CSC8499 Individual Project: Implementing a Contract Compliance Checker for Monitoring Contracts

Ioannis Sfyrakis

MSc in Advanced Computer Science,
School of Computing Science, University of Newcastle, U. K.
i.sfyrakis@ncl.ac.uk

Abstract. Businesses use contracts to regulate interactions between them. Contract monitoring is an active research topic where we observe the interactions between the business parties. We use a Contract Compliance Checker (CCC) for monitoring contracts that was developed in Newcastle University. The CCC uses ECA rules written in a semi-formal language called EROP in order to specify a business contract. A current limitation of the CCC is that it uses an old version of Drools and that it uses hardwired Business Events. The main aim of this dissertation was to migrate to the latest version of Drools and enhance the architecture of the CCC. The main enhancement added was to transform the CCC to a web service that accepts Business Events from the outside world. The web service follows the REST architectural style and was implemented using Java related technologies. We consider the old CCC version as a reference implementation. Testing of the CCC web service was performed using the same contract as the old version. Results show that both the old version and the enhanced version of the CCC have the same behavior when a specific business contract is used.

Declaration: I declare that this dissertation represents my own work except where otherwise explicitly stated.

1 Introduction

Contracts are used in business-to-business (B2B) interactions in order to regulate them. A contract is a legally binding agreement that has two or more participants and stipulates a number of rights, obligations and prohibitions. A right is what the contracting party is allowed to do, an obligation is what the contracting is expected to do or they risk being penalized and a prohibition is what the contracting party is not expected to do unless they want to be penalized [1]. As an example, a contract between two parties a Buyer and a Seller is presented below. In the contract, C stands for "clause", and rights, obligations and prohibitions are shown for each clause:

- C1: The Buyer has the right to place a **Buy Request** with the Seller to buy an item, as long as it is from Monday to Saturday and from 9am to 5pm (Right).

- C2: The Seller is obliged to respond with either **Buy Confirmation** or **Buy Rejection** within 3 days of receiving the Buy Request (Obligation).
- C3: The Buyer can use its discretion to either **Pay** for or **Cancel** the **Buy Request** within 7 days of receiving a confirmation (Obligation). Canceling the **Buy Request** is prohibited to the Buyer in any other condition (Prohibition).

Contractual interactions are implemented as cross-organizational business processes executed between the two parties in a loosely coupled manner. This implies that each operation execution involves the two parties, which produce two independent and possibly conflicting outcomes. Normally the parties are interested in monitoring/enforcing the contractual interaction at runtime. Contract monitoring is all about observing the interaction between parties. Similarly, the focus of contract enforcement is on preventing as much as possible, violations of the contract.

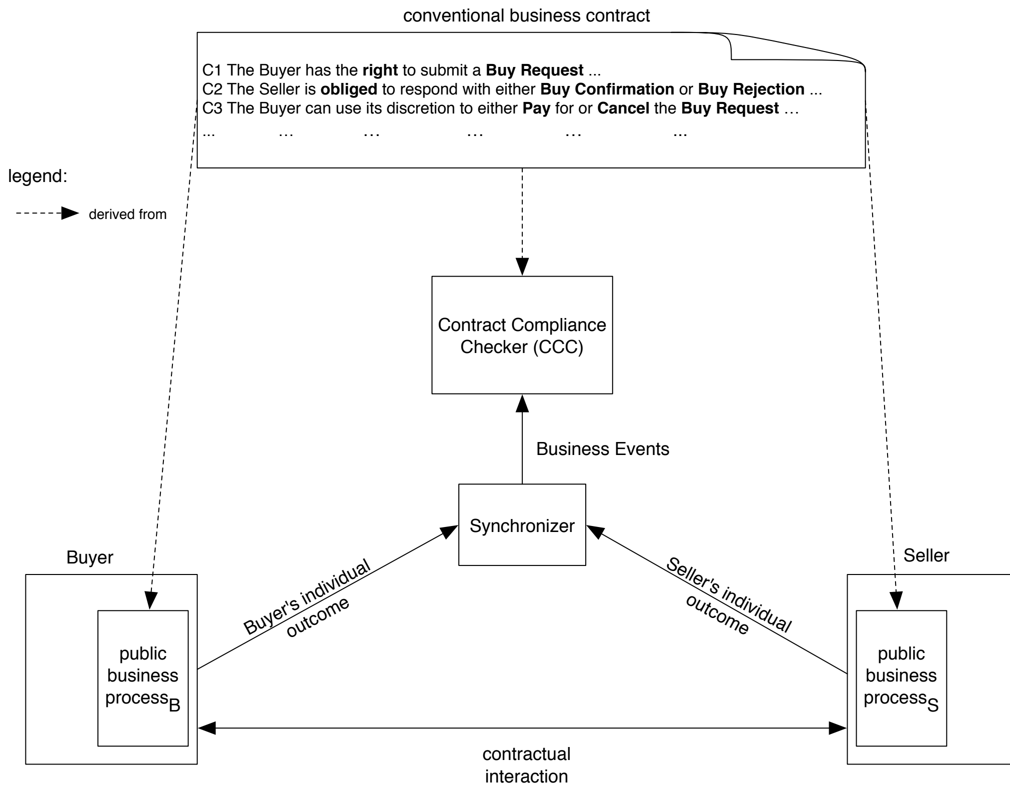

Fig. 1. contract monitor

Fig. 1 shows a sample scenario with the three clauses from the example contract. This architecture for the contract monitor consists of a Buyer side and a Seller side. Each side has a public business process that is derived from the contract. A formal version of the contract is provided to the Contract Monitor. While both parties interact with each other to fulfill the contract, there is a synchronizer component that takes as input each party's individual outcome. This component exists because of potential conflicting outcomes. For example, the Buyer can issue a Buy Request and the Seller can accept such a business operation that is compliant with current contract. Buyer side sends a message to the synchronizer component if the operation is successful or a failure. The same happens from the Seller side, which sends a message to the synchronizer if the outcome of the operation is a success or failure. The main goal of the synchronizer component is to compose a single outcome of the pair of individually produced outcomes and to notify the result to the Contract Monitor and to the participants. Then the Contract Monitor receives the business event and determines if this event is contract compliant or not. To determine if a given business event is contract compliant or not the Contract Monitor needs to be provided with a formal version of this business contract that it can interpret and manipulate. Let us call it the e-contract model or specification.

E-contracts are specified in domain specific languages, designed to capture key contractual concepts such as rights, obligations, prohibitions, normative statements, role players and deadlines. We call these languages contract languages.

A number of contract languages have been discussed in literature. Some of them are based on Deontic Logic[2, 3] and others are based on Law-Governed Interaction (LGI)[4]. The main problem with these contract languages is that they are not easy to map into an existing technology such as RosettaNet[5]. However, no consensus has been reached so far about the best alternative. The choice of one in favor of another is a matter of debate and perhaps of personal preference. This suggests that contract languages are still a research topic. Additionally, the design and implementation of the contract monitor itself is also a research problem. To help cover the research gap, in this dissertation we propose the use of EROP, an ECA-based contract language for expressing e-contracts that are manipulated by a Contract Compliance Checker (CCC). The EROP and CCC are currently the subjects of research projects of the school of computer science of Newcastle University. Preliminary implementations have been produced. For instance, a working version of a CCC implemented in Drools 4 [6]version exists. However, it suffers from several limitations including not being able to accept business events from outside the context of the implementation but relies on hardcoded events being fed to the CCC engine. The aim and contribution of this dissertation is to redesign the CCC to enhance some of its limitations and to migrate it to the latest Drools 5 version.

1.1 Aim And Objectives

Precisely, this project aims at enhancing the CCC that has been already designed and implemented. The Drools engine that is currently used will be updated to the latest technology. Also, the CCC will be exposed as a web service that can be utilized by other services and technologies rather than being a self contained desktop application. For example, the new implementation will allow to input a new business event to the CCC or add a new set of rules using a request send by a client. All these new features will enhance the current functionality of the CCC and make it more suitable for research experiments. The main objectives of our research are:

- to evaluate critically related work on monitoring contracts and undertake an analysis of the old CCC;
- to enhance the design of the CCC in terms of architecture and functionality;
- to design and develop a high quality prototype that exposes the CCC as a RESTful web service;
- to evaluate critically the new implementation of CCC against the proposed requirements and perform tests using an example contract;
- to outline possible ways of improving the CCC in the future.

1.2 Structure of Dissertation

This dissertation features 10 major sections. Section 2 will aim to discuss related work on monitoring contracts. A review of the implementation of the old CCC is discussed in Section 3. Section 4 discusses how the enhanced CCC is designed. Section 5 outlines the specific technologies used during the implementation of the CCC. Section 6 describes how the new CCC was implemented. Featured, in Section 7 are the testing approaches that were followed in order to test the CCC. Meanwhile, Section 8 evaluates the enhanced CCC built for this paper and performs a critical analysis of each objective we aim to achieve. Section 9 presents the conclusions. Finally, Section 10 outlines possible future work on CCC.

2 Related Work

In this section we present research work that is relevant to our work conducted by various academic, and industry research groups in the area of monitoring electronic contracts. The subsections that follow intend to critically evaluate (1) contract languages that can be used to represent electronic versions of contracts; (2) different ways of monitoring electronic contracts; and (3) contract monitor systems that have already been developed by research or industry groups.

2.1 Contract Languages

In order to monitor the enactment of a contract, that contract needs to be converted to an electronic version. This can be achieved by specifying a contract using a contract representation language that can express the contract in a machine amenable notation. Each contractual clause is encoded according to the notation of the contract language that has been chosen. According to Hvitved [7] there are three main categories of contract formalisms: (deontic) logic based formalisms [8–10], ECA-based formalisms [11] and action/trace based formalisms [12, 13]. Also, other contract formalisms worth mentioning use defeasible reasoning [10, 8] or finite state machines [14].

Table 1 shows the main advantages and disadvantages of the contract paradigms mentioned. Our choice is to use a language that implementors find easy to write and understand. On this basis we have chosen an ECA notation. We expect that implementors will intuitively encode contractual clauses in ECA rules that are compatible with current business technologies such as RosettaNet and can be easily integrated.

In [15] the authors suggest a list of nine desirable features that in their opinion contract languages should provide, including formality, expressiveness, usability, declarativeness and consistency checks. Though we agree with the list, we feel for implementors an equally important feature is implementability defined as a language we can effectively and efficiently implement. This is in fact a salient feature of our ECA-based contract language.

2.2 Contract Monitoring

Contract monitoring and/or enforcement is an active research topic. An early work in this area is Law-Governed Interaction (LGI) [4, 16]. LGI is an architecture for contract monitoring and enforcement that consists of three parts: (1) Law-Governed Interaction Model; (2) Moses middleware which runs the LGI architecture and (3) two law languages based in Prolog and Java respectively that specify the interaction between two or more autonomous agents. Moses middleware includes Controllers that are located between the interacting parties. The main objective of Controllers is to receive events and take actions according to a knowledge base that contains rules. These rules can be written in any one of the two supported languages and are stored in Law Servers. This approach is similar to our Contract Monitor's way of inferring action based on a knowledge base of rules. Moses components need to be installed in each interacting party. In contrast, our Contract Monitor acts as a trusted third party, so we do not have to install it in all participants. A limitation of LGI with respect to our work is that they do not account for timing or message validity constraints.

Table 1. Main Advantages and Disadvantages of Contract Formalisms

Contract Languages	**Advantages**	**Disadvantages**
(Deontic) Logic based languages	• rigorous formal approach	• not easily understandable by technical and business people • not widely used in business world and industry • static
ECA-based languages	• widely used in business world and industry • intuitive for technical and business people to write and understand	• not strictly rigorous approach
Defeasible reasoning languages	• rigorous formal approach	• not easily understandable by technical and business people

Heimdhal [17] is a middleware platform that can monitor and enforce history-based policies. All interactions between the participants are constantly monitored and authorized by Heimdhal. The events that come out of the interactions are asserted against a rule base of policies. If the policies are found to match a rule then they either impose or fulfill obligations or impose compensations. Heimdahl contains a policy monitor similar to our Contract Monitor that monitors and enforces Service Level Agreements (SLA). Thus, the main focus of Heimdhal is on enforcing resource usage policies (e.g "No job added for users above their monthly CPU quota of 10 hours"). The policies are defined using xSPL language that follows the ECA paradigm. It is declarative in nature. However, the language is difficult for non-technical users to write the policies. Also its focus is on expressing non-functional requirements rather than functional ones. Additionally, Heimdhal does not take into account rights or prohibitions in its policies but only obligations. Finally, there is not any explicit support for exception handling.

In [18] the authors present a mediating entity defined as Synchronization Point (SP) that monitors the events generated by the participants of a collaborating business process. Each SP contains a knowledge base of contract clauses that are written as ECA rules using Protégé 2000. Protégé Axiom Language (PAL) is a query language that can be used to search a knowledge base according to a number of criteria. In order to generate ECA rules for SP the contract written in natural language, first has to be converted to SP relationships and Satisfiability (Sat) functions. Relationships represent an obligation, permission or prohibition and are related to a set of Sat functions. The goal of Sat functions is to take as input, objects or events and examine if the SP relationships hold between the input objects or events. The SP handles exceptions that can happen in the future by warning the related parties. The problem with this approach is that it does not take into account unexpected failures such as technical failures. Even though, SP uses ECA rules similar to what our Contract Monitor use, we cannot assess if the rule is an obligation, right or prohibition.

