

A Declarative Control Language for Dependable XML Message Queues

Alexander Böhm

Carl-Christian Kanne

Guido Moerkotte

Universität Mannheim, Germany

`alex|cc|moer@pi3.informatik.uni-mannheim.de`

Abstract

We present a novel approach for the implementation of efficient and dependable web service engines (WSEs). A WSE instance represents a single node in a distributed network of participants that communicate using XML messages. We introduce a fully declarative language custom-tailored to XML message processing that allows to specify business processes in a concise manner. To support the efficient and reliable evaluation of our language, we show how to augment a native, transactional XML data store with efficient and reliable XML message queues.

1. Introduction

Web services proliferate as the technology of choice for the implementation of complex, distributed business processes that connect a variety of participants with heterogeneous hardware and software platforms. The corresponding standards and tools are based on mature, widely deployed technology such as HTTP and XML, and allow developers to concentrate on application-specific problems instead of burdening them with decisions related to infrastructure, such as format conversions and protocol details.

A simple, but adequate model for a network of web services is a set of nodes that interact by producing and consuming XML messages. In this paper, we introduce a novel approach for the implementation of dependable nodes in such networks, liberating developers from even more infrastructural details.

Before we turn to a more detailed description of our work, let us briefly motivate our approach by reviewing conventional implementation techniques for web services. Today's systems usually implement web service protocols as an additional tier on top of existing middleware solutions [2], further aggravating the problem of complexity and poor integration that already plagues these systems [22]: Typically, the actual business processes are specified using imperative, high-level languages such as Java, C# or C++. An external call to such a web service travels through the various layers: The XML-based web service invocation is transformed into the middleware's representation, again

transformed into the programming language's representation, with further transformations thrown in as other components such as relational DBMSs are accessed. Delivering the result requires a reverse traversal of this "transformation chain". This not only hurts performance, but also reduces developer productivity, because each layer requires at least some separate design and coding that is not related to the actual application domain. The interaction of various configuration options and code fragments is difficult to understand, optimize, and maintain.

As more critical business processes are automated using web services, their *dependability* becomes more important and complicates implementation further. In the above scenario, this requires the integration of transaction processing monitors (TP monitors) across all layers to guarantee transactional semantics in case of error conditions induced by application, system or network failures.

The goal of our research is to investigate how to design a Web Service Engine (WSE) that avoids the complexity of the conventional architecture while maintaining flexibility, compatibility and dependability.

Our main contributions are:

1. We propose a novel, fully declarative application language custom-tailored to XML message processing.
2. To support the efficient evaluation of our language, we show how to extend a native XML data store with transactional XML message queues.

This paper is organized as follows. Sec. 2 derives both functional and non-functional requirements for a Web Service Engine (WSE). In Sec. 3, we propose an architecture for a WSE, based on one component for message storage and one for controlling the message flow. Message flow is specified by a declarative, rule-based XML processing language detailed in Sec. 4. Sec. 5 discusses related work. Sec. 6 concludes the paper.

2. Requirements

In the following, we use an example business procurement scenario to illustrate various issues a Web Service Engine (WSE) has to address. Figure 1 gives an overview of

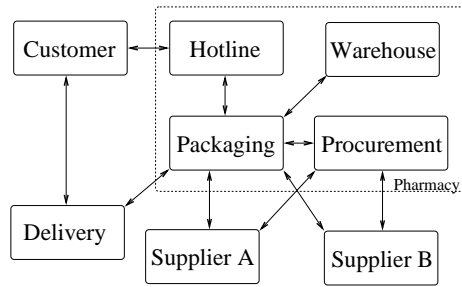


Figure 1. Example web service orchestration

the involved components. The central element of the scenario is a pharmacy wholesale application, consisting of an ordering hotline, a warehouse, a procurement and a packaging department. The hotline communicates with the shop's customers and forwards incoming orders to the packaging. There, they are decomposed, and the corresponding items are either retrieved from the warehouse (if in stock) or forwarded to the procurement department for ordering from one of the two external suppliers. Finally, when the order can be completely fulfilled, it is delivered to the customer via an external delivery service. While this example is far from modeling a complete business process with all the involved complexity, it illustrates some of the fundamental operations and tasks that have to be performed in a distributed business scenario.

