Implementing a Rule-Based Contract Compliance Checker

Massimo Strano, Carlos Molina-Jimenez, Santosh Shrivastava

Newcastle University, Newcastle upon Tyne, UK
{massimo.strano, carlos.molina, santosh.shrivastava}@ncl.ac.uk

Abstract. The paper describes the design and implementation of an independent, third party contract monitoring service called Contract Compliance Checker (CCC). The CCC is provided with the specification of the contract in force, and is capable of observing and logging the relevant business-to-business (B2B) interaction events, in order to determine whether the actions of the business partners are consistent with the contract. A contract specification language called EROP (for Events, Rights, Obligations and Prohibitions) for the CCC has been developed based on business rules, that provides constructs to specify what rights, obligation and prohibitions become active and inactive after the occurrence of events related to the execution of business operations. The system has been designed to work with B2B industry standards such as ebXML and RosettaNet.

1 Introduction

There is a growing interest - both within industry and academia - in exploring innovative ways of automating the management and regulation of business interactions using electronic contracting systems. By regulation we mean monitoring and/or enforcement of business–to–business (B2B) interactions to ensure that they comply with the rights (permissions), obligations and prohibitions stipulated in contract clauses. A well designed electronic contracting system can play a central role in ensuring that the business processes of partners perform actions that comply with the contract in force, detecting violations, facilitating dispute resolution and determining liability by providing an audit trail of business interactions. Within this context, we consider the design and implementation of an independent, third party contract monitoring service called *Contract Compliance Checker* (CCC). The CCC is provided with the specification of the contract in force, and is capable of observing and logging the relevant B2B interaction events, as well as determining whether the actions of the business partners are consistent with the contract. We consider here the basic functionality of the CCC, that of a passive observer. It is possible to extend this functionality to make the CCC into a *contract enforcer* that ensures that business partners execute only those operations that are permitted by the contract; however, this aspect is not considered in this paper.

To realise the third party service, the computer infrastructure of business partners concerned with B2B messaging must be instrumented to create a monitoring channel to the CCC for it to observe the relevant B2B events accurately. More precisely, we assume the existence of a *monitoring channel* with the properties: (i) transmission and processing delays of events originating at business partners to the CCC are bounded and known; and (ii) events are delivered exactly once to the CCC in temporal order. We also assume that all clocks within the system are synchronised to a known accuracy. Given the above assumptions, we concentrate in this paper on how an appropriately structured contract can be used to analyse the events collected at the CCC for compliance checking. Subsequently we explain compliance checking in more detail as well as what B2B events need to be collected at the CCC.

We have developed a contract specification language called *EROP* (for Events, Rights, Obligations and Prohibitions) for the CCC, based on business rules, that provides constructs to specify what rights, obligation and prohibitions become active and inactive after the occurrence of events related to the execution of business operations [1, 2]. Our language is particularly suited to the specification of *exceptional* (or *contingency*) clauses that come in force when the delivery obligation stated in the 'primary clause' is not fulfilled (breach or violation of the contract). As we argue in [2], in electronic contracts it is important to distinguish violations caused by infrastructure level problems, arising primarily because of the inherently distributed nature of the underlying computations (e.g., clock skews, unpredictable transmission delays, message loss, incorrect messages, node crashes etc.) from those that are not and are mostly human/organisation related. Our language takes this factor into account and provides intuitive ways of specifying the consequences of the above problems.

In this paper we describe the design and implementation of a CCC service for contracts written in the EROP language. The service relies on the JBoss Rules [3], commonly known as Drools, for rule management. For each partner, the current set of business operations that the partner can execute are classified into *Rightful*, *Obligatory* and *Prohibited* and are explicitly stored in the current ROP set and available for consultation and update. Additional Java components for Drools implement the functionality required for the manipulation of ROP sets, historical queries and timer management.

To be effective, a third party service must be able to work with standards compliant B2B messaging systems. Our system has been designed to work with industry standards such as ebXML [4] and RosettaNet [5]. Thus, we require that business interactions between partners are based on the model presented in the next Section, that preserves the essential aspects of these standards, abstracting away low level protocol details.

The rest of this paper is organized as follows. The next Section defines the basic concepts of this work and presents a sample contract used further on to provide an example; Section 3 introduces the architecture of the CCC. Section 4 focuses on the implementation of the CCC itself. Section 5 elaborates on the translation of the EROP language to Drools rule files; Section 6 discusses related work, and finally Section 7 presents concluding remarks.

2 Contracts and Business Operations

Contract clauses state what business operations the partners are permitted (equivalently, have the right), obliged and prohibited to execute. Informally, a right is something that a business partner is allowed to do; an obligation is something that a business partner is expected to do unless they wish to take the risk of being penalised; finally, a prohibition is something that a business partner is not expected to do unless they are prepared to be penalised. The clauses also stipulate when, in what order and by whom the operations are to be executed. For instance, for a buyer-seller business partnership, the contract would stipulate when purchase orders are to be submitted, within how many days of receiving payment the goods have to be delivered, and so on.

As an example, a hypothetical contract between a buyer and seller is shown below. In this example, clause C1 grants the buyer a right; similarly, clause C2 imposes an obligation on the seller. Of particular interest is C7, which illustrates a clause that takes into account problems caused by infrastructure level problems; our study of messaging standards such as eBXML [4], RosettaNet [5] suggests that at the highest level of specification (e.g., legal English), such problems can be referred to as business problems (problems caused by semantic errors in business messages, preventing their processing) and technical problems (problems caused by faults in networks and hardware/software components). This aspect is discussed further below.

