Babel: An XML-Based Application Integration Framework

Huaxin Zhang and Eleni Stroulia

Computing Science Department,
University of Alberta,
Edmonton, AB, T6G 2E8, Canada
{hxzhang,stroulia}@cs.ualberta.ca

Abstract. One of the major problems in integrating independently developed applications is the divergence between the data and control-of-processing models assumed by these applications. Research on database integration has focused on establishing and maintaining a canonical schema on top of the schemas of the underlying databases. At the same time, web-accessible software systems have been adopting a multi-layer architecture style, with databases in the lowest tier, business logic in the middle tier and user interfaces in the top-most tier. However, as the time-to-market window shrinks, new software is presented with the challenge of reusing and integrating the functionalities of existing whole applications, instead of simply their database back-ends. The Babel framework provides support for specifying existing applications in terms of the functionalities they deliver and the data they manipulate. In addition, it supports the specification of the "logic" defining how these functionalities should be integrated. Based on these specifications, Babel produces a run-time mediator that monitors the behavior of the underlying applications, evaluates the defined logic on the global state of the integrated system, and generates triggers for new functionalities to be accomplished according to these rules.

1 Motivation and Background

As the number of alternative technologies underlying software development increases, the need to develop methods and tools to support the integration of heterogeneous software assets becomes more pressing.

Research efforts to that end aimed originally at integrating heterogeneous databases. As the object-oriented paradigm usurped the more traditional procedural approach to software design and development, object-oriented databases were proposed as an alternative to relational databases. As a result, a new area of research was born, focusing on methods for developing object-oriented views of relational data [20]. However, even when developed with the same design paradigm for the same application domain, different databases differ in terms of the syntax they adopt for the representation of (potentially the same) data and the semantic [7] interpretation of the data by the applications supported by these databases. Thus, a substantial research effort has been invested in integrating the schemas of heterogeneous databases or constructing unified views of their data [2,3,4] and creating new query languages to query distributed heterogeneous databases [10].

A. Banks Pidduck et al. (Eds.): CAiSE 2002, LNCS 2348, pp. 280-295, 2002.

More recently, the advent of XML has inspired a whole new area of database research aiming at developing database support for managing XML data [6] and integrating XML-based databases [11]. At the same time, XML provides a flexible syntax for representing semi-structured data and can be used as the medium to represent information extracted from all types of different sources, such as HTML documents [24], and textual information repositories [25]. The scope of the integration problem has thus expanded from database to information integration [13].

The next logical step in this evolution of the integration research agenda is application integration; not only should data, structured in databases or unstructured in informal repositories, be integrated, but also complete applications. The problem of application integration however raises a new important question: how to coordinate the behavior of independent applications that have been designed with different assumptions regarding their context of operation and different expectations on how they may be called from their environment [22,23].

Several approaches have been developed to address this problem, making strong assumptions regarding the underlying integration infrastructure as well as the reengineering of the applications to be integrated. Object-oriented integration frameworks, such as CORBA for example, require that the applications to be integrated are developed in the object-oriented style and that the interfaces of their classes be represented in a proprietary specification language in order to be advertised and brokered by the central mediator (Object Request Broker). Such approaches are therefore applicable to a specific type of applications and require specialized technical skills and substantial development effort.

This is exactly the motivation behind more lightweight, event-driven approaches, such as XWrap/CQ [26] and CoopWare [5,12]. CQ aims at developing a framework for integrating applications, by monitoring the updates to the information they contain in distributed open environments such as the web for example. Furthermore it combines traditional database-style pull-based query-answering services with push-enabled event-driven update monitoring services. CoopWare is an event-based integration architecture based on active-database technology; update events in the underlying databases trigger rules in a central coordinator module that can, in turn, invoke SQL-like transactions in the underlying databases.

In our work, we have also been investigating methods for lightweight integration of distributed, heterogeneous, independently developed applications. Babel [21], like CQ, assumes XML as the syntax for representing the data extracted from the underlying applications. Like CQ and CoopWare, it assumes that the coordination behavior of the mediator is triggered by events generated by the underlying resources. Unlike CQ and CoopWare, Babel does not assume SQL-like queries integrating the underlying resources; instead, it assumes that the underlying applications are wrapped with a particular style of adapter wrappers that expose a task-based interface of the applications' behaviors of interest. In the context of the CelLEST project [17], we have also developed Mathaino [9], a tool that implements a semi-automated method for construction of such wrappers for legacy systems running on ASCII-based protocols such as tn3270. Furthermore, Babel provides a design-time environment, with an easy-to-use interface, for specifying the logic of the coordination among the wrapped applications being integrated.