In a web service environment, the involved business partners communicate by using SOAP [18]. SOAP is a messaging standard supporting both synchronous and asynchronous message exchange using transport protocols such as HTTP and SMTP. Message header and body information is encoded using XML. There are numerous "horizontal" web service extensions that extend SOAP e.g. in order to incorporate cryptography or reliable messaging [2].

Given the SOAP standard, the specification of the functionality of a web service is the specification of XML message flow. The job of a WSE as participant of a business process is to react to incoming messages by inspecting them, combining them with other local information, transforming them and sending result messages.

This section motivates our proposed XML message processing architecture by deriving both the functional and non-functional requirements a WSE has to fulfill in order to efficiently execute distributed business processes.

2.1. Application Language

This section discusses the requirements for a language for web service implementation. All of these requirements are driven by the desire to transfer the responsibility for infrastructure-related tasks from the developer to the WSE, increasing his productivity.

2.1.1. Message Processing Language Today, messaging systems such as [15] usually provide application programming interfaces (APIs) for imperative high-level, object-oriented languages such as Java or C++ that are used to implement the application logic. In a web service environment where communication is entirely message-based, there are recurring processing patterns such as retrieving a message, examining and eventually transforming the content, and sending the resulting message to another component. In order to perform these tasks, each of the above operations has to be reflected by corresponding application code, thus making web service development with general-purpose programming languages very labor-intensive and requiring dozens of lines of auxiliary code apart from the actual business logic. To simplify and speed up software development for web services, the application language of a WSE has to directly support the involved processing patterns and communication models.

2.1.2. Native XML Support The messages exchanged in a WSE are not opaque container objects but contain the business data as XML, a standardized, machine-readable format. Today's messaging systems lack native XML processing support, leading to an impedance mismatch between the message format used for communication and the data representation of the programming language (e.g. a Java object). To remove the burden of designing and manually implementing an optimal XML representation from the programmer, our language must support XML as a primary data type. It must further provide powerful primitives to query structure and content of XML documents, to transform existing and to construct new XML messages.

2.1.3. Declarativity Another drawback of using imperative, high-level programming languages for implementing web services is the limited optimization potential. Because imperative languages explicitly specify all steps of an application and their order, there is little freedom for the execution engine to optimize the resulting algorithm. We believe a declarative language is better suited to our domain: We need a language that allows to specify the desired structure and contents of outgoing messages based on contents and structure of incoming and stored messages, and that has a clear semantics and processing model. This allows us to investigate alternatives on how to build an efficient WSE which may reorder, factorize, simplify and in other ways optimize the application specification, as long as the resulting system behaves as specified.

2.2. Data Access and Retention

Even after a business transaction has been successfully executed, the exchanged messages often have to be retained and declarative queries on messages that have already been processed must be possible. For example, when an order

confirmation message is processed, all the individual items that were ordered must be retrieved from previous messages. Other reasons for retention of processed messages are accounting purposes, legal obligations, conflict handling, or data mining applications. Additionally, in order to allow reconstructing the entire conversation that has taken place, not only the messages received but also the outgoing communication has to be retained for maintaining a complete communication record.

2.3. Dependability

Transaction Support Communication in a web service environment represents critical business data, loss or inconsistency of which may incur severe financial damage. To avoid this, the WSE has to guarantee transactional properties when processing messages. In our pharmacy example, the hotline forwards the customer's orders to the packaging department. The customer's message is deleted at the hotline after a corresponding message has been inserted at the packaging component. The system has to make sure that these two operations are executed with all-or-nothing atomicity, as partial execution would lead to loss of data or duplicate shipping. Additionally, once arrival has been acknowledged at a particular component, message storage has to be durable, again to avoid losing critical business data. In a distributed environment, these demands involve additional complexity and communication overhead as they require additional precautions such as distributed commit protocols [21].