C1: The Buyer has the right to submit a Purchase Order, as long as the submission happens from Monday to Friday and from 9am to 5pm.

C2: The Seller has the obligation to either accept or refuse the Purchase Order within 24 hours. Failure to satisfy this obligation will abort the business transaction for an offline resolution.

C3: If the Order is accepted, the Seller has the obligation to submit an invoice within 24 hours. If the order is rejected, the business transaction is considered concluded.

C4: After receiving an invoice, the Buyer has the obligation to pay the due amount within 7 days.

C5: Cancellation of a Purchase Order by the Buyer eliminates all obligations imposed on the Seller and the Buyer and concludes the business transaction. If a payment had been received before a cancellation, it will be refunded.

C6: Once payment is received, the Seller has the obligation to ship the goods within 7 days. The shipment of goods will conclude the business transaction.

C7: If the payment fails for technical or business reasons, the Buyer's deadline to respond to the invoice is extended by seven days, but the Seller gains the right to cancel the Purchase Order.

C8: The buyer and the Seller are obliged to stop the execution of the business transaction upon the detection of three consecutive failures to execute a given business operation. Possible disputes shall be sorted offline.

We assume that interaction between partners takes place through a well defined set of primitive *business operations* $B = \{bo_1,\dots, bo_n\}$ such as *purchase order submission*, *invoice notification*, and so on; each operation typically involves the transfer of one or two business documents. A bo_i is supported by a *business conversation*: a well defined message interaction protocol with stringent message timing and validity constraints (normally, a business message is accepted for processing only if it is timely and satisfies specific syntactic and semantic validity constraints). RosettaNet Partner Interface Processes and ebXML industry standards serve as good examples of such conversations.

We assume that the execution of a bo_i generates an *initiation outcome* event, one from the set {*InitSucc, InitFail*}, and if the initiation succeeds (the event is *InitSucc*), an execution outcome event, one from the set $\{$*Success, BizFail, TecFail*$\}$. These are the events (together with their attributes described subsequently) that are sent to the CCC. The rationale is as follows.

B2B messaging is typically implemented using Message oriented Middleware (MoM) that permits loose coupling between partners (e.g., the partners need not be online at the same time), we assume what follows. To guarantee that a bo_i conversation protocol is started only when both business partners are ready for the execution of a business operation, they execute an initiation protocol; the actual conversation protocol is executed if initiation succeeds. We then assume that an initiation protocol for the execution of a bo_i generates an *initiation outcome* event from the set $\{$*InitSucc, InitFail*$\}$ respectively for initiation success or failure. Following ebXML specification [4], we assume that once a conversation is started, it always completes to produce an *execution outcome* event from the set $\{$*Success, BizFail, TecFail*$\}$ which represent respectively a successful conclusion, a business failure or a technical failure. *BizFail* and *TecFail* events model the (hopefully rare) execution outcomes when, after a successful initiation, a party is unable to reach the normal end of a conversation due to exceptional situations. *TecFail* models protocol related failures detected at the middleware level, such as a late, syntactically incorrect or missing message. *BizFail* models semantic errors in a message detected at the business level, e.g., the goods-delivery address extracted from the business document is invalid. For additional details, see [6, 7], that also describes the details of synchronization required to ensure that the above events are mutually agreed outcome events between the partners.

The contract stipulates how and when rights, obligations and prohibitions are granted or revoked to business partners. We call *ROP sets* the sets of rights, obligations and prohibitions currently in force for a participant. A business operation is said to be *contract compliant* if it matches the ROP set of the participant that executes it, while also matching the constraints set by the contract clauses for its execution. A $bo_i \in B$ is said to be *out of context* if it does not. *Unknown* business operations, not present in B, are taken to be non-contract compliant. The task of the CCC during the execution of a contract consists in verifying that the operations executed by the participants are contract compliant by matching them with their ROP sets and verifying their contractual constraints, and in modifying those ROP sets as specified by the contract clauses.

2.1 The EROP Language

The EROP language describes business contracts with ECA rules that explicitly manipulate the partners' ROP sets, which are then used to monitor contract compliance. This section will present a brief tutorial for the language.

We use the keywords ***roleplayer***, ***businessoperation*** and ***compoblig*** as follows. ***Roleplayer*** declares a list of role players involved in the contract; for example, ***roleplayer*** *buyer, seller* declares the two role players of our example.

businessoperation declares a list of known business operations, for example, ***businessoperation*** *PurchaseOrderSubmission, InvoicePayment*.

A composite obligation is a tuple of obligations with a single deadline, to be executed OR–exclusively to satisfy the composite obligation. We use ***compoblig*** to specify and name composite obligations; for example, the composite obligation from clause

C2 of our contract example that stipulates that upon receiving a purchase order, a seller is obliged to either accept or refuse it, can be specified as ***compoblig*** *RespondTo-Order(POAcceptance, PORejection)*, where *RespondToOrder* is the name of the composite obligation.

