VIDRE – A Distributed Service-Oriented Business Rule Engine based on RuleML

Christoph Nagl, Florian Rosenberg, Schahram Dustdar
VitaLab, Distributed Systems Group, Information Systems Institute
Vienna University of Technology
1040 Vienna, Argentinierstrasse 8/184-1, Austria
{nagl, florian, dustdar}@infosys.tuwien.ac.at

Abstract

Business rules provide an elegant solution to manage dynamic business logic by separating business knowledge from its implementation logic. The drawback of most existing business rule approaches is the lack of standardization and interoperability. The lack of service-orientation and remote accessibility of business rule engines makes it hard to use business rules in distributed environments. This paper contributes the design and implementation of VIDRE *(Vienna Distributed Rules Engine), a service-oriented business rule engine based on RuleML.* VIDRE *enables enterprise applications to access business rules as easy as accessing a database, by exposing rules as Web services.* VIDRE *uses RuleML as an interlingua to represent facts, rules, and queries. One of the main contributions of the* VIDRE *approach is the ability to distribute rules and facts across various rule engines, therefore, enabling powerful ways of separating and executing business rules within intra- and interorganizational boundaries.*

1. Introduction

Today's businesses are changing rapidly and organizations have to deal with the dynamic changes of market economics. Such competitive business environments require business applications being flexible and adaptive enough, to meet frequent changes in business conditions and business policies [18]. Software applications have to keep track with changing requirements, thus, responding to market changes rapidly and effectively is a key factor for success.

Several industrial domains (e.g., supply-chain management) are highly distributed. This organizational structure is also reflected in the IT infrastructure, where reliable distributed systems are necessary for running the business. Many companies use business rule engines as one part of the IT systems to manage their dynamic business policies and rules. Business rules technology induces a paradigm shift in business application development by separating business logic from the application code. They explicitly specify policies and knowledge, hence, they facilitate maintenance and adaption of enterprise applications.

Besides the use of business rules, enterprise applications increasingly rely on the principles of the service-oriented architecture (SOA) to realize standardized interfaces for providing their software services within and across organizational boundaries. Combining business rules with a service-oriented architecture can help to develop more flexible and adaptive enterprise applications.

In this paper, we argue that current business rule approaches lack a sophisticated way to be integrated within SOAs due to a number of current shortcomings.

The first problem is the lack of a remote interface to allow a remote rule invocation from distributed applications. Most rule engines are currently used as a library which is linked to the enterprise applications. The rules are typically invoked via proprietary interfaces provided by the rule engines (e.g., rules for calculating the discount of a customer). The accessibility and reusabilty of these rules is limited to local applications (running on the same machine) although other enterprise applications may need this set of rules for achieving a task.

The second problem is the lack of standardization of a rule representation language. Each rule engine vendor proposes a custom format for representing their rules, thus a number of different dialects have emerged. This diversity makes it hard to use heterogeneous rule engines in the same enterprise environment or switch to another rule engine. It also hampers using rule engines in distributed environments where other applications and services can use the rule engine as a computational entity.

The third problem concerns the execution of business rules. A remote communication facility among rule engines would allow to retrieve data from other rule engines to perform the inference process. We refer to such a process as *distributed rule execution*. Currently, the lack of such a distributed rule execution can be seen as a consequence of the first two problems.

This paper describes the design and implementation of VIDRE[1], a business rule engine with several novel features to address the aforementioned drawbacks of existing approaches: Firstly, we combine the rule-driven approach with SOA to allow an easy integration into client applications and make it easy for applications to consume business rule services (i.e., business rules exposed as Web service). Secondly, VIDRE tries to overcome the interoperability problem of current rule engines by providing a lightweight plug-in mechanism which allows exchanging business rule engines being completely transparent for client applications. Furthermore, we use RuleML [13], a current standardization effort for rule markup, as an interlingua for exchanging and representing business among several heterogeneous rule engines. Thirdly, we present VIDRE's opportunity to execute distributed business rules which allows the execution of rules over several rule engines. Such a distributed execution is highly relevant in environments, where rules are distributed among several rule engines.

This paper is organized as follows: Section 2 introduces some basic concepts which are needed for the VIDRE approach. Section 3 describes the design and implementation aspects of VIDRE. In Section 4, we evaluate and discuss our approach by applying it to an example from the supply-chain management domain. In Section 5 we briefly discuss the related work in this area and in Section 6 we conclude this work and highlight some future work.

2. Background

In this section we briefly introduce some concepts technologies needed for the VIDRE approach.

Business Rules. The Business Rules Group [17] defines a business rule as a statement that defines or constrains some aspects of business, and it is intended to assert business structure or to control or influence the behavior of the business. The business rules approach distinguishes between terms (definitions), facts (connection between terms) and rules (constraints, derivation or reaction rules). Terms and facts express business knowledge (e.g., customer age is $\geq$ 18) whereas rules are used to guide the decision making, derive new information (derivation rules) or to trigger actions based on certain conditions (reaction rules). Additionally, queries can be formulated to retrieve the desired information from the knowledge base.

[1]A demo is available at `http://www.vitalab.tuwien.ac.at/projects/vidre`.

The usage of business rules is reasonable in knowledge and decision intensive domains such as insurance, supply-chain management or finance. They can be used to specify domain knowledge and manage it independently of the enterprise application.