The rest of the paper is organized as follows. Section 2 discusses the overall Babel architecture, its components and its approach to data modeling and coordination control. Section 3 briefly describes the design-time environment of Babel that can be used to support the specification of the coordination logic. Section 4 discusses the run-time behavior of Babel. Section 5 describes Babel's implementation and our experiments with it evaluating its performance, scalability and robustness. Finally, Section 6 summarizes the approach and concludes by identifying its novel contributions to the area.

2 The Babel Architecture

Figure 1 diagrammatically depicts the overall architecture of the Babel framework. It consists of two loosely coupled environments: a design-time environment called Visual X-logic Generator (VXG), which supports the "visual programming" of the mediation logic, and a run-time environment, which monitors the execution of the underlying wrapped applications and executing the mediation logic.

The run-time mediator of Babel enables the reactive integration of heterogeneous applications, wrapped within a particular type of adapter wrappers. These wrappers "translate" the original application interface into an event-based interface; they execute specific tasks on the original applications in response to input events, and they produce output events in response to the completion of these tasks. These events are monitored by the Babel mediator and recorded in the mediator's repository. In addition to maintaining the task history, the Babel mediator also maintains a registry of Event-Condition-Action (ECA) rules and workflow-session definitions, which specify the "logic" of the mediation. Upon receiving a task event form one of the wrapped applications, the mediator identifies the relevant rules and sessions. The events resulting from the rule execution are forwarded to the appropriate wrappers that proceed to execute the corresponding tasks on their underlying applications.

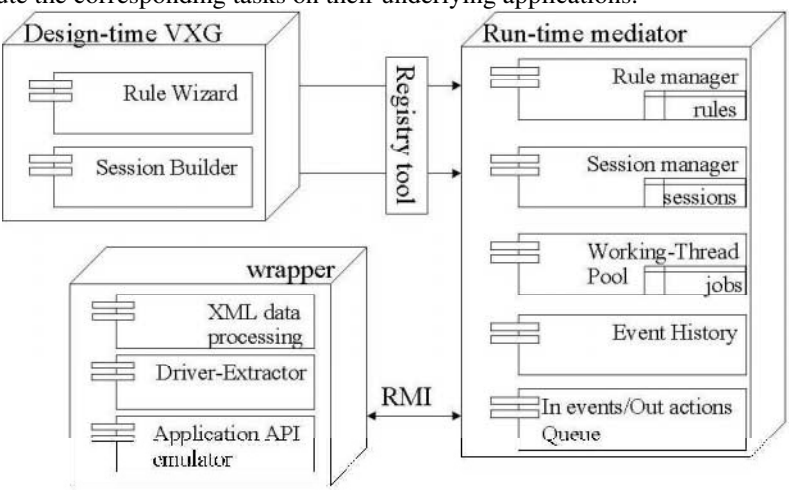

Figure 1: The overall Babel Architecture

VXG helps to alleviate the problem of "programming" the mediation logic. It consists of a "Rule Wizard", which guides the user to define the mediation application's ECA rules. VXG also provides a "Session Builder", i.e., a lightweight workflow-

process definition tool, to define higher-level coordination logic based on these ECA rules. Upon the completion of the visual programming, VXG "compiles" XSLT programs, implementing the defined rules and sessions, which are then registered with the run-time Babel mediator.

2.1 Application Wrapping

As is the case with all environments aiming at the integration applications of developed independently, Babel requires that all applications at the bottom tier of the run-time architecture be wrapped, in order to expose a canonical behavior to the mediator and thus hide their heterogeneity. There are two dimensions of heterogeneity that the Babel wrappers are designed to abstract away: the *data model* underlying the individual integrated applications and their *protocol of interaction* with their external environment.

Domain-specific data consumed as input (and produced as output) by the underlying applications are transformed from (and into) a canonical data model; this data model is the "lingua franca" of the overall mediation application [1]: all data exchanged at run-time between the wrappers and the mediator, as well as the coordination logic of the mediation application, are represented in terms of this model. It is therefore imperative that the syntax of the data-modeling language is expressive enough to succinctly model the data of realistically complex applications, so as not to complicate coordination logic that is based on top of it. Babel adopts XML as the syntax in which to specify the canonical data model of the mediation application. XML enables data modeling in terms of a "flexible" object-oriented syntax (XML schemas include support for inheritance and aggregation and allow for missing attributes). Furthermore, XML documents are self-documenting, since they include their underlying domain model in the tags surrounding their data, and can therefore be exchanged among applications, as long as these applications "agree" on the semantics underlying the domain model.

