A Rule-based Middleware for Business Process Execution

Adrian Paschke[1] and Alexander Kozlenkov[2]

[1]RuleML Inc., Canada
`Adrian.PaschkeATgmx.de`
[2]Betfair Ltd., London
`alex.kozlenkovATbetfair.com`

Abstract. While past research in service oriented computing has focused on the fairly static functional description and the operators of services the dynamic and flexible composition and collaboration of services in business processes in style of semi-autonomous multi-agent systems has not been as thoroughly explored. Executable business process description languages such as BPEL provide only limited expressiveness to describe (business) decision logic and conditional reaction logic. In this paper we propose a rule-based representation approach to describe service-oriented business processes and implement a distributed execution middleware for such rule-based process specifications using rule engines as execution environments which are deployed as distributed agents / web-based inference services. This declarative rule-based approach which allows for semi-autonomous decisions and reactions within service-based business process execution has the potential to profoundly change the way IT services are used and collaborate on the Web.

1 Introduction

Flexibility in dynamically composing new business processes and integrating heterogenous information systems (HIS) enabling ad-hoc cooperations is one of the main aims of the recent service oriented computing (SOC) paradigm. While past research in SOC has focused on the fairly static functional description and the operators of services - including publication, discovery, selection, binding and composition of basic services - the flexible and possibly self-autonomous collaboration, management and coordination of complex IT services in business processes is still an open task.

The currently most well-known language for formally describing business processes and business interaction protocols in a machine-readable way is the Business Process Execution Language (BPEL) [7]. BPEL is an orchestration language for web services providing facilities to enable sending and receiving messages. It describes processes as activities with control flow (e.g. for executing commands in sequence or in parallel). However, its expressiveness for representing conditional decision logic, e.g. for conditionally branching in parallel concurrent channels (split) or merging channels (join), is rather limited and constraint to procedural if-then-else flow control constructs. That is, BPEL only expresses qualifying conditions in the sense of material truth implications. For instance, a rule set "*if given p and if p then q and if q then r*" is expressible as a nested truth implication rule $(p \supseteq q) \supseteq r)$, but not as an automatically chained rule set as in logic programming based approaches. Accordingly, the BPEL approach is not modular and larger rule-based decision logic, such as business rules, can not be easily expressed in a compact declarative way as in logic programming. Moreover, the treatment of variables in BPEL is on the level of pure value assignments and ground pattern matching and a declarative (typed) unification as in logic programming approaches is missing.

In this paper we propose a declarative rule and event-based middleware for business process orchestration which combines technologies from declarative rule-based programming, in particular logic programming, with enterprise service technologies for complex event processing (CEP). This rule-based approach has the potential to profoundly change the way IT services are used and collaborate in business processes. Akin to multi-agent systems (MAS) the rule-based logic layer which wraps the existing web services and data sources allows for semi-autonomous decisions and reactions which are necessary e.g., for IT service management (ITSM) processes such as such service level management (SLM), change management, availability management (see ITIL for an overview [8]), business activity monitoring (BAM), and business process management (BPM) such as service execution in workflow-like business processes. Ultimately, our rule-based design artifact might put the vision of highly flexible and adaptive service supply chains into large scale practice. In this paper we contribute with a declarative rule-based service-oriented methodology and a scalable middleware to operationalize such a distributed rule-based approach where event-based communication and rule-based behavioral and decision logic plays a central role in connecting the various IT resources and Web-based services to business service networks. This addresses an urgent need businesses do have nowadays: to declarative describe and dynamically change their decision and business process logic which underpins their applications and service offerings in order to adapt to a flexible business environment.

The rest of the paper is organized as follows: In section 2 we give an overview of the relevant concepts. In section 3 we discuss the use of rules for describing conditional business process execution flows and introduce our rule-based approach combining standard derivation rules to represent decision logic and messaging reaction rules to describe message-driven process workflows. In section 4 we introduce the main components of our rule-based inference middleware. Section 5 concludes this paper.

2 Background

In this section we give an insight into the applied technologies and define relevant concepts and technologies.