Rule Engine Components. A typical rule engine consists of several components: a *rule base* contains all the rules for execution. The *working memory* holds the data on which the rule engine operates. The *Pattern Matcher* decides which rules from the rule base apply, given the content of the working memory. An *inference engine* works in discrete cycles and is used to find out which rules should be activated in the current cycle (including the activated ones from previous cycles) by using the pattern matcher. All activated rules from the conflict set are ordered to form the *agenda* (the list whose right hand side will be executed). The process of ordering the agenda is called the *conflict resolution*. To complete a cycle, the first rule on the agenda is fired and the entire process is repeated. Typical rule engines either implement a forward or backward-chaining strategy. *Backward chaining* works backward from a conclusion to be proven to determine if there are facts in the working memory to prove the truth of the conclusion [16]. *Forward chaining* takes all the facts stored in the working memory and attempts to apply as many rules as possible. The inference engine works from the initial content of the workspace towards the final conclusion [16].

Rule Markup Initiative. The Rule Markup Initiative aims to provide a standard rule language and to provide an interoperability platform integrating various business rule languages, inference systems, and knowledge representation paradigms. It has gained increasing momentum within the standards, academic, and industrial communities [13]. The RuleML language offers XML syntax for rules knowledge representation, interoperable among major commercial and non-commercial rule systems. We make extensive use of RuleML as rule exchange format among several heterogeneous business rule engines by using it similar to SOAP in the Web service domain. Every request and response to and from a rule engine at each service provider is encoded in RuleML. An example is presented in the next section.

Java Rule Engine API. The Java Rule Engine API – specified by the Java Specification Request (JSR) 94 – is a Java community effort defining an API for rule engines in the Java world [7]. The JSR 94 specification standardizes a set of fundamental rule engine operations. This operations include parsing rule sets, adding objects to the inference process, firing rules and getting objects from the engine as a result. The main goal of the JSR 94 specification

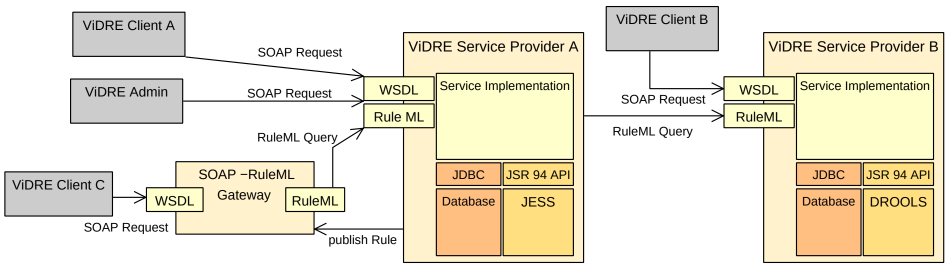

Figure 1. Architecture of VIDRE

is to integrate different rule engines with a simple adapter into client applications without settling with one rule engine vendor. The specification provides two main parts, a rule administration API and a rule runtime API. The administration interface is primarily used for loading and managing rule sets (a collection of rules for solving a specific task). The runtime API is for executing specific rule sets by using a rule session, which represents the connection between the client and a specific rule engine. The main critique of rule engine vendors is the lack of a standard rule representation format. In Section 3, we describe how we combine JSR94 with RuleML to achieve the desired interoperability.

3. The VIDRE Approach

The design goal of VIDRE is the ability to access business rules in a service-oriented way by hiding the rule implementation and execution. A business rule engine is encapsulated in a so-called VIDRE service provider (VSP).

Due to aforementioned heterogeneity of the different rule engines the architecture of VIDRE is based on the Java Rule Engine API (JSR 94). As mentioned before, this API has several drawbacks which makes it hard to provide interoperability with other applications or other business rule engines. Additionally, it does not allow to access a business rule engine remotely. Therefore, VIDRE combines the JSR 94 standard with RuleML [13]. In particular the object-oriented version of the RuleML in version 0.9 is used as it is expressive enough to represent facts, queries and even Java objects. Combining these efforts gives us the ability to plugin different rule engines which support the JSR 94 API standard without changing the knowledge or rulebase.

Figure 1 depicts the distributed architecture of our approach. Each VSP offers a generic RuleML interface which accepts valid RuleML documents as input. These RuleML documents can contain RuleML queries and RuleML facts which will be used for the evaluation of the queries. Similar to SOAP requests, RuleML documents can be sent via any arbitrary transport channels such as HTTP, JMS or even SMTP. The RuleML interface response is also encoded in a RuleML document. Besides the generic RuleML interface each service provider publishes two WSDL interfaces for accessing the client runtime and administration interfaces as a Web service. These two interface offer the functionality provided by JSR94 as a Web service to allow the management and execution of rule sets.

Two possibilities exist to access a VSP. Firstly, a client can use the runtime WSDL interface to fire rules for a given business document and/or to issue queries encoded in RuleML. The administration WSDL interface is primarily designed for managing and maintaining rule sets. The separation of the administration and client runtime interface allows more flexibility in implementing security policies or access control. The JSR 94 API is used as a standard API to access the Web service interface on the client side. A client library implements the JSR 94 API and encapsulates the underlying implementation details to the programmer. Therefore, the location and the provider of the used business rule engine are transparent to the end-user. Even the rule representation language is hidden by the client libraries since the JSR 94 Runtime API supports only Java objects as input to the rule engine.