Different applications provide different modes of interaction with their environment. Legacy applications, running on mainframe hosts, are designed to have their services invoked from a ASCII terminal, implementing a protocol such as tn3270 or vt100. Others, designed in the client-server style, make their services accessible through HTTP requests issued by browsers. The Babel application wrappers abstract away these differences by viewing the applications as performing *tasks*. Tasks are abstractions of the interesting capabilities of the underlying applications [16], and are defined in terms of their type and their input and output information. Wrappers expose two types of events for each distinct task of their underlying application: a *task-initiation* event and *task-execution* event. Task-execution events flow from the wrappers to the mediator. Each such event constitutes, in effect, a record of a single execution of the task in question by the underlying application. The generation of task-execution events by the wrappers enables the mediator to monitor the behaviors of interest of the wrapped applications. The mediator generates task-initiation events, based on the task-execution events it has received and the coordination logic it enacts. Each such event

[1] We use the term "application mediation" to denote a specific instantiation of the Babel run-time environment, including the wrapped original applications, the domain model of the data exchanged among the wrappers and the Babel mediator, and the specific coordination rules and workflow sessions.

provides the necessary data for the application wrapper to execute the corresponding task of the underlying application. In this model, the integrated applications are viewed as components that accomplish a set of tasks by receiving events containing the required input for these tasks and by responding with events recording the task execution.

The domain-specific information contained in task-initiation and task-execution events is represented in terms of the canonical domain model of the mediation application. Each individual application wrapper is responsible for parsing the information contained in task-initiation events and using it to "drive" the underlying application so as to accomplish the desired task. Similarly, the wrapper is responsible for translating the data produced by the task execution into the canonical domain model to generate the corresponding task-execution event to the mediator.

Consider for example a digital library application that can be queried with a particular keyword, in response to which, it returns a set of three books with titles matching the keyword. An instance of this task, as represented in Babel, is shown in Figure 2.

```
<task>
    <t_type>Library1BookSearch</t_type>
    <t_id>1</t_id>
    <input>
        <information> type=book
                id=1
                keyword="databases"
        </information>
    </input>
    <output>
        <information> type=book
                id=1
                keyword="databases"
                title="Principles of Distributed Database Systems"
                author[1].last="Ozsu"
        </information>
        <information>
            ...
        </information>
        <information>
            ...
        </information>
    </output>
</task>
```

Figure 2: Representing the Library "BookQuery" task in Babel.

This task takes as input the "keyword" of a "book" and returns as output three book instances, each of which has four attributes, "author's last name", "keyword", "title", and "subject". In order to treat input and output objects uniformly, the task data model uses the "information"-structured data as input or output. The various types of "information" manipulated by the applications tasks constitute the overall domain of the mediation application. Information objects in Babel are represented in terms of the object type, object Id, and for each one of interesting attributes, an XPath-like expression characterizing the attribute in the context of the object and the attribute value. Finally, each task is characterized in terms of its "type", which uniquely identifies the wrapper consuming initiation events and produces execution events for this task, and its "Id", uniquely identifying each distinct execution of this task type on the underlying application. Note that task-initiation events forwarded by the mediator to the

wrappers do not have values for the expected output information objects; task-execution events, on the other hand, have values for all pieces of information included in the task input and output.

2.2 Coordination Logic

After wrapping the applications of interest, the next step towards developing a mediation application with Babel is to define the "logic" of how these applications will be coordinated at run time. The Babel framework supports two types of coordination logic: Event-Condition-Action rules and lightweight workflow sessions.

2.2.1. Even-Condition-Action Rules: The Event-Condition-Action (ECA) rule paradigm was developed in the context of active-databases research and was originally intended to support daemons monitoring transactions of interest and triggering further transactions in response [27]. The underlying intent of this paradigm is to enable the specification of high-level consistency constraints and to provide a mechanism for their maintenance. Often, application integration is motivated by similar needs, i.e., to maintain a high-level of consistency over the loosely integrated applications. For example, it is often desired to propagate data-entry transactions form one application to another and to trigger desired global side-effects of a transaction in one application to related ones. This is why we adopted ECA rule-based integration as the basic level of coordination logic supported by Babel.

A Babel *rule* is an *event-condition-action* triple that specifies what task-initiation event should be invoked as the *action* generated in response to a task-execution *event* received, in the context of a particular *condition*. Babel rules are also specified in XML, in terms of a high-level, natural-language description of the rule, a type, and a unique id. Both the condition and the action elements of a rule are optional. Rules with no conditions are always applicable, upon receipt of their triggering event. Rules with no actions are only used to advance the state of the workflow sessions.

The conditions of Babel's rules are logical compositions of simple *constraints*. An individual constraint is a predicate that must hold true between the input information provided by the task-execution event triggering the rule and the information contained in the task (–initiation and –execution) events recorded in the mediator's history. In addition to defining the rules' conditions, constraints are also used to define how the information provided as input to the generated task-initiation event, i.e., the action of the rule, should be constructed. The action data could may be constructed based on rule parameters input by the mediation designer, the generating-event data as well as data of the tasks contained in the mediator's history.

Consider, for example, a help-desk mediation application that integrates multiple libraries and bookstores. One service that this mediation application provides is to receive through email keywords from its customers and, in response, search for relevant books in all the partner libraries. To support this service, the two rules, shown in Figure 3(a), have to be defined.