Ludwig et al. [19] propose a Simple Obligation and Right Model (SORM). SORM can be used to specify electronic contracts for runtime monitoring. The main idea in this paper is that rights and obligation have a dual nature, i.e., a right for one party is and obligation for the other party. There are three types of rights and obligations: state obligations/rights, maintain a particular state, action obligations/rights, denote the promise to execute a certain action, and option obligations/rights to tolerate certain actions executed by another party. The authors discuss that rights and obligations of a contract can be modified during the enactment of the contract. Three actions are introduced that support the modification of rights/obligations: add, remove or change the current right/obligation. Some obligations remain constant and cannot be changed and are called background obligations. These are grouped together. Additionally, obligations are grouped together according to the state of the contract. This makes it easy to identify the obligations that should be modified when there is a state change in the contract. The model proposed in this paper is similar to the model in

our Contract Monitor regarding two aspects. Firstly, the rights and obligations are enforced dynamically during the enactment of the contract. Secondly, we use add/remove operations in order to modify the state of rights and obligations. One concept that is not present in SORM is prohibition in relation with rights and obligations of a contractual party.

Research conducted by Linington [20] discusses the usage of Model Driven Development (MDD) in order to generate software systems that can monitor the enactment of electronic contracts. Following the model driven approach system, designers produce an abstract model of the business in a domain specific language or a metamodel. Next they define a transformation model derived from a running solution that uses resources from the infrastructure. Both metamodels created can reuse the transformations, if the metamodels are reliable and well written. In addition, the tools built to execute the transformations can be long-lived. Contract monitoring in this regard contains two metamodels, (1) the notification metamodel that produces the calls to the infrastructure by connecting contractual parties with the monitor, and (2) the monitoring metamodel, that provides a mapping to a form that influences the decision process of the monitor during the business interaction. This paper discusses sub-contracting, multiple contract instances monitoring, nested executions, among other topics. However, it does not discuss the topic of deadlines and business or technical failures during the execution of electronic contracts.

3 Analysis of the old Contract Compliance Checker

This section intends to describe the old architecture of the Contract Compliance Checker. Additionally, it discusses the old implementation and provides a rationale for migrating to new technology.

3.1 Old Architecture of Contract Compliance Checker

As shown in Fig. 1, the CCC is deployed as a neutral observer of the interaction of the contracting parties, for instance a buyer and a seller. It observes business events (bevent) that notify of the execution of operations. The objective of CCC is to observe and log all the interactions between relevant parties to determine if their actions are contract compliant. Each business partner includes a private business process and a public process. Both types of business processes collaborate in order to execute the contract. Also, there are two logical channels: One is the monitoring channel that delivers business events (bevent) to the CCC. The other is the conversation channel that is used for business conversations between the contractual parties.
Fig. 2 presents the architecture of CCC. The ROP sets depicted in this figure store the current set of rights, obligations and prohibitions of the contractual parties.

The bevent logger is a storage facility for persisting records regarding all the events processed by the CCC. The bevent queue is a queue that stores bevents until they are retrieved for processing by the relevance engine.

The contract rules is the rule base repository and includes a number of ECA rules that describe the contract under monitoring. Rules react to events that correspond to bevents. Furthermore, rules include actions to add and delete rights, obligations, prohibitions. Thus, these (add/del) operations are executed in order to update the ROP sets.

The timer keeps track of deadlines related to the rights, obligations, and prohibitions that are stored in the ROP sets. The relevance engine is responsible to set or reset deadlines. If a deadline expires then a timeout event is sent to filter for mismatched operations (filter mism. $\mathrm{bo_i}$).

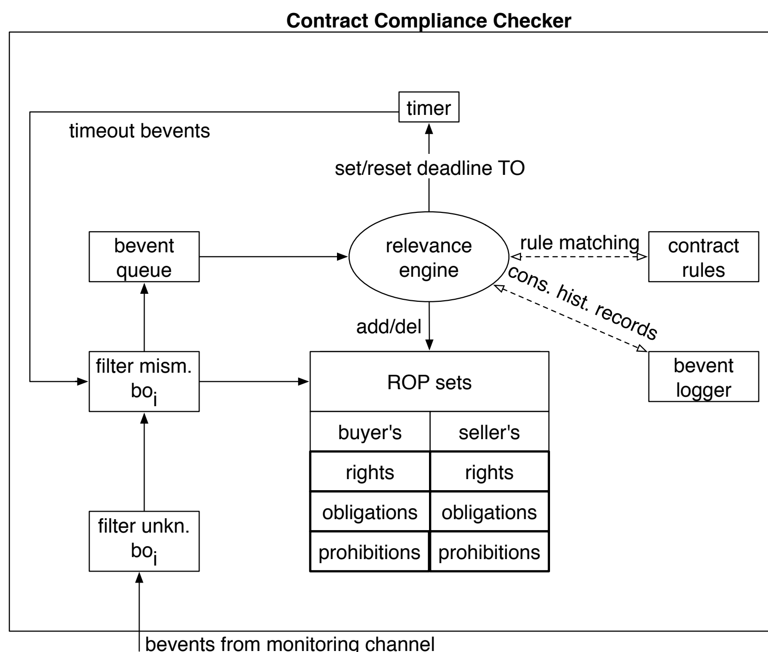

Fig. 2. Architecture of the CCC [1]

When bevents arrive at the CCC from the monitoring channel they need to pass through two levels of filters. The first filter is the filter for unknown operations (filter unkn. $\mathrm{bo_i}$) and its objective is to filter out unknown business operations.

The second filter is the filter for mismatched operations (filter mism. $\mathrm{bo_i}$). If a bevent is a mismatched business operation then the event gets filtered out and sent to the bevent logger. Additionally, timeout bevents are also examined by the second filter before they are added to the bevent queue. The main goal of the two filters is to exclude non-compliant business operations from reaching the relevance engine.

The relevance engine removes a bevent from the head of the bevent queue, tries to match the bevent to a rule from the rule repository and then trigger the relevant rule.

3.2 Implementation of the Contract Compliance Checker

Business interactions between partners generally take place through a set of well defined business operations. For instance, in our contract example buy request, buy confirmation are considered as business operations. At implementation level, we assume that each business operation is supported by a business conversation which can be implemented in RosettaNet Partner Interface Processes[5] or ebXML[21] industry standards.

A business conversation is a message interaction protocol with message timing and validity constraints. The execution of a business operation generates an initiation and an execution outcome event. The first outcome event returns an InitSucc event if the initiation succeeds and InitFail if the initiation fails. The second outcome event, following ebXML specification, returns a successful conclusion (Success) or a business failure (BizFail) or a technical failure (TechFail). All the business conversations between the contractual parties are implemented using a Message oriented Middleware (MoM) as in Fig. 4 below.

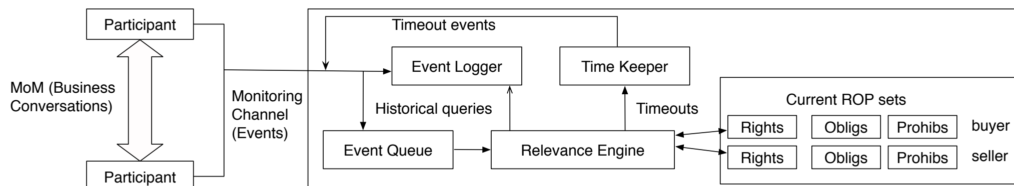

Fig. 3. The Contract Compliance Checker [22]

Main components as seen in Fig. 3 include Event Logger, Time Keeper, Event Queue, and the Relevance Engine. Each component is defined in a class. Event Queue is implemented as First In, First Out (FIFO) queue that adds an incoming Event to the end of the queue and removing an Event from the head of the queue. These Events are added by the participants through the monitoring channel as seen in Fig. 4. Also, Events can be added by the Time Keeper as timeout events. The only component that can remove events from the Event Queue is the Relevance Engine.

The main objective of the Time Keeper is to manage deadlines expiry from the current ROP sets and offers operations to add or remove a deadline. A deadline is represented internally as Java timers. When a deadline expires, the corresponding timer notifies the Time Keeper passing relevant parameters and

data. Then the Time Keeper makes a new instance of Event of the relevant type, appending Timeout to the name.

The job of the Event Logger is to maintain the historical database. Logging events in the database, submitting boolean and numerical queries are the operations supported by the Event Logger. The Relevance Engine uses Drools engine [6] to decide which rule to trigger and offers four operations. The first operation is to add an Event for processing, the second is to initialize a new contract for a new business interaction, the third is to process the Event Queue and the fourth is to verify that the queue is empty.

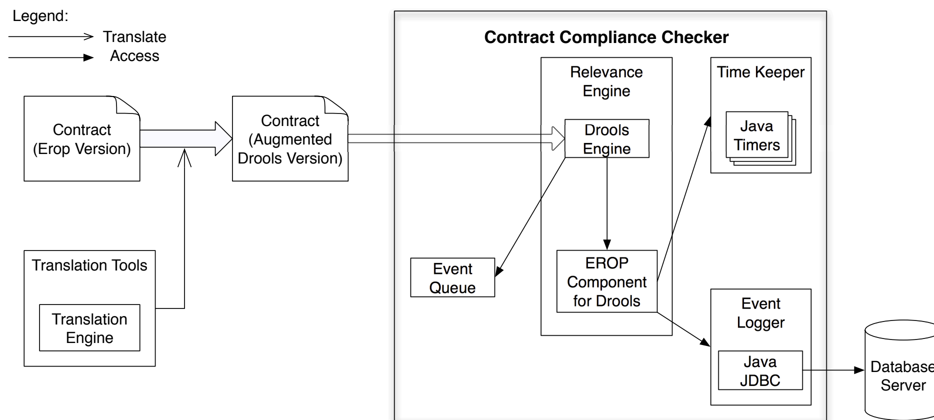

Fig. 4. Implementation Details for the Contract Compliance Checker[22]

3.3 The EROP Ontology

The EROP ontology models the execution of business operations between partners, and checks if the actions are contract compliant. Each of the EROP ontology class corresponds to a Java class. Consequently, the EROP ontology includes the following classes:

- Role Player: an entity used by one of the interacting parties that plays a role defined in the contract.
- Business Operation: a business activity that is defined in the contract.
- Right: A Business Operation that a Role Player has the right to execute.
- Obligation (Simple): A Business Operation that has to be executed by the Role Player.
- Prohibition: A Business Operation that should not be executed by the Role Player.

- Composite Obligation: A collection of Obligations which include a single deadline. In order for the Composite Obligation to be satisfied an Obligation has to be executed by the Role Player.
- ROP Entity: A Right, Obligation or Prohibition.
- Deadline: A time constraint to exercise a right, prohibition, simple and composite obligation.
- ROP Set: A set of rights, obligations and prohibitions that belong to the Role Player. A Role Player is associated with only one ROP Set.
- Event: It is a message that includes a record about the occurrence of a business activity.

Finally, Fig. 5 shows a UML diagram that outlines the class hierarchy for ROPEntity. ROPEntity is the superclass of Right, Obligation and Prohibition classes. Additionally, CompositeObligation class is the subclass of Obligation.

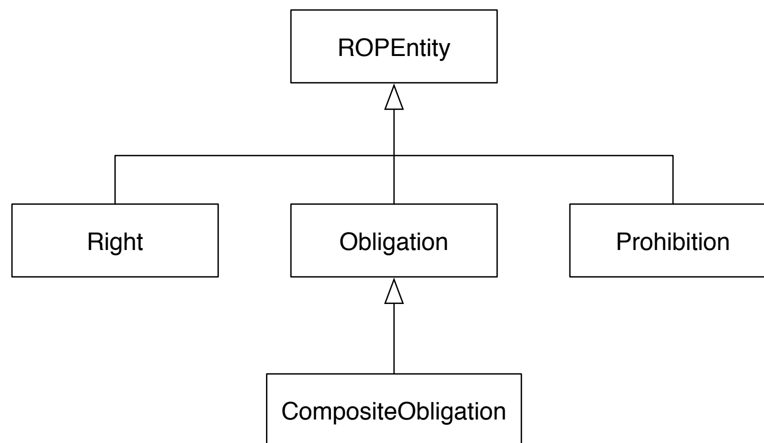

Fig. 5. Descendants of the class ROPEntity[22]

3.4 EROP language

EROP language is a domain specific language designed for expressing electronic contracts. It provides the designer with specific constructs for capturing contract concepts like role players, business operations, obligations, prohibitions, rights and operators for manipulating those concepts. For instance, one can add an obligation for payment to the role player buyer. EROP can be mapped into JBoss Drools and executed by its rule engine. Each concept of EROP like role players is mapped into a java object that can be executed by the rule engine.

Rules in EROP consists of the following: an event part that matches a specific type; condition that specifies a set of Boolean expressions; and the action that specifies a list of statements, usually data modifications.

A business operation is contract compliant when it satisfies several constraints: (1) the event attribute where we assert the attribute of an event to be correct, (2) historical constraint where we assert if a given event or number of events exist or not, (3) ROP constraints where we assert if the business operation matches a role player's ROPSet and (4) rule action constraints that can either modify ROP sets of role players or terminate the current operation.

EROP language uses the entities described in the previous subsection and realizes the model mentioned above. A full example written in EROP language is presented in appendix D. To help with our discussion the first part of the contract introduced in Introduction section will be used as an example.

The syntax of each rule follows the following structure:

rule *"ruleName"*
  when
    triggerBlock
  then
    actionBlock
end

The trigger block of a rule decides when the rule is triggerable. When the rule is triggered the action block is executed. It contains a set of boolean expressions in conjunction that must contain an event match expression. This expression compares the fields of an event object with a tuple of values that include the **botype** (type of event), **outcome** (outcome of the event), **originator** (name of the role player that initiated the business operation), **responder** (name of the role player that the originator is trying to interact with) or **timestamp** (time the event was received).

The action block contains a number of actions: += (add business operations or composite obligations from ROP sets), -= (remove business operations or composite obligations from ROP sets), pass (no effect) and terminate (concludes the execution of the contract). For instance, the following line from the example contract adds a new obligation to the seller to respond to the buyer within 72 hours.

```
seller.obligs += RespondToBuyRequest("72h");
```

Two other elements can appear in the trigger block, conditional statements and status guards. Conditional statements have the following structure:

```
if conditions then
        actionBlock
[else
        actionBlock]
endif
```

Conditions used in the if statement are the same as the conditions in a trigger block. Status guards (Success, InitFail, BizFail, TecFail, Otherwise) are used to group actions for conditional execution according to the outcome of a business operation.