Robustness Web services inherently interact with a multitude of heterogeneous systems of varying quality. Further, business processes constantly evolve. As a result, a corresponding execution system has to be capable of dealing with many kinds of errors that may be encountered in such an open, distributed architecture. This includes application level problems such as messages with invalid structure.

2.4. Parallelization

A business transaction such as ordering medicine in the above pharmacy scenario usually consists of multiple steps, often involving human interactions or other long-lasting operations such as delivering the ordered items to the customer. Thus, these business transactions are long-lived, including steps that span several days or even longer.

Exclusively locking resources and thus effectively preventing other clients from accessing the system until a transaction is finally completed is not feasible for a business application. Instead, a WSE must be capable of handling multiple application instances in parallel and particularly allow for multiple clients simultaneously accessing the system.

Apart from serving multiple communication partners concurrently, a WSE must also support parallel execution of business applications, e.g. when deployed on a hardware

cluster or executed on symmetric multiprocessing hardware to increase processing performance. As a result, it has to allow for concurrent and synchronized access to the underlying message store by multiple processes. This includes support for intra-application parallelization, as parts of a business transaction can usually be executed in an arbitrary order or even in parallel. For instance, in our pharmacy example, the accounting could be done while the order is being shipped to the customer.

3. Architecture

Reviewing the requirements from Sec. 2, we identify the core tasks of a WSE as the *control* and *storage* of XML messages. We propose an architecture consisting of two components that reflect these core tasks: A message control component executes the declaratively specified applications by receiving, transforming, creating and sending messages. A transactional message store component provides reliable storage of and efficient access to the currently processed XML message.

In this section, we present our overall approach for the realization of these two components, and discuss corresponding design decisions.

3.1. Message Store

The message store decouples the acceptance of an incoming XML fragment from the actual processing by means of message queues. Queues are used for supporting message-based data exchange in many enterprise-class applications and integration solutions [10, 15]. They provide both fast insertion and retrieval operations while preserving the order of messages.

One reason for the usage of message queues is to avoid resource contention. A queue allows to defer the processing of a message until there are enough resources. Storing messages in a well-defined abstract data structure before processing them also has an advantage for reasons of dependability: It allows to apply transaction processing techniques to ensure ACID properties on the message queues.

There are two existing areas of research, each covering one important aspect of transactional XML message queues. We review them here to motivate our approach before describing our design.

3.1.1. Existing Approaches

Queue-enabled relational DBMS Integration of queues into database management systems (DBMSs) has been repeatedly advocated in the database community [16, 17], eventually leading to queue support in major object-relational database systems such Oracle Streams Advanced Queuing [13] or Microsoft SQL Server Service Broker [25]. An important effect of this integration is the ability to access messages *and* regular persistent data in a uniform manner.

Unfortunately, the existing DBMSs that support message queues provide reliable queues for tuple-structured data, but not for XML messages.

XML Data Stores Management of XML data collections is the domain of XML data stores (XDSs). These systems are DBMSs custom-tailored to process XML fragments. They allow efficient data access both in main memory and on secondary storage, thus avoiding impedance mismatches and also allowing for data retention. An XDS also incorporates support for transactions and various declarative XML query languages, such as XPath and XQuery [5, 8]. One of the first enterprise-level commercial XDSs is IBM DB2 [7]. However, no existing XML data store supports message queues.

3.1.2. Integration of Queues into an XDS As argued above, neither queue-enabled conventional DBMS nor native XDS meet the requirements stated in Sec. 2. With XML fragments as the only communication primitive, we chose to extend an existing XDS with a message queue data structure to achieve reliable and efficient message handling.

Due to space constraints, we only give a brief summary of the challenges resulting from incorporating transactional message queue support into an XDS. We were able to solve these issues by using Natix [11], an XDS with a highly modular architecture that allows to incorporate new data structures without modification to the existing subsystems.

