3rd East European Workshop on Rule-Based Applications

Cottbus, Germany, September 21, 2009

Editors:
Adrian Giurca
Grzegorz J. Nalepa
Gerd Wagner

Brandenburgische
Technische Universität
Cottbus

Preface

This volume presents some results of researchers in a rule-based modeling and reasoning community. The maturity of the research in the discipline and the recent development in commercial/industrial rule applications provided an opportunity to produce this workshop.
The workshop aims to be a common space where those with experience or interest in rule modeling, rule languages and rule engines meet researchers with expertise in other areas such as: Artificial Intelligence, Business Process Modeling, Cloud Computing, Intelligent Agents, Model-Driven Architecture, and Semantic Web.
We look on contributions addressing, but not limited to, the following topics:

- Artificial Intelligence Rules and Rule Systems
- Best Practices in Business Rules
- Combining rules and ontologies
- Rules in Enterprise Modeling
- Implemented tools and systems
- Rules and Web Services Integration
- Rules Modeling and Business Processes(including Production Rules and ECA Rules)
- Rule base Visualization, Verbalization, Validation, Verification and Exception Handling.
- Rule-based agents modeling and simulation
- Rule-based modeling of mechanisms, policies, strategies and contracts.
- Rule Engines Architectures
- Rules in Web 2.0 and Enterprise 2.0

This year we accepted five papers from the area of reaction rules applications, rule verification, rule-based reasoning with ontologies, rule engines, and rule-based reasoning with reaction rules. Boehm and Kanne introduced messaging rules as a programming model for enterprise application integration, Sergey Lukichev wrote on the declarative approach for anomaly detection in production rule bases using semantic constraints, Nalepa et al, reports on HeaRT, a complete custom rule runtime environment executing XTT2 rule bases, Papataxiarhis et al, uses SWRL and OWL ontologies to create *i-footman* a knowledge-based framework aiming to provide assistive services to football managers and Emilian Pascalau wrote on buil-in actions and predicates for JSON rules.
The organizers would like to thank all who contributed to the success of the workshop. We thank all authors for submitting papers to the workshop, and we thank the members of the program committee as well as the external reviewers for reviewing and collaboratively discussing the submissions. For the submission and reviewing process we used the EasyChair system, for which the organizers would like to thank Andrei Voronkov, the developer of the system.

Adrian Giurca, Grzegorz J. Nalepa and Gerd Wagner

Program Committee

Grigoris Antoniou, FORTH, Greece

Costin Badica, University of Craiova, Romania

Nick Bassiliades, Aristotle University of Thessaloniki, Greece

Aïcha-Nabila Benharkat, Institut National des Sciences Appliquées de Lyon, France

Jens Dietrich, Massey University, New Zealand

Dragan Gasevic, Athabasca University, Canada

Adrian Giurca, Brandenburg University of Technology, Germany

Ion Iancu, University of Craiova, Romania

Isambo Karali, University of Athens, Greece

Antoni Ligeza, AGH University of Science and Technology, Poland

Grzegorz J. Nalepa, AGH University of Science and Technology, Poland

Viorel Negru, West University of Timisoara, Romania

Adrian Paschke, Freie Universität Berlin

Paula Lavinia Patranjan, SKYTEC AG, Germany

Mark Proctor, Drools, Redhat

Dave Reynolds, Hewlett-Packard Semantic Web Research, England

Kuldar Taveter, Tallinn University of Technology, Estonia

Gerd Wagner, Brandenburg University of Technology, Germany

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

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 [5] 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 [5] 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 [5].

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]. Due to space constraints, we omit a discussion of the system management patterns and refer the interested reader to an extended version of this paper [4], where they are described in detail. In our rule execution engine, most of these system management patterns are provided by an integrated, interactive debugger. This saves developers from manually incorporating system management patterns into their applications.

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 [5]. 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 functions, 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";
```

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, et al. XQuery 1.0: An XML query language. Technical report, W3C, January 2007.
4. Alexander Böhm and Carl-Christian Kanne. Messaging rules as a programming model for enterprise application integration. Technical report, University of Mannheim, 2009. http://db.informatik.uni-mannheim.de/publications/TR-2009-006.pdf.
5. Alexander Böhm and Carl-Christian Kanne. Processes are data: A programming model for distributed applications. In *Web Information Systems Engineering*, 2009.
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 *SAC*, pages 1645–1649, 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 *KES*, 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 *SYNASC*, page 5, 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 *Mobile Object Systems*, pages 49–64, 1996.

The Declarative Approach for Anomaly Detection in Production Rule Bases with Semantic Constraints

Sergey Lukichev

Institute of Informatics, Brandenburg University of Technology at Cottbus
slukichev@canto.de

Abstract. In this paper we present a rule-based (declarative) approach for rule verification. We focus on anomalies, which may appear in rule bases, containing production rules and semantic constraints. The presented approach defines special rules, called verifier rules, which look for anomalies in business rules. The approach is flexible and easy to maintain in the sense that verifier rules can easily be added or modified if new anomalies are found and have to be detected. Examples of verifier rules in Drools syntax are given at the end of the paper.
Keywords: *Verification, integrity constraints, Jena, Drools.*

1 Introduction

It is widely recognized that the process of verification is an important part of quality assurance for rule systems. The common definition of the term "verification" is given in [10]: verification of rules includes checking the knowledge base for logical anomalies such as redundant rules, contradictory rules, and missing rules. Such verification is called anomaly detection. It is important to note that mentioned anomalies are not necessarily errors, but rather potential errors, also known as "attention points ... that might give hints for incorrect rules or wrongly expressed knowledge"[13]. Anomalies may appear in rule bases, for instance, as a result of rules refactoring or due to various communication problems when the knowledge is incorrectly transferred from the business expert to the knowledge engineer. In this paper we focus on two cases of the ambivalence anomaly [10]: The ambivalent rule pair anomaly and the semantic constraint violation by condition and postcondition.

We choose these anomalies because they may appear in rule bases, which contain semantic constraints: special rules, which define an inconsistent state of the knowledge. The main novelty of the work is the use of the declarative approach for anomaly detection. The core of the presented approach are verifier rules, which we introduce in Section 2.2. These verifier rules detect anomalies in production rules, expressed in terms of the production rule metamodel, which is defined in Section 2.3. In Section 3 we define anomalies in production rule bases by means of the model theory. We provide verifier rules, which detect previously defined anomalies in Section 4. In Section 5 we discuss soundness and

completeness of the presented approach. Section 6 references related works on rule verification and concludes the paper.

2 Production Rules and Verifier Rules

Before we start introducing the declarative rule verification approach, we explain what is a knowledge base with production rules and semantic constraints. We define a production rule metamodel (Section 2.3), which is used later in Section 4 on verification as the vocabulary for verifier rules. We introduce the concept of verifier rules and their advantages in Section 2.2.

2.1 Production Rule

According to the OMG Production Rules Representation [3], a production rule is "a statement of programming logic that specifies the execution of one or more actions in the case that its conditions are satisfied" of the form:

if [condition] then [action-list]

We define the condition part of the production rule as a literal conjunction (a conjunction of atoms or negation of atoms) and denote it as $cond(r)$, here r is a production rule. The rule action part defines an ordered list of actions. The OMG PRR [3] defines three state changing actions: Update action, assert action, and retract action. In our approach we consider only two types of actions: Assert and retract. An update action is implemented by a sequence of retract action (remove old facts) and assert action (assert new facts). This consideration with two types of action is common for some rule languages, for instance, Jena 2. The action part of the production rule is also called a postcondition and denoted as $pcond(r)$ where r is a production rule. In this paper a postcondition is a literal conjunction, where a positive atom means the assert action and a negative atom means the retract action.

Definition 1 (Knowledge Base). *A production rule base R together with a vocabulary V of R, semantic constraints C, and a fact set X forms a* knowledge base $KB = \langle X, V, C, R \rangle$.

The fact set X consists of a set of ground facts (variable-free atoms). It is important to note that in our verification approach anomalies are detected by examining the syntax of a rule base and the content of the fact set is not taken into account.

2.2 Verifier Rules for Anomaly Detection

We distinguish between business rules, which have to be verified, and verifier rules, which are used for business rules verification.

A *verifier rule* is a production rule, which is executed when a business rule base contains an anomaly and generates a report about it.

A *rule-based verification approach* employs verifier rules for detecting anomalies in business rules.

The declarative verification approach has a number of advantages such as: *i).* Simplicity of implementation. Anomalies are described by means of special rules, called verifier rules, which are executed by a rule engine. It means that rule-based verification is about writing verifier rules, which is easier than developing and implementing algorithms; *ii).* Easiness of maintenance. When new anomalies are discovered, new verifier rules can be added easily in order to detect them. No additional anomaly-detection programming and algorithm development is required; *iii).* Support for various rule languages. Verifier rules are expressed in terms of a generic rule metamodel, which, if general and flexible enough, can be used for various rule languages.

An example of a verifier rule, expressed informally is "If the body of a rule contains a conjunction of an atom and its negation then this rule contains *unsatisfied condition anomaly*". In contrast to business rules, which are based on terms from a business vocabulary, the vocabulary of verifier rules is based on the rule metamodel. In our example terms "atom", "negation", and "rule body" are from the rule metamodel and not from a business domain.

We express verifier rules using Drools syntax and execute them in the Drools rule engine, however, any other rule language and an engine can be used.

We assume that business rules are verified at the design/development stage [6] and verifier rules are not placed into the system with business rules.

2.3 Production Rule Metamodel

Business rules can be formalized as production rules. Verifier rules, which we present in this paper, are used to find anomalies in production rule bases. In this work we consider a sample metamodel for production rules(Figure 1), which describes the structure of a production rule base with semantic constraints. The metamodel is very close to the metamodel of Jena rules [2], where a rule condition is a conjunction of either triple atoms or built-in atoms. Semantic constraints in Jena can be expressed by means of OWL DL. Therefore, verifier rules, expressed in terms of the metamodel, depicted in Figure 1, can verify Jena rules. Production rule conditions have the expressive power of RDF: to represent the binary existential-conjunctive fragment of predicate calculus.

A rule set consists of rules. A rule has a name and a unique identifier. It has a list of atoms, interpreted as conjunction, as a head and a list of atoms as a body. An abstract class Atom has an identifier and refers to the rule it belongs to.

We distinguish two types of atoms: The BuiltinAtom, which represents various built-ins and TripleAtom. The TripleAtom class has the attribute isNegated and refers to the abstract class Term, which represents subject, predicate and object of the triple atom. Class TripleAtom can represent Jena triples. We interpret a negation as an absence of a fact in the working memory, which is the common interpretation of negation in production rule systems, for instance, in Jena and JBoss Rules.

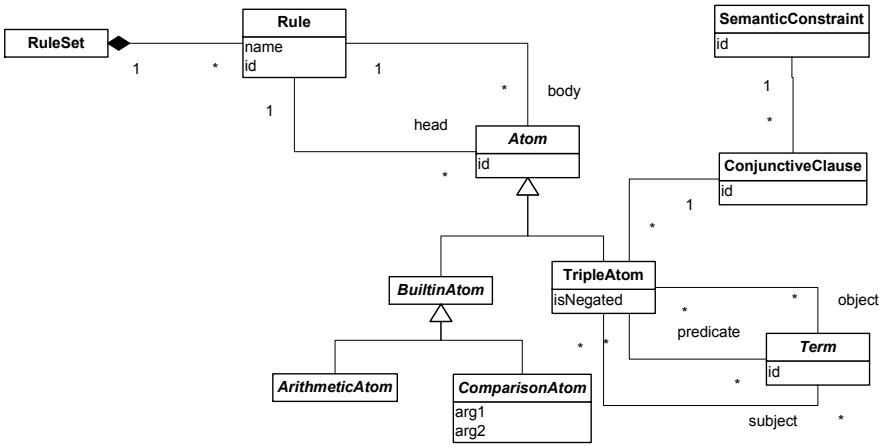

Fig. 1. The generic rule metamodel

A BuiltinAtom is either an ArithemticAtom, which represents various built-ins for arithmetic functions, or a ComparisonAtom, which represents comparison built-ins such as, for instance, greaterThan or lessThan. A SemanticConstraint is a list of ConjunctiveClause's with meaning of disjunction of clauses.

The idea of such structure is to represent constraints in the disjunctive normal form. It allows relatively simple verifier rules for anomaly detection. The practical goal is to verify business rules, expressed in Jena against semantic constraints, expressed in some Description Logic language, for instance, OWL DL. In this paper we consider semantic constraints in the disjunctive normal form and do not explain in details how OWL DL axioms are transformed into logical formulae in DNF. Issues of translating from Description Logic into predicate logic are described in various books and articles ([4], [14]). For instance, in [4] the expressive power of DL is compared against the expressive power of predicate calculus. In particular, it describes the translation from descriptions to predicate calculus, which introduces new variables whenever a new quantifier appears. Here we only give an example of an OWL axiom and corresponding formula in DNF. Let us consider a constraint: *Every eligible driver must have a training certificate*. It is formalized by means of the following OWL DL axiom:

$$\text{Class}(\text{EligibleDriver}, \text{restriction}(\text{hasCert}, \text{someValuesFrom}(\text{TrainingCert})))$$

This DL axiom corresponds to the following predicate logic formula:

$$\forall x(\text{EligibleDriver}(x) \rightarrow \exists y|\text{hasCert}(x, y) \wedge \text{TrainingCert}(y))$$

This implication is equivalent to the following formula in the disjunctive normal form:

$$\forall x \exists y(\neg\text{EligibleDriver}(x) \vee \neg\text{hasCert}(x, y) \vee \neg\text{TrainingCert}(y))$$

A translation of an OWL axiom to the disjunctive normal form is always possible. Existentially quantified variables in the resulting formula in DNF are replaced using skolemization: an existentially quantified variable x is replaced by a function, which takes as parameters all universally quantified variables that precedes x in the formula's quantified variables list.

3 Ambivalence Anomaly

In this section we define the ambivalence anomaly [10], using the model theory. Verifier rules for ambivalence detection are more complex in the sense that they have more atoms in conditions than verifier rules for anomalies, where semantic constraints are not involved (for instance, subsumption, contradictions).

3.1 Semantic Constraint Violation

First, we define a violation of a semantic constraint by a general logical formula. In the following $\models$ denotes the satisfaction relation. $\mathcal{I}_\mathcal{V}$ denotes an interpretation over some valuation.

Definition 2 (Semantic constraint violation by a logical formula). *A formula F* violates semantic constraint *c if the following holds for all interpretations $\mathcal{I}$:*

$$\mathcal{I}_\mathcal{V} \models^t F \text{ then } \mathcal{I}_\mathcal{V} \models^f c$$

In words, if the formula holds then the constraint does not hold. As an example, let us consider a production rule **PR** and a constraint **Constr**:

PR: Eligible(?driver) $\wedge\neg$hasTrainingCertification(?driver, 'true') $\Rightarrow$ HighRiskDriver(?driver)
Constr: Eligible(?driver) $\wedge$ hasTrainingCertification(?driver, 'true')

It is obvious that if the condition of PR holds, then Constr does not hold. The business problem, caused by this anomaly is that the rule with such condition is meaningless since existence of facts, on which it holds, is prohibited by the constraint. Or, in other words, the condition of such rule is never satisfied. It is important to note, that in the example above such variable as ?driver is local at the rule level. The condition of the rule violates the constraint only when the variable is unified with the same value in both the rule and the constraint.

Ambivalent Rule Pair. A rule pair is ambivalent if the condition of one rule subsumes the condition of the other and the conjunction of their postconditions violates the semantic constraint.

Definition 3. *A rule pair $r_1, r_2 \in R$ is* ambivalent *if $cond(r_1)$ subsumes $cond(r_2)$ and there is a semantic constraint $c \in C$ such that the conjunction of postconditions $pcond(r_1) \wedge pcond(r_2)$ violates c.*

The more general case of this anomaly is when conditions of two rules have a non empty intersection, which may lead to firing of both rules for some state.

Let us consider production rules PR1 and PR2 and the semantic constraint Constr1:

PR1 age(?driver, ?x) ∧ ?x> 26 ⇒NormalDriver(?driver)
PR2 age(?driver, ?x) ∧ ?x> 70 ⇒SeniorDriver(?driver)
Constr1 ¬(SeniorDriver(?driver) ∧ NormalDriver(?driver))

The condition of PR1 subsumes the condition of PR2, however, their right-hand sides are different. It means that if these rules are executed, the fact base will contain two facts: "The driver is a normal driver" and "the driver is a senior driver", which is prohibited by the semantic constraint.

Semantic Constraint Violation by Condition and Postcondition A production rule has a semantic constraint violation by condition and postcondition if the state of the fact base after execution of the rule violates a semantic constraint.

Definition 4. *A production rule r has a* semantic constraint violation by condition and postcondition *if exists $c \in C | cond(r) \wedge pcond(r)$ violates c.*

Let us consider the production rule PR3 and the constraint Constr2:

PR3 Provisional(?car) ∧ age(?car, ?x) ∧ ?x> 3 ⇒NotEligible(?car)
Constr2 ¬(NotEligible(?car) ∧ Provisional(?car))

If PR3 is executed then the fact base contains two facts: "a car is provisional" and "a car is not eligible", which is prohibited by the semantic constraint.

4 Verifier Rules for Ambivalence Detection

In this section we define verifier rules in two ways: We give a semi-formal definition of a verifier rule in English, and then express the rule using Drools syntax. Drools syntax ([1]) is technical, but readers, who are familiar with Java programming language and predicate logic can easily understand it. In addition, we give explanations and comment every verifier rule, expressed in Drools syntax. We remind, that semantic constraints are in DNF, i.e. consist of disjunction of conjunctive clauses.

Verifier Rule 1 (Ambivalent rule pair). A rule pair r_1, r_2 is ambivalent if there is a semantic constraint c such that:

1. cond(r_1) subsumes cond(r_2);
2. Every conjunctive clause of c contains an atom, which is oppositely negated to some atom a either from pcond(r_1) or from pcond(r_2);
3. In pcond(r_1) exists an atom, which is in c;
4. In pcond(r_2) exists an atom, which is in c.

Condition 1 is important in order to make sure that rules can be executed on the same facts (See [10] for definition of subsumption). Condition 2 in this rule checks whether constraint c is violated by either atoms from the head of rule 1 or from the head of rule 1. However, this does not guaranty that the conjunction of rule heads violates c, since it is possible that the constraint is violated by the head of just one rule. For instance, if the head of the rule is $A \wedge B$ and constraint is $\neg A \wedge \neg B$ then the constraint is violated no matter of the head of the second rule. Additional conditions 3 and 4 guarantee that each rule head contains at least one oppositely negated atom from c and, therefore, a conjunction of rule heads violates the constraint.