Structure of Rules and Trigger Blocks A rule follows the syntax ***rule*** *ruleName* ***when*** *triggerBlock* ***then*** *actionBlock* ***end***. The expression *triggerBlock* contains an event match and a list (possibly empty) of conditions; a rule is *relevant* only when the event match and the conditions are satisfied. The event match takes the form *e* ***matches*** *(field operator value [, field operator value]*)*, where e is a placeholder for the event object being currently processed, and *field* is any of *botype* (the business operation type), *outcome* (the outcome of the operation), *originator* and *responder* (the role players initiating and responding to the operation), and *timestamp*. An *operator* is a boolean comparison operator: ==, !=, <, >, and so on. A *value* is a legitimate constant expression for that comparison.

Conditions are Boolean expressions that restrict the cases where a rule triggers. They verify the compliance of a business operation with a participant's ROP set and can also evaluate historical queries. Historical queries search for events in the historical log that match certain constraint, and can be *boolean* or *numeric*, respectively if they verify their presence or if they count the number of occurrences. Boolean queries take the form **happened**(*businessOp, originator, responder, outcome, timeConstraint*), where "*" can be used as a wildcard. Numeric queries have the ***counthappened*** keyword in place of ***happened***.

The compliance of a business operation with the ROP set of a participant can be tested with *businessOperation* ***in*** *roleplayer*. The keyword ***in*** can also be employed to test the presence of composite obligations in a participant's obligation set.

The Action Block The *actionBlock* is a list of actions where each action is $+=$, $-=$, *pass* or *terminate*. Actions $+=$ and $-=$ respectively add and remove business operations or composite obligations from the ROP sets; *pass* has no effect, while *terminate* concludes the execution of a contract. The use of $+=$ and $-=$ to add or remove rights, prohibitions and obligations (simple or composite) is demonstrated in the following statements:

```
roleplayer.rights += BusinessOper(expiry);      roleplayer.rights -= BusinessOper;
roleplayer.prohibs += BusinessOper(expiry);     roleplayer.prohibs -= BusinessOper;
roleplayer.obligs += BusinessOper(expiry);      roleplayer.obligs -= BusinessOper;
roleplayer.obligs += Obligation(expiry);        roleplayer.obligs -= Obligation;
```

$expiry$ is a deadline constraint imposed on a role player to honour his contractual right, obligations and prohibitions; the absence of a deadline indicates a duration up until the contract terminates. Notice that obligations with no deadlines are pointless as their fulfillment cannot be verified.

Conditional statements can also appear in the *actionBlock* of a rule, using the syntax ***if*** *conditions* ***then*** *actionBlock [**else** actionBlock]* ***endif***. Conditions of *if*-statements are the same ones used in a trigger block.

In an *actionBlock* the *status guards* ***Success, InitFail, BizFail, TecFail, Otherwise*** can be used to group actions for conditional execution according to the outcome of a business operation, with ***Otherwise*** used as a catchall case.

2.2 Language Example

In order to showcase the EROP language, we will present in this Section some significative rules of the EROP version of the sample contract given earlier. First of all, the declaration section:

```
roleplayer buyer, seller;
businessoperation POSubmission, Invoice, Payment, POCancellation, Refund;
businessoperation GoodsDelivery, POAcceptance, PORejection;
compoblig RespondToPO (POAcceptance, PORejection);
```

Here follow the rules derived from clauses C3, C4, C6 and C7 of the sample contract in Section 2. Note that in general the mapping between rules and clauses is N to N; in some cases, several clauses are mapped into a single rule, while in others many rules derive from a single clause. In the simplest case the mapping is one to one.

Rules R3, R4 and R8 presented below could also be written using status guards in the action block and removing the constraint on the outcome from the event matches. Both forms are equivalent, and choosing one over the other comes down to style preferences. Rule 3 below derives from clause C3, while Rule 4 derives from clause C4.

```
rule "R3"
  when
   e matches (botype == "POAcceptance",
    outcome == "Success"
    originator == "seller",
    responder == "buyer")
   RespondToPO in seller.obligs
  then
    seller.obligs -= RespondToPO;
    seller.obligs += Invoice("24h");
end
```

```
rule "R4"
  when
   e matches (botype == "Invoice",
    outcome == "Success",
    originator == "seller",
    responder == "buyer")
   Invoice in seller.obligs
  then
   seller.obligs -= Invoice;
   buyer.obligs += Payment("7d");
end
```

Rule 6 derives from clauses C6 and C7, while Rule 8 derives from clause C8.

```
rule "R6"
  when e matches (botype == "Payment",
    originator == "buyer",
    responder == "seller")
   Payment in buyer.obligs
  then
   Success:
    buyer.obligs -= Payment;
    seller.obligs += GoodsDelivery("7d");
   TecFail:
   BizFail:
    buyer.obligs -= Payment;
    buyer.obligs += Payment("7d");
    seller.rights += POCancellation();
   Otherwise:
    pass;
end
```

```
rule "R8"
  when
    e matches (botype == "Payment,
      "originator == "buyer",
      responder == "seller")
    e.outcome != "Success"
    counthappened("Payment", "buyer",
      "seller", "InitFail", "*")
     + counthappened("Payment", "buyer",
      "seller", "TecFail", "*")
     + counthappened("Payment", "buyer",
      "seller", "BizFail", "*") >= 3
  then
    terminate("TecFail");
end
```

3 Architecture of the CCC

The events supplied by the business partners to the CCC (shown in Fig. 1) carry information on undertaken business operations: the outcome, one of *InitSucc* or *InitFail*