Order Document collections in XDS are unordered sets of documents. In contrast, a message queue is ordered, has exactly one head, where new messages are enqueued, and one tail, where messages are dequeued. To avoid starvation effects where some messages are stored but never processed because new messages arrive, the queue order must be respected.

Transactional Semantics To provide reliable message handling primitives, queues must be supported by the transaction subsystem to guarantee isolation and recovery. Several queue operations must be executable in a single transaction. This is necessary to guarantee that dequeuing a message is only committed if it could be successfully processed, where successful processing implies queuing of new messages. Dequeuing the processed and queuing the resulting message must be atomic.

Transactional processing involves a significant overhead. For parts of applications that are not critical (e.g. product announcements or newsletters), we allow to create transient queues whose contents are lost in case of system crashes.

Concurrency A strict conservation of order in the message queue, combined with transactional semantics, would prohibit the required parallel processing of messages. In systems that follow this approach [25], the next message can only be dequeued after the dequeuing of the current message has committed, to guarantee order preservation even if transactions abort. Hence, we relaxed the strict ordering of

messages in a queue, and allow parallel dequeuing of messages, as advocated by Bernstein et al [6], and implemented in Oracle AQ [15]. In case of a transaction rollback, we logically undo the dequeue operation by re-enqueuing dequeued messages at the tail instead of the head. As a result, these messages are reprocessed before any new messages, albeit not in the same order as they originally arrived. We tolerate this loss of full serializability for the sake of increased concurrency.

3.2. Message Control

3.2.1. Rule Language The conventional approach to implement a web service, given a message store as explained above, is a set of server processes which retrieve messages from queues, process them, and enqueue reply messages. These server processes are usually specified as imperative programs (e.g. in Java or C++). Our research focuses on an alternative approach: The fully declarative specification of message flow using a rule language. We will elaborate on our rule language in Sec. 4. Essentially, a rule in our language specifies how to react to a new message in a queue, and all relevant system events are modeled as XML messages. To avoid impedance mismatch and data conversion, our rule language directly operates on queues and XML messages, leveraging standard XML query languages for specification of conditions and actions on messages.

3.2.2. Execution Engine Our rule language is complemented by an engine process which executes business processes according to a set of specified rules. We will now describe the engine's processing model, the mapping of rule executions onto message store transactions, and the integration of remote services.

Processing Model Our processing model is represented by a simple processing loop. Each loop iteration determines an unprocessed message, dequeues this message, finds all the rules that are triggered, and executes the corresponding actions. This may result in forwarding the message to another queue or in the creation of new messages. Those new messages are processed in subsequent iterations over the loop.

Transaction Mapping An important research question in this respect is how to map the execution of rules onto message store transactions. This greatly affects the level of concurrency. If every triggered rule is executed as a separate transaction, then system states are visible to other transactions where the effect of a single message is only partly incorporated, making it difficult to understand and reason about the overall system behavior [10]. If all rules that are triggered by the arrival of a new message are executed in a single isolated transaction, the induced synchronization might block a large amount of concurrent activity, in particular if processing the message results in new messages which recursively trigger yet more rules.

We are still investigating this area, but have a tentative approach for now: Every iteration of our processing loop is mapped to one message store transaction, and hence only rules triggered directly by a message are executed in the same transaction. Rules that are triggered recursively due to transitive message creation are executed in subsequent iterations, hence separate transactions. Further, the set of triggered rules is determined before any actions are executed. This "snapshot semantics" makes the interaction of rules and action simpler to understand and reason about. In the future, we intend to allow the explicit specification of spheres of control [19] together with rules or rule groups, to fine-tune the set of guaranteed transaction properties to the application.

Business Processes A business process has a longer lifetime than a single message store transaction. Using the basic local transactions (as explained above) as building blocks, higher-level transaction and cooperation protocols can be specified in form of rules for message processing, allowing explicit specification of advanced concepts such as compensating transactions [3, 14] in form of message processing rules.