The first rule of Figure 3(a) specifies, that when the mediator receives a task-execution event from its EmailReceipt wrapper, it should check whether the subject of the received email is " BookQuery". If it is, then the mediator should generate a task-initiation event for its "Library1BookSearch" wrapper. The input information for this task-initiation event should be "keyword" whose value should be whatever is in the

"Content" of the received email. The second rule defines the reaction of the mediator, when it receives a task-execution event from its "Library1BookSearch" wrapper. Upon receipt of such an event, the mediator should look in its history of task events to identify all the past receipts of " BookQueries" for the same keyword as the keyword appearing in the input of the current task-execution event. For all such tasks in its history, the mediator should generate a task-initiation event of type " EmailSend" and it should forward the current task's output book to the senders of the corresponding "BookQueries", found in the history. Note that the representations of rules shown in Figure 3(a) are simplified demonstrations; rules are actually represented as XSLT programs.

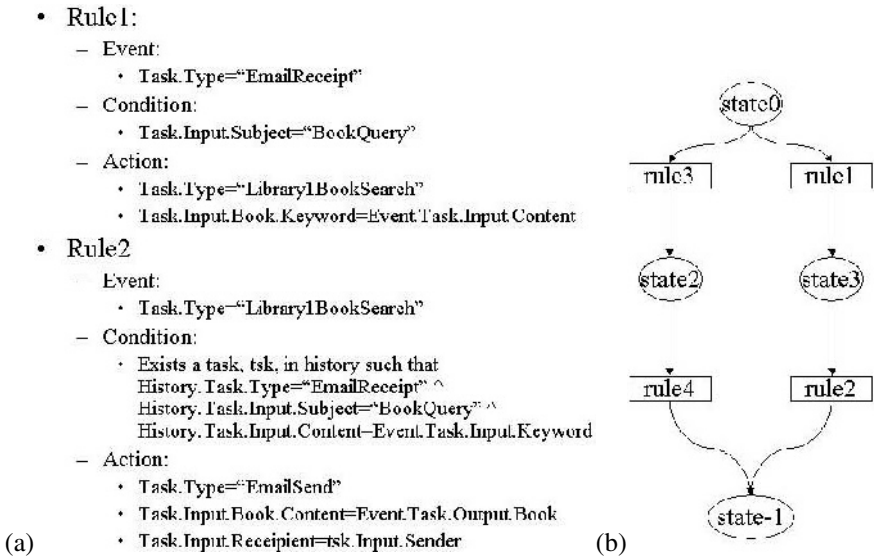

Figure 3: Two rules (a) and a session (b) for the Book Help Desk.

2.2.2 Workflow Sessions: Coordination based solely on ECA rules is not sufficient. Although, side effects can be effectively specified and maintained using such rules, more complex, multi-step processes are often desired. Workflow management systems provide the automated coordination, control and communication of work needed for multiple task execution [15], but to the best of our knowledge , no existing workflow model applies a similar event-based mediation approach yet.

In order to incorporate a lightweight workflow processing capability in the Babel mediator, we have extended the coordination language of Babel with the concept of a *session*. A session is a lightweight process that coordinates multiple tasks by combining multiple rules. Rule composition is based on sequencing, branching and looping operations. WFMC specifies nine valid run-time states for a workflow process instance [18]. The more limited Babel session incorporates the following states for its session instances:

- *Not Started:* the initial state of every session; a session requires the occurrence of a particular task-execution event to start;

- *Running:* after a session has started (the initiating task-execution event has occurred) and one of the rules applicable to this event has been successfully applied;
- *NotRunning:* after a session has transitioned to its new state and no incoming event has triggered any of the rules enabling a new transition from this state; and
- *Terminated:* after the session has completed, i.e., it has reached a state where no further rules are relevant.

The expressiveness of a workflow depends heavily on its flexibility. Several flow control formalisms being applied in workflow management systems (WFMS), such as Petri nets [1] and statecharts [19]. Babel is an event-based mediator and its control flow relies on (task-execution) events as triggers. Babel internally represents the defined workflow sessions as *Transition Vectors*. A *Transition Vector* holds all transitions involved in a workflow session definition. Each transition is defined in terms of a (from-state, rule, to-state[2]) triple. A session consists of a number of transitions, one or more of which may be *source* transitions. Source transitions have the 0^{th} state as their "from-state", and they specify when a new session instance should be initialized. When the mediator receives a new task-execution event, it checks the received event against all the start-up transitions of the registered workflow sessions. If one of the rule conditions in these transitions is met, a new instance of this session is spawned and its state advances from the 0^{th} state to the to-state specified by the transition. If the rule conditions of multiple start-up transitions are successful, the one with the lowest order in the transition vector will be followed. After a session instance is created, it switches to a "not-running" process state until another task-execution event is received. If any of the rules applicable to a received task-execution event is associated with a transition possible in the current state of the session instance, this session instance switches to the "running" process state and the rules' conditions are evaluated; if successful, the session instance advances to the to-state of the successful transition. If the all transitions at the current state of the session instance fail (the rules of the transitions fail), the session remains at its original state. Transitions with "to-state" as "-1" are *sink* transitions. Once a session instance passes one of its sink transitions, it expires, i.e., it switches to the "terminated" process state.