Secondly, the SOAP-RuleML gateway offers a way to access business rules as well-defined Web services. A VSP provides the ability to publish several rules and queries as Web services at runtime via the administration interface. These rules can be easily accessed by the client application by just using the corresponding WSDL file published on the SOAP-RuleML gateway.

VIDRE Client A and B in Figure 1 use the WSDL interface to send a RuleML query directly to the business rule engine. The necessary transformation from Java objects to RuleML is transparent to the client side because it is hidden by the JSR 94 library implementation. VIDRE Client C uses the SOAP-RuleML gateway to invoke published business rules directly as Web services. For example, a ruleset for calculating the discount can be published as `DiscountCalculationService` to the SOAP-RuleML gateway, thus being available for the invocation as Web service.

The main advantage of VIDRE's service-oriented approach is, beside the ease of integration, the ability to perform distributed rule execution. The common rule and query representation language enables the rule engine (encapsulated in a VSP) to comprise knowledge from other VSPs with only few extensions to the rule engine. For example the prototype implementation uses the Jess [8] rule engine with minor extensions described later. The distributed execution is described in detail in Section 3.2.

3.1. VIDRE Service Provider Implementation

The core tasks of a VSP are i) implementing a plugin-mechanism by using JSR 94, ii) the transformation of RuleML requests to the Java object (and vice-versa) for creating a rule session and executing it by using the plugged-in rule engine and iii) providing the distributed rule execution mechanism.

The plugged mechanism for enabling a specific rule engine is based on the implementation of the `RuleMLTransformer` and the `ServiceProvider` interfaces. The used interfaces and factories are specified in service configuration files. The `ServiceProviderFactory` interface is used for retrieving a concrete `RuleMLTransformer` as well as the `ServiceProvider` for the plugged-in rule engine.

The core of the service implementation is the `RuleEngineService` class. This class holds a reference to the service provider API of the registered rule engine. This API is used to create rule runtime and rule administration interfaces for the registered rule engine. Additionally, it handles the RuleML transformations by using a concrete `RuleMLTransformer`.

RuleML Transformations

All incoming and outgoing requests to a VIDRE service provider are encoded in RuleML. The service implementation has to take care of translating between RuleML and the target language of the rule engine plugged in. In the VIDRE implementation transformers for Jess to RuleML and vice versa are implemented. Java objects are translated to RuleML with the Java reflection mechanism. These translators can be composed to achieve incremental translations. For example, Jess to Drools translation can be easily achieved by composing the Jess to RuleML translator and the RuleML to Drools translator. The drawback of this approach is that we can only use a proper subset given by the least common denominator of the supported rule-engine capabilities. These constraint is induced by the different rule representation languages. Changes to the vendors' rule language can be easily introduced by simply writing a new translator or adopting the old ones.

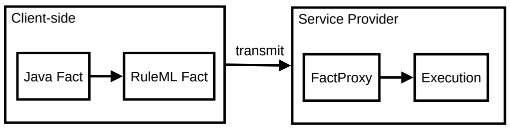

Figure 2. Adding a Java Fact

Transforming Objects to RuleML. In Figure 2 the process of adding a new fact to a VIDRE service providers is depicted. Whenever a client rule session is executed the objects contained in the session have to be transformed into an equivalent RuleML representation in order to transmit data to the service.

```
public class Order {
  private int id;
  private String customerId;
  private String product;
  private int quantity;
  // getters and setters
}
```

Listing 1. Java Fact

When adding a new fact, the first step is the transformation of a Java fact to a RuleML fact at the client-side. For example, Listing 1 defines a class `Order` which represents an order in our SCM example. Submitting this fact to a VIDRE service provider can easily be established by adding the object to a rule session. The client library uses the `Object2RuleML` transformer to transform an instance of this class into an equivalent RuleML representation. The resulting RuleML document is shown in Listing 2.

```
<Atom>
  <Rel>Order</Rel>
  <oid><Ind>1ca1a68</Ind></oid>
  <slot>
    <Ind>id</Ind>
    <Ind>785</Ind>
  </slot>
  <slot>
    <Ind>customerId</Ind>
    <Ind>154</Ind>
  </slot>
  <slot>
    <Ind>product</Ind>
    <Ind>Mainboard XY</Ind>
  </slot>
  <slot>
    <Ind>quantity</Ind>
    <Ind>1</Ind>
  </slot>
</Atom>
```

Listing 2. RuleML Fact

The fact name is marked up as the relation name, e.g., `<Rel>Order</Rel>`. On the same level, the four object attributes are marked up using the slotted RuleML syntax. For each object attribute, a slot is used to group the property name and the corresponding property value together. Property names and values are marked up as individual constants `<Ind>`. Accordingly, attribute values which represent variables are marked up as `<Var>`. This slotted RuleML syntax is needed to reconstruct the object internals. On the server-side (see right part of Figure 2) the RuleML representation can be transformed back into an object by using VIDRE's `Object2RuleML` transformer. A `FactProxy` is necessary because the plugged-in rule-engines can only operate with Java objects on the server-side. Such Java object instances are created on the client-side and, therefore, are not available on the server-side. The `FactProxy` is used to create a transient object representation of the submitted RuleML in-memory. These generated objects can be added to the rule-session on the server-side and act as input for the rule execution cycle.

