A Model for Checking Contractual Compliance of Business Interactions

Carlos Molina–Jimenez, Santosh Shrivastava, and Massimo Strano

Abstract—The electronic representation of a contract for a business-to-business (B2B) partnership should be such that it can be used by a monitoring service for compliance checking of B2B interactions at run time, ensuring that the interactions match the rights and obligations that each partner has promised to honour. With this view in mind, the paper develops a model for checking contractual compliance of business interactions. Specifically, the paper develops a novel way of representing contract clauses using business rules, that is specially suited to compliance checking and describes what events need to be captured from the underlying messaging middleware and how they can be processed in a careful manner to evaluate contractual compliance.

Index Terms—Electronic contracts, contract monitoring, compliance, B2B messaging, fault tolerance, distributed systems.

1 INTRODUCTION

In the business world, legal contracts form the basis to regulate the interaction between two or more trading parties. When conducting business electronically, electronic contracts would be needed to regulate business interactions. By regulation we mean monitoring and/or enforcement of the business-to-business (B2B) interactions to ensure that they comply with the rights, obligations and prohibitions stipulated in the contract. Appropriately specified electronic contracts can play a central role in compliance checking and enforcement.

We primarily focus on the terms and conditions of B2B legal contracts concerned with purchase order fulfilment, supply chain management etc., rather than service level agreements (SLAs) that specify quality of service, such as bandwidth and response time (although the ideas of this paper can be extended to cover SLAs as well).

Business interactions stipulated in contractual clauses can be arbitrarily complex; we assume that they can be decomposed into primitive business operations defined by some standard specific to the business domain such as RosettaNet Partner Interface Processes (PIPs) [1] and ebXML transactions [2]. Examples of such primitive business operations are PO submission, invoice notification, payment submission, PO cancellation, etc., where PO stands for Purchase Order. At the communication level, the execution of each business operation results in the execution of a well defined protocol between the two parties for sending or requesting one or more business documents. We refer to this B2B message based interaction as a business conversation, (conversation, for short).

Consider now that as a part of ongoing interaction between a buyer and a seller, the buyer makes a payment: has the buyer met his obligation? Could it be that the payment event, as observed by the seller, occurred after the deadline stipulated in the contract, thus from his view the buyer has not met the obligation? What if the buyer makes a payment well within the deadline but, unknown to him, the attempt does not succeed due to a technical problem at the seller’s end? To aid the resolution of such issues, we introduce an independent, third party monitoring service called Contract Compliance Checker (CCC) that is provided with the specification of the contract in force and is capable of observing significant events related to partner interactions. The function of the CCC is to maintain a record of the observed events and act as an arbiter to help provide answers to the kinds of questions raised earlier. We chose the third party approach on practical grounds, noting that e-commerce applications routinely make use of third party services (examples include payment gateways, certification authorities, timestamping services, various brokering services), so a CCC service would be a natural fit. The task facing the CCC is made harder due to the nature of the underlying distributed execution environment for B2B interactions that is characterized by three factors discussed below:

- Loose coupling: B2B interactions take place in a loosely coupled manner, typically using message oriented middleware (MoM) where business partners are not required to be on-line at the same time. As a result, interacting partners rarely have an up-to-date information on the state of other partners, so there is a danger of partners getting out of synchrony with each other (state misalignment).
- Timing and validity constraints: Messages in a given conversation (for example, as stipulated in RosettaNet PIPs [1], [3] and ebXML [2] transactions) are subjected to various timing (deadlines) as well as validity constraints. A business message is accepted for processing only if it is timely and satisfies specific syntactic and semantic validity constraints.
Such constraints can be yet another cause of state misalignment between the partners. For example, imagine that a business partner sends a message as part of a conversation that represents the execution of a business operation. It is possible that the message is delivered but not taken up for processing at the receiver’s end after failing to satisfy a validity test; if this happens, the sender’s and the receiver’s views over the outcome of the operation will diverge: the sender will assume that the operation succeeded, whereas the receiver will assume that the operation failed.

- Faulty environment: Business interactions encounter software, hardware and network related problems (e.g., clock skews, unpredictable transmission delays, lost and incorrect messages, node crashes etc.).

In order to establish the validity of the actions of trading partners with respect to the contract in force, compliance checking must take into account the impact of the above factors. Further, the CCC should be able to work hand in hand with well-known industry standards on B2B messaging and provide easy to use notations and techniques for representing contract clauses suited to compliance checking. Existing work on contract specification and monitoring has paid attention to some but not all of the above requirements simultaneously. We remedied the situation by presenting a technique for representing contractual rights, obligations and prohibitions of trading partners and discuss what events need to be captured from the underlying messaging middleware and how they can be processed in a careful manner by the CCC for evaluating contractual compliance. The paper essentially develops an event based model of a CCC. The main contributions of the paper are enumerated below.