The first part of a contract expressed in the EROP language is used for declaring the role players, the business operations and composite obligations that are used in the rule section. Role players include a list of parties involved in the contract and are declared by using the keyword **roleplayer**. For instance, in our contract example there are two role players, the buyer and seller. Business operations that are used in the contract can be declared using the **businessoperation** keyword. The keyword **compoblig** is used to declare composite obligations. The code snippet below shows the declaration and rule section derived from C1 of our contract.

```
roleplayer buyer, seller
businessoperation buyRequest,buyConfirmation
businessoperation payment, buyRejection
businessoperation cancelation

rule "R1"
  when
    e matches(botype == BuyRequest)
  then
    Success:
      if e.originator == buyer
        && BuyRequest in buyer.rights
        && e.weekday in [Monday ... Saturday]
        && e.time in [9 ... 18]
      then
        seller.obligs += RespondToBuyRequest("72h");
      endif
    Otherwise:
      pass;

end
```

3.5 Implementation of the Relevance Engine

Fig. 6 shows the structure of the Relevance Engine. It consists of Drools Engine and EROP components for Drools. Drools uses a set of rules to declaratively alter the state of the system without having to rely on static, hardcoded knowledge. The heart of Drools is its Inference Engine that can scale to a large number of rules and facts. The main purpose of the Inference Engine is to match facts and data against rules which in turn will result in actions. The matching of new or existing facts against rules is performed by the Pattern Matcher component of the Inference Engine. There are a number of different algorithms for performing

pattern matching such as Linear, Rete, Treat and Leaps algorithms. Currently, Drools implements the Rete algorithm.

According to Fig. 6 Production Memory stores the rules and Working Memory stores the facts that the Inference Engine matches against. Facts are asserted into the Working Memory where they can be altered or removed. The Agenda component manages the order of execution and if there any conflicting rules it uses a Conflict Resolution strategy that executes a rule according to its salience. Rules with higher salience are given higher priority for the rule engine to execute them.

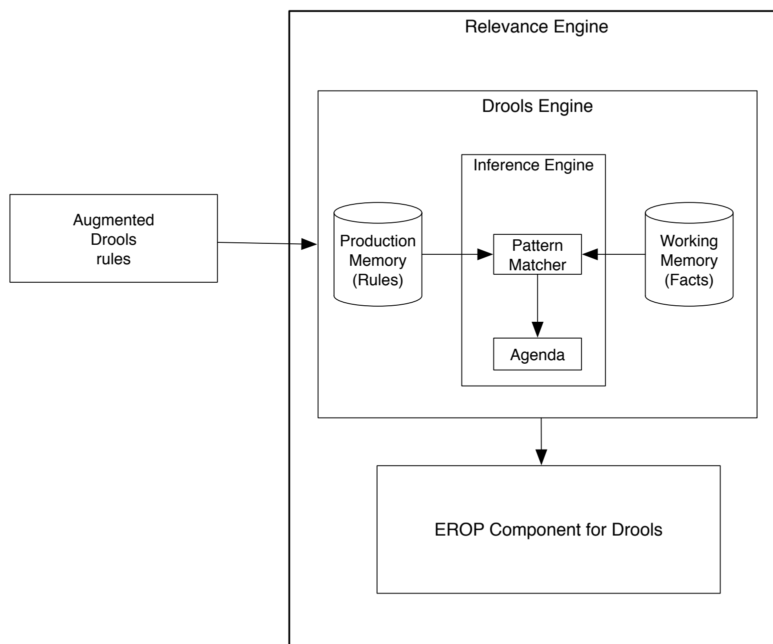

Fig. 6. High-level view of Rule Engine

There are two ways that rules can be executed: Forward Chaining and Backward Chaining. A system that implements both is called Hybrid Chaining System. Currently, Drools provides hybrid chaining, it supports both forward and backward executions. Forward chaining is data-driven and reactive, where facts are asserted in working memory, and the outcome is one or more rules to be true. Backward chaining starts with a conclusion that the rule engine tries to satisfies. If the conclusion cannot be satisfied then it searches for another conclusion that it can satisfy. Agenda then executes all the rules that are scheduled.

3.6 Augmented Drools

To execute a contract written in EROP language like the one presented before the designer needs to translate it (either manually or mechanically) into an electronic version. We assume the first alternative in this dissertation, so we translate the business contract into Augmented Drools (AD) with the addition of the Java implementation of the EROP ontology. Augmented Drools retains the same expressiveness of EROP and can be run in Drools engine but it is more implementation oriented. Additionally, EROP language can be mapped completely into AD. The snippet below shows the AD version of rule “R1” shown in p. 13 and 14. The full version of the contract rules in AD are presented in Appendix D.

The first part of a rule in EROP is the declaration section which also exists and serves the same purpose in AD. All objects and entities are declared in this AD section such as Role Players, Business Operations, Composite Obligations, Role Players’ ROP sets and instances of Relevance Engine and Event Logger. All global objects in the declaration section use the global Drools keyword and then the object to declare, its name and a semicolon.

```
global RelevanceEngine engine;
global EventLogger logger;
global RolePlayer buyer;
global RolePlayer seller;
global ROPSet ropBuyer;
global ROPSet ropSeller;
global TimingMonitor timingMonitor;

global BusinessOperation buyRequest;

rule "Buy Request Received"
when
    $e: Event(type=="Buy Request", originator=="buyer",
    responder=="seller", status=="success")
    eval(ropBuyer.matchesRights(buyRequest))
then
    ropBuyer.removeRight(buyRequest, seller);
    BusinessOperation[] bos = {buyConfirmation, buyRejection};
    ropSeller.addObligation("React To Buy Request", bos, buyer, 3);
end
```

In the first two lines we declare the instances of Relevance Engine, Event-Logger and TimingMonitor that we will use. A buyer and a seller are declared as instances of RolePlayer and their ROP sets are also declared. Finally, the business operations that will be used in subsequent rules are declared.

Rules in AD follow the same structure as EROP with a trigger and action block. The event matching is translated in AD using the syntax

```
$e: Event(attribute==value,[attribute==value]*)
```

$e is the event. The Drools keyword eval is used to evaluate boolean expression in the left hand side of each rule. Actions in AD use methods calls in order to add or remove a right, obligation, prohibition. In our snippet above we use the method addObligation to add a new composite obligation to the ROP set of the seller with the name “React To Buy Request” for a particular buyer.

3.7 The Historical Database

There are four tables in the Historical Database. One for the Role Player, one for possible status outcomes, one for Event types and one for the Event history. Out of these four tables only the fourth is actively used for events history. Consequently, the Event history table is created empty before the first run of the system and is populated with data during the contract's lifetime. All the data in the database table are available to be queried so that we gain more information and insight for activities performed in the past during a contract's execution.

Two categories of queries exist: boolean queries and numerical queries. The first category verifies if an event matching a certain number of constraints exists in the historical database. The second category count the number of occurrences of a certain event that exists in the Historical database.

3.8 Some Limitations of the old CCC implementation

1. If an event is contract compliant then CCC updates the ROP sets accordingly. Otherwise, if an event is not contract compliant CCC does not update the ROP sets.
2. The CCC can only accept additions or deletion of rights, prohibitions or obligations. Thus, it supports only two states during the monitoring of the contractual interactions. For example, an obligation can either be active or inactive. When a deadline has passed the ROP sets should update automatically rather than waiting for an event to update them.
3. The old CCC implementation is a self contained application where all events and rules are hardcoded. They are specified in a static manner inside the code. The file CCCExperiment.java represents a sample execution of a contract using the CCC. The contract and the events that are fed to CCC are hardcoded. Therefore, this prevents dynamism in the system in the sense that it cannot monitor different contracts unless the file CCCExperiment.java is edited by hand and the whole CCC recompiled. To solve this limitation events should be added from outside the CCC, using for example a RESTful interface.
4. Currently, all the events are stored in a mysql database, in the eventhistory table. We can only use a mysql database otherwise we would have to migrate the code that sends the SQL queries to the new database vendor. This can be improved by using Java Persistence API (JPA) for managing relational data and provide in this way an abstract data layer for the CCC. The advantage of this approach is that we decouple the way we store data and which database

type we use from the CCC implementation. Therefore, we can use any type of database that supports JPA.

5. The old CCC implementation uses Drools 4. This version of Drools provides a rule engine that does not perform very well in terms of raw speed and data loading time as opposed to the latest Drools Expert version. Additionally, Drools 4 does not provide backward chaining, which is 'goal-driven', meaning that we start with a conclusion which the engine tries to satisfy. The latter can be achieved using the latest version of Drools which is now considered a hybrid chaining system that satisfies both the forward and backward chaining paradigm. Therefore both a data-driven and a goal-driven approach can be used with Drools Expert 5.4.0.

From the above mentioned issues with the old CCC, we will address the last three limitations in this dissertation and we will suggest some ideas about addressing 1. and 2. in Future Work section.

As we have discussed above a number of ways exist that we can improve the architecture of the CCC, such as providing a RESTful interface that consumes business events. The elements of this improved architecture and related technologies are presented in the next sections.

4 Design

In this section we intend to describe the enhanced design and architecture of CCC and how its components integrates with each other.

4.1 Architecture of CCC web service.

An abstract view of the new architecture that we suggest for the CCC is presented in Fig. 7. With respect to the old architecture of the CCC (see Fig. 2) the new architecture incorporates an event composer, rule editor, an event queue, a reply queue, a RESTful interface, the CCC engine, a Database Decoupler and a Database Server.

The new architecture regards the CCC as a RESTful web service that can receive input requests from the outside world. The architecture currently is made of three main tiers: Client, Application, and Data Store Tier. The Application Tier consists of three layers: the Presentation Layer, CCC Logic Layer and Data Access Layer. All the tiers of the architecture are explained in the following sections.

4.2 Client Tier

The Client Tier consists of two elements, the Event Composer and the Rule Editor. The first element is actually a client, in our case the synchronizer, that can produce HTTP requests for the CCC service to consume. Therefore, a number of event messages can be sent to the CCC web service. The second element is

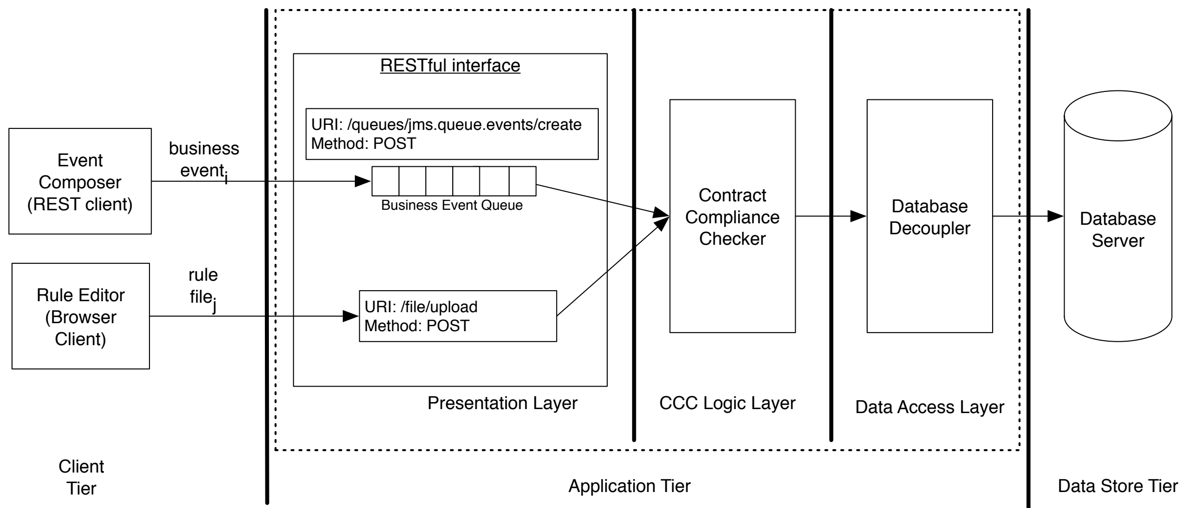

Fig. 7. High-level CCC web service architecture

basically an HTML form that sends HTTP POST requests to the CCC service in order to upload new rule files. This can also be achieved by any client that can compose a HTTP POST request and attach the file to upload according to the specification that the CCC uses.

4.3 Application Tier

The Application Tier is the main building block for the CCC web service. It consists of three layers: the Presentation, CCC Logic and Data Access Layer.