The transformations on the server side are much more difficult because they include transformations of RuleML rules to rules for a concrete rule engine, such as Jess in our implementation.

The RuleML example in Listing 3 matches all `Order` objects which have `productId` equals 5 (the id represents a PC) and orders the needed parts (mainboard, memory, etc.) at the suppliers by asserting a new fact called `OrderParts`.

```
<Implies>
  <oid><Ind>If a computer is ordered, then order
        individual parts at supplier.</Ind>
  </oid>
  <head>
    <Atom>
      <Rel>OrderParts</Rel>
      <slot>
        <Ind>quantity</Ind>
        <Var>quantityVar</Var>
      </slot>
      <slot>
        <Ind>orderId</Ind>
        <Var>orderIdVar</Var>
      </slot>
    </Atom>
  </head>
  <body>
    <And>
      <Atom>
        <Rel>Order</Rel>
        <slot>
          <Ind>orderId</Ind>
          <Var>id</Var>
        </slot>
        <slot>
          <Ind>productId</Ind>
          <Ind>5</Ind>
        </slot>
        <slot>
          <Ind>customerId</Ind>
          <Var>customerIdVar</Var>
        </slot>
        <slot>
          <Ind>quantity</Ind>
          <Var>quantityVar</Var>
        </slot>
      </Atom>
    </And>
  </body>
</Implies>
```

Listing 3. RuleML Rule

The challenge is to transform this rule representation into a semantically equivalent representation in Jess. This transformation is done by VIDRE's `RuleML2JessTransformer`. In Listing 4 the resulting Jess rule is shown. Jess has a LISP-like syntax. A rule in Jess consists of two parts, separated by the => symbol. The first part of the rule (the "if" part or left-hand-side) consists of patterns that match facts. The second part consists of actions which take place when the left hand side of the rule is satisfied.

The RuleML Initiative proposes to use XSLT transformations [5] to accomplish this conversions, but this simple example illustrates that this is a real tough challenge. Furthermore, with XSLT the translated document has to be parsed again by the rule-set provider in order to be accessible by the rule engine. VIDRE translates RuleML to Jess at runtime, therefore, an efficient and fast translation is inevitable. VIDRE uses the Jess Java API to add rules and facts directly to the rule engine, hence no temporary representation is needed. Furthermore, the Jess Java API is easier to use and one can do more complex translations.

```
(defrule OrderParts
  "If a computer is ordered, then order individual parts
   at supplier."
  (Order (customerId ?customerIdVar)
         (orderId ?orderIdVar)
         (productId 5)
         (quantity ?quantityVar))
  =>
  (bind ?orderParts (new OrderParts))
  (call ?orderParts setQuantity ?quantityVar)
  (call ?orderParts setOrderId ?orderIdVar)
  (definstance OrderParts ?OrderParts))
```

Listing 4. Jess Rule

Representing Reaction Rules. One of the main advantages in rule-based systems is the ability to perform specific actions according to different situations. Reaction rules are important to picture complex business processes. The current RuleML design focuses on derivation rules. Reaction rules are addressed in a specific working group. The most promising effort to represent reaction rules is implemented in OO JDrew [9].

VIDRE encodes reaction rules like derivation rules. The client makes no difference between them. The differentiation is done on the server side, when the rule engine has to decide whether to assert a new fact (as done in derivation rules) or to call a predefined function. All user defined functions, for example an order-product function, are known to the rule engine, therefore, the rule engine looks up the function name and gets a handle to the function. If no function is returned the rule engine assumes that the rule is a derivation rule.

3.2. Distributed Rules and Knowledge

One of the main advantages of the VIDRE approach is the ability to distribute rules and facts. This section discusses the approach adopted in VIDRE to accomplish a distributed inference process.

Distributed Rule Execution

The distributed rules execution does not really differ from the normal, location bound, rules execution. The activity diagram in Figure 3 illustrates both, the distributed, and the normal rules execution.

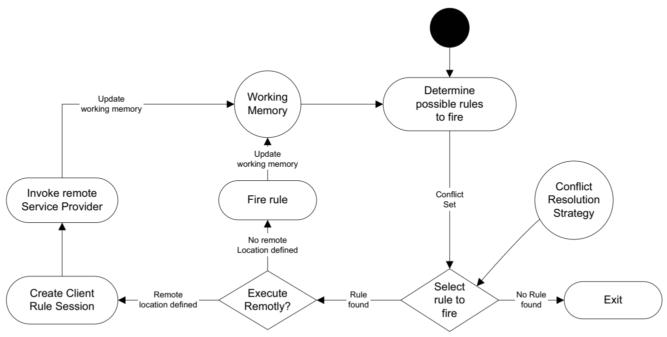

Figure 3. Activity Diagram – Rule Execution

Once the rule execution cycle starts, the pattern matcher determines which rules are activated. The list of activated rules, together with any other rules activated in previous cycles, form the conflict set. The rule engine uses a conflict resolution strategy to select the next rule to fire. If no rule can be fired the rule execution cycle terminates, otherwise VIDRE's inference engine has to decide whether this rule is executed local or remotely. A detailed description how this is implemented in VIDRE is given in the next section. For a normal location-bound rule the consequence part of the rule is executed and newly asserted facts are added to the working memory. These newly derived facts potentially activate or block rules in the conflict set, thus the execution cycle starts all over. If the activated rule is a distributed rule, so that the consequence of the rule should be executed at a remote service provider, VIDRE creates a client session and invokes the remote service provider. The result of the remote service invocation is added to the local working memory and again the rule execution cycle starts.

