Contract Specification for Compliance Checking of Business Interactions

Thesis by

Massimo Strano

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

University of Newcastle upon Tyne
Newcastle upon Tyne, UK

2010
(Submitted January 20, 2010)

To Christine.

Acknowledgements

I would like to thank first of all my supervisor, Santosh Shrivastava, for his continued support and encouragement during my time in Newcastle University. His advice has been invaluable, and his patience in helping me clarify and express the ideas and plans I had in mind has been the most influential factor. To Carlos Molina-Jimenez I owe thanks for his patient commitment in reading my drafts and giving me much needed advice, and for his willingness to talk and provide feedback during many meetings with him and my supervisor.

My research has been funded in part by a PhD studentship from the UK EPSRC e-Science Pilot Project: “GOLD (Grid-based Information Models to Support the Rapid Innovation of High Value Added Chemicals)”.

I would also like to acknowledge all of my friends and colleagues in the School, first of all John Colquhoun. John and the others have always been ready with helpful advice, support and company whenever needed. I would also like to acknowledge Davide Sottara of the University of Bologna for many enlightening technical discussions.

I owe an acknowledgement of a more personal nature to my family: my parents, who have always encouraged me to pursue my ambitions, and my wife Christine, for her love and support during these years.

Abstract

In the business world, contracts are used to regulate business interactions between trading parties. When business transactions are conducted over an electronic channel, electronic forms of contracts are needed; and because of the additional capabilities of an electronic means, their function can be extended to include compliance checking for the interactions of the parties, and enforcement of contractual clauses when needed.

A contract is assumed to be a document that stipulates a list of clauses stating rights, obligations and prohibitions, and their associated constraints, that business partners are expected to honour. Compliance checking is taken to mean checking if business operations executed by business partners match with their rights, obligations and prohibitions as stipulated in the contract. We intend enforcement as making sure that business operations match the rights, obligations, and prohibitions of the parties, possibly compensating for deviations from expected behaviour.

In traditional business interactions, compliance checking and enforcement are carried out manually. With electronic business interactions, such tasks can ideally be automated. This requires a model for the process of checking contract compliance, and an electronic language for the specification of the actual contract.

The first main contribution of this thesis is such a model. The EROP model (from Events, Rights, Obligations and Prohibitions), composed of an ontology and an architecture, observes the interactions between the business partners, forms an interpretation of their outcome from a neutral perspective and checks their contractual compliance by matching executed operations with their sets of rights, obligations, and prohibitions, and reacting accordingly to them. Implementations of the EROP ontology and of an experimental prototype of the architecture are also presented.

The second main contribution of this thesis is the EROP language, designed to specify contractual compliance, and to regulate execution of business operations through the manipulation of the sets of rights, obligations and prohibitions of the business partners. The EROP language is rule-based and event-driven, and, in a similar fashion to contracts in natural language, contractual clauses are expressed as business rules, conditional statements associating events and conditions to lists of actions altering the rights, obligations and prohibitions of the participants. The practicality of the approach taken with the EROP language is evaluated presenting a larger, complete scenario and a number of smaller ones taken from comparable work. Notes on the translation of the EROP language to one on a lower level of abstraction that relies on the implementation of the EROP ontology are also presented. The Appendix presents a formal grammar for the language.

Declaration

All work contained within this thesis represents the original contribution of the author; any collaborative work has been explicitly acknowledged. Much of the material in this thesis has been published in conference proceedings or in journal articles as listed below. The material in Chapter 3 has been published in 2 and 4, and the material in Chapter 4 has been published in 1 and 3. Part of the material in Chapter 5 has been published in 1, while the material in Chapter 6 has been published in 3.

1. M. Strano, C. Molina-Jimenez, and S. Shrivastava, "A Rule-based Notation to Specify Executable Electronic Contracts", in *Proc. In'l Symp. RuleML 2008 (RuleML'08).* Springer, LNCS vol. 5321, 2008, pp. 81-88.

2. C. Molina-Jimenez, S. Shrivastava, and M. Strano, "A Model for Checking Contractual Compliance of Business Interactions". Under review for journal publication.

3. M. Strano, C. Molina-Jimenez, and S. Shrivastava, "Implementing a Rule-Based Contract Compliance Checker". *9th IFIP Conference on e-Business, e-Services, and e-Society (I3E'2009).* Nancy, France, Sep 2009.

4. C. Molina-Jimenez, S. Shrivastava, and M. Strano, "Exception Handling in Electronic Contracting". *Proc. 11th IEEE Conf. on Commerce and Enterprise Computing (CEC'09).* IEEE Computer Society, Vienna, Jul 2009.

Contents

Chapter 1

Introduction

1.1 Contracts in the Business World

> Society is indeed a contract. . . It is a partnership in all science; a partnership in all art; a partnership in every virtue, and in all perfection. - *Edmund Burke, British statesman and philosopher (1729-1797)* [1]

Contracts, and the partnerships that contracts define, are indeed a vital component for the smooth operation of human society, and have been so since ancient times. Even very primitive societies practiced the principle of *do ut des* – "I give (to you) so that you will give (me)" – expressing the fundamental idea of a fair exchange between two willing parties. The Latin aphorism *pacta sunt servanda* ("agreements must be kept") is considered even today the basic principle of contract law [2].

Today contracts are everywhere. Most of them are simple agreements we take so much for granted that we do not bother to put them in writing, such as buying a newspaper or treating ourselves to a cup of coffee; others have a stronger impact on our life and are written and notarized, such as employment contracts, which dictate the day-to-day activities we execute in exchange for the means of our subsistence, or contracts signed to purchase a house. As the quote at the beginning of this page implies, our society is composed of an intricate web of contracts. All economic production and exchange processes are organized through contracts; contracts are the instruments and the means for the organization of exchange relations [3].

Formally, a contract has been defined as an agreement voluntarily entered by both parties [4] that creates and defines the obligations between them [5]. Contracts are *binding*: the promises exchanged can be *enforced* in a court of law if they are breached [6]. Recourse to arbitration of the court of law is undesirable, as it can be time consuming and expensive, and it could damage the reputation of the involved parties. For these reasons, in common business practice enterprises aim to increase mutual trust and reduce disputes, attempting to solve privately those that arise. This still requires significant resources; specialized personnel needs to be hired to write contracts, verify them for correctness and consistency, negotiate their approval with the other party, mediate the business interactions between the parties and finally handle dispute resolution, including dealing with the court of law as the last resort.

The advent of globalization and electronic commerce has increased the number of contracts an enterprise could potentially participate in; however, many companies find keeping up with the requirements of the enlarged electronic market an unbearable strain to keep up with. *Electronic contracting* (e-contracting) aims at the automation of contract establishment and execution to reduce costs and time constraints to a more reasonable level; this improved efficiency can also open the way for new opportunities and developments, such as micro-contracting [7] and micro-transactions [8].

To make automation as complete as possible, electronic contracts should be as free from ambiguities as possible. In traditional business practice before the rise of e-business and e-contracting, ambiguities could be solved by human agents, often with only negligible impact on the time scale for contract execution. This is not the case with modern e-business: appealing for human clarification could be an unaffordable luxury. Therefore an electronic contract has to make explicit knowledge that would traditionally be kept implicit, or assumed to be clear from the context. An e-contract will have to explicitly declare the agents, or *role players*, participating in the transaction, and the *business operations* that can be executed within the context of the contract; it will also have to specify both *functional* and *non-functional requirements* of the participants. Functional requirements refer to the terms and conditions expressing what business operations the role players are permitted, obliged and prohibited to execute; they also stipulate when and in what order the operations can be executed. For example, in a partnership for the sale of goods (an example of which will be discussed in Section 1.2 of this Chapter), the contract will stipulate when it is possible for a buyer to submit a purchase order, and within how many days of a payment goods have to be delivered. Non-functional requirements, stated in terms of Service Level Agreements, are mainly concerned with the Quality of the Service. For example, a non-functional requirement might specify that an offered service is expected to have a repsonse time below 25 milliseconds during business hours.

The work presented in this thesis focuses on functional requirements. These can be expressed in an electronic contract by explicitly enumerating the *rights, obligations* and *prohibitions* of the role players – informally, the business operations that role players are allowed to execute, those that role players must execute (or face being sanctioned), and those that role players must not execute (or face being sanctioned); and the *clauses*, statements defining how the rights, prohibitions and obligations of the role players change as a consequence of their exercising rights, satisfying or violating obligations and violating prohibitions, and how and when the execution of a contract terminates.

Once a contract has been expressed in a fashion suitable for electronic usage, to achieve complete automation of contract execution there should be some way to automate *compliance checking.* A business operation can be said to be *contract compliant* if its execution took place in accordance with the the role players' rights, obligations and prohibitions as stipulated in the contractual clauses. In traditional business practice, contract compliance is intuitively verified by the human agents involved in the contract execution. For the same reasons of time and efficiency discussed above about solving ambiguities, an automated system cannot rely on a human's feedback on whether a certain business operation does or does not comply with the contract. Such an automated system will therefore need to be able to derive compliance information from an electronic contract, and to evaluate autonomously during the execution of a contract whether the interaction follows what has been established (i.e., if it is compliant) or not.

During a the execution of a contract, an automated system has to be able to handle *exceptional situations* that affect the normal execution of business operations. These *exceptions* can be classified into two categories: *business failures*, caused by issues at the business level (e.g., the address for goods delivery is invalid, or a payment has been rejected by a bank) and *technical failures*, caused by issues with the communication infrastructure (e.g., lost or incorrect messages). Similarly to what was discussed earlier on for compliance checking, this would be the domain of human agents in traditional business making; to automate exception handling, an automated system has to be provided with explicit, unambiguous guidelines dictating how to deal with them, up to and including when the situation is to be considered not recoverable. In this case, the execution of the contract should be terminated to solve the issue causing the exception offline.

1.2 A Contract Scenario

Let us consider this sample contract between two enterprises, called simply *Buyer* and *Seller*, for the purpose of selling some unspecified goods. This contract sample will be referred to throughout this work. In the text, **C** will stand for "clause", and rights, obligations and prohibitions will be explicitly highlighted. It is worth clarifying that this contract is not meant to be a complete and legal agreement as it is; a practical contract would have additional clauses detailing other aspects of the interaction.

- **C1**: The Buyer has the right to submit a purchase order, as long as it is from Monday to Friday and from 9am to 5pm (Right).

- **C2**: The Seller has the obligation to either accept or reject a purchase order within 24 hours (Obligation). Failure to satisfy this obligation will abort the business transaction for an offline resolution.

- **C3**: If the purchase order is accepted, the Seller is obliged to submit an invoice within 24 hours (Obligation), or the business transaction will be aborted for an offline resolution. If the purchase order is rejected, the transaction is considered concluded.

- **C4**: After receiving an invoice, the Buyer is obliged to respond within seven days, either paying the due amount or cancelling the purchase order (Obligation). Failure to satisfy this obligation will abort the business transaction for an offline resolution. Cancelling the purchase order is prohibited to the Buyer in any other condition (Prohibition).

- **C5**: Cancellation of a purchase order by the Buyer eliminates all obligations on both parties and concludes the business transaction. If a payment has been received before a cancellation, the Seller has the obligation to completely refund it (Obligation).

- **C6**: Once the payment is received, the Seller is obliged to ship the ordered goods within seven days (Obligation). A successful delivery will conclude the business transaction. Failure to satisfy this obligation will abort the business transaction for an offline resolution.

- **C7**: If a payment fails for technical or business related 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 (Right). In any other condition, the Seller is prohibited to cancel the purchase order after accepting it (Prohibition).

- **C8**: Buyer and Seller are obliged to stop the execution of the business transaction upon the detection of three failures to execute a payment (Obligation). Possible disputes shall be resolved offline.

This contract sample defines two role players, the *Buyer* and the *Seller*, and the following business operations: *Purchase Order Submission, Purchase Order Rejection, Purchase Order Acceptance, Invoice Submission, Payment, Goods Delivery* and *Refund*. The clauses define how the rights, obligations and prohibitions for the role players change during the course of the interaction.

A successful execution of this contract will see a Buyer's purchase order followed by the Seller's acceptance and invoice, then by the Buyer's payment, followed by goods delivery. A Seller's rejection of a purchase order, or a Buyer's cancellation of a purchase order after an invoice are also considered to conclude the transaction successfully, in the sense that there are no pending disputes to resolve offline.

Cancellation of a purchase order is normally a prohibited operation, bringing the end of a transaction for offline resolution. Cancellation is allowed only in certain circumstances: by the Buyer when invoiced, in case the asked price is considered unacceptable (**C4**), and by the Seller in the case a payment operation fails because of business or technical failures (**C7**). In the scenario presented here, clause **C7** deals with technical and business failures by granting an extension of the payment deadline, for the purpose of allowing the parties to locate and solve the issues that caused a payment to fail. This optimistic measure has been added to reduce the chances of having to appeal to the court of law. To avoid the abuse of this, two provisions have been made: first of all, in case the payment deadline is extended after an exception occurs, the Seller gains the right to cancel the purchase order, which was previously prohibited. In this way deadline extensions will continue only as long as the Seller is willing to go ahead with them. Secondly, in case three failures occur, clause **C8** dictates that the transaction has to stop; this assures both parties that neither will have the power to indefinitely extend the payment deadline by maliciously simulating a failure.

1.3 Overview of This Work

This thesis is dedicated to the presentation of EROP, a system for the specification of electronic contracts and the monitoring of compliance of business interaction. EROP stands for *Events, Rights*, *Obligations* and *Prohibitions*; as will be shown in the following chapters, it is based on the premise of observing events generated by the role players' execution of business operations, and monitoring contract compliance by matching those events against the rights, obligations and prohibitions of the role players.

The material presented in this work is organized as follows. Chapter 2 presents background information that is relevant for the subsequent chapters. We present a discussion of the lifecycle of a business partnership and of the research areas necessary for the creation of a system for compliance checking, and a list of desirable features for contract representation languages. We also describe the most significative and influential of the works in the current state of the art within the context of a framework derived from the preceding discussion. Chapter 3 presents the EROP model. This includes the *EROP ontology*, a set of concepts used to model B2B interaction; an execution model for business operations; a discussion on the representation of contractual clauses as rules; and finally the *Contract Compliance Checker*, an event-driven architecture [9] to monitor for compliance of business interactions with the contract stipulated between the participants.

Chapter 4 introduces the EROP language, an event-driven rule-based language for electronic contracts. A primer is given for the syntax and usage of the language, followed by a discussion of the EROP representation of the sample contract presented in Section 1.2. Chapter 5 evaluates the EROP language, discussing the specification of contracts taken from a selection of scenarios we thought worthy of attention, and an evaluation of the EROP language against the list of desirable features for contract representation languages presented in Chapter 2.

Chapter 6 presents the implementation necessary to run a complete EROP system, including the EROP ontology and the Contract Compliance Checker; it then discusses a prototype implementation that has been undertaken, and how the EROP language translates to the rule language used internally by the Contract Compliance Checker. Chapter 7 presents our concluding remarks and describes areas of interest for future work. The Appendix presents a formal grammar for the EROP language written in the input format for ANTLR [82], a well known parser generator.

Chapter 2

Related Work

This chapter will present a review of relevant literature and technologies, within the context of a framing description of the fundamental concepts of electronic business and of its necessary infrastructure. The review will be concluded with a discussion locating within the same framework the contributions of this work, and comparing them with those of the works reviewed here.

2.1 Stages of the Lifecycle of a Business Partnership

Five stages can be identified in the lifecycle of a business partnership. The first four of them always occur, and the successful completion of each of them is necessary for the execution of the following stage. The last stage is not mandatory, but, as will be shown below, is needed to attempt a recovery from disputes that might have occurred earlier on.

- **Discovery**: In this stage, one or more enterprises advertise their availability to enter into a business partnership on suitable private or public communication channels. Some other enterprise finds these advertisements of availability on a channel and establishes contact, using the same channel or a different one. An exchange of messages then ensues to verify the availability of all involved parties, and to set up the next stage. In traditional business practice the Discovery stage is informal: advertising and communications are conducted over a variety of channels, from telephones to print to the Internet, but they all have in common an unstructured approach that is difficult to automate completely and systematically.

- **Negotiation**: In this stage, two or more enterprises enter into a negotiation over an agreed upon private channel on the terms and conditions of their partnership, with the intent of producing a contract. This stage can be abstracted as an exchange of documents containing draft versions of a contract, that the parties modify and send back for evaluation, until a version is found that meets the approval of all parties. In traditional business practice the Negotiation stage usually happens through real-life meetings of human agents, who discuss proposed contractual clauses within the context of competent legal jurisdiction until reaching an agreement.

- **Contract Recording**: In this stage the contract agreed upon in the Negotiation stage is deposited with a trusted third party, to serve as a reference for possible future disputes, in order to avoid accusations of malicious modifications of the contract text. In traditional business practice of many legislations, this involves a notary, a human agent recognized by the local government, and tasked with the authentication of legal documents.
- **Contract Execution**: In this stage the business partners interact to achieve the targets they set for the business partnership, following the patterns laid down in the contract. In traditional business practice, human agents of the parties run each and every step of all interactions. Execution of business operations is flexible and informal, and the agents have a certain amount of leeway; consider for example a scenario where a seller offers a discount larger than what is declared in a contract in order to entince a buyer into submitting a larger purchase order.
- **Dispute Resolution**: In this stage an attempt is made to resolve any disputes that have arisen during Contract Execution that could not be resolved within the boundaries established by the contract. All involved parties will refer to the notarized version of the contract and to any existing records of the business activities that went on during contract execution, and try to negotiate a compromise between their position, sometimes using the services of a trusted third party to arbitrate the process. A successful resolution will avoid a court case, and ideally could lead back to the Contract Execution stage to continue with the business interaction. In traditional business practice, this stage is just as flexible and informal as the Contract Execution stage, and the agents can attempt an impromptu resolution diplomatically, sometimes even stretching the contract beyond its intended meaning – for example, consider a seller trying to appease a buyer who is complaining about a late goods delivery; the seller's agent might offer a promise of discounts on future purchases to appease the buyer and avoid going to court.

2.2 Electronic Contracts for Business Partnership

The stages of the lifecycle of a business partnership described in the previous section have evolved within the context of traditional business practice: regardless of the communication channel employed, the presence of humans is mandated by the great number of decisions to be taken at every step, many of them requiring not only knowledge of the contractual clauses, but also of the relevant local laws and of the policies and traditions of all involved parties, as well as the capability of making autonomous decisions in a short time frame. The intervention of skilled humans is therefore necessary to write, interpret and verify contracts, and subsequently to monitor interactions for compliance and enforce contractual terms. The use of electronic contracts can bring several advantages, such as an accelerated contractual lifecycle [10], brought up by the automation of as many of the activities listed above as possible, with as little human intervention as possible. Bringing business interactions to an electronic form requires work in three areas:

- **Contract Representation Language**: A language for the electronic representation of the knowledge that would be held by human agents in traditional business practice. That includes specification of contractual clauses, internal policies of the participants, relevant laws, involved role players, and so on.
- **Compliance Monitoring**: A system to monitor the participants' behaviour for compliance, possibly to enforce contractual clauses if and when they are violated, and to record the history of the transaction.
- **Theoretical Model**: A model underpinning electronic business, including an ontology of relevant classes, work on formal verification of electronic contracts, and so on.

There is a significative interest in the research world in electronic business, both from the academic and the business environment, taking a variety of different approaches and focusing on different aspects. Many current technologies give different levels of attention on each of the three areas listed above; there is also a strong measure of cross pollination between different authors. Furthermore, some works are relevant even if their interest in automation of business interaction is tangential. In the particular case of works dealing with or presenting contract representation languages, these will be evaluated against a list of desirable features presented next in Section 2.2.1, to provide a conceptual framework for their discussion.

2.2.1 Desirable Language Features

Electronic contracts should be used on automated systems, and so should run with little or no human maintenance. They must therefore be made free from the ambiguities that are so frequently present in conventional contracts written in a natural language, and that need to be resolved by humans as the need arises. With this view in mind, a list of desirable features can be compiled for contract representation languages:

Declarative approach: A language should follow a declarative, high level style, concentrating as much as possible on "what" rather than "how". For example, it should be possible to express what the consequences are for a successful submission of a purchase order, without having to delve into the details of the conversation that occurs between the involved parties.

It can be argued that the use of mechanisms to support inheritance in a language for electronic contracts is part of a declarative approach. The possibility to extend incrementally contracts by specializing their vocabulary, in a parallel with inheritance in computer programming, can be argued to help encouraging reuse of existing work (code in the case of a program; clauses in the case of a contract). This could simplify the work of those that write contracts, and allow for a conceptual organization that is tidier and easier to maintain and debug.

Implementability: We can define a language as *implementable* if it is possible to efficiently and effectively implement it. A side effect of this definition is that contracts written in an implementable language should always provide programmers with useful information about how to implement a given functionality using current technology. An implementable language should involve only measurable parameters and observable conditions, and should not require interpretation to generate executable code. For example, expressions like "Customer's storage usage should not be high" or "Seller must eventually receive payment" are not implementable, while "Customer's storage usage should be below 2 Gbytes", and "Seller must receive payment within three days" are acceptable.

Expressiveness: A language should have enough expressive power to specify typical contractual clauses found in most practical applications, and be able to describe both functional and non-functional requirements. Functional requirements are related to high level business operations executed between the interacting parties and involve exchange of one or more business documents (examples are purchase order submissions, invoice notifications and so on). Non-functional requirements are central in service provision contracts (also known as service agreements) that specify the expected Quality of Service from the provider and the expected behaviour from the consumer.

Exception handling: It should be possible to specify how to deal with the consequences of exceptional situations that arise because of the inherently distributed nature of the underlying computations. Firstly, there should be easy ways to specify how to deal with any software and/or hardware related problems encountered during business interactions (e.g., unpredictable transmission delays, message loss, corrupted messages, semantically invalid messages, node failures, timeouts, etc.). Secondly, because B2B interactions typically take place between partners that are loosely coupled and in a peer-to-peer relationship, those partners can sometimes get out of synchrony and perform erroneous, even mutually conflicting operations; an example is a buyer cancelling an order after it has been paid, or, worse, shipped by the seller. A language for the specification of electronic contracts should provide easy ways to specify how to deal with such conflicting situations. Exception handling mechanisms have been studied intensively in the field of fault-tolerant systems. [11] discusses the need for exception resolution when exceptions can be raised concurrently by different cooperating threads in an application. A common feature of much work in electronic contracting is their focus on the logical aspects of business interaction, without taking into account either the impact of timing, or the fault tolerant and concurrency issues we address in this work. An exception is [12], where the authors highlight the complexity of handling exceptional situations, such as the cancellation of a purchase order, due to infrastructure or human related events; they argue that to be effective, a compensation mechanism should take into account the state of the two interacting applications.

Usability: A language should be easy to use for the humans that write the contracts: it should be relatively intuitive and fast to learn, and it should take less time to accomplish particular tasks using it. While the notion of usability being desirable is intuitively acceptable, verifying whether a language is useable is difficult, as there is no tried and ready method to formally determine it. What is possible to do, however, is to have a survey among people working in environments that could adopt the language, collect their feedback and explore its implications.

Verifiability: A specification should be amenable to formal verification, possibly to a fully automated analysis, although a semi-automatic one might be acceptable. Even when electronic contracts are written by skilled humans, there is still a significant chance of introducing inconsistencies and loopholes, especially in the case of an existing contract that is being amended. Verification is a complex topic in its own right, and plenty of work has been produced in this area.

In this Section we presented three simple conceptual frameworks for the discussion and classification of relevant works; the first, presented in Section 2.1, is focused on the stages of the lifecycle of a business partnership; the second, presented in Section 2.2, is focused on the research areas that work on electronic business contracting covers; the third, presented in Section 2.3, is focused on desirable features in languages for representation of business contracts. The rest of this chapter will be dedicated to this discussion, and to an evaluation of the work presented here in comparison with these.

2.3 The 4W Framework

The 4W framework, presented and discussed in [13, 14], has been created to reason about the requirements of the participants in business-to-business relationships. This framework aims to identify the concepts that appear in an electronic contract and in the environment it is defined for, as well as to describe the relations between these concepts. According to the authors, 4W can help to identify the aspects of a contract that can be improved and the new opportunities that can be introduced to the participants.

The authors' first step is to define an ontology for the e-contracting domain, and to populate it with concepts they start from the concept of contract. The definition used is the following: a contract is a legally enforceable agreement in which two or more parties commit to certain obligations in return for certain rights. From this definition, they extract these groups of concepts:

- The *Who* concepts that model the *actors* participating in the negotiation and the execution of a contract.

- The *Where* concepts that model the *context* of the contract. This can mean the jurisdiction chosen to handle disputes that cannot be resolved peacefully, but also geographical aspects and especially business context - it is often the case in business practice that one contract depends on the existence and successful execution of another [15].
- The *What* concepts that model the *exchanged values* and their exchange - the rights and obligations that are assigned and withdrawn during a contractual interaction.
- The *HoW* concepts that model the *means and processes* to negotiate and execute a contract.

The name of "4W" for this framework arises from the four Ws in *Who, Where, What* and *HoW*. The authors' second step is to identify the relationships between these four group of concepts, and between the groups and the concept of contract. The actors (*Who* group) are involved in the negotiation and execution of a contract, supplement the contract context (*Where* group) and are assigned rights and obligations (*What* group) during the execution stage. The contract context (*Where* group) affects the means and processes (*How* group). The rights and obligations manipulated during the execution (*What* group) are concerned with the means and processes (*How* group). The actors (*Who* group) are assigned and use rights and obligations(*What* group). Finally, the contract is related to all of these groups, as they are defined in it.

Finally, after a discussion on the legal implications of the use of electronic contracting, the authors use the 4W framework to analyze and discuss two research works, CrossFlow [16] and ebXML [17].

2.4 Complex Event Processing and Rapide

Complex Event Processing (CEP) is presented in [18]. CEP is a set of tools and techniques to analyze and control event-driven information systems. The techniques and tools presented are not primarily aimed at contract representation, but have more general application in enterprise systems. CEP is based on the concept of *event*, a record of an activity in a system, and of their aggregation in *complex events*, to signify an activity that takes place over a time interval, and that is the result of a sequence of smaller, simpler activities. The author presents a model for the construction of enterprise systems that process streams of complex events to detect certain event patterns and react accordingly.

The instrument to specify event patterns and their processing rules is **Rapide**, a rule language – that is, a language where the most important statements are *rules*, constructs of the form *Event – Condition – Action* (ECA). Rapide has a very rich syntax, allowing for the definition of extremely sophisticated rules. Rules are used to describe the behaviour of *Event Processing Agents* (EPAs), objects that monitor the execution of certain types of events to detect certain given patterns, and react by executing certain given actions. The execution of these actions will generate more events, that can be monitored by other agents. In this way, EPAs can be connected to form *Event Processing Networks*. Rapide provides constructs to specify how events flow between the EPAs of a network, such as *filters*, which use event patterns to filter out relevant events, and *maps*, which aggregate event tuples according to given patterns.

