Reihe Informatik
006 / 2009

Messaging Rules as a Programming Model for Enterprise Application Integration

Alexander Böhm Carl-Christian Kanne

University of Mannheim
alex|cc@db.informatik.uni-mannheim.de

Messaging Rules as a Programming Model for Enterprise Application Integration

Alexander Böhm and Carl-Christian Kanne

University of Mannheim, Germany
alex|cc@db.informatik.uni-mannheim.de

Abstract. Rule-based systems and languages are successful in many application areas such as business rules or active database systems. The goal of the Demaq project is to investigate the feasibility and benefits of using a declarative, rule-based programming language to simplify the development of complex, distributed applications. For this purpose, we propose a novel programming paradigm based on messaging, queues and declarative rules.
We focus on evaluating whether the proposed, rule-based approach can be used to implement complex application patterns. We use Enterprise Application Integration (EAI) as an example application domain, as EAI applications involve multiple, heterogeneous systems with complex interaction patterns [12]. We discuss whether and how these application patterns can be implemented using our rule language.

1 Introduction

Today, distributed systems are implemented using imperative programming languages, such as Java or C#, and executed by multi-tiered application servers [2]. To facilitate application development, rule-based languages have been proposed to simplify various aspects of these complex processing systems. This includes active rules for database systems [13], dynamically controlling application aspects using business rules [10, 14], or implementing basic message filtering and forwarding tasks in message broker components [11]. These languages help to simplify the implementation of individual aspects of these systems using declarative, rule-based facilities. However, due to the various different languages, heterogeneous programming models and runtime systems involved, the overall complexity of application development remains high. This reduces the productivity of programmers and may also have a significant impact on the runtime performance as applications are difficult to implement, maintain and optimize [15].

The Demaq project [4] aims at simplifying the development of complex messaging-based applications by using a novel programming model. Our focus is on describing the complete business logic of a distributed application by using exclusively a rule-based programming language.

In this paper, our goal is to demonstrate the feasibility of using a declarative, closed rule language for the implementation of complex, distributed applications. For this purpose, we analyze whether the typical processing patterns

characteristic for Enterprise Application Integration (EAI) can be implemented with our proposed approach. Choosing EAI as an example application domain is particularly appealing, as EAI applications involve complex interactions among several heterogeneous and distributed systems that are difficult to implement. These processing patterns characteristic for this kind of applications have been identified and systematically discussed in the literature [12].

The remainder of this paper is organized as follows. In Section 2 we introduce our programming model that allows to implement the business logic of distributed applications based on message queues and declarative application rules. Section 3 reviews the typical processing patterns in EAI applications and discusses whether and how these patterns can be implemented using our rule language. We conclude the paper in Section 4 and give an outlook to our future work.

2 Rule-Based Programming Model

Our approach is based on a simple, rule-based programming model that allows to implement the business logic of a node participating in a distributed application. It consists of four major components.

1. *Messages* are used to exchange data with remote communication partners and for representing node-internal, intermediate state.
2. *Message properties* allow to annotate messages with additional metadata.
3. *Message queues* provide asynchronous communication facilities and allow for transactional, reliable and persistent message storage.
4. *Declarative rules* operate on these queues and the messages stored within them and are used to implement the application logic. Every rule specifies how to react to an arriving message by creating resulting messages.

Unfortunately, due to space constraints, we cannot discuss all details of our programming model. Instead, we give a brief overview of the key components and concepts, and refer the interested reader to [4] for an in-depth discussion, including design and performance aspects of the Demaq rule execution engine.

2.1 Message Queues

An application in our model is based on an infrastructure of message queues. For this purpose, it incorporates two different types of message queues. *Gateway queues* provide incoming and outgoing interfaces for message exchange with remote systems. *Basic queues* are used for local, persistent message storage and to pass intermediate information between application rules.

All messages in our model exclusively use XML as the underlying data format. This refers to the messages exchanged with external systems, as well as to local messages sent between the queues of an application. Having XML as an extensible and expressive message format facilitates to interact with all kinds of remote services, while a uniform data format for all data avoids performance-consuming representation changes.

Example 1. This example demonstrates how to create the infrastructure of message queues that underlie an application. First, a incoming gateway queue `incomingMessages` is created in line 1. This queue can be used to exchange messages with remote systems using HTTP as underlying transport. As HTTP is a synchronous transport protocol, the gateway queue is associated with a corresponding `response` queue. All messages inserted into this `response` queue will be sent as synchronous replies to incoming requests.