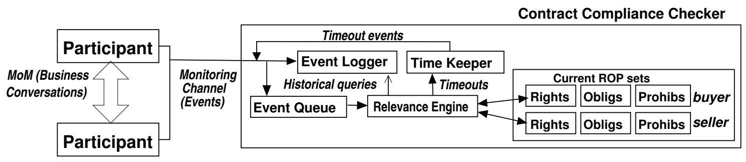

Fig. 1. The Contract Compliance Checker.

for initiation outcomes, and, if initiation succeeds, one from the set $\{$*Success, BizFail*, *TecFail*$\}$, the operation's originator and responder, and a timestamp. Events are forwarded to the **Event Logger**, that keeps a history of the business interaction as seen by the CCC, to be queried when evaluating rules with historical constraints. The **Event Queue** holds all events awaiting to be processed. The current **ROP sets** are the sets of rights, obligations and prohibitions currently assigned to the role players (to the buyer and seller in our example). The **Time Keeper** keeps track of the deadlines of rights, obligations and prohibitions. When a timeout expires (e.g. obligation deadline expiration), the Time Keeper generates a *timeout event* and forwards it to the Event Logger and the Event Queue. The **Relevance Engine** analyses queued events and triggers any relevant rules among those it holds in its working memory, following this algorithm:

1. Fetch the first event e from the Event Queue;
2. Identify the relevant rules for e;
3. For each relevant rule r, execute the actions listed in its right hand side, either ROP set manipulation or termination of a contract instance.

4 Implementation of the CCC

Figure 2 presents a diagram of the implementation of the CCC. Its main components were identified in Section 3 as the Event Queue, the Time Keeper, the Event Logger and the Relevance Engine. The Event Queue, defined in the class *EventQueue*, is a First In, First Out queue of incoming Event objects, owned by the Relevance Engine. The Event Queue offers two operations: adding an Event to the end of the queue, and taking an Event out of the head of the queue. Events are added by the participants (simulated in our prototype), and by the Time Keeper (timeout events). Only the Relevance Engine removes Events from the Event Queue.

The Time Keeper, defined in the class *TimeKeeper*, manages the deadlines for the expiry of ROP Entities, and offers two operations: adding and removing a deadline. Deadlines are internally represented using Java Timers, stored in a hash table indexed by the name and type of the ROP Entity they refer to, and the involved role players. Whenever a deadline expires, its corresponding Java Timer notifies the Time Keeper, passing as parameters the relevant data - Business Operation type, relevant Role Players, and so on. The Time Keeper then instantiates a new Event of the relevant type, appending *Timeout* to the name. The outcome of the new Event object is set to *timeout*.

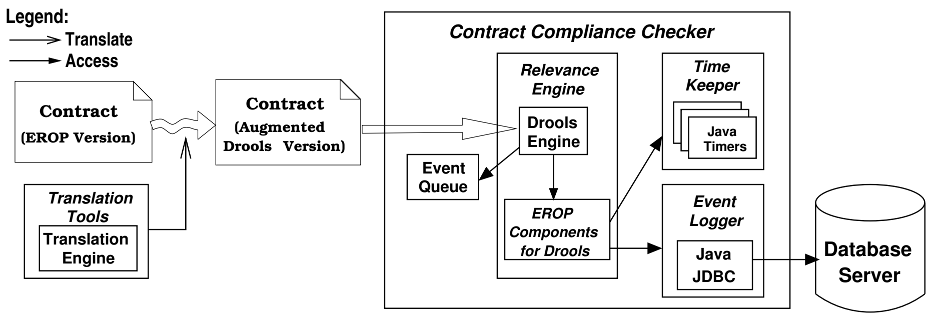

Fig. 2. Implementation Details for the Contract Compliance Checker

The Event Logger maintains the historical database and offers three operations: logging events in the database, submitting boolean queries and submitting numerical queries. The Relevance Engine (RE) relies on an instance of the Drools rule engine [3] to power its decision making capability. It offers four operations: adding an Event for processing, initializing a contract instantiation to start a new business interaction, processing the Event queue and verifying that the Event queue is empty. As anticipated earlier on, the RE's algorithm presented in Section 3 to trigger relevant rules is implemented using the recognize-act cycle of the Drools engine: the RE inserts the events retrieved from the Event Queue in Drools' working memory, to trigger a recognize-act cycle to identify any relevant rule and executes their right-hand-side actions.

4.1 Implementation of the Relevance Engine

Drools powers the decision making capability of the Relevance Engine. A rule engine is a software system that uses a set of rules to define and direct its own activity, instead of relying on static, hardcoded knowledge like a conventional system. Knowledge is therefore separated from the rest of the execution environment, and segregated in a *rule base*, or *knowledge base*, so as to be altered by users when needed without having to alter the execution environment. Drools is a *forward chaining* [8] rule engine, where *facts*, items of knowledge that are atomic from the perspective of the system, are brought in and stored for evaluation in the *working memory*, a buffer area separated from the rule base. Every time the working memory is altered by adding, removing or modifying facts, the rule engine starts a *recognise-act cycle*, examining all rules to find those for which the left hand side conditions match the current state of the working memory (*triggered* rules). The actions in the right hand side of these rules are then executed, and the facts that triggered any rules are removed from the working memory. This generally alters the working memory, so the recognise-act cycle is restarted, until no rule is triggered.