Distribution To keep the core rule execution engine simple, we have chosen to incorporate the actual communication with external nodes in form of special "gateway" queues. If a message is inserted into such a queue, a separate communication manager process tries to deliver it according to a specified protocol. The enqueued XML fragment specifies recipient(s) and other metadata, and the actual message data. Other types of gateway queues represent incoming external messages, i.e. web service calls to the local node. Such incoming messages enqueued by the communication manager and messages created by other local rules are treated in the same way by the execution engine.

The interaction of the communication manager with the message store is determined by the messaging protocol. For example, when using distributed commit protocols such as 2PC [21] for reliable communication with remote services, this has to be reflected by corresponding prepare and commit phases on the message store. Simpler transaction protocols may commit the dequeuing transaction as soon as the related system call succeeds.

4. Message Control Language

Our language is based on Event-Condition-Action (ECA) rules that have proven their usability and effectiveness in various application scenarios and technologies such as active database systems [24] and workflow management systems [2]. The core design concept of our message control language is simplicity. To keep our execution engine simple and extendable, the language introduces only a few, fundamental new primitives while

```

on message into [class] QueueName
define
    variable as SimpleExpr
    ...
if SimpleExpr
define
    variable as SimpleExpr
    variable as { ComplexExpr }
    ...
do ACTION(SimpleExpr,...)

```

Figure 2. Generic rule structure

leveraging existing XML query languages where possible.

Our language comprises two sublanguages: One for queue definition, corresponding to the "static" application structure, and one for queue processing rules, specifying the "dynamic" application behavior. We will now briefly sketch the Queue Definition Language (QDL) before explaining some of the core concepts of the Queue Rule Language (QRL) in greater detail.

4.1. Queue Definition Language (QDL)

The QDL describes the queues and queue classes used in an application. Queues have a unique name and can be persistent or transient (see Sec. 3.1.2). Further, queues can be grouped into classes. This simple hierarchical organization allows to specify rules that apply not only to single queues, but to a whole group of queues.

In addition to simple local queues, there are several special queue types. In Sec. 3.2.2, we have introduced gateway queues, that represent communication endpoints to remote web services. Specification of gateway queues contains the supported messaging protocols and/or horizontal web service extensions. Another special queue type is the *echo queue*. Messages sent to echo queues start a timer and are re-enqueued at the specified return queue once the timeout has expired. Specifying rules on echo queues allows to react to time-related events, such as expiry dates of drugs in our pharmacy example.

4.2. Queue Rule Language (QRL)

A QRL program consists of a set of rules of the general form shown in Figure 2. QRL rules are so-called ECA (Event-Condition-Action) rules that specify an **action** that reacts to certain **events**, provided that a particular **condition** is true. To keep the system behavior simple, the set of rules is static and cannot be modified as part of rule execution. Below, we elaborate on the components of a rule.

4.2.1. Events The only event considered by our system is the insertion of a message into a particular queue or class of queues. All kinds of events that are relevant for applications can be modeled as a message, due to the advanced queue types explained in Sec. 4.1. Hence, we do not need to burden our rule language with a complex event specification language, keeping it simple. Note that although rules cannot be created dynamically, it is still possible to attach rules to dynamically created queues: By associating a rule with a queue class instead of a particular queue, the rule will be triggered for all queues of that class, no matter how they were created.

4.2.2. Variables and Expressions Rule conditions and parameters of actions are specified based on variables. Variables are defined by binding them to expressions in declarative XML query languages. We use two languages, depending on the purpose of the variables: For variables that are utilized as action parameters (the second **define** clause), we use the full power of XQuery [8] to allow construction of result messages and access to other XML messages and data stored in the XDS. For condition variables (first **define** clause), we use XPath [5]. Here, efficiency is crucial, as a large number of potential rules have to be tested for every message, and XPath can be evaluated much more efficiently than XQuery.