Another relevant aspect of Rapide and of the model presented in [18] is its treatment of causality relationships between events. Rapide allows the construction of *causal maps*, specifying the causal connections between event types. These allow the use of a *causal operator* in the rules for the EPAs, so that it is possible to write rule conditions that impose a given causal relationship between events. For example, in our Buyer-Seller scenario, one could conceive of a rule to process a cancellation of a purchase order that is executed if and only if the cancellation has been caused by a previous business failure in a payment operation. It is possible to write such rules in Rapide, increasing the flexibility and the power of the language.

Within the context of the relevant research areas presented in Section 2.2, [18] offers a model, the CEP model, and a language, Rapide, but no monitoring and no enforcing architecture. The focus is on the Contract Execution phase of the business partnership lifecycle, with the proviso that one does not “write a contract” as such with Rapide, but defines a network of rule-powered agents to process events. As a language, Rapide takes a strongly declarative approach, at a high level of abstraction. It is very powerful and flexible, and the examples supplied in [18] provide strong proof of its expressiveness. Exception handling is less of a concern compared to other proprieties; there is not much attention spared in the book for how a system based on Rapide could handle the failures and problems that could easily arise in a distributed environment; however, it is conceivable to use the constructs of Rapide to simulate a limited handling of exceptions that could partially solve this.

As for usability, the great expressive power and flexibility of Rapide come at the price of great complexity. Non-technical people looking to write an electronic representation of a business contract would have a hard time familiarizing themselves with the syntax and semantics of the language; the same goes for the generic and comprehensive CEP model underpinning Rapide. Furthermore, there is no significant mention of methods for formal verification, despite the complexity of the agent networks that it is possible to specify with Rapide. Because of these shortcomings, Rapide appears more suited for use as an abstract language to describe implementation-independent business rules.

2.5 The Synchronization Point Model

In [19, 20] the authors present an approach for the management of shared business processes within a *Virtual Enterprise* (VE), a kind of organization created ad hoc to support a partnership for a specific collaboration. Virtual Enterprises are usually temporary, but their life does not have to be brief, or bound to a limited number of repetitions of the same interaction. During their lifecycle, VEs offer a virtual "private space" where the participants can share the resources and information they want to commit for the project in a controlled fashion.

Such an arrangement needs architectural support: the VE participants could have heterogeneous internal organizations and different policies. The authors propose in [20] a model based on the presence of *Synchronization Points* (SPs) in the VE. Synchronization Points are mediating entities, run in collaboration by the participants, that monitor the interactions between the participants. These go through *process services*, abstract representations of the public business processes that the participants expose for access in the VE. The SPs monitor the events generated by the participants' access of the process services and coordinate data flow, control flow and transaction flow in reaction to these events, all according to the contracts agreed upon by participants.

The model employed for contract access and contract representation is presented in [19]. Before the initiation of the VE, contracts are translated as knowledge bases, written in the form of ECA rules using Protege [21], an environment for the development of ontologies and knowledge bases based on those ontologies; interrogation is executed using Protege's query language PAL.

The ECA rules are generated from the natural language contracts passing through an intermediate step, the generation of *SP relationships* and *Sat functions*. Each SP relationship assert a given link between two objects of two participants. For every SP relationship, there is a set of associated Sat functions (Sat stands for satisfiability) that takes as input one or more objects or events, and returns *true* if and only if the SP relationships hold between the input objects or events.

The two works, [20] and [19], present a well-developed model, more directly focused on contract representation and monitoring than the more general CEP model of [18]; like this one, the focus of the Synchronization Point model is on the Contract Execution step. An architecture for monitoring business interactions is also presented, and a short presentation of its implementation is given. No contract representation language is presented as such; contracts are represented as knowledge bases for Protege, which allows for a very high level, declarative writing style, but could be challenging for non-technical personnel, reducing its usability. While it is possible to write rules to handle exceptional situations, they must be foreseen and planned for; unforeseen failures, such as technical ones, cannot easily be reasoned about.

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 relatively great. It is not clear whether it is possible to generate automatically the Sat functions, it appears that one or more stages will have to be done by humans. This reduces the implementability of the model. As for formal methods of verification, they are not mentioned, but such an effort appears to be a reasonable endeavour.

2.6 Law-Governed Interaction and Moses

In [22] the author gives an overview of his own and his collaborators' work of many years. This consists of the Law Governed Interaction (LGI) model, the Moses middleware to run the LGI architecture and two *law-languages* to specify the interaction laws that govern a distributed group of software agents. The basic concepts of LGI were presented first in [23], and then developed further in many papers such as [24].

Law-Governed Interaction is described in [22] as a decentralized mechanism for the coordination and control of distributed systems. In LGI, the interactions of a group of software agents are governed by an explicitly specified policy, the *interaction law* (or simply the *law*) of this group.

The law is enforced in a decentralized manner, associating a *controller* component with each agent. All the messages the agent sends and receives have to pass through the controller, that is trusted to mediate all the interactions of its associated agent by observing the sent and received messages, and by interpreting the law at hand to decide how to react. The decision is made taking into consideration the local state of the interaction. The controller enforces the law by preventing its violation, rather than reacting to it. This means that prohibited exchanges of messages are guaranteed not to happen, and that actions mandated by the law are guaranteed to happen. Controllers can be instantiated and run by the participants to the interaction, or by trusted third parties that are willing to authenticate them and vouch for their trustworthiness.

Moses is the middleware that supports LGI. It is implemented as a Java package, and it provides the implementation of a number of tools needed to support LGI. Among these are the *Controller Pool*, the *Controller Manager*, and the *Law Server*. The *Controller Pool* is the process that operates the controllers, providing secure communication channels between each controller and its associated agent, and between controllers to mediate for their agents. Controllers can be created directly by the participants, or by the *Controller Manager*, a server that creates and maintains controllers on the Controller Pool on behalf of the participants. The Controller Manager is typically run by a trusted third party. One or more *Law Servers* are used to publish and retrieve laws. Controllers retrieve laws from a Law Server when they are created, or whenever the current laws are altered. A Law Server can serve many controllers, or alternatively agents could have their own private servers to maintain laws they are using. This allows a flexible adoption of policies for the participants. Moses includes other tools: a *Law Tester* to help testing laws, various security tools to deal with the minutiae of secure communication, and interfaces to interact with controllers and Law Servers.

LGI supports two law-languages, built by extending Java and Prolog. The authors supplemented Java and Prolog respectively with libraries of classes and methods that implement the ontology of the LGI model, so that laws are effectively legitimate programs in those languages, and inherit their writing style, as well as their advantages and their limitations. The Prolog law-language is more declarative and more compact, and it is possible to write laws directly in ECA format. The Java law-language is significantly more efficient (the author reports that typical average evaluation times are about 40 times shorter with Java), and easier to debug; and as Java is familiar to more people, the Java law-language should be relatively more useable.

Both law-languages have some common aspects, being implementations of the same ontology. As they are both implemented as the extension of actual programming languages, they are both implementable according to the definition presented in Section 2.2.1. They are both sufficiently expressive, as shown by the examples shown in [22] and in the numerous examples in the Moses documentation. Support is available to handle exceptions of a technical nature, such as communication issues, but not for issues with the semantic interpretation of the messages within a running business transaction. The Law Tester server allows for a limited form of verification, providing syntactic verification of laws and experimental testing of laws by submitting simulated events. There is no apparent support for consistency and correctness tests, nor for testing the presence of loopholes and infinite loops.

2.7 Heimdahl

Heimdahl was presented in [25] and [26]. In these works the authors present a middleware platform for the monitoring and enforcement of obligation policies, Heimdahl itself, and xSPL, a language for the definition of these policies. Enforcement involves asserting the presence of certain events in the future if certain events occur in the present; Heimdahl itself executes compensatory actions if the expected future events do not occur. For example, in our Buyer-Seller scenario the execution of a purchase order by a buyer asserts the presence of a payment by the buyer in the future; if this does not occur, the monitor takes a compensatory action, such as fining the buyer.

Heimdahl constantly monitors the interactions of the participants, intercepting their communications, but never stopping them; all actions are always authorized. Intercepted communications are composed into events; these events are evaluated against a rule base of policies to determine whether any applies; if that is the case, the ones that do apply can impose or fulfill obligations, or impose compensations. Events that cause the imposition or the fulfillment of obligations and compensations are logged for future reference.

The xSPL language is used to write the obligation policies Heimdahl employs. This language is an extension of the Security Policy Language (SPL) presented in [27]. Statements in xSPL are rules in Event-Condition-Action form. It is possible to write rules with conditions that refer to events occurred in the past (as recorded by Heimdahl's history log), and with actions that assert the occurrence in the future of given events with given attributes, which is the way obligations are implemented in xSPL. To handle the case where the obliged events do not happen, rules can also include the specification of what actions Heimdahl should take to compensate.

The xSPL language is declarative, but the syntax to handle the assertion of future events is slightly quirky and could be difficult to grasp for non-technical people, reducing usability. The language is implementable, as it is not permitted to write non-observable conditions, but its expressiveness is limited, as xSPL is more suited to express non-functional requirements than functional ones. It is not possible to forbid the execution of actions that could have negative consequences, or that are not allowed at a given time of the business partnership. Exception handling also is not supported directly. All these features could be simulated in a limited fashion with the use of longer, complex constructs, that however would make xSPL less useable. While it is possible to conceive formal verification methods for this language, this topic has not been considered by the authors in the works presented here.

2.8 Business Contract Language and Business Contract Architecture

In [28] the authors present a model to express the organizational structures within a business partnership in terms of *communities*, groups of objects working together to achieve common goals, similar to the Virtual Enterprises of [20]. *Rights* and *obligations* are modelled and treated as objects in their own right, and so the problem of constraining and directing the actions of objects in the system is transformed in the problem of creating and manipulating right and obligation objects. A contract in this model specifies what actions the objects in a community can be involved with, and how rights and obligations are manipulated in response to these actions. The authors also present a testing implementation for a checking mechanism that verifies that the actions executed within the community follow what is dictated in the contract. The architecture relies on a *contract repository* to supply access to the contract in force, and on a *monitor* that uses the knowledge stored in the repository to check on incoming events and report on their correctness.

The model presented in [28] is expanded in [29], and used to underpin Business Contract Language (BCL). *Prohibitions* are introduced to model actions that should not be executed by any object in the community. BCL itself is an event-driven language, with event patterns inspired by [18]. Contracts can be written using two syntacting forms: an XML-based one used internally, cumbersome to read and to maintain, and a more conventional, human-readable one for design purposes. The two notations are equivalent in all aspects, and a translator exists to convert documents written in the first into the second notation. Policies are statements that assert what events are generated for given occurrences of event patterns and conditions.

In [30] the author presents an adaptation of the techniques of Model Driven Development (MDD) to the generation of the software systems needed to monitor the execution of electronic contracts. In a model driven approach, of which the OMG's specific version is presented in [31], the system designers create an abstract model of the business using a suitable domain specific language, or metamodel; after this, they define a transformation metamodel, mapping from the business model to a target model that is derived from the running solution on which deploy the system. If the source and target metamodels are reliable and well written, then the transformations are resuable. The author then presents two metamodels, the *notification metamodel*, providing a target for the generation of the calls to the infrastructure linking the participants to the monitor, and the *monitoring metamodel*, that provides a target for the mapping of the contract into a form suitable for consultation during the business interaction.

BCA (Business Contract Architecture), an architecture for the management and monitoring of electronic business contracts written in BCL, is presented in [32], [33] and [34]. The architecture presented in these works has been designed to provide support for all the phases of a partnership lifecycle as described in Section 2.1. The components of a BCA system are:

- The **Contract Repository** stores and serves various categories of contractual templates used to create contract.
- The **Notary** stores instances of contracts after they have been agreed upon and checked for validity.
- The **Legal Rules Repository** stores and serves the policies pertaining to a particular legislative domain.
- The **Contract Validator** ensures the validity contract instances, checking for *competence* (the actual capacity of the parties to execute what is agreed in the contract), *clarity* (the lack of ambiguity of contract templates), *legal purpose* (the lack of conflicts between the contract and the local policies stored in the Legal Rules Repository) and *consideration* (the lack of conflicts between the contractual template and the elements in the contract instance that describe what is exchanged).
- The **Contract Negotiator** runs the negotiation process; this is a non-mandatory component, to be used if the parties do not or cannot carry the negotiation out themselves.
- The **Contract Monitor** monitors the interactions of the parties, recording all important events and signalling any case of contract non-performance to the Contract Enforcer if one is detected.
- The **Contract Enforcer** acts to enforce the contractual terms, in one of two ways. It could act on the involved parties to ensure their behaviour conforms to the contract, or it could inform the Contract Validator, which would blacklist the responsible parties.

The body of work reviewed here presents a detailed, well developed model for community-oriented business interaction, an architecture to negotiate, manage, verify and monitor electronic contracts, and BCL, a language for the creation of business contracts. The architecture of BCA provides support for the Negotiation, Recording, Execution and Dispute Resoultion stages of a business partnership lifecycle as presented in Section 2.1. As for BCL, when analyzed in the light of the criteria laid out in Section 2.2.1, it can be said to be declarative and expressive. Implementability is guaranteed by the use of the model-driven approach described in [30]; a carefully written transformation metamodel will replace any non-implementable contractual clause in the business model into an implementable one in the target model. Usability is taken care of by having a more human friendly notation, that can be translated in a more machine friendly notation for runtime execution. Verifiability and exception handling appear possible, and, while they are not the main objects of interest of the works discussed here, hints are given throughout the work presented here about how to handle those.

2.9 Non Intrusive Monitoring and BPEL

Compliance monitoring is investigated in [35] and [36] 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 of the composed services, called *requirements*, and a set of behavioural properties of the interacting agents, the *assumptions*. Requirements and assumptions are extracted from a BPEL [37] specification of the composition process, and are expressed using *event calculus* [38], a logic based formalism to represent actions and their effects.

In [35] the event calculus expressions that specify requirements are expressed in an abstract, symbolic form suitable for research and design purposes; an XML schema is presented in [36] to allow electronic manipulations of the event calculus expressions. The framework monitors the interactions between the partners using a non-intrusive approach, by intercepting the events exchanged between the services that compose the system and its users, and the effects of these events on the state of the composite system. This monitoring is performed in parallel with the operation of the system and does not affect its operation and its performance.

The framework presented in these works is able to monitor three types of deviations from the expected behaviour (that is, from the sets of assumptions and requirements derived from the BPEL specification):

- **Inconsistencies evidenced from recorded behaviour**: these inconsistencies arise when an assumption cannot be logically derived from the set of recorded events. That implies that some of the recorded events violate that assumption.

- **Inconsistencies evidenced from the expected behaviour of the system**: these inconsistencies arise when an assumption cannot be logically derived from the set of recorded events and the set of all other assumptions. That implies that some of the behavioural properties and assumptions have not been realized.

- **Unjustified system behaviour**: these inconsistencies arise when the conditions of a requirement are satisfied by the recorded system behaviour (ie, by the set of recorded events) but violated by the expected system behaviour. That implies that one of the shared services made an assumption about that requirement which is not consistent with other assumptions given for the service it deploys.

In conclusion, the work presented by the authors covers in depth two of the three areas necessary for a complete system to monitor business interactions, discussing a detailed theoretical model and a language to describe the knowledge required for monitoring. An architecture for non-intrusive compliance monitoring is also discussed, but not a complete running implementation. The language used in this work is BPEL, which is not specifically geared for business contracts, but to specify web service interactions. Business interactions can be described by mapping the business operations offered within the partnership to the web services used to access and control those electronically. As a language, this solution follows a less declarative approach, as contract writers have to reason in terms of web services rather than business interactions, but it is implementable, as it refers to the requirements and assumptions of actual, implemented services. Expressiveness should be guaranteed as long as the mapping of business operations to the web services is complete; that is, if all operations can be mapped to a corresponding web service. Exception handling is also supported by the model's capability to detect and report the three types of inconsistencies in the behaviour of the services and of the agents; however, the model's capability to express and execute remedial actions to recover from exception appears more limited. Verifiability is not touched upon, but it appears possible using the powerful model discussed. Usability appears to be a more significant issue. Non-technical people could find the complex model and the abstract notation daunting to use for commercial purpose.

2.10 Defeasible Logic in RuleML

RuleML [39] is an XML-based generic and semantically neutral language for the representation of rules. In [40] the author presents extensions to RuleML with constructs to support rule writing that makes use of Defeasible Logic and Deontic Logic. Defeasible Logic [41] is a non-monotonic logic proposed to formalize defeasible reasoning. Deontic Logic [42] is a branch of logic that is concerned with reasoning about obligations, permissions and related concepts. The author presented in previous work the addition of deontic modalities to defeasible logic [43] and the use of deontic logic to reason about violations of obligations in [44]; this work is integrated in [40] to allow the writing of rules dealing with deontic concepts in RuleML, such as *contrary-to-duty obligations*, introduced to compensate for previous unfulfilled obligations. Real life contracts often make provisions to deal with unfulfilled obligations, and most of the time these provisions impose contrary-to-duty obligations in the attempt to recover from the breach of contract. The introduction of contrary-to-duty obligations in RuleML allows a formal representation of this practice in rule form.

The author then proceeds to show how to transform a contract written in a natural language into a RuleML rule base that can be formally verified, and then fed to a rule engine to monitor the execution of the contract at run time. It is implicitly assumed, however, that significant parts of this transformation could not be automated; we refer here to the extraction of facts, definitions and normative rules from the natural language contract.

Overall, [40] and the other work from the same author described above, [43] and [44], tie together other existing works in Deontic Logic and Defeasible Logic with a well established technology, RuleML, covering as a result all the areas indicated in Section 2.1: a model to formalize and reason about contracts, a language for the representation of contracts and, using existing RuleML engines such as the Java Rule Engine API [45], an implementation to run contracts. However, the communication model of the business partners is not touched upon, and it is therefore not clear how monitoring is actually executed. Furthermore, the extended RuleML defined in [40] does not prevent the use of non-implementable conditions, which can be reasoned about abstractly but cannot be translated automatically into instructions for a machine. Exception handling capabilities appear to be limited to the recovery from breaches of contract by means of only contrary-to-duty obligations, with no provision for recovery from technical or business-related issues. The model presented allows the language to have a very high level, declarative style. RuleML being based on XML, usability is another concern; the translation process discussed in [40] ultimately has to be executed by humans, who would then find themselves in the position of having to write directly in XML, unless provided with additional tools that are not in the scope for this work, such as graphic interfaces or human friendly languages that could map into XML. Verifiability is touched upon, but not discussed in detail.

2.11 SORM and Cremona

SORM [46] is a model for the representation and management of *promises* for the implementation and the supervision of electronic contracts. It is based on the duality of two deontic concepts that other models keep separate, rights and obligations; in SORM they are treated as the two faces of a promise. The party committing to a promise is obliged to fulfill it, while the party to whom the promise is made holds the right that the promise will be fulfilled. Three types of dual obligations/rights can be distinguished: *state obligations* and *state rights*, promises to maintain a particular state (e.g., for a Service Level Agreement), *action obligations* and *rights to have an action performed*, promises by a party to execute a given action in certain circumstances (e.g., to perform a payment), and *option obligations* and *rights to act*, promises to tolerate an action performed by another party (e.g., access a given service).

The parties own sets of obligations that are currently in force; these sets can be modified during runtime. Managing large sets of obligations one by one can be cumbersome, so the authors introduce *OR states* (where OR stands for Obligations-Rights), associated with given sets of obligations, and operations to change the state of a participant to a new OR state, transparently assigning to the participant the set of obligations associated with the new state. Not all obligations are imposed as a consequence of an OR state change; some, the *background obligations*, are in force independently of the current state, and can be imposed or removed without interfering with it. Obligations imposed or removed as a consequence of a change of OR state are *state-based obligations*. The authors conclude affirming that SORM could be used to design a formal contract language.

Cremona [47], a related work from the same research group, is an architecture for middleware to implement the WS-Agreement [48] standard. The WS-Agreement defines a protocol to negotiate agreements between service providers and service clients; a negotiation usually follows this template:

- The would-be client finds a provider for the desired service.
- The client proposes an agreement stating the desired service capacity and Quality of Service requirements.
- The provider derives its own resource requirements for the requested capacity and QoS level; this could inlude prioritization of resource allocation if enough resources are not available at the time of the client's proposal.
- Using the data generated in the previous step, the provider accepts or rejects the agreement proposed by the client.

WS-Agreement, and therefore Cremona as one of its implementations, covers the Negotiation stage of the Business Partnership Lifcycle shown in Section 2.1. If examined in the light of the list of relevant research areas of Section 2.2, it matches most closely the Contract Representation Language area (albeit specializing in the Negotiation stage); a theoretical model is given some attention. Monitoring of compliance for approved agreements is not in the scope of the specification.

2.12 EU CONTRACT Project

CONTRACT [49] is a research project funded by the European Commission, aiming to developing tools to model, build, verify and monitor multi-agent electronic business systems as dictated by electronic contracts.

In [50], the authors present an administrative architecture to support the management of electronic contracts, and more precisely the process to define the ontology to use in a business partnership given the contract underpinning it. These steps are covered:

- Off-line verification of the contracts to be enacted for consistency, and to check whether their aims are achievable given the possible states the system can reach.

- Definition and generation of the application specific processes for the administration of the contracts, such as enactment, monitoring, updating, termination, renewal and so on.

- Definition of the roles played by the agents to interact.

- Identification of the components and the services used by the agents to play their roles in the partnership.

Processed contracts are then stored in a *Contract Store* together with the information generated from their analysis.

Monitoring in particular is discussed in more detail in [51], where the authors present a framework for monitoring the behaviour of agents and detect any violations of agreed contrctual norms. The focus of their investigation is mainly on *corrective monitoring*, where violations are detected as they occur and remedied taking corrective measures, as opposed to *predictive monitoring*, where violations are predicted observing the agents' behaviour and steps are taken to completely avoid violations; however, there is a brief discussion on how to use the observation of agents' behaviour to realize predictive monitoring.

The monitoring architecture discussed in this work includes the following components:

- A set of **Observers**, objects that subscribe to the communication channels between agents and between the agents and the services, and report the intercepted messages to the Monitor.

- A **Mapper**, translating contracts obtained from a Contract Store in a form that can be used by the Monitor.

- A **Monitor**, including a *Matching Engine* that matches observed messages received from the Observers with the contract representations obtained from the Mapper, and an *Explanation Generator*, that when a violation occurs prepares a report on which contractual clauses were violated, when and why, and passes it to a *Manager* object.

The contracts fed to the Mapper are written in an XML-based language presented in [52]; the Mapper maps them to *Augmented Transition Networks* (ATNs) [53], which are the actual representation using by the Monitor. ATNs were originally developed to process natual language; they are directed labelled graphs representing contractual clauses, where nodes correspond to states of the contract, and edges are labelled with messages sent and received by the agents. The authors in [51] present a complete mapping to transform the XML representation of a contract into a set of ATNs.

The work of the CONTRACT project covers all of the three research areas presented in Section 2.2, albeit with different attention, presenting a monitoring framework and a theoretical model, and discussing work towards adapting an existing XML language. The tools and services discussed cover the Contract Recording and Contract Execution stages of a business partnership lifecycle as presented in Section 2.1. As for the contract representation language, which is the work of another research group, it can be analyzed in the light of the features described in Section 2.2.1 in this manner. The language has a relatively high level, declarative style, and it is sufficiently expressive to specify typical contractual clauses. However, its implementability is not clear, as it is possible to represent concepts that are intuitively acceptable to humans, but not unambiguously translatable in a form acceptable by machines. A limited form of exception handling appears to be possible, but it is not the main focus of the CONTRACT project, and neither is usability. Verifiability is paid careful attention, with the presentation of an administrative architecture to execute formal verification procedures.

2.13 EROP in Context

After presenting the most important of the relevant technologies, it is time to discuss the position and the claims of the work presented here, the EROP model, architecture and language, using the same discussion frameworks used to analyze the above mentioned related works.

EROP concentrates on the Execution and Dispute Resolution stages of the business partnership lifecycle; the other stages are interesting fodder for future work, but for the moment they are not the focus of investigation. All the three areas presented in Section 2.2 are covered, albeit with varying degrees of focus. In the following chapters, I will present the theoretical model underpinning EROP, the architecture for the monitoring system used for compliance checking and the language used for contract description.

The most important contribution of the EROP model is arguably the extended capability to reason about and provide for unforeseen exceptional situations. Technical exceptions, arising from infrastructure issues, and business exceptions, arising from issues connected with the business process proper, are differentiated, and syntactic constructs are defined to reason about them. This feature is an effect of the way the model and the language take into consideration the distributed nature of the underlying computation and the use of B2B messaging protocols (such as RosettaNet [54] and ebXML [17]). The lower levels of the interaction are abstracted away at the level EROP operates, leaving the contract writers to concentrate only on the outcome and the consequences of business operations.

Another important contribution is the presentation of a language for contract description that is both relatively useable for non-technical people, by using a metaphor of manipulation of rights, obligations and prohibitions for the involved participants, and relatively easy to implement using the implementation of the ontology used in our model. Chapter 6 will show how contracts written in the EROP language can be mapped to rule bases making use of our ontology.

My work was greatly influenced by Rapide and the CEP model of [18]. Rapide and EROP share the use of the Event-Condition-Action paradigm, and the use in the rule heads of event pattern matching. When designing the EROP language, I aimed for a simpler, more specialized language that could deal specifically with the manipulation of rights, obligations and prohibitions, as well as with the recognization and handling of various types of exceptional situations, so that they would be *reified*, i.e., become “first class citizens”, objects in their own right, so to speak, of the language semantics. This made EROP easier to implement, and arguably easier to learn for non technical people.

The idea of having rights, obligations and prohibitions as “first class citizens” of the EROP ontology was inspired by SORM [46] as well as BCL [29]. In comparison with SORM, EROP adds prohibitions, removes the concept of OR state, and makes the treatment of exceptional situations explicit. The reason for this is that [46] does not show the mapping between higher level manipulations of participants’ states and obligations on one side and the B2B messaging protocols used to implement the interaction.

The management of expiry dates using timers in the Synchronization Point model [20, 19] inspired the timing management solutions adopted in EROP. One of the main issues of the SP model, however, is the semantic distance between the business rules in natural language and the abstract model needed to create a rule base in Protege. EROP offers a language with a shorter semantic distance to the business rules in natural language, as it offers the intuitive concepts of rights, obligations and prohibitions.

The Law Governed Interaction (LGI) architecture of [22] offers law-languages, obtained developing an ontology for the abstract model in a host language – Java and Prolog in the case of LGI. This concept is also present in EROP; our equivalent of a law language is an existing rule language, JBoss Rules, also known as Drools, augmented with the implementation of the EROP ontology. However, this would force writing contracts in Drools, which might not be ideal for non-technical people, as was observed in our discussion of LGI. Furthermore, using LGI requires the interacting participants to integrate the LGI technology within their organizations, which could be impractical, and requires a significant level of trust between the participants. EROP relies on the services of a trusted third party, freeing the participants from the need to overhaul their infrastructure in order to engage in an interaction, cutting organizational stress and costs.