```
rule "Ambivalent rule pair"
    when
        $sc :SemanticConstraint()

        $r1 :Rule()
        $r2 :Rule(id != r1.id)

        # Check that the body of r1 subsumes the body of r2
        forall(
           $atom: Atom(ruleId == $r1.id, body == true)
           DuplicateAtom(
                left == $atom,
                right memberOf $r2.body
           )
        )

        # Check that every conjunctive clause is violated
        # by conjunction of pcond(r1) and pcond(r2)
        forall(
           $clause:ConjunctiveClause(constrId == $sc.id)

           exists(
              $triple:TripleAtom(clauseId == $clause.id)
              or(
                   OppositelyNegatedAtoms(
                     left == $triple,
                     right memberOf $r1.head
                   )

                   OppositelyNegatedAtoms(
                     left == $triple,
                     right memberOf $r2.head
                   )
              )
           )
        )

        # At least one atom from the head of r1 and r2
        # must be in some conjunctive clause of sc
        exists(
           $a1:Atom(ruleId == $r1.id, body == false)

           OppositelyNegatedAtoms(
                 left memberOf $sc,
                 right == $a1
           )
        )
        exists(
           $a1:Atom(ruleId == $r2.id, body == false)

           OppositelyNegatedAtoms(
                    left memberOf $sc,
                    right == $a1
           )
        )
```

```
    then insert(new AmbivalentRulePair( $r1, $r2, $sc ));
end
```

We have to explain some predicates, used in the rule above. `DuplicateAtom` checks for atoms with the same subject, predicate and object. Technically, such atoms can be derived by additional rules of the verifier rule base before the verifier rule is executed. We call such rules "supplementary rules" since they do not detect anomalies, but derive intermediate facts, needed by verifier rules. The predicate `OppositelyNegatedAtoms` checks for conjunctions of an atom and its negation (for instance, $p(s, o) \wedge \neg p(s, o)$). The subsumption check in the rule above works for triple atoms and for built-ins it is different.

Verifier Rule 2 (Semantic constraint violation by condition and postcondition). A rule r violates semantic constraint c by condition and postcondition if

1. Each atom of every conjunctive clause of c has an oppositely negated atom either in $\mathrm{cond}(r)$ or in $\mathrm{pcond}(r)$;
2. At least one atom of $\mathrm{pcond}(r)$ is in c;
3. At least one atom of $\mathrm{cond}(r)$ is in c.

First condition checks whether condition or postcondition of the rule violates every conjunctive clause of the constraint, and therefore, violates the constraint. Conditions 2 and 3 check that the conjunction of condition and postcondition violates the constraint. This verifier rule in Drools syntax:

```
rule "semantic constraint violation by condition and postcondition"
    when
        $sc :SemanticConstraint()

        $r :Rule()

        forall(
           $clause:ConjunctiveClause(constrId == $sc.id)
           exists(
              $triple:TripleAtom(clauseId == $clause.id)
              or(
                   OppositelyNegatedAtoms(
                     left == $triple,
                     right memberOf $r.body
                   )

                   OppositelyNegatedAtoms(
                     left == $triple,
                     right memberOf $r.head
                   )
              )
           )
        )

        # At least one atom from the the body of r and the head of r2
        # must be in some conjunctive clause of sc
        exists(
           $a:Atom(ruleId == $r.id, body == false)

           OppositelyNegatedAtoms(
                 left memberOf $sc,
                 right == $a
           )
        )
```

```
        exists(
           $a1:Atom(ruleId == $r.id, body == true)