XPath and XQuery expressions are evaluated with respect to a context. We evaluate the expressions with respect to the root node of the currently processed XML message. We further extend XPath and XQuery by user-defined functions such as `qs:queue('')`, to allow access to other messages in the XDS.

4.2.3. Condition The condition is represented as an XPath expression that may refer to any variables of the first define clause. Only if this expression is true with respect to the currently processed message, the second define clause is evaluated and the rule's action is executed.

4.2.4. Action Our rule language relies on a few basic actions. It provides actions to enqueue messages, to dynamically create and destroy queues, and delete messages from queues to prevent their processing. Actions have parameters, which are specified using XPath expressions that may refer to variables from any of the two define clauses.

The `enqueue` action inserts a message into a queue and takes as parameters the target queue name and the message body. The `delete` action deletes messages. The messages to be removed have to be supplied as a set-valued parameter. The `createQueue` action creates a new queue, specifying the queue name and queue class as parameters. Only regular queues (no gateway or echo queues) can be created this way. `destroyQueue` deletes a queue created by `createQueue` by specifying its name.

4.3. Examples

To illustrate how our rule language is used, we present sample rules for our pharmacy example from Sec. 2. Due to space constraints, we simplified the result messages generated by the rules.

4.3.1. Fundamental Messaging The following rule causes a welcome message to be sent to customers who send a message that contains a login action element in the message body. The rule assumes that messages enqueued to a queue whose name equals "reply" plus the username will cause the message to be sent to the remote user. The corresponding string is created by using XPath's *concat* function that returns a concatenation of its arguments [5].

```

on message into "hotline"
if //body/action="login"
define
    targetqueue as concat("reply",./body/username)
    contents as { <text>Welcome</text> }
do ENQUEUE($targetqueue, $contents)

```

4.3.2. Dynamic Queue Creation In our pharmacy, all customer requests are handled by the hotline queue. As explained in the previous example, there is supposed to be a message queue for the communication with each user. The following rule creates such a queue when a user registers, and specifies the queue class "userreply". Further rules in our application are associated with this class, and arrange for any enqueued messages to be annotated with metadata and forwarded to a gateway queue which sends the message.

```

on message into "hotline"
if //body/action="register"
define targetqueue as concat("reply",./body/username)
do CREATEQUEUE($targetqueue,"userreply")

```

4.3.3. Queue Browsing and Transformation This rule illustrates how queue browsing and result message construction can be implemented in our system. Whenever a customer decides to confirm an order, all items previously inserted into his shopping cart queue (represented by an "orders" queue) have to be forwarded to the packaging department. We can access the order queue using the `qs:queue()` function. Analogously, other data stored in the XDS could be accessed using the regular `fn:collection()` function of XQuery.

```

on message into "hotline"
if //body/action="confirmOrder"
define
  orderq as concat("orders",//body/username)
  orderid as //body/orderID
  items as
    {for $entry in qs:queue($orderq)//entries
    where $entry/@entryType='orderItem'
    and $entry/@orderID=$orderid
    return <item>$entry</item>}
  content as { <action> assembleOrder
    <items>$items</items> </action>}
do ENQUEUE("packaging", $content)

```

5. Related Work

Whenever asynchronous, message-based communication is required, queues are the data structure of choice for providing the necessary message storage and forwarding operations. Thus, queues are used in a vast number of messaging and middleware solutions. Except for their usual application as message stores, the usage of queues has also been proposed for a number of other tasks such as modeling recoverable state machines [23] in the context of middleware systems. Apart from messaging and middleware solutions, queues are also applied in other domains, for example as communication channels in a robust workflow management systems [20], showing their flexibility as universal data structures.

Based on their success in various fields of application and due to repeated statements from the database community saying that queues are interesting data structures [16] that should be incorporated into all distributed computing architectures generally [6] and into database systems in particular [17], major database vendors started to integrate queue structures and native messaging support into their systems.