- A simple execution model of business operations representing business conversations that takes into account the impact of the above three factors. In particular, the model describes what synchronization mechanisms are necessary to prevent state misalignment. Despite the simplicity, the model is practical enough to represent well-known B2B messaging standards, yet it abstracts away complex implementation specific protocol details.
- A novel way of representing contract clauses using business rules that is specially suited to compliance checking.
- A rigorous description of the CCC architecture in terms of the components for business rule specification, event management and logging; this serves as the basis for developing practical contract representation languages and third party services for monitoring and enforcing contract compliance.

This paper is organised as follows. In Section 2 we present the relevant concepts that underpin the CCC model: contracts, compliance checking, business conversations and the execution model of business operations. In Section 3 we discuss the details of the CCC and present its uses in Section 4. Brief implementation details are given in Section 5. Related work is discussed in Section 6. Finally, in Section 7 we summarise the work and suggest future research directions.

2 OVERALL VIEW

2.1 Overall Architecture

The CCC is a neutral entity (conceptually located between the interacting parties, see Fig. 1), that is provided with an executable specification of the contract in force; it is able to observe and log the relevant B2B interaction events which it processes to determine whether the actions of the business partners are consistent with respect to the contract. Business partners (buyer and seller in the figure) interact by executing business operations, which implies execution of corresponding conversations. This activity can be viewed as the business partners taking part in the execution of a shared or public business process, where each partner is responsible for performing their part in the process. In the figure, this is represented by the two public business processes Public business process$_B$ and Public business process$_S$, belonging to the buyer and seller, respectively. Private business process$_B$ and Private business process$_S$ represent, respectively, the buyer’s and seller’s private business processes, that is, processes that are internal to their businesses. We do not elaborate on the overall structure of public and private business processes, as it is not directly relevant within the context of this paper.

![Fig. 1. Abstract view of the architecture.](image)

The figure depicts two logical communication channels: one is for facilitating business conversations and the other is a monitoring channel to deliver business events (bevents) to the CCC. Precise details of interaction between the monitoring channel and the CCC (the components inside the dotted box) are discussed subsequently. We do not elaborate on the actual implementation of the two channels, other than to note that their realisation is usually based on some form of MoM.

2.2 Rights, Obligations and Prohibitions

The clauses of a given contract can be abstracted as six sets (three for each partner) that contain the list
of Rights (R), Obligations (O), Prohibitions (P) that the two business partners are expected to honour under the observance of their associated constraints.

Informally, a right is something that a business partner is allowed to do; an obligation is something that a business partner is expected to do unless they wish to take the risk of being penalized; finally, a prohibition is something that a business partner is not expected to do unless they are prepared to be penalized. Note that some authors point out the fine distinction between the terms permissions and rights; in our model we have taken the rather pragmatic approach of treating the two terms as equivalent. In the same order, it can be argued that prohibitions are not strictly necessary since they could be modelled as complements of rights. We include them in our model to differentiate between invalid operations (those that are not processed at all) and prohibited operations (those whose execution results in sanctions).

Business conversations are initiated and responded by role players that operate on behalf of their enterprises. As explained in Section 3.2, for simplicity, our contract examples consider only two role players: buyer and seller.

As an example, we show below seven clauses from a hypothetical contract where we explicitly identify a few rights, obligations, prohibitions and constraints. Note that these clauses are not meant to form a complete legal contract as they have been simplified and adjusted to illustrate research issues; a practical contract would have additional clauses detailing other aspects of the interaction.

These seven clauses compose what we call a contractual transaction which is initiated by the buyer with the intention of placing a single purchase order with the seller to buy some goods. To keep our model simple, we assume that contractual transactions execute independently from each other; that is, to place another purchase order, the buyer would need to start a second contractual transaction whose execution might possibly overlap, but will not interfere, with the first.