Additionally, two local queues are created in lines 3 and 4. These queues are used for local message forwarding and storage. The `transient` mode indicates that no persistence guarantees need to be given, while a `persistent` queue mode requires messages to be recovered in case of application or system errors.

```
1 create queue incomingMessages kind incoming interface "http" port "2342"
2   response outgoingMessages mode persistent;
3 create queue customerCare kind basic mode transient;
4 create queue customerOrders kind basic mode persistent;
```

2.2 Message Properties

Every message in our system is an XML fragment that was either received from an external source or generated by a local application rule. Apart from their XML payload, messages can be associated with additional metadata annotations that are kept separate from the XML payload. These *properties* are key/value pairs, with unique names as their key and a typed, atomic value. Apart from setting properties in application rules, several properties are provided by the runtime system, such as creation timestamps or transport protocol information.

2.3 Declarative Application Rules

In our programming model, the complete business logic of a distributed application is implemented using declarative rules operating on message queues. Conceptually, every application rule is an Event-Condition-Action (ECA) rule [13] that reacts to messages by means of creating new messages.

Event Our rules react to a single kind of event, which is the arrival of a message at a queue of the application. These messages may either be received from external communication endpoints or result from the execution of another, local application rule. To keep our rule language simple and comprehensible to application developers, the insertion of a message in a queue of the system is the *only* event type that is supported. Other event sources, such as timeouts or various kinds of error notifications [16], are translated to corresponding messages that can be handled by application rules.

Condition Instead of developing a novel expression language from scratch, our approach builds on the existing declarative XML query language XQuery [3] and the XQuery Update Facility [8], which enables XQuery expressions to perform side-effects. This approach is inspired by other rule languages (e.g. [1, 6, 7]) that

have successfully been built on the foundation of XML query languages. Consequentially, in our model all rule conditions and other business logic are described using XQuery expressions that are evaluated with the triggering message as the context item [3].

Action We restrict the set of resulting actions that a rule may produce to a single kind of action, which is to enqueue a message into a queue of the application. This restriction keeps our rule language closed, i.e. all actions produced by application rules can be directly reacted on by other rules. At the same time, it still allows application rules to implement the message flow within the local queues of an application and to external systems using gateway queues.

To express the messaging actions resulting from rule execution, we have extended the XQuery Update Facility with an additional `enqueue message` updating expression.

Example 2. Below, we show an example of a simple application rule that performs content-based message forwarding. The rule definition expression in line 1 is used to create a new rule (named `exampleRule`) which handles messages enqueued into the `incomingMessages` queue. In the rule body (lines 2 to 12), XQuery expressions are used to analyze the structure of the incoming message using path expressions (lines 5 and 8) and to forward the message to an appropriate queue for further processing using the `enqueue message` updating expression. When a message with an unexpected structure is encountered, an error notification is sent back to the sender (line 12).

```
1  create rule exampleRule for incomingMessages
2  let $request := .
3  return
4  (: dispatch message to appropriate destination – message dispatcher :)
5    if($request//complaint)
6    then
7      enqueue message $request into customerCare
8    else if($request//order)
9    then
10     enqueue message $request into customerOrders
11   else
12     enqueue message <error>...</error> into outgoingMessages;
```

2.4 Persistent State Management

Our programming model achieves data persistence by storing the complete *message history*. All messages received from and sent to external systems are stored persistently in the queues of an application. Thus, queues are not only used as staging areas for incoming and outgoing messages, but also serve as durable storage containers. In our model, there are no auxiliary runtime contexts or other constructs for maintaining state. Instead, message queues are the only way to persistently store state information in the form of messages.

Application rules may recover state information by querying the message history. For complex, state-dependent applications, queries to the message history are a frequent operation. To simplify this reoccurring task of history access,

our rule language incorporates the concept of slicings, which define application-specific *views* to the message history. Slicings allow developers to declaratively specify which parts of the message history are relevant for application rules, and to access them by using a simple function call.

Example 3. In this example, we illustrate how slicings can be used to organize and access the message history. The slicing definition expression (lines 1 and 2) defines a new slicing with name `customerOrdersByID` for the messages in the `customerOrders` queue. For each distinct `customerID`, a separate slice will be created. Each slice contains all messages that share the same key, which is the `customerID` in this example. The `require` expression is used to indicate that only the last ten messages for each slice need to be preserved. Older messages may be safely removed from the message history using the garbage collection facilities of the rule execution engine [4].