2.1 Service Oriented Computing

The computing paradigm that utilizes services for developing applications in open distributed environments such as the Web is known as Service Oriented Computing (SOC). The vision is to build large scale service supply chains (also known as business services networks) which enable enterprises to define and execute web services based transactions and business processes across multiple business entities and domain boundaries using standardized (web) protocols. Web services, which emerged in the last time as the prevailing technology for implementing IT services on the web, have received a great commitment from industry and research. A service-oriented architecture (SOA) expresses a perspective of software architecture that defines the use of loosely coupled software services to support the requirements of the business processes and software users. In a SOA environment, resources on a network are made available as independent services that can be accessed without knowledge of their underlying platform implementation. A SOA is not tied to a specific technology. It may be implemented using a wide range of technologies, including REST, DCOM, CORBA or Web Services. Service Component Architecture (SCA) is a set of specifications which describe a model for building applications and systems using a SOA. Recently semantic web services (SWS) which provide an approach for representing the functionality of Web Services with the help of Semantic Web ontologies and BPEL (business process execution language) for web service orchestration attracts much attention from industry.

2.2 Complex Event Processing

Complex Event Processing (CEP) is a paradigm, which allows real-time reactions to detected complex events and state changes. A complex event type (i.e., the detection condition of a complex event) is built from occurred atomic or other complex event instances according e.g. to the operators of an event algebra. Event processing describes the process of selecting and composing complex events from raw events (event derivation), situation detection (detecting transitions in the universe that requires reaction either "reactive" or "proactive") and triggering actions as a consequence of the detected situation (complex event + conditional context). Events can be processed in real time without persistence (short-term) or processed in retrospect as a computation of persistent earlier events (long-term), but also in aggregated from, i.e., new (raw) events are directly added to the aggregation which is persistent. It can be done either actively (pull-model), i.e., based on actively monitoring the environment and, upon the detection of an event, trigger a reaction, or passively (push-model), i.e., the event occurrences are detected by an external component and send as event messages to the event system for further processing.

2.3 Declarative Rule Programming and Business Rules Management

Rule based systems have been investigated comprehensively in the realms of declarative programming and expert systems over the last two decades. And, in recent years we have seen the rise of a new type of software called business rule management systems (BRMS). These are systems to externalize business rules and to provide a facility for business rule management. Using (inference) rules has several advantages: reasoning with rules is based on a semantics of formal logic, usually a variation of first order predicate logic, and it is relatively easy for the end user to write rules. The basic idea is that users employ rules to express *what* they want, the responsibility to interpret this and to decide on *how* to do it is delegated to an interpreter (e.g., an inference engine or a just in time rule compiler). Early manifestations of business rule engines which have their roots in the realm of artificial intelligence and inference systems were complex, expensive to run and maintain and not very business-user friendly. Improved technology providing enhanced usability, scalability and performance, as well as less costly maintenance and better understanding of the underlying inference systems makes the current generation of business rule engines (BRE) and rules technology more usable.

There are different types of rules reaching from derivation rules, transformation rules to reaction rules and integrity rules. For the representation and execution of rules there are various ways, e.g., if-then constructs in procedural programming languages such as Java or C/C++ (with control flow), decision tables/trees, truth-functional constructs based on material implication, implications with constraints (e.g., OCL), triggers and effectors (e.g., SQL trigger), ECA rules as in active databases, production rules, or logical knowledge representation (KR) approaches based on subsets of first order logic and non-monotonic logic such as logic programming (LP) techniques. In this paper we exploit logic based rule languages; that is, derivation rules represented as logic programs, which has several advantages:

1. reasoning with derivation rules is based on a semantics of formal logic - a variation of first order predicate logic - enabling automated rule chaining by resolution and variable unification, which alleviates the burden of having to implement extensive control flows as in imperative programming and allows for easy extensibility without affecting the underlying mechanisms and architectures,
2. it allows for a compact and comprehensible human and machine-oriented representation and high levels of automation and flexibility to adapt to rapidly changing requirements, and
3. it is relatively easy for the end user to write rules and hence rapidly engineer and maintain rule-based decision and reaction logic

3 Rules for Business Processes Execution

In this section we argue for a rule-based approach to describe service-oriented business processes. In particular, we draw on backward-reasoning logic programming (LP) concepts to formalize decision logic in terms of derivation rules and combine them with forward-directed messaging reaction rules in order to exploit the benefits of both worlds.