- C1: The buyer has the right to submit a Purchase Order (RIGHT), between 9 am and 5 pm, Mon to Fri. (CONSTRAINT).
- C2: The seller has the obligation to respond to the Purchase Order with acceptance or rejection within 24 hours (OBLIGATION).
- C3: The seller’s failure to respond to the Purchase Order within the 24 hours deadline will result in an abnormal contract termination with off-line settlement.
- C4: If the Purchase Order is accepted, the seller is obliged to submit an invoice within 24 hours (OBLIGATION).
- C5: The buyer has the obligation to pay the due amount within seven days of receiving the invoice (OBLIGATION).
- C6: The seller is obliged to deliver the goods within seven days of receiving payment (OBLIGATION).
- C7: This contractual transaction terminates when either

2.3 Contract–Compliant Operations

Informally, we say that a given business operation (bo) executed by a role player is contract–compliant if the execution took place in accordance with the rights, obligations and prohibitions and their associated constraints stipulated in the contractual clauses. Several constraints can be associated to a given operation; usually, they can be grouped into three categories:

- Initiator’s and responder’s identities An operation is contract–compliant only when it is initiated and responded by the role players stipulated in the contract. For instance, C1 of the contract example stipulates that a PO operation has to be initiated by the buyer, to be counted as contract–compliant. It is worth mentioning that in practice the role players involved in the execution of an operation might be constrained by more general and fairly complex role patterns that might take into account sequence and number of operations where the role player has been involved [4].
- Date and time of occurrence of operation To be contract–compliant, an operation should be initiated at and/or terminated by the stipulated date and time. For example, C1 of our contract example implies that PO operations initiated on Saturdays and Sundays are non-contract-compliant.
- History of previous operations These constraints capture causality between operations and other events (eg. timeouts). For example, a contract might stipulate that a given boj is contract–compliant only if it is (or is not) preceded by boi. In our contract example, clause (C4) implies that an invoice operation not preceded by a PO is non–contract–compliant.

2.4 Business Conversations

We describe timing and message validity aspects of conversations using RosettaNet specification as an example [1], [3]. RosettaNet specifies a number of Partner Interface Processes (PIPs), that define basic business operations such as Request Purchase Order, Notification of Invoice etc. Each PIP specifies the choreography of the message dialogue for a conversation, and includes business action and business signal messages; the latter consist of Acks and Nacks (positive and negative, acknowledgements, respectively). There are two kinds of PIPs: single–action and two–action. In a single–action interaction only a single electronic document is exchanged, whereas a two–action interaction involves exchange of two documents: a request and its response.

A received document is accepted for processing by the 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 standard (this validation can be performed within the public process); and in addition,

2) **Content-validation**: a base–validated document must also be semantically valid: document contents should satisfy some arbitrary, application specific correctness criteria. This validation is trading partner specific and normally performed within the receiver’s private business process.

Fig. 2 shows examples of single–action (Notification of Invoice PIP 3C3) and two–action (Request Purchase Order PIP 3A4) PIPs.

![Image](image-url)

**Legend:** action message signal message base-val base validation

Fig. 2. Examples of RosettaNet PIPs.

![Image](image-url)

**Legend:** action message signal message base-val base validation

Fig. 3. Use of notification of failure.

A shown in the figure, nModel the receiver of an action message is obliged to acknowledge it by sending a signal message back within two hours. Since the content-validation is normally performed at the application level (within the private process of the receiver) and could take arbitrary amount of time, the RosettaNet standard specifies that the acknowledgement is sent after base-validation only. 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., seller, for PIP 3C3) and its responder (buyer, for PIP 3C3) could end up with contradictory views of a PIP execution. For example, in PIP 3C3, if the ReceiptAck signal message is lost or arrives after the two hour limit, the buyer’s and seller’s views could respectively be successful and failed termination; subsequent executions of public business processes at each end could diverge, causing business level errors. RosettaNet relies on Nacks to synchronize partners at PIP level and minimize the errors propagated to the business application [3]. An Ack is sent to indicate that an action message has been received and successfully base–validated. A Nack is sent to indicate that an action message has been received but failed its base–validation. However, as discussed below, some errors inevitably could propagate to the business level.

The impact of base and content validation (base-val and content-val, respectively) is illustrated in Fig. 3, where for simplicity we use a single action PIP (PIP 3C3). The figure illustrates how the states of the buyer and 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 that state misalignment has occurred. For this, RosettaNet provides a special Notification of Failure PIP 0A1. The intention is to enable the receiver (the seller in this example) to take application specific actions to re–synchronize. Alternatively, mutual inconsistency can be prevented from occurring in the first place by wrapping business conversations with explicit outcome synchronization as discussed next.

### 2.5 Execution Model of Business Operations

We develop a model of business operations between loosely coupled processes with outcome synchronization that prevents state misalignment. We assume that partners implement business operations according to this model. Also, given the wide variety of primitive events that can be generated at both sides of a conversation (send, receive, timeout, retry,...), it is worthwhile to examine if any aggregation of these primitive events can be performed to produce only a few significant events containing enough information for a third party to judge the development of the business interaction; we call them **business events** (**bevents** for short). The idea of primitive event aggregation is captured by our **execution model of a business operation** (**bexec**) shown in Fig. 4.