Presentation Layer The Presentation Layer is the web service endpoint for the RESTful interface. This interface allows the CCC to receive events from the outside world. This layer consists of the following Uniform Resource Identifier (URI) address

```
/queues/jms.queue.events/create
```

that is used by the client to create a new message for the Business Event Queue. A URI is a string of characters that identify a resource. The main job of the URI address is to accept XML messages that corresponds to event instances. The structure of this message can be seen in the snippet below.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<event>
<originator>buyer</originator>
```

```
<responder>seller</responder>
<type>BuyRequest</type>
<status>success</status>
</event>
```

Thus, the RESTful interface accepts an HTTP POST request from an Event Composer that includes the XML message as payload. The message is sent over the network as a BytesMessage and then serialized to a Data Transfer Object using Java Architecture for XML Binding (JAXB). Afterwards, the object that results from the serialization is passed to the CCC logic layer.

The second URI address

```
/file/upload
```

accepts a HTTP POST request which includes a rule file sent from an HTML form. The rule file is uploaded to the application server and is passed to the CCC logic layer in order to update the rules currently in memory with the new rules.

CCC Logic Layer The CCC Logic Layer contains the main functionality of the CCC service. This layer accepts as input the serialized Event Data Transfer Object which triggers the execution of the CCC engine to adjust the state of the EROP ontology and pass the outcome as a Data Access Object to the Data Access Layer. Furthermore, the CCC Logic Layer accepts a rule file as input that is placed in a certain folder on the application server and triggers the recompilation of all rule files that exist in that folder. The result of this is that all the rules in the knowledge base are updated with the new information.

Data Access Layer The Data Access Layer (DAL) provides simplified access to data stored in persistent storage of the Data Store Tier. The DAL hides this complexity of the underlying data store from the external world and makes transparent the manipulation of data.

4.4 Data Store Tier

The Data Store Tier is the last tier in the architecture and includes the persistent storage of data. Because we are using a DAL we are not depending on a particular database type but we can choose any that is supported by the DAL.

5 Specific Technologies Used in the Implementation

5.1 JBoss Application Server

JBoss Application Server (AS) [23] is an application server that supports the Java Platform, Enterprise Edition (Java EE). The latest version (version 7) of JBoss AS was used in this project. It is officially certified for Java EE 6 Web

Profile and includes smaller code size and a great reduction in startup time. Also, a brand new kernel is employed which includes two main projects: JBoss Modules and Modular Service Container (MSC). The main objective of JBoss Modules is to handle the class loading of resources in the container. This improves the modularity of the application server. The MSC provides a way to install, uninstall and manage services that are used by the container. Also, it enables resources injection into services and dependency management between services.

The application server file system is divided into two main parts: the standalone and domain server part. The standalone part of the file system is used when the application server is configured as a standalone service, whereas the domain server part manages and coordinates a number of instances of the application server. In our project we configure the application server as a standalone server.

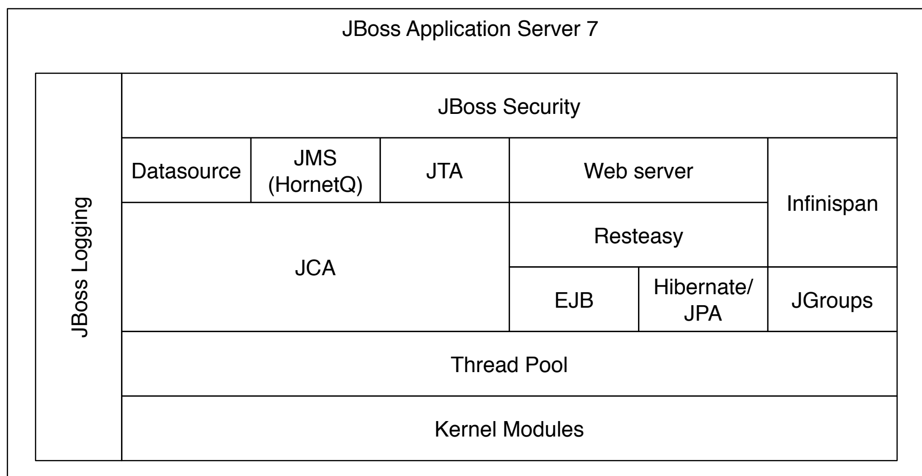

Fig. 8. Core modules of JBoss AS 7

Figure 8 shows the main modules that are included in JBoss AS 7. Each particular module that we use in our project will be discussed in the following subsections.

5.2 HornetQ

HornetQ is an example of a Message Oriented Middleware (MOM). It supports Java Message Service (JMS) 1.1 API and is used to provide an asynchronous messaging system that has both a JMS and a RESTful interface. We can send a RESTful message to HornetQ and then it can be consumed as a generic JMS message. The same can happen in reverse where we can have a JMS producer and a RESTful consumer for the messages.

5.3 RESTEasy

RESTEasy provides various frameworks for building RESTful web services and java applications. It is a fully certified and portable implementation of JAX-RS specification and includes a JAX-RS Client framework for building RESTful clients using annotations. Additionally, it is smoothly integrated with EJB.

5.4 Hibernate

Hibernate is an Object-Relational Mapping (ORM) solution for the Java language. ORM refers to a technique of mapping data between an object model representation to a relational data model representation. Hibernate provides a framework that achieves the mapping of an object-oriented domain model to a traditional relational database.

5.5 EJB

Enterprise JavaBeans (EJB) is a server side component based architecture that is used to build modular enterprise applications. The EJB specification is one of several Java APIs in the Java EE specification. It includes two main server side component types: session beans that clients can invoke them and message driven beans that act as event listeners. Additionally, EJB specification deals with persistence by providing a Java Persistence API (JPA) for integrating an EJB bean with a database. Entity beans are used to provide integration between EJB and JPA. All persistence actions are managed by the entity manager.

5.6 JBoss Drools

Drools is a business rule management system (BRMS) with a forward and backward chaining inference based rules engine. It is known as a production rule system, using an enhanced implementation of the Rete algorithm. Drools supports the JSR-94[24] standard for its business rule engine and enterprise framework for the construction, maintenance, and enforcement of business policies in an organization, application, or service.

Drools is a rules engine implementation based on Charles Forgy's Rete algorithm[25] tailored for the Java language. Adapting Rete to an object-oriented interface allows for more natural expression of business rules with regard to business objects. Drools is written in Java, but able to run on Java and .NET.

The original version of CCC uses the rule engine of Drools 4 for executing the rules.

5.7 Drools 4

Drools 4 main components is the rule engine which uses the RETE and LEAPS algorithm, rule flow and a basic Business Rule Management System (BRMS). Rule flow allows designers to specify the order in which rule sets should be

evaluated by using a flow chart. This allows you to define which rule sets should be evaluated in sequence or in parallel, to specify conditions under which rule sets should be evaluated, etc. The main prerequisite for running Drools 4 rules is java 1.4. There is also an optional Eclipse plugin that is helpful during editing rules. Also, the main 4 libraries have to be added to the java class path. These 4 libraries are the following:

- drools-core.jar - includes the runtime component which contains both the RETE engine and the LEAPS engine. This is the only library we need when we are pre-compiling rules and deploying via Package or RuleBase objects.
- drools-compiler.jar - contains the compiler/builder components that build executable rule bases from rule sources. This library depends on drools-core.
- drools-jsr94.jar - is the JSR-94 compliant implementation, that is in essence a wrapper for drools-compiler component.
- drools-decisiontables.jar - this is the decision tables 'compiler' component, which uses the drools-compiler component. This library can use spreadsheets such as Excel files to define decision tables.

5.8 Drools 5

Drools starting from version 5.0 is split into 4 main sub projects:

- Drools Guvnor is a web application that is used as a repository for Drools Knowledge Bases. This web application contains rich web based GUIs, editors, and tools to help the business or technical user to manage a large number of rules. Knowledge Bases include rules, workflows, processes that can be stored in Guvnor.
- Drools Expert (rule engine), is a declarative rule based environment that gives the ability to focus on "what can be achieved", and not "how this can be achieved". The main advantage of this ability is that by using rules it is easy to express solutions to difficult problems and consequently have those solutions verified. Furthermore, rules have a high level of readability.
- jBPM is a Business Process Management (BPM) Suite. The responsibilities of jBPM include the modeling, monitoring and executing of business processes. The jBPM is considered a light-weight, extensible workflow engine that is written in Java. Additionally, it allows you to execute business processes using the latest Business Process Modeling Notation (BPMN 2.0) specification.
- Drools Fusion is used to perform complex event processing as an independent module and is well integrated with the other sub-projects of Drools. Drools Fusion has the ability to understand and handle events, select a set of interesting events that exist in a cloud or stream of events, find the relevant relationships among these events, execute actions based on the relationships detected and the temporal reasoning between events.

Table 2 summarizes the main features of Drools 4 and 5.

Table 2. Main Features of Drools 4 and Drools 5

Drools 4	Drools 5
forward chaining rule engine	forward chaining rule engine
rule flow	backward chaining rule engine
RETEOO algorithm	RETEOO algorithm
LEAPS algorithm	event processing
	temporal reasoning
	business process/workflow management
	improved speed
	Drools Knowledge Base repository

5.9 Maven

Maven is a tool that can be used to automatically manage all the required dependencies a software project needs for compilation, testing, reporting, and deployment. The main file that maven uses is the POM (Project Object Model) file. It is an xml file that contains all the information required by the maven builder.

6 Implementation

In this section, we first describe the migration steps for upgrading Drools to the latest version and specifically the code that we changed during this process. Then we discuss how the implementation was undertaken for the project and how we configured each component so that we can execute CCC in an application server as a web service.

6.1 Migration from Drools 4 to Drools 5.4.0

The original version of CCC uses Drools 4 which was released in 2007. The latest version of Drools 5 released in 2012 provides better speed and a less loading time for data and rules. Also, Drools 5 provides support for complex event processing and process workflow that can be leveraged to improve CCC. The above capabilities and features are not provided by Drools 4. Therefore, migrating to Drools 5 is the obvious choice if we want to improve CCC and integrate complex event processing and process workflow in the future. The oldest Java version required by Drools 5.4.0 is 1.5.

A number of steps were taken in order to migrate the current codebase to the latest Drools version (5.4.0). First phase includes the steps we took in order to use Maven for building the CCC code. Second phase outlines the changes in the code that are needed in order to use the latest Drools version.

Phase 1 - Use of Maven for building project

- Built initial version of pom.xml including Drools 4 and other libraries dependencies
- Place all .drl files in source folders in resources folder.
- Edit file CCCExperiment.java and change the file path of SimpleContract.drl so that it points to the resources folder.
- Edit file RelevanceEngine.java and change the file path of TestingContract.drl so that it points to the resources folder.

Phase 2 - Migrating codebase to Drools 5.4.0

- updated pom.xml with Drools 5.4.0 library dependencies for CCC.
- Edit file RelevanceEngine.java and change classes that are used in new version of Drools. Replace RuleBase to KnowledgeBase, StatefulSession to StatefulKnowledgeSession, PackageBuilder to KnowledgeBuilder, PackageBuilderErrors to KnowledgeBuilderErrors, Package to KnowledgePackage.

6.2 XML messages Flow

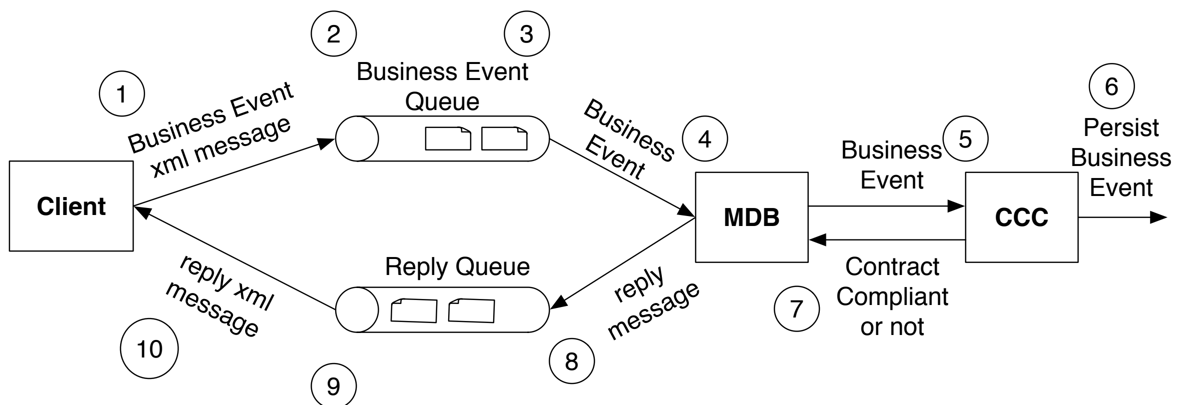

Fig. 9. XML Messages Flow

Fig. 9 shows the flow of messages when a client sends a Business Event to the CCC which is regarded as a web service and the client wants to know if the Business Event is contract compliant or not. The client in our context is the synchronizer that sends the Business Event and receives a result message. The result message contains the information if the Business Event previously sent is contract-compliant or not. The CCC is assumed to be instrumented with the rules that represent the contract. We describe the 10 steps involved in this process below:

1. Client sends a Business Event as an XML message to the Business Event queue using an HTTP POST request.
2. The message is received and stored in the Business Event queue.
3. HornetQ instance removes the Business Event from the queue and forwards it to the Message Driven Bean (MDB).
4. The EJB container instantiates the MDB that receives the Business Event, serializes it to a BusinessEvent object and sends it to the EventQueue of the CCC (see Fig. 3).
5. Eventually the Business Event message is processed by the rule engine.
6. BusinessEvent is sent to persist its state to the database server (see Fig. 7).
7. After CCC finishes processing the BusinessEvent it sends to the MDB its decision about the BusinessEvent found is contract compliant or not.
8. MDB receives the reply from the CCC and adds it to the reply queue (see Fig. 9).
9. Reply queue sends the reply to the client.
10. Client receives the reply XML message and displays its contents.

6.3 Application Tier Implementation

As we have presented in section 4 the CCC web service architecture consists of a Client, Application and Data Store Tier. In the following sub sections we intend to describe the three layers of the Application Tier, which is a central element of our architecture. It is packaged as an Enterprise Archive (EAR) in order to package the modules we need for this project. The modules packaged can be seen in Fig. 10. The commons module includes all the internal code of the CCC engine and the Entity class used for sending data to the database through the persistence layer and can be used from the other modules. The ejb module contains the Message Driven Bean (MDB) that consumes the messages from the RESTful queue that HornetQ provides. The web module contains the RESTful interface and html code needed for the user interface, as well as various configuration files. Finally, the application.xml file is the deployment description for this application modeled as an enterprise application and is located in the META-INF subdirectory of the application archive.