           OppositelyNegatedAtoms(
                    left memberOf $sc,
                    right == $a
           )
        )
    then insert( new SemcCVByCondPcond( $r, $sc ));
end
```

5 Soundness and Completeness

The discussion concerning soundness and completeness of the proposed verification solution is about relations between model-theoretic definitions of anomalies (Section 3) and verifier rules, which detect defined anomalies (Section 4). Informally, a solution is sound if it solves the problem for which it is developed. In our case, the solution is a set of verifier rules for the detection of different anomalies. The converse of the soundness property is the completeness property. A solution is complete if its set of verifier rules detects all possible anomalies.

As an example, let us check the soundness of the Verifier Rule 1. Let $A(r_1, r_2, c)$ be an anomaly, detected by Verifier Rule 1 and r_1, r_2 are rules and c is the semantic constraint. We say that this verifier rule is *sound* if the following two conditions hold:

1. r_1 subsumes r_2;
2. $pcond(r_1) \wedge pcond(r_2)$ violates c (as it is defined in Definition 3).

The first condition is based on the definition of rule subsumption. Here we assume that the subsumption of rules is sound. The second condition follows from conditions 2,3,4 of Verifier Rule 1 since these conditions check that $pcond(r_1) \wedge pcond(r_2)$ violates every conjunctive clause of c and, therefore, violates c.

The presented verification approach does not guarantee **completeness**: As it is stated in [8] and further supported in the tutorial and survey on rule verification by O'Keefe [9], "verification, based upon anomaly detection is a heuristic approach, rather than deterministic, for two reasons. First, detected anomalies may not be errors, and errors may exist that are not related to known anomalies. Second, some of the methods for detecting anomalies are themselves heuristic, and thus do not guarantee detection of all identifiable anomalies".

6 Related Works and Conclusions

In this paper we have presented the rule-based approach for production rule verification. The approach is based on verifier rules, which, been expressed in terms of the rule metamodel, analyze the business rule base and derive certain facts if anomalies are detected. We have also demonstrated two verifier rules, which detect two particular types of the ambivalence anomaly: The ambivalent rule pairs anomaly and the semantic constraint violation by condition and postcondition.

There are a lot of works on rule verification, for instance, [5], [11] and others. However, the most related work is the Drools verifier from JBoss [12]. This verifier employs similar declarative approach for anomaly detection. It uses Drools syntax for expressing both verifier rules and production rules. The main difference with the presented approach is that JBoss verifier does not check for anomalies, which include semantic constraints. This is mainly because Drools does not support constraints yet.

The presented verification approach is implemented in the Jena Rule Verifier [7]. The verifier already checks for a number of anomalies and the upcoming further work is towards extensions of its functionality for detection of various anomalies, which include semantic constraints.

References

1. Business Rules Management System Drools. Project homepage. http://www.jboss.org/drools.
2. Jena - A Semantic Web Framework for Java. Project homepage. http://jena.sourceforge.net.
3. Production Rule Representation (PRR). OMG Adopted Specification. http://www.omg.org/spec/PRR/1.0/.
4. Alex Borgida. On the relative expressiveness of description logics and predicate logics. *Artificial Intelligence*, 82:353–367, 1996.
5. G. Castore. Validation and verification for knowledge based control systems. In *Proceedings of the First Annual Workshop on Space Operations, Automation and Robotics, NASA,*, pages 197–202, 1987.
6. Antoni Ligeza. *Logical Foundations for Rule-Based Systems (Studies in Computational Intelligence) (Studies in Computational Intelligence)*. Springer-Verlag New York, Inc., 2006.
7. Sergey Lukichev. The jena rule verifier project, 2009. https://sourceforge.net/projects/jenaruleverifie.
8. L. A. Miller. Dynamic testing of knowledge bases using the heuristic testing approach. *Expert Systems with Applications*, 1(3):249–269, 1990.
9. Robert M. O'Keefe and Daniel E. O'Leary. Expert system verification and validation: a survey and tutorial. *Artificial Intelligence Review*, 7(1):3–42, 1993.
10. A. D. Preece. Foundation and application of knowledge base verification. *International Journal of Intelligent Systems*, 9(8):683–701, 1994.
11. A. D. Preece, R. Shinghal, and A. Bakarekh. Verifying expert systems: a logical framework and a practical tool. *Exp. Stst. Appl.*, 5:421–436, 1992.
12. Toni Rikkola. Rule analytics module, 2008. http://www.jboss.org/community/docs/DOC-11890.
13. Silvie Spreewenberg. Using verification and validation techniques for high-quality business rules. *Business Rules Journal*, 4(2), 2003.
14. Pascal Hitzler Rudi Studer and York Sure. Description logic programs: A practical choice for the modelling of ontologies. In *1st WS on Formal Ontologies meet Industry*, 2005.

HeKatE Rule Runtime and Design Framework*

Grzegorz J. Nalepa, Antoni Ligęza,
Krzysztof Kaczor, Weronika T. Furmańska

Institute of Automatics,
AGH University of Science and Technology,
Al. Mickiewicza 30, 30-059 Kraków, Poland
{gjn,ligeza,kk,wtf}@agh.edu.pl

Abstract. The HeKatE Project aims at providing a complete hierarchical design and implementation framework for rules. Principal ideas of the project include an integrated hierarchical design process covering stages from conceptual, through logical to physical design. These stages are supported by specific knowledge representation methods: ARD+, XTT2, and HMR. Practical design and implementation support using these methods is provided by the HeKatE design environment called HaDEs. A complete custom rule runtime environment HeaRT is provided to run XTT2 rule bases. The engine offers a number of rule-base quality analysis plugins.

1 Introduction

Rule-based systems [1] constitute one of the most powerful and most popular class of intelligent systems. They offer a relatively easy way of knowledge encoding and interpretation. Formalization of knowledge within a rule-based system can be based on mathematical logic or performed on the basis of engineering intuition. Practical design methodologies for intelligent systems remain a field of active development. Developing such a methodology requires an integration of accurate knowledge representation and processing methods [2], as well as practical tools supporting them. Some of the important features of such approaches are: scalable visual design, automatic code generation, support for existing programming frameworks. At the same time quality issues, as well as a formalized description of the designed systems should be considered.

In this paper a new rule runtime and design framework is presented. The *HeKatE* project (see `hekate.ia.agh.edu.pl`) aims at providing an integrated methodology for the design, implementation, and analysis of rule-based systems [1,3]. An important goal of the project is to allow for an easy integration of knowledge and software engineering methods, thus providing a *Hybrid Knowledge Engineering* methodology. The project delivers new knowledge representation methods and practical tools supporting the design process.

* The paper is supported by the HeKatE Project funded from 2007–2009 resources for science as a research project.

The main paradigm for rule representation, namely the eXtended Tabular Trees (XTT) [4], ensures high density and transparency of visual knowledge representation. Contrary to traditional, flat rule-based systems, the XTT approach is focused on *groups of similar rules* rather than single rules. Such groups form decision tables which are connected into a network for inference.

A top-down design methodology based on successive refinement of the project is introduced. It starts with development of an Attribute Relationship Diagram (ARD) which describes relationships among process variables. Based on the ARD model, a scheme of particular XTT tables and links between them are generated. The tables are filled with expert-provided definitions of constraints over the values of attributes; they are in fact the rule preconditions. The code for rules representation is generated and interpreted with provided inference engine. A set of tools supporting the design and development stages is described.

This paper provides an overview of the project, its objectives and tools in Sec. 2. The rule formulation with XTT is shortly described in Sec. 3. Then in Sec. 4 HeKatE design toolchain called HaDEs is introduced. The HeaRT inference engine described in Sec. 5. Then a short comparison to selected existing solutions is given in Sec. 6. Concluding remarks are given in the final section.

2 HeKatE Project Overview

The main principles of the HeKatE project are based on a critical analysis of the state-of-the art of the rule-based systems design (see [5]). They are:

- *Formal Language for Knowledge Representation.* It should have a precise definition of syntax, properties and inference rules. This is crucial for determining its expressive power, and solving formal analysis issues.
- *Internal Knowledge Base Structure.* Rules working within a specific context, are grouped together and form the extended decision tables. These tables are linked together forming a structure which encodes the flow of inference.
- *Systematic Hierarchical Design Procedure.* A complete, well-founded design process that covers the main phases of the system lifecycle, from the initial conceptual design, through the logical formulation, all the way to the physical implementation, is proposed. Verification of the system model w.r.t. critical formal properties, such as determinism and completeness is provided.