Since version 8, the Oracle database provides native queue support [15] with a vast number of features, including message transformation based on XSLT and XPath [13]. It offers various interfaces such as the Java Message Service (JMS) or callback-enabled language bindings for imperative, high-level languages such as Java and Visual Basic. As seen above, these APIs require several lines of code to even execute fundamental messaging operations and make web service development unnecessarily complex.

Recently, other major database companies have started to integrate queuing solutions with their products. With SQL Server 2005, Microsoft introduces the Service Broker, a queue-based database extension for asynchronous messaging. A brief overview of the features and the numerous benefits received from a deep database integration is given by [25]. Researchers from IBM analyzed how to integrate IBM's WebSphere messaging solution and DB2 UDB

[10]. In contrast to the two other database systems described above and to our native implementation, they consider a full integration into the database to be a too complex task. Instead, they propose introducing a wrapper for converting the JMS objects used in WebSphere from and to relational data before processing them in the database, thereby suffering from an impedance mismatch and processing overhead for each message. Despite functional restrictions of their prototype, the authors already encounter severe limitations due to insufficient system integration, e.g. when browsing queues or handling side effects.

The concept of Event-Condition-Action (ECA) rules and its application in the context of active database systems is described in [24]. There are several proposals how ECA rules can be used in the context of XML processing systems. For example, Bailey et al [4] focus on providing a framework for keeping data consistent in case of document updates. Additionally, Bonifati et al [9] examine how ECA-based active rules can be used for communicating repository changes (e.g. by sending an SMS to a customer) and discuss potential rule management issues. However, these approaches do neither consider dependability aspects nor efficient data structures supporting rule evaluation.

Both the ActiveViews system [1] and the XL platform [12] share our approach to use database technology for building and optimizing the various aspects of a system for executing ecommerce applications such as web services. Developed on top of Java and an XML repository, ActiveViews [1] advocates the use of a declarative view specification language as well as the view and trigger concepts. These specifications are processed by an application generator, creating Java code for both the application logic and browser-based graphical user interfaces. ActiveViews fundamentally differs from our approach of integrating the application logic into the database core. Instead, it is a three-tiered system with the XDS as storage engine, a Java application server and a browser- and applet-based presentation layer.

The creators of the XL platform [12] avoid such impedance mismatches by using XML as the only primary data type. They develop a web service platform based on W3C standards and create the XL programming language. The expression language of XL is an extended version of XQuery that also allows for document updates. Additionally, it adopts elements of the Java syntax for auxiliary constructs such as exceptions and corresponding exception handlers. XL programs are executed by a corresponding runtime system, performing automatic optimizations e.g. by adopting techniques from compiler construction. However, as already described by the authors [12], important aspects such as dependability and transaction support are not considered in XL. Additionally, they observe that the proposed programming language

is only of limited declarativity, thus restricting the automatic optimization potential. Our approach of directly integrating the web service execution engine into a native XML database system allows us to provide full recovery and transaction support for both local and remote communication. Additionally, we believe that rule-based application languages are much more declarative, thus providing a greater potential for automatic optimizations.

6. Conclusion and Future Work

We derived the requirements and presented an architecture for a Web Service Engine (WSE). Our system design provides sophisticated and reliable XML messaging by enriching a native XML data store with transactional queues. To efficiently implement business applications, we introduced a fully declarative, ECA rule-based language customized to XML message processing. We provided several examples to illustrate how such application rules can be used to model various aspects of a business scenario.

There is still much work to be done to achieve our goal of building a high-performance and dependable web service engine. One of the most interesting and challenging research opportunities is the mapping of rule evaluation to transactions. Additionally, we want to investigate optimization and translation of our rule language in order to allow for high-performance parallel processing and efficient internal representation.

Another important issue is the interaction of our low-level, rule-based application language with high-level business process languages such as e.g. BPEL [3], and whether it is possible to automatically convert application representations from one language to another.