In order to prevent ambiguities, no two transitions with the same "from-state" and the same "rule" are allowed in the transition vector of a session. Moreover, each transition vector must have at least one start-up transition and one terminating transition.

Consider for example, the BookServices application introduced above. An additional service could be to search from for a book-price quote, if the same user searches for the same book title three times. This new service and the one we described in the section above are not independent; they are in fact two mutually exclusive alternatives, and it would be more efficient if this fact were represented with a workflow session. Figure 3(b) diagrammatically depicts the transition vector of such a session. Here rule3 is similar in content to rule1, with two exceptions: it is conditioned upon finding at least three similar "BookQuery" task-execution events in the history, and instead of generating a LibrarySearch task-initiation action, it generates a " BookstoreSearch"

[2] We use the term "state" to denote (a) the state of the process executing a session instance and (b) the logical states comprising a workflow session definition. Wherever ambiguous, we use the term "process state" to denote the former.

action. Rule 4 is similar with rule 2, in that it sends query result of the book from a book dealer (www.amazon.com) instead of from a library. This session will be activated when a task-execution event of the type " BookQuery" is received, in which case, either rule3 or rule1 will be successful. If rule3 succeeds, the first transition will lead the session instance to state2, and rule1 will not be evaluated. If rule3 fails, i.e., the first two times that the user requests information about the same book, rule1 will succeed and the session instance will move to state3. From state2 and state3, the session instance will move to state-1, i.e., it will expire, when it receives a "LibraryBookSearch" or a "BookstoreSearch" task-execution event when rule4 or rule2 will be applied correspondingly. The reason that rule 3 is always checked before rule 1 is that rule 3 is inside a transition with a lower order than rule 1.

3 VXG: The Babel Design-Time Environment

VXG, the Babel design-time environment supports the specification of ECA rules and workflow sessions through two visual-programming components, the *Rule Wizard* and the *Session Builder*. The Babel user can use VXG to define rules and sessions with simple dialog operations and drag-and-drop actions in response to the wizard prompts. The storyboard of Figure 4 below illustrates the rule- and session- definition process with Babel's VXG.

The interface supports the specification of rules (steps a – h) and sessions (steps i – l). To define a rule, the user has first, to give a short description (in English) of the logic that the rule is intended to implement (step a). Next, the user has to choose the type of events that can potentially activate the rule and the type of events that must have already occurred in the past on which the rule's activation may be conditioned (step b). The interface enables the user to select both these event types from a drop-down menu that is configured to include as choices all wrappers known to the system. These event types appear as components in the main panel of the interface. Then (steps c,d) the user can define functions on the various types of information associated with the chosen event types by dragging and dropping the corresponding ports of the selected components. Next (step e) the user selects the type of action that will result from the rule's invocation and defines the action's input in terms of information contained in the triggering and condition events (step f). Finally, the user may inspect and edit the XSLT program implementing the defined rule (step g), and, if it is correct, the rule is saved in the system's repository under a user-provided file name (step h).

To define a session, the user has, first, to import the relevant rules by selecting their specifications from the system (step i). The description of the imported rules is shown as a tool tip, when the user moves the mouse over the icon corresponding to the rule (step j). Then, in the main panel of the interface, she can select design a state-transition network, by defining new states and by using the imported rules to define the transition between them (steps k,l). The sessions defined in this manner are also "translated" by VXG into XSLT programs that are saved in Babel's environment.

Figure 4: Defining ECA Rules and Workflow Sessions with Babel's VXG.

At the end of the process, VXG produces the XSLT programs corresponding to the defined rules and sessions. No XSLT knowledge is required to perform the tasks by using the Babel design-time to write the complex coordination logic.

4. The Babel Run-Time Environment

At run time, ECA rules and workflow sessions are dynamically registered with the Babel mediator. At the same time, the mediator receives task-execution events from the wrappers of the underlying applications. Whenever a data stream comes to the mediator, the mediator parses it, infers its content type, i.e., whether it is a rule or a session or a task event, and validates it according to the appropriate schema. If the input is a rule definition, the Babel mediator stores it in its *rule repository*. From this point onward, task-execution events of the type specified in the event field of the reg-

istered rule will activate this rule. If the input is a session definition, the Babel mediator registers this session with its *session manager*. If the input is a task-execution event, the Babel mediator places it into its *event queue* and gets ready for receiving the next input.