Heimdahl shares with EROP the notion of using compensation as a means to recover from violations. However, in EROP compensations are obligations, in a manner broadly similar to the approach taken in [44] with its contrary-to-duty obligations. This approach has the advantage of treating compensatory actions in the same way other actions are treated, with no need for a special syntax and a separate treatment.

As mentioned above, EROP shared with the model based on Defeasible Logic presented in [43, 44] the notion of compensating for violations by imposing new obligations, imposing a higher degree of uniformity in the language. However, the DL model has a weakness similar to the one exhibited by SORM and discussed earlier on: the lack of a mapping between higher level rules and and the B2B messaging protocols used to implement the interaction, especially when it is necessary to deal with failures at the lower level. EROP integrates these concerns into its model, so that it is possible to reason about this, and take action to recover the situation.

BCL shares with EROP the *reification* of rights, obligations and prohibitions: their definition as “first class citizens” of the ontology used to model the system and its environment. This has been used in EROP to create a contract representation language that is intuitive to grasp for non-technical people, leveraging the familiarity to business-oriented personnel with reasoning in terms of rights, obligations, prohibitions and their manipulation. A stronger connection with underlying B2B interaction protocols makes it possible in EROP to reason about exceptional situations, and to take action to remedy them.

The model for Non-Intrusive Monitoring (NMI) presented in [35] shares with the EROP model the reification of exceptional outcomes of the execution of a business operation. While NMI derives its exception types from discrepancies and inconsistencies between actually observed behaviours and expected behaviour derived from current assumptions about the state of a transaction, EROP derives its exception types from the underlying communication protocols used to execute the actual business operation. A significant difference between NMI and EROP is the one between NMI’s reliance on BPEL as its contract representation language, opposed by the simpler and more intuitive EROP language.

The CONTRACT project shares with EROP the reliance on a third party to provide compliance checking and enforcement. As also mentioned above, this choice frees the participants from the need to overhaul their infrastructure in order to engage in an interaction. The most important differences are EROP’s greater connection with the underlying B2B protocol; the language used for CONTRACT has no clearly defined notion of technical or business failure, and so it is not easily possible to reason about them. Furthermore, the EROP language guarantees implementability, and is arguably easier to learn and use productively than the more complex language supported by CONTRACT.

Chapter 3

Model

This chapter will discuss the ontology used to model contracts and the definition of contract compliance, as well as presenting the execution model for business operations and the architecture of the Contract Compliance Checker (CCC). Material from this chapter appeared in [55, 56].

3.1 Contracts and the EROP Ontology

In this Section we discuss the relationship between contracts and contract instances, and introduce the *EROP ontology*, a set of concepts within the domain of B2B interaction. The EROP ontology is employed to model the evolution of interactions between business partners, and to reason about the compliance of their actions with their stated objectives in the contract.

3.1.1 Instantiation of Contracts

Some contracts define an interaction that is only executed once and is not expected to be repeated, at least not in the short term; an example of such contracts could be an agreement to purchase a house. However, it is often the case that contracts are intended to be be executed many times; for example, the contract presented in Section 1.2 could be executed a number of times by the signing Buyer and Seller for many different purchases of goods.

The execution of an electronic contract can be seen as the execution of a particular kind of program. Electronic contracts are analogous to types or classes, which can be instantiated when needed during a business transaction. For example, the contract for the Buyer-Seller contract of Section 1.2 is instantiated anew every time the signing parties want to trade. These contract types can be parametrized creating templates that can be used in a generic fashion to allow a certain amount of dynamically typed programming, similar to template classes in C++ or generics in Java; for example, the Seller of the above mentioned Buyer-Seller contract could use a generalized version of it to sell goods to any interested customer.

Again like programs, the execution of an electronic contract requires the declaration of a set of entities for which object bindings are established at instantiation. This is the set of *roles* that can be played by agents during the execution of a contract. According to [57], a role is a set of connected behaviours, rights and obligations as conceptualized by actors in a social situation. *Buyer* and *Seller* are examples of roles in the sample contract presented in Section 1.2. Roles are played by *role players*, agents, not necessarily human, employed by the interacting parties. Different agents could play the same roles in different instantiations of a contract, and do this in the employ of different parties.

When a contract is instantiated, roles have to be bound to the agents playing them for the duration of that instantiation. This binding defines a *role player* – an agent playing a role. The scope of the binding defining a role player is restricted to that contract instance, and all references in the contract to a role will be taken to refer to the associated role player. The binding of roles to agents to define role players is part of the theory of Role Based Access Control [58], and while the work presented in this thesis relies on this topic, it is not an object of our investigation.

3.1.2 The EROP Ontology

Here is a list of the classes in the EROP ontology, illustrated using examples from the sample contract of the Buyer-Seller scenario presented in Section 1.2.

- *Contract*: An electronic document describing roles and business operations, and detailing how
- *Role*: a set of connected behaviours, rights and obligations as conceptualized by actors in a social situation.
- *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. *Buyer* and *seller* are examples of role players.
- *Business transaction*: The whole of the execution of a contract, from the moment it starts until its successful or unsuccessful conclusion.
- *Business operation*: an activity defined in the contract for the ultimate purpose of producing value, executed as a shared interaction between two role players using a B2B messaging protocol. Business operations make up the vocabulary of a business contract. *Submit Purchase Order*, *Send Invoice*, and *Cancel Purchase Order* are examples of business operations.
- *Deadline*: A time constraint on the execution of business operations, involved in defining rights, obligations and prohibitions.
- *Right*: A business operation that a role player is allowed to execute. It can have a deadline; if it does not, it is assumed to last until revoked, or until the end of the business partnership.

In our Buyer-Seller scenario, the right to submit a purchase order is a right with no deadline.

- *Rights Set*: A set of all the rights granted to a role player at a given time. Rights sets are dynamic: rights can be added and removed as the business transaction progresses.

- *Obligation (simple)*: A business operation that a role player must execute, or face the penalty of being sanctioned. It always has a deadline, because an obligation that does not expire is not enforceable – the obliged party can always claim they will satisfy it at an indeterminate future time– and therefore is meaningless. An example of an obligation from our buyer-seller scenario is the obligation to pay an invoice within seven days that is imposed on the buyer after the seller sends an invoice.

- *Composite obligation*: A set of business operations a Role Player must execute exactly one of. An example of a composite obligation from our buyer-seller scenario is the obligation to either accept or refuse a purchase order imposed on the seller after receiving such an order from the buyer.

- *Obligations Set*: A set of all the obligations, simple or composite, granted to a role player. Obligations sets are dynamic: obligations can be added and removed as the business transaction progresses.

- *Prohibition*: A business operation that a role player must not execute, or face the penalty of being sanctioned. Prohibitions are explicitly dealt with in the EROP model, rather than treated as a complement of the rights or as negative obligations. The motivation for this is the need to differentiate and reason about prohibited operations, the execution of which results in a sanction, from operations that are unexpected within the context of the contractual clauses and should not be executed by the role players at a given time. An example of a prohibition in our Buyer-Seller scenario is the one assigned at the beginning of the execution of the contract that forbids the Seller from cancelling a purchase order.

- *Prohibitions Set*: A set of all the prohibitions granted to a role player. Prohibitions sets are dynamic: prohibitions can be added and removed as the business transaction progresses.

- *ROP entity*: A right, obligation or prohibition.

- *ROP set*: A set (possibly empty) of all the rights, obligations and prohibitions in force for a Role Player at a given time. Each Role Player has exactly one of them.

- *Event*: A message carrying information about something happening within the context of the business transaction. Events in the EROP ontology are *composite* [59], arising from the execution of a business operation, which takes place over a given time interval as the result of a sequence of smaller and simpler activities defined in a B2B messaging protocol, such as ebXML [17].

The contract signed by the business partners defines, explicitly or implicitly, a set $RP = \{rp_1, \ldots, rp_m\}$ of role players and a set $B = \{bo_1, \ldots, bo_n\}$ of business operations. Any other operation not in B is said to be an *unknown operation*.

3.1.3 Formal Definitions: Deadlines, Rights, Obligations and Prohibitions

The execution of business operations is often (but not always) constrained by a deadline, an expression t that evaluates to a time, and can be *absolute* or *relative*, depending on whether it evaluates to a specific point in time (e.g., at 12:00 on 1st January 2009) or to a point in time at a given distance in the future (e.g., 24 hours and 30 minutes from now). In the EROP model, absolute deadlines can be specified using the syntax of a SQL DATETIME type, so that “at 12:00 on 1st January 2009” can be expressed as “01-01-2009 12:00:00”. Relative deadlines can be specified using the syntax $[Tc]+$, where c is a one letter code for a time unit - s for seconds, m for minutes, h for hours, d for days, w for weeks and y for years - while T is a positive integer for the amount of time units. Therefore, “24 hours and 30 minutes from now” can be expressed as “24h30m”.

A *right* is formally defined as a tuple (bo, t) where $bo \in B$ is a business operation and t is an optional deadline. If there is no deadline, it is replaced with $\emptyset$. The right to submit a purchase order in our buyer-seller scenario is expressed with (*Submit Purchase Order*, $\emptyset$), as it always remains in force throughout the execution of a contract. A business operation $bo_i \in B$ *matches* a right $r = (bo, t)$ (indicated with $bo_i \vdash r$) if and only if $bo = bo_i$.

A (simple) *obligation* is formally defined as a tuple (bo, t) where $bo \in B$ is a business operation and t is a mandatory deadline. The obligation to pay a received invoice within seven days is expressed with (*Invoice Payment*, “7d”). A business operation $bo_i \in B$ *matches* an obligation $o = (bo, t)$ (indicated with $bo_i \vdash o$) if and only if $bo = bo_i$.

A *composite obligation* is formally defined as a tuple (X, t) where $X \subseteq B$ is a set of known business operations and t is a mandatory deadline. The obligation to either accept or reject a purchase order within 24 hours is expressed with ({*Accept Purchase Order, Reject Purchase Order*}, “24h”). A business operation $bo_i \in B$ *matches* a composite obligation $o = (O \subseteq B, t)$ (indicated with $bo_i \vdash o$) if and only if $bo \in O$.

A *prohibition* is formally defined as a tuple (bo, t) where $bo \in B$ is a business operation and t is an optional deadline. If there is no deadline, it is replaced with $\emptyset$. The prohibition to cancel a purchase order in our Buyer-Seller scenario is expressed with (*Cancel Purchase Order*, $\emptyset$), as it remains in force until explicitly removed. A business operation $bo_i \in B$ *matches* a prohibition $p = (bo, t)$ (indicated with $bo_i \vdash p$) if and only if $bo = bo_i$.

3.1.4 Contract Compliance of Business Operations

A given business operation executed by a role player can be informally said to be *contract-compliant* if its execution took place in accordance with the rights, obligations and prohibitions, together with any additional constraints, stipulated in the contractual clauses. These constraints can be grouped into the following three categories.

- *The identities of initiator and responder match the role players stipulated in the contract.* Business operations must be initiated and responded only by the role players explicitly declared to be able to in the contract. For instance, clause C1 of the contract example of the previous chapter stipulates that the business operation *Purchase Order Submission* would be considered contract-compliant only if the role player *buyer* initiates the operation and the role player *seller* is its responder.

- *The time of occurrence of a business operation matches what is stipulated in the contract.* To be contract-compliant, an operation must be initiated and/or concluded at the stipulated time. For example, clause C1 of our sample contract stipulates that purchase orders initiated on Saturdays and Sundays, or on weekdays outside office time, are not contract-compliant.

- *The history of the business transaction before the operation matches what is stipulated in the contract.* Business contracts often impose as a condition for contract-compliance the validity of a set of causal relationships between operations and other events, such as other operations, timeouts and so on. For example, a contract might stipulate that a certain business operation bo_i is contract compliant if and only if it is preceded by the successful execution of another operation bo_j. In our contract example, clause C4 stipulates that the business operation *Invoice Submission* is contract-compliant if and only if it is preceded by the successful execution of *Purchase Order Submission.*

The above criteria can be used to express a formal definition of contract compliance. Let us use the notations R_{rp}, O_{rp}, P_{rp} to denote respectively the set of rights, the set of obligations and the set of prohibitions currently in force for a role player $rp \in RP$. These three sets will be collectively referred to as the *ROP sets* for the role player rp, and denoted as ROP_{rp}.

A business operation $bo_i \in B$ is said to *match* a role player's set of rights R_{rp} (or a role player's set of obligations O_{rp}, or the set of prohibitions P_{rp}) if and only if it matches a right $r \in R_{rp}$ (respectively, an obligation $o \in O_{rp}$, or a prohibition $p \in P_{rp}$). This can be denoted with the expression $bo_i \vdash R_{rp}$ (respectively, $bo_i \vdash O_{rp}$ or $bo_i \vdash P_{rp}$). A business operation is also said to match a role player's ROP set ROP_{rp} (denoted with $bo_i \vdash ROP_{rp}$) if and only if $bo_i \vdash R_{rp} \vee bo_i \vdash O_{rp} \vee bo_i \vdash P_{rp}$.

In general, the execution of a business operation *bo* initiated by a role player *rp* is said to be contract compliant if and only if these three requirements are satisfied:

1. $bo \in B$ (that is, *bo* must not be an unknown operation).

2. $bo_i \vdash ROP_{rp}$

3. The execution of *bo* must satisfy all the constraints belonging to the three groups defined above as defined in the contract.

Executions of business operations that satisfy the first two criteria but not the third are called *out of context business operations*. An example of such an operation from the Buyer-Seller scenario in Section 1.2 would be an attempt by the Seller to invoice the Buyer before a purchase order is accepted. Another more general example would be the execution of an obliged business operation after the deadline for the obligation expired. The deadline is a constraint from the second group defined above, and executing the business operation after its expiry would violate it. Therefore, its execution would be considered non-compliant.

3.2 Execution of Business Conversations

In this Section certain assumptions will be made concerning the manner the interacting business partners choose to conduct their business. Let us assume that the two partners chose to do so over the Internet under the guidance of a conventional paper-based contract. Their interaction results in the exchange of several messages transmitted over a suitable conversation channel. The conversation channel between the partners can be realised using a number of technologies, such as SOAP [60] over HTTP.

3.2.1 Execution Model

The fulfilment of intended business objectives requires the partners to exercise their rights and their obligations, and this in turn requires them to exchange messages carrying business documents and to act on them. This activity can be viewed as the business partners taking part in the execution of a *public business process*, that can also be called a *shared* or *cross-organizational* business process, where each partner is responsible for performing their assigned part. In Figure 3.1, $PubBizProc_B$ and $PubBizProc_S$ are the two halves of the public business process the buyer and the seller share for the execution of the contract discussed in the previous Chapter. To preserve their autonomy, the buyer and the seller conceal behind their public business processes those aspects of their business they do not wish to disclose, the *private business processes*; in Figure 3.1 these are *Private Business* $Process_B$ and *Private Business* $Process_S$ respectively for the buyer and the seller.

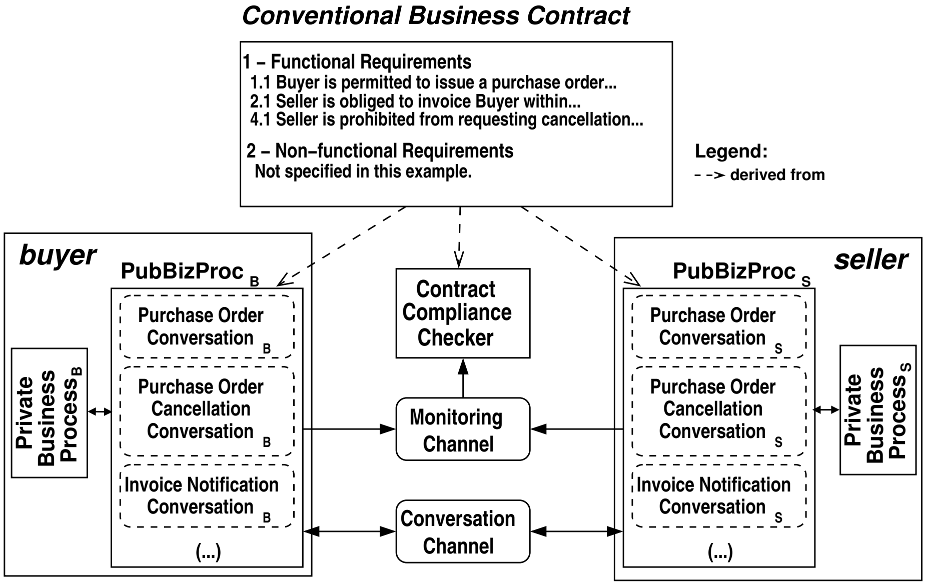
Conventional Business Contract
1 – Functional Requirements
1.1 Buyer is permitted to issue a purchase order...
2.1 Seller is obliged to invoice Buyer within...
4.1 Seller is prohibited from requesting cancellation...
2 – Non-functional Requirements
Not specified in this example.
Legend:
- -> derived from
buyer
PubBizProc B
Private Business Process B
Purchase Order Conversation B
Purchase Order Cancellation Conversation B
Invoice Notification Conversation B
(...)
Contract Compliance Checker
Monitoring Channel
Conversation Channel
seller
PubBizProc S
Purchase Order Conversation S
Purchase Order Cancellation Conversation S
Invoice Notification Conversation S
(...)
Private Business Process S

Figure 3.1: Private and Public Business Processes.

The shared business process is derived from the contract by mapping each business operation stipulated in the contractual clauses into the two complementary halves of a *business conversation*, as shown in Figure 3.1. For example, the business operation *Purchase Order Submission* from clause 1.1 is mapped into the two business conversations *Purchase Order Conversation*$_B$ and *Purchase Order Conversation*$_S$; likewise, the business operation *Invoice* from clause 2.1 is mapped into the two business conversations *Invoice Notification Conversation*$_B$ and *Invoice Notification Conversation*$_S$. Consequently, executing a business operation results in the execution of the corresponding business conversations.

Business conversations are carried out employing a B2B messaging protocol, and are subject to timing and validity requirements that can be discussed using a model inspired by RosettaNet [54, 61]. There are two kinds of these conversations, shown in Figure 3.2: single-action and double-action. In a single-action conversation a single electronic document is sent, and then its receipt is acknowledged; a double-action conversation involves an exchange of two documents, a request and a response, with each receipt acknowledged.

A document is accepted for processing by its receiver only if the document is received within the set timeout period (if applicable) and the document is valid. There are two validity checks that must be met:

1. **Base-validation**: the document must be syntactically valid; this involves verification of a static set of syntactical and data validation rules, according to the specification laid down in the standards being used;

2. **Content-validation**: a base-validated document must also be semantically valid, in the sense

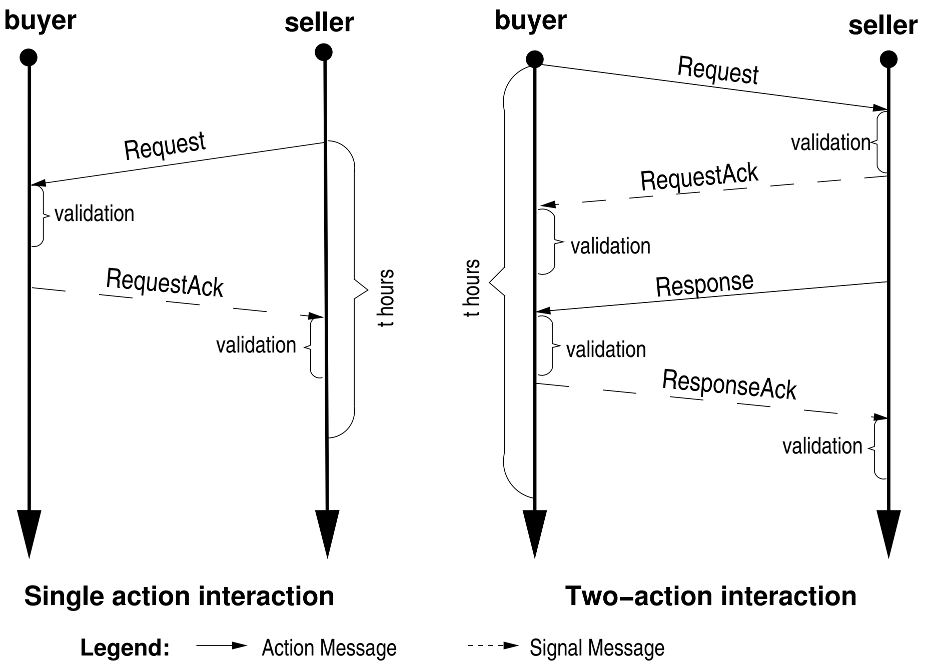

Figure 3.2: Interaction patterns.

that its contents should satisfy some given, application specific criteria. This validation could vary from trading partner to trading partner and may well be performed within the receiver's Private Business Process.

The interactions shown in Figure 3.2 depict what should happen logically; in reality, each such interaction maps onto several messages that are required in any protocol implementation (connection establishment, acknowledgements and non-acknowledgements, retries, etc.).

RosettaNet specifies about a hundred Partner Interface Processes (PIPs), that define basic business conversations such as PIP 3A4, *Request Purchase Order*, and PIP 3C3, *Notification of Invoice*. Each PIP document specifies an element of the vocabulary of the business interaction by dictating the choreography of the message dialogue. As shown in Figure 3.3, this includes *business action* messages (messages carrying a business document) and *business signal* (Acks and Nacks) messages.

In RosettaNet, a buyer is expected to use the *Request Purchase Order* PIP 3A4 to express its desire to pay. Similarly, the seller is expected to use PIP 3C3 (*Notification of Invoice*) to invoice the buyer. A graphical representation of the two PIPs is shown in Figure 3.3.

As shown in the figure, the receiver of an action message is required to acknowledge it by sending a signal message back within two hours. Although each PIP performs a conceptually simple action, in an asynchronous environment, such as the Internet, where communication and processing delays can be unpredictable, we face the problem that the PIP initiator (e.g., the seller for PIP 3C3) and its responder, could (e.g., the buyer for PIP 3C3) could end up with contradictory views of the

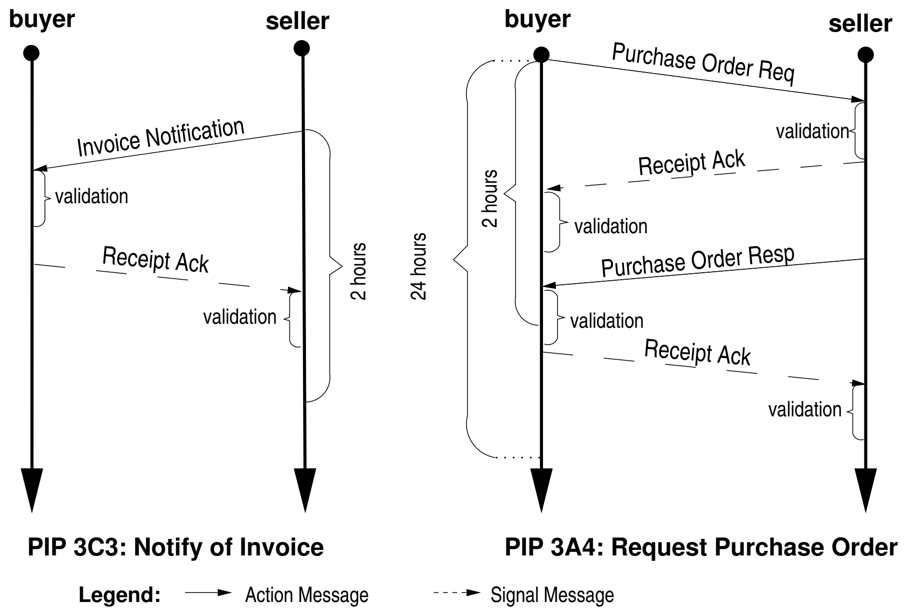

Figure 3.3: Examples of RosettaNet PIPs.

outcome of the execution of the PIP. For example, in PIP 3C3, if the *ReceiptAck* signal message is lost or arrives after the two hour limit, the buyer's and the seller's views of the outcome could be respectively a successful and a failed termination; subsequent executions of public business processes at each end could diverge, causing errors at the business level.

RosettaNet relies on negative acknowledgements to synchronize trading partners at PIP level, and minimize the errors propagated to the business application [61]. A positive acknowledgement is sent to indicate that an action message has been received and successfully base-validated. A negative acknowledgement is sent to indicate that an action message has been received but failed its base-validation. However, some errors inevitably could propagate to the business level. The impact of base and content validation is illustrated in Figure 3.4, illustrating how the states of the buyer and the seller can become mutually inconsistent when the action message is base-valid but content-invalid. When the buyer discovers the error, it signals a failure to alert the seller; for this Rosettanet provides a special *Notification of Failure* PIP 0A1 to take application specific actions to re-synchronise. At the same time, the seller is convinced that the outcome of the execution of the PIP is successful, and so starts the execution of the next PIP it would normally execute for a success. Thus the two business partners find themselves out of synchrony, and could end up performing mutually inconsistent actions before re-synchronization occurs. A way out of it is to execute an explicit synchronization of the outcomes, as discussed in the next Subsection.

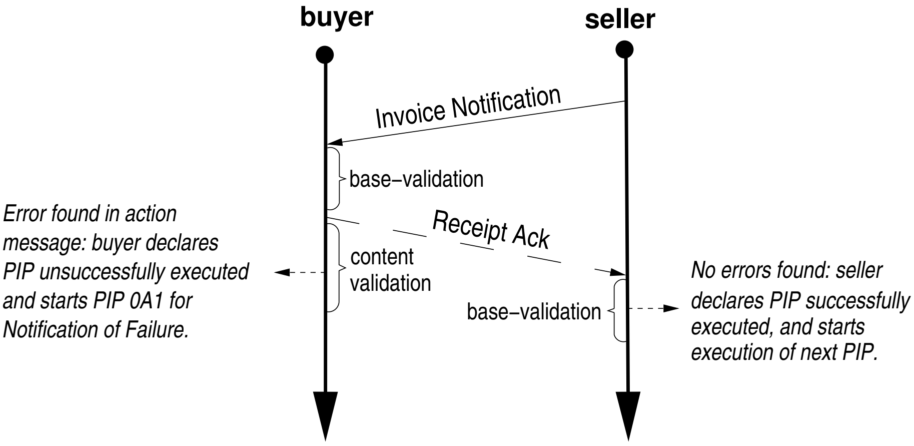
buyer
seller
Invoice Notification
base–validation
Receipt Ack
content validation
base–validation
Error found in action message: buyer declares PIP unsuccessfully executed and starts PIP 0A1 for Notification of Failure.
No errors found: seller declares PIP successfully executed, and starts execution of next PIP.

Figure 3.4: Diverging views of PIP outcome.

3.2.2 Outcome of Business Operations

Given the wide variety of events that can be generated at both sides of a business conversation (send, receive, timeout, retry, . . .), it is worthwhile examining if any aggregation can be performed to make only a few significant events visible to a party interested in observing the development of the business interaction. This idea is captured by our execution model of a business operation shown in Figure 3.5.