For instance, a rule set from a Service Level Agreement (contractual service agreements describing non-functional properties such as quality of service) might define three different service schedules:

```
if current time between 8am and 6pm then prime schedule.
if current time between 6pm and 8am then standard schedule.
if current time between 0am and 4am then optional maintenance schedule.

if prime schedule then the service level "average availability"
has a low value of 98%, a median of 99% and a high value of 100%
and a response time which must be below 4 seconds.
...
if standard schedule then the responsible role for service outages
is the second admin.
```

As shown e.g. in the RBSLA project [9] such rule sets can be adequately represented as logic programs and easily extended with further rules, e.g. rules describing escalation levels if service levels are missed. Integrated into messaging reaction rules this rule-based decision logic can serve as expressive conditional logic in the reactive control flow of the business process execution and for the execution of activities in process instances.

Messaging reaction rules describe (abstract) processes in terms of message-driven conversations between parties and represent their associated interactions via constructs for asynchronously sending and receiving event messages. That is, the control flow of a business process is defined by the order of sending and receiving message constructs in messaging reaction rules. In contrast to standard Event-Condition-Action (ECA) rules which typically only have a global state, messaging reaction rules maintain a local conversation state which reflects the process execution state and support performing of different activities within process instances managed in simultaneous conversation branches.

The main language constructs of messaging reaction rules (as implemented in the rule engine Prova [5]) are: *sendMsg* predicates to send outbound messages, reaction *rcvMsg* rules which react to inbound messages, and *rcvMsg* or *rcvMult* inline reactions in the body of messaging reaction rules to receive one or more context-dependent multiple inbound event messages:

```
sendMsg(XID,Protocol,Agent,Performative,Payload |Context)
rcvMsg(XID,Protocol,From,Performative,Paylod|Context)
rcvMult(XID,Protocol,From,Performative,Paylod|Context)
```

where *XID* is the conversation identifier (conversation-id) of the conversation to which the message will belong. *Protocol* defines the communication protocol such as JMS, HTTP, SOAP, Jade etc. *Agent* denotes the target party of the message. *Performative* describes the pragmatic context in which the message is send. A standard nomenclature of performatives is, e.g. the FIPA Agents Communication Language ACL or the BPEL activity vocabulary. *Payload* represents the message content sent in the message envelope. It can be a specific query or answer or a complex interchanged rule base (set of rules and facts).

For instance, the following messaging reaction rule waits for an inbound query: (variables start with upper case letters)

```
% receive query and delegate it to another party
rcvMsg(CID,esb, Requester, acl_query-ref, Query) :-
  responsibleRole(Agent, Query),
  sendMsg(Sub-CID,esb,Agent,acl_query-ref, Query),
  rcvMsg(Sub-CID,esb,Agent,acl_inform-ref, Answer),
  ... (other goals)...
  sendMsg(CID,esb,Requester,acl_inform-ref,Answer).
```

When activated by a received inbound event message which unifies with the rule head, it first selects the responsible role for the query (first goal in the rule body). This selection logic is e.g. implemented by standard derivation rules as outlined above ("responsible role is the second admin during standard schedule"). Then the rule sends the query in a new sub-conversation to the selected party and waits for the answer to the query. Remarkably, messaging reaction rules do not require separate threads for handling multiple conversation situations simultaneously. That is, the rule execution blocks until an answer message is received in the inlined sub-conversation which activates the rule execution flow again, e.g. to prove any subsequent goals of the rule. Data from the received answers is bound to variables as usual in logic programming including also backtracking to several variable bindings. Finally, in this example the rule sends back the answer to the original requesting party and terminates (finitely succeeds).