4.1 Processing Task-Execution Events

An independent thread, inside the Babel mediator, processes the event queue, according to First-In-First-Out policy. Given the type of the task-execution event being processed, the session manager, first, evaluates whether a new session instance should be initialized. If any of the start-up transitions of the registered sessions are conditioned upon a rule applicable to this task type and the rule is successful, a new instance is created (as discussed in Section 2.2 above). Second, all the currently active session instances are examined to evaluate whether they can advance to a subsequent state; these which are at a state waiting for the application of a rule conditioned upon an event of the current type are added to the *job pool*. Finally, for each of the registered rules applicable to the type of the task-execution event at hand, a corresponding "task-rule" *job* is added to the *job pool*.

Babel's *job pool* is a data structure that maintains a bag of jobs, i.e., pairs of task-execution events and the applicable rules. The Babel mediator processes these jobs in parallel -- each rule or session is applied on the task event by an independent thread. Babel adopts a *Working-Thread Pool* model for its parallel. When the Babel mediator is initialized, a fixed amount of threads are created and organized in a working thread pool. Working threads continuously look for pending jobs in the Job Pool. A thread either starts working on the selected job on successful job retrieval, or suspends itself until new jobs become available.

This Working Threads Pool (WTP) paradigm shortens processing latency because creating a thread is an expensive operation to be avoided during mediation runtime. Meanwhile, keeping a fixed number of threads ensures the stability of the system because threads will not grow indefinitely to exhaust system resources. Furthermore, since the jobs are dynamically assigned to the worker threads, workload is balanced among the threads. When all the worker threads have completed their processing, Babel dispatches all the task-initiation events generated to the corresponding wrappers, and updates its information repository to record these new events.

4.2 Application-Wrapper Processing

When the Babel mediator generates a task-initiation event for an application wrapper, the wrapper starts "driving" the underlying application in order to execute the task corresponding to the event it received.

A wrapper consists of three layers: the XML data interface layer, the driver and extractor layer, and the API to the underlying applications layer. The XML data interface layer retrieves from the mediator task-initiation events and parses them to extract the input required by the underlying application to execute the task. This layer also compiles the data output by the underlying application, after it has completed the task in question, into task-execution events and sends them back to the mediator. This layer is quite general and is shared by all application wrappers. The driver-and-extractor layer consists of wrapper-specific modules. The driver 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 is detected and sent to the Babel mediator by the extractor module. These two modules operate on top of the API layer, which consists of the APIs specific to the underlying applications.

The wrappers in the Babel environment can be constructed using the Mathaino tool of the CelLEST environment [9]. Mathaino (semi-)automatically constructs front-end interfaces for legacy user interfaces, based on traces of the legacy system-user interaction. Other types of applications, such as web-based applications accessible through browser thin-clients, are wrapped manually.

The Babel wrappers are heavy-weighted components that monitor and drive the underlying application. The Babel mediator communicates with the wrappers by sending and receiving data asynchronously. The Babel architecture also allows the wrappers to be built as light-weighted components, extensions of the mediator itself. The communication of the mediator with these extensions is synchronous. The Babel architecture provides a framework for building such extension services as hooks to this framework.

5 Implementation Overview and Performance Evaluation

Both the design-time and the run-time environments of the Babel framework are implemented using JDK1.3. The XML parser is provided by JAXP 1.1 [8] and the XSLT processing engine for the run-time rule and session enactment is based on SAXON 6.2.2 [14]. The mediator run-time includes an information repository module, currently implemented with flat XML files. At run time, the task-execution (-initiation) events from (to) the wrappers to (from) the mediator are XML data exchanged over RMI.

We evaluated the performance of the run-time Babel mediator in terms of efficiency, scalability and robustness with simulations performed on sun4u SPARC-SUNW, Ultra-30, under SunOS 5.6. Our experiments were designed to evaluate the performance of the Babel mediator, independent of the network latency in delivering events from (to) the wrappers and the wrapper processing itself.

A wrapper sends events to the mediator periodically, with constant time t_p between subsequent events. Timing starts when the mediator receives the event. The mediator de-serializes the object received from the RMI interface, reorganizes the data into a "raw" XML document, parses and validates it, and saves it in the event queue. Eventually, this event is retrieved from the queue and processed by the Rule Manager and the Session Manager. Then it is saved in the mediator's history. All these activities are recorded for mediator performance profiling. In the following simulation, costs in various aspects will be estimated.

5.1 Efficiency

For each experiment run, the time between every two events, sent to the mediator, is constant t_p. Suppose an event-message is received at t_r and that this message is processed and the resulting messages (if there are any) are placed in the message dispatching queues of each wrapper at t_f. Then the mediation processing time is $t_s = t_f - t_r$. The time elapsed between t_s and t_p measures the mediation efficiency, independent of

the wrapper performance and communication issues. Different values of t_p were chosen as multiples of 200 milliseconds to evaluate the mediation performance.