Our execution model incorporates four stages: initiation, synchronisation of initiation outcomes, actual protocol execution and synchronisation of execution outcomes. B2B messaging is typically implemented using Message Oriented Middleware (MoM) that permits loose coupling between partners (that is, the partners need not be online at the same time). We therefore assume that an initiation protocol is required to get the partners ready for execution of the business conversation. After the initiation protocol is concluded, both the initiator and the responder eventually produce either *InitSucc* or *InitFail* to declare that locally the initiation was successful or failed. The parties then have to reach a consistent view of the outcome of the initiation protocol. To achieve this, they engage in a synchronization protocol. Such a protocol could be based on the three way handshake used in TCP [62], as discussed in [63]; this synchronization of outcomes is represented as *init sync protocol* in Fig. 3.5. Naturally, the init sync protocol declares *InitSucc* only when both partners declare success and *InitFail* in any other possible combination of local outcomes. Assuming initiation succeeds, the actual conversation protocol is executed. During the execution of the conversation, the private business processes that each partner has in place in the corresponding operation are involved for all the choices that are taken internally.

Following the ebXML specification [17], we assume that once a conversation is started, it always completes to produce at each side one of three possible events: *Success*, *BizFail* or *TecFail*, rep-

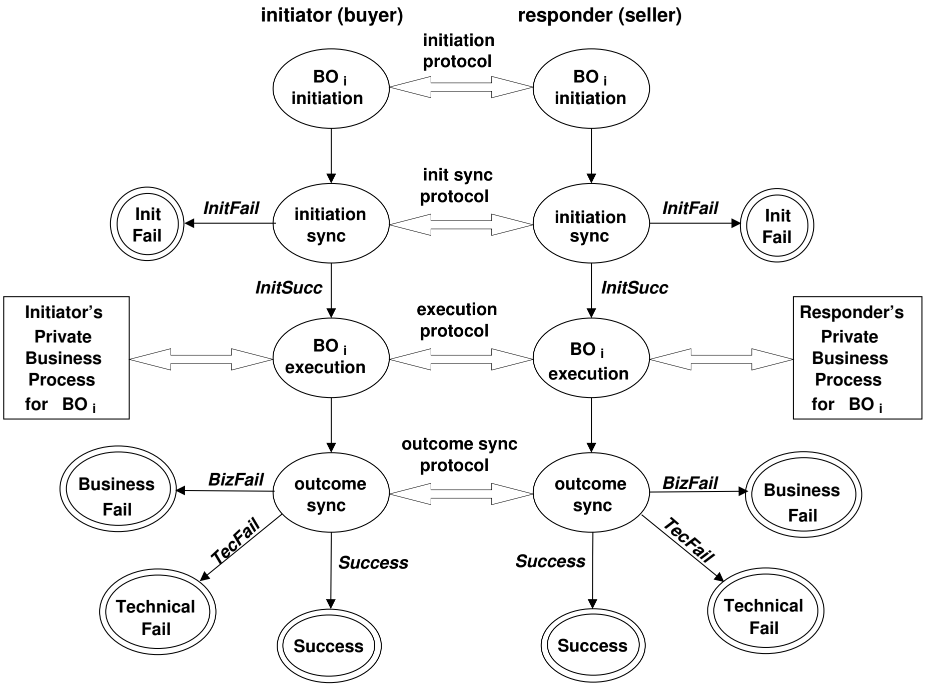

Figure 3.5: Execution model of a business conversation

resenting success, business failure and technical failure, respectively. When a party considers that the conversation completed successfully, it generates a *Success* event. *BizFail* and *TecFail* events model the 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 late, syntactically incorrect or missing messages. *BizFail* models semantic errors detected at the business level in a successfully received document, e.g., an invalid address for goods delivery or a payment for the wrong amount. To guarantee that both partners have consistent views about their conversation outcomes, they can execute a synchronisation protocol (represented as *outcome sync protocol* in Fig. 3.5). In general, the execution of the outcome synchronisation protocol can be regarded as a composition of the events generated by the two parties, according to a given composition logic, shown in Table 3.1.

This synchronisation is executed as part of the conversation between the partners, for example as suggested in [63], and it ensures that the parties regard the outcome of a business conversation in the same way, and that a conversation is regarded as successful if and only if both parties have generated a *Success*. More precisely: (a) identical outcome events are composed into a composite event of the same type; (b) if one of the outcome events is *TecFail* then the composite event is of type *TecFail*, irrespective of the type of the other event; (c) if one of the outcome events is *BizFail* and the other is *Success*, then the composite event is of type *BizFail*.

An alternative choice for synchronization is to have an event composing service, which for example

Table 3.1: Outcomes of Event Composition.

Originator	**Responder**	**Composite event**
$Success$	$Success$	$Success$
$Success$	$BizFail$	$BizFail$
$BizFail$	$Success$	$BizFail$
$BizFail$	$BizFail$	$BizFail$
$Success$	$TecFail$	$TecFail$
$BizFail$	$TecFail$	$TecFail$
$TecFail$	$Success$	$TecFail$
$TecFail$	$BizFail$	$TecFail$
$TecFail$	$TecFail$	$TecFail$

could be offered by the CCC itself. Such an Event Composer would receive all the basic events generated, and match up the ones pertaining to the same conversations to generate a composite event according to the logic given in Table 3.1. This is discussed in more detail in [64].

Independently of the chosen solution, we assume that the synchronised outcome events to do with initiation (*InitSucc*, *InitFail*) and execution (*Success*, *BizFail*, *TecFail*), from now on referred collectively as *conversation events*, are notified to the Contract Compliance Checker through a monitoring channel (Fig. 3.6; see Section 3.3).

3.3 Contract Compliance Checker

The Contract Compliance Checker (shown in Figure 3.6) is the heart of our architecture. The CCC is a neutral entity, conceptually located between the interacting parties; its function is to observe the conversation events they produce and to infer from these whether the business operations these events relate to are contract compliant or non-contract compliant. The CCC relies on the existence of a monitoring channel that could be realised by a publish/subscribe service to which the CCC subscribes and the interacting parties publish.

3.3.1 Assumptions

We assume that the business partners operate in good faith and supply the conversation events to the CCC; in particular, they do not generate malicious events. Naturally, practical systems might require security mechanisms for event generation and transmission to the CCC. We leave this as a subject for future research, as mentioned in Chapter 7. We require, however, that the CCC itself must observe the business interaction accurately; a correctly-functioning CCC should never treat a successful operation as if it were a technical or business failure, and vice versa. A potential threat that can affect the functionality of the CCC is the failure of any of its components; if the CCC is

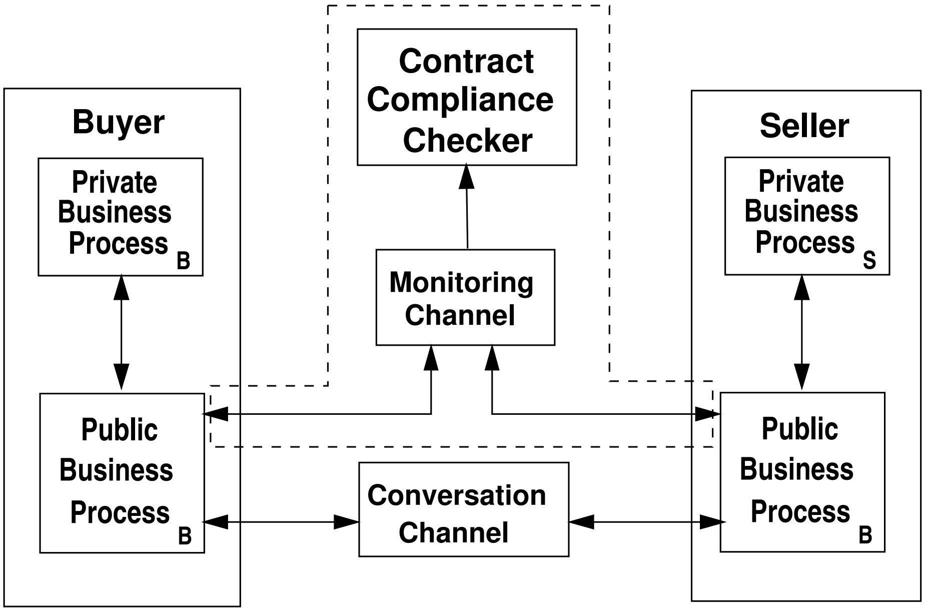
Contract Compliance Checker
Buyer
Private Business Process B
Monitoring Channel
Seller
Private Business Process S
Public Business Process B
Conversation Channel
Public Business Process B

Figure 3.6: Abstract view of the architecture

not completely functional at all times during a business transaction, the interaction would not be observed accurately. For instance, if the Event Logger, discussed in detail in Subsection 3.3.2, failed for a given time interval, all events received during that time interval would not be recorded, and references to them in the contract would not work as expected.

We also have to make assumptions on the MOM used for the communications and monitoring channels; in particular, we require that the monitoring channel always functions correctly, and that its transmission and processing delays are bounded and known. The reason for this is to prevent the CCC from getting a view of the outcome of a business operation that diverges from the view held by the participants. In the case they declare a success but the message notifying it reaches the CCC late, the CCC would flag the operation as having concluded late, while the participants have declared it successful. This divergence cannot be handled within the system, and needs to be resolved outside of the context of the contract execution. This resolution process will be able to make use of the historical data stored by the Event Logger, a component of the CCC introduced in the following Subsection. An analysis of the relevant events, and in particular of their timestamps, would allow to reconstruct the right sequence of events with their right timings. The execution of the business transaction could then be resumed from there.

To summarize, in order to ensure that the monitoring process itself does not inject errors, our model needs to work under the following assumptions:

- The components inside the dashed box of Figure 3.6 function correctly.
- The clocks of all parties are synchronized to a master global clock with a known accuracy, ϵ.

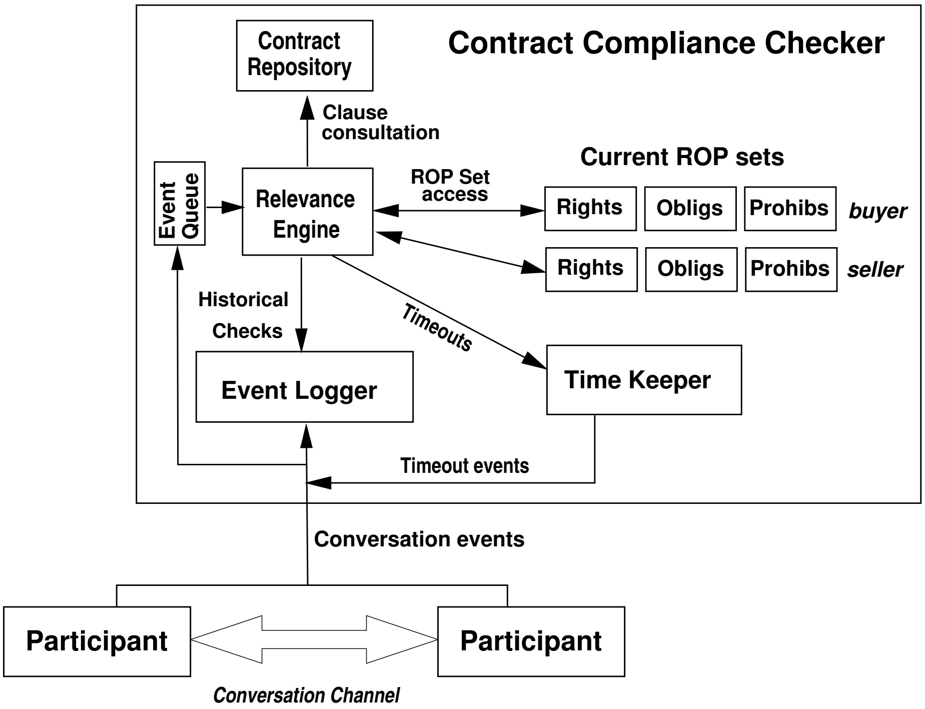

Figure 3.7: Abstract view of the architecture

Thus the difference between the readings of any two clocks is never larger than 2ϵ.

- Events are time stamped at the source.
- Events are delivered exactly once to the contract compliance checker in temporal order.
- The transmission and processing delays (TPD) from the synchronising facilities to the Event Queue of the CCC, discussed further on, are bounded and known.
- The CCC has a timer process for generating timeout events as per the contract. To guarantee that this timer does not unnecessarily generate a timeout event about the absence of an operation when the outcome event about the contract execution of the operation is on its way to the Event Queue, all timeout events are delayed by the quantity TPD + 2ϵ. This quantity compensates for transmission and processing delays and any error in clock synchronisation.
- The buyers and sellers infrastructure components can fail by crashing but they eventually have to recover; however all conversation events that are generated are supplied to the monitoring channel as per TPD.

3.3.2 Architecture of the CCC

The architecture of the CCC is shown in Figure 3.7. It is built on an Event-Condition-Action (ECA) mechanism that reacts to conversation events as the business partners execute business operations. We assume that events contain the following attributes:

- the name of the business operation they refer to;
- an outcome, one from the set {*InitSucc, InitFail*} if it is an initiation event, or one from the set {*Success, BizFail, TecFail*} if it is an outcome event;
- the names of the initiating and responding role players;
- a timestamp of the occurrence of the event, as specified among the assuptions listed in the previous Subsection;
- a unique ID for the business transaction the event pertains to, used internally to assign events to the right business transactions.

The information in the timestamp can also be accessed for contract writing convenience using virtual attributes like *year, month, day, hour, minute*, *weekday* and so on, that are internally mapped to the appropriate field of the timestamp.

The events received by the CCC are forwarded to the *Event Logger* for future historical references, and to the *Event Queue*, a FIFO queue where they are stored in temporal order. Events are processed by the *Relevance Engine* (RE), the heart of the CCC. The RE maintains the *contract repository*, a rule base derived from the contract, and keeps up to date the ROP sets of the role players. This is accomplished with the following algorithm:

1. Receive an Event e from the communication channel;
2. Analyse the contract repository and identify relevant rules for e;
3. For each relevant rule r, execute the actions listed in its right hand side.

The actions in the right hand side of a rule can terminate the execution of a contract or manipulate a role player's ROP set. In this manner, the CCC knows exactly what rightful, obligatory and prohibited operations the business partners can execute, and their associated deadlines. These deadlines are managed by the *Time Keeper*, which is tasked to track them, and to generate special timeout events when they expire. These events are then forwarded to the Event Queue and the Event Logger in the same fashion of externally generated events.

3.3.3 Verification of Compliance by the CCC

Compliance checking consists, as defined in Section 3.1.4, of determining whether the execution of a given business operation, initiated by a given role player at a given time, matches what the contract specifies in terms of identity of role players, timing, history and ROP sets of the involved role players.

The CCC verifies contract compliance of business operations by reacting to the events generated because of their execution. Upon receiving an event generated by a role player rp for the execution of a business operation bo, the CCC verifies first whether the constraint $bo \vdash ROP_{rp}$ holds. If it does not, then the operation is out of context, and therefore non-contract compliant. If it holds, then the CCC verifies additional constraints, such as the initiator's and the responder's identities, the date and time of occurrence and the history of the business transaction, to determine if the operation is contract compliant according to the executed contract.

If the operation is ultimately deemed contract compliant, the CCC will react by updating the ROP sets of the role players as disposed by the contract in execution. But if the operation is non-contract compliant, the events generated by its execution will be ignored, in the sense that they do not alter the role players' ROP sets. If deemed necessary, however, the CCC can be instrumented to record the occurrence of all non-contract compliant operations and separate them in corresponding logs of unknown and out of context business operations. These records could be useful in off-line examination of the interaction.

3.4 Representation of Contractual Clauses with Rules

Business contracts are written in a natural language, because of tradition and of legal constraints. A document in a natural language is not suitable for direct electronic interpretation, because of the ambiguities inherent in natural languages. Therefore an electronic representation of the contents of the contract is usually generated for monitoring and enforcing purposes. Researchers agree that the generalized version of this problem is hard to automate; *rebus sic stantibus* much of the translation work has to be done by humans. It is often the case, however, that commercial contracts are composed out of relatively standardized templates, for which a translation could be prepared once and reused.

This Section will present an abstract model for the representation of contractual clauses with *business rules* in *Event-Condition-Action* (ECA) form. According to [65],

> A business rule is a statement that defines or constrains some aspect of the business. It is intended to assert business structure or to control or influence the behaviour of the business. The business rules that concern the project are atomic – that is, they cannot be broken down further.

3.4.1 Structure of a Rule

Rules in ECA form consist of three parts:

- the *event* part, specifying a type match of an event;
- the *condition* part, specifying a set of Boolean expressions in conjunction;
- the *action* part, specifying a list of statements, usually data modifications.

If an ECA rule looks very much like an *if*-statement, that's because it is: the statements in the action part are executed if and only if the Boolean expressions in the event and condition parts are true. The translation of a natural language contract yields a set of ECA rules, a *rule base.* The details of the translation process will not be discussed here, but the existence of a *verified* and *consistent* set of rules will be assumed. A verified rule base is one that does not contain errors - that is, actions that run contrary to the letter, as well as the spirit, of the original contract. For instance, a rule that prohibits a buyer to pay, or that only grants a deadline for payment of 10 seconds, runs contrary to the intended purpose of our sample contract, and its presence in a rule base derived from that contract would be erroneous. A consistent rule base is one that does not contain rules in conflict over the actions to take in response to any event. For example, a rule base for our sample contract would not be consistent if it contained two rules that, in case of a successful payment, respectively forbid and oblige the seller to deliver the purchased goods. As mentioned above, we will not delve further in these matters, which are at least partly connected with the translation process, and therefore not in the scope of this work.

It is worth observing that in general the correspondence between contractual clauses expressed in natural language and rules is not one-to-one, but many-to-many. Furthermore, designers can take different alternatives to convert clauses into rules , depending on where and how conditions are expressed and verified.

In the EROP model, a rule r has the form $((e \equiv \mathit{eventType}), c_1, \ldots, c_n) \rightarrow a_1, \ldots, a_m$, where e is an event, $e \equiv \mathit{eventType}$ is an *event match*, $c_1, \ldots, c_n$ are conditions, which can be constraints on event attributes, on the historical records or on the current *ROP* sets, and $a_1, \ldots, a_m$ are actions. This is an example of a rule:

$$\begin{gathered}((e \equiv POSubmissionSuccess), (PurchaseOrderSubmission \vdash R_{e.originator}),\\ (e.outcome == \text{``}Success''))\\ \rightarrow \{O_{e.responder} + = (\mathit{ReactToPurchaseOrder}\}, \text{`}24h'))\}\end{gathered}$$

This rule can be read in English as "If the event currently being examined is of type *Purchase Order Submission*, while its outcome is successful and its originator has the right to initiate the operation *Purchase Order Submission*, then impose on the responder the obligation to execute the operation *React to Purchase Order* within 24 hours".

The event match and the conditions in the left hand side of a rule express the constraints for contract-compliance introduced in Subsection 3.1.4. In order to consider the execution of a business operation as contract-compliant, all of them must be verified.

Event Matches: An event match is a Boolean expression of the form $e \equiv eventType$ evaluating to TRUE if and only if e is of type *eventType*. Legitimate event types are names of business operations (to indicate an event pertaining to the initiation or execution of a business operation), names of rights, obligations or prohibitions with the *Timeout* suffix (to indicate an event pertaining the expiry of the ROP entity's deadline), or the special type *Init* to indicate the beginning of a new business interaction. For example, the constraint that an event e should be of type POSubmission can be expressed as $e \equiv POSubmission$.

Event Attribute Constraints: An event attribute constraint is a Boolean expression asserting the value of an attribute of en event. As an example, the constraint that the originator of an event should be the role player *buyer* can be expressed as *e.originator == "buyer"*.

Historical Constraints: A historical constraint is a Boolean expression used to assert the presence or absence of a number of events in the historical records of the business partnership. Historical constraints can be *Boolean*, asserting the existence or absence of at least one given event, or *numeric*, asserting the existence or absence of a given number of events of a given type. Boolean constraints take the form *happened(eventType, originator, responder, timeConstraint, outcome)*; numeric constraints take the form *countHappened(eventType, originator, responder, timeConstraint, outcome) == N*, where *eventType* is a legitimate event type as defined above, *originator* and *responder* are the two involved role players, *timeConstraint* is an expression asserting a constraint on the timestamp of recorded events, and *outcome* is a legitimate outcome description - one of *InitSucc, InitFail, Success, TecFail* or *BizFail.* As an example, a constraint imposing that the originator of an event e submitted a successful purchase order before midnight of 1/1/2008 can be expressed with *happened("POSubmission", "buyer", "seller", "timestamp" < '1-1-2008 00:00').*

ROP Constraints: A ROP constraint is a Boolean expression asserting that a business operation matches a role player's ROP set. It has the form $BOType \vdash ROPSet_{roleplayer}$. For example, a constraint imposing that the originator of an event e should have the right to execute a purchase order is $POSubmission \vdash R_{e.originator}$.

Rule Actions: Actions in the right hand side of a rule can either modify the ROP sets of role players or conclude the business transaction. A transaction can be closed with *terminate(outcome)*, where *outcome*, like in the case of business operations, can be *Success*, *TecFail* or *BizFail.* The outcome will be broadcast to all involved roleplayers. Manipulation of the ROP sets is done with the C++-inspired += and -= operators respectively to add or remove ROP entities to them, using this syntax:

```
roleplayer.ropset += BusinessOp|CompositeOblig ([deadline])
roleplayer.ropset -= BusinessOp|CompositeOblig ([deadline])
```

where *roleplayer*, *BusinessOp* and *CompositeOblig* are respectively role players, business operations and composite obligations; *ropset* is one of *rights, obligs* or *prohibs* (respectively, the set of rights, obligations or prohibitions), and *deadline* is a legitimate deadline for the expiry of the new ROP entity (which can be $\emptyset$ in case of a right or prohibition with no expiry).

3.5 Exception Handling in EROP

The handling of exceptional situations was described in Chapter 2 as an important, desirable feature for a contract specification language to have. There are two main reasons for its importance. Firstly, electronic contracts have to be able to take into account the distributed nature of the computations underlying a business transaction, and that can be achieved only by paying due attention to the impact of software, hardware and network related problems (such as clock skews, unpredictable transmission delays, incorrect or lost messages, and so on).

Secondly, a system such as EROP would be mostly used in B2B settings where partners are only loosely coupled. Because of the potential complexity of business transactions, there is a danger that business partners could get out of synchrony with each other; this could divert the transactions from their normal paths, eventually leading to contract violations.

Let us consider this example. In the Buyer-Seller scenario, after a successful purchase order is accepted and the buyer is invoiced, payment fails repeatedly for technical reasons, caused by a problem outside the control of the participants, such as the credit card processing system of the seller's bank being down. In traditional business practice this would be solved in person over a telephone call, bending the contractual clauses for the sake of recovering the exceptional situation. In electronic contracts, the possibility should be available to make plans for alternative courses of action to reduce, if not remove, the risk of having a dispute to resolve offline; in the situation presented above, for example, there could be a set of rules disposing that, in the case of a failure of the credit card processing system, the buyer should be allowed to pay using an alternate channel, such as a bank transfer or a cheque.

The EROP model can be employed to reason about exceptional situations, and making plans to attempt the recovery of exceptional situations. These will be executed through the manipulation of the ROP Sets, granting new rights, assigning prohibitions and imposing compensatory obligations –

the contrary-to-duty obligations of [44] – in order to attempt different execution paths. The EROP model uses the same features (manipulation of ROP sets) used in normal operation for exception handling. This has the advantage of allowing a homogeneous treatment of exceptional and normal situations.

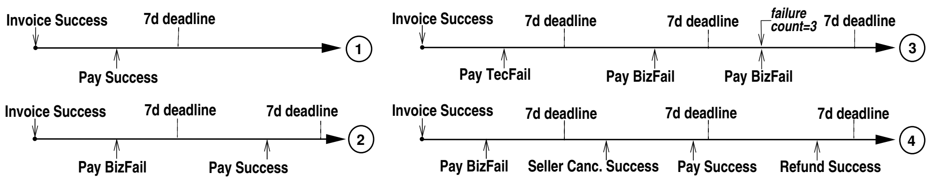

Figure 3.8: Execution of payment conversations with success and failure outcomes.

In many cases, exceptional situations arise from the non-satisfaction of obligations, which then expire. The EROP model can be used for the expression of clauses to dynamically extend obligation deadlines, such as clause C7 of the Buyer-Seller contract. This clause can result in quite complex execution patterns, such as the ones displayed in Fig. 3.8: the Figure shows four possible timelines of the Payment conversation. In the first scenario the payment succeeds in the first attempt within the seven day deadline (7d). In the second, it fails once due to a BizFail, so a seven day deadline extension is granted to the buyer, and the right to cancel is granted to the seller. The buyer succeeds in his second attempt (Pay Success) while the seller decides not to cancel. In the third scenario, the payment fails three times (a *TecFail* followed by two *BizFail*) without cancellation from the seller, so the business transaction is stopped after three failures. In the last scenario the payment succeeds in the second attempt (*Pay Success*) while the seller successfully exercises his right to cancel (*Seller Canc. Success*) after the buyer's first attempt to pay fails (*Pay BizFail*); if the execution of *Pay Success* and *Seller Canc. Success* conversations over- laps, it is possible that (as shown in the figure) the event *Pay Success* arrives at the CCC after *Seller Canc. Success*; consequently, the seller executes a Refund conversation that succeeds.

The EROP rules derived from clause C7 are presented in the Section 4.4, and are able to capture the complex patterns discussed above in a concise manner.

Chapter 4

Language

This chapter will introduce the EROP language, providing an informal overview of its features and syntax, and the full EROP version of the sample contract presented in Section 1.2. A formal grammar for the EROP language is presented in the Appendix for reference purposes. Material from this chapter appeared in [66, 67].

4.1 Structure of a Contract

An electronic contract written in the EROP is composed of two sections: the declaration section, where all the entities used in the contract are declared, and the rules section, where all the rules in the contract are declared.

The examples and snippets of code in this chapter all come from the EROP version of the Buyer-Seller contract that will be presented in Section 4.4.

4.2 Declaration Section

This section of the contract contains the declarations of the names of role players, business operations and composite obligations used in the rules.

Role players and business operations are expected to be defined elsewhere; role players could be defined in a system for access control, such as an RBAC system [58], while an agreed upon standard such as RosettaNet [54] could be employed to define the conversations that make up the business operations. The declarations in an EROP contract of business operations and role players are needed to make these visible in the contract itself.

The keyword ***roleplayer*** introduces a comma-separated list of role players, and the keyword ***businessoperation*** introduces a comma-separated list of business operations closed by a carriage return. Legitimate names for role players and business operations must begin with a letter, but can include letters, numbers, and all the other characters that can be used in a Java identifier.

It is not possible to use reserved words of the EROP language; it is also not possible to reuse a name that has already been used, as in this case the later definition will just hide the earlier one. Conventionally, role player names begin with a lowercase letter, and business operation names begin with an uppercase letter.