```
<!-- deployment descriptor : application.xml -->
<display-name>CCCRest-ear-ear</display-name>
    <module>
        <web>
            <web-uri>CCCRest-web.war</web-uri>
            <context-root>/CCCRest-ear-web</context-root>
        </web>
    </module>
    <module>
        <ejb>CCCRest-ejb.jar</ejb>
    </module>
<library-directory>lib</library-directory>
```

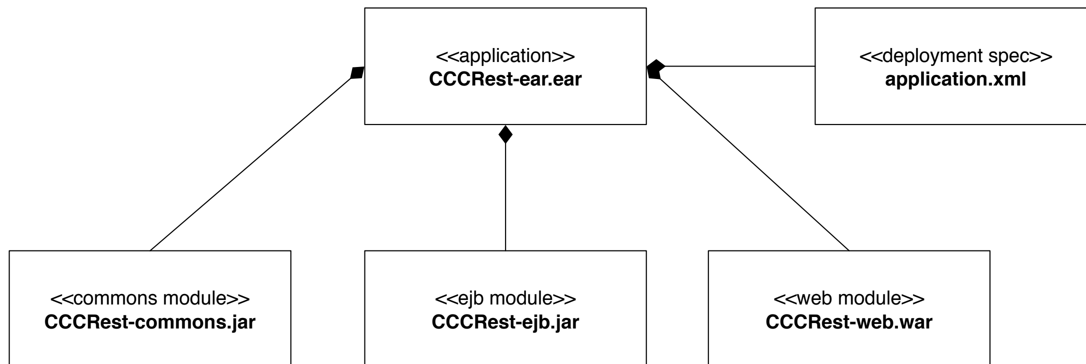

Fig. 10. project structure

The snippet above shows the most relevant part of the deployment descriptor that we use for our project. A deployment descriptor (DD) is an XML file that contains elements describing how we assemble and deploy the application in a particular environment. The display-name element is required to specify the application display name. The module element is used to define a module within an Enterprise application. In our case we use it twice, one to define an ejb element and another to define a web element. The ejb element defines an Enterprise JavaBeans (EJB) module and includes the path to the jar archive in the application. The web element defines a web application module. This element contains a web-uri that defines the location of the path and the name of the war file. Also, it contains a context-root element that includes the context for the web application. Finally, the library-directory element specifies the directory inside the EAR that contains the jar files that can be used by all the modules in the EAR. For instance, the CCCRest-commons.jar is included in the lib folder so that the other modules can use that jar archive.

Presentation Layer Implementation This layer includes the web service that a client can interact and send XML messages or upload a new rule file to the service. Fig. 11 presents the modules of the uk.ac.ncl.web package. The modules in this package first provide an empty Class (RestApplication) that is used to initialize the RESTful interface of Resteasy.

Second, the UploadFileService Class provides the RESTful endpoint service that the client can interact with. In this class public method uploadFile is called from an html form that sends the file to be uploaded to the application server. Another two private methods are used to get the name of the file (getFileName) and write the byte stream to a particular file path on the application server. Currently it saves the file in the /drools/upload directory in JBoss AS base directory. The folder path to save the new rule file can be changed in the enumeration file RuleFilesEnum in package uk.ac.ncl.model.

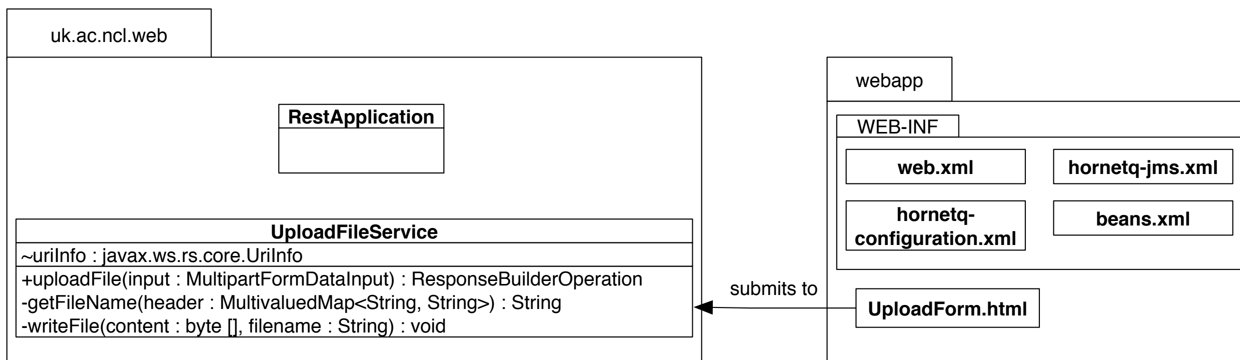

Fig. 11. Package uk.ncl.ac.web and folder webapp UML diagram

Maven requires all web application resources such as html or Java Server Pages (JSP) file to be added in folder webapp. As we can see in Fig. 11 only one html file (UploadForm.html) is added in folder webapp. This file is an html form that lets a user select a new rule file and submit the form with the file using a POST request to the UploadFileService. In addition to web application resources we can use the WEB-INF folder to store all the required configuration files. HornetQ provides the interface for the RESTful queue and it can be configured using the hornetq-jms.xml file. For example, the snippet below shows how we instruct HornetQ to create a new JMS queue with entry name /queue/events and make it available for lookup via Java Naming and Directory Interface (JNDI). Also, HornetQ exposes the JMS queue using a RESTful interface that we can send messages to the queue by any client that can understand HTTP.

```
<!-- jms queues configuration : hornetq-jms.xml -->
<hornetq-server>
   <jms-destinations>
      <jms-queue name="events">
         <entry name="/queue/events" />
      </jms-queue>
      <jms-queue name="replyQueue">
         <entry name="/queue/replyQueue" />
      </jms-queue>
   </jms-destinations>
</hornetq-server>
```

Another configuration file that is used for HornetQ is hornet-configuration.xml. This is the main configuration file for HornetQ server. We use all the default set properties, so a file with a single empty configuration element is sufficient for our project.

The file called web.xml is the Web Application Deployment Descriptor of the web application. This XML document defines everything the server needs

to know: servlets and other components like filters or listeners, initialization parameters, container-managed security constraints, resources, welcome pages, etc. In our project we use it to configure Resteasy in order to expose the UploadFileService as a RESTful service and enable access to HornetQ's queues using REST.

Such configuration files include the two configuration files for HornetQ (hornetq-configuration.xml, hornetq-jms.xml), configuration for RESTful service (web.xml) and for configuring enterprise java beans (beans.xml).

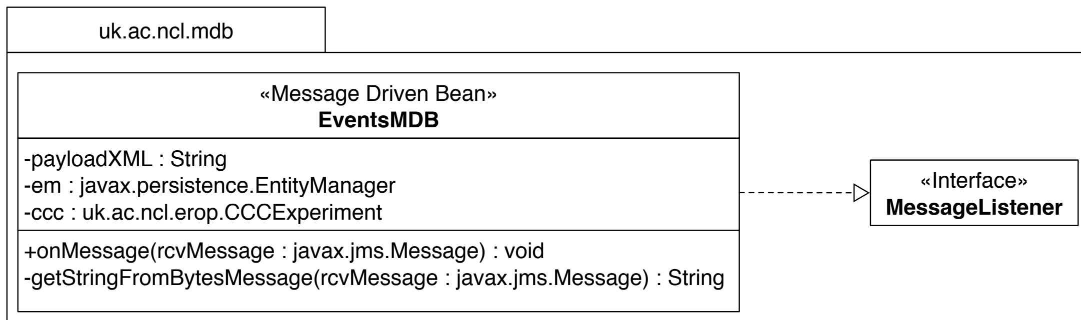

Fig. 12. Package uk.ncl.ac.mdb UML diagram

Fig. 11 shows the UML diagram for the EventsMDB Class. This class resides in package mdb and is a part of CCCRest-ejb.jar module. A Message Driven Bean has only two states: does not exist state and ready to receive messages state. It can receive messages provided from a JMS queue. In our case we use HornetQ to provides us with the queue. When there is a new message in the queue it is sent to the MDB. We can configure the MDB to listen to a particular JMS queue using annotations.

```
@MessageDriven(name = "EventsMDB", activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationType",
        propertyValue = "javax.jms.Queue"),
    @ActivationConfigProperty(propertyName = "destination",
        propertyValue = "queue/events"),
    @ActivationConfigProperty(propertyName = "acknowledgeMode",
        propertyValue = "Auto-acknowledge") })
```

The snippet above shows how we configure the EventsMDB to listen to queue with name events using annotations. The @MessageDriven annotation defines the bean as MDB with the name EventsMDB. The activationConfig attribute describes how the MDB is configured by using the @ActivationConfigProperty. The destinationType property defines what is the type of the destination and in our MDB it is a JMS queue. The JNDI name of the queue is added by

using the destination property, with "queue/events" for our MDB. Finally, the acknowledgeMode is used with the value Auto-acknowledge to define the type of acknowledgement the MDB will use. Using Auto-acknowledge the messages will be acknowledged automatically.

The EventsMDB Class consists of two methods (onMessage, getStringFromBytesMessage) and three class variables. The onMessage method is executed when a message arrives from the queue. The first thing we do when we receive a new message is to set the EntityManager that we will be using in later stages for persistence. Since the message is sent as a ByteMessage we first have to convert it to an XML String using the getStringFromBytesMessage method. The next step is to unmarshall the XML as a BusinessEvent object using Java Architecture for XML Binding (JAXB). The CCC engine is started if it has not been started before by a previous business event. A new CCC Event is instantiated from the BusinessEvent representation and then is fed to the CCC engine for processing. When CCC finishes with processing the BusinessEvent, returns a CCCResponse object that contains the information if the BusinessEvent is contract compliant or not. This object is serialized to an ObjectMessage and send to the replyQueue as a JMS message.

CCC Logic Layer Implementation In this subsection we describe the implementation of the CCC Logic Layer that contains the EROP ontology we presented in a previous section. Fig. 12 shows the packages that exist in the CCCRest-commons module. This subsection will focus on the EROP package and the other three packages will be described in the next subsection.

All the classes for the EROP package are shown in Fig. 13 and a more detailed view of each class is presented in Appendix F. The class that is instantiated when we first receive a new business event from the queue is CCCExperiment. This class is built as a Singleton object that we can reference throughout our code. It makes a new instance of a CCCExperiment or it returns an instance that was created before. This ensures that we always have access to an instance of CCCExperiment. The only parameter that is passed to this class is the path of the rule file for the contract we want to execute. Then we create a new of the EventLogger class to log the business events to the database. The next step includes the creation of a new RelevanceEngine using the EventLogger instance and rule file path as parameters. If any error occurs an exception is thrown. After we instantiate a new RelevanceEngine we create a new TimeKeeper instance using the RelevanceEngine and EventLogger instances and throw any exceptions that might occur. All the previous steps are part of the initialization process of the CCC engine.

As soon as the initialization process executes successfully CCC is ready to accept events for processing. At this point we can start the simulation process by executing the startSimulation method with the list of events as a parameter. Then we initialize the contract in the RelevanceEngine with the TimeKeeper. We create an new event to signal that the simulation has started and we log this event to the database using the EventLogger's method logEvent with event as

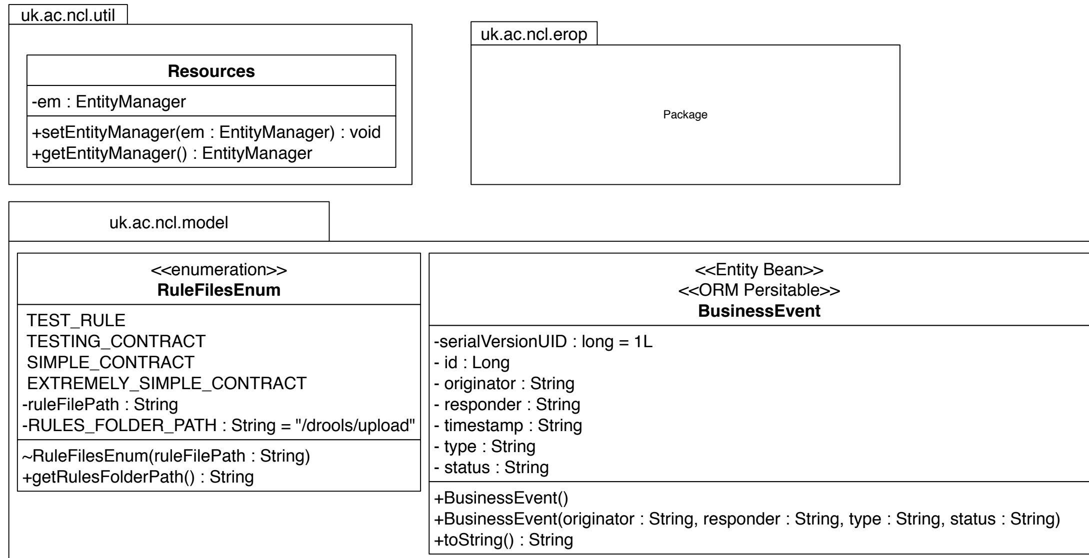

Fig. 13. CCCRest-commons package diagram

parameter. The event is added to the RelevanceEngine instance. Afterwards, we can continue the simulation with the rest of events in our list. The procedure is the same as we have outlined above: for each event we first pause for four seconds, we then log the event to the database and add it to the RelevanceEngine.

The RelevanceEngine class acts as a wrapper for Drools rule engine and its responsibilities include the initialization of the contract, the addition and process of an event. When we create a new RelevanceEngine instance we check if there is an EventLogger present and if the rule file of the contract exists. Also, new KnowledgeBase and StatefulKnowledgeSession instances are created as well as a new event queue as linked list. KnowledgeBase is considered the repository of all the knowledge definitions of the CCC. Similarly, a StatefulKnowledgeSession allow us to make several invocations to the rule engine and provides multiple reasonings for the same set of data while keeping the same state. If there are no exceptions we build a new KnowledgePackage instance from the rule file of the contract and then we add it to the current KnowledgeBase of the RelevanceEngine. During initialization of the contract we require that a TimeKeeper instance is send as a parameter. In this stage, we add all the global objects that we want to have access from inside the rules. These include the BusinessOperation instances that the rule file requires (BuyRequest), the RolePlayer instances (Buyer, Seller) and the ROPSet instances for each RolePlayer.

When an event is added to the RelevanceEngine it is also added to the tail of the event queue. Processing events requires a valid instance of EventLogger

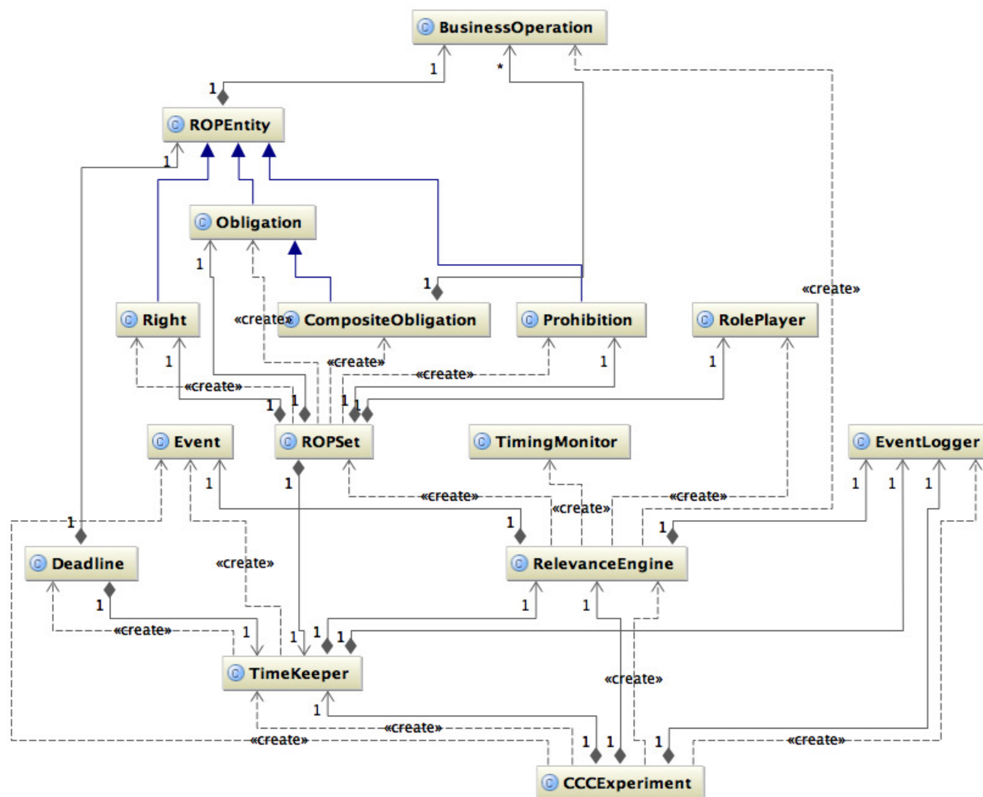

Fig. 14. Package uk.ac.ncl.erop UML class diagram

otherwise we throw an exception that an EventLogger does not exist so we cannot process events. If an EventLogger instance exists then we can poll the queue for an event and insert it to the working session of the rule engine for pattern matching when the rules fire. This is how RelevanceEngine works in the context of the CCC.

Data Access Layer Implementation The Data Access Layer Implementation includes the Object Relation Mapper that we use. We have chosen to use Hibernate as an ORM solution to drive the persistence of the event object. Hibernate uses persistence.xml file to configure the persistence and is also used by JPA. This file is placed in META-INF directory inside the ejb module. The snippet below shows the configuration for the persistence. The persistence-unit attribute is required to have a unique name (RopePU) in the current scoped class loader and also the transaction type that we are using. The provider attribute specifies that we are using HibernatePersistence as the implementation of the JPA Entity

Manager. The jta-data-source includes the jndi name of the database (RopeDS) the persistence unit RopePU maps to. The properties element is used to further configure Hibernate.