Distribution with Meta-Rules

Meta rules are distinguished from ordinary rules by their role which is to instrument the reasoning required to solve the problem, rather than to actually perform that reasoning [6]. Meta rules are often used in rule-based systems to encode preferences concerning the behavior of the inference process or they influence the conflict resolution strategy.

The following examples are based on the prototype implementation which currently only supports Jess, therefore, the examples are illustrated in the Jess rule language. The described methods can be applied to arbitrary rule engines. Jess does not provide any mechanism for defining meta-rules explicitly, therefore, VIDRE uses the salience property of Jess. In the conflict resolution Jess favors activated rules with a higher salience. This property ensures that meta rules are always evaluated before the business rules, hence, we can influence the inference process.

The question arises, how meta rules can be used for distributed rule execution? First, we have to distinguish between two types of distributed rules. The first type is characterized in that the consequence part of the rule is remote. This means that the assertion of a derived fact or the enabled action should be performed on a remote machine. Here, meta-rules play an important role. We have to encode that the consequence of the activated rule is a remote-rule and we have to encode the location where the action should take place.

VIDRE adds, for each remote rule, a meta-rule and a meta-fact. Listing 5 depicts such a meta-rule in the Jess language, which orders a mainboard remotely at a supplier for each newly placed PC order. The meta-fact `META_LOCATION` associates to each `OrderMainboard` object a URI. The URI encodes the target location where the consequence should be executed. The prototype implementation uses the prefix `META` for meta-information.

```
(defrule OrderMainboard
  "On reception of a new computer order, order a
   mainboard remotely at the supplier"
  (OrderParts (orderId ?orderIdVar)
              (quantity ?quantityVar))
  =>
  (bind ?OrderMainboard (new OrderMainboard))
  (call ?OrderMainboard setQuantity ?quantityVar)
  (call ?OrderMainboard setOrderId ?orderIdVar)
  (definstance OrderMainboard ?OrderMainboard))

(assert (META_LOCATION (name "OrderMainboard")
  (uri "rules://madrid.vitalab.tuwien.ac.at/supplier")))

(defrule META_REMOTE_OrderMainboard
  "if a remote location is defined call
   defRemoteInstance"
  ?obj <- (OrderMainboard)
  (META_LOCATION (name "OrderMainboard") (uri ?uri))
  =>
  (defRemoteInstance ?obj ?uri))
```

Listing 5. Remote Jess Rule

The meta-rule `REMOTE_OrderMainboard` matches all newly instantiated `OrderMainboard` objects if the object is a remote fact and, therefore, a `META_LOCATION` fact is defined. The function `defRemoteInstance` is a user defined Jess function which is responsible to define the fact at the remote location.

The second type is characterized in that the premises contain remote-facts. This is the more complicated case. A remote premise is only reasonable in combination with backward chaining. In order to perform backward chaining in Jess the remote fact has to be declared as backward chaining reactive. With backward chaining it is possible to incorporate data from outside the knowledge base, e.g., a relational database. Jess reasoning engine is strictly a forward chaining engine; backward chaining is simulated in terms of forward chaining rules.

VIDRE uses the backward chaining reactive facts in order to pull required data needed for the inference process from other VIDRE service providers. In Listing 6, all customers where the order quantity is larger than 10 are updated to premium customers. The required data can be pulled from a remote service provider (as defined by the `META_LOCATION` fact). To accomplish this, VIDRE hooks into the backward chaining mechanism of Jess. For each backward chaining reactive pattern on the left-hand side of the rule, Jess automatically asserts a fact named after the desired fact with prefix `"need-"`, e.g., `(need-Order nil nil)`.

This fact can be used to trigger a forward chaining rule, with the premise `(need-Order nil nil)`.

Listing 6 demonstrates a remote backward chaining fact. The function `do-backward-chaining` declares the `Order` fact as backward chaining reactive fact. Like in the remote-rule example the location of the remote service provider is encoded in the `META_LOCATION` fact. The meta-rule `META_REMOTE_Order` is activated as soon as the `do-backward-chaining` function was called and a remote location for the remote fact was specified. The function `runRemoteQuery` is a another user defined Jess function, which issues a query to VIDRE's service provider with the given URI.

```
(defrule PremiumCustomer
  (Order (customerId ?customerIdVar)
         (orderId ?orderIdVar)
         (productId ?productIdVar)
         (quantity ?quantity))
  (test (> ?quantity 10))
  =>
  (bind ?PremiumCustomer (new PremiumCustomer))
  (call ?PremiumCustomer setCustomerId ?customerIdVar)
  (definstance PremiumCustomer ?PremiumCustomer))

(do-backward-chaining Order)

(assert (META_LOCATION (name "Order")
  (uri "rules://rome.vitalab.tuwien.ac.at/retailer")))

(defrule META_REMOTE_Order
  "pull facts from remote location"
  ?obj <- (need-Order (customerId ?customerId)
                      (orderId ?orderId)
                      (productId ?productId)
                      (quantity ?quantity))
  (META_LOCATION (name "Order") (uri ?uri))
  =>
  (runRemoteQuery ?obj ?uri))
```