The following code snippet, taken from the EROP contract that will be presented further on in this chapter, declares the role players *buyer* and *seller* and the business operations *POSubmission*, *InvoicePayment*, *POAcceptance* and *PORejection* (where *PO* stands for *Purchase Order*).

```
roleplayer buyer, seller
businessoperation POSubmission, InvoicePayment
businessoperation POAcceptance, PORejection
```

The keyword ***compoblig*** defines a composite obligation, as discussed in Chapter 2. It is followed by the name of the composite obligation, and then by a bracketed list of the business operations that are to be executed OR-exclusively to satisfy the obligation. The following snippet defines a composite obligation to either accept or reject a purchase order, which will be used in the rules derived from the clause C2.

```
compoblig RespondToPO (POAcceptance, PORejection)
```

4.3 Rule Syntax

The syntax of a rule is

```
rule "ruleName"
  when
    triggerBlock
  then
    actionBlock
end
```

The name of a rule identifies the rule uniquely. It has to begin with a letter, and can contain letters, numbers and all the symbols allowed in a Java identifier. It is good practice to give meaningful names to rules.

The following snippet presents an example of a rule, derived from clause C1 of our Buyer-Seller contract.

```
rule "R1"
  when
```

```
    e matches (botype == POSubmission)
  then
    Success:
      if e.originator == buyer
        && POSubmission in buyer.rights
        && e.weekday in [Mon ... Fri]
        && e.time in [9 ... 17]
      then
        seller.obligs += RespondToPO("24h");
      endif
    Otherwise:
      pass;
end
```

4.3.1 Trigger Block

The trigger block of a rule determines when that rule is triggered and its action block executed. It is composed of a set of one or more boolean expressions in conjunction, one of which must be an *event match*, that will be discussed in Section 4.3.1.1. The expressions of the trigger block are evaluated during the recognise-act cycle, and the rule is considered triggered if and only if the result of their conjunction evaluates to true.

The syntax of a trigger block is

```
eventMatch
[condition]*
```

The additional conditions in a trigger block can be:

- Constraints on the fields of the current event;
- Constraints on the ROP sets of the participants;
- Historical constraints.

and they will be discussed in Section 4.3.1.2.

4.3.1.1 Event Matches

An event match is a boolean expression that compares the fields of an event object with a tuple of values. It follows the syntax

```
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 type of the event; it can be a business operation type as defined in the declaration section, one of those with the suffix *Timeout* to indicate a timeout in its execution, or the special type *Init* that indicates the beginning of a new contract.
- **outcome**: the outcome of the event. It can be one of *Success, TecFail, BizFail, InitSuccess, InitFail*, if the event represents respectively a successful conclusion of a business operation, a technical failure, a business failure, a successful initiation or an initiation failure.
- **originator**: the name of the role player that initiated the execution of the business operation; it must be one of the role player names in the declaration section. In the case of a Timeout, it is the name of the role player that was expected to initiate the execution the business operation that timed out.
- **responder**: the name of the role player that the originator is trying to interact with. In the case of a Timeout, it is the name of the role player that was expected to respond to the operation that timed out.
- **timestamp**: the instant in time when the event was received.

An *operator* in an event match can be any boolean comparison operator that can be applied to an event field: ==, !=, <, > and so on.

A *value* in an event match is a Java expression that evaluates to the same type of the fields with which it is compared, so that strings are evaluated with strings, timestamps with timestamps and so on.

In the sample rule above, the event match is the line saying *e matches (botype == "POSubmission")*; this evaluates to true if and only if the event *e* is of type *POSubmission.* Here is a more complex event match:

```
e matches (botype == Payment, originator == purchaser,
  responder == supplier, outcome == success)
```

This event match evaluates to true if and only if the event being processed reports the successful completion of a business operation of type *Payment* initiated by the role player *purchaser* with the role player *supplier.*

4.3.1.2 Event Field Constraints

It is possible to add additional constraints on the fields of the current event being examined outside the event match. These constraints are functionally the same as the ones in the event match; the purpose for allowing them is to leave greater flexibility to contract writers. It is, in short, a form of syntactic sugar.

An event field constraint follows this syntax:

```
e.field operator value
```

where *field*, *operator* and *value* can be the same as in event matches. Event field constraints can be used to build more complex boolean expressions.

4.3.1.3 ROP Set Constraints

Constraints on the presence or absence of particular business operations or composite obligations in role players' ROP sets can be expressed with the operators *in* and *not in*, to test respectively for presence or absence. The syntax for these constraints is

```
businessOperation|compositeObligation in|not in roleplayer
```

where business operations, composite obligations and role players have previously been defined in the declaration section of the contract.

In the following sample rule, the ROP set constraints in the trigger block test for the presence of the business operation *POSubmission* in *buyer*'s rights set and the absence of the composite obligation *RespondToPO* in *seller*'s obligations,

```
rule "PurchaseOrderSubmission"
  when
    e matches (botype == POSubmission, originator == buyer,
      responder == seller, outcome == Success)
    POSubmission in buyer.rights
    RespondToPO not in seller.obligations
  then
    # Actions follow...
end
```

4.3.1.4 Historical Constraints

Historical constraints can be used to condition the triggering of rules on the presence or absence of certain events in the Historical Database. They are Boolean expressions that involve one or more *historical queries.* There are two types of historical queries: the *boolean* ones, that evaluate to a Boolean value and assert the presence (or absence) of at least one instance of an event matching certain specified conditions, and the *numerical* ones, that evaluate to a non-negative integer and count the occurrences of a given number of events matching certain specified conditions.

Boolean queries are expressed with the keyword ***happened***, using this syntax:

```
happened(businessOp, originator, responder, outcome,
timeConstraint)
```

where *businessOp* is a legitimate business operation, *originator* and *responder* are the involved roleplayers, *outcome* is one of *Success, TecFail, BizFail, InitSuccess, InitFail* and *timeConstraint* is a string expressing a constraint on the timestamp of the acceptable events in the Historical Database. A "*" can be used as a wildcard for any of these fields.

A Boolean query to verify that the an event was recorded for a successful *Payment* operation between the roleplayers *buyer* and *seller* at any time in the past would look like this:

```
happened(Payment, buyer, seller, Success, *)
```

Numerical queries are expressed with the keyword *counthappened*, using the same syntax of a Boolean constraint:

```
happened(businessOp, originator, responder, outcome,
timeConstraint)
```

A numerical query to count the occurrences of a Payment between buyer and seller that failed for technical reasons would look like this:

```
counthappened(Payment, buyer, seller, TecFail, *)
```

4.3.2 Action Block

The action block of a rule determines the consequences of that rule being triggered. Two kinds of actions can appear in the right hand sides of EROP rules: manipulations of participants' ROP sets, and terminations of the current contract instance. Conditional statements can appear in the action block to organize the actions.

4.3.2.1 ROP Set Manipulation

Manipulation of the ROP sets is done in EROP with the C++-inspired += and -= operators respectively to add or remove ROP entities to them, using this syntax:

```
roleplayer.ropset += BusinessOp|CompositeOblig ([deadline])
roleplayer.ropset -= BusinessOp|CompositeOblig
([deadline])
```

where *roleplayer*, *BusinessOp* and *CompositeOblig* are respectively role players, business operations and composite obligations and have been previously declared; *ropset* is one of *rights, obligs* or *prohibs* (respectively, the set of rights, obligations or prohibitions), and *deadline* is an expression that evaluates to a date. Deadlines can be *absolute*, referring to a specific point in time in a "DD/MM/YYYY HH:MM:SS" format (e.g., "01/04/2009 12:00"), or they can be *relative* to the current time, referring to a point in time at a specified distance in the future. The syntax for a relative deadline is

```
[value timeinterval]+
```

where *timeinterval* is one of ***Y, M, d, h, m, s***, respectively indicating years, months, days, hours, minutes and seconds, and *value* is the number of those; letters indicating time intervals should not repeat, and values should make sense for the corresponding time interval (no hour greater than 23, no minute greater than 59, and so on). For example, the expression "3d12h" means "Three days and twelve hours from now".

The following code snippet shows three actions that respectively add an obligation to execute *Invoice* to the seller's obligation set, remove the right to execute *POSubmission* from the buyer, and add the prohibition to execute *GoodsDelivery* to the buyer's prohibition set.

```
seller.obligs += Invoice("24h");
buyer.rights -= POSubmission;
buyer.prohibs += GoodsDelivery;
```

4.3.2.2 Conclusion of a Contract Instance

Contract instances can be terminated with a special type of action within the action block. A contract instance can terminate with one of four states: success, technical failure, business failure, or initiation failure, just like a business operation. Initiation failure cannot be declared from within a running contract instance, and is declared automatically when an attempt to initiate one fails.

The keyword to use to terminate a contract instance is *terminate*, using this syntax:

```
terminate("status")
```

where *status* is one of *Success*, *TecFail*, *BizFail* or *InitFail*, respectively for a successful conclusion, a technical failure, a business failure or an initation failure. The outcome of the contract instance is communicated to the roleplayers, and it can be reused at a higher level of interaction.

4.3.2.3 Conditional Statements

Actions can be organized using two conditional structures, *if-then-else-endif* statements and the *status guards*. Both are a form of syntactic sugar, that do not alter substantially the language's functionality but make it "sweeter" for humans to use, allowing a writing style that is more natural and more productive.

The *if-then-else* statement is used, like in ordinary programming, to allow conditional execution of actions in the right hand side of rules, depending on the value of a boolean expression. As an example, the *if-then-else-endif* statement allows to rewrite the two rules

```
rule "Rule1"
  when
    e matches (botype == SomeBO, ...)
    booleanConditions
  then
    actionBlock1
end

rule "Rule2"
  when
    e matches (botype == SomeBO, ...)
    !booleanConditions
  then
    actionBlock2
end
```

as the single rule

```
rule "RuleIfThen"
  when
    e matches (botype == SomeBO, ...)
  then
    if booleanConditions
    then
      actionBlock1
    else
      actionBlock2
    endif
end
```

The keyword *else* is not mandatory, and if not used (that is, if a rule only has an if-then construct) no action block in the *if-then-endif* statement will be executed if its condition is not triggered.

Status guards are more specialized conditional statements to control action execution according to an event's outcome. They are the keywords *Success, TecFail, BizFail, InitFail, Other*, and they are used to identify action blocks that have to be executed in the case the event under scrutiny has an outcome status respectively of successful, technical failure, business failure, initiation failure, or any one that was not covered in the same rule. The use of status guards therefore allow to rewrite a set of rules such as this:

```
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 "RuleForOtherOutcomes"
  when
    e matches (botype == SomeBO)
    ((e.outcome != Success)||(e.outcome != TecFail))
  then
    actionBlock3
end
```

as the following single rule:

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

4.4 The Buyer-Seller Contract in EROP

In this section we are going to present the EROP version of the contract for the Buyer-Seller scenario presented in Section 1.2. For the convenience of the reader, here are the original clauses:

- **C1**: The Buyer has the right to submit a purchase order, as long as it is from Monday to Friday and from 9am to 5pm (Right).

- **C2**: The Seller has the obligation to either accept or reject a purchase order within 24 hours (Obligation). Failure to satisfy this obligation will abort the business transaction for an offline resolution.

- **C3**: If the purchase order is accepted, the Seller is obliged to submit an invoice within 24 hours (Obligation), or the business transaction will be aborted for an offline resolution. If the purchase order is rejected, the transaction is considered concluded.

- **C4**: After receiving an invoice, the Buyer is obliged to respond within seven days, either paying the due amount or cancelling the purchase order (Obligation). Failure to satisfy this obligation will abort the business transaction for an offline resolution. Cancelling the purchase order is prohibited to the Buyer in any other condition (Prohibition).

- **C5**: Cancellation of a purchase order by the Buyer eliminates all obligations on both parties and concludes the business transaction. If a payment has been received before a cancellation, the Seller has the obligation to completely refund it (Obligation).

- **C6**: Once the payment is received, the Seller is obliged to ship the ordered goods within seven days (Obligation). A successful delivery will conclude the business transaction. Failure to satisfy this obligation will abort the business transaction for an offline resolution.

- **C7**: If a payment fails for technical or business related 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 (Right). In any other condition, the Seller is prohibited to cancel the purchase order after accepting it (Prohibition).

- **C8**: Buyer and Seller are obliged to stop the execution of the business transaction upon the detection of three failures to execute a payment (Obligation). Possible disputes shall be resolved offline.

First of all, let us declare the roleplayers, the business operations and the composite obligations used in this contract. PO will stand for “Purchase Order”.

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

Rule R1 is derived from clauses C1 and C2. It triggers when a successful purchase order is executed by the buyer within the acceptable time limits, and imposes on the seller the composite obligation to either accept or reject the order within 24 hours. An unsuccessful order, or one submitted for example during the weekend, would be ignored.

```
rule "R1"
  when
    e matches (botype == POSubmission)
  then
    Success:
       if e.originator == buyer
        && POSubmission in buyer.rights
        && e.weekday in [Mon ... Fri]
        && e.time in [9 ... 17]
      then
        seller.obligs += RespondToPO("24h");
      endif
    Otherwise:
      pass;
end
```

Rule R2Acceptance is derived from clause C3. It triggers when the seller accepts a purchase order while under the obligation to react to one, and it replaces that obligation on the seller with a new one, requiring to invoice the buyer within 24 hours.

```
rule "R2Acceptance"
```

```
  when e matches (botype == POAcceptance)
  then
    Success:
      if e.originator == seller
        && RespondToPO in seller.obligs
      then
        seller.obligs -= RespondToPO;
        seller.obligs += Invoice("24h");
      endif
    Otherwise:
      pass;
end
```

Rule R2Rejection is also derived from clause C3, but it triggers when the seller rejects a purchase order. In this case, the execution of the business interaction terminates successfully.

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

Rule R2Timeout is derived from clause C2. It triggers when the seller does not satisfy his obligation to accept or reject a purchase order; in this case the contract execution terminates with a business failure, possibly to be solved offline.

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

Rule R3 derives from clause C3 and triggers when the seller does not satisfy the obligation to submit an invoice. In this case the business transaction is terminated with a status of *BizFail* for offline resolution.

```
rule "R3"
  when
    e matches (botype == InvoiceTimeout, originator == seller)
    Invoice in seller.obligs
  then
    terminate ("BizFail");
end
```

Rules R4 and R5 are derived from clause C4. Rule R4 triggers when the seller successfully invoices the buyer; its effect is to remove the obligation to invoice from the obligations of the seller, and the prohibition to cancel the purchase order from the prohibitions of the buyer. The rule also imposes on the buyer the composite obligation to either pay the invoiced amount or to cancel the purchase order.

```
rule "R4"
  when e matches (botype == Invoice)
  then
    Success:
      if e.originator == seller
        && Invoice in seller.obligs
      then
        seller.obligs -= Invoice;
        buyer.prohibs -= POCancellation;
        buyer.obligs += RespondToInvoice("7d");
      endif
    Otherwise:
      pass;
end
```

Rule R5 triggers if the buyer does not comply with his obligation to pay an invoice or cancel the purchase order. In this case, the execution of the contract is terminated with a status of *BizFail*, and the issue should be solved offline.

```
rule "R5"
  when e matches (botype == RespondToInvoiceTimeout)
  then
      if e.originator == buyer
        && RespondToInvoice in buyer.obligs
      then
        terminate("BizFail");
      endif
end
```

Rules R6 and R6Refund are derived from clause C5. Rule R6 triggers when the buyer cancels a purchase order while the composite obligaiton to either cancel the order or pay is in force. Its effect is to remove the now satisfied obligation and to terminate successfully the execution of the contract.

```
rule "R6"
  when e matches (botype == POCancellation)
  then
    Success:
      if e.originator == buyer
        && RespondToInvoice in buyer.obligs
      then
        buyer.obligs -= ReactToInvoice;
        terminate ("Success");
      endif
    Otherwise:
      pass;
end
```

Rule R6Refund triggers when the buyer successfully cancels a purchase order, and, in the case the buyer previously paid the invoice, obliges the seller to refund the paid amount.

```
rule "R6Refund"
  when
    e matches (botype == POCancellation, originator ==  buyer,
      outcome == Success)
  then
    if happened(Payment, buyer, seller, Success, *)
    then
      seller.obligs += Refund("24h");
    endif
end
```

Clauses C5, C6 and C7 are expressed by the rule R6, which triggers when the buyer executes a payment. If the payment is successful, then the historical record will be searched to see if a cancellation had been successfully executed; if that is the case, then the seller is obliged to refund the paid amount to the buyer. Otherwise, the buyer's obligation to respond to the invoice is removed, while an obligation is imposed on the seller to deliver the paid goods.

If the payment is not successful, however, the deadline for the buyer to pay is extended by seven more days, but the seller will gain the right to cancel the purchase order, which was forbidden earlier. The rationale for this, as explained earlier in Section 1.2, is that this gives the interacting parties the opportunity to sort out the issues that caused the failure; to avoid abuse of this arrangement, the seller is able at any time to cancel the purchase order, and therefore abort the transaction.

```
rule "R6"
  when e matches (botype == Payment, originator == buyer)
  then
    Success:
      if RespondToInvoice in buyer.obligs
        && !happened(POCancellation, seller, buyer, Success, *)
        && !happened(POCancellation, buyer, seller, Success, *)
      then
        buyer.obligs -= RespondToInvoice;
        seller.obligs += GoodsDelivery("7d")
      else
        seller.obligs += Refund("24h");
      endif
    Otherwise:
      if RespondToInvoice in buyer.obligs
      then
        buyer.obligs -= ReactToInvoice;
        buyer.obligs += ReactToInvoice("7d");
        seller.prohibs -= POCancellation;
        seller.rights += POCancellation("7d");
      else
        pass;
      endif
end
```

Rule R7 derives from clause C7 and is triggered when the seller exercises the right to cancel a purchase order issued in rule R6. This removes the buyer's obligation to react to the invoice and terminates successfully the transaction.

```
rule "R7"
  when
    e matches (botype == POCancellation, originator == seller,
      outcome == Success)
    POCancellation in seller.rights
  then
      buyer.obligs -= ReactToInvoice;
      terminate ("Success");
end
```

Rule R8 derives from clause C6 and is triggered when the seller delivers the purchased goods. This satisfies his obligation, which is removed; contract execution is then successfully terminated.

```
rule "R8"
  when
    e matches (botype == GoodsDelivery, originator == seller)
    GoodsDelivery in seller.obligs
  then
    Success:
      seller.obligs -= GoodsDelivery;
      terminate ("Success");
    Otherwise:
      pass;
end
```

Rule R8Timeout also derives from clause C6 and is triggered when the seller's obligation to deliver the purchased goods times out. In this case, the execution of the contract is terminated for offline resolution with a *BizFail* outcome.

```
rule "R8Timeout"
  when
    e matches (botype == GoodsDeliveryTimeout, originator == seller)
    GoodsDelivery in seller.obligs
  then
      terminate ("BizFail");
```

```
end
```

Rule R9 is derived from clause C8 and it triggers whenever a payment by the buyer does not terminate with a successful outcome. In this case, the historical record is consulted, and if there are at least three failures, including the one currently being evaluated, the execution of the contract is terminated with a *BizFail* outcome.

```
rule "R9"
  when e matches (botype == Payment)
  then
   Success:
     pass;
   Otherwise:
     if (countHappened(Payment, buyer, InitFail, *)
       + countHappened(Payment, buyer, TecFail, *)
       + countHappened(Payment, buyer, BizFail, *)) >= 3
     then
       terminate ("BizFail");
     endif
end
```

Rule R10 derives from clauses C4 and C7; it handles the case where a role player attempted to cancel a submitted purchase order while being forbidden to do so. In this case, contract execution will terminate with a *BizFail* outcome.

```
rule "R10"
  when
    e matches (botype == POCancellation)
  then
    if
      (e.originator == buyer && POCancellation in buyer.prohibs)
      || (e.originator == seller && POCancellation in seller.prohibs)
    then
      terminate ("BizFail");
    endif
end
```

4.5 Conclusion

In this chapter we presented the syntax of the EROP language, together with an informal presentation of its semantics. We also showed how the Buyer-Seller scenario of Section 1.2 can be written in the EROP language, and discussed the meaning and workings of each rule.

Chapter 5

Evaluation

The previous chapters have presented the EROP model and language, shown the details of an experimental prototype and demonstrated their use in a detailed scenario. This chapter is intended to evaluate how the language introduced in this work stands against the list of desirable features for notations to represent electronic contract presented in Chapter 2, and repeated here for the convenience of the reader. Material from this chapter appeared in [66].

Declarative approach: A language should follow a declarative, high level style, concentrating as much as possible on "what" rather than "how". For example, it should be possible to express what the consequences are for a successful submission of a purchase order, without having to delve into the details of the conversation that occurs between the involved parties.

It can be argued that the use of mechanisms to support inheritance in a language for electronic contracts is part of a declarative approach. The possibility to extend incrementally contracts by specializing their vocabulary, in a parallel with inheritance in computer programming, can be argued to help encouraging reuse of existing work (code in the case of a program; clauses in the case of a contract). This could simplify the work of those that write contracts, and allow for a conceptual organization that is tidier and easier to maintain and debug.

Implementability: We can define a language as *implementable* if it is possible to efficiently and effectively implement it. A side effect of this definition is that contracts written in an implementable language should always provide programmers with useful information about how to implement a given functionality using current technology. An implementable language should involve only measurable parameters and observable conditions, and should not require interpretation to generate executable code. For example, expressions like "Customer's storage usage should not be high" or "Seller must eventually receive payment" are not implementable, while "Customer's storage usage should be below 2 Gbytes", and "Seller must receive payment within three days" are acceptable.

Expressiveness: A language should have enough expressive power to specify typical contractual clauses found in most practical applications, and be able to describe both functional and non-functional requirements. Functional requirements are related to high level business operations executed between the interacting parties and involve exchange of one or more business documents (examples are purchase order submissions, invoice notifications and so on). Non-functional requirements are central in service provision contracts (also known as service agreements) that specify the expected Quality of Service from the provider and the expected behaviour from the consumer.

Exception handling: It should be possible to specify how to deal with the consequences of exceptional situations that arise because of the inherently distributed nature of the underlying computations. Firstly, there should be easy ways to specify how to deal with any software and/or hardware related problems encountered during business interactions (e.g., unpredictable transmission delays, message loss, corrupted messages, semantically invalid messages, node failures, timeouts, etc.). Secondly, because B2B interactions typically take place between partners that are loosely coupled and in a peer-to-peer relationship, those partners can sometimes get out of synchrony and perform erroneous, even mutually conflicting operations; an example is a buyer cancelling an order after it has been paid, or, worse, shipped by the seller. A language for the specification of electronic contracts should provide easy ways to specify how to deal with such conflicting situations.

Usability: A language should be easy to use for the humans that write the contracts: it should be relatively intuitive and fast to learn, and it should take less time to accomplish particular tasks using it. While the notion of usability being desirable is intuitively acceptable, verifying whether a language is useable is difficult, as there is no tried and ready method to formally determine it. An exhaustive treatment of the usability of the EROP language is out of the scope of this work, and will have to be postponed.

Verifiability: A specification should be amenable to formal verification, possibly to a fully automated analysis, although a semi-automatic one might be acceptable. Even when electronic contracts are written by skilled humans, there still is a significant chance of introducing inconsistencies and loopholes, especially in the case of an existing contract that is being amended. Verification is a complex topic in its own right, and while there is a wealth of published work on formal verification, the complications are such that an exaustive treatment of verifiability is out of the scope of this work, to be left as a topic for future investigation.

The rest of the chapter will be dedicated to the presentation of a small selection of scenarios that will be implemented in the EROP language. Three of the following scenarios have been taken from relevant papers presented in Chapter 2; another has been inspired by Pandora [68], a commercial online music streaming service. The chapter will be closed by a discussion on the presence of the desirable features mentioned above in the EROP language.

5.1 Scenario: Goods Purchase, Variations on the Theme

5.1.1 Introduction

Governatori presents in [40] a sample contract for the sale of goods to demonstrate his approach for the specification of contracts in RuleML[39]. This contract resembles the the one presented in Chapter 1 and translated in the EROP language in Chapter 4, and we are going to reuse the work already presented there. Here we present only the clauses of immediate interest and leave out those that deal with issues not directly related to contract compliance, such as the jurisdiction the role players submit to. Rights, obligations and prohibitions will be highlighted in the text.

- ...
- **3 Price Policy**
  - 3.1 A "Premium Customer" is a customer who has spent more than $10000 on goods. Premium customers are entitled to a 5% discount on new orders.
  - 3.2 Goods marked as "special order" are subject to a 5% surcharge. Premium customers are exempt from special order surcharge.
  - 3.3 The 5% discount for premium customers does not apply for goods in promotions.
- **4 Purchase Orders**
  - 4.1 The Purchaser shall follow the Supplier's price lists at (http://supplier/catalog1.html).
  - 4.2 The Purchaser shall present the Supplier with a purchase order for the provision of Goods within 7 days of the commencement date (OBLIGATION).
- **5 Service Delivery**
  - 5.1 The Supplier shall ensure that the Goods are available to the Purchaser under Quality of Service Agreement (http://supplier/qos1.htm). Goods that do not conform to the Quality of Service Agreement shall be replaced by the Supplier (OBLIGATION) within 3 days from the notification by the Purchaser (RIGHT), otherwise the Supplier shall refund the (Purchaser) and pay the Purchaser a penalty of $1000 (OBLIGATION).
  - 5.2 The Supplier shall on receipt of a purchase order for (Goods) make them available within 1 day (OBLIGATION).
  - 5.3 If for any reason the conditions stated in 5.1 or 5.2 are not met, the Purchaser is entitled to charge the Supplier the rate of $100 for each hour the Goods are not delivered (OBLIGATION).
- **6 Payment**

  - 6.1 The payment terms shall be in full upon receipt of invoice (OBLIGATION). Interest shall be charged at 5% on accounts not paid within 7 days of the invoice date (OBLIGATION). The prices shall be as stated in the sales order unless otherwise agreed in writing by the (Supplier).
  - 6.2 Payments are to be sent electronically, and are to be performed under standards and guidelines outlined in PayPal.
- ...

5.1.2 EROP Version

The contract snippet presented earlier contains a mix of higher level and lower level information, relative to the abstraction level the CCC and the EROP language operate in. From the point of view of the CCC, and therefore of the contract writer, information like the discount for Premium Customers, the web location of the catalogue or the amount of dollars per hour of the fine is more about the "how" of the involved business operations, and less about the "what". These and other facets of the contract can be hidden within the complexity of the business conversations, allowing the contract writer to concentrate on their outcomes and their consequences.

From the contract segment above it is possible to derive two role players, *supplier* and *purchaser*, and the following business operations:

- *POSubmission*: Submission of a purchase order for goods that are not marked as special order. The conversation for this business operation will also include steps to negotiate a discount for Premium Customers if needed.
- *SpecialPOSubmission*: Submission of a purchase order for goods marked as being special order. The conversation for this business operation will also include steps to negotiate the surcharge for the special order, and to negotiate a discount for Premium Customers if needed.
- *GrantPremiumStatus*: Grant the status of "Premium Customer" to a role player. Modelling this as a business operation allows to model a check for a customer's status with a historical check: if the purchaser has Premium Customer status, a successful *GrantPremiumStatus* will be present in the history.
- *Payment*: Full and complete payment of the purchase order.
- *GoodsDelivery*: Delivery of the ordered goods.
- *FinePayment*: Payment of a penalty.
- *ReplacementClaim*: Request for the replacement of an unsatisfactory shipment of goods.

- *Refund*: Refund of a previously completed payment.
- *InterestPayment*: Payment of the additional charge of 5% interest on the cost of purchase order, imposed if the payment for the ordered goods is late.
- *HourlyPenalty*: Payment of a penalty that increases by the hour, until the business operation is concluded.

What follows is a significative subset of the EROP rules that are needed to represent a complete electronic contract implementing the contract in natural language described above. Rules have been left out if they are trivial; e.g., removing the purchaser's obligation to pay when the payment is complete, or aborting the business transaction if a purchase order fails. Also not presented is the handling of the possibility of the purchaser first not satisfying the obligation to pay and then not satisfying the obligation to pay with interest; this would be resolved offline in a real life case.

At the beginning of the business interaction, the supplier starts with no obligations and no prohibitions, but with the right to grant premium status to the purchaser (as per clause 3.1), while the purchaser starts with no obligations, no prohibitions, and two rights, for the submission of special and non-special purchase orders, with a deadline of seven days (as per clauses 3.2, 4.1).

The following rule handles a successful, non-special purchase order, and corresponds to clauses 5.2 and 6.1.