In the HeKatE approach the control logic is expressed using forward-chaining decision rules. They form an intelligent rule-based controller or simply a business logic core. The controller logic is decomposed into multiple modules represented by attributive decision tables. The emphasis of the methodology is its possible application to a wide range of intelligent controllers. In this context two main areas have been identified in the project: control systems, in the field of intelligent control, and business rules [6] and in the field of software engineering.

HeKatE introduces a formalized language for rule representation [5]. Instead of simple propositional formulas, the language uses expressions in the so-called

attributive logic [3]. This calculus has stronger expressiveness than the propositional logic, while providing tractable inference procedures for extended decision tables [7]. The current version of the rule language is called XTT^2 [8]. The current version of the logic, adopted for the XTT^2 language, is called ALSV(FD) (*Attributive Logic with Set Values over Finite Domains*).

HeKatE also provides a complete hierarchical *design process* for the creation of the XTT-based rules.

- The main phase of the XTT rule design is called the *logical design*. This phase is supported by a CASE tool called HQed.
- The logical rule design process may be supported by a preceding *conceptual design* phase. In this phase the rule prototypes are built with the use of ARD. The principal idea is to build a graph, modelling functional dependencies between attributes on which the XTT rules are built. The version used in HeKatE is called ARD+ as discussed in [9,10]. The ARD+ design is supported by two visual tools, VARDA and HJed.
- The practical implementation on the XTT rule base is performed in the *physical design* phase. In this stage the visual XTT model is transformed into an algebraic presentation syntax called HMR. A custom inference engine, HeaRT, runs the XTT model.

Let us now shortly describe the main aspects of the XTT rule formalization.

3 Main Aspects of the XTT Rule Language Formalization

The so-called ALSV(FD) attributive logic [3,5] has been introduced with practical applications for rule languages in mind. In fact, the primary aim of the presented language is to extend the notational possibilities and expressive power of the XTT-based tabular rule-based systems [8,5]. Some main concepts of the logic are: attribute, atomic formulae, state representation and rule formulation.

After [3] it is assumed that an *attribute* A_i is a function (or partial function) of the form $A_i\colon O \rightarrow 2^{D_i}$. Here O is a set of objects and D_i is the domain of attribute A_i. As we consider dynamic systems, the values of attributes can change over time (or state of the system). We consider both *simple* attributes of the form $A_i\colon T \rightarrow D_i$ (i.e. taking a single value at any instant of time) and *generalized* ones of the form $A_i\colon T \rightarrow 2^{D_i}$ (i.e. taking a set of values at a time); here T denotes the time domain of discourse.

The *atomic formulae* can have the following four forms: $A_i = d$, $A_i = t$, $A_i \in t$, and $A_i \subseteq t$, where $d \in D$ is an atomic value from the domain D of the attribute and $t \subseteq D$, $t = \{d_1, d_2, \ldots, d_k\}$, is a (finite) set of such values. The *semantics* of $A_i = d$ is straightforward – the attribute takes a single value. The semantics of $A_i = t$ is that the attribute takes *all* the values of t (see [5]).

An important extension in ALSV(FD) over previous versions of the logic [3] consists in allowing for explicit specification of one of the relational symbols $=$,$\neq$,$\in$, $\notin$, $\subseteq$, $\supseteq$, $\sim$ and $\not\sim$ with an argument in the table.

From the logical point of view the *state* is represented by the current values of all attributes specified within the contents of the knowledge-base, as a formula:

$$(A_1 = S_1) \wedge (A_2 = S_2) \wedge \ldots \wedge (A_n = S_n) \tag{1}$$

where A_i are the attributes and S_i are their current values; note that $S_i = d_i$ ($d_i \in D_i$) for simple attributes and $S_i = V_i$, ($V_i \subseteq D_i$) for generalised ones, where D_i is the domain for attribute A_i, $i = 1, 2, \ldots, n$.

Now, consider a set of n attributes $\mathbf{A} = \{A_1, A_2, \ldots, A_n\}$. Any XTT rule is assumed to be of the form:

$$(A_1 \propto_1 V_1) \wedge (A_2 \propto_2 V_2) \wedge \ldots (A_n \propto_n V_n) \longrightarrow RHS$$

where $\propto_i$ is one of the admissible relational symbols in ALSV(FD), and *RHS* is the right-hand side of the rule covering conclusions. In practise the conclusions are restricted to assigning new attribute values, thus changing the system state. State changes trigger external callbacks that allow for communication with the environment. The values that are no longer valid are removed from the state.

Based on the ALSV(FD) logic the XTT rule language is provided [4,8,5]. The language is focused not only on providing an extended syntax for single rules, but also allows for an explicit structurization of the rule base. XTT introduces explicit inference control solutions, allowing for a fine grained and more optimized rule inference than in the classic Rete-like [11] solutions. The representation has a compact and transparent visual representation suitable for visual editors.

Knowledge representation with XTT incorporates extended attributive table format. Similar rules are grouped within separated tables, and the whole system is split into such tables linked by arrows representing the control strategy. Consider a set of m rules incorporating the same attributes $A_1, A_2, \ldots, A_n$: the preconditions can be grouped together and form a regular matrix, as in Table 1.

Table 1. A general scheme of an XTT table

Rule	A_1	A_2	$\ldots$	A_n	H
1	$\propto_{11} t_{11}$	$\propto_{12} t_{12}$	$\ldots$	$\propto_{1n} t_{1n}$	h_1
2	$\propto_{21} t_{21}$	$\propto_{22} t_{22}$	$\ldots$	$\propto_{2n} t_{2n}$	h_2
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
m	$\propto_{m1} t_{m1}$	$\propto_{m2} t_{m2}$	$\ldots$	$\propto_{mn} t_{mn}$	h_m

In Table 1 the symbol $\propto_{ij} \in \{=, \neq, \in, \notin\}$ for simple attributes and $\propto_{ij} \in \{=, \neq, \subseteq, \supseteq, \sim, \not\sim\}$ for the generalized ones. In practical applications, however, the most frequent relations are $=$, $\in$, and $\subseteq$, i.e. the current values of attributes are *restricted* to belong to some specific subsets of the domain.

Efficient inference is assured thanks to firing only rules necessary for achieving the goal. It is achieved by selecting the desired output tables and identifying the

tables necessary to be fired first. The links representing the partial order assure that when passing from a table to another one, the latter can be fired since the former one prepares an appropriate context knowledge. Hence, only rules working in the current context of inference are explored. The partial order between tables allows to avoid examining rules which should be fired later. The details of the complete inference solution for XTT are given in [5].

Let us now move to practical issues concerning both the design and the implementation of XTT-based systems.

4 HaDEs Design Framework

The HeKatE design process is supported by a number of tools. They help with the visual design and the automated implementation of rule-based systems (see `https://ai.ia.agh.edu.pl/wiki/hekate:hades`). The complete framework including the previously discussed methods and tools is depicted in Fig. 1.

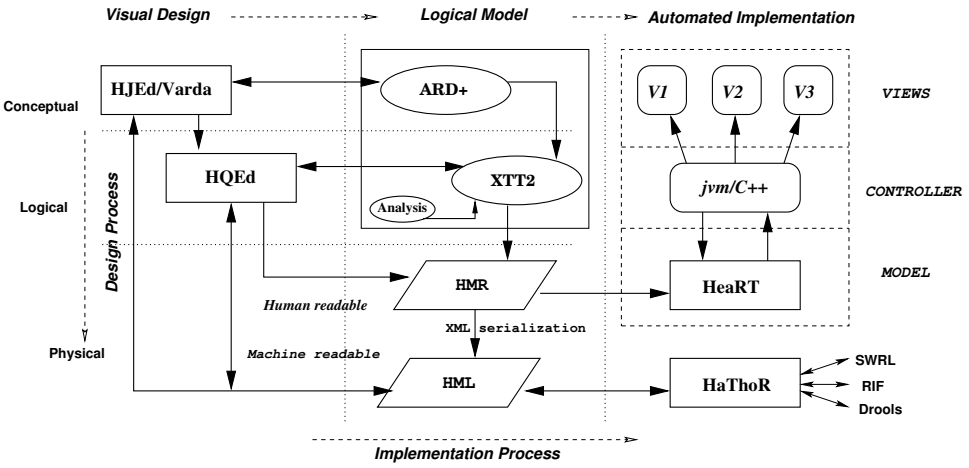

Fig. 1. The complete design and runtime framework

The ARD+ design process is supported by the HJEd visual editor. It is a cross-platform tool implemented in Java. Its main features include the ARD+ diagram creation with on-line design history available through the TPH diagram. An example of a design capturing functional dependencies between system attributes is shown in Fig. 2. Once created, the ARD+ model can be saved in a XML-based HML (HeKatE Markup Language) file. The file can be then imported by the HQEd design tools supporting the logical design.

VARDA is a prototype semivisual editor for the ARD+ diagrams implemented in Prolog, with an on-line model visualization with Graphviz. The tool also supports prototyping of the XTT model, where table headers including a de-

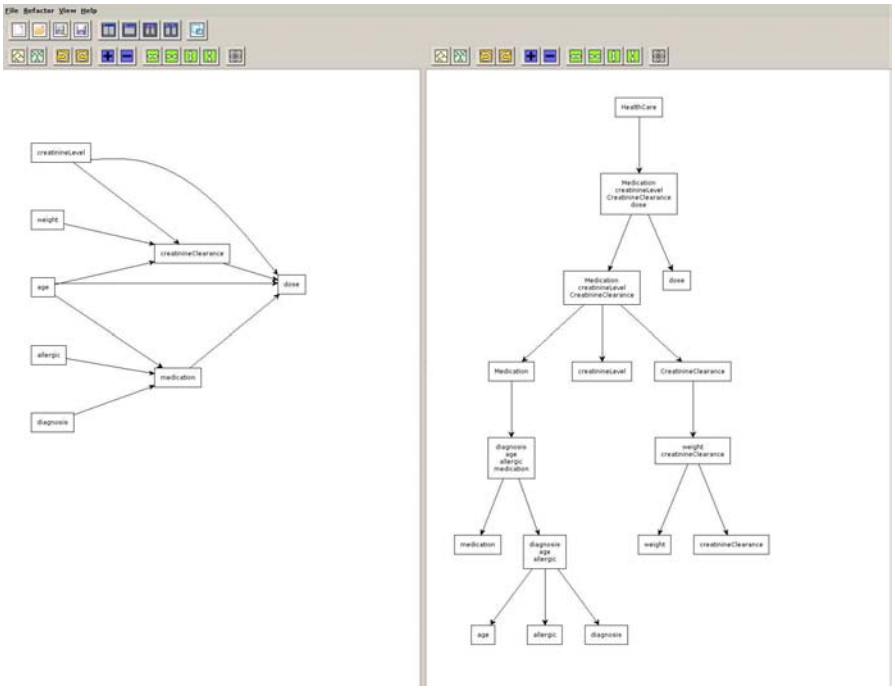

Fig. 2. ARD+ design in HJEd

fault inference structure are created, see Fig. 3. The ARD+ design is described in Prolog, and the resulting model can be stored in HML.

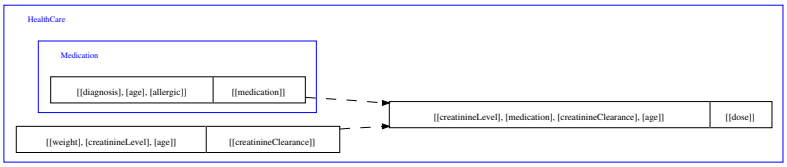

Fig. 3. XTT model generation in VARDA

HQEd provides support for the logical design with XTT, see Fig. 4. It is able to import a HML file with the ARD+ model and generate the XTT prototype. It is also possible to import the prototype generated by VARDA. HQEd allows to edit the XTT structure with on-line support for syntax checking on the table level. Attribute values entered are checked against domains and some possible anomalies are eliminated.

The editor is integrated with a custom inference engine for XTT[2] called HeaRT. The role of the engine is twofold: run the rule logic designed with the

use of the editor, as well as provide on-line formal analysis of the rulebase. The communication uses a custom TCP-based protocol.

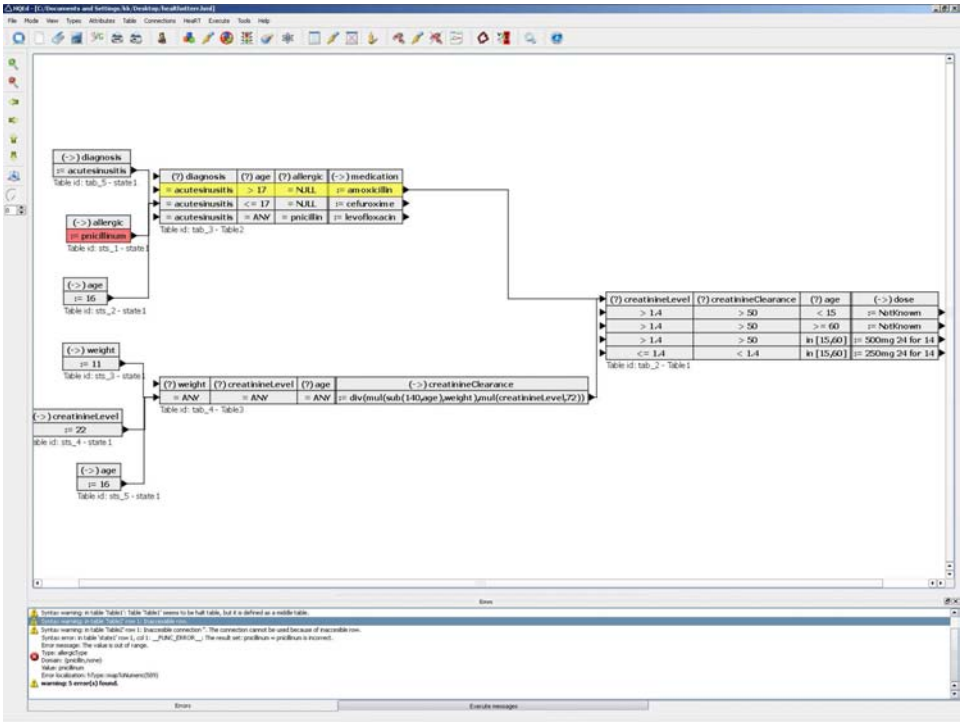

Fig. 4. XTT model edition in HQEd with anomalies detected

HaThoR is the HeKatE rule translation framework. Its goal is to provide rule import and export modules for other languages including RDF and OWL for ARD and RIF and SWRL (possibly R2ML) for XTT. It is mainly implemented in XSLT with some extra plugins integrated with HeaRT implemented in Prolog. An experimental module allows to translate visual XTT representation to a dedicated UML representation using an XMI-based serialization.

5 HeaRT Rule Runtime

HeKatE RunTime (HeaRT) is a dedicated inference engine for the XTT^2 rule bases, (see `https://ai.ia.agh.edu.pl/wiki/hekate:heart`). It is implemented in Prolog in order to directly interpret the HMR representation which is generated by HQEd. HMR (HeKatE Meta Representation) is a textual representation of the XTT^2 logic designed by HQEd. It is a human readable form, as opposed to the machine readable HML format. HeaRT allows to: store and export models in HMR files, and verify HMR syntax and logic. An example excerpt of HMR is:

```
xschm th: [today,hour] ==> [operation].
xrule th/1:
  [today eq workday, hour gt 17] ==> [operation set not_bizhours].
xrule th/4:
  [today eq workday, hour  in [9 to 17]] ==> [operation set bizhours].
```

The first line defines an XTT table scheme, or header, defining all of the attributes used in the table. Its semantics is as follows: "the XTT table *th* has two conditional attributes: *today* and *hour* and one decision attribute: *operation*". Then two examples of rules are given. The second rule can be read as: "Rule with ID *4* in the XTT table called *th*: if value of the attribute *today* equals (=) value *workday* and the value of the attribute *hour* belongs to the range ($\in$) $< 9, 17 >$ then set the value of the attribute *operation* to the value *bizhours*".

The engine implements the inference based on ALSV(FD). It supports four types of inference process, Data and Goal Driven, Fixed Order, and Token Driven [5]. Inference is based on assumption, that the system is deterministic. Conflicts should be handled during design process or detected by verification.

HeaRT also provides a modularized verification framework, also known as HalVA (HeKatE Verification and Analysis). Verification and analysis module implements: simple debugging mechanism that allows tracking system's work, logical verification of models (several plugins are available, including completeness, determinism and redundancy checks), and syntactic analysis of HMR files using a DCG grammar of HMR. The verification plugins can be run from the interpreter or indirectly from HQEd using the communication protocol.

The engine has communication and integration facilities. HeaRT supports Java integration based on callbacks mechanism and Prolog JPL library. It allows for a direct interaction via Prolog console based on callbacks mechanism. HeaRT can operate in two modes, stand-alone and as TCP/IP server, offering TCP/IP integration mechanism with other applications. It is possible to create console or graphical user interface build on Model-View-Controler design pattern.

There are two types of callbacks related to attributes in HMR files. 1) input used to get attribute value from user. This can be done by console or graphical user interface. 2) output used to present an attribute value to user. Callbacks can be use to create GUI with JPL and SWING in Java.

To make HeaRT integration easier, there are three integration libraries, JHeroic, PHeroic or YHeroic. *JHeroic* library was written in Java. Based on JHeroic one can build applets, desktop application or even JSP services. It is also possible to integrate HeaRT with database using ODBC, or Hibernate. *YHeroic* is a library created in Python. It has the same functionality as JHeroic but is easier to use because of Python language nature. *PHeroic* is the same library but created in PHP5. It can be used in a dynamic web page based on PHP.

6 Related Solutions

Here, the focus is on two important solutions: CLIPS and its Java-based incarnation – Jess, as well as Drools, which inherits some of the important CLIPS

features, while providing a number of high-level integration features. Other environments include LPA VisiRule.

XTT provides an expressive, formally defined language to describe rules. The language allows for formally described inference, property analysis, and code generation. Additional callbacks in rule decision provide means to invoke external functions or methods in any language. This feature is superior to those found in both CLIPS/Jess and Drools. On the other hand, the main limitation of the HeKatE approach is the state-base description of the system, where the state is understood as the set of attribute values.

The implicit rule base structure is another feature of XTT. Rules are grouped into decision tables during the design, and the inference control is designed during the conceptual design, and later on refined during the logical design. Therefore, the XTT representation is highly optimized towards rulebase structurization. This feature makes the visual design much more transparent and scalable.

In fact all the Rete-based solutions seek some kind of structurization. In the case of CLIPS it is possible to modularize the rulebase (see chapter 9 in [1]). It is possible to group rules in modules operating in given contexts, and then provide a context switching logic. Drools 5 offers Drools Flow that allows to define rule set and simple control structure determining their execution. In fact this is similar to the XTT-based solution. However, it is a weaker mechanism that does not correspond to table-based solution.

A complete design process seems to be in practice the most important issue. Both CLIPS and Jess are classic expert system shells, providing rule languages, and runtimes. They are not directly connected to any design methodology. The rule language does not have any visual representation, so no complete visual editors are available. Implementation for these systems can be supported by a number of external environments (e.g. Eclipse). However, it is worth emphasizing, that these tools do not visualize the knowledge contained in the rule base.

Drools 5 is decomposed into four main parts: Guvnor, Expert, Flow, Fusion. It offers several support tools, including an Eclipse-based environment. A "design support" feature, is the ability to read Excel files with simple decision tables. While this is a valuable feature, it does not provide constant syntax checking.

It is crucial to emphasize, that there is a fundamental difference between a graphical user interface like the one provided by generic Eclipse-based solutions, and *visual design support and specification* provided by languages such as XTT for rules, and in software engineering by UML. Other dedicated visual rule design languages include URML [12] that provides a UML-based representation for rules. Here focus is on single rules, not on decision tables, like in XTT.

7 Conclusions

The primary area of interest of this paper is to introduce the main concepts of the HeKatE project, its methods and tools. The main motivation behind the project is to speed up and simplify the rule-based systems design process, while assuring the formal quality of the model. The HeKatE design process

is practically supported by a number of tools presented in the paper. These include the HeKatE design environment called HaDEs and the rule runtime called HeaRT. The up-to-date results of the project, as well all the relevant papers are available at the project website see `http://hekate.ia.agh.edu.pl`.

HeKatE project ends in November 2009. Therefore, future work includes a tighter tool integration, as well as modeling complex cases in order to identify possible limitations of the methodology. Providing a comparative studies modelling the same cases in XTT, CLIPS and Drools is planned.

References

1. Giarratano, J.C., Riley, G.D.: Expert Systems. Thomson (2005)
2. van Harmelen, F., Lifschitz, V., Porter, B., eds.: Handbook of Knowledge Representation. Elsevier Science (2007)
3. Ligęza, A.: Logical Foundations for Rule-Based Systems. Springer-Verlag, Berlin, Heidelberg (2006)
4. Nalepa, G.J., Ligęza, A.: A graphical tabular model for rule-based logic programming and verification. Systems Science **31**(2) (2005) 89–95
5. Nalepa, G.J., Ligęza, A.: Hekate methodology, hybrid engineering of intelligent systems. International Journal of Applied Mathematics and Computer Science (2009) accepted for publication.
6. Ross, R.G.: Principles of the Business Rule Approach. 1 edn. Addison-Wesley Professional (2003)
7. Ligęza, A., Nalepa, G.J.: Knowledge representation with granular attributive logic for XTT-based expert systems. In Wilson, D.C., Sutcliffe, G.C.J., FLAIRS, eds.: FLAIRS-20 : Proceedings of the 20th International Florida Artificial Intelligence Research Society Conference : Key West, Florida, May 7-9, 2007, Menlo Park, California, Florida Artificial Intelligence Research Society, AAAI Press (may 2007) 530–535
8. Nalepa, G.J., Ligęza, A.: Xtt+ rule design using the alsv(fd). In Giurca, A., Analyti, A., Wagner, G., eds.: ECAI 2008: 18th European Conference on Artificial Intelligence: 2nd East European Workshop on Rule-based applications, RuleApps2008: Patras, 22 July 2008, Patras, University of Patras (2008) 11–15
9. Nalepa, G.J., Ligęza, A.: Conceptual modelling and automated implementation of rule-based systems. In: Software engineering : evolution and emerging technologies. Volume 130 of Frontiers in Artificial Intelligence and Applications. IOS Press, Amsterdam (2005) 330–340
10. Nalepa, G.J., Wojnicki, I.: Towards formalization of ARD+ conceptual design and refinement method. In Wilson, D.C., Lane, H.C., eds.: FLAIRS-21: Proceedings of the twenty-first international Florida Artificial Intelligence Research Society conference: 15–17 may 2008, Coconut Grove, Florida, USA, Menlo Park, California, AAAI Press (2008) 353–358
11. Forgy, C.: Rete: A fast algorithm for the many patterns/many objects match problem. Artif. Intell. **19**(1) (1982) 17–37
12. Lukichev, S., Wagner, G.: Visual rules modeling. In: Sixth International Andrei Ershov Memorial Conference Perspectives of System Informatics, Novosibirsk, Russia, June 2006. LNCS, Springer (2005)

i-footman: A Knowledge-Based Framework for Football Managers

Vassilis Papataxiarhis, Vassileios Tsetsos, Isambo Karali, Panagiotis Stamatopoulos, Stathes Hadjiefthymiades

Department of Informatics and Telecommunications, National and Kapodistrian University of Athens, Panepistimiopolis, Ilissia, GR-15784, Athens, Greece, {vpap, b.tsetsos, izambo, takis, shadj}@di.uoa.gr

Abstract. i-footman constitutes a knowledge-based framework aiming to provide assistive services to football managers. More precisely, the system accommodates a number of managing processes (e.g., selection of team composition) in the context of football by adopting a declarative approach. It also takes advantage of Semantic Web technologies in order to represent and manage the required application models. i-footman is based on a flexible architecture that facilitates possible extensions in the functionality and the quality of the provided services. The paper presents the overall architecture as well as certain implementation details of the system.

Keywords: knowledge-based system, reasoning methods, ontology, rules

1 Introduction

This work presents an extensible knowledge-based framework aiming to provide team management services in the context of football. The system allows the development of services that can be used by football managers. Specifically, i-footman (*i*ntelligent *foot*ball *man*ager) supports some pre- and in-game decisions needed to be taken by the user. The basic functionality of i-footman consists of proposing a "good" tactical formation, an effective composition of players and certain tactical instructions.

Football does not constitute a scientific field or a domain with explicitly expressed and commonly accepted knowledge. This is due to the fact that knowledge about football does not stem only from expertise but also from experience. Moreover, there are times when domain experts (i.e. football managers) act differently to each other in order to face a certain situation (e.g., suspension of a player) making the knowledge about football subjective. Hence, i-footman does not aim to capture the complete knowledge about the application domain of football, but to provide effective means of adapting and extending the required knowledge according to the user needs.

The suggestions of the system are based on domain-specific knowledge such as the opponent's team tactic and the features of the available players. Such kind of knowledge had to be modeled through appropriate application models. However, no such model was identified in the relevant literature. Additionally, i-footman is a

decision support system and it calls for effective modeling of human knowledge and expertise. Hence, the system had to exploit proper knowledge technologies and representation formalisms that would facilitate the extensibility of the knowledge base. Consequently, modern techniques (e.g. Semantic Web technologies) were adopted in the context of i-footman in order to represent knowledge through expressive languages. The adoption of rules and ontologies take advantage of the declarative programming paradigm by exploiting natural forms of knowledge.

The rest of this paper is organized as follows. Section 2 discusses the architecture of i-footman and some modeling issues, as well. A functionality and performance evaluation that has been performed in a simulated environment is presented in Section 3. Finally, some concluding remarks are provided in the last section of the paper.

2 System Architecture and Application Models

Two ontological models have been developed in the context of i-footman for modeling football players and teams, accordingly. Furthermore, a large set of rules was composed in order to express more complex concepts and relations. Two domain experts were interviewed in order to acquire the domain knowledge captured by models and rules[1]. The Pellet reasoner (v. 1.5.1) [6] that is based on Description Logics (DLs) [1] is responsible for reasoning over the ontologies while the rule engine Jess [4] performs the execution of rules. The selection of the reasoning modules is based on the performance evaluation presented in [5]. The integration of ontologies and rules is not a straightforward task, since there is no single reasoning module that can seamlessly handle both formalisms. Hence, i-footman performs reasoning tasks in a sequential manner and the results of the ontological reasoning are provided as input to the rules execution process. A generic view of the framework architecture is depicted in Fig. 1.

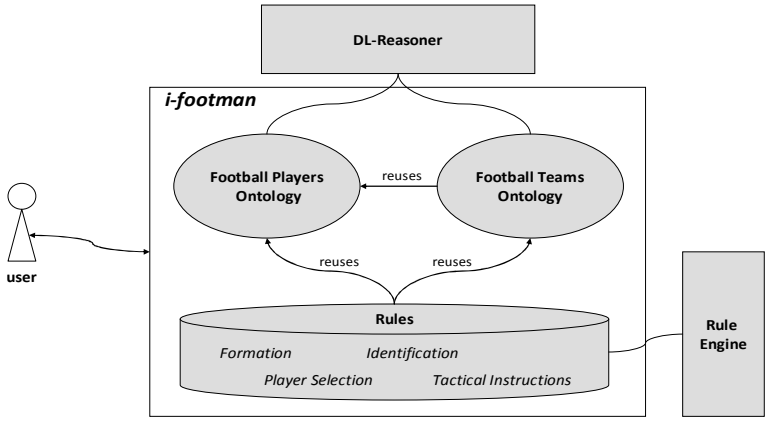

Fig. 1. i-footman Architecture

[1] All the application models and rules developed in the context of i-footman are provided online in *http://www.di.uoa.gr/~vpap/i-footman/*

Football Players Ontology