```
<!-- Hibernate configuration: persistence.xml -->
<persistence-unit name="RopePU" transaction-type="JTA">
  <provider>org.hibernate.ejb.HibernatePersistence</provider>
  <jta-data-source>java:jboss/datasources/RopeDS</jta-data-source>
  <jar-file>lib/CCCRest-commons.jar</jar-file>
  <properties>
    <property name="hibernate.hbm2ddl.auto" value="update" />
    <property name="hibernate.show_sql" value="true" />
  </properties>
</persistence-unit>
```

The first property is used to update the schema in the database if there are any changes when the application starts again and the second property displays the sql code for each query that is executed.

6.4 Data Store Tier Implementation

This tier includes the type of database we use. Any database that is supported by Hibernate can be used. We have chosen MySQL database to persist the event objects.

6.5 Client Tier Implementation

We have implemented a command-line application that reads a number of XML files for testing purposes. Each xml message represents a Business Event that we send to our web service for consuming. The goal of the command-line application is to take the position of the synchronizer that would send the event to the CCC web service. As soon as the command-line application reads the folder with the XML messages we then initialize the connection to our queue and send the xml messages in the correct order for the web service to consume. The command-line application was also build to assist us with testing the CCC web service we have build. Testing is described in the following section.

7 Testing

The CCC web service was tested using the example contract we presented in the Section 1. The example contract was installed in both the old CCC implementation and the enhanced CCC. We consider the old version to be a reference implementation that works correctly with the example contract and we use it to evaluate the enhanced CCC. Therefore, if the enhanced CCC outputs the same result as the old CCC then we consider the enhanced CCC as a correct implementation.

Our method of testing includes evaluating the messages for the example to be contract compliant or non contract compliant using both the old and new version of CCC. We first follow the correct sequence of messages and assert the outcome of each message in both versions and the we deliberately send a non contract compliant message to evaluate if the enhanced CCC returns the correct outcome.

Table 3 shows the results from testing the CCC. Currently, we incorporate six test cases. The first five test cases follow the sequence of messages send in the right order and follow a correct execution of the example contract. The results of executing the first test cases show that both the reference CCC version and the enhanced CCC version output the same result. In other words, both versions correctly show that the messages for the first five test cases are contract compliant which is the correct behaviour. The last test case presents the scenario that we first send a Buy Request message to the Buyer and then we send a Pay message to the Seller. The results of this test case show that both CCC version have correctly determined that Pay message is non contract compliant. This is the correct behavior as a Pay message is not allowed to be sent right after a Buy Request to the Seller following the rules of the example contract we are using to test the CCC.

Apart from scenario testing, we have used unit testing for each class of the CCC web service. The reason we have used unit testing is to make sure that each class implements its requirements correctly. The unit tests were written using JUnit.

8 Evaluation

This section of the dissertation intends to evaluate the enhanced CCC that was developed. First, we summarize what we tried to accomplish. Second, we discuss our findings from the Related Work Section. Third, we evaluate the design and architecture of the CCC. Finally, we evaluate the implementation of the enhanced CCC. In particular, we evaluate each of the features we added to the new CCC.

Contract monitoring is an active research topic. Businesses want to monitor contractual interactions during the execution of the contract. In this dissertation, our goal was to first migrate to the latest version of Drools that the CCC uses and then enhance the CCC.

During our research of related work in contract monitoring we have examined different categories of contract languages and a range of software systems that are similar to the CCC. Our findings show that formal contract languages based on Deontic logic or Defeasible reasoning [8–10] employ a rigorous formal approach but are not easily understandable by technical and business people or widely used in business world and industry. In contrast, ECA-based languages [11] are widely used in industry and are intuitive for business people to write and understand. This is the main reason that an ECA-based language called EROP is used for specifying the rules of a contract for the CCC.

Table 3. Scenario Testing of CCC

Test Case No	**Test Case**	**Purpose of Test**	**Old CCC outcome**	**Enhanced CCC outcome**
1	Send Buy Request business event	Find out if the outcome of Buy Request event is consistent and correct in both implementations.	contract compliant	contract compliant
2	Send Buy Confirmation business event	Examine if the outcome of Buy Confirmation business event is consistent and correct in both implementations.	contract compliant	contract compliant
3	Send Buy Reject business event	Examine if the outcome of Buy Reject business event is consistent and correct in both implementations.	contract compliant	contract compliant
4	Send Pay business event	Examine if the outcome of Buy Reject business event is consistent and correct in both implementations.	contract compliant	contract compliant
5	Send Cancel business event	Examine if the outcome of Buy Reject business event is consistent and correct in both implementations.	contract compliant	contract compliant
6	Send Pay business event	Find out if the Pay business event after a Buy Request is allowed or not.	non contract compliant	non contract compliant

We have analyzed a number of contract monitors that are similar our CCC. First of all, the Moses middleware [4, 16] that runs the Law Governed Interaction architecture uses a similar approach. It uses Controllers that receive events and take actions according to a knowledge base that contains rules. This behavior is the same that we use in the CCC, we send events to the CCC and we take actions according to a specified set of rules. The main difference is that our CCC is used as a third party entity rather than located in the middle of the interacting parties.

Another contract monitor that is similar to our CCC is Heimdhal [17], which is a middleware platform that can monitor and enforce history based policies. It contains a policy monitor similar to our CCC that monitors and enforces Service Level Agreements (SLAs). Therefore, the focus of Heimdhal is to monitor resource usage policies rather than monitoring business contract interactions.

An outcome from our research of related work is that we were exposed to different architectures that are used to monitor contracts at runtime. This enabled us to design a better architecture after critically evaluating other similar architectures. Our CCC web service as seen in Fig. 7 has a modularized architecture and is divided to three tiers and three layers of functionality. The Client Tier includes the Event Composer that sends the Business Event and the Rule Editor that sends the new rule file to the CCC web service. The Application Tier is divided in three layers of distinct functionality and responsibilities. The main

goal of the Presentation Layer is to accept the new messages and eventually send them to the CCC Logic Layer for processing. The Business Event message sent is persisted to a database using the Data Access Layer to abstract away from the specific implementation of the Data Store Tier that hosts the database server.

One of the advantages of such multi-tier architecture is that we can adjust and improve a certain module without having to rewrite the whole application again. Therefore, we have managed to build a loosely coupled architecture that does not have a strong dependency to other modules. A disadvantage of this is that we have a multi faceted architecture that is more difficult to manage than if we had for example a monolithic architecture where all functionality is placed in one big complex module.

Table 4 summarizes the main features that were implemented for the enhanced CCC to address the limitations of the old CCC. We have managed to build the CCC as a RESTful web service that we can send business events using the HTTP protocol. This a big improvement from the old implementation that used only hardcoded events. This also makes the enhanced CCC more dynamic in nature as we do not depend on hardwired events. Another main characteristic of the enhanced CCC is that follows the REST paradigm to build the web service instead of being a self-contained application. The advantage of this is that any client that can send an HTTP request can interact with the enhanced CCC and this was not possible with the old version.

Hardcoded rules is one of the limitations that we managed to address and provide a way for a user to upload a new set of rules to the enhanced CCC. This greatly improves the enhanced CCC and there is possibility in the future that we can execute different contracts that we have uploaded using this feature. The old CCC uses a specific database vendor and its persistence logic is tightly coupled with the database. We wanted to abstract away from the specific persistence implementation of a database vendor. This was achieved by using an Object Relationship Mapper (ORM) that manages the database implementation for us. Another limitation that was addressed is migrating from Drools 4 to the latest version of Drools. This improves the performance of our rule engine and opens the possibility of using other components of Drools that deal with complex event processing and business workflows. Finally, we have added two external queues to our web service that are used to place the incoming messages to a business event queue and a reply queue where we place the messages that client will receive. This improves our reliability of the CCC web service because we are using HornetQ managed queues. HornetQ can be configured to have durable messages. For example if the application server crashes then the messages in the queue are not lost because they are saved in the file system of the application server.

The first objective of our dissertation was to evaluate critically, related work in monitoring contracts and analyze the old CCC. In section 2 we evaluated related work in contract monitoring and contract languages. For the second part of this objective, we analyzed the old CCC and discovered a number of limitations

for the old CCC. The limitations that we have addressed in this project from our analysis are shown in the first column of Table 4.

The second and third objective to enhance the design of the CCC and expose it as a RESTful web service have been met. The features we have added to the enhanced CCC can be seen in the second column of Table 4. In particular we map each limitation we found from our analysis to an enhancement we have managed to include to the new CCC.

Table 4. Summary of features added in CCC to address limitations

Old CCC	**Enhanced CCC**
Hardcoded business events	Send business events to the web service using HTTP.
Self-contained java application	Web service that follows REST paradigm.
Hardcoded rules.	Upload a new rule file to CCC to update the rule repository.
Persistence coupled to a specific database vendor.	Uses an Object Relationship Mapper (Hibernate)
Uses Drools 4.	Migrated to Drools 5.4.0
Uses only one internal queue	Uses an external and internal event queue in CCC and a reply queue.
Monolithic architecture	Modularized loosely coupled architecture

In the previous section, we described our testing process we have used to ensure the high quality of the CCC. We have used unit testing and scenario testing. We have managed to cover most of the code base using unit tests but more specific testing would be required when we use different contracts. Additionally, during our scenario testing we could have used more messages that are not contract compliant to further test the correct behavior of the enhanced CCC in relation with the old one. The results of testing both versions of CCC show that they return the same outcome when they are fed the same sequence of events. Thus, our fourth objective related to testing the CCC is met.

9 Conclusion

The main aim of this dissertation was to migrate an implementation of a contract compliance checker implemented in Drools 4 to the latest version of Drools 5. In pursuit of this aim, we evaluated in related work contract languages and research conducted in contract monitoring. We learnt that a number of similar systems exist that take similar approaches on building a contract monitor and as a web service. Also, we learnt that a number of contract formalisms exist that are formally defined. In our project we took a less formal approach but more

pragmatic approach that can be understood by a wider audience, as opposite to people with expertise in formal notations.

We performed an analysis of the old CCC and identified some of its limitations. We selected some of them for improvement and discuss others in Future Work. In sections 5, 6, 7 we described our enhanced architecture for the CCC and focused on each layer separately. The implementation section shows how we implemented the system and what technologies were used to accomplish this. Finally, we presented how we tested the CCC and what are our testing results and evaluated if we met our objectives.

The enhanced CCC can accept events from outside world, as opposed to the internal files, but we have not managed to perform and test this exhaustively for example by using a large number of different contracts. We have only tested with the help of the example contract we presented in the introduction. A realistic business contract can have many more clauses and, in extension, rules for the CCC. Another limitation of the enhanced CCC, not addressed due to time constraints, is that we have not integrated the historical queries that exist in the old version and we have not build a more generic loader to load dynamically all the objects needed such as required role players, business operations etc. Finally, only a command-line client was developed for testing purposes rather than a client with a graphical user interface.

Even though, the enhanced CCC is far from being a complete implementation that can be used for realistic applications, we claim that we have greatly transformed the old version to a RESTful web service that employs a loosely coupled architecture and can accept events from the outside world. Now, we have at our disposal a ready system that we can integrate with other web services such as the synchronizer we presented in this paper.

10 Future Work

There are number of issues we would want to fix in the future for the CCC. Currently, if we have a deadline that a Buy Request has to be submitted by Friday, the CCC does not update automatically the state of the business operation in the ROP set when the current day is Saturday as it was supposed to do. Therefore we need to employ a scheduling mechanism or Timing Monitor that would fire an event to automatically update the business operation in the ROP sets to a correct state.

Another issue we would want to address is that now we have only two states for the business operations inside the ROP sets. They are either active or inactive. Adding more states such as expired or satisfied would better describe the life cycle of the contract during runtime. This would also make formal verification easier.

Building a resolution strategy when we have conflicting rules specifically targeted towards business contracts would greatly improve the CCC. Currently, we rely on Drools internal conflict resolution strategy that can execute a rule before another that has higher priority but we would want to provide a better

resolution strategy that is build specifically for business contracts. For example, we would want to know if we have two rules: (1) buyer gets a discount of 10 percent after buying two products and (2) buyer gets a 20 percent discount after buying two products, then which of the two rules should we execute.

In the future, we would also like to improve the security of the CCC web service and include authentication and non repudiation mechanisms that ensure that communications are secure and that we interact we the correct synchronizer. Finally, we would be keen on improving the reliability of the CCC by introducing fault-tolerance techniques when we consume many messages. This could be achieved by using remote message queues that can be used if the current queues cannot be reached.

Acknowledgments. I would like to thank Ellis Solaiman, Carlos Molina-Jimenez for supporting me and providing very valuable guidance during the course of the project and Massimo Strano for providing valuable insight to the internals of the CCC.

References

1. Molina-Jimenez, C., Shrivastava, S., Strano, M.: A model for checking contractual compliance of business interactions. IEEE Transactions on Service Computing **5** (2012) 276–289
2. Governatori, G., Milosevic, Z., Sadiq, S.: Compliance checking between business processes and business contracts. In: 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06), Ieee (2006) 221–232
3. Marjanovic, O., Milosevic, Z.: Towards formal modeling of e-Contracts. In: Enterprise Distributed Object Computing Conference, 2001. EDOC '01. Proceedings. Fifth IEEE International, IEEE Comput. Soc (2001) 59–68
4. Minsky, N., Ungureanu, V.: Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. ACM Transactions on Software Engineering and Methodology **9** (2000) 273–305
5. RosettaNet: Rosettanet implementation framework: Core specification, version v02.00.01. http://www.rosettanet.org (2002) Accessed 30/08/2012.
6. JBoss: Drools. http://www.jboss.org/drools/ (2012) Accessed 30/08/2012.
7. Hvitved, T.: A Survey of Formal Languages for Contracts. In: Fourth Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'10). (2010) 29–32
8. Governatori, G., Pham, D.H.: DR-CONTRACT: an architecture for e-contracts in defeasible logic. International Journal of Business Process Integration and Management **4** (2009) 187
9. Lee, R.M.: A logic model for electronic contracting. Decision Support Systems **4** (1988) 27–44
10. Governatori, G.: Representing business contracts in RuleML. International Journal of Cooperative Information Systems **14** (2005) 181–216
11. Linington, P., Milosevic, Z., Cole, J., Gibson, S., Kulkarni, S., Neal, S.: A unified behavioural model and a contract language for extended enterprise. Data & Knowledge Engineering **51** (2004) 5–29

12. Andersen, J., Elsborg, E., Henglein, F., Simonsen, J.G., Stefansen, C.: Compositional specification of commercial contracts. International Journal on Software Tools for Technology Transfer **8** (2006) 485–516
13. Prisacariu, C.: A formal language for electronic contracts. In: Formal Methods for Open Object-Based Distributed Systems. (2007) 174–189
14. Molina-Jimenez, C., Shrivastava, S., Crowcroft, J., Gevros, P.: On the monitoring of contractual service level agreements. Proceedings. First IEEE International Workshop on Electronic Contracting, 2004. (**2**) 1–8
15. Elgammal, A., Turetken, O., Heuvel, W.j.V.D., Papazoglou, M.: On the formal specification of business contracts and regulatory compliance. In: Fourth Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'10). (2010) 29–32
16. Minsky, N., Ungureanu, V.: Scalable regulation of inter-enterprise electronic commerce. In: Electronic Commerce. Number November. (2001) 219–232
17. Gama, P., Ribeiro, C., Ferreira, P.: Heimdhal: A history-based policy engine for grids. In: Sixth IEEE International Symposium on Computing and the Grid. (2006) 480–488
18. Perrin, O., Godart, C.: An approach to implement contracts as trusted intermediaries. In: Proceedings. First IEEE International Workshop on Electronic Contracting, 2004., Ieee (2004) 71–78
19. Ludwig, H., Stolze, M.: Simple obligation and right model (sorm) - for the runtime management of electronic service contracts. In Bussler, C., Fensel, D., Orlowska, M.E., Yang, J., eds.: WES. Volume 3095 of Lecture Notes in Computer Science., Springer (2003) 62–76
20. Linington, P.: Automating support for e-business contracts. International Journal of Cooperative Information **14** (2005) 77–98
21. OASIS: ebxml: Business process spec. schema tech. spec. v2.0.4. http://-docs.oasisopen.org/ebxml-bp/2.0.4/OS/specebxmlbp-v2.0.4-Spec-os-en.pdf (2006) Accessed 30/08/2012.
22. Strano, M., Molina-Jimenez, C., Shrivastava, S.: Implementing a rule-based contract compliance checker. In: Software Services for e-Business and e-Society. Volume 305. (2009) 96–111
23. JBoss: Jboss application server 7. http://www.jboss.org/jbossas/ (2012) Accessed 30/08/2012.
24. JCP: Jsr-94 java rule engine api. http://jcp.org/aboutJava/communityprocess/final/jsr094/index.html (2004) Accessed 30/08/2012.
25. Forgy, C.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial intelligence **19** (1982) 17–37

11 Appendix A: Running old CCC with Drools 4

In order to run CCC using Drools 4, all the libraries mentioned in Section 6.1 are required to be specified in the class path of the operating system in use. Therefore, the whole distribution for Drools version 4.0.2 would need to be downloaded from JBoss Drools website. The distribution not only contains the main Drools libraries but also the dependencies for these libraries. Also, a drools plugin (JBoss Tools) for eclipse should be installed so that rules can be edited and debugged. The oldest java version to run CCC successfully is 1.5. Another prerequisite for

running CCC is a new MySQL installation. Additionally, the CCC is provided with a contract hardwired into the code. The steps required to run the old CCC are outlined below:

1. Import CCC project to Eclipse IDE.
2. Add a new runtime Drools version in "Installed Drools Runtime" preference. Choose a runtime Drools version according to our requirements. In this case Drools version 4 should be chosen.
3. Add a new User library and select drools-core.jar, drools-compiler.jar, drools-jsr94.jar, drools-decisiontables.jar. Then, add the Drools user library to the build path in Eclipse IDE.
4. Add all the libraries present in Drools 4.0.2 /lib folder as external jar libraries.
5. MySQL is used for backend and all the required information such as username, password can be changed in file /uk/ac/ncl/erop/EropDbConstValues.java. Furthermore, a mysql connector library should be added to the build path as an external jar library (mysql-connector-java-5.0.8.jar).
6. The correct rule file name and path have to be entered in file CCCExperiment.java. In more details the correct information has to be entered in line 14: `relevanceEngine = new RelevanceEngine("src/SimpleContract.drl", logger);`
7. Several changes in RelevanceEngine.java need to be done. The first change is to disable time monitoring which is used for testing performance. Therefore, in line 22: `boolean performanceTestingOn = true;` instead of true, false has to be added. Other changes include enabling (removing comments) lines 114 to 119 so that all the global objects required by the contract are added to the working memory. Also, lines 126 and 128 have to be commented out because they are not needed by the contract.
8. CCC can be run using the Eclipse IDE and choosing CCCExperiment.java for the Run As option. Fig. 9 below shows a sample running of the CCC using Drools 4.