The corresponding messaging reaction rule of the responsible party which receives the query message, derives the answer and sends it back in the sub-conversation might look like this:

```
% answers query
rcvMsg(XID, esb, From, Performative, [X|Args]):-
  derive([X|Args]),
  sendMsg(XID,esb,From, answer, [X|Args]).
```

The core components of a business process execution language namely the ability to realize separate activities and to control their cooperation can be realized by messaging reaction rules. The behaviour of the inbound links defined by the *rcvMsg* reaction rules is similar to BPEL links. However, the declarative rule-based approach provides a much more expressive and compact declarative programming language to represent complex event processing logic in arbitrary combination with conditional decision logic implemented in terms of derivation rules. and complex event-based workflow patterns (as described, e.g. by Van der Aalst et al. [11]) such as Join, Simple Merge, Cancel Activity, Multi-Choice, Structured Loop, Deferred Choice, Milestone. To exemplify this, figure 1 shows an event-based workflow which is implemented in terms of messaging reaction rules as follows:

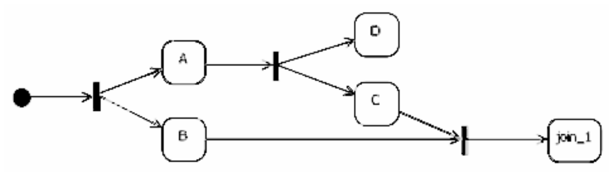

Fig. 1. Example Rule-Based Workflow

```
process_join() :-
    iam(Me),
    init_join(XID,join_1,[c(_),b(_)]),
    fork_a_b(Me,XID).
fork_a_b(Me,XID) :-
    rcvMsg(XID,self,Me,reply,a(1)),
    fork_c_d(Me,XID).
fork_a_b(Me,XID) :-
    rcvMsg(XID,self,Me,reply,b(1)),
    join(Me,XID,join_1,b(1)).
fork_c_d(Me,XID) :-
    rcvMsg(XID,self,Me,reply,c(1)),
    % Tell the join join_1 that a new pattern is ready
    join(Me,XID,join_1,c(1)).
% invoked by join once all the inputs are assembled.
join_1(Me,XID,Inputs) :-
    println(["Joined for XID=",XID," with inputs: ",Inputs]).
```

```
% Prints
% Joined for XID=agent@hostname001 with inputs [[b,1],[c,1]]
```

For the reason of space we refer to [4] for an implementation of typical workflow patterns in terms of messaging reaction rules in Prova including also patterns which can not be easily represented in a compact way in BPEL it such as Deferred Choice (although BPEL is Turing-complete).

Conditional control flow logic for process execution is implemented in terms of messaging reaction rules which communicate via event messages. Process instances are related to conversations uniquely identified by conversation ids. The local states are maintained in these conversations and events are assign to their corresponding conversations (process instances). Actions can be triggered either by sending outbound messages (e.g. wrapping a remote procedure call to a web service and transported it via SOAP) or by directly making procedural calls via the rich built-in interfaces to query e.g. external data sources and the tight Java integration of the rule engine Prova. For instance, the following rule receives a remote procedure call with the operation name and the list of arguments bound to variables, dynamically instantiates the Java class *SOAPClient* (fully qualified class name) and calls the *invoke* method of the Java object with the operation name and the arguments, binds the result (object) to the variable *Result* and sends it back to the original requester.

```
rcvMsg(XID, esb, From, soap_rpc, [Operation|Args]):-
  Result = rblsa.utils.SOAPClient.invoke(Operation,Args),
  sendMsg(XID,esb,From, answer, Result).
```

In summary, the major benefit of the described rule-based approach is the tight integration of standard derivation rules and messaging reaction rules. Complex (business) decision logic, such as business rules, can be implemented in a declarative way in terms of derivation rules (as logic programs) and used to prove conditional goals mixed in arbitrary combinations with send and receive message constructs in messaging reaction rules. As a result, the declarative rule-based approach provides a much more expressive and compact representation language for describing business process executions in terms of message conversations between parties as the current languages such as BPEL or BPML do. The Role Activity Diagram shown in figure 2 gives an example of a typical message-driven process in a virtual organization, namely "scheduling of a meeting" where the agents (parties) involved in the process are implemented as rule-based agent services (see http://ibis.in.tum.de/projects/paw/ruleml-2007/).