Drools also allows the definition of *globals*, objects that reside in a special area of the working memory and persist between recognise-act cycles, not triggering new ones even if they are altered. Globals usually act as hooks to external services, and are therefore the only channel to the outside world that a running Drools system has. To implement our system, we have a global for a reference to the running RE, used for housekeeping purposes, and one for a reference to the Event Logger, used to provide access to the historical log. There also is a global for each Role Player and their ROP

Sets. Events in the Event Queue awaiting processing are inserted one by one in Drools' working memory to start a recognise-act cycle, which implements the rule matching and triggering algorithm described in Section 3.

The reason for choosing a rule engine to power the Relevance Engine is the small semantic gap between EROP rules and business rules; EROP rules are fundamentally business rules that make use of the EROP ontology introduced in Section 4.2. This makes the translation process from EROP to Drools relatively straightforward, as shown in Section 5.

The reasons for choosing Drools as the particular rule engine in our system are its availability with an Open Source license, and a number of useful features, notably its use of Forgy's Rete algorithm [9], a relatively efficient algorithm to search the knowledge base for relevant rules, which is the most computationally intensive task in a rule engine. Another notable feature is the possibility to write the right hand sides of rules directly in a programming language (specifically Java, Python or Groovy). This last feature allows a more direct, simpler mapping to the implementation of the EROP ontology.

4.2 The EROP Ontology

The *EROP ontology* is a set of the concepts and relationships within the domain of B2B interaction employed to model the execution of business operations between partners, to reason about the compliance of their actions with their stated objectives in their agreements. The EROP ontology includes the following classes:

Role Player: an agent, not necessarily human, employed by one of the interacting parties, that takes on and plays a role defined in the contract.

Business Operation: a specific activity defined in the contract for the purpose of producing value.

Right: A Business Operation that a Role Player is allowed to execute.

Obligation (Simple): A Business Operation that a Role Player must execute.

Prohibition: A Business Operation that a Role Player must not execute.

Composite Obligation: A set of Obligations with a single deadline; a Role Player must execute exactly one of the set to satisfy the Composite Obligation.

ROP Entity: A right, obligation or prohibition.

Deadline: A time constraint that can be imposed on rights, and prohibitions and is always imposed on both simple and composite obligations.

ROP Set: A set (possibly empty) of rights, obligations and prohibitions belonging to a Role Player. Each Role Player has exactly one ROP Set.

Event: A message carrying a record about the occurrence of a business activity, such as initiating or concluding a business operation, the expiry of a deadline, and so on.

The classes of the EROP ontology are implemented by the Java classes *RolePlayer, BusinessOperation, Right, Obligation, Prohibition, CompositeObligation, ROPEntity, Deadline, ROPSet* and *Event*. The class *ROPEntity* is the parent of classes *Right, Obligation* and *Prohibition*, and the ancestor of *CompositeObligation*, as shown in the UML diagram presented in Figure 3.

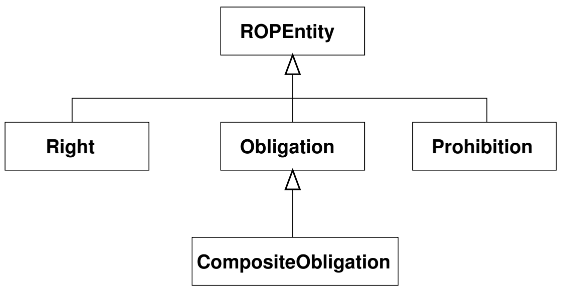

Fig. 3. Descendants of the class ROPEntity

The remaining classes, *Event, BusinessOperation, RolePlayer* and *ROPSet*, do not belong to an inheritance hierarchy.

4.3 The Historical Database

The Historical Database contains four tables: one for the Role Players, one for the relevant Event types, one for the possible status outcomes, and one for the Event history proper. The first three tables remain unaltered by the CCC for all its lifetime, and are supposed to be prepared in advance by an ancillary application. The fourth table, the actual Event history, is created empty before the first run of the system, and is filled during the contract's lifetime. Whether it has to be emptied between successive runs of the same contract depends on the conditions in the contract itself; it makes sense to allow for the possibility of writing clauses that refer to past iterations of the same contract to alter the ROP sets of the participants, e.g., a clause providing a 10% discount to buyers with at least three successfully completed purchase orders that were paid on time. Therefore there is no special provision to erase the the Event history, in order to leave the choice to do this to the involved parties.

As explained in Section 2, historical queries can be classified into two main categories: *boolean* queries, verifying whether an Event matching a given set of constraints is logged in the historical database, and *numeric* queries that count the number of occurrences of such logged Events.

In either case, the set of constraints is the same: the business operation type, the originating and responding Role Players, the Event's outcome, and a temporal constraint. An example of an acceptable set of historical constraints would be *originator = "buyer", responder = "seller", type = "PurchaseOrder", status = "Success", timeConstraint = "timestamp < '15/12/2009 10:00:00'"*. The given set of constraints is used to build a SQL query, and the answer of the database server is then analysed to generate the appropriate response. Rule R8, presented in Section 2 includes a numeric query; this maps to the method *countHappened()* of the class *EventLogger* after translation. Similarly, a boolean query would map to *happened()*. These methods build the SQL statements for the historical database from the received parameters, submit them to the database server, then parse the results and return them. So the numerical query in R8 is translated into the SQL statement SELECT COUNT(*) FROM *eventhistory* WHERE *type*='Payment' AND *originator*='buyer' AND *responder*='seller' AND *timestamp* >= '1 Jan 2008' AND *outcome*='success'. The result of the query, as per the SQL standard, is the number of rows in the *eventhistory* table that record events within the desired constraints.