Application rules may access an individual slice using the `slice` function call. In the example below, it is used by the rule to access all messages for a particular customer (line 6). Depending on the number of items a customer has ordered, the message is either forwarded to the `importantCustomers` queue or handled locally.

```
1  create slicing property customerOrdersByID
2    queue customerOrders value //customerID require count(history()) eq 10;
3
4  create rule checkCustomerImportance for customerCare
5  let $customerID := //customerID
6  let $ordersForCustomer := slice($customerID, "customerOrdersByID")
7  let $orderedItems := count($ordersForCustomer//item)
8  return
9    if($orderedItems gt 20)
10   then enqueue message . into importantCustomers
11   else  ... (: handle locally:) ;
```

3 Implementing EAI Patterns

Our rule-based programming model aims at simplifying the development of complex, distributed applications by describing their business logic by means of message queues and declarative application rules. To verify the practical feasibility and benefits of such a programming model, Enterprise Application Integration (EAI) applications are of particular interest.

The goal of Enterprise Application Integration (EAI) is to integrate several applications and computer systems, which may be of heterogeneous architectures and distributed across multiple sites, into a single, combined processing system. Typically, the involved components are integrated using messaging. In practice, the task of application integration may become arbitrarily complex, depending on the kinds, numbers and peculiarities of the systems involved.

The various characteristic messaging patterns that evolve in EAI architectures have been identified in the reference work of Hohpe and Woolf [12]. The resulting library of patterns can be categorized into six distinct classes that refer to the use of various messaging protocols, encodings and transport endpoints, message construction, transformation as well as message routing and analysis.

In the following sections, we briefly review these patterns and discuss whether and how they can be implemented using our rule language. We use italics (*pattern*) to refer to the individual patterns [12].

3.1 Messaging Endpoints

Typically, EAI involves accessing and interacting with a multitude of heterogeneous systems. The way in which messages are exchanged heavily depends on the individual systems involved. Interaction styles include *polling consumers* that actively pull messages from remote systems, or *event-driven consumers* that are triggered by external event sources.

In our model, these different styles of *messaging gateways* are implemented using gateway queues. Incoming gateway queues allow to receive notification messages from external systems, while outgoing gateways allow to send data to other systems, and to actively poll them for new data.

Once a message has been received, it is handled by *message dispatchers* that forward messages to the appropriate destination, or by *selective consumers* that only react to particular types of messages. In our model, these patterns can be realized using corresponding path expressions. In the example below, the `consumer` rule implements a *selective consumer* that only reacts to order messages.

Some patterns do not need to be manually implemented by developers, but are instead automatically provided by our processing model [4]. It includes strong transactional guarantees for rule processing (which subsume the *transactional client* pattern) and allows multiple concurrent execution threads (*competing consumers*) for a single message queue.

The *durable subscriber* pattern, which avoids losing messages while not actively processing messages for a particular queue can be implemented by using a `persistent` queue mode (as in line 3 of the example below).

```
1 (: gateway queue - messaging gateway:)
2 create queue incomingMessages kind incoming interface "smtp" port "25"
3   mode persistent;      (: - transactional client:)
4 create rule consumer on incomingMessages
5 if(//order)     (: - selective consumer :)
6 then ...
7 else ();
```

3.2 Different Message Types and Message Construction

Depending on the involved systems, there are various types of messages that have to be handled by an EAI application. This includes *command messages* that reflect remote procedure calls (RPC), *document messages*, which are used for data transfer, or *event messages*, that inform an application of the occurrence of a particular event. As our rule language is based on XML as the underlying message format, these various message styles can be easily created by choosing an appropriate XML schema. Moreover, messages can be easily annotated with a *format indicator*, identifying the XML schema a message conforms to.

Apart from the message payload, messages are associated with additional, transport-related metadata. This includes *return addresses* for asynchronous

transports and auxiliary transport protocol information for the *request-reply* pattern reflecting synchronous protocols that require resulting messages to be sent over the same connection (e.g. socket) as the initial request. Moreover, *correlation identifiers* can be used to associate messages with other, related ones (e.g. all messages belonging to the same transaction). In our programming model, all these patterns are conveniently handled using (system-provided) message properties. This includes the *return address* (line 11 in the example below) and system-provided *correlation identifiers* for synchronous transports (line 13).