The Football Player Ontology (FPO) has been developed in Ontology Web Language (OWL) [2] and, in particular, the OWL-DL version. It is the first ontology, according to our knowledge, that focuses on the description of football players. FPO provides an extensive vocabulary referring to the various player characteristics, focusing on the features used by a manager in order to make a tactical decision, and the tactical instructions that a football player could follow. Some key concepts of the ontology concern the football players ("Football_Player"), the positions they can hold during a football match ("Position") and the abilities of the footballers ("Ability"). Each ability that describes a certain player is assigned a value denoting its respective quality. This fact facilitates the definition of complex concepts through conditions of equivalence. For instance, the following statement defines the concept of "creative" middlefielders:

$$\text{fpo:CreativeMiddlefielder} \equiv (\text{fpo:hasPassing.GoodAbility} \sqcup \text{fpo:hasPassing.VeryGoodAbility}) \sqcap \text{fpo:playsInPosition.Middlefielder} \quad (1)$$

Football Teams Ontology

The Football Teams Ontology (FTO) provides the terms for describing the main features of football teams and the tactical guidelines they could follow. This vocabulary is also expressed in terms of OWL-DL and is strongly related to the FPO. Specifically, the FTO reuses the FPO vocabulary extending it properly.

The FTO terms could be distinguished to three parts. Firstly, the ontology defines some generic team features (e.g., the players of a team). Secondly, the ontology takes advantage of DLs in order to classify the various team instances into certain categories (e.g., teams that attack from the wings). Finally, the FTO models the various tactical instructions allowing the execution of rules that will follow.

Rules

The rules constitute the declarative part of the knowledge base and are expressed in terms of the Semantic Web Rule Language (SWRL) [3]. The basic idea behind the adoption of SWRL was the combination of ontologies and rules in the same logical language. The rules reuse the vocabulary provided by the FPO and FTO in order to define more complex relationships and their structure was based on the knowledge acquired by the experts. They can also be distinguished in four main categories[2]:

1. Identification Rules. These rules aim to identify the weaknesses and the advantages of the opponent. They are mainly based on the opponent's team formation and the features of players in order to deduce the capabilities of the opponent. For instance, the following SWRL rule expresses the knowledge that a team plays well the counter attack when two or more of its offensive players are quick[3]:

[2] For space limitation reasons, a simplified version of the rules is presented here. The complete form of rules can be found online in *http://www.di.uoa.gr/~vpap/i-footman/rules.owl*.

[3] QuickOffensivePlayer is an FPO concept defined through necessary and sufficient conditions.

$$\text{fto:hasStartingPlayer}(?t1,?p1) \wedge \text{fto:hasStartingPlayer}(?t1,?p2) \wedge \text{fpo:QuickOffensivePlayer}(?p1) \wedge \text{fpo:QuickOffensivePlayer}(?p2) \rightarrow \text{fto:dangerousAtCounterAttack}(?t1,\text{true}). \quad (2)$$

2. Formation Rules. They are responsible for specifying the tactical formation that the team will follow during a match. In particular, the number of defenders, middlefielders and attackers is determined as well as the positions that they should cover. The following rule describes a case where three central defenders are used:

$$\text{fto:myTeamPlaysAgainst}(?t1,?t2) \wedge \text{fto:TeamWith3CentralDefenders}(?t2) \wedge \text{fto:TeamWith3CentralPlayers}(?t2) \wedge \text{fto:TeamWithSideMFs}(?t2) \wedge \text{fto:TeamWith2Attackers}(?t2) \rightarrow \text{fto:playsWith3CentralDefenders}(?t1, \text{true}). \quad (3)$$

3. Player Selection Rules. They make use of players' features and tactical formation of the teams. i-footman proposes the appropriate players to form the team's composition according to the classification that has been already completed through the ontology reasoning processes. A typical player selection rule follows:

$$\text{fto:myTeamPlaysAgainst}(?t1,?t2) \wedge \text{fpo:playsWith1Striker}(?t1) \wedge \text{fpo:GoodStriker}(?p1) \wedge \text{fpo:isMemberOf}(?p1,?t1) \rightarrow \text{fpo:isSuggestedTo}(?p1,?t1). \quad (4)$$

4. Tactical Instructions Rules. They specify the tactical instructions suggested by the system. These rules take advantage of the opponent weaknesses and strengths specified by the identification rules and information coming from formation rules. The following rule denotes that if a team does not use side defenders then the opponent should identify this weakness and attack from the wings.

$$\text{fto:myTeamPlaysAgainst}(?t1,?t2) \wedge \text{fto:TeamWithNoBacks}(?t2) \wedge \text{fto:TeamWithWingers}(?t1) \rightarrow \text{fto:shouldAttackFromTheWings}(?t1, \text{true}). \quad (5)$$

3 Evaluation

Since no actual data and statistics were available during the development of i-footman, the evaluation was accomplished through simulations. Specifically, the simulation is based on two scenarios and takes advantage of two computer games that focus on the tactical management of football teams: the Championship Manager 2008 (Eidos) and Football Manager 2008 (Sports Interactive). Such computer software simulate a football match according to the tactical instructions that have been given as input to both teams and generates a final result for that game. Moreover, the platforms provided some players' and teams' statistics that have been inserted to the ontologies.

Regarding the first evaluation scenario, two teams with similar ratings (according to the evaluation provided by each platform) were selected. Specifically, Barcelona FC (Spain) was guided by i-footman while the computer handled the team of Real Madrid FC (Spain). 40 games were simulated in each platform (80 in total). The computer handled both teams in the first 20 games (in each platform) with no external interference while i-footman managed the team of Barcelona in the next 20 games.

The match results of the first scenario are presented in Fig. 2. In a total of 40 matches without the intervention of i-footman, Barcelona won 14 times, Real Madrid

won 16 times and 10 games came to draw. Furthermore, Barcelona scored 45 goals while Real Madrid scored 61 goals. Afterwards, the same number of matches was executed with i-footman controlling Barcelona and the results seemed to be in favour of Barcelona. Although the number of Barcelona's wins did not increased substantially (only by 1), the number of losses decreased by 50% (8 instead of 16). Furthermore, Barcelona scored 51 times (instead of 45) while opponent scored 36 (instead of 61). This means that the recommendations of i-footman lead to significant improvement of the team performance.

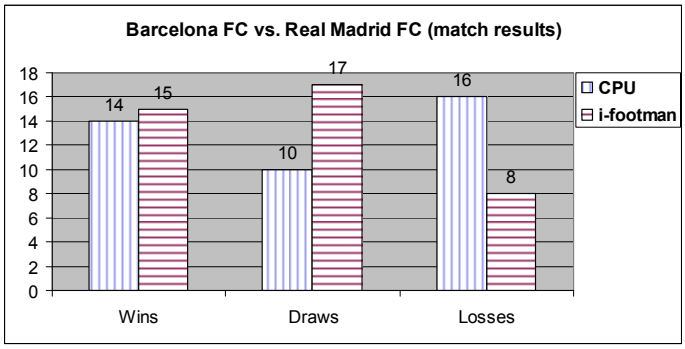

Fig. 2. Match results of the first scenario

The second scenario concerned the evaluation of i-footman when playing against an opponent with better ratings than the one managed by the system. Specifically, Olympiacos FC (Greece) was handled by the system and played against Real Madrid. The results of this scenario were very similar to the former with a minor improvement of the total number of wins accomplished by i-footman.

Generally, i-footman seems to perform well in such simulation environments. Although the performance of the user's team does not improve substantially, the adoption of i-footman seems to tackle the opponent's performance successfully. This stems from the fact that both goals and wins of the opponent's team are obviously decreased. The large number of rules that refer to the defending strategy of the team may lead to that. Specifically, the tactical formation and composition of the team are mainly adapted to the advantages and less to the weaknesses of the opponent's team. This could be improved by extending and modifying the rules of the knowledge base (possibly through interviews of more domain experts).

Regarding the time performance of the framework, the expected response time was 7741ms (as measured on a typical Desktop PC). Such time seems reasonable since it includes all the reasoning processes (i.e. hierarchy classification, instance checking of FPO and FTO ontologies and rules execution) that were performed.

4 Conclusions and Future Work

We have presented a knowledge-based framework able to support decision making in the context of football. More precisely, the paper focuses on the architecture of i-footman, its application models and the simulation results derived during the evaluation process. The main issue that has been identified during development was

the lack of an integrated reasoning framework capable of handling ontologies and rules. Today, there is no efficient reasoning module that can reason over both formalisms seamlessly. Hence, the developer has to perform ontological reasoning and provide the results as input to the rule engine and vice versa in order to achieve effective knowledge management.

As described, i-footman was designed to facilitate the addition of possible extensions regarding its functionality by modifying the knowledge used to describe the application domain of football. This process would be facilitated by the exploitation of learning techniques that target to automate the generation of the declarative part of the knowledge base (i.e. the rules). This could be achieved either by accessing actual statistical data about players and teams or by taking the results that arise from the simulation platforms as real. Since no real data are available, the exploitation of learning algorithms over virtual data seems to be more feasible. Finally, more expressive knowledge representation languages that support fuzziness could be adopted in order to deal with knowledge uncertainty issues.

Acknowledgements

This work is supported by the Special Research Grants Account of the University of Athens through the Kapodistrias Programme (Research Grant Number: 70/4/7819).

The authors would also like to thank the football managers Panagiotis Lemonis (Olympiacos FC, 2007/08) and Nikolaos Nioplias (Greece Under 21's National Team, 2007/08) who provided basic knowledge for understanding the domain of football and valuable comments towards the improvement of this work.

References

1. Baader, F., Calvanese, D., McGiuness, D., Nardi, D., Patel-Schneider, P.F.: The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge: Cambridge University Press (2003)
2. Dean, M., Schreiber, G., Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D.L., Patel-Schneider, P.F., Stein, L.A.: *OWL Web Ontology Language Reference.* W3C Recommendation 10 February 2004. Retrieved April 22, 2008, from http://www.w3.org/TR/owl-ref/ (2004)
3. Horrocks, I., Patel-Schneider, P., Harold, B., Tabet, S., Grosof, B., Dean, M.: SWRL: A Semantic Web Rule Language Combining OWL and RuleML. World Wide Web Consortium Member Submission, http://www.w3.org/Submission/SWRL/ (2004)
4. Jess, The Rule Engine For the Java Platform. Retrieved April 22, 2008, from http://www.jessrules.com/jess/index.shtml (2008)
5. Papataxiarhis, V., Tsetsos, V., Karali, I., Stamatopoulos, P., Hadjiefthymiades, S.: "Developing rule-based applications for the Web: Methodologies and Tools", chapter in ***"Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches"*** (Eds. Adrian Giurca, Dragan Gasevic and Kuldar Taveter), Information Science Reference (2009)
6. Sirin, E., Parsia, B., Grau, B., C., Kalyanpur A., & Katz, Y. Pellet: A practical OWL-DL reasoner, *Journal of Web Semantics*, 5(2) (2007)

Built-ins for JSON Rules

Emilian Pascalau[1]

Hasso Plattner Institute, Germany
emilian.pascalau@hpi.uni-potsdam.de

Abstract. JSON Rules is a declarative rule language for the World Wide Web. It has been created to satisfy at least the following list of requirements: (1) create and execute rules in browser; (2) support for ECA and PR rules; (3) the Working Memory contains event-facts. Here we extend the language with the concept of built-ins (predicates and actions). We focus on the relation with RIF-DTB, however with a strong emphasis on the environment where the rules are going to be executed: the web browser. As such we introduce here an initial set of built-ins, as well as the architectural aspects that should be taken into consideration for an engine implementing the JSON Rules language.

1 Introduction

There is a multitude of rule languages out there (i.e. Drools [1], F-Logic [2]) each of them with their specificities. However interoperability in a world that moves at high speeds and where business are interconnected is a must. RIF is the W3C Rule Interchange Format[1] and also the name of the W3C group that it is in charge of this standard. Although the main goal of RIF is to provide a rule interchange language, it is more than that. RIF provides different versions, called dialects in order to tackle the serious trade-offs in nowadays rule languages. Almost a mandatory request for any rule language it is to provide at least the guidelines for translating to and from RIF.

JSON Rules introduced initially in [3] and later extended in [4] is a rule language designed to be run in a web browser and to model web based scenarios. The main goals of the language are: to move the reasoning process to the client-side; to offer support for intelligent user interfaces; to handle business workflows; to allow rule-based reasoning with semantic data inside HTML pages (reasoning with RDFa [5]); to create intelligent mashups - rule based mashups; to enable users to collaborate and share information on the WWW (rule sharing and interchange). JSON Rules fulfills the following requirements: rules run in the browser; uses Event-Condition-Action rules; rules are defined on top of DOM structure; uses DOM events plus user defined events; actions are defined by users and can be any JavaScript function calls. JSON Rules language uses a special type of Working Memory (WM) – the Document Object Model (DOM) [6] of the page. As in any forward chaining engine, the main effect of JSON