Listing 6. Remote Jess Fact

Managing the Rules Writing (distributed) facts and rules hand-crafted is a cumbersome and error-prone task. It is even getting worse if they have to be written in RuleML on the client-side. Therefore, we implemented the VIDRE administration client, a Java Swing GUI, which provides support for authoring distributed rule-sets. The client supports all management tasks of VIDRE and allows a rapid development of rule-based applications. The administration client allows to connect to each service provider via its Web service interface. Some demo videos can be found on the project Web page.

4. Evaluation and Discussion

This section presents a case study implementation based on VIDRE, where a built to order scenario from the computer manufacturing domain is implemented in terms of business rules. At first we describe the setup for our case study. Afterwards, we analyze the case study based on well-known software quality attributes and compare the VIDRE based solution with a pure Web service based solution without using rules. This version uses Web services to implement the business logic of the participants (supplier, manufacturer, retailer) without using a rule engine, thus, it hard-codes the rules for stock management, order processing, etc.

4.1 Case Study Scenario

In Figure 4, we have depicted a simplified version of our supply chain example. This illustrated setup consists of three companies: a computer retailer, a supplier and a manufacturer. In a real world example, the computer retailer would not only rely on one supplier. It would be a better strategy to order the parts at the supplier offering the best price. In our case study we have only one supplier, thus we need no rules for price comparison, although this could be easily represented in terms of business rules.

Location A - Computer Retailer. The computer retailer, deploys a Web application, the SOAP-RuleML gateway and a VSP. The Web server and application server do not need to be deployed at one location, thus they communicate using VIDRE's Web service interface. All the business logic needed by our computer retailer is implemented in terms of business rules and executed with VIDRE. The Web application provides two essential functions to customers. Firstly, an order page, where customers can place orders for different computer configurations. After submitting an order the Web client uses VIDRE's client library to assert a new order fact at the service provider. The Web client calls VIDRE the same way as a standalone client. Secondly, a query form is deployed where customers can track their orders. With this form the customers can track their orders and query VIDRE's working memory. Beside the Web application, our SOAP-RuleML gateway is deployed at the Web server. As aforementioned, VIDRE enables accessing business rules as Web services. For our computer retailer example the order business rules and the query to track the order status are exposed as Web services.

VIDRE's service provider is deployed to an application server. The available computer configurations are stored in terms of rules. Each computer configuration, consists of different parts. The computer retailer has no stock, thus, each order is decomposed in multiple orders to a supplier company. This part is solved with distributed business rules. Beside the order rules, the VSP manages rules to keep track of his own orders at the supplier. Furthermore, we need a rule which is activated when all parts are available. This rule is responsible for initiating the assembly of the computer. Another rule matches assembled computers and initiates the shipping procedure. The complete business order logic is

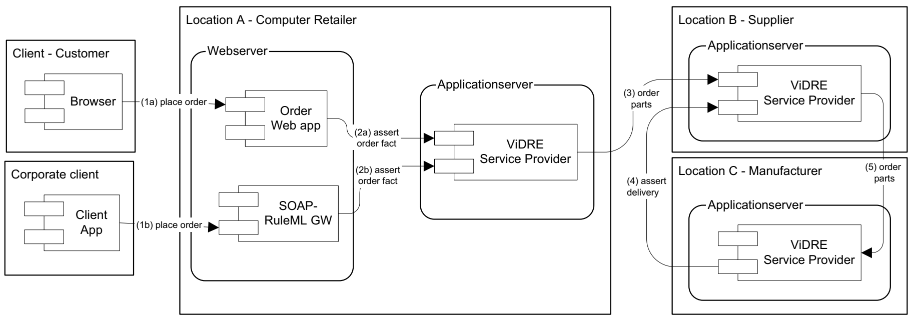

Figure 4. Deployment View

represented with business rules.

Location B - Supplier. At this location, a VSP is deployed to an application server. The supplier acts as an intermediary company and provides the computer retailer with all the needed parts. Efficient inventory management is inevitable for companies to save costs associated with storing inventory. The supplier uses business rules to keep track of the inventory. For each part, a rule is responsible for reordering the part at the manufacturer, if it runs out. This rule is implemented with distributed business rules and is very similar to the order rules of the computer retailer. The only difference is the type of triggering.

Each order is represented by one fact associated with a unique order identification and is retracted on order completion. If an order fact is added to the working memory, a rule is fired. This rule examines the inventory and initiates the shipping of the ordered parts. If the part is not available, the rule issues an order to the manufacturer, but only if no order exists. On completion of an order a distributed rule informs the computer retailer, that the ordered part has been shipped successfully. This, rule is implemented the same way as an order rule, but this time it triggers the assembling at the computer retailer.

Location C - Manufacturer. The manufacturer also deploys a VIDRE's service provider. The manufacturer is implemented in a similar way as the supplier. The main task of the VSP is to keep track of the orders and to manage the inventory. The orders are also represented by facts.

