xsl:element name="description">

</xsl:element>
<xsl:element name="mediator-specific-information">

</xsl:element>
<xsl:element name="inputfieldalias">
<xsl:copy-of select ="$variable0"/>
<xsl:copy-of select ="$variable1"/>
<xsl:copy-of select ="$variable2"/>

</xsl:element>
<xsl:element name="outputfieldalias">
<xsl:element name="info">
<xsl:element name="description">

</xsl:element>
<xsl:element name="xpath-to-attribute">

</xsl:element>
```

```
</xsl:element>

</xsl:element>

</xsl:element>

</saxon:output>
</xsl:template>
</xsl:stylesheet>
]]>
</xslrule>
</rule></transition>
<transition from = "1" to = "-1">
<rule task_type="hollis.harvard.edu+booksearch" wrapper="email"
rule_id="rule2001022314133745" >
  <xslrule><![CDATA[
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 saxon:trace="no" xmlns:saxon="http://icl.com/saxon"
 extension-element-prefixes="saxon"
 xmlns:ua="http://www.cs.ualberta.ca/Functions">
<xsl:output method="xml" omit-xml-declaration="no" indent="yes"
 encoding="iso-8859-1"
 doctype-system="task.dtd"
/>
<xsl:strip-space elements="description * "/>
<xsl:variable name="filesep" select="system-property('file.separator')" />
<xsl:template match="/">
 <xsl:variable name="variable0" select="task/inputfieldalias/info[@type='UserName']">
</xsl:variable>
 <xsl:variable name="variable1" select="task/inputfieldalias/info[@type='EmailAddress']">
</xsl:variable>
 <xsl:variable name="variable2" select="task/inputfieldalias/info[@type='BookTitle']">
</xsl:variable>
<xsl:variable name="trigger">
1
</xsl:variable>
<xsl:if test="$trigger!=1">
<xsl:message terminate="yes" >
Rule Not Triggered
</xsl:message>
</xsl:if>
<xsl:variable name = "index">
<xsl:number/>
```

```
</xsl:variable>
<saxon:output file="output{$filesep}email{$filesep}{$index}.xml" method="xml" indent="yes">
<xsl:element name="task">
<xsl:attribute name = "resource"><xsl:value-of select="'babel'"/></xsl:attribute>
<xsl:attribute name = "task_type"><xsl:value-of select="'email replay'"/></xsl:attribute>
<xsl:element name="description">

</xsl:element>
<xsl:element name="mediator-specific-information">

</xsl:element>
<xsl:element name="inputfieldalias">
<xsl:copy-of select ="$variable0"/>
<xsl:copy-of select ="$variable1"/>
<xsl:copy-of select ="$variable2"/>

</xsl:element>
<xsl:element name="outputfieldalias">
<xsl:element name="info">
<xsl:element name="description">

</xsl:element>
<xsl:element name="xpath-to-attribute">

</xsl:element>

</xsl:element>

</xsl:element>

</xsl:element>

</saxon:output>
</xsl:template>
</xsl:stylesheet>
]]>
</xslrule>
</rule></transition>
</session>
```

Email Query Task

```
<?xml version="1.0" encoding="utf-8"?>
<task resource="email.ualberta.ca" task_type="Email Receiver">
<description>Emailing a library search task</description>
<inputfieldalias>
<info type="UserName">
<value>hxzhang</value>
<name/>
<description>user's name</description>
<xpath-to-attribute/>
</info>

<info type="EmailAddress">
<value>hxzhang@cs.ualberta.ca</value>
<name/>
<description>user email</description>
<xpath-to-attribute/>
</info>

<info type="Book">
<value>TI programming</value>
<name>SearchText</name>
<description>the keyword of the book</description>
<xpath-to-attribute>Book.SearchText</xpath-to-attribute>
</info>

</inputfieldalias>
<outputfieldalias/>
</task>
```

Hollis Book Search Request Task

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE TASK SYSTEM "task.dtd">
<task task_type="hollis.harvard.edu+booksearch">
        <description/>

        <mediator-specific-information>
                <project>HOLLIS</project>
                <bkrmiserverurl>cambria.cs.ualberta.ca</bkrmiserverurl>
                <bkrmiserverport>8200</bkrmiserverport>
                <babelrmiserverurl>carvel.cs.ualberta.ca</babelrmiserverurl>
                <babelrmiserverport>1099</babelrmiserverport>
                <reconnectflag>false</reconnectflag>
        </mediator-specific-information>

        <inputfieldalias>
                <info type ='Book'>
                        <name>SearchText</name>
                        <description/>
              <xpath-to-attribute>Book.SearchText</xpath-to-attribute>
                        <value>TI programming</value>
                </info>
        </inputfieldalias>

</task>
```