Figure 5 shows a negligible service time for pure event processing with no rules triggered, which means that the message queuing, parsing, validating and information repository updating costs are trivial. The other three lines depict the average processing time when 1 rule, 2 rules, or 3 rules are triggered by the incoming task-execution event, correspondingly. In addition to the number of rules triggered, the complexity of the actual rules triggered may also have an effect to the processing times; the reported simulation results are averaged 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 decreases as the time gap increases, and at certain time gap, the average processing time becomes constant. A similar number reported in [5,12] is higher.

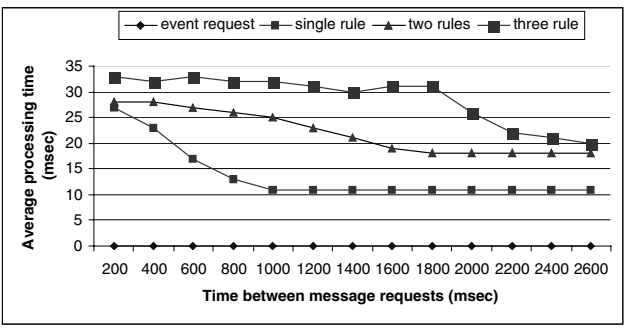

Figure 5: Average processing time (ts) vs. time between message requests.

5.2 Scalability

To evaluate the mediator's scalability, we defined several rules to be triggered by events of the same task type. After the mediator receives an event of this type, the Rule Manager retrieves 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, 1 to 10 rules were registered to Babel and the corresponding average mediator processing time T_s (measured in the same way as first simulation) was recorded. Standard deviation ranges from *3.2–5.8*.

Figure 6: Average processing time (t_s) vs. Number of rules r_n.

As can be seen from Figure 6, 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. We can therefore infer that the parallel rule-processing model scales well with large number of concurrently triggered rules.

5.3 Robustness

Finally, another important metric to evaluate is the mediator's capability of handling message flooding, i.e., a large amount of events simultaneously sent to the mediator. In order to test this capability, we started a lot of wrapper instances on the machine on which the mediator runs, and bombarded 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.

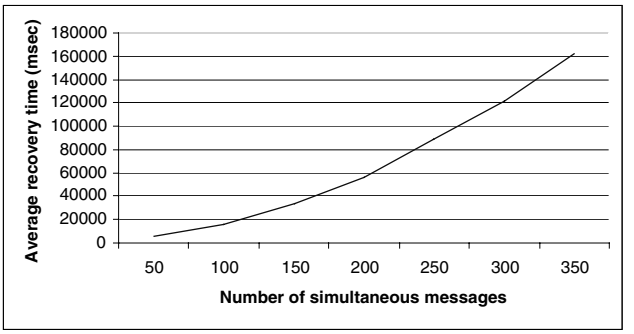

Figure 7: Average recovery time vs. Number of simultaneous messages.

As shown in Figure 7, the mediator is able to recover within a very short period of time (5.317 sec) for n_r=50 (standard deviation ranges from 0.49—0.83 sec). The mediation slows down as the number of messages increases and reaches 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.

6 Conclusions and Further Work

In this paper, we discussed Babel, an application-integration framework. Babel is certainly not unique in providing an infrastructure for rule-based mediation over independent applications. CoopWare [5,12] is a similar framework, on which in fact we based our evaluation of Babel. However, Babel's approach to application integration is novel in the following respects.

First, it enables both rule- and session-based integration of the underlying applications. Reactive rule-based integration is sufficient for establishing consistency rules between pairs of tasks across applications. However, as the execution of rules produces new events that may cause new rules to apply, tracing the overall consequences of the rules becomes diff icult. If the rules are fairly independent, i.e., not many task-initiation *actions* and task-execution *events* refer to the same task types, then there are few secondary effects of the rule execution, and therefore few long-term conse-

quences. If however this is not the case, and tightly controlling the overall integration behavior is important, then more complex process descriptions are required, and this type of integration is also supported by Babel, with workflow sessions.

Second, the coordination logic of a mediation application developed with Babel, i.e., the event-condition-action rules and the workflow sessions, are declaratively specified in XML. They are therefore easy to inspect and modify. Even more importantly, the VXG interface supports the specification of the coordination logic and thus alleviates the burden of the programmer. At the same time, by virtue of the fact that the logic is compiled into XSLT programs that are executable by the run-time Babel mediator, the mediation applications are easily extensible. If the programmer has a strong XSLT background, he/she can bypass VXG and manually program more complex rules to be executed by the mediator.