References

- [1] S. Abiteboul, B. Amann, S. Cluet, A. Eyal, L. Mignet, and T. Milo. Active views for electronic commerce. In *VLDB*, pages 138–149, 1999.
- [2] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer-Verlag, 2004.
- [3] T. Andrews, F. Curbera, H. Dholakia, Y. Golland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for web services version 1.1. Technical report, 2003. <http://ifr.sap.com/bpel4ws/index.html>.
- [4] J. Bailey, A. Poulouvasilis, and P. Wood. An event-condition-action language for xml. In *WWW*, pages 486–495, 2002.
- [5] A. Berglund, S. Boag, D. Chamberlin, M. Fernandez, M. Kay, J. Robie, and J. Siméon. XML path language (XPath) version 2.0. Technical report, World Wide Web Consortium (W3C), 2005.
- [6] P. Bernstein, M. Hsu, and B. Mann. Implementing recoverable requests using queues. In *SIGMOD Conference*, pages 112–122, 1990.
- [7] K. S. Beyer, R. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G. M. Lohman, B. Lyle, F. Ozcan, H. Pirahesh, N. Seemann, T. C. Truong, B. Van der Linden, B. Vickery, and C. Zhang. System RX: One Part Relational, One Part XML. In *Proc. SIGMOD Conference*, pages 347–358, 2005.
- [8] S. Boag, D. Chamberlin, M. Fernández, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML query language. Technical report, World Wide Web Consortium, 2005.
- [9] A. Bonifati, S. Ceri, and S. Paraboschi. Active rules for XML: A new paradigm for e-services. *VLDB J.*, 10(1):39–47, 2001.
- [10] S. Doraiswamy, M. Altinel, L. Shrinivas, S. Palmer, F. Parr, B. Reinwald, and C. Mohan. Reweaving the tapestry: Integrating database and messaging systems in the wake of new middleware technologies. In *Data Management in a Connected World*, pages 91–110, 2005.
- [11] T. Fiebig, S. Helmer, C-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a native XML base management system. *VLDB Journal*, 11(4):292–314, 2002.
- [12] D. Florescu, A. Grünhagen, and D. Kossmann. XL: a platform for web services. In *CIDR*, 2003.
- [13] C. Foch. Oracle streams advanced queuing user’s guide and reference, 10g release 2 (10.2), 2005.
- [14] H. Garcia-Molina and K. Salem. Sagas. In Umeshwar Dayal and Irving L. Traiger, editors, *Proc. SIGMOD*, pages 249–259, 1987.
- [15] D. Gawlick and S. Mishra. Information sharing with the oracle database. In *DEBS*, 2003.
- [16] J. Gray. Queues are databases. In *Proceedings 7th High Performance Transaction Processing Workshop. Asilomar CA, Sept 1995.*, 1995.
- [17] J. Gray. The next database revolution. In *SIGMOD 2004*, pages 1–4, New York, NY, USA, 2004.
- [18] M. Gudgin, M. Hadley, N. Mendelsohn, J. Moreau, and H. Nielsen. SOAP version 1.2. Technical report, The World Wide Web Consortium (W3C), 2003.
- [19] Charles T. Davies Jr. Data processing spheres of control. *IBM Systems Journal*, 17(2):179–198, 1978.
- [20] F. Leymann and D. Roller. Building a robust workflow management system with persistent queues and stored procedures. In *ICDE*, pages 254–258, 1998.
- [21] C. Mohan, H. Strong, and S. Finkelstein. Method for distributed transaction commit and recovery using byzantine agreement within clusters of processors. *Operating Systems Review*, 19(3):29–43, 1985.
- [22] M. Stonebraker. Too much middleware. *SIGMOD Rec.*, 31(1):97–106, 2002.
- [23] S. Tai, A. Totok, T. Mikalsen, and I. Rouvellou. Message queuing patterns for middleware-mediated transactions. In *SEM*, pages 174–186, 2002.
- [24] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996.
- [25] R. Wolter. An introduction to SQL server service broker. Technical report, Microsoft, 2005.