B2B messaging is typically implemented using MoM that permits loose coupling between partners (e.g., the partners need not be online at the same time). To guarantee that a business partner starts sending conversation messages only when its counterpart is ready to converse, we need to wrap the conversation with connection management protocols for starting and terminating the conversation. This is of course a standard practice in protocol engineering (for example, connection management for TCP messages in the Internet [5]). We thus split the execution of a business operation into two steps,
Business operation initiation: We assume that for each operation there is an initiator (the party that expresses interest in the execution) and a responder (the party invited to take part in the execution). The initiation (Fig. 4-a) involves the execution of an initiation protocol (init protocol) started by the initiator. Once this protocol is started, the initiator eventually produces either InitS\_B or InitF\_B to declare that locally the initiation was successful or failed, respectively. Similarly, on the other side, the responder produces InitS\_S or InitF\_S to declare that locally, the initiation was successful or failed. The init protocol counts for situations where no outcome is produced by the responder, say because it did not receive the invitation message or was unable or unwilling to respond.

Synchronization of initiation outcomes: In this stage (Fig. 4-b) the initiation syn protocol is executed to produce an Init composite outcome. The protocol declares InitS only when both partners locally declare success and InitF when at least one of the parties declares InitF or fails to provide its local outcome after the expiry of a time out.

Conversation execution: This stage (Fig. 4-c) takes place only when the composite outcome from initiation syn is InitS. Following ebXML specification [2], we assume that once a conversation is started, it always completes to produce at each side one of three possible local outcome events: Success, BizFail or TecFail, repre-

senting, success, business failure and technical failure, respectively. When a party considers that the conversation completed successfully, it generates Success. BizFail and TecFail model 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 errors detected at the middleware level, such as missing or syntactically incorrect message (message that fails base-valuation). BizFail models semantic errors (content-valuation problems) in a message, detected at the business level, e.g., the goods-delivery address extracted from the business document is invalid.

Synchronization of conversation results: The synchronization protocol (execution syn) executed at this stage (Fig. 4-d) guarantees mutually agreed conversation outcomes. This synchronization is intended to prevent the kind of situation depicted in Fig. 3. execution syn can be regarded as an event composer programmed to execute a given composition logic. Our execution syn synchronizer operates under the following logic: (a) identical local outcome events are composed into a composite outcome event of the same type; (b) if one of the local outcome events is TecFail then the composite outcome event is of type TecFail, irrespective of the type of the other event; (c) if one of the local outcome events is BizFail and the other is not TecFail, then the composite outcome event is of type BizFail.

In summary, the execution of a business operation generates InitF if the initiation attempt fails, otherwise an outcome from the set \{Success, BizFail, TecFail\} is generated. We assume that synchronizers are able to perform outcome synchronization in the presence of failures (lost, delayed messages, node crashes etc.); it is the responsibility of the business partners to deploy them, perhaps as suggested in [6] and [7].

3 CONTRACT-COMPLIANCE CHECKER

3.1 Assumptions

At the heart of our architecture is the Contract Compliance Checker (CCC) shown in Fig. 1. The CCC must observe business interactions accurately. For example, a correctly-functioning CCC should not flag a given operation as missing or late when that operation successfully completed on time. Potential threats that can affect compliance checking by the CCC are misbehaviour on part of the business partners and/or the failure of any of the components within the dashed box shown in Fig. 1; for instance a failure in the monitoring channel can prevent timely bevents from reaching the contract compliance checker on time (and therefore appear as untimely). Our model works under the following assumptions:

- Business partners correctly implement the execution model of business operations. In other words, bevents are correctly generated by the partners.
- The components inside the dashed box of Fig. 1 function correctly.
• The clocks of all the parties are synchronized to a master global clock with a known accuracy, \( \epsilon \). Thus the difference between the reading of any two clocks is never larger than \( 2\epsilon \).

• Bevents are time stamped at the source.

• Bevents are delivered exactly once to the CCC in temporal order.

• The transmission and processing delays (TPD) of bevents from the source (initiation or execution synchronizer) to the bevent queue of the CCC are bounded and known (as discussed subsequently, the CCC is provided with a queue for events that need processing).