5 Translation to Drools

The Java implementation of the EROP ontology presented earlier extends the rule language offered by Drools to reason about contract compliance; we call this extended language *Augmented Drools* (AD). Because of its origin, Augmented Drools is more verbose and Java-like than the EROP language, as well as less abstract and human-readable. It also needs to have additional code for housekeeping purposes, necessary to manipulate the EROP ontology, such as lines to instantiate and assign objects and arrays.

The EROP language maps completely into Augmented Drools; it is possible to write contracts in AD with the same expressive power of EROP, but, as mentioned above, the resulting code is more implementation-aware yet less declarative in style and less readable. Most importantly, however, AD can run on available software - the Drools rule engine. The problem of creating a compiler for the EROP language therefore reduces to the translation of EROP to AD. Such a translator has not yet been implemented, however it is seen to be a straightforward task that we are planning to complete in the future. The rest of this section will show how EROP statements map into AD statements.

5.1 Declarations in Augmented Drools and EROP

A rule file in Augmented Drools starts, like in EROP, with the declaration of the objects and entities used in the file: Role Players, Business Operations and Composite Obligations, together with be declarations of the Role Players' ROP Sets, and of the currently running Relevance Engine and Event Logger for reference in the rules. The definition of global indentifiers is done with the **global** Drools keyword, followed by the class of the object to declare, its name and a semicolon. As an example, here is the part of a sample contract where identifiers are declared:

```
global RelevanceEngine engine;              global BusinessOperation purchaseOrder;
global EventLogger logger;                  global BusinessOperation finePayment;
global RolePlayer buyer;                    global BusinessOperation payment;
global RolePlayer seller;                   global BusinessOperation poAcceptance;
global ROPSet ropBuyer;                     global BusinessOperation poRejection;
global ROPSet ropSeller;                    global BusinessOperation goodsDelivery;
```

Here we declare the instances of the Relevance Engine and Event Logger to use, two Role Players, *buyer* and *seller*, their two ROP Sets, and the Business operations used in subsequent rules. Business Operation names begin in lowercase here as they are Java objects and follow Java style rules; *po* stands for *Purchase Order*.

The syntax to define rules is the same in Drools and EROP, as the second is derived from the first: ***rule*** *RuleName* ***when*** *triggerBlock* ***then*** *actionBlock* ***end***. Rule names must be unique within a rule file. Comments in AD, like in Drools, are preceded by a hash sign (#), and continue until the end of the line.

Event matching, done in EROP with the syntax *e* ***matches*** *(attribute == value, [attribute == value]*)* (see Section 2), translates to the AD syntax *$e: Event (attribute == value, [attribute == value]*)*, where $*e* is an event placeholder variable.

Other conditions outside the event match are written using the Drools construct *eval*, that evaluates boolean expressions in the left hand side of rules. Therefore, historical queries of the form *happened(businessOperation, originator, responder, status, timeConstraint)* would map to ***eval*** *(eventLogger.happened (businessOperation, originator, responder, status, timeConstraint))*, where *eventLogger* is the running instance of the class *EventLogger*. Numerical queries would similarly translate in a similar fashion to calls to the *countHappened()* method using *eval*.

The test for the presence of a ROP Entity in a Role Player's ROP Set, expressed in EROP with *ROPEntity* ***in*** *rolePlayer.rop* where *rop* is one of *rights, obligs* or *prohibs*, maps to AD as a method call of the class *ROPSet*; ***eval****(playersROPSet.matchesRights (BOType))* for rights, and so on. *Eval* is here used again to evaluate a boolean method call.

5.2 Actions in EROP and Augmented Drools

ROP sets are manipulated in EROP with the += and -= operators, e.g. *seller.obligs += Invoice("24h")*. This maps in AD to a method call of the class *ROPSet*, such as *ropSeller.addObligation("Invoice", "24h")*.

The EROP keyword *terminate* maps to the AD statement *engine.terminate()*, where *engine* is the current instance of the RE; the argument of *terminate* is passed to this method. Executing *engine.terminate()* concludes the current contract instance and notifies its participants of the termination and of its outcome.

5.3 Conditional structures

In general, EROP rules using *if-then-else* statements or status guards generally map to more than one rule in AD. So an EROP rule with status guards maps to as many AD rules as the number of guards used in it; each of those AD rules will have a constraint on the outcome of the event under scrutiny added to its trigger block matching the corresponding status guard. So a rule of the form

```
rule "RuleForManyOutcomes"
  when e matches (botype == SomeBO)
  then
    Success:
      actionBlock1
    TecFail:
      actionBlock2
    Otherwise:
      actionBlock3
end
```

would be mapped to the following rules:

```
rule "RuleForSuccess"
 when
  e matches (botype == SomeBO,
   outcome == "success")
 then actionBlock1
end
```

```
rule "RuleForTechnicalFail"
 when
  e matches (botype == SomeBO,
   outcome == "tecfail")
 then actionBlock2
end
```

```
    rule "RuleForOther"
     when e matches (botype == SomeBO)
       ((e.outcome != "success")||(e.outcome != "tecfail"))
      then actionBlock3
    end
```

A rule with an *if-then-else* statement would be similarly mapped to two AD rules, one with the *then*-block and one with the *else*-block.