The project manager creates a possible date for the meeting from the public organizational calendar (accessed e.g. via iCAL web service interface) and proposes this date to all project members. The members compare this date with their personal calendars and send counter-proposals if the deadline does not fit according to their personal decision logic. The manager then creates a new proposal. This process is repeated until all agents agreed on the proposed meeting date; the manager then creates an entry in the public calendar and informs all member agents about this date. The members add this date to their personal (not public) calendars. That is, the members implement their own, self-autonomous decision logic for accepting proposals or making counter-proposals. For instance, they can lie and pretend they have no time at a certain date or propose strategic counter-proposals.

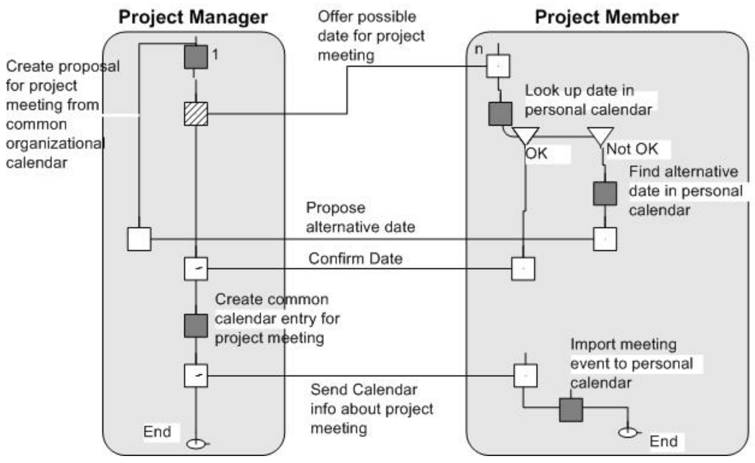
Project Manager
Project Member
Offer possible date for project meeting
Create proposal for project meeting from common organizational calendar
1
n
Look up date in personal calendar
OK
Not OK
Find alternative date in personal calendar
Propose alternative date
Confirm Date
Create common calendar entry for project meeting
Import meeting event to personal calendar
Send Calendar info about project meeting
End
End

Fig. 2. Project Meeting

4 A Rule-Based Middleware Model for Service Oriented Computing

In this section we will introduce the main components of the rule-based middleware architecture to deploy and execute distributed rule-based business process specifications. We focus on the technical aspects of the middleware and on the machine-to-machine communication between automated rule inference services deployed on the middleware for execution rule-based business process spcifications. Figure 3 illustrates the architecture of the middleware.

Several Prova rule engines [5] as execution environments are deployed as distributed web-based services. Each service runs a local rule base which implements the decision and reaction logic how to access the local services, applications, data sources (e.g. web services, Java object representations such as EJBs, databases etc.) and react to incoming events messages according to the defined rule-based execution logic. Some of the rule inference nodes might act as global orchestration nodes which manage the global rule-based process flow and the delegation of event messages to the corresponding inference services which simply act as rule-based wrapper to access the local data sources and services in a rule-based way. To communicate between the rule inference service, RuleML, the current de-facto standard language for web rules, is used as common rule interchange format into which the platform-specific Prova language is translated. An enterprise service bus (ESB) is used as object broker for the web-based inference services and as asynchronous messaging middleware between the services. Different transport protocols such as JMS, HTTP or SOAP (or Rest) can be used to transport rule sets, queries and answers as payload of event messages in Reaction RuleML format between the inference services. Direct access to external web services and applications is also possible via the ESB.