In our model, advanced message properties such as *message expiration*, that requires that a message is only valid as long as a particular condition holds, can be modeled using declarative message retention facilities. In the example below, the `require` expression is used to indicate that only the last message in the `notifications` queue should be retained (line 3).

```
1  (: only preserve last notification received - message expiration :)
2  create slicing property lastNotification
3    queue notifications require count(history()) eq 1;
4
5  (: synchronous gateway queue - request reply:)
6  create queue incomingMessages kind incoming interface "http" port "80"
7    response outgoingMessages mode persistent;
8
9  create rule sendReply for incomingMessages
10 (: retrieving sender address using property - return address :)
11 let $sender := property("comm:From") (:not used any further:)
12 (: retrieving system-managed correlation identifier:)
13 let $correlationID := property("comm:CorrelationID") (:not used any further:)
14 (: invoking an external service as a reply - command message:)
15 let $result := <rpc:updateQuantity xmlns:rpc="http://www.example.com">
16    <rpc:Arguments count="2">
17      <rpc:argument name="itemID" type="integer">{//itemID/text()}</rpc:argument>
18      <rpc:argument name="quantity" type="float">{//quantity/text()}</rpc:argument>
19    <rpc:Arguments>
20 </rpc:updateQuantity>
21 enqueue message $result into outgoingMessages;
```

3.3 Message Routing

Message routing patterns describe the various styles of message flow between the components of an application. The most basic form of message routing is to sequentially forward a message from one processing step to the next, thus forming a processing chain. In our rule language, this *pipes and filters* pattern corresponds to a message being forwarded from one queue to another, with individual rules implementing the processing steps as in the example below.

Instead of simply forwarding messages, *content-based routers* can be used to send messages to an appropriate processing step based on their payload. The rule in lines 7-10 of the example below implements this pattern. It forwards all order messages to the `ordersQueue` and enqueues all other messages to another queue for further analysis.

Other basic message routing patterns include *message filters*, which filter out unnecessary messages from a message stream, or *splitters*, which split a message into individual parts and forward them to separate consumers (line 14ff in the example below). *Aggregators* can be used to combine multiple messages to a single, large one. Line 17 in the example below implements a message aggregator

that combines several messages from the message history into a single message. Here, the implementation of the *aggregator* pattern is greatly simplified by the sequence-oriented data model of XQuery underlying our application rules.

In contrast to the basic message routing patterns discussed above, the *process manager* is a general-purpose pattern that represents complex message routing operations. Process managers are e.g. required when multiple messages should be created in a particular sequence or when performing other, context-dependent operations that cannot be expressed with basic patterns. Line 20ff of the example below shows the implementation of a simple process manager that routes three messages to destination queues in a particular sequence.

```
1  (: add timestamp and forward to next processing step - pipes and filters:)
2  create rule addTimeStamp for incomingMessages
3  let $result := <result><timestamp>{fn:current-dateTime()}</timestamp>{.}</result>
4  return enqueue message $result into nextStep;
5
6  (: forward message to appropriate rule - content based router:)
7  create rule contentBasedRouter for nextStep
8  if(//orderMessage)
9  then enqueue message . into ordersQueue
10 else enqueue message . into anotherQueue ;
11
12 create rule simplePM for anotherQueue
13 (: split input message into two parts - splitter :)
14 let $firstMessage := //part1
15 let $secondMessage := //part2
16 (: combine all order messages into a large one - aggregator :)
17 let $thirdMessage := <orders>{slice(//customerID, "ordersByCustomerID)}</orders>
18 return (
19 (: generating sequence of messages - simple process manager :)
20   enqueue message $firstMessage into someQueue,
21   enqueue message $secondMessage into anotherQueue,
22   enqueue message $thirdMessage into anotherQueue );
```

3.4 Message Transformation

Message transformation patterns describe how both structure and content of messages may be modified and adapted by EAI applications. This includes *message translators* that transcode messages from one format to another, for example by converting a list of comma-separated values into XML format. As XML is the only data format in our rule language, message translators are limited to performing schema-to-schema transformations (i.e. translating from one XML schema to another).