5.4 Examples of Translation to Augmented Drools

To offer a translation example from EROP to Augmented Drools, we will now show how rules R3 and R8 from the sample contract fragment discussed in Section 2 are mapped to Augmented Drools.

```
rule "R3"
  when
    $e: Event(type=="POAcceptance", originator=="seller",
      responder=="buyer", outcome=="Success")
    eval(ropSeller.matchesObligations("RespondToPO"));
  then
    ropSeller.removeObligation("ReactToPO");
    ropBuyer.addObligation(payment, "24h");
end

rule "R8"
  when
    $e:  Event (type=="Payment", originator=="buyer",
      responder=="seller", outcome=="success")
    eval(ropBuyer.matchesObligations(payment))
    eval(eventLogger.countHappened("Payment", "buyer", "*", "InitFail", "*")
      +eventLogger.countHappened("Payment", "buyer", "*", "TecFail", "*")
      +eventLogger.countHappened("Payment", "buyer", "*", "BizFail", "*")
      >= 3 )
  then
    engine.terminate("BizFail");
end
```

5.5 Performance Considerations

The CCC depends on the Drools rule engine to perform the most computationally intensive task, the selection of the relevant rules for incoming events. This is accomplished by the recognize-act cycle of the rule engine, using the Rete algorithm presented in [9]. The performance of the Rete algorithm depends on the number of facts in the working memory and in the characteristics of the rule base; in our system, only one fact is evaluated at a time, and so the number of facts in the working memory is not an issue. Performance is therefore determined by the characteristics of the rule base; specifically, by its size, and by how much overlap there is between the conditions on the left hand sides of rules. In general, the time needed for a recognize-act cycle grows as the size of a rule base grows; the effects of the size of a rule base on performance are discussed in [10].

The Rete algorithm uses a dataflow network to represent the left hand side conditions of the rules. Rules with common conditions share nodes in this network; the more conditions are shared, the more nodes are shared, and the more efficient a recognize-act cycle is. In our system rules are written taking a contract in natural language as a starting point. While it is reasonable to expect a certain amount of overlap between rule conditions (e.g., all rules about operations initiated by a given role player are going to share a condition asserting that role player's identity as the initiator), our experiments did not readily suggest criteria to predict the exact amount of overlap. Much depends on the definition of business operations and on writing style; equivalent contracts can be written with strongly diverging rule bases. Future work on EROP will include an investigation on the best practices of contract writing in order to achieve more efficient dataflow networks for the Rete algorithm.

Our experiments showed, however, that the code for the CCC only adds a very small constant factor to the time needed for recognize-act cycles, and so its impact on efficiency can be neglected. The overall time needed to process an event remained of the order of magnitude of milliseconds; considering that time scales in business relationships are of the order of magnitude of hours, days, or even longer, efficiency does not appear to be a limiting factor for our system.

6 Related Work

The implementation of languages for specification and monitoring of electronic contracts is an active research topic; however formal treatments and abstract models have received greater attention. In [11], a mediating entity, the Synchronization Point (SP), has a similar role to our CCC, hosting a knowledge base of contract clauses, consulted whenever the participants send an event at the conclusion of a business operation. The knowledge base is written as ECA rules using Protege [12], and interrogated using its query language PAL. The authors describe a method to generate ECA rules from an abstract model of a contract; however, the semantic distance between the model and the business rules in a natural language contract appears to be greater than the distance between our EROP rules and natural language rules.

Heimdahl [13] is another ECA-based work comparable to ours. It employs a policy monitor similar to our CCC to decide which actions are legal, and to enforce the contractual clauses. Enforcement involves asserting the presence of certain events in the future if certain events occur in the present; the monitor executes compensatory actions if the expected future events do not occur. Heimdahl's focus is on the monitoring and enforcing of SLA, so there is not much scope for the concepts of business operations and mechanisms for exception handling as offered by EROP.

Law-Governed Interaction [14, 15] is an early work in the implementation of an architecture for contract monitoring and enforcement. The Moses middleware presented in [14] has Controllers located between the interacting parties, receiving events and taking actions based on a knowledge base of rules; rules are stored by Law Servers and can be written in customized versions of Prolog or Java. Moses is an integrated system that requires Moses components to be installed within all the participants; this is in contrast to the CCC that has been designed to act as a third party service.

Compliance monitoring is investigated in [16] within the context of service based systems – systems composed dynamically from autonomous web services and coordinated by a composition process. A framework is proposed for the monitoring of compliance of such composite systems with a set of behavioural properties extracted from a BPEL specification of the composition process. At runtime, the events exchanged between the interacting parties are intercepted and processed similarly to what our CCC does, watching for violations of specified behaviour in a non-intrusive manner. Currently the specification of requirements to monitor are expressed in an abstract event calculus language that is not suitable for use by non technical people in a business environment.

Non intrusive monitoring for agent-based contract systems is investigated in [17], as part of the EU Contract Project [18]. In this work, a group of observers monitor the messages exchanged by the interacting agents, and the observed communications patterns are then matched with expected patterns derived from running contracts to detect violations. It is assumed that all messages relevant to the ongoing business transactions are visible and comprehensible to the observers.