[1] http://www.w3.org/2005/rules/

rules execution is the update of the WM i.e. the DOM of the page. Therefore, the language design is tailored to the environment where rules are going to be executed.

With respect to its condition language, JSON Rules was influenced in its syntax by other rule languages such as Drools [1]. Any valid JavaScript function call is allowed as actions. This covers also the OMG Production Rule Representation ([7]) and RIF Production Rule Dialect [8].

Any JSON Rule is identified by an `id`. A rule may have a `priority` - if this attribute is missing then its implicit value is `1`. The rule engine executes the rules in a down counting order (from greater priority to lower priority). However, the execution order for rules with the same priority is irrelevant. One significant property of a JSON Rule is the `appliesTo` property. This property stores an array of URLs to which this rule refers to i.e. a list of URLs defining the context in which this rule can be applied. Based on this property, rules are grouped in rule sets.

An `EventExpression` is an expression capturing any DOM Event [9]. Therefore, it has a `type`, `target`, `timeStamp` and `phase` properties, each of them allowing a `JSONTerm` as a value. Inside the rule engine, `EventExpression` is matched with DOM Events at their occurrence (we allow only atomic events with no duration i.e. the events are consumed immediately when they occur) time. Consuming events yields into event-facts creation in the WM and such facts are immediately consumed. The engine itself is event-based i.e. all internal processes and activities are event-driven. This is imposed by the environment where the engine runs - the Web browser. Generally, the engine consumes events, through its Working Memory, check conditions and execute actions.

A JSON Rule may contain a list of conditions, logically interpreted as a conjunction of atoms. The language provides the following types of atoms:
`Description`, `JavaScriptBooleanExpression`, `XPathCondition`, `NodeEquality` and `Negation`. The `Description` conditional is similar to Drools [1] pattern. With such conditional property restrictions can be defined as well as property bindings, in a Drools like fashion. `JavaScriptBooleanCondition` stands for JavaScript boolean expression; any JavaScript boolean expression can be used.
The `XPathCondition` is used to test that a DOM Node is found in the list of nodes returned by evaluating an XPath expression. The last two `NodeEquality` and `Negation` are pretty much obvious. They are used to test equality between two terms, respectively to negate a conditional atom.

As usual, a rule has associated a list of actions that have to be performed, if the rule conditions hold. An action is a call to any available JavaScript function. Actions are executed sequentially. The reader may notice that such kind of actions can determine also side effects i.e. more than simple updates on the Working Memory (communication with a server that has no effect on the WM).

As stated above since it is almost mandatory for any rule language to provide means of translation to and from RIF this paper makes the first steps towards

translation from RIF to JSON Rules and vice versa by tackling the problem of built-ins.

2 Built-ins

This section deals with built-ins for the JSON Rules language, along with the architectural and technical aspects imposed by the context (web browsers) and the programming language (JavaScript, ECMAScript [10]) in which the JSON Rules [3] engine is running and has been implemented.

Built-in means constructed as a non-detachable part of a larger structure. In case of rule systems built-ins encapsulate commonly used functionality, provided as predicates or functions. Built-ins also help in maintaining the clean declarative design of rules.

As described in the Conclusion of RIF-UCR [11] *"the goal of the RIF working group is to provide representational interchange formats for processes based on the use of rules and rule-based systems. These formats act as "interlingua" to interchange rules and integrate with other languages, in particular (Semantic) Web mark-up languages."*

JSON Rules foresees compliance with Rule Interchange Format (RIF)[2]. As such the JSON Rules aims to provide the built-ins defined in RIF-DTB [12], beside others that refer to the mashup context.

2.1 RIF built-ins short intro

RIF-DTB [12] is the reference document concerning built-in datatypes, predicates, functions supposed to be supported by RIF dialects such as: RIF Core Dialect [13], RIF Basic Logic Dialect [14] and RIF Production Rules Dialect [8]. According to RIF-DTB document each dialect takes use of a superset or a subset of datatypes, predicates or functions defined in RIF-DTB. The datatypes taken into account by the RIF-DTB [12] are imported either from *W3C XML Schema Definition Language (XSD)* [15] or from *rdf:PlainLiteral: A Datatype for RDF Plain Literals* [16]. Predicates or functions have been ported from *XQuery 1.0 and XPath 2.0 Functions and Operators* [17] or from *rdf:PlainLiteral: A Datatype for RDF Plain Literals* [16].

The list of predicates and function built-ins taken into consideration by RIF-DTB comprises among others: predicates for `literal` comparison, `numeric` functions and predicates, function and predicates on `boolean` values, on `string`, on date/time and duration, on `XMLLiterals`, on `rdf:PlainLiteral`.

2.2 JSON Rules context prerequisites

The general technical guide line that governs the architectural aspects and any other aspects is the compliance and conformity with the ECMAScript standard [10].

[2] `http://www.w3.org/2005/rules/`

Beside the built-ins (predicates and functions) classification introduced in RIF-DTB [12] that span all the RIF dialects, we introduce here a set of built-in actions that serve a different purpose, mainly actions needed in dealing with mashups. As introduced in [4] one of the main purposes of JSON Rules is to be an execution language for mashups.

Although we are not going to address here the problem of translating RIF constants' names, symbols, or namespaces, minimal introduction of ECMAScript concepts are necessary.

One of the most important concept is the *closure* concept. The ECMAScript [10] standard explains a *closure* as a *"function with some arguments already bound to values"*. Others define a closure[3] as *"an expression (typically a function) that can have free variables together with an environment that binds those variables (that "closes" the expression)"*.

As stated in [18] *"JavaScript's extreme dynamism equips it with tremendous flexibility, and one of the most interesting yet least understood facets of its dynamism involves context"*.

On top of these two concepts (*closure*, *context*) the concept of *namespace* is defined. We understand by the notion of namespace hierarchies of nested objects as defined also in [18].

Another characteristic of JavaScript is that it is weakly typed.

2.3 Guidelines for defining JSON Rules built-ins

There are a couple of general guidelines that should be taken into account when defining functions and predicates, either user defined or built-in:(1) in order to avoid name clashing in JavaScript global context it is recommended to use proper namespaces; (2) predicates should always return a default value, `undefined` value should be avoided.

To simplify the process of creating namespaces JavaScript libraries such as Dojo[4] could be used.

Because the `function` word is reserved in JavaScript its usage should be avoided in a different context other than defining a JavaScript function.

Another important issue that must be taken into account is that the JavaScript code for a page runs in a common context, and as such any redefinition of the same object will override the initial implementation.

2.4 From RIF-DTB to JSON Rules built-ins

This section explains how RIF built-in datatypes, predicates and functions are translated into JSON Rules, respectively JavaScript.

In order to preserve the semantics type mapping is necessary. Table 1 defines the mapping between RIF built-in datatypes and JavaScript datatypes. E4X [19] datatype is also taken into account. Since JavaScript is weakly typed in most

[3] `http://www.jibbering.com/faq/faq_notes/closures.html`
[4] `http://www.dojotoolkit.org/`

of the cases a custom type must be built. In Table 1 this is emphasized by the `cust. type` notation. The same table specifies for each custom type, the base JavaScript type or types on which the custom type must be based.

Table 1. Mapping of RIF built-in Datatypes to JavaScript Datatypes

RIF datatype	**JavaScript datatype**	**RIF datatype**	**JavaScript datatype**
xs:string	String	xs:nonNegativeInteger	Number, cust. type
xs:normalizedString	String, cust. type	xs:unsignedLong	Number, cust. type
xs:token	String, cust. type	xs:unsignedInt	Number, cust. type
xs:language	String, cust. type	xs:unsignedShort	Number, cust. type
xs:Name	String, cust. type	xs:unsignedByte	Number, cust. type
xs:NCName	String, cust. type	xs:positiveInteger	Number, cust. type
xs:ID	String, cust. type	xs:decimal	Number, cust. type
xs:IDREF	String, cust. type	xs:boolean	Boolean
xs:NMTOKEN	String, cust. type	xs:dateTime	Date
xs:ENTITY	String, cust. type	xs:date	Date, cust. type
xs:NOTATION	String, cust. type	xs:time	Date, cust. type
xs:anyURI	String, cust. type	xs:gYearMonth	Date, cust. type
xs:hexBinary	Array, cust. type	xs:gMonthDay	Date, cust. type
xs:base64Binary	Array, cust. type	xs:gYear	Date, cust. type
xs:float	Number, cust. type	xs:gDay	Date, cust. type
xs:double	Number	xs:gMonth	Date, cust. type
xs:duration	String+Date, cust. type	xs:NMTOKENS	String+Array, cust. type
xs:integer	Number, cust. type	xs:IDREFS	String+Array, cust. type
xs:nonPositiveInteger	Number, cust. type	xs:ENTITIES	String+Array, cust. type
xs:negativeInteger	Number, cust. type	xs:QName	String, cust. type
xs:long	Number, cust. type	xs:anyType	E4X XML object
xs:int	Number, cust. type	rdf:PlainLiteral	String
xs:short	Number, cust. type	rdf:XMLLiteral	E4X XML object
xs:byte	Number, cust. type		

As previously stated datatypes as well as functions and predicates must be grouped by means of proper namespaces. In the RIF-DTB case datatypes are identified by the following namespace: `http://www.w3.org/2001/XMLSchema#`. As convention this gets translated into `org.w3c.xs`. For example the `xs:duration` datatype is identified with: `http://www.w3.org/2001/XMLSchema#duration`. This one gets translated into `org.w3c.xs.Duration`. The namespace for RDF datatypes is `org.w3c.rdf`.

In a similar way the namespaces for the RIF predicates and functions are: `org.w3c.rif.pred` and respectively `org.w3c.rif.func`.

The implementation for custom datatypes is inspired from J2EE implementation (i.e. `javax.xml.datatype.Duration`). The `Duration` custom type provides the following list of methods: `add(org.w3c.xs.Duration rhs)`; `addTo(Date date)`; `compare(org.w3c.xs.Duration duration)`; `getDays()`; `getHours()`; `getMinutes()`; `getMonths()`; `getSeconds()`; `getSign()`; `getYears()`; `isLongerThan(org.w3c.xs.Duration duration)`; `isShorterThan(org.w3c.xs.Duration duration)`; `negate()`.

In the RIF-DTB context built-in predicates and functions are defined with the following artifacts: (1) name of the built-in; (2) external schema of the built-in (the signature of the built-in); (3) for a RIF built-in function - how it maps its arguments into a result - in RIF terms this means the mapping of $I_{external}(\sigma)$ in the formal semantics of [20] and [14]; (4) for a RIF built-in predicate - how it gives the truth value when arguments are substituted with values from the domain - this corresponds to the mapping $I_{truth} \circ I_{external}(\sigma)$ in the formal semantics of [20] and [14]; (5) the domains for the built-ins' arguments.

RIF-DTB explains that *"typically, built-in functions and predicates are defined over the value spaces of appropriate datatypes, i.e. the domains of the arguments. When an argument falls outside of its domain, it is understood as an error."*

In RIF-DTB the predicate evaluating greater than is defined as follows:

name pred:numeric-greater-than
schema $(?arg_1 ?arg_2; pred : numeric-greater-than(?arg_1 ?arg_2))$
domains xs:integer, xs:double, xs:float, or xs:decimal for both arguments
mapping When both a_1 and a_2 belong to their domains
$I_{truth} \circ I_{external}(?arg_1 ?arg_2; pred : numeric-greater-than(?arg_1 ?arg_2))(a_1 a_2) =$ t if and only if $op : numeric-greater-than(a_1, a_2)$ returns true, as defined in [17], f otherwise. Also RIF-DTB specifies that *"if an argument value is outside of its domain, the truth value of the function is left unspecified."*

The numeric-greater-than predicate could be implemented in JSON Rules as depicted in Example 1.

Since the hyphen based notation used by RIF in the predicates and function names can not be used in JavaScript the camel hump notation should be used instead. As such the name `numeric-greater-than` gets translated into `numericGreaterThan`.

The mapping of the signature is obvious, with respect to the list of arguments.

Based on Table 1 `xs:integer`, `xs:double`, `xs:float`, or `xs:decimal` RIF-DTB datatypes are translated into custom types but based on the JavaScript `number` datatype.

With respect to RIF note that if the type of any of the arguments is outside of the specified domain then no result should be given. This situation can be dealt at least in three ways: (1) the value returned could be the JavaScript special value `undefined`; (2) JavaScript isNaN() function could be used and in case of true the `Number.NaN` should be returned; (3) an exception could be raised.

However since the RIF specification states that this predicate should behave as defined in [17], in case of a NaN value, false is returned. As such this approach can not be used here. In other cases any of the three approaches could be used.

Example 1 (Defining a built-in predicate).