Normalizers are used to unify several different incoming messages to a canonical format. In our model, they can be easily implemented by defining corresponding message translators and incoming gateway queues, that convert the incoming messages into the expected schema and enqueue them to the same queue for further processing. The purpose of an *envelope wrapper* is to wrap a message into a metadata-carrying envelope. In the example below (line 14ff), we implement a simple envelope wrapper that encloses a message into a SOAP envelope. *Content enrichers* are used to imbue a message with additional information that cannot be found in the input message. In our example, the `slice` history access function is used to retrieve the address for a customer from the master data stored in the queues of the system (line 7). In contrast to the content enricher, the *content*

filter is used to strip unnecessary content from a message. In the example rule, this is done by using a path expression that excludes anything but the `item` elements from the initial message. By combining the content enricher and filter patterns with message history access operations, the *claim check* pattern, which temporarily removes (e.g. sensitive) message parts, can be easily implemented.

```
1  create rule prepareMessage for prepareOutgoing
2  let $initialMessage := .
3  (:translate customer info - message translator:)
4  let $message := <result>
5  <custName>{concat(//customer/fName/text(), //customer/lName/text())}</custName>
6  <!-- enrich with master data - content enricher / claim check -->
7  <address>{slice(//customerID/text(), "customersByID")/address/*}</address>
8  <!-- filter out unnecessary information - content filter -->
9  <orderedItems>{$initialMessage//item}</orderedItems>
10 </result>
11 (:add SOAP envelope - envelope wrapper:)
12 let $env := <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
13     <soap:Header/>
14     <soap:Body>{$message}</soap:Body>
15   </soap:Envelope>
16 return enqueue message $env into outgoingMessages;
```

3.5 Messaging Channels

Messaging channel patterns describe various message transports that can be used to disseminate the messages in an EAI application. In our model, these various *message channels* are provided by corresponding gateway queues. This includes *point-to-point messaging*, which is implemented by using synchronous or asynchronous gateway queues or *publish-subscribe channels* which are implemented by gateway queues and message history access to retrieve the list of subscribers (as in lines 12-15 of the example below). Similar to the *publish-subscribe channels*, a *message bus* can be implemented with a combination of gateway queues and history access to identify the systems connected to the message bus.

In our model, special purpose channels such as the *dead-letter channel* containing messages that could not be sent to remote systems or the *invalid message channel* for messages that could not be handled (e.g. due to an unexpected schema) are reflected by corresponding queues. Whenever message processing or validation fails, a corresponding error notification message is sent to these *error queues*, allowing developers to react to the error by means of compensating application rules. In the example below (line 8), such an error queue is assigned to an application rule. Thus, all messaging errors that are encountered when processing this rule will be reflected by error notifications sent to the `transportFailures` error queue, where they can be handled by corresponding rules.

Finally, guaranteed message delivery (as required for the *guaranteed delivery channel* pattern) that persists messages to guarantee delivery even in case of system failures, can be provided by requesting a `persistent` queue mode (as in line 1 of the example below).

```
1  create queue sendToSubscribers kind basic mode persistent;
2
3  (: guaranteed delivery using persistent queue mode:)
4  create queue outgoingMessages kind outgoing interface "smtp" port "25"
5    mode persistent;
```

```
6
7  (:errorqueue - dead-letter channel and invalid message channel:)
8  create rule pubsub for sendToSubscribers errorqueue transportFailures
9  let $payload := //payload
10 let $topic := //topic
11 let $subscribers := slice($topic, "subscribersByTopic")
12 (: forward message to all subscribers - pub/sub :)
13 for $address in $subscribers/address/email
14 return enqueue message $payload into outgoingMessages
15    with comm:To value $address (: set SMTP parameters :)
16    with comm:Subject value "Subscription notification";
```

3.6 System Management

Typically, EAI applications involve complex, distributed setups that involve various heterogeneous systems and applications. Hence, developers have to be provided with corresponding monitoring and debugging facilities that help to understand the behavior of such a complex system, both during the software development process and when operating in production. Various pattern have been identified that can be used for *monitoring* EAI applications. These include having a *control bus*, that provides an integrated channel for system diagnosis information and the possibility to *wire tap* the message flow between the involved systems. The *message history* provides lineage information for each individual message, allowing to track and (recursively) identify the components a particular message results from.

Apart from supporting the monitoring of EAI applications, several system facilities have been identified that simplify the application development. This includes a *message store*, that allows to persistently store messages, that can thus be referred to by other messages. This pattern is an integral part of our programming model, which exclusively model application state in terms of the messages stored in the queues of the system. Another system facility are *smart proxies* that allow for automatic message correlation. This task is performed by synchronous gateway queues in our model.