Finally, the event-based description of the underlying application services is similar to the recently proposed WebServices WSDL model. In this model too, applications are viewed as components that export services; each service consists of a set of operations (≈tasks) and each operation is defined in terms of two messages, one to provide the input necessary for the method implementing the operation (≈task-initiation event) and one to request the output of the method (≈task-execution event). An interesting difference is that in the Babel model, task-initiation events flow from the mediator to the wrappers and task-execution events flow from the wrappers to the mediator, where in WebServices, messages providing the service input and requesting the service output flow in the same direction, from the client to the server. WSFL, the workflow-level standard of the WebServices stack, is agnostic with respect to the integration style of the underlying server applications. In our future work, we plan to investigate whether or not (and how exactly) the Babel coordination logic can be translated in WSFL.

References

1. 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.
2. Batini, C., Lenzerini, M. and Navathe, S.B., “A comparative analysis of methodologies for database schema integration”, ACM Computing Surveys, Vol. 18, No. 4, Dec. 1986, pp. 323-364.
3. E. Bertino, “Integration of heterogeneous database applications though object-oriented interface”, Information systems, 1989, pp. 407-420.
4. 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, February 1986, pp. 301-310.
5. A. Gal, J. Mylopoulos, “Supporting Distributed Autonomous Integration Services Using Coordination”, International Journal of Cooperative Information Systems, vol.9, no.3 pp. 255-282, 2001.
6. 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), pp. 25-30, 1999.
7. S. Heiler, “Semantic Interoperability”, ACM Computing Surveys, 27(2):271-273, June 1995.
8. JavaTM APIs for XML Processing (JAXP), http://java.sun.com/xml/xml_jaxp.html

9. R. Kapoor, " Mathaino: Device Retargetable User Interface Migration UsingXML", University of Alberta, TR-01-112001.
10. 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.
11. C. Baru, A. Gupta, B. Ludascher, R. Marciano, Y. Papakonstantinou, P. Velikhov, V. Chu, "XML-based information mediation with MIX", ACM SIGMOD, pp. 597-599, 1999.
12. J. Mylopoulos, A. Gal, K. Kontogiannis, M. Stanley, "A Generic Integration Architecture for Cooperative Information Systems", Proceedings of the 1st IFCIS Intl. Conference on Cooperative Information Systems, Brussels, Belgium, pp. 208-217, June 1996.
13. Y. Papakonstantinou, H. Garcia-Molina, J. Widom, "Object Exchange Across Heterogeneous Information Sources", In Proceedings of Eleventh International Conference on Data Engineering, Taipei, Taiwan, 251-260, 1995.
14. XSLT processor implementation, http://users.iclway.co.uk/mhkay/saxon/
15. A. Sheth, D. Georgakopoulos, S.M.M. Joosten, M. Rusinkiewicz, W. Scacchi, J. Wileden, A. Wolf, "Reports from the NSF Workflow and Process Automation in Information Systems", ACM SIGMOD Record, Volume 25 Number 4 December 1996.
16. 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, pp. 400-410, Montreal, Quebec, Canada.
17. E. Stroulia, M. El-Ramly, P. Sorenson, R. Penner, "Legacy Systems Migration in Cel L-EST", Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland (June 4-11, 2000), pp. 790. IEEE Computer Society Press.
18. "The Workflow Reference Model", Workflow Management Coalition, Document Number TC00-1003, http://www.wfmc.org/standards/docs/tc003v11.pdf.
19. D. Wodtke and G. Weikum, "A formal foundation for distributed workflow execution based on statecharts", 6th International Conference on Database Theory (ICDT 97), pp. 230-246, 1997.
20. G. Wiederhold: Mediation and Software Maintenance; Technical Note STAN-CS-TN-95-26.
21. H. Zhang, and E. Stroulia, "Babel: Application Integration through XML specification of Rules", 23rd International Conference on Software Engineering (ICSE 2001), 12-19 May 2001, Toronto, Canada. 831-832. IEEE Computer Society Press.
22. D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch, or Why it is hard to build systems out of existing parts. In Proceedings 17th International Conference on Software Engineering, pp. 179-185, Seattle, Washington, April 1995. ACM SIGSOFT.
23. K. J. Sullivan and J. C. Knight. Experience Assessing an Architectural Approach to Large-Scale Systematic Reuse. Proceedings of 18th International Conference on Software Engineering, pp. 220-229, IEEE Computer Society Press.
24. A. Sahuguet and F. Azavant. "Looking at the Web through XML Glasses", Conference on Cooperative Information Systems, pp. 148-159, 1999.
25. B. Adelberg. NoDoSE: A tool for semi-automatically extracting semi-structured data from text documents. In Proc. Intl. Conference on Management of Data, pp. 283-294, 1998.
26. L. Liu and C. Pu. CQ: A personalized update monitoring toolkit. In Proc. of the ACM SIGMOD Conf., pp. 547--549, Seattle, 1998.
27. U. Dayal, E.N. Hanson, and J. Widom. Active database systems. In W. Kim, editor, Modern Database Systems: The Object Model, Interoperability, and Beyond. ACM Press, New York, 1994.