The implementation of the case study is done in terms of a number of different business rules. Some example rules are depicted in Section 3.

4.2 Analysis

For comparing the VIDRE approach with a state of the art architecture, we have implemented the same functionality by using a pure Web service based solution. Each supply chain participant is implemented as a single Web service which hard codes the rules in the service implementation. We want to analyze and compare the different approaches with respect to the following software quality attributes: maintainability, extensibility, reusability, scalability and performance.

Maintainability. The business rules approach uses declarative programming techniques to allow separation between business data and business logic. Business rules describe what to do, not how to do it, thus making it easier to manage and maintain complex business processes. VIDRE's administration client facilitates the creation and maintenance of rule sets to a number of VSPs. For example, the customer policies of our supply chain can be edited within the administration client and can be redeployed without changing a line of source code. Furthermore, the administration client allows testing changes to rule sets immediately through a VSP's runtime interface.

Extensibility. Extensions can be deployed without stopping a VSP. In large production systems the number of business policies can reach up to thousands of business rules. Extensions to these systems pose though challenges to developers. An appraisal of side effects is very difficult. In rule based systems, new business policies can be added to rule sets with less concerns about side effects. The rule engine is responsible for activating and firing the rules. Conflicting rules are resolved by the conflict resolution strategy of the rule engine. In our supply chain context, a new policy for inventory management can be easily added with VIDRE's administration client to existing rule sets.

Reusability. Business rules make the business logic explicit. Conventional implementations mix up the business logic with the code necessary to execute the logic. Hence, the reuse of business logic is very hard to accomplish and is often not very efficient. Usually only components and tech-

Transition	VIDRE	Pure Web service Solution
(1a) place order	32 ms	34 ms
(1b) place order	55 ms	-
(2a) assert order fact	54 ms	37 ms
(2b) assert order fact	42 ms	-
(3) order parts	32 ms	27 ms
(4) order parts	38 ms	29 ms
(5) assert delivery	31 ms	31 ms

Table 1. Performance Metrics

nical assets, such as logging and tracing are reused. With business rules, one can address the reuse of business logic. For example, stock policies or registration validation rules can be reused within intra or even inter-organizational applications.

Scalability. Scalability is not the main focus of VIDRE, nevertheless, it is an important issue to large scale distributed systems. The distribution of rules gives us the ability to distribute the load on different machines. Here, meta rules can be used to specify alternate service providers with the same rule set registered. The system can grow smoothly and economically as long as the entry point of the clients does not become an bottleneck.

Performance. Performance is a main issue in most enterprise systems, thus, it is important to analyze the timing behavior of our approach compared to a pure SOA implementation. Table 1 summarizes the observed performance metrics of a VIDRE service invocation. The measured values refer to our supply chain example depicted in Figure 4. We have added measuring points to each transition in our VIDRE based, as well as to our pure Web service implementation.

The performance of our VIDRE approach does not differ significantly from the pure Web service based implementation. The entry point of our application is in both implementations the order Web page. Both implementations start the order workflow by invoking the servlet of the Web application. By now, the implementations are nearly equal. From that point on, the implementations starts to diverge. The VIDRE implementation creates a new rule session and initializes the session with an order fact. Behind the scenes this fact is encoded in RuleML. When the rule session is executed, the RuleML document is packaged in a SOAP envelope and is sent to VIDRE's service provider endpoint (2a). In our case study implementation the overhead of encoding the document in RuleML does not carry authority. In a real world application this possibly results in some performance overhead.

Beside the Web page, the VIDRE implementation has a second entry point, the SOAP-RuleML gateway. The performance value (2a) includes the invocation of our SOAP-RuleML Gateway and the translation of the SOAP document to an equivalent RuleML representation. The invocation of the service provider(2b) is done through the native RuleML interface, thus, there is no overhead in creating a SOAP envelope.

The distributed rules execution (3, 4, 5) also use the native RuleML execution to avoid RuleML encoding twice. Compared to the other implementation, the overhead of invoking the rule engine and the encoding of RuleML is negligible in our scenario.

Compared to this little tradeoff in performance (for encoding RuleML documents), the plus gained through the higher maintainability and reusability increases the overall benefit of the VIDRE based solution, in case many business rules are involved.

5. Related Work

Recently, business rules get a lot of attention from academia as well as from industry. A number of business rules bases approaches for realizing service-oriented solutions are proposed (e.g., [2, 10]).

The work presented in [1] comes closest to ours w.r.t. the distributed execution of business rules. It proposes an extension to the Jess rule engine to allow a distributed rule execution. Their distributed inference mechanism uses a shared working memory (SWM) to store the facts and each rule engine uses this memory to access the facts. We do not use a shared memory, instead, we propose to use meta-rules to handle the distribution. Furthermore, we use RuleML as interlingua and provide translators from RuleML to the corresponding rule engines, thus our approach is not limited to a concrete rule engine.

Schmidt [15] proposes a distributed rule execution approach by encoding RuleML rules in the SOAP header which are processed by different SOAP intermediaries. Based on some conditions in one of the business rules a different execution path can be chosen. The focus of this work is on rule execution but it is not discussed how distributed rules can be specified and executed.