Finally, the *test message* pattern simplifies application debugging by allowing developers to inject test messages and see how system reacts. In our language, this pattern can be implemented by using an additional gateway queue and forwarding rule as in the example below.

```
1  (: - wiretap:)
2  create queue wiretapGateway kind outgoing interface "smtp://loghost/" port "25"
3    mode transient;
4  create rule wiretap for someQueue
5  enqueue message . into wiretapGateway;
6
7  (: - test message :)
8  create queue testmessages kind incoming interface "smtp" port "4000"
9    mode transient;
10
11 create rule forwardTestmessage for testmessages
12 let $targetQueue := //targetqueue/text()
13 return enqueue message . into $targetQueue;
```

To safe developers from manually including management facilities into their applications, the Demaq runtime system includes an interactive debugger and a

corresponding visual interface [5]. The debugger provides convenient, message-based access to the message store, the message history and runtime system statistics. It allows to display the content of messages and queues, inject arbitrary messages and incorporates sophisticated application analysis features such as break- and watchpoints, thus facilitating system and application management. Hence, the debugger provides developers with instances of the *control bus, message history, wire tap, message store* and *test message* pattern without requiring any auxiliary application code.

4 Conclusion

We have discussed a novel programming model that allows to implement the business logic of complex, distributed applications using message queues and declarative application rules.

Using the typical, complex application patterns from Enterprise Application Integration (EAI) as an example domain, we have illustrated the practical feasibility of using exclusively a rule-based language for the development of complex messaging applications. We have demonstrated how various patterns can be implemented in our rule language. Surprisingly, even complex messaging patterns that require sophisticated development in today's EAI solutions [12] could often be implemented with only a few lines of code.

The focus of this paper was to systematically evaluate our rule-based approach in the light of typical patterns occurring in the context of EAI applications. Apart from investigating these patterns in isolation, we have also verified the applicability of our language to implement complete, distributed applications from various domains, including several distributed and workflow applications. These applications and the Demaq rule execution engine are freely available at [9] under an open-source license.

References

1. Serge Abiteboul, Bernd Amann, Sophie Cluet, Adi Eyal, Laurent Mignet, and Tova Milo. Active views for electronic commerce. In *VLDB*, pages 138–149, 1999.
2. Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architectures and Applictions*. Springer-Verlag, 2004.
3. Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML query language. Technical report, W3C, January 2007.
4. Alexander Böhm and Carl-Christian Kanne. Processes are data: A programming model for distributed applications. In *Tenth International Conference on Web Information Systems Engineering (WISE), Poznan, Poland.*, 2009.
5. Alexander Böhm, Erich Marth, and Carl-Christian Kanne. The Demaq system: declarative development of distributed applications. In *SIGMOD Conference*, pages 1311–1314, 2008.
6. Angela Bonifati, Stefano Ceri, and Stefano Paraboschi. Pushing reactive services to XML repositories using active rules. *Computer Networks*, 39(5):645–660, 2002.

7. François Bry and Paula-Lavinia Patranjan. Reactivity on the web: paradigms and applications of the language XChange. In Hisham Haddad, Lorie M. Liebrock, Andrea Omicini, and Roger L. Wainwright, editors, *SAC*, pages 1645–1649. ACM, 2005.
8. Don Chamberlin, Daniela Florescu, Jim Melton, Jonathan Robie, and Jérôme Siméon. XQuery Update Facility 1.0. Technical report, W3C, August 2007.
9. Demaq project homepage, June 2009. http://www.demaq.net/.
10. Jens Dietrich. A rule-based system for ecommerce applications. In Mircea Gh. Negoita, Robert J. Howlett, and Lakhmi C. Jain, editors, *KES*, volume 3213 of *Lecture Notes in Computer Science*, pages 455–463. Springer, 2004.
11. Craig B. Foch. Oracle streams advanced queuing user's guide and reference, 10g release 2 (10.2), 2005.
12. Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns*. Pearson Education, Inc., 2004.
13. Norman W. Paton and Oscar Díaz. Active database systems. *ACM Computing Surveys*, 31(1):63–103, 1999.
14. Mark Proctor. Relational declarative programming with JBoss Drools. In Viorel Negru, Tudor Jebelean, Dana Petcu, and Daniela Zaharie, editors, *SYNASC*, page 5. IEEE Computer Society, 2007.
15. Michael Stonebraker. Too much middleware. *SIGMOD Record*, 31(1):97–106, 2002.
16. Jim Waldo, Geoff Wyant, Ann Wollrath, and Samuel C. Kendall. A note on distributed computing. In Jan Vitek and Christian F. Tschudin, editors, *Mobile Object Systems*, volume 1222 of *Lecture Notes in Computer Science*, pages 49–64. Springer, 1996.