```
rule "R1NormalOrder"
  when
    e matches (botype == POSubmission, originator == purchaser,
      outcome == Success)
    POSubmission in purchaser.rights
  then
    supplier.obligs += GoodsDelivery("24h");
    purchaser.obligs += Payment("7d");
end
```

In the rule above, receiving an event for a successful submission of a purchase order has the consequence of imposing an obligation on the supplier to deliver the ordered goods and on the purchaser to pay for those goods. If the purchaser has Premium Customer status (in our model, if a successful business operation *GrantPremiumStatus* has occurred), the conversation implementing the business operation *POSubmission* would have included steps to include the discount owed to Premium Customers.

The following rule, similar to the previous one, handles a successful special purchase order, and corresponds to clauses 5.2 and 6.1.

```
rule "R1SpecialOrder"
  when
    e matches (botype == SpecialPOSubmission, originator == purchaser,
      outcome == Success)
    SpecialPOSubmission in purchaser.rights
  then
    supplier.obligs += GoodsDelivery("24h");
    purchaser.obligs += Payment("7d");
end
```

Rule R2 below derives from clause 6.1, and handles the case of the purchaser not fulfilling his obligation to pay for the ordered goods. In this rule, a successful payment is detected as late if there is a timeout for the obligation to pay in the history of the transaction. This will have the consequence of imposing a new obligation on the purchaser to pay interest on the due amount, with a deadline of one month. This deadline is not present in the original contract fragment; it was added to maintain the notation implementable (an open-ended obligation is not implementable, as discussed earlier in Chapter 2), and also because in real life no supplier would accept the use of such an unspecified, unconstrained penalty.

```
rule "R2"
  when
    e matches (botype == Payment, originator == purchaser,
      outcome == Success)
    Payment in purchaser.obligs
    happened(PaymentTimeout, purchaser, supplier, *, *)
  then
    purchaser.obligs -= Payment;
    purchaser.obligs += InterestPayment("1m");
end
```

Rule R3 below derives from clauses 5.2 and 5.3, and triggers when the ordered goods are delivered by the supplier. Here, the presence of a *GoodsDeliveryTimeout* event in the history of the transaction indicates a late delivery; in this case, a new obligation to pay a penalty is imposed on the supplier, with a deadline of one month. Once again, as in the case of rule R2, this deadline is not present in the original contract fragment, but was added for practical reasons, and to keep the notation implementable.

```
rule "R3"
```

```
  when
    e matches (botype == GoodsDelivery, originator == supplier)
    GoodsDelivery in supplier.obligs
  then
      supplier.obligs -= GoodsDelivery;
      purchaser.rights += ReplacementClaim;
      if (happened(GoodsDeliveryTimeout, supplier, purchaser, *, *))
        supplier.obligs += HourlyPenalty("1m");
      endif
end
```

The following rule, R4, derives from clause 5.1, and triggers when the purchaser requires the replacement of a shipment of goods because of quality issues. In this case, the supplier will have a deadline of three days to execute a new shipment of goods.

```
rule "R4"
  when
    e matches (botype == ReplacementClaim, originator == purchaser,
      outcome == Success)
    ReplacementClaim in purchaser.rights
  then
    supplier.obligs += GoodsDelivery("3d");
end
```

Rule R5 below also derives from clause 5.1, and triggers when the supplier does not satisfy the obligation to deliver a replacement shipment of goods after a purchaser's claim. In this case new obligations to pay a fine and to refund the paid price replace the old one. Deadlines of three days that was not present in the original contract fragment have been imposed on those obligations, as once again it would not be neither realistic nor implementable to leave them open-ended.

```
rule "R5"
  when
    e matches (botype == GoodsDeliveryTimeout, originator == supplier)
    GoodsDelivery in supplier.obligs
    happened(ReplacementClaim, purchaser, supplier,
      timestamp < '3:00', *)
  then
    supplier.obligs -= GoodsDelivery;
    supplier.obligs += Refund("3d");
    supplier.obligs += FinePayment("3d");
end
```

5.2 Scenario: Travel Agency

5.2.1 Scenario

Perrin and Godart present in [19] the following scenario, forming part of a hypothetical contract for the offer and purchase of holiday packages by travel agents:

- The travel agent will offer a proposition to the customer which should be delivered by the 15th December 2003. The proposition can be delivered more than one time before this date.
- The customer should accept one proposition by the 31st December 2003. This can be done only once.
- Payment by credit card is due within seven days after travel selection and acceptance.
- Payment by credit card can be tried twice.
- If payment by credit card fails, another payment mean is accepted, but only one attempt is allowed.
- Items must be sent to the customer within four days after the payment is validated by the bank.

This set of clauses can be rewritten to highlight rights, obligations and prohibitions in this manner:

- **C1:** The travel agent has the right to offer any number of propositions to the customer until the 15th December 2003.
- **C2:** The customer has the right to accept exactly one proposition by the 31st December 2003. Further communications of acceptance will be ignored.
- **C3:** Once a travel proposition has been accepted, the customer has the obligation to pay by credit card within seven days.
- **C4:** Failures of any kind while paying can only occur twice. At the third failure, the customer loses the obligation to pay by card and is obliged to pay by some other means (e.g., bank transfer).

- **C5:** If the payment by other means mentioned above fails in any manner, the transaction is aborted, and both parties lose all pending rights, obligations and prohibitions.
- **C6:** If payment is successful, the travel agent is obliged to deliver the necessary items to the customer within four days.

As can be observed in clause C4, all kinds of failure (initiation, business and technical) are taken into account, both for credit card payments and for payments that use other means.

5.2.2 EROP Version

We can identify two role players, *travelAgent* and *customer*, and the business operations *PropOffer* (the proposal of a holiday package), *PropAccept* (acceptance of a holiday proposal), *CCPayment* (payment by credit card), *OtherPayment* (payment by some non credit card means), *ItemDelivery* (delivery of necessary items, such as ticket, brochures and so on).

Initial attribution of the right to execute *PropOffer* as dictated by clause C1 is set up by rule R1, presented below:

```
rule "R1"
  when
    e matches (botype == Initialization)
  then
    travelAgent.rights += PropOffer("15/12/2003");
end
```

The travel agent's use of the right assigned to him in C1 and the consequences for the customer, established in C2, are handled by rule R2, presented below:

```
rule "R2"
  when
    e matches (botype == PropOffer, originator == travelAgent,
      responder == customer, outcome == Success)
    PropOffer in travelAgent.rights
  then
    customer.rights += PropAccept("31/12/2003");
end
```

The customer's acceptance, regulated by clause C2, and the imposition on the customer of the obligation to pay if the acceptance is valid, regulated by C3, are handled by the following rule, R3:

```
rule "R3"
  when
    e matches (botype == PropAccept, originator == customer,
      responder == travelAgent, outcome == Success)
    PropAccept in customer.rights
    !happened(PropAccept, customer, travelAgent,
      success, *)
  then
    customer.obligs += CCPayment("4d");
    travelAgent.rights -= PropOffer;
end
```

The following rule removes the customer's obligation to pay by credit card and imposes a new one to pay by some other means if payment by credit card fails for the third time, as disposed by clause C4.

```
rule "R4"
  when
    e matches (botype == CCPayment, originator == customer,
      responder == travelAgent)
    e.outcome != Success
    CCPayment in customer.obligs
    (countHappened(CCPayment, customer, travelAgent,
        TecFail, *)
      +countHappened(CCPayment, customer, travelAgent,
        InitFail, *)
      +countHappened(CCPayment, customer, travelAgent,
        BizFail, *) >=2
  then
    customer.obligs -= CCPayment;
    customer.obligs += OtherPayment("7d");
end
```

Clause C5 dictates that a single failed non credit card payment aborts the transaction. Rule R5, presented below, handles this:

```
rule "R5"
  when
```

```
    e matches (botype == OtherPayment, originator == customer,
      responder == travelAgent)
    e.outcome != Success
    OtherPayment in customer.obligs
    !happened(OtherPayment, customer, travelAgent,
      TecFail, *)
    !happened(OtherPayment, customer, travelAgent,
      BizFail, *)
    !happened(OtherPayment, customer, travelAgent,
      InitFail, *)
  then
    customer.obligs -= OtherPayment;
    abort;
end
```

Clause C6 disposes that, if a payment is successful, the travel agent gains the obligation to send the necessary items for travelling to the customer. This is handled by the following two rules, R6-CCPayment and R6-OtherPayment:

```
rule "R6-CCPayment"
  when
    e matches (botype == CCPayment, originator == customer,
      responder == travelAgent, outcome == Success)
    CCPayment in customer.obligs
  then
    customer.obligs -= CCPayment;
    travelAgent.obligs += ItemDelivery("4d");
end

rule "R6-OtherPayment"
  when
    e matches (botype == OtherPayment, originator == customer,
      responder == travelAgent, outcome == Success)
    OtherPayment in customer.obligs
  then
    customer.obligs -= OtherPayment;
    travelAgent.obligs += ItemDelivery("4d");
```

```
end
```

5.3 Rental of Grid node time

5.3.1 Scenario

Gama and Perreira present in [26] Heimdall, a platform to support the definition and enforcement of obligation-based policies. Together with other less detailed scenarios, they introduce one for a Quality of Service agreement policy, where a user pays for five hours of computation time in a grid node to run a simulation. The node is obliged to provide the hours of computation time within 24 hours. If the simulation finishes before that time (that is, if the user releases the resource before the five hours expire), the obligation is considered fulfilled. If the obligation is not fulfilled, the grid node will be penalized by being blacklisted.

This can be expressed by the following set of clauses:

- **C1**: Once the user completes a purchase, the grid node has the obligation to supply access within 24 hours. Failure to do so will give the user the right to blacklist the grid node.
- **C2**: The obligation is considered satisfied when the user releases the access to the grid node, or when the five hours expire.

5.3.2 EROP Version

Two role players can be identified, *user* and *gridNode*, and the business operations *PurchaseQoS* (purchasing a block of five hour computation time under the Quality of Service agreement described earlier on), *GrantAccess* (granting access to the grid node), *ReleaseAccess* (releasing access to the grid node), and *Blacklist* (adding a participant to a blacklist of unreliable parties).

It will be assumed that *ReleaseAccess* can be originated by *user* and by *gridNode*; when user releases access, it is taken to indicate that his simulation is complete. When gridNode releases access, it is taken to indicate that the five hours deadline has expired. It will also be assumed that the power to blacklist is bestowed to the role players, as in the original work it is not clear if this is the case or if this power belongs to a third party.

Clause C1 can be expressed by the following rules:

```
rule "R1"
  when
    e matches (botype == PurchaseQoS, originator == user,
      outcome == Success)
    PurchaseQoS in user.rights
  then
    gridNode.obligs += GrantAccess("24h");
end

rule "R2"
  when
    e matches (botype == GrantAccessTimeout,
      originator == gridNode)
    GrantAccess in gridNode.obligs
  then
    user.rights += BlackList();
end

rule "R3"
  when
    e matches (botype == GrantAccess, originator == gridNode,
      outcome == Success)
    GrantAccess in gridNode.obligs
  then
    gridNode.obligs -= GrantAccess;
    user.rights += ReleaseAccess("5h");
end
```

Clause C2 can be expressed by the following rules:

```
rule "R4"
  when
    e matches (botype == ReleaseAccess, originator == user,
      outcome == Success)
    ReleaseAccess in user.rights
  then
    user.rights -= ReleaseAccess;
    success;
end

rule "R5"
  when
```

```
    e matches (ReleaseAccessTimeout, originator == user)
    ReleaseAccess in user.rights
  then
    user.rights -= ReleaseAccess;
    gridNode.rights += ReleaseAccess();
end
```

5.4 Scenario: Music Streaming Service

5.4.1 Scenario

Pandora[68] is an online music streaming service. Customers can opt to pay a yearly subscription to play music without advertisement, or can listen to music for free, but with advertisement. Currently Pandora is based on a traditional client-server architecture, with the users connecting to the Pandora web service to login and stream music; the logic for the management of the rights, obligations and prohibitions of the parties has been implemented ad-hoc for this system, and is completely controlled by Pandora.

In this section a study for a replacement will be presented for Pandora's in-house rights management system. In a real life case, this would be similar to purchasing an off-the-shelf component; the main advantage for Pandora would be the ability to save the work needed to maintain and update their in-house technology. The main advantage for users would be having their interaction with Pandora verified by a supposedly reliable, neutral third party that does not have an interest in siding with Pandora unfairly.

5.4.2 EROP Version

From the information offered in the Frequently Asked Questions[69], it is possible to model the significant part of the interaction of the Pandora service with its customer with the following set of clauses.

- **C1**: Users have the right to play music using the Pandora streaming service.
- **C2**: Users have the right to purchase an yearly subscription to the Pandora music service. If they do so, they will be able to avoid advertisement.
- **C3**: Pandora has the right to advertise to users that are not currently in a paid subscription, or that let their subscription expire without renewing it.

- **C4**: Pandora will refund users that move to a country where Pandora is not licensed to offer its music service, such as the UK. In these countries its streaming server will be inaccessible.

Two role players can be derived from the scenario description above, *user* and *pandora*. The following business operations can be also defined:

- *PlayMusic*: Connection to the Pandora server in order to play streaming music.
- *Subscribe*: Purchase an yearly subscription.
- *Advertise*: Offer an advertisement to a customer.
- *RequestRefund*: Request the refund of the last yearly subscription.
- *Refund*: Refund the cost of the last yearly subscription.

When first registering, *user* gains the right to execute PlayMusic, and *pandora* gains the right to execute *Advertise*. This is modelled by the following rule, that maps to C1.

```
rule "R1"
  when
    e matches (botype == init)
  then
    user.rights += PlayMusic();
    pandora.rights += Advertise();
end
```

If a user purchases a yearly subscription, rule R2 below is triggered, removing Pandora's right to advertise to him, but adding an obligation to renew the subscription after one year.

```
rule "R2"
  when
    e matches (botype == Subscribe, originator == user,
      outcome == Success)
  then
    pandora.rights -= Advertise;
    user.obligs += Subscribe("12m");
end
```

According to clause C3, if a user does not renew a subscription (that is, if the obligation to subscribe again times out) then the user reverts to a free subscription, and Pandora acquires again the right to advertise. This is handled by rule R3.

```
rule "R3"
  when
    e matches (botype == SubscribeTimeout, originator == user)
    Subscribe in user.obligs
  then
    user.obligs -= Subscribe;
    pandora.rights += Advertise();
end
```

If a user moves outside the US, it will become impossible to access the Pandora servers to play music. In this case playing music will fail, and the user acquires the right to ask for a refund. This is handled by rule R4, which maps to clause C4.

```
rule "R4"
  when
    e matches (botype == PlayMusic, originator == user,
      outcome == TecFail)
    PlayMusic in user.rights
    happened(Subscribe, user, pandora, timestamp < '1y', *)
    !Advertise in pandora.rights
  then
    user.rights += RequestRefund();
end
```

If a refund is requested, then rule R5, which derives from clause C4 above, imposes that Pandora refunds the last yearly subscription. A deadline of 24 hours is added that is not in the clauses above, as it would not have been neither realistic nor implementable not to have one.