In [11, 12], we have shown a loosely-coupled integration of business rules into the well-known WS-BPEL [19] language, a standard for describing Web service business processes. The approach uses an enterprise service bus (ESB) and business rules by using an interceptor pattern [14] to Web service invocations (e.g., WS-BPEL invoke activity) from a WS-BPEL engine. In contrast to the approach presented in this paper, our previous work focused on accessing different rule engines with the same API and generating Web service for it. In this paper we focus on using a common rule representation language and enabling a distributed rule execution by using meta rules.

In the area of business rule engines, a number of different tools are available. Drools [3] is one of the most popular open source Java business rule engines. The Drools rule engine uses an enhanced version of the Rete algorithm [4] called the Rete-OO algorithm. Drools is a purely forward chaining system.

Jess [8] (Java Expert System Shell) is a robust and mature expert system shell for developing rule-based applications in Java. Jess has many unique features including backward chaining and working memory queries, and can directly manipulate and reason about Java objects. Both rule engines do not allow a distributed rule execution and can be used only as library which is linked to the application.

6. Conclusions

In this paper the design and implementation of VIDRE, a distributed service-oriented rule engine, is presented. VIDRE offers several novel features: Firstly, we combine the rule-based approach with service-oriented computing to allow an easy integration into client applications and make it easy for applications to consume business rule services (i.e., business rules exposed as Web service). Secondly, VIDRE overcomes the interoperability problem of current rule engines by providing a lightweight plug-in mechanism allowing to exchange business rule engines being completely transparent for client applications. Thirdly, we introduce distributed business rules which allow the execution of rules over several distributed rule engines.

Currently, VIDRE is in a stable state, nevertheless, there are several issues which need to be addressed in future work. VIDRE offers two different interfaces for managing business rules, a runtime and an administration interface but it lacks a role-based authentication and authorization mechanism for rule management to secure the knowledge from unwanted changes. We plan to add this feature by using WS-Security mechanisms.

VIDRE currently uses Jess as the only plugged-in rule engine, nevertheless, the API support for other popular rule engines is already included. Thus, we plan to add support for the Drools rule engine, a mature open-source project.

References

[1] F. Cabitza, M. Sarini, and B. D. Seno. Djess - a context-sharing middleware to deploy distributed inference systems in pervasive computing domains. In *Proceedings of the International Conference on Pervasive Services (ICPS'05)*, pages 229–238, July 2005.

[2] A. Charfi and M. Mezini. Hybrid Web Service Composition: Business Processes Meet Business Rules. In *Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC'04)*, November 2004.

[3] Drools. Java Rule Engine, 2005. `http://drools.org/` (accessed March 2006).

[4] C. L. Forgy. Rete: A fast algorithm for the many pattern/-many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[5] B. N. Grosof, M. D. Gandhe, and T. W. Finin. Sweetjess: Translating DAMLRuleML to JESS. In *RuleML*, 2002.

[6] P. Jackson. *Introduction to Expert Systems, 3rd Edn.* Harlow, England: Addison Wesley Longman., Boston, MA, USA, 1999.

[7] Java Community Process. JSR 94 Java Rule Engine API, 2005. `http://www.jcp.org/en/jsr/detail?id=94`.

[8] JESS. Java rule engine, 2005. http://herzberg.ca.sandia.gov/jess/ (accessed December 2005).

[9] OO jDREW. Java Rule Engine, 2005. http://www.jdrew.org/oojdrew/ (accessed December 2005).

[10] B. Orriëns, J. Yang, and M. Papazoglou. A Rule Driven Approach for Developing Adaptive Service Oriented Business Collaboration. In *Proceedings of the the 3rd International Conference on Service Oriented Computing (ICSOC'05), Amsterdam, The Netherlands*, Dec. 2005.

[11] F. Rosenberg and S. Dustdar. Business Rules Integration in BPEL – A Service-Oriented Apporach. In *Proceedings of the 7th International IEEE Conference on E-Commerce Technology (CEC'05)*, 2005.

[12] F. Rosenberg and S. Dustdar. Design and Implementation of a Service-Oriented Business Rules Broker. In *Proceedings of the 1st IEEE International Workshop on Service-oriented Solutions for Cooperative Organizations (SoS4CO'05)*, 2005.

[13] RuleML. The Rule Markup Initiative, 2005. `http://www.ruleml.org/` (accessed March 2006).

[14] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 1 edition, 2000.

[15] R. Schmidt. Web services based execution of business rules. In *International Workshop on Rule Markup Languages for Business Rules on the Semantic Web, 14 June 2002, Sardinia (Italy)*, 2002.

[16] S.Petrovic. Rule-Based Expert Systems, 2006. `http://www.cs.nott.ac.uk/~sxp/ES3/index.htm` (accessed March 2006).

[17] The Business Rules Group. Defining Business Rules - What Are They Really? July 2000. `http://www.businessrulesgroup.org/first_paper/br01c0.htm`.

[18] B. von Halle. *Business Rules Applied: Building Better Systems Using the Business Rules Approach*. Wiley, 1st edition, 2001.

[19] WS-BPEL. Business Process Execution Language for Web Services Version 1.1. `http://www.ibm.com/developerworks/library/ws-bpel/`, May 2003.