Hollis Book Search Result Task

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE TASK SYSTEM "task.dtd">
<task task_type="Library Search">
   <inputfieldalias>
                <info type="Book">
                        <name>SearchText</name>

                        <description/>
                        <xpath-to-attribute>Book.SearchText</xpath-to-attribute>
                        <value>TI programming</value>
                </info>
        </inputfieldalias>
        <outputfieldalias>
                <info type="Book">
                        <name>BookInfo</name>

                        <description/>
                        <xpath-to-attribute>Book.BookInfo</xpath-to-attribute>
                        <value>    AUTHOR: Stanford Research Institute.                TITLE:
Programming; a context for decision-making in government and          industry, a background
paper, prepared in: management and          social systems area.                PUB. INFO:
Menlo Park, Calif., 1965.                DESCRIPTION: iv, 25 l. illus. 28 cm.
SUBJECTS: *S1 System analysis.                *S2 Decision making.
*S3 Industrial management.                LOCATION: Baker
Business:                HGCO                S782
</value>
                </info>
                <info type="Book">
                        <name>BookTitle</name>

                        <description/>
                        <xpath-to-attribute>Book.BookTitle</xpath-to-attribute>
                        <value>Programming; a context for decision-making in government and
        industry, a background paper, prepared in: management and
        social systems area.
</value>
                </info>
                <info type="Book">
                        <name>PubInfo</name>

                        <description/>
```

<xpath-to-attribute>Book.PubInfo</xpath-to-attribute>
<value>Menlo Park, Calif., 1965.
</value>
</info>
</outputfieldalias>
</task>

Email Answer Task

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE TASK SYSTEM "task.dtd">
<task task_type="babel.ualberta.ca+emailanswer">
   <inputfieldalias>
                <info type = 'UserName'>
                <description>the library user emailing the request</description>
                <name/>
                 <xpath-to-attribute></xpath-to-attribute>
                 <value>Huaxin</value>
                </info>
           <info type = 'EmailAddress'>
                  <description>user email Address</description>
                  <name/>
                   <xpath-to-attribute></xpath-to-attribute>
                   <value>hxzhang@cs.ualberta.ca</value>
                </info>

        </inputfieldalias>

        <outputfieldalias>
                <info type="Book">
                        <name>BookInfo</name>

                        <description/>
                        <xpath-to-attribute>Book.BookInfo</xpath-to-attribute>
                        <value>    AUTHOR: Stanford Research Institute.                TITLE:
Programming; a context for decision-making in government and          industry, a background
paper, prepared in: management and          social systems area.                PUB. INFO:
Menlo Park, Calif., 1965.                DESCRIPTION: iv, 25 l. illus. 28 cm.
SUBJECTS: *S1 System analysis.                *S2 Decision making.
*S3 Industrial management.                LOCATION: Baker
Business:                HGCO                S782
</value>
                </info>
                <info type="Book">
                        <name>BookTitle</name>

                        <description/>
                        <xpath-to-attribute>Book.BookTitle</xpath-to-attribute>
                        <value>Programming; a context for decision-making in government and
```

```
        industry, a background paper, prepared in: management and
        social systems area.
</value>
                    </info>
                    <info type="Book">
                            <name>PubInfo</name>

                            <description/>
                            <xpath-to-attribute>Book.PubInfo</xpath-to-attribute>
                            <value>Menlo Park, Calif., 1965.
</value>
                    </info>
            </outputfieldalias>
</task>
```