```
rule "R5"
  when
    e matches (botype == RequestRefund, originator == user,
      outcome == Success)
    RequestRefund in user.rights
  then
    pandora.obligs += Refund("24h");
end
```

5.5 Discussion

In this section, the EROP language is going to be evaluated in the light of the features of contract specification languages presented and discussed in Subsection 2.2.1.

5.5.1 Declarative Approach

Chapter 3 presented a two level abstract model: a lower level defining the details of the execution of business operations, that relies on the use of one or more B2B messaging protocols, and a higher one defining the consequences of the outcomes of executed business operations. Explicit separation of these concerns allows for a higher level, more declarative approach to contract writing, which is desirable, as discussed in Chapter 2.

Let us consider the first scenario of this chapter, presented in Section 5.1. In the original contract in natural language from [40], as well as in the RuleML version presented in the same work, a significant number of clauses was devoted to define what in our conversion are details of the execution of business operations; even clauses that deal directly with higher level matters have to refer to these lower level details, to the detriment of the declarative level of the notation. The EROP version separates the two abstraction levels, and allows the writer to concentrate only on the flow of operations. We would therefore claim that the approach taken in this work allows a more declarative approach than [40].

In the second scenario of this chapter, presented in Section 5.2, and derived from [19], the original contract is written at a higher level of abstraction compared with the previously mentioned [40], and the abstract model presented in [19] adopts a more declarative approach, focusing on the enforcement of satisfaction of constraints. However, there is no explicit separation of business operation details from the consequences of business operations; this leaves the door open for a less declarative writing style. The separation of abstraction levels of EROP allows the EROP version to maintain, within the context of its model, an approach as declarative as that of [19].

In the third scenario of this chapter, presented in Section 5.3, the original contract segment presented is written at a level of abstraction comparable with the previous one above. The abstract model presented in [26] focuses on the enforcement of constraints on ordering and outcome of events. This approach is similar to the one presented in this work; the separation of abstraction levels in EROP makes it possible for the EROP version to maintain an approach at least as declarative as that of [26].

In the fourth scenario presented here, the "original contract segment" was reconstructed *a posteriori* from the documentation presented on [69], but it is reasonable to expect it to resemble Pandora's actual internal policy, apart from additional information needed to locate it within the US legal context.

Overall, we believe EROP to allow for a high level, declarative style of contract specification, satisfying this criterion.

5.5.1.1 Inheritance

In Subsection 2.2.1, it was suggested that the use of inheritance could be considered as coming under the umbrella of declarative approach. The EROP language, however, does not at the moment offer support for extending and reusing contracts through this mechanism. Its inclusion could be considered a logical step to take in order to pursue a more declarative approach; at one point, the introduction of basic inheritance features for business operations was considered as a potential feature. This would have had an interesting consequence: if a business operation A is defined as a subclass of another operation B, then a role player having the obligation, (or the right, or the prohibition) to execute operation B can satisfy it (or exert the right, or violate the prohibition) by executing operation A – and from the point of view of the CCC, receiving an event notifying the execution of operation A can satisfy the wait for an event notifying for the execution of operation B. Therefore, the definition of an inheritance hierarchy among business operations is mirrored by analogous inheritance hierarchies for events and ROP entities.

The most significant advantage of inheritance in business operations would have been a greater flexibility for writing contracts. Let us assume that the business operation *Purchase Order Submission* (PO Submission) is extended by the operation *Special Purchase Order Submission* (*SPO Submission*). As discussed above, the obligation to execute a *PO Submission* can also be satisfied by executing *SPO Submission.* But the contract in force can be written in such a way that having the obligation to execute *SPO Submission* rather than the normal *PO Submission* has different consequences; for example, discounted prices, or faster delivery times, or *in situ* technical support. This kind of flexibility can enrich significantly the expressive power of a contract written in the EROP language.

However, there are disadvantages as well. First of all, the implementation of such an inheritance mechanism would have required a significant time. Secondly, the availability of this feature would have increased the complexity as well as the flexibility of the EROP language, at the risk of making the language less usable, as it would become harder to learn to use correctly and efficiently, as well as more difficult to verify formally. The introduction of inheritance also introduces a new set of issues to check in a contract; examples of such issues are circular inheritance (A extending B extending C extending A), and the violation of the Liskov Substitution Principle [70] – the principle stating that all semantic properties of objects of a type T should also be shared by all objects of a type S that extends T. Let us consider the example of the business operation *Special Purchase Order Submission* extending *Purchase Order Submission* discussed above. A badly written contract using those two operations could maliciously or involuntarily alter the properties of these by assigning different rights, obligations or prohibitions as the consequences of executing them, thus violating the Liskov Substitution Principle.

A badly written contract could also maliciously or involuntarily remove the semantic differences between parent and child classes, thus rendering the subclassing moot. For example, let's assume that the difference between a *Special Purchase Order Submission* and an ordinary *Purchase Order Submission* is the guarantee of *in situ* technical support; a badly written contract could be construed in such a way that the promised right to technical support is not granted after a successful *SPO Submission.* This would make defining *SPO Submission* useless. This is another issue that ideally should be checked in the formal verification stage.

5.5.2 Implementability

All of the constructs introduced in Chapter 4 and employed here have been shown in Chapter 6 to be either implemented, or at least implementable. Thus the problem of maintaining implementability is avoided by allowing only implementable constructs in the definition of the EROP language. There is no space, for example, for open obligations, because the grammar of the language does not leave space for it. We believe this shows the EROP language to be implementable, satisfy this criterion.

5.5.3 Expressiveness

The Buyer-Seller scenario has been discussed in detail in Chapters 2 and 4, and a representative set of other scenarios have been presented in this chapter. These scenarios are relatively varied, and form a good selection of different business practices and activities, given the stated limit of using functional requirements. (non-functional requirements will be the subject of future investigation, as mentioned in Chapter 7).

While there is no tried and true procedure to verify expressiveness, we believe that, given the stated limit of using functional requirements, the EROP language is expressive enough for most common commercial applications.

5.5.4 Exception Handling

The discussion in Chapters 3 and 4 show the support provided in EROP for reasoning about exceptional situations and for giving contract writers the instruments to resolve them. Chapter 4 shows examples of rules that handle exceptions to give the interacting partners the opportunity to resolve pending issues; the discussion in Chapter 3 shows the complex patterns of interaction it is possible to capture and model with the simple instruments available. We believe this shows EROP provides sufficient support for exception handling.

5.5.5 Usability

As discussed in Section 2.2.1, while the notion of usability being a desirable feature for a contract specification language is intuitive, verifying whether a language is useable is difficult, as there is no tried and ready method to formally determine it. We believe we can make a case for EROP being quite useable, with its declarative approach and relatively intuitive syntax, based on the idea of granting and removing rights, obligations and prohibitions, a notion non-technical people should be able to grasp easily.

5.5.6 Verifiability

Verifiability is a complex topic in its own right, and plenty of work has been produced in this area. Proving that the EROP language is verifiable goes beyond our objectives in this thesis, but it will be the work of future investigation, as discussed in Chapter 7.

Chapter 6

Implementation

The ideas and concepts presented in the previous chapters, and culminating in the model shown in Figure 6.1, have to be appropriately tested and evaluated before working on a full-featured implementation. Therefore we created a scaled down implementation within a simplified context, to reduce the weight of engineering issues that, as important and necessary as they would be in a real life system, would bring small interesting contributions to this work.

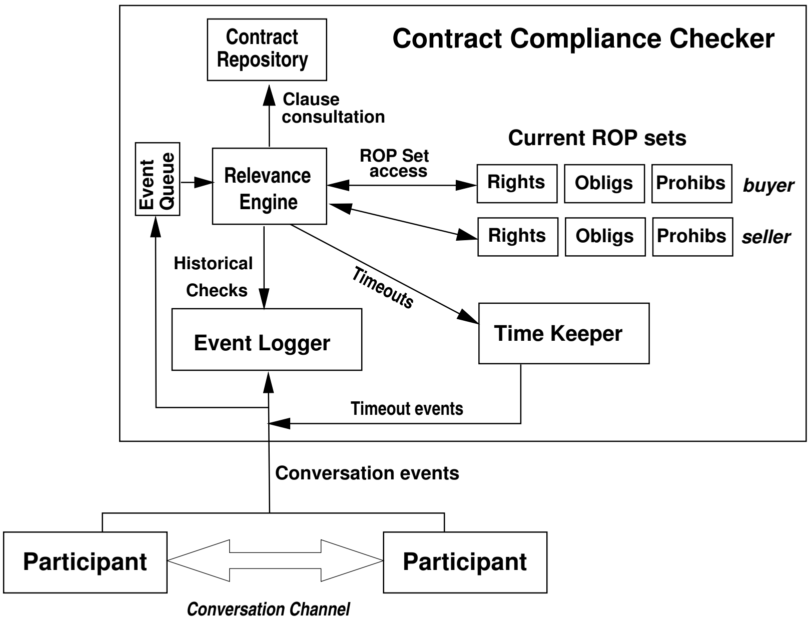

Figure 6.1: Abstract Model of Contract Compliance Checker

A full implementation, for which we present a diagram in Figure 6.2, would be a large and complex piece of software engineering. We therefore decided to opt for a scaled down prototype, making a number of simplifying choices, presented in the next section, and concentrating instead on the novel aspects of our research work.

In the rest of this chapter we are going to present what simplifying choices were made to the

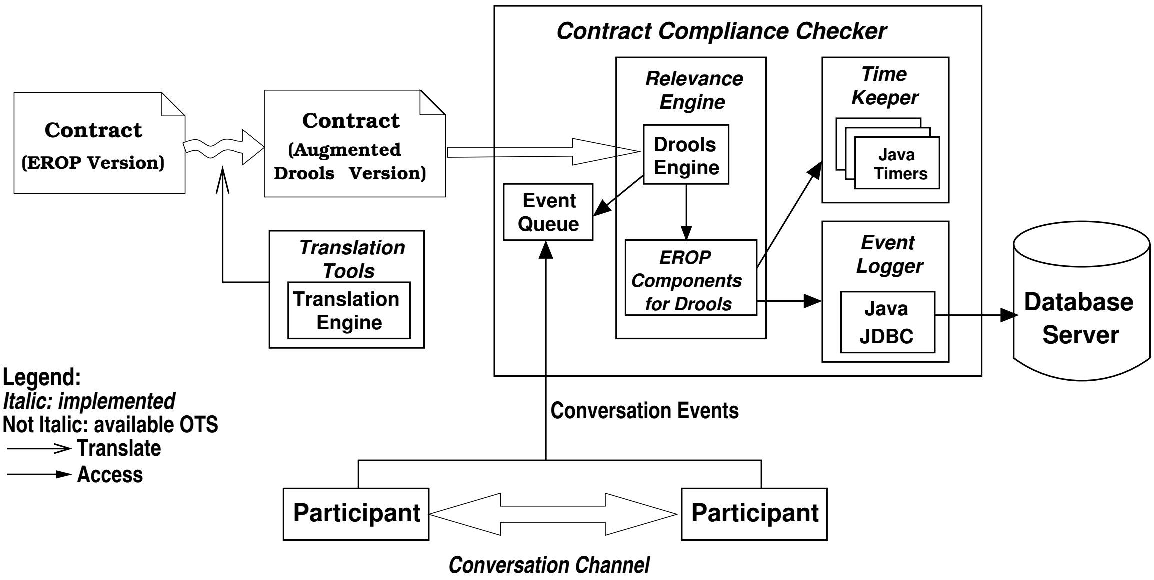

Figure 6.2: Implementation Details for the Contract Compliance Checker

EROP architecture of Fig. 6.2 as it was presented in Chapter 3 in order to obtain our simplified architecture, what implementation choices have been made, how the components of the EROP architecture have been designed and implemented in our experimental prototype, and how the EROP language translates to the language used by the CCC. Material from this chapter appeared in [67].

6.1 Simplifying Choices

6.1.1 Scale

Figure 6.3 presents a diagram for the architecture of our experimental prototype.

In a real life implementation, it would be reasonable to expect that the EROP architecture should be capable of managing any number of contracts for any number of parties, and of instantiating and running any of those contracts any number of times, without interferences between instances. In our prototype, we accomplish this by having a CCC deployed for each individual contract, and having instances of contracts mapped one-to-one to instances of the CCC, so that, for each new contract instance, a new instance of the CCC would be created.

The overall result of this choice is that conversations executed in parallel, even when they are instances of the same contract, do not interfere with each other.

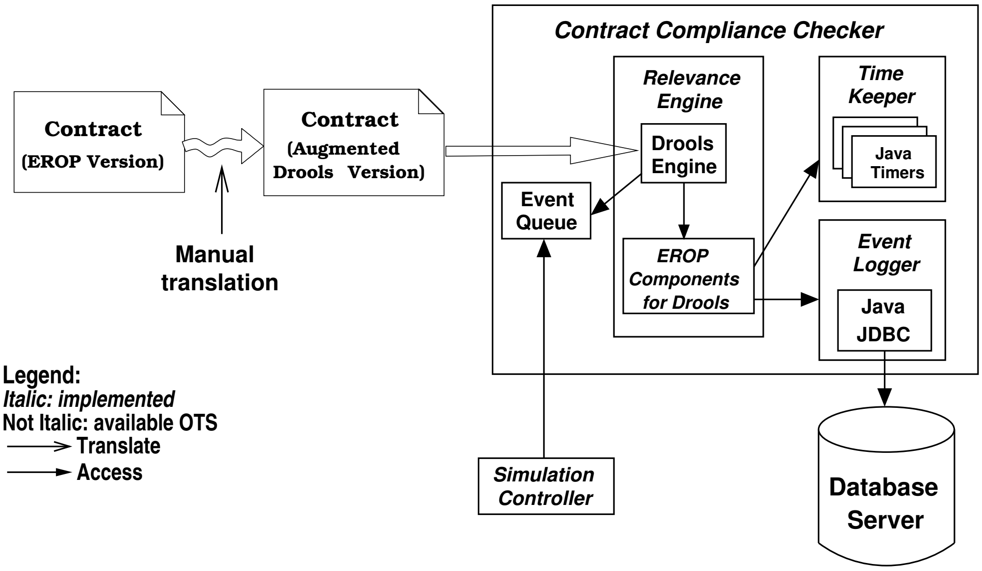

Figure 6.3: Implementation Details for the CCC Prototype

6.1.2 Context

In a realistic implementation the CCC would live within the context of a web service, and, as can be observed in Fig. 6.2, would have access to the Message Oriented Middleware used by the parties to interact. In order to concentrate on the implementation details rather than on the engineering details, we opted to simulate this running environment by feeding a stream of composite events to the CCC. This relieved us of the necessity to write the communication infrastructure of the system and its client side, as well as the synchronization process that generates the composite events.

6.1.3 Language Translation and Checking

A real life EROP system would ship with a complete set of tools for the language as it was presented in Chapter 4, to verify the syntactic correctness of the contracts written by EROP users and to translate it in a form directly usable by the CCC.

We decided to leave work on these for the future in order to reduce the complexity of the work needed, and opted here to present an intermediate language, the mapping of which to EROP on one side and to its implementation on the other will be intuitive, but the details of which do not contribute significantly to this research.

6.2 Implementation Choices

6.2.1 Database Engine

As we anticipated in Chapter 2, and show in Figure 6.2, the Event Logger relies on a database to store historical data reliably and access them efficiently. The database engine chosen for the back end of this version of the EROP architecture is MySQL[71], as it is open sourced, well documented and well performing. However, given reasonable adherence to the SQL:2003[72] standard, any database engine would be suitable for the task, as care was taken to avoid having the implementation being dependent of the particular features of a specific database engine.

6.2.2 Decision Engine

The decision making capability of the CCC is supplied by Drools[73], a rule engine released by JBoss. A rule engine [74] 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 in a conventional system. In this way, specific knowledge is separated from the rest of the execution environment, and segregated in a *rule base*, or *knowledge base* [75], because it could need to be altered more often than the execution environment by the users of the system, to respond to mutated environment conditions. Drools is a *forward chaining*[76] rule engine, where *facts*, items of knowledge that are atomic from the perspective of the system, are constantly added to the system for evaluation, and stored in the system's *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 the ones 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 again, until no rule is triggered.

Drools also allows the definition of *globals*, objects that reside in a special area of the working memory, persist between recognise-act cycles and do not trigger new ones when altered, added or removed. They usually act as hooks to external services, and are therefore the only channel to the outside world a running Drools system has.

To implement our system, we have one global for a reference to the running Relevance Engine, used for housekeeping purposes, and one for a reference to the Event Logger, used to provide access to the historical log. There is also a global for each Business Operation, each Role Player and their ROP Sets; the reasons for this choice are their permanence throughout the life cycle of a contract, and for easier referral in rules, as anything that is not a global must be a fact in working memory, and must be pattern matched to be referred to, thus making the programming style extremely burdensome.

Composite events coming in the Relevance Engine from the Event Queue (after being put there by the Event Composer, or, in our simulation, from a previously prepared schedule of events) are inserted one by one in Drools' working memory, so as to start a recognise-act cycle, which is the means to implement the rule matching algorithm described in Chapter 3

The reason for choosing a rule engine to power the Relevance Engine is the small semantic gap between EROP rules and business rules; as previously discussed in Chapter 4, EROP rules are fundamentally business rules that make use of the EROP ontology. This makes the translation process from EROP to Drools relatively straightforward, as it will be shown further on in this chapter.

The reasons for picking 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[77], a relatively efficient algorithm for searching the rule base and matching it with the working memory, which is the most computationally intensive task in a rule engine. Another notable feature is the possibility to write the consequents of rules (their right hand sides, which in EROP are mainly actions) directly in a programming language (specifically Java, Python and Groovy). This last feature allows a more direct, simpler mapping to the implementation of the EROP ontology.

6.3 Implementation of the EROP Ontology

6.3.1 The EROP Ontology

In Chapter 3 we presented the *EROP ontology*, a set of concepts and of their relationships within the domain of B2B interaction that we employ to model the evolution of interactions between business partners, for the purpose of reasoning about the compliance of their actions with their stated objectives in their agreements. We are going to briefly summarize the EROP ontology here for the convenience of the reader.

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*: an activity defined in the contract for the ultimate purpose of producing value, executed as a shared interaction between two role players using a B2B messaging protocol. Business operations make up the vocabulary of a business contract.
- *Deadline*: A time constraint on the execution of business operations, involved in defining rights, obligations and prohibitions.

- *Right*: A business operation that a role player is allowed to execute. It can have a deadline; if it does not, it is assumed to last until revoked, or until the end of the business partnership.

- *Obligation (simple)*: A business operation that a role player must execute, or face the penalty of being sanctioned. It always has a deadline, because an obligation that does not expire is not enforceable – the obliged party can always claim they will satisfy it at an indeterminate future time– and therefore is meaningless.

- *Prohibition*: A business operation that a role player must not execute, or face the penalty of being sanctioned. Prohibitions are explicitly dealt with in the EROP model, rather than treated as a complement of the rights or as negative obligations.

- *Composite obligation*: A set of business operations a Role Player must execute exactly one of.

- *ROP entity*: A right, obligation or prohibition.

- *ROP set*: A set (possibly empty) of all the rights, obligations and prohibitions in force for a Role Player at a given time. Each Role Player has exactly one of them.

- *Event*: A message carrying information about something happening within the context of the business transaction.

- *Business transaction*: The whole of the execution of a contract, from the moment it starts until its successful or unsuccessful conclusion.

6.3.2 Ontology Implementation

The classes in the EROP ontology listed above map one to one to the Java classes *RolePlayer, BusinessOperation, Right, Obligation, Prohibition, CompositeObligation, ROPEntity, Deadline, ROPSet* and *Event*, which implement the operations described in Chapter *Model.* The class *ROPEntity* has been defined as the parent of classes *Right, Obligation* and *Prohibition*, and the ancestor of *CompositeObligation.* The relationships between those classes are shown in the UML diagram presented in Figure 6.4.

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

6.4 The Contract Compliance Checker

The main components of the Contract Compliance Checker were identified in Chapter 3: the Event Queue, the Time Keeper, the Event Logger and the Relevance Engine. The Event Queue, defined in the class *EventQueue*, is implemented as a First In, First Out queue of Event objects, owned by

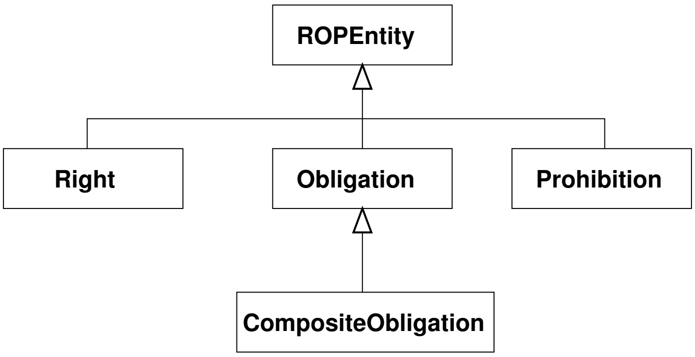

Figure 6.4: Descendants of the class ROPEntity

the Relevance Engine; incoming Events are stored in this structure, that acts as a buffer. The Event Queue offers two operations:

- adding an Event to the end of the queue;
- taking an Event out of the head of the queue.

Events are added by the Event Composer (which in our prototype is simulated), or by the TimeKeeper if they are timeouts. Only the Relevance Engine takes Events out of the queue.

The Time Keeper, defined in the class *TimeKeeper*, manages the deadlines for the expiry of ROP Entities, and offers two operations:

- adding a deadline;
- removing a deadline.

Deadlines are internally represented as Java Timer objects, and are stored in a hash table, indexed by a key constructed out of the name and type of the ROP Entity they refer to, and the involved role players. Whenever a deadline expires, its corresponding Java Timer object notifies the TimeKeeper object, passing as parameters the relevant data (Business Operation type, relevant Role Players, and so on). The TimeKeeper object then instantiates a new Event of the relevant type, appending to the name *Expiry* if the deadline was for the expiry of a ROP Entity, or *Timeout* if the deadline was for a pending timeout. The status of the new Event object is set to *timeout.*

The Event Logger maintains a connection to the historical database and offers two operations:

- logging events in the database;
- querying the database.

The way the Event Logger works will be discussed in more detail in the next Section.

The Relevance Engine relies on an instance of the Drools rule engine 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;
- verifying that the Event queue is empty.

We presented the algorithm followed by the Relevance Engine in Chapter 3:

1. Receive an Event e from the Event Composer;
2. Analyse the contract repository and identify relevant rules for e;
3. For each relevant rule r, execute the actions listed in its right hand side.

This algorithm is implemented in this manner. The Relevance Engine is sent a signal every time a new Event is added to the queue; eventually, the RE takes the first Event object in the queue and feeds it to the instance Drools engine. It is this last component that matches the Event and its context (history, etc.) to the rule base, identifies the relevant rules and executes their right-hand-side actions.

6.5 The Historical Database

6.5.1 Tables

The Historical Database contains four tables:

- a table for the Role Players;
- a table for the relevant Event types;
- a table for the possible status outcomes;
- a table 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. Let us think for example of a clause that provides a 10% discount to buyers with at least three successfully completed purchase orders that were paid on time. Therefore we make no special provision to erase the content of the Event history, in order to leave the choice to do this to the involved parties.

6.5.2 Historical Queries

All historical queries can be classified into two main categories: *boolean* queries, verifying whether an Event verifying a set of constraints exists, and *numeric* queries that count the number of Events verifying a set of constraints.

In either case, the set of constraints is the same: the originating and responding Role Players, the Event type, 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 = "before 10am of 10 February 2008"*. In our prototype, however, we assume that *timeConstraint* is expressed as a valid SQL expression, for the sake of implementation simplicity; the temporal constraint shown earlier would therefore be the string *timestamp <'10/02/2008 10:00:00'*.

The given set of constraints is then used to build a SQL query to submit to the database server. If the historical query is boolean, a SELECT SQL statement is used; if the historical query is numeric, a SELECT COUNT statement is used instead. The answer of the database server is then analysed to generate the response to the historical query.

Let us examine two examples that need historical queries: two similar rules that could be added to the EROP version of the Buyer-Seller example shown in Chapter 4, one to give the Buyer the right to submit purchase orders at a discounted price if the Buyer successfully paid any number of purchase orders since January 2008, and one to give the Buyer the same right if he successfully paid at least three purchase orders during the same time interval.

The first rule requires a boolean query, and should be triggered right after a purchase order has been successfully paid. The EROP version of the rule is:

```
rule "GrantRightToDiscountedPurchase1"
  when
    e matches (botype==Payment, originator == buyer,
      responder == seller, outcome == Success)
    happened(Payment, buyer, seller, Success,
      timestamp >= '1 Jan 2008')
  then
      buyer.rights -= POSubmission;
      buyer.rights += DiscountedPOSubmission;
end
```

The second rule requires a numerical query, and should be triggered right after a purchase order has been successfully paid like the previous one. The EROP version of this rule is:

```
rule "GrantRightToDiscountedPurchase2"
```

```
  when
    e matches (botype==Payment, originator == buyer,
      responder == seller, outcome == Success)
    countHappened(Payment, buyer, seller, success,
      timestamp >= '1 Jan 2008') >= 3
    then
      buyer.rights -= POSubmission;
      buyer.rights += DiscountedPOSubmission;
end
```

The historical queries in the examples above map respectively to calls to the methods *happened()* and *countHappened()* of the class *EventLogger* after translation. These methods build the SQL queries for the historical database from the received parameters, submit them to the database server, then parse the results and return them. The first historical query, a boolean one, is translated into the SQL query

```
SELECT * FROM eventhistory WHERE type='Payment'
  AND originator='buyer' AND responder='seller'
  AND timestamp >= '1 Jan 2008' AND status='success'
```

If the result set is empty, the *happened()* method will return false; if it is not empty, it will return true, as there will be at least one occurrence of an event within the desired constraints. The second example query, a numerical one, is translated into the SQL query

```
SELECT COUNT(*) FROM eventhistory WHERE type='Payment'
  AND originator='buyer' AND responder='seller'
  AND timestamp >= '1 Jan 2008' AND status='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. This number is then returned by the method *countHappened()*.

6.6 Translation

6.6.1 Augmented Drools

The Java implementation of the EROP ontology presented earlier in this Chapter extends the rule language offered by the Drools rule engine, adding Java constructs to reason about and to manipulate the ROP sets of the business partners. We call this extended language *Augmented Drools* (AD).

Because of its origin, Augmented Drools is more verbose than the EROP language, and also less abstract and human-readable, and more Java-like in style, especially in the notation for actions in rule right hand sides. It also needs to have additional lines of code for housekeeping purposes, that are necessary for the Java-based EROP ontology, but are not necessary for a human reader, such as lines to instantiate and assign objects and arrays.

The EROP language maps completely into Augmented Drools, so that it is possible to write contracts in Augmented Drools with the same expressive power of EROP, but, as mentioned above, are more verbose, more implementation-aware yet less declarative in style and less rich in syntactic sugar[78], but, most importantly, can be run directly on available software - the Drools rule engine.

The problem of creating an appropriate implementation for the EROP language therefore becomes a problem of translating EROP to Augmented Drools. We will show in the rest of this section how EROP statements map into Augmented Drools statements.

6.6.2 Definitions in Augmented Drools and EROP

A rule file in Augmented Drools, like one in EROP, starts with the definitions of the objects and entities used in the file. After the Java statements to import the EROP ontology classes, there is a section to define global identifiers, such as Role Players, Business Operations and Composite Obligations. AD also needs instances of all EROP Ontology classes to be declared here, so there must 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 EventLogger logger;
global RolePlayer buyer;
global RolePlayer seller;
global ROPSet ropBuyer;
global ROPSet ropSeller;
global BusinessOperation purchaseOrder;
global BusinessOperation finePayment;
global BusinessOperation payment;
global BusinessOperation poAcceptance;
global BusinessOperation poRejection;
global BusinessOperation goodsDelivery;
```

Here we declare, in order, the instances of the Relevance Engine and Event Logger to use, the two Role Players *buyer* and *seller*, their two ROP Sets, and the Business Operations *Purchase Order, Fine Payment, Purchase Order Acceptance, Purchase Order Rejection* and *Goods Delivery* (Business Operation names start in lowercase here because they are Java object and follow Java style rules, and *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*** *conditions* ***then*** *actions* ***end***. Rule names must not be duplicated.

Comments in AD, like in Drools, are preceded by a hash sign (#), and continue until the end of the line.

6.6.3 Rule conditions in EROP and Augmented Drools

The triggering conditions in the left hand sides of EROP rules must include one event match, and can include zero or more of constraints on event attributes, historical queries, or constraints on ROP sets.

Event matching is done in EROP with the syntax *e* ***matches*** *(attribute == value, [attribute == value]*)*, where *e* is a placeholder variable and *attribute* is the name of an Event attribute - one of originator, responder, outcome, timestamp and so on. This translates to the Augmented Drools syntax *$e: Event (attribute == value, [attribute == value]*)*, where the placeholder event variable is indicated with $*e*.

Constraints on event attributes can be imposed outside the event match too, to express constraints different from equality, but also for purposes of expressivity. They can be temporal or Boolean comparisons. All of these constraints map to AD statements that require the use of the *eval* construct, supplied by Drools to evaluate boolean expressions in the left hand side of rules.

Temporal constraints follow the syntax *e* ***before—after*** *time* and are used to impose that the event has occurred respectively before or after the specified time. These map to AD expressions of the form ***eval****(e.before(time))* or ***eval****(e.after(time))*.

Constraints on event attributes can also be Boolean comparisons, such as, for example, checking if the originator is different from *buyer*. These constraints follow the syntax *e.attribute operator value*, where *operator* is one of the usual Java Boolean operators (==, !=, ≥, ≤, and so on). These map to AD expressions of the form ***eval****(e.getAttribute operator value)*.

Historical queries in EROP are introduced by the keyword *happened*, or by the keyword *counthappened* if a numeric result is desired rather than a boolean one. In AD, they require the use of the *eval* construct, supplied by Drools to evaluate boolean expressions in the left hand side of rules, and of a method call to an *EventLogger* instance, so that the expression

```
happened(businessOperation, originator, responder,
  status, timeConstraint)
```

would become

```
eval(eventLogger.happened(businessOperation, originator,
  responder, status, timeConstraint))
```

where *eventLogger* is the running instance of the class *EventLogger*.

The presence or absence of a Business Operation in a Role Player's ROP Set is evaluated in EROP with

```
BOType in rolePlayer.ROPSet
```

where ROPSet is one of *rights, obligations* or *prohibitions*. This expression translates to AD as one of

```
eval(playersROPSet.matchesRights(BOType))
eval(playersROPSet.matchesProhibitions(BOType))
eval(playersROPSet.matchesObligations(BOType))
```

depending on whether the ROP Set referred in the EROP expression is the set of rights, prohibitions or obligations. Note once again the use of the *eval* construct to evaluate a boolean method call.

6.6.4 Actions in EROP and Augmented Drools

Two kinds of actions can appear in the right hand sides of EROP rules: manipulations of participants' ROP sets, and terminations of the current contract instance.

Manipulation of the ROP sets is done in EROP with the C++-inspired += and -= operators, like this:

```
seller.obligs += Invoice("24h");
buyer.rights -= POSubmission;
buyer.prohibs += GoodsDelivery;
```

In AD modifications to a role player's ROP Set need to go through method calls, provided, as mentioned above, by the class *ROPSet*, so that the previous example translates to AD as

```
ropSeller.addObligation("Invoice", buyer, "24h");
ropBuyer.removeRight("POSubmission");
ropBuyer.addProhibition("GoodsDelivery", null);
```

In the case of composite obligations, an extra line of code is needed to add a new one. Here is an example that adds the composite obligation *React To Purchase Order*, composed out of the Business Operations *PO Acceptance* and *PO Rejection*, to the obligation set of the Role Player *seller*:

```
BusinessOperation[] bos = poAcceptance, poRejection;
ropSeller.addObligation("React To Purchase Order", bos, buyer, 3);
```

The first line of code defines an array of Business Operation objects, which are then used by the method *addObligation* of the class *ROPSet* to define the new composite obligation.

In EROP the execution of a contract is concluded using the keyword *terminate*, which takes an argument, the outcome of the execution. In a manner similar to the execution of a business operation, the outcome can be *Success*, *TecFail*, *BizFail* or *InitFail*, respectively for a successful conclusion, a failure caused by technical problem, one caused by business issues, or a failure in the initiation of the contract execution. Note that a successful conclusion does not necessarily imply that goods or services have been exchanged, but only that the interaction can be closed without any dispute that needs to be resolved offline. For example, in our Buyer-Seller scenario, if the seller decides to reject a buyer's purchase order, the transaction is considered to have concluded successfully, as there are no pending disputes to solve offline.

The EROP keyword *terminate* maps to the AD statement *engine.conclude()*, which takes as a parameter a string representing the outcome of the contract execution; *engine* is the current reference to the Relevance Engine. This method concludes the currently running contract instance and notifies its participants of the termination and of its outcome.

6.6.5 Conditional structures

In Chapter 4 we presented two conditional structures: the *if-then-else-endif* statements and the *status guards*. Both are a form of syntactic sugar, that do not alter substantially the language's functionality but make it "sweeter" for humans to use, allowing a writing style that is more natural and more productive.

The *if-then-else* statement is used, like in ordinary programming, to allow conditional execution of actions in the right hand side of rules, depending on the value of a boolean expression. As an example, the *if-then-else-endif* statement allows to rewrite the two rules

```
rule "Rule1"
  when
    e matches (botype == SomeBO, ...)
    booleanConditions
  then
    actionBlock1
```

```
end


rule "Rule2"
  when
    e matches (botype == SomeBO, ...)
    !booleanConditions
  then
    actionBlock2
end
```

as the single rule

```
rule "RuleIfThen"
  when
    e matches (botype == SomeBO, ...)
  then
    if booleanConditions
    then
      actionBlock1
    else
      actionBlock2
    endif
end
```

It is also possible to not use the *else* keyword, and in this case no action block in the *if-then-endif* statement will be executed if its condition is not triggered.

The use of *if-then-else-endif* statements, as mentioned above and explained in more detail in Chapter 4, allows for a more productive and comfortable rule writing style. The translation of rules with an *if-then-else-endif* block to AD, however, goes in the opposite direction of the rewriting illustrated above: the EROP rule *RuleIfThen* shown above maps to the AD versions of *Rule1* and *Rule2* above. In general, a rule in the EROP language containing an *if-then-else* statement is mapped to two rules in AD, one with the *if*-condition added to the *when*-condition set and the *then*-action block added to the right hand side of the AD rule, and another with the negated *if*-condition added to the *when*-condition set and the *else*-action block added to the right hand side of the second AD rule. This second AD rule is not generated for an *if* statement without an *else* part. Status guards are more specialized statements for conditional control of action execution according to an event's outcome. They are the keywords *Success, TecFail, BizFail, InitFail, Other*, that are used to identify action blocks that have to be executed in the case the event under scrutiny has an outcome status respectively of successful, technical failure, business failure, initiation failure, or any one that was not covered in the same rule. The use of status guards therefore allow to rewrite a set of rules such as this:

```
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 "RuleForOtherOutcomes"
  when
    e matches (botype == SomeBO)
    ((e.outcome != Success)||(e.outcome != TecFail))
  then
    actionBlock3
end
```

as the following single rule:

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

```
end
```

The translation to AD, like for the more general conditional *if-then-else*, goes in the opposite direction compared with the rewriting shown above. The EROP rule *RuleForAllOutcomes* shown above, for example, would map to the three AD equivalents of the EROP rules *RuleForSuccess*, *RuleForTechnicalFail* and *RuleForOtherOutcomes* shown previously. In general, 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 check on the outcome of the event under scrutiny added to its *when*-condition matching the corresponding status guard.

6.7 Buyer-Seller Contract in Augmented Drools

In this Section we are going to show how some significant clauses of the EROP version of the Buyer-Seller contract, presented in Chapter 4, maps to Augmented Drools.

For the convenience of the reader, here is the text of the contract in English language:

- **C1:** The buyer has the right to submit a Purchase Order (PO) from Monday to Friday, between 9 am and 5 pm.
- **C2:** The seller is obliged to either accept or refuse the PO within 24 hours. Failure to satisfy this obligation will abort the business transaction for an offline resolution.
- **C3:** If the PO is accepted, the seller is obliged to submit an invoice within 24 hours. If the PO is rejected, the transaction is considered concluded.
- **C4:** After receiving an invoice, the buyer is obliged to respond to the invoice within seven days, either cancelling the PO or paying the due amount. Failure to satisfy the obligation will abort the business transaction for offline resolution.
- **C5:** Cancellation of a PO by the buyer eliminates all obligations imposed on the seller and the buyer and concludes the business transaction. If a payment has been received before the cancellation, however, the seller has the obligation to refund it.
- **C6:** Once payment is received, the seller is obliged to ship the goods between seven 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 PO.
- **C8:** Buyer and seller are obliged to stop the execution of the business transaction upon the detection of three failures to execute the payment. Possible failures shall be sorted offline.

Of the rules derived from these clauses, we are going to present the translation to AD of rules R1, R2Acceptance and R4. For the convenience of the reader, here are the EROP versions of these rules.