```
org=new function(){};
org.w3c=new function(){};
org.w3c.rif=new function(){};
org.w3c.rif.pred=new function(){};
org.w3c.rif.pred.numericGreaterThan=function(arg1,arg2) {
    try{
        if (isNaN(arg1) || isNaN(arg2))
            return false;
        if (arg1>arg2)
            return true;
            else
                return false;
    }catch(e){
        //log error
        //console.log(e); i.e. if Firebug is used
    }
    return false;
}
```

This approach could be used in general to map all RIF built-in predicates and functions.

In addition the approach could be used to define libraries of actions oriented towards usage of well known services such Twitter[5], Facebook[6], Youtube[7] and so forth through their APIs.

2.5 JSON Rules built-ins for mashups

In mashups case there could be identified a set of operations that happen regularly, such as loading of services, memory cleaning etc. These type of functionality is provided under the `org.jsonrules.builtin.mashups.system` namespace. Functionality such as `load` or `memoryCleanUp` is provided as built-in functions the named namespace.

The existence of services is a prerequisite for the mashups, and as such they have to be available for the mashup, when this is instantiated.

Recall that we have in mind mashups modeled with JSON Rules that run in the browser [4]. All the involved services interact with each other in a common context which is the *choreographer* page. The choreographer is accessible to the user through a web browser.

[5] `http://twitter.com/`
[6] `http://www.facebook.com/`
[7] `http://www.youtube.com/`

However firstly they need to be made available to the choreographer. The browser loads the content of the choreographer and a DOM Basic Event `load` it is raised. These functions should be used in general in relation with a `load` event, because its main purpose is to make available for the choreographer the necessary services.

The term services here has a broader understanding and comprises at least the following: web page, RPC service, AJAX object.

The signature of `org.jsonrules.builtins.mashups.load` function is: `org.jsonrules.builtins.mashups.load($what,$where)`. In RIF-DTB terminology the *signature of a function* is the `schema of a function`.

The `$what` argument refers to what needs to be loaded. this could be an URI (Uniform Resource Identifier) or it could be a reference to an AJAX object.

$where argument refers to the place where the service or the response of an AJAX request will be stored. This could be for example an `iframe`, a `div`, both of them identified by an id, a reference to a JavaScript object.

3 Conclusions

This paper touched the problem of built-ins for JSON Rules in relation with RIF. This is a step towards translation JSON Rules to and from RIF. RIF built-in datatypes, predicates and functions have been taken into consideration, as well as other type of built-ins that in principal are useful in the mashups context. To maintain similar behavior functionality grouped, namespaces have been suggested for RIF-DTB and mashups environments. Beside all these technical aspects necessary for built-ins definition in the context of JSON Rules has been taken into account.

References

1. Proctor, M., Neale, M., Frandsen, M., Jr., S.G., Tirelli, E., Meyer, F., Verlaenen, K.: Drools 4.0.7. `http://downloads.jboss.com/drools/docs/4.0.7.19894.GA/html_single/index.html` (May 2008)
2. Kifer, M., Lausen, G., Wu, J.: Logical foundations of object-oriented and frame-based languages. Journal of the ACM **42** (1995) 741–843
3. Giurca, A., Pascalau, E.: JSON Rules. In: Proceedings of the Proceedings of 4th Knowledge Engineering and Software Engineering, KESE 2008. Volume 425., CEUR Workshop Proceedings (2008) 7–18
4. Pascalau, E., Giurca, A.: A Rule-Based Approach of Creating and Executing Mashups. In: Proceedings of the 9th IFIP Conference on e-Business, e-Services, and e-Society (I3E 2009). LNCS, Springer (2009) 82–95 forthcoming.
5. Adida, B., Birbeck, M.: RDFa Primer Bridging the Human and Data Webs. W3C Working Draft (October 2008) `http://www.w3.org/TR/xhtml-rdfa-primer/`.
6. Hors, A.L., Hegaret, P.L., Wood, L., Nicol, G., Robie, J., Champion, M., Byrne, S.: Document Object Model (DOM) Level 3 Core Specification. W3C Recommendation (April 2004) `http://www.w3.org/TR/DOM-Level-3-Core/`.

7. OMG: Production Rule Representation (PRR), Beta 1. Technical report, OMG (November 2007)
8. de Sainte Marie, C., Paschke, A., Hallmark, G.: RIF Production Rule Dialect. W3C Working Draft (July 2009) http://www.w3.org/TR/rif-prd/.
9. Hohrmann, B., Hegaret, P.L., Pixley, T.: Document Object Model (DOM) Level 3 Events. Technical report, W3C (December 2007)
10. ECMA: ECMAScript Language Specification. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf (December 1999)
11. Paschke, A., Hirtle, D., Ginsberg, A., Patranjan, P.L., McCabe, F.: RIF Use Cases and Requirements. W3C Working Draft (December 2008) http://www.w3.org/TR/rif-ucr/.
12. Polleres, A., Boley, H., Kifer, M.: RIF Datatypes and Built-Ins 1.0. W3C Working Draft (July 2009) http://www.w3.org/TR/rif-dtb/.
13. Boley, H., Hallmark, G., Kifer, M., Paschke, A., Polleres, A., Reynolds, D.: RIF Core Dialect. W3C Working Draft (July 2009) http://www.w3.org/TR/rif-core/.
14. Boley, H., Kifer, M.: RIF Basic Logic Dialect. W3C Working Draft (July 2009) http://www.w3.org/TR/rif-bld/.
15. Peterson, D., Gao, S.S., Malhotra, A., Sperberg-McQueen, C.M., Thompson, H.S., Biron, P.V.: W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes. W3C Candidate Recommendation (April 2009) http://www.w3.org/TR/2009/CR-xmlschema11-2-20090430/.
16. Bao, J., Hawke, S., Motik, B., Patel-Schneider, P.F., Polleres, A.: rdf:PlainLiteral: A Datatype for RDF Plain Literals. W3C Candidate Recommendation (June 2009) http://www.w3.org/TR/rdf-plain-literal/.
17. Malhotra, A., Melton, J., Walsh, N.: XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Recommendation (January 2007) http://www.w3.org/TR/xpath-functions/.
18. Russell, M.A.: Dojo The Definitive Guide. O'REILLY (2008)
19. ECMA: ECMAScript for XML (E4X) Specification. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-357.pdf (December 2005)
20. Boley, H., Kifer, M.: RIF Framework for Logic Dialects. W3C Working Draft (July 2009) http://www.w3.org/TR/rif-fld/.