Work on contract monitoring from the perspective of a model-driven approach is presented in [19]. The paper presents a metamodel level discussion on a variety of topics, including sub-contracting, simultaneous execution of several interleaving contract instances, nested executions, multiple monitoring and so on. Our work can be considered a concrete instance of some of the metamodels of the paper, that takes into consideration a number of practical issues not touched upon there, such as the treatment of deadlines, and of technical and business failures.

7 Conclusion and Future Work

In this paper we have presented the implementation of a prototype for a Contract Compliance Checker supporting contracts written in the EROP language. Our system is designed as a third party service monitoring B2B interactions for compliance. Our current design operates under the assumption that the business partners operate in good faith: they do not knowingly generate malicious events. Enhancements required to prevent abuse of the service is a topic for further investigation. Future work will also include completing a translator for EROP into Augmented Drools, an evaluation of the system in realistic settings, an investigation into the impact of contract writing style on efficiency and validation of the rule bases for consistency.

8 Acknowledgements

We would like to thank Davide Sottara of the Engineering Department of the University of Bologna for fruitful discussions on the inner workings of Drools and on its performance.

This work has been funded in part by UK Engineering and Physical Sciences Research Council (EPSRC), Platform Grant No. EP/D037743/1, "Networked Computing in Inter-organisation Settings".

References

1. M. Strano, C. Molina-Jimenez, and S. Shrivastava. A Rule-based Notation to Specify Executable Electronic Contracts. In *Proc. Int'l Symp. on Rule Representation, Interchange and*

Reasoning on The Web (RuleML-2008), volume 5321 of *LNCS*, pages 81– 88. Springer, Oct 2008.

2. C. Molina-Jimenez, S. Shrivastava, and S. Strano. Exception Handling in Electronic Contracting. In *Proc. 11th IEEE Conf. on Commerce and Enterprise Computing (CEC'09)*, pages —-, Jul 20–23rd, Vienna, Austria, 2009. IEEE Computer Society.
3. JBoss Rules. `http://www.jboss.org/drools/`, 2009.
4. ebXML: Business Process Spec. Schema Tech. Spec. v2.0.4. `http://docs.oasisopen.org/ebxml-bp/2.0.4/OS/spec/ebxmlbp-v2.0.4-Spec-os-en.pdf`, 2006.
5. Implementation Framework - Core specification, Version V02.00.01. `http://www.rosettanet.org/`, Mar 2002.
6. C. Molina-Jimenez and S. Shrivastava. Maintaining Consistency Between Loosely Coupled Services in the Presence of Timing Constraints and Validation Errors. In *Proc. 4th IEEE European Conf. on Web Services, (ECOWS'06)*, pages 148–157. IEEE CS Press, 2006.
7. C. Molina-Jimenez, S. Shrivastava, and N. Cook. Implementing Business Conversations with Consistency Guarantees Using Message-Oriented Middleware. In *Proc. 11th IEEE Int'l Conf. (EDOC'07)*, pages 51–62. IEEE CS Press, 2007.
8. A. Cawsey. Forward Chaining Systems. `http://www.macs.hw.ac.uk/~alison/ai3notes/subsection2_4_4_1.html`, 1994.
9. C.L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *IEEE Computer Society Reprint Collection*, pages 324–341, 1991.
10. D. Brant, T. Grose, B. Lofaso, and D. Miranker. Effects of Database Size on Rule System Performance: Five Case Studies. In *VLDB '91: Proceedings of the 17th International Conference on Very Large Data Bases*, pages 287–296, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.
11. O. Perrin and C. Godart. An Approach to Implement Contracts as Trusted Intermediaries. *Proc. of the 1st IEEE Int'l Workshop on Electronic Contracting*, 2004.
12. N.F. Noy, M. Crubezy, R.W. Fergerson, H. Knublauch, S.W. Tu, J. Vendetti, and M.A. Musen. Protege-2000: an Open-Source Ontology-Development and Knowledge-Acquisition Environment. In *AMIA Annual Symposium Proceedings*, page 953, 2003.
13. P. Gama, C. Ribeiro, and P. Ferreira. Heimdhal: A History-Based Policy Engine for Grids. *Proc. of the 6th IEEE Int'l Symposium on Cluster Computing and the Grid (CCGRID'06)*, pages 481–488, 2006.
14. N. H. Minsky and V. Ungureanu. Law-governed Interaction: a Coordination and Control Mechanism for Heterogeneous Distributed Systems. *ACM Trans. Softw. Eng. Methodol.*, 9(3):273–305, 2000.
15. N.H. Minsky and V. Ungureanu. Scalable Regulation of Inter-enterprise Electronic Commerce. *Electronic Commerce: 2nd Int'l Workshop*, November 2001.
16. G. Spanoudakis and K. Mahbub. Non Intrusive Monitoring of Service Based Systems. *Int'l Journal of Cooperative Information Systems*, 15(3):325–358, 2006.
17. N. Faci, S. Modgil, N. Oren, F. Meneguzzi, S. Miles, and M. Luck. Towards a Monitoring Framework for Agent-Based Contract Systems. In *CIA '08: Proc. 12th Int'l Workshop on Cooperative Information Agents XII*, pages 292–305, Berlin, Heidelberg, 2008. Springer-Verlag.
18. The IST Contract Project. `http://www.ist-contract.org/`.
19. P. F. Linington. Automating Support for E-Business Contracts. *Int'l Journal of Cooperative Information Systems*, 14(2-3):77–98, September 2005.