```
rule "R1"
  when
    e matches (botype == POSubmission)
  then
    Success:
       e.originator == buyer
        && POSubmission in buyer.rights
        && e.weekday in [Mon ... Fri]
        && e.time in [9 ... 17]
      then
        seller.obligs += RespondToPO("24h");
      endif
    Otherwise:
      pass;
end

rule "R2Acceptance"
  when e matches (botype == POAcceptance)
  then
    Success:
      if e.originator == seller
        && RespondToPO in seller.obligs
      then
        seller.obligs -= RespondToPO;
        seller.obligs += Invoice("24h");
      endif
    Otherwise:
      pass;
end

rule "R4"
  when e matches (botype == Invoice)
  then
    Success:
      if e.originator == seller
        && Invoice in seller.obligs
      then
        seller.obligs -= Invoice;
        buyer.prohibs -= POCancellation;
        buyer.obligs += RespondToInvoice("7d");
      endif
    Otherwise:
      pass;
end
```

Rule R1 maps to the two following AD rules:

```
rule "R1Success"
  when $e: Event (type == "POSubmission", e.originator == "buyer" )
    eval(e.getOutcome() == "Success")
    eval(e.getWeekday() <= 5)
    eval((e.getHour() < 17) && (e.getHour >= 9))
  then
    BusinessOperation[] bos = poAcceptance, poRejection;
    ropSeller.addObligation("RespondToPO", bos, buyer, "24h");
end

rule "R1Otherwise"
  when $e: Event (type == "POSubmission")
    eval(e.getOutcome() == "success")
  then
    pass;
end
```

Rule R2Acceptance maps to the two following AD rules:

```
rule "R2AcceptanceSuccess"
  when $e: Event (type == "POAcceptance", originator == "seller")
    eval(ropSeller.matchesObligations("RespondToPO"))
    eval(e.getOutcome() == "success")
  then
    ropSeller.removeObligation(RespondToPO);
    ropSeller.addObligation(Invoice, buyer, "24h");
```

```
end


rule "R2AcceptanceOtherwise"
  when $e: Event (type == "POAcceptance")
    eval(e.getOutcome() != "Success")
  then
    pass;
end
```

Rule R4 maps to the two following AD rules:

```
rule "R4Success"
  when $e: Event (type == "Invoice", e.originator == "seller")
    eval(ropSeller.matchesObligations("Invoice"))
    eval(e.getOutcome() == "Success")
  then
    ropSeller.removeObligation(Invoice);
    ropSeller.removeProhibition(POCancellation);
    ropBuyer.addObligation(RespondToInvoice, seller, "7d");
end


rule "R4Otherwise"
  when $e: Event (type == "Invoice")
    eval(e.getOutcome() != "Success")
  then
      pass;
end
```

6.8 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 [77]. The performance of the Rete algorithm is determined by two factors: the number of facts in the working memory, and the characteristics of the rule base. In the EROP model, only one fact is evaluated at a time, and so in our case 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 many triggering conditions (the ones on the left hand sides of rules) are shared by its rules. When it comes to the size of the rule base, the time needed for a recognize-act cycle grows as the number of rules grows; the effects of the size of a rule base on performance are discussed in [79].

The Rete algorithm uses a dataflow network to represent the left hand side conditions of the rules. Each condition appearing in the left hand side of a rule is associated with one of the non-root nodes in the network, so that a complete left hand side of a rule corresponds to a path from the root node to a leaf node. This network is traversed during the execution of the recognize-act cycle. The style used to write rules has an effect on the network generated at runtime, and therefore on execution times. Rules with common conditions share nodes in this network; the more conditions are shared, the more nodes are shared, and the more efficient the execution of 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 duplication among 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.

Chapter 7

Conclusions

This thesis presented the most prominent components for EROP, a system for the controlled execution of electronic business contracts. EROP stands for *Events, Rights, Obligations* and *Prohibitions*; this acronym comes from our fundamental principle of observing the events generated by the role players' execution of business operations, and of monitoring contract compliance by matching those events against the rights, obligations and prohibitions of the role players. EROP includes a language for the specification of electronic contracts, presented in Chapter 4. The main feature of the EROP language is the direct manipulation of the sets of rights, obligations and prohibitions using an intuitive, easy to understand syntax. The EROP language is underpinned by the EROP ontology presented in Chapter 3, a collection of classes within the domain of B2B interaction used to model the evolution of the execution of an electronic contract and to reason about the compliance of the role players' actions with the clauses of the contract.

Contracts are written in the EROP language for two purposes: dictating the changes to the ROP sets of the role players as a response to the captured events, and specifying when business operations can be considered contract compliant. Both of these tasks are carried out by the Contract Compliance Checker; the architecture of the CCC was presented in Chapter 3, and its implementation was discussed in Chapter 6.

Chapter 5 has been dedicated to an evaluation of the capabilities of the EROP language. In order to do so, we have presented electronic contracts written in EROP for a selection of relevant and interesting scenarios. We then discussed them within the context of a list of desirable features for a contract specification language such as EROP that was presented in Chapter 2, together with a discussion on the relevant areas of research that cover the creation of a system for the specification and execution of electronic contracts such as EROP, and a presentation of other relevant work in the same areas.

7.1 Future Work

It has been said that in science, a good question is one that generates more questions besides its own answer; that has been certainly the case of our work on EROP. Here follows a list of such questions – research areas that we will touch upon in the future.

Verification: Contracts are written by humans, and even skilled humans make mistakes; therefore, electronic contracts have to be carefully verified before they can be used. Contracts have to be *correct*, in the sense that they must comply with the intended spirit of the business partnership, and be free from unintended loopholes; they also should be *complete*, that is, capable of dealing with any possible foreseen or unforeseen event, or at least as close to completeness as possible. Contracts should also be *consistent*, free from rules mandating conflicting outcomes to the same event, as well as *efficient*, both in the sense of being written using the smallest possible number of clauses (for example, removing unreachable and duplicate clauses), and in the sense of allowing the interacting partners to achieve their intended purpose by executing the smallest possible number of business operations. Future work will have to focus on investigating algorithms and techniques, such as those discussed in [80], to verify these properties for EROP contracts.

Extension of the capabilities of the EROP language and model: Currently EROP is designed only for the specification and compliance monitoring of functional requirements (introduced in Chapter 1), that refer to the terms and conditions expressing what business operations are permitted, obliged and prohibited to execute, as well as stipulate when and in what order the operations can be executed. However, in order to make EROP more generally useful in current practice, it should be possible to to express in an EROP contract non-functional requirements (also introduced in Chapter 1), Service Level Agreements asserting commitments to maintain a certain level of Quality of Service.

Trust: Currently EROP is designed to operate under the assumption that the interacting business partners operate in good faith, with no reason to behave maliciously – that is, to generate unreliable or incorrect events or to willingly ignore a running business transaction. This issue, together with other related issues, such as guaranteeing of authenticity and non-repudiation, could be addressed by integrating in the EROP system a non-repudiable interaction mechanism, such as the one described in [81].

Translation: In Chapter 6 we discussed how to translate EROP rules into rules written in an augmented version of Drools. Currently, this translation has to be executed by hand; preliminary work has begun on an automated translator, and in the near future, the completion of such a translator will be one of our targets.

Bibliography

[1] P. Hulme and L. J. Jordanova. *The Enlightenment and its Shadows.* Taylor and Francis, 1990. ISBN 0-41-5042313.

[2] H. Wehberg. Pacta Sunt Servanda. *American Journal of International Law*, page 775, Oct 1959.

[3] A. Picot, R. Reichwald, and R. Wigand. *Information, Organization and Management.* Springer, 2008.

[4] A. Ruff. *Contract Law.* Sweet and Maxwell, second edition, 1999.

[5] J. W. Salmond and J. Williams. *Principles of the Law of Contracts.* The Carswell Company (Ltd), second edition, 1945.

[6] A. Sullivan. *Economics: Principles in Action.* Pearson Prentice Hall, 2003. ISBN 0-13-063085-3.

[7] P.. Grefen and S. Angelov. E-business track on tau-, mu-, pi-, and epsilon-contracting. In *CAiSE '02/ WES '02: Revised Papers from the International Workshop on Web Services, E-Business, and the Semantic Web*, pages 68–77, London, UK, 2002. Springer-Verlag.

[8] D. Chaffey. *E-business and E-commerce Management: Strategy, Implementation and Practice.* Financial Times/Prentice Hall, 2007.

[9] P. Chakravarty and M.P. Singh. Incorporating Events into Cross-Organizational Business Processes. *IEEE Internet Computing*, 12(2):46, 2008.

[10] P.R. Krishna and K. Karlapalem. Electronic Contracts. *IEEE Internet Computing*, 12(4):60–68, 2008.

[11] J. Xu, A. Romanovsky, and B. Randell. Concurrent Exception Handling and Resolution in Distributed Object Systems. *IEEE Transactions on Parallel and Distributed Systems*, 11(10):1019–1032, October 2000.

[12] P. Greenfield, A. Fekete, J. Jang, and D. Kuo. Compensation is not Enough. In *Proc. 7th IEEE Int'l Enterprise Distributed Object Computing Conf. (EDOC'03)*, pages 232–239. IEEE CS, 2003.

[13] S. Angelov and P. Grefen. The 4W Framework for B2B E-contracting. *International Journal of Networking and Virtual Organisations*, 2(1):78–97, 2003.

[14] S. Angelov and P. Grefen. A Conceptual Framework for B2B Electronic Contracting. In *Collaborative Business Ecosystems and Virtual Enterprises: IFIP TC5/WG5. 5 Third Working Conference on Infrastructures for Virtual Enterprises (PRO-VE'02) May 1-3, 2002, Sesimbra, Portugal*, pages 143–150. Kluwer Academic, 2002.

[15] S. Angelov and P. Grefen. A Framework for Analysis of B2B Electronic Contracting Support. CTIT Technical Report series 01-38, 2001.

[16] Crossflow project. `http:\\www.crossflow.org`, 2000.

[17] 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.

[18] D. Luckham. *The Power of Events: an Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002.

[19] 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.

[20] O. Perrin and C. Godart. A Model to Support Collaborative Work in Virtual Enterprises. *Data Knowl. Eng.*, 50(1):63–86, 2004.

[21] 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.

[22] N. Minsky. Law Governed Interaction (LGI): A Distributed Coordination and Control Mechanism (An Introduction, and a Reference Manual). `http://www.cs.rutgers.edu/minsky/papers/manual.pdf`, 2005.

[23] N. Minsky. The Imposition of Protocols Over Open Distributed Systems. *IEEE Transactions on Software Engineering*, 17(2):183–195, 1991.

[24] N. Minsky and V. Ungureanu. Scalable Regulation of Inter-enterprise Electronic Commerce. *Electronic Commerce: 2nd Int'l Workshop*, November 2001.

[25] P. Gama and P. Ferreira. Obligation Policies: an Enforcement Platform. In *Sixth IEEE International Workshop on Policies for Distributed Systems and Networks, 2005*, pages 203–212, 2005.

[26] 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.

[27] C. Ribeiro, A. Zuquete, P. Ferreira, and P. Guedes. SPL: An Access Control Language for Security Policies with Complex Constraints. In *Proceedings of the Network and Distributed System Security Symposium*, pages 89–107, 2001.

[28] P.F. Linington and S. Neal. Using Policies in the Checking of Business to Business Contracts. In H.Lutfiyya, J.Moffat, and F.Garcia, editors, *Fourth IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 207–218. IEEE Computer Society, June 2003.

[29] P. F. Linington, Z. Milosevic, J. Cole, S. Gibson, S. Kulkarni, and S. Neal. A Unified Behavioural Model and a Contract Language for Extended Enterprise. *Data and Knowledge Engineering*, 51(1):5–29, October 2004.

[30] P. F. Linington. Automating Support for E-Business Contracts. *International Journal of Cooperative Information Systems*, 14(2-3):77–98, September 2005.

[31] OMG MDA Guide Version 1.0.1, omg/2003-06-01 ed. The Object Management Group (OMG), June 2003.

[32] Z. Milosevic. *Enterprise Aspects of Open Distributed Systems.* PhD thesis, Computer Science Department, The University of Queensland, October 1995. PhD Thesis.

[33] Z. Milosevic and A. Bond. Electronic Commerce on the Internet: What Is Still Missing? In *In Proc. of the 5th Conf. of the Internet Society*, 1995.

[34] Z. Milosevic, D. Arnold, and L. O'Connor. Inter-enterprise Contract Architecture for Open Distributed Systems: Security Requirements. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 1996. Proceedings of the 5th Workshop on*, pages 68–73, 1996.

[35] G. Spanoudakis and K. Mahbub. Non Intrusive Monitoring of Service Based Systems. *Int'l Journal of Cooperative Information Systems*, 15(3):325–358, 2006.

[36] K. Mahbub and G. Spanoudakis. Run-time Monitoring of Requirements for Systems Composed of Web-services: Initial Implementation and Evaluation Experience. In *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on*, volume 1, pages 257–265, July 2005.

[37] M.B. Juric. *Business Process Execution Language for Web Services BPEL and BPEL4WS 2nd Edition.* Packt Publishing, 2006.

[38] M. Shanahan. The Event Calculus Explained. *Lecture Notes in Computer Science*, 1600:409–430, 1999.

[39] RuleML. The RuleML Markup Initiative. `http:\\www.ruleml.org`, Feb 2005.

[40] G. Governatori. Representing Business Contracts in RuleML. *International Journal of Cooperative Information Systems*, 14(2-3):181–216, 2005.

[41] D. Nute. Defeasible Logic. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, pages 353–395. Oxford University Press, 2001.

[42] Stanford Encyclopedia of Philosophy. Deontic Logic. `http://plato.stanford.edu/entries/logic-deontic/`, Feb 2006.

[43] G. Governatori and A. Rotolo. Defeasible Logic: Agency, Intention and Obligation. *Lecture Notes in Computer Science*, 3065:114–128, 2004.

[44] G. Governatori and A. Rotolo. Logic of Violations: A Gentzen System for Reasoning with Contrary-to-Duty Obligations. *Australasian Journal of Logic*, 4:193–215, 2006.

[45] The Java Rule Engine API. `http://jcp.org/en/jsr/detail?id=094`.

[46] H. Ludwig and M. Stolze. Simple Obligation and Right Model (SORM) for the Runtime Management of Electronic Service Contracts. *Lecture Notes in Computer Science*, 3095:62–76, 2004.

[47] H. Ludwig, A. Dan, and R. Kearney. Cremona: an Architecture and Library for Creation and Monitoring of WS-Agreements. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 65–74, New York, NY, USA, 2004. ACM.

[48] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web Services Agreement Specification (WS-Agreement). In *Global Grid Forum*, 2004.

[49] The CONTRACT Project. `http:\\www.ist-contract.org`.

[50] S. Miles, N. Oren, M. Luck, S. Modgil, N. Faci, C. Holt, and G. Vickers. Modelling and Administration of Contract-based Systems. In *Proceedings of the AISB 2008 Symposium on Behaviour Regulation in Multi-agent Systems*, pages 19–24, 2008.

[51] N. Faci, S. Modgil, N. Oren, F. Meneguzzi, S. Miles, and M. Luck. Towards a Monitoring Framework for Agent-based Contract Systems. In *Proceedings of the 12th international workshop on Cooperative Information Agents XII*, pages 292–305. Springer, 2008.

[52] S. Panagiotidi, J. Vazquez-Salceda, S. Alvarez-Napagao, S. Ortega-Martorell, S. Willmott, R. Confalonieri, and P. Storms. Intelligent Contracting Agents Language. *Behaviour Regulation in MAS, AISB*, pages 49–55, 2008.

[53] WA Woods. Transition Network Grammars for Natural Language Analysis. *Communications of the ACM*, 13(10):591–606, 1970.

[54] RosettaNet. `http://www.rosettanet.org/`, Mar 2002.

[55] C. Molina-Jimenez, S. Shrivastava, and M. Strano. A Model for Checking Contractual Compliance of Business Interactions. Under review for journal publication.

[56] C. Molina-Jimenez, S. Shrivastava, and M. Strano. Exception Handling in Electronic Contracting. In *Proc. 11th IEEE Conf. on Commerce and Enterprise Computing (CEC'09)*, page To appear, Jul 20–23rd, Vienna, Austria, 2009. IEEE Computer Society.

[57] B. J. Biddle. Recent Developments in Role Theory. *Annual Review of Sociology*, 12(1):67–92, 1986.

[58] D. Ferraiolo, J. Cugini, and D.R. Kuhn. Role-based Access Control (RBAC): Features and motivations. In *Proceedings of 11th Annual Computer Security Application Conference*, pages 11–15, 1995.

[59] P. R. Pietzuch, B. Shand, and J. Bacon. Composite Event Detection as a Generic Middleware Extension. *IEEE Network*, 18(1):44–55, 2004.

[60] W3C. Simple Object Access Protocol (SOAP). `http://www.w3.org/TR/soap/`, 2007.

[61] RosettaNet. Implementation Framework, Version V02.00.01 – High Availability Features – Technical Recommendation. `http://tinyurl.com/4tv96t`, 2004.

[62] A.S. Tanenbaum. *Computer networks.* Prentice Hall PTR, 2002.

[63] C Molina-Jimenez and S. Shrivastava. Maintaining Consistency between Loosely Coupled Services in the Presence of Timing Constraints and Validation Errors. In *Proc. of the European Conf. on Web Services (ECOWS2006).* IEEE Computer Society, Washington, DC, USA, 2007.

[64] C Molina-Jimenez, S. Shrivastava, and N. Cook. Implementing Business Conversations with Consistency Guarantees using Message-oriented Middleware. In *Proc. 11th IEEE Intl EDOC Enterprise Computing Conf.(EDOC 2007)*, page 5162. IEEE Computer Society, Washington, DC, USA, 2007.

[65] The Business Rules Group. Defining Business Rules: What Are They Really? `http://www.businessrulesgroup.org/first_paper/br01c0.htm`, 2001.

[66] M. Strano, C. Molina-Jimenez, and S. Shrivastava. A Rule–Based Notation to Specify Executable Electronic Contracts. In *Proc. Int'l Symp. RuleML 2008 (RuleML'08)*, pages 81–88. Springer, LNCS vol. 5321, 2008.

[67] M. Strano, C. Molina-Jimenez, and S. Shrivastava. Implementing a Rule–Based Contract Compliance Checker. In *Proc. 9th IFIP Conference on e-Business, e-Services, and e-Society (I3E'2009)*, page To appear, Nancy, France, 2009. Springer.

[68] Pandora Internet Radio. `http://www.pandora.com`.

[69] Pandora: Frequently Asked Questions. `http://blog.pandora.com/faq/`.

[70] B. H. Liskov and J. M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16:1811–1841, 1994.

[71] MySQL AG. MySQL Database Engine. `http://www.mysql.com`.

[72] Overview of SQL:2003. `http://www.wiscorp.com/SQL2003Features.pdf`.

[73] Drools by JBoss. `http://www.jboss.org/drools/`.

[74] J.C. Giarratano and G. Riley. *Expert systems.* PWS, 1995.

[75] A.J. Gonzalez and D.D. Dankel. *The Engineering of Knowledge-based Systems.* Prentice Hall Englewood Cliffs, NJ, 1993.

[76] Alison Cawsey. Forward Chaining Systems. `http://www.macs.hw.ac.uk/~alison/ai3notes/subsection2\_4\_4\_1.html`, 1994.

[77] 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.

[78] P.J. Landin. Programming without Imperatives: an Example. *UNIVAC SP Research Report (March, 1965)*, 1965.

[79] 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.

[80] The CONTRACT Project. Contracting Language Verification Algorithms (Deliverable). `http://www.ist-contract.org/index.php?option=com_docman&task=doc_details&gid=28&Itemid=44`, Oct 2008.

[81] N. Cook, P. Robinson, and S. Shrivastava. Design and Implementation of Web Services Middleware to Support Fair Non–repudiable Interactions. *International Journal of Cooperative Information Systems*, 15:565–597, 2006.

[82] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

Appendix A

A Grammar for the EROP Language

The grammar and the semantics of the EROP language has been introduced in an intuitive, informal fashion that is useful to grasp the basic principles. However, as a first step for the development of an EROP to Augmented Drools translator, as described in Chapter 7, we present here a grammar for the EROP language, written in the input format for ANTLR [82], a popular parser generator.

```
// Grammar for EROP language

grammar Erop;

// Package specification
//@header { package uk.ac.ncl.erop; }

// Contract definition
contractDocument
: WS? declarationSection WS? ruleSet WS?;

// Structure of the declaration section
declarationSection
: declaration (WS declaration)*;

declaration
: businessOpDeclaration | roleplayerDeclaration | compobligDeclaration;

businessOpDeclaration
: BUSINESSOP upalphanum (COMMA upalphanum)* SEMICOLON;
```

```
roleplayerDeclaration
: ROLEPLAYER alphanum (COMMA upalphanum)* SEMICOLON;

compobligDeclaration
: COMPOBLIG upalphanum BRA upalphanum (COMMA upalphanum)+ KET SEMICOLON;

// Rule set structure
ruleSet : singlerule (WS singlerule)*;

// Rule structure
singlerule
// : 'rule' WS rulename WS lhs WS rhs WS 'end';
: RULE WS rulename WS WHEN WS lhs WS THEN WS rhs WS END;

rulename
: '\"' upalphanum '\"';

// Left hand side structure
lhs : eventmatch (WS constraint)*;

eventmatch
: alphanum WS MATCHES WS upalphanum;

constraint
: attributeConstraint | historicalQuery | ropConstraint;

attributeConstraint
: roleplayerConstraint | outcomeConstraint | timeConstraint;

roleplayerConstraint
: alphanum DOT (ORIGINATOR|RESPONDER) WS? (EQUALS|NOTEQ) WS? alphanum;

outcomeConstraint
: alphanum DOT OUTCOME WS? (EQUALS|NOTEQ) WS? outcome;
```

```
timeConstraint
: timeDirectComparison | timePartialComparison;

timeDirectComparison
: alphanum DOT TIMESTAMP WS? (EQUALS|NOTEQ|BEFORE|AFTER) WS? absoluteTime;

timePartialComparison
: alphanum DOT DAY WS? (EQUALS|NOTEQ) WS? WEEKDAY
|alphanum DOT DAY WS? (IN|NOTIN) WS?
SQUAREBRA WEEKDAY DOT DOT WEEKDAY SQUAREKET
|alphanum DOT DATE WS? (EQUALS|NOTEQ|BEFORE|AFTER) WS? DIGIT DIGIT
|alphanum DOT DATE WS? (IN|NOTIN) WS?
SQUAREBRA DIGIT DIGIT DOT DOT DIGIT DIGIT SQUAREKET
|alphanum DOT MONTH WS? (EQUALS|NOTEQ|BEFORE|AFTER) WS? MONTHID
|alphanum DOT MONTH WS? (IN|NOTIN) WS?
SQUAREBRA MONTHID DOT DOT MONTHID SQUAREKET
|alphanum DOT YEAR WS? (EQUALS|NOTEQ|BEFORE|AFTER)
WS? DIGIT DIGIT DIGIT DIGIT
|alphanum DOT MONTH WS? (IN|NOTIN) WS?
SQUAREBRA DIGIT DIGIT DIGIT DIGIT
DOT DOT DIGIT DIGIT DIGIT DIGIT SQUAREKET
;

historicalQuery
: (HAPPENED | CTHAPPENED) WS? BRA upalphanum COMMA WS? alphanum
COMMA WS? alphanum COMMA WS? genericString COMMA WS? outcome KET;

ropConstraint
: upalphanum IN|NOTIN alphanum DOT ropset;

// Right hand side structure
rhs : rhsaction (WS? rhsaction)*;

rhsaction
// : addaction|remaction|termaction|passaction SEMICOLON;
: (ifstatement|termaction|passaction|addaction|remaction) WS? SEMICOLON;
```

```
// Support for if-then-else-endif statement
ifstatement
: IF WS condition WS THEN WS rhs WS (ELSE WS rhs WS)? ENDIF;

condition
: BRA WS? NOT? constraint WS? ((AND|OR) condition ) WS? KET;

termaction
: TERMINATE WS? BRA outcome KET;

passaction
: PASS;

addaction
: alphanum DOT ropset WS? ADDROP WS? upalphanum BRA timeSpec? KET;

remaction
: alphanum DOT ropset WS? REMROP WS? upalphanum BRA timeSpec? KET;

// Rules for both lhs and rhs
outcome
: SUCCESS | TECFAIL | INITFAIL | BIZFAIL;

ropset
: RIGHTS|OBLIGS|PROHIBS;

timeSpec
: absoluteTime | relativeTime;

absoluteTime
: DQUOTE DIGIT DIGIT DASH DIGIT DIGIT DASH DIGIT DIGIT DIGIT DIGIT
WS DIGIT DIGIT COLON DIGIT DIGIT COLON DIGIT DIGIT DQUOTE;

relativeTime
: relTimeElement+;
```

```
relTimeElement
: DIGIT+ ('s'|'m'|'h'|'d'|'M'|'Y');

// Token for declaration section
ROLEPLAYER
: 'roleplayer';
BUSINESSOP
: 'businessoperation';
COMPOBLIG
: 'compoblig';

// Tokens for Basic rule structure
RULE: 'rule';
END: 'end';
WHEN: 'when';
THEN: 'then';

// Tokens for left hand side
MATCHES
: 'matches';
HAPPENED
: 'happened';
CTHAPPENED
: 'counthappened';
BEFORE: 'before';
AFTER: 'after';
BOTYPE: 'botype';
ORIGINATOR: 'originator';
RESPONDER
:  'responder';
OUTCOME: 'outcome';
TIMESTAMP: 'timestamp';
DAY : 'day';
DATE: 'date';
SECOND
```

```
: 'second';
MINUTE
: 'minute';
HOUR: 'hour';
MONTH
: 'month';
YEAR: 'year';
IN: 'in';
NOTIN: '!in';
EQUALS: '==';
NOTEQ
: '!=';
AND: '&&';
OR: '||';
NOT: '!';
WEEKDAY
: 'Mon' | 'Tue' | 'Wed' | 'Thu' | 'Fri' | 'Sat' | 'Sun';
MONTHID
: 'Jan' | 'Feb' | 'Mar' | 'Apr' | 'May' | 'Jun'
|'Jul' | 'Aug' | 'Sep' | 'Oct' | 'Nov' | 'Dec';

// Tokens occurring in both lhs and rhs
SUCCESS: 'Success';
TECFAIL: 'TecFail';
BIZFAIL: 'BizFail';
INITFAIL: 'InitFail';

// Right hand side tokens

ADDROP: '+=';
REMROP: '-=';
TERMINATE: 'terminate';
PASS: 'pass';
OBLIGS: 'obligs';
RIGHTS: 'rights';
PROHIBS: 'prohibs';
```

```
// Tokens for Right hand side: structured statements
IF: 'if';
//THEN: 'then';
ELSE: 'else';
ENDIF: 'endif';

// Tokens for Right hand side: status guards
OTHERWISE: 'Otherwise';

// Identifiers, with uppercase and lowercase initials
upalphanum
: UPPER (LOWER | UPPER | DIGIT)*;

alphanum
: LOWER (LOWER | UPPER | DIGIT)*;

genericString
: DQUOTE (LOWER | UPPER | DIGIT | WS | SEMICOLON
| COLON | COMMA | QUOTE | DOT | DASH | BACKSLASH)* DQUOTE;

// Alphabet, numbers
LOWER: 'a'..'z';
UPPER: 'A'..'Z';
DIGIT: '0'..'9';

// Various characters
SEMICOLON
: ';';
COLON
: ':';
HASH: '#';
BRA: '\(';
KET: '\)';
COMMA: ',';
QUOTE: '\'';
```

```
DQUOTE: '\"'; // "
SQUAREBRA: '\[';
SQUAREKET: '\]';
DOT : '\.';
DASH: '-';
BACKSLASH
: '\\';

// Whitespace includes spaces, newlines and tabs
WS
: (' ' | '\t' | '\r' | '\n' )+;
```