Babel Runtime /peripheral initialization script

```
@echo off
REM IN ORDER TO RUN BABEL_DEMO:
REM 1. INSTALL JDK1.1.8 & JDK1.1.3
REM 2. INSTALL BLACK SMITH
REM 3. PROVIDE "javai.dll, java_g.dll" etc. UNDER THE SAME DIRECTORY FOR "BKJAVA.CLASS"
REM 4 CHANGE EVERY REFERENCE FROM "CARVEL" TO "LOCAL MACHINE"
REM 5. DO NOT CHANGE PASSWORD FOR EMAIL ACCOUNT
REM 6. Generate all new STUBs!



REM ENVIRONMENT
set SCREEN_WIDTH=480
set SCREEN_HEIGHT=160

set HOME=h:\thesis\icsedemo

rem set JDK118PATH=%HOME%\jdk\jdk1.1.8
rem set JDK13PATH=%HOME%\jdk\jdk1.3
set JDK118PATH=d:\jdk1.1.8
set JDK13PATH=d:\jdk1.3.1

set CODE_HOME=%HOME%\code

set LIB=%CODE_HOME%\lib
set BKDRIVER=%CODE_HOME%\bksmith
set BKPROXY_HOME=%CODE_HOME%\kapoor\ca\ualberta\cs\bkproxy
set BABEL_HOME=%CODE_HOME%\hxzhang\ca\ualberta\cs\babel
set EMAIL_SMTP_HOME=%CODE_HOME%\hxzhang\ca\ualberta\cs\babel\wrapper
set EMAIL_POP_HOME=%CODE_HOME%\hxzhang\ca\ualberta\cs\pop3
set TEST_HOME=%CODE_HOME%\hxzhang\ca\ualberta\cs\babel\client

set
CLASSPATH=.;%CODE_HOME%;%BKDRIVER%;%LIB%\dtdparser.jar;%LIB%\httpclient.
jar;%LIB%\xerces.jar;%LIB%\saxon.jar

echo Run the BlackSmith Proxy Server
```

```
start "BlackSmith TELNET Proxy" /POS=10, 400, %SCREEN_WIDTH%, %SCREEN_HEIGHT% %JDK118PATH%\bin\java.exe kapoor.ca.ualberta.cs.bkproxy.BKRmiServer debug


echo Run the BlackSmith Wrapper Server, don't start till BlackSmith Server Started!!
pause
start "BlackSmith RMI Wrapper Server" /POS=10, 200, %SCREEN_WIDTH%, %SCREEN_HEIGHT% %JDK13PATH%\bin\java hxzhang.ca.ualberta.cs.babel.wrapper.BKWrapperServer


REM clean former stuff
rem cd %BABEL_HOME%\output\email.ualberta.ca
rem del *.xml
rem cd %BABEL_HOME%\output\hollis.harvard.edu
rem del *.xml


echo BUILDING AND START EMAIL SMTP WRAPPER
rem cd %EMAIL_SMTP_HOME%
rem del *.class *~
rem javac *.java
rem cd %HOME%
rem rmic hxzhang.ca.ualberta.cs.babel.wrapper.EmailWrapperServer
rem cd %EMAIL_SMTP_HOME%
start "Email SMTP Wrapper" /POS=0, 600, %SCREEN_WIDTH%, %SCREEN_HEIGHT% %JDK13PATH%\bin\java.exe hxzhang.ca.ualberta.cs.babel.wrapper.EmailWrapperServer


echo BUIDLING BABEL AND START BABEL
rem cd %BABEL_HOME%
rem del *.class *~
rem javac *.java
rem cd %HOME%
rem rmic hxzhang.ca.ualberta.cs.babel.BabelRMIImpl
rem cd %BABEL_HOME%
rem copy data\tasks.xml.back data\tasks.xml
start "Babel Mediator" /POS=500, 300, %SCREEN_WIDTH%, %SCREEN_HEIGHT% %JDK13PATH%\bin\java.exe% hxzhang.ca.ualberta.cs.babel.Babel
```

```
REM Now we have 5 ACTIVE COMMAND WINDOWS

REM VXG make two rules
REM DEMO PROCESURE 1:
REM 1. (incoming)set task type to "hollis...Select top 5 books", set info type "book", make the variable
REM 2. (objective)set the task type "email....", set constraint on PCDATA, set the objective!
REM 3. (objective) clone
REM 4. set the New Task tasktype to email.answeer
REM 5. DnD (!! the output should be full to make email wrapper work)
REM 6. save it as rule1.xml

REM DEMO PROCESURE 2:
REM BETTER START ALL OVER TO AVOID EXTRA VARIABLES
REM 1. incoming task type : email...library search
REM 2. set all those new task values
REM 3 set new task type to hollis..search
REM 4. d&d
REM 5. save it as rule2.xml



echo Register the Business Rules
pause
cd %TEST_HOME%
java hxzhang.ca.ualberta.cs.babel.client.BabelClient rule1.xml
java hxzhang.ca.ualberta.cs.babel.client.BabelClient rule2.xml



echo Run Email POP3 Wrapper Server
rem cd %EMAIL_POP_HOME%
rem del *.class *~
rem %JDK13PATH%\bin\javac.exe *.java
REM REMOVE HISTORY
del %EMAIL_POP_HOME%\uidl.log
start "Email POP3 Wrapper" /POS=10, 10, %SCREEN_WIDTH%, %SCREEN_HEIGHT%
%JDK13PATH%\bin\java.exe hxzhang.ca.ualberta.cs.pop3.POP3Wrapper
```