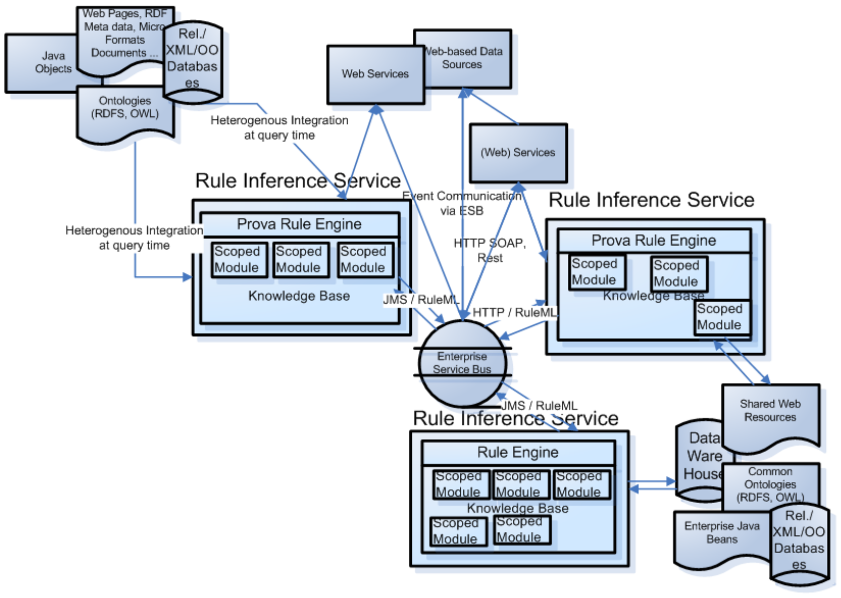

Fig. 3. Rule-based Middleware

4.1 Reaction RuleML as Platform-Independent Rule Interchange Format

The Rule Markup Language (RuleML) [1] is a modular, interchangeable rule specification standard to express both forward (bottom-up) and backward (top-down) rules for deduction, reaction, rewriting, and further inferential-transformational tasks. It is defined by the Rule Markup Initiative [2], an open network of individuals and groups from both industry and academia that was formed to develop a canonical Web language for rules using XML markup and transformations from and to other rule standards/systems. The language family of RuleML covers the entire rule spectrum, from derivation rules to reaction rules including rule-based complex event processing (CEP) and messaging (Reaction RuleML [10]), as well as verification and transformation rules.

Reaction RuleML [10, 9] is a general, practical, compact and user-friendly XML-serialized sublanguage of RuleML for the family of reaction rules. It incorporates various kinds of production, action, reaction, and KR temporal/event/action logic rules as well as (complex) event/action messages into the native RuleML syntax using a system of step-wise extensions. The general syntax of a reaction rule consists of six partially optional parts:

```
<Rule style="active" evaluation="strong">
   <label> <!-- metadata  --> </label>
   <scope> <!-- scope --> </scope>
   <qualification> <!-- qualifications --> </qualification>
   <oid> <!-- object identifier --> </oid>
   <on>  <!-- event -->  </on>
   <if>  <!-- condition --> </if>
   <then> <!-- conclusion --> </then>
   <do>  <!--  action -->  </do>
   <after> <!-- postcondition --> </after>
   <else> <!-- else conclusion --> </else>
   <elseDo> <!-- else/alternative action --> </elseDo>
   <elseAfter> <!-- else postcondition --> </elseAfter>
</Rule>
```

Inbound and outbound messages $< Message >$ are used to interchange events (e.g. queries and answers) and rule bases (modules) between the agent nodes:

```
<Message mode="outbound" directive="pragmatic performative">
  <oid> <!-- conversation ID--> </oid>
  <protocol> <!-- transport protocol --> </protocol>
  <sender> <!-- sender agent/service --> </sender>
  <content> <!-- message payload -->  </content>
</Message>
```

- @*mode* = *inbound*|*outbound* - attribute defining the type of a message
- @*directive* - attribute defining the pragmatic context of the message, e.g. a FIPA ACL performative
- < *oid* > - the conversation id used to distinguish multiple conversations and conversation states
- < *protocol* > - a transport protocol such as HTTP, JMS, SOAP, Jade, enterprise Service Bus (ESB) ...
- < *sender* >< *receiver* > - the sender/receiver agent/ service of the message
- < *content* > - message payload transporting a RuleML / Reaction RuleML query, answer or rule base

The directive attribute corresponds to the pragmatic instruction, i.e. the pragmatic characterization of the message context. External vocabularies defining pragmatic performatives might be used by pointing to their conceptual descriptions. The typed logic approach of RuleML enables the integration of external type systems such as Semantic Web ontologies or XML vocabularies. A standard nomenclature of pragmatic performatives is defined by the Knowledge Query Manipulation Language (KQML) and the FIPA Agent Communication Language (ACL) which defines several speech act theory-based communicative acts. [3]

4.2 Enterprise Service Bus as Communication Middleware

To seamlessly handle asynchronous message-based interactions between the rule inference services and with other applications and web-based services an enterprise service bus (ESB), the Mule open-source ESB [6], is integrated as communication middleware. The ESB allows deploying the rule-based agents as highly distributable rule inference services installed as Web-based endpoints in the Mule object broker and supports the Reaction RuleML based communication via arbitrary transport protocols such as JMS, HTTP, SOAP (more than 30 protocols are supported) between them . That is, the ESB provides a highly scalable and flexible application messaging framework to communicate synchronously but also asynchronously between services.