```
CCCExperiment (2) [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java (Jun 17, 2012 5:14:28 PM)
Initializing objects...
Driver registration success
Connection established
TimeKeeper successfully instantiated
Initializing contract...
Initialization complete
- Adding new event to queue
+ Processing event: Event - Originator: none, Responder: none, Timestamp: 6/17/12 5:14 PM, Type: init, Status: success
Catchall rule triggered - Event received of type init from none to none with status success
  Deadline added: Purchase Order-Obligation-buyer-seller
Initialization rule triggered
- Adding new event to queue
+ Processing event: Event - Originator: buyer, Responder: seller, Timestamp: 6/17/12 5:14 PM, Type: Purchase Order, Status: success
Catchall rule triggered - Event received of type Purchase Order from buyer to seller with status success
  Deadline removed: Purchase Order-Obligation-buyer-seller
PO Received Rule triggered
- Adding new event to queue
+ Processing event: Event - Originator: seller, Responder: buyer, Timestamp: 6/17/12 5:14 PM, Type: Purchase Order Acceptance, Status: success
Catchall rule triggered - Event received of type Purchase Order Acceptance from seller to buyer with status success
- Adding new event to queue
+ Processing event: Event - Originator: buyer, Responder: seller, Timestamp: 6/17/12 5:14 PM, Type: Fine Payment, Status: success
Catchall rule triggered - Event received of type Fine Payment from buyer to seller with status success
- Adding new event to queue
+ Processing event: Event - Originator: buyer, Responder: seller, Timestamp: 6/17/12 5:14 PM, Type: Purchase Order, Status: success
Catchall rule triggered - Event received of type Purchase Order from buyer to seller with status success
- Adding new event to queue
+ Processing event: Event - Originator: seller, Responder: buyer, Timestamp: 6/17/12 5:14 PM, Type: Purchase Order Acceptance, Status: success
Catchall rule triggered - Event received of type Purchase Order Acceptance from seller to buyer with status success
- Adding new event to queue
+ Processing event: Event - Originator: buyer, Responder: seller, Timestamp: 6/17/12 5:14 PM, Type: Fine Payment, Status: success
Catchall rule triggered - Event received of type Fine Payment from buyer to seller with status success
- Adding new event to queue
+ Processing event: Event - Originator: buyer, Responder: seller, Timestamp: 6/17/12 5:14 PM, Type: Purchase Order, Status: success
Catchall rule triggered - Event received of type Purchase Order from buyer to seller with status success
- Adding new event to queue
+ Processing event: Event - Originator: seller, Responder: buyer, Timestamp: 6/17/12 5:14 PM, Type: Purchase Order Acceptance, Status: success
Catchall rule triggered - Event received of type Purchase Order Acceptance from seller to buyer with status success
- Adding new event to queue
+ Processing event: Event - Originator: buyer, Responder: seller, Timestamp: 6/17/12 5:14 PM, Type: Fine Payment, Status: success
Catchall rule triggered - Event received of type Fine Payment from buyer to seller with status success
- Adding new event to queue
+ Processing event: Event - Originator: buyer, Responder: seller, Timestamp: 6/17/12 5:14 PM, Type: Purchase Order, Status: success
Catchall rule triggered - Event received of type Purchase Order from buyer to seller with status success
- Adding new event to queue
+ Processing event: Event - Originator: seller, Responder: buyer, Timestamp: 6/17/12 5:14 PM, Type: Purchase Order Acceptance, Status: success
Catchall rule triggered - Event received of type Purchase Order Acceptance from seller to buyer with status success
- Adding new event to queue
+ Processing event: Event - Originator: buyer, Responder: seller, Timestamp: 6/17/12 5:14 PM, Type: Payment, Status: success
Catchall rule triggered - Event received of type Payment from buyer to seller with status success
- Adding new event to queue
+ Processing event: Event - Originator: seller, Responder: buyer, Timestamp: 6/17/12 5:14 PM, Type: Goods Delivery, Status: success
Catchall rule triggered - Event received of type Goods Delivery from seller to buyer with status success
*** Simulation Complete - Waiting...
```

Fig. 15. Running old CCC with Drools 4

12 Appendix B: Running CCC with Drools 5.4.0

After migrating from Drools 4 to Drools 5.4.0 we can use maven to run CCC as a self-contained command-line application. In order to build the project into a jar library file, we execute the following command:

```
mvn clean compile assembly:single
```

After executing the above command we can run the CCC from the command-line using the following line:

```
java -jar target/CCCMigration-1.0-SNAPSHOT-jar-with-dependencies.jar
```

As we can see in the Fig. below, the output from the execution is the same as in Fig. 10.

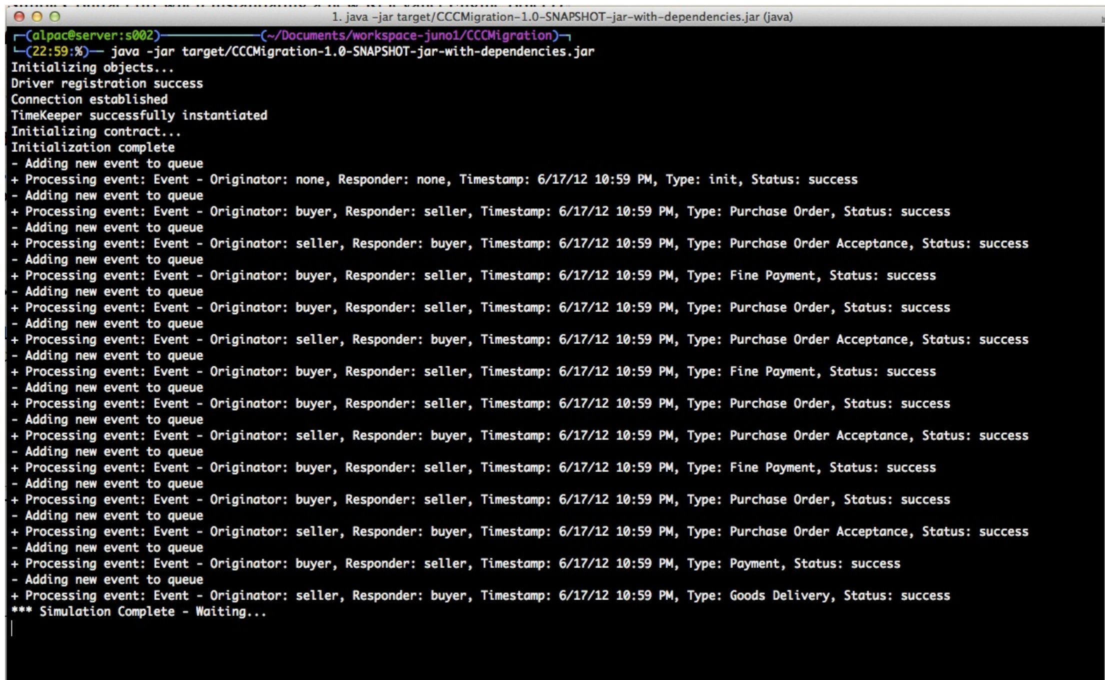

Fig. 16. Running CCC using Drools 5.4.0

13 Appendix C: Running CCC Web service with Drools 5.4.0

In order to run CCC as a web service a number of requirements have to be met. The first requirement is an installation and configuration of JBoss Application Server version 7.1.0. Then the second requirement is to update the pom.xml to include all the dependencies needed for the RESTful web service.

After the requirements have been met JBoss can be started using the run.sh script that can be found in the root folder of the project. The next step includes the execution of a maven process to compile the code for CCC, package code and libraries as a war file and deploy it to the running JBoss AS. All these goals can be met by executing the following command:

```
mvn clean package jboss-as:deploy -Dmaven.test.failure.ignore=true
```

When maven has deployed the new version of the CCC we can start sending HTTP POST requests which contain the events involved in the contract example shown in page 1 namely Buy Request, Pay, Cancel following the XML structure for an event as discussed in Section 4. Following this way we can test if CCC reacts appropriately to the event entered. A running output of a BuyRequest event sent to the web service is shown in Fig. 17 below. Fig. 18 shows the

BuyRequest event from the perspective of the client and the XML message it pulls from the queue.

```
2. ./run.sh (sh)
00:37:31,427 INFO  [class uk.ac.ncl.mdb.EventsMDB] (Thread-2 (HornetQ-client-global-threads-32174685))
Received message: <?xml version="1.0" encoding="UTF-8" standalone="yes"?><event><originator>buyer</orig
inator><responder>seller</responder><status>success</status><type>BuyRequest</type></event>
00:37:31,431 INFO  [class uk.ac.ncl.mdb.EventsMDB] (Thread-2 (HornetQ-client-global-threads-32174685))
Received BusinessEvent: BusinessEvent [originator=buyer, responder=seller, type=BuyRequest, status=succ
ess]
00:37:31,432 INFO  [class uk.ac.ncl.erop.CCCExperiment] (Thread-2 (HornetQ-client-global-threads-321746
85)) Initializing objects...
00:37:31,804 INFO  [stdout] (Thread-2 (HornetQ-client-global-threads-32174685)) TimeKeeper successfully
 instantiated
00:37:31,806 INFO  [class uk.ac.ncl.mdb.EventsMDB] (Thread-2 (HornetQ-client-global-threads-32174685))
event: Event - Originator: buyer, Responder: seller, Timestamp: 30/08/2012 00:37:31, Type: BuyRequest,
Status: success
00:37:31,815 INFO  [class uk.ac.ncl.mdb.EventsMDB] (Thread-2 (HornetQ-client-global-threads-32174685))
CCC started? true
00:37:31,817 INFO  [class uk.ac.ncl.erop.RelevanceEngine] (Thread-2 (HornetQ-client-global-threads-3217
4685)) - Adding new event to queue
00:37:31,820 INFO  [class uk.ac.ncl.erop.RelevanceEngine] (Thread-2 (HornetQ-client-global-threads-3217
4685)) + Processing event: Event - Originator: buyer, Responder: seller, Timestamp: 30/08/2012 00:37:31
, Type: BuyRequest, Status: success
00:37:31,823 INFO  [class uk.ac.ncl.mdb.EventsMDB] (Thread-2 (HornetQ-client-global-threads-32174685))
cccResponse: CCCResponse [isContractCompliant=true]
00:37:31,834 INFO  [stdout] (Thread-2 (HornetQ-client-global-threads-32174685)) <?xml version="1.0" enc
oding="UTF-8" standalone="yes"?>
00:37:31,835 INFO  [stdout] (Thread-2 (HornetQ-client-global-threads-32174685)) <result>
00:37:31,836 INFO  [stdout] (Thread-2 (HornetQ-client-global-threads-32174685))     <contractCompliant>
true</contractCompliant>
00:37:31,836 INFO  [stdout] (Thread-2 (HornetQ-client-global-threads-32174685)) </result>
00:37:31,856 INFO  [stdout] (Thread-2 (HornetQ-client-global-threads-32174685)) Hibernate: insert into
eventhistory (originator, responder, status, timestamp, type, id) values (?, ?, ?, ?, ?, ?)
```

Fig. 17. Running CCC as a web service

```
<terminated> RESTClient [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java (Aug 30, 2012 12:50:24 AM)
bevent1.xml
BusinessEvent [originator=buyer, responder=seller, type=BuyRequest, status=success]
log4j:WARN No appenders could be found for logger (org.jboss.resteasy.plugins.providers.DocumentProv
ider).
log4j:WARN Please initialize the log4j system properly.
Send business event: BusinessEvent [originator=buyer, responder=seller, type=BuyRequest, status=succ
ess]
Status:201
pullConsumers: <http://localhost:8088/CCCRest-ear-web/queues/jms.queue.replyQueue/pull-consumers>
consumeNext url: http://localhost:8088/CCCRest-ear-web/queues/jms.queue.replyQueue/pull-consumers/at
tributes-0/3-queue-jms.queue.replyQueue-1346283419014/consume-next-1
response: 200
CCCResponse [isContractCompliant=true]
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<result>
    <contractCompliant>true</contractCompliant>
</result>
```

Fig. 18. CCC web service client

14 Appendix D: Example Contract Code Listing in Augmented Drools

```
package ExampleContract

# Import Java classes for EROP support
import uk.ac.ncl.erop.*;

# Global variables (persistent objects passed from outside)
global RelevanceEngine engine;
global EventLogger logger;

global RolePlayer buyer;
global RolePlayer seller;
global ROPSet ropBuyer;
global ROPSet ropSeller;
global TimingMonitor timingMonitor;


global BusinessOperation buyRequest;
global BusinessOperation payment;
global BusinessOperation buyConfirmation;
global BusinessOperation buyRejection;
global BusinessOperation cancelation;
```

```
# Rule 0: initialize the ROP sets for buyer and seller.
# This rule is launched only when the contract is set up.
# In this limited scenario the seller has no ROP, but the buyer
# starts with the right to submit a buy request.

rule " Initialization "
    when
        $e: Event (type == "init")
    then
        System.out.println("* Initialization when");
        # Add buyer's right to submit an order
        ropBuyer.addRight(buyRequest, seller, (String)null);
        System.out.println("* Initialization rule triggered");
end

# Rule 1: having received a Buy Request event from the buyer, his right to
    submit another
# is temporarily revoked until the current one is completed. The seller gains
# an obligation to either accept or reject the Buy Request.
rule "Buy Request Received"
    when
        # Verify type of event, originator, and responder
        $e: Event(type=="Buy Request", originator=="buyer", responder
    =="seller", status=="success")
        eval(ropBuyer.matchesRights(buyRequest))
    then
        # Remove buyer's right to place other Buy Requests
        ropBuyer.removeRight(buyRequest, seller);
        # Add seller's obligation to either accept or reject order
        BusinessOperation[] bos = {buyConfirmation, buyRejection};
        ropSeller.addObligation("React To Buy Request", bos, buyer, 3);
        System.out.println("* Buy Request Received rule triggered");
end

# Rule 2: having received a reject Buy Request event from the seller, the
    pending obligation
# is satisfied. Restore buyer's right to submit Buy Requests.#
rule "Buy Request Rejected"
    when
        $e: Event(type=="Buy Request Rejection", originator=="seller",
    responder=="buyer", status=="success")
        eval(ropSeller.matchesObligations("React To Buy Request"));
    then
        # Buyer's Obligation is satisfied, remove it
        ropSeller.removeObligation("React To Buy Request", buyer);
```

```
        # Restore buyer's right to submit other Buy Requests
        ropBuyer.addRight(buyRequest, seller, (String)null);
        System.out.println("* Buy Request Rejected rule triggered");
end

# Rule 3: having received an accept Buy Request event from the seller, the
    pending obligation
# is satisfied . New obligation on buyer to pay seller .
rule "Buy Request Accepted"
    when
        $e: Event(type=="Buy Request Acceptance", originator=="seller",
    responder=="buyer", status=="success")
        eval( ropSeller .matchesObligations("React To Buy Request"));
    then
        # Buyer's Obligation is satisfied , remove it
        ropSeller .removeObligation("React To Buy Request", buyer);
        # Add new obligation for buyer to pay a bill to the seller within 7
    days!
        ropBuyer.addObligation(payment, seller, 7);
        System.out.println("* Buy Request Accepted rule triggered");
end

# Rule 4: the obligation to react to the buyer's PO times out. The
    obligation
# is now irrelevant - timeout is treated as a refusal .
rule "React To Buy Request Timeout"
    when
        $e: Event(type=="React To Buy Request Timeout", originator=="
    seller", responder=="buyer")
        eval( ropSeller .matchesObligations("React To Buy Request"));
    then
        # Seller's Obligation is irrelevant , remove it
        ropSeller .removeObligation("React To Buy Request", buyer);
        # Restore buyer's right to submit other Buy Requests
        ropBuyer.addRight(buyRequest, seller, (String)null);
        System.out.println("* React To Buy Request Timeout rule triggered")
    ;
end

# Rule 5: buyer pays. Obligation satisfied , The buyer regains the right to
    submit Buy Requests.
rule "Payment Received"
    when
        $e: Event(type=="Payment", originator=="buyer", responder=="
    seller", status=="success")
```

```
        eval(ropBuyer.matchesObligations(payment))
    then
        # Buyer's Obligation is satiasfied , remove it.
        ropBuyer.removeObligation(payment, seller);
        # Restore buyer's right to submit other Buy Requests
        ropBuyer.addRight(buyRequest, seller, (String)null);
        System.out.println("* Payment rule triggered");
end
```

15 Appendix E: Example Contract Code Listing in EROP language

```
roleplayer buyer, seller;
businessoperation buyRequest, payment, buyConfirmation;
businessoperation buyRejection, cancellation;
compoblig RespondToBuyRequest(buyConfirmation, buyRejection);

rule "R1"
    when
        e matches (botype == "buyRequest")
    then
        Success:
            if e.originator == buyer
               && buyRequest in buyer.rights
               && e.weekday in [Monday ... Saturday]
               && e.time in [9 ... 18]
            then
                seller.obligs += RespondToBuyRequest("72h");
            endif
        Otherwise:
            pass;
end

rule "R2Confirmation"
    when
        e matches (botype == "buyConfirmation")
    then
        Success:
            if e.originator == seller
               && RespondToBuyRequest in seller.obligs
            then
                seller.obligs -= RespondToBuyRequest;
                buyer.obligs += Payment("7d");
                buyer.rights += Cancellation("7d");
            endif
        Otherwise:
```

```
        pass;
end

rule "R2Rejection"
   when
      e matches (botype == "buyRejection",
         outcome == Success, originator == seller)
      RespondToBuyRequest in seller.obligs
   then
      seller.obligs -= RespondToBuyRequest;
      terminate("Success");
end

rule "R2Timeout"
   when
      e matches (botype = buyRejectionTimeout, originator == seller)
      RespondToBuyRequest in seller.obligs
   then
      seller.obligs -= RespondToBuyRequest;
      terminate ("BizFail");
end

rule "R3Payment"
   when
      e matches (botype = payment, originator== buyer)
      payment in buyer.obligs;
   then
      buyer.obligs -= Payment("7d");
      terminate("Success");
   endif
end

rule "R3Cancellation"
   when
      e matches (botype = cancellation, originator== buyer)
      cancellation in buyer.rights;
   then
      buyer.rights -= Cancellation("7d");
      terminate("Success");
   endif
end
```

16 Appendix F: UML class diagram for uk.ac.ncl.erop package

Fig. 19. Detailed UML class diagram for uk.ac.ncl.erop package