4.3 Prova as Platform-specific Rule-based Process Flow Execution Environment

Prova [5, 9] is a highly expressive Semantic Web rule engine. Prova follows the spirit and design of the recent W3C Semantic Web initiative and combines declarative rules, ontologies and inference with dynamic object-oriented Java API calls and access to external data sources such as relational databases or enterprise applications and IT services. One of the key advantages of Prova is its elegant separation of logic, data access, and computation and its tight integration of Java and Semantic Web technologies. It includes numerous expressive features and logical formalisms such as:

- Easy to use and learn ISO Prolog related scripting syntax
- Well-founded Semantics for Extended Logic Programs with defeasible conflict resolution and linear goal memoization
- Order-sorted polymorphic type systems compatible with Java and Semantic Web ontology languages RDF/RDFS and OWL
- Seamless integration of dynamic Java API invocations
- External data access by e.g., SQL, XQuery, RDF triple queries, SPARQL
- Meta-data annotated modular rule sets with expressive transactional updates, Web imports, constructive views and scoped reasoning for distributed rule bases in open environment such as the Web
- Verification, Validation and Integrity tests by integrity constraints and test cases
- Messaging reaction rules for workflow like communication patterns based on the Prova Agent Architecture
- Global reaction rules based on the ECA approach
- Rich libraries and built-ins for e.g. math, date, time, string, interval, list functions

For a detailed description of the syntax, semantics and implementation of several of these formalisms see e.g. [9].

5 Conclusion

In this paper we have proposed a declarative rule-based representation approach for executable business processes descriptions where complex conditional workflows and enterprise integration patterns are implemented in terms of rules. The rule based logic language combines derivation rules as means to represent conditional decision logic such as business rules with messaging reaction rules to describe conversation based process flow between parties/services. The conversational interactions take place via event messages. We have implemented a rule-based middleware which deploys Prova rule engines as distributed inference service used to execute the rule-based business process descriptions. At the platform-independent level it uses Reaction RuleML as a compact, extensible and standardized rule and event interchange format between the platform-specific Prova services. A highly scalable and efficient enterprise service bus is integrated as a communication middleware platform and web-based agent/service object broker. The realization of business processes by means of rules provides an expressive orchestration language which forms the technical foundation to integrate the business rules technology with the enterprise service technology.

References

1. H. Boley. The rule-ml family of web rule languages. In *4th Int. Workshop on Principles and Practice of Semantic Web Reasoning*, Budva, Montenegro, 2006.
2. H. Boley and S. Tabet. Ruleml: The ruleml standardization initiative, http://www.ruleml.org/, 2000.
3. FIPA. Fipa agent communication language, http://www.fipa.org/, accessed dec. 2001, 2000.
4. A. Kozlenkov. Prova workflow patterns, http://www.prova.ws/etc/mule-prova-agents.zip, 2007.
5. A. Kozlenkov, A. Paschke, and M. Schroeder. Prova, http://prova.ws. 2006.
6. Mule. Mule enterprise service bus, http://mule.codehaus.org/display/mule/home, 2006.
7. OASIS. Web services business process execution language version 2.0. available at http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf, 2007.
8. OGC. *Office of Government Commerce: Service Delivery.* The Stationary Office, London, 2001.
9. A. Paschke. *Rule-Based Service Level Agreements - Knowledge Representation for Automated e-Contract, SLA and Policy Management.* Idea Verlag GmbH, Munich, 2007.
10. A. Paschke, A. Kozlenkov, H. Boley, M. Kifer, S. Tabet, M. Dean, and K. Barrett. Reaction ruleml, http://ibis.in.tum.de/research/reactionruleml/, 2006.
11. W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. distributed and parallel databases. *MIS Quarterly*, 14(1), 2003.