• The CCC has a timer process for generating timeout events as per the contract. To guarantee that this timer does not erroneously generate a timeout event about the absence of an operation when the business event about the execution of the operation is on its way to the event queue, all timeout events are delayed by the quantity \( TPD + 2\epsilon \). This quantity compensates for transmission and processing delays and any error in clock synchronization.

• The buyer’s and seller’s infrastructure components can fail by crashing and eventually recovering, however, all bevents that are generated are supplied to the monitoring channel as per TPD.

Any practical system design will have to make engineering decisions on realistic ways of meeting these assumptions. In particular, in B2B settings where it is appropriate to assume that business partners operate in good faith (they do not generate malicious events, say with incorrect time stamps or send malicious messages to each other), the implementation of the execution model of business operation need not contain any security mechanisms. Otherwise, it will be necessary to introduce cryptographically secure authentication and non-repudiation mechanisms in the implementation of the execution model of business operation (for example, as discussed in [8]) to prevent malicious business partners from abusing the service.

### 3.2 Role Players

A role player is an entity (not necessarily human) authorized to perform a function within the contractual interaction. Normally, each party is represented by several role players. For simplicity, in our scenario we assume that each party is represented by a single role player, namely, a buyer and a seller. We use \( RP = \{rp_1, \ldots, rp_n\} \) to represent the set of all the role players that participate in a contractual interaction.

### 3.3 Business Operations

A business operation is a business activity whose execution involves the two contracting parties. The set of valid business operations \( B = \{bo_1, \ldots, bo_n\} \) contains all the elementary business operations stipulated in the contract. We say that a given \( bo_i \) is a valid business operation only if \( bo_i \in B \). In other words, \( B \) represents the vocabulary of the shared business process. Business operations accidentally or intentionally left unspecified by contract designer are called unknown business operations and represented by set the \( B = \{bo_i | bo_i \notin B\} \).

In our buyer–seller example, the set \( B \) would contain \( B = \{SubPO, AcceptPO, RejectPO, ...\} \); where \( Sub, Accept, Reject \) and PO stand for submission, acceptance, rejection and purchase order, respectively; CancelPayment would be an unknown business operation, as it is not stipulated in the contract.

### 3.4 Rights, Obligations and Prohibitions

Contractual clauses stipulate role players’ rights, obligations and prohibitions to execute operations from the set \( B \), frequently under the constrain of deadlines.

A deadline is a time constraint to execute an operation. We use the time expression \( t \) to represent deadlines; the absence of \( t \) is taken as no deadline constraint.

**Rights:** A right is a role player’s privilege to execute an operation against its counterpart, under the constraint of an optional deadline. We define an individual right \( r_i \) as a set of operations, \( r_i \subseteq B \). To exercise \( r_i \) the role player can execute a single operation from the set, under the constraint of an optional deadline. We use the following naming convention for a right. When \( |r_i| = 1 \), that is, the right is mapped onto a single operation, we use the same name to refer to both the right and the operation. However, when \( |r_i| \geq 2 \), the name of the right is chosen different from those of the operations. An example of a right is \( SubPO \), stipulated in clause C1 of our example, without a deadline. Such a right is mapped onto a single operation of the same name: \( SubPO = \{SubPO\} \).

**Obligations:** An obligation is a role player’s commitment to execute an operation for the benefit of its counterpart, under the constraint of a mandatory deadline. We define an individual obligation \( o_i \) as a set of operations, \( o_i \subseteq B \). To fulfil \( o_i \), the role player can execute a single operation from the set, under the constraint of a mandatory deadline \( t \). Each individual obligation can be regarded as a tuple \((o_i, t)\). To refer to obligations, we use the same naming convention discussed above in the definition of rights. As an example, take the seller’s obligation \( RespondPO \) stipulated in clause C2. The obligation \( RespondPO \) is mapped onto two operations, namely, \( AcceptPO \) and \( RejectPO \); thus \( RespondPO = \{AcceptPO, RejectPO\} \) with \( t = 24hrs \) deadline. To fulfil obligation \( RespondPO \), the seller needs to execute either operation \( AcceptPO \) or \( RejectPO \) within 24 hours.

**Prohibitions:** A prohibition is a role player’s commitment not to execute an operation against its counterpart within the constraint of an optional deadline \( t \). A failure to observe the commitment normally results in sanctions. For the sake of uniformity with the definitions of rights and obligations, we define an individual prohibition \( p_i \).
as the set \( p_i \subseteq B \), where \( |p_i| = 1 \). Note that a prohibition is mapped onto a single operation only since there is no practical value in mapping a prohibition onto a set with multiple operations. To refer to prohibitions, we use the same naming convention discussed above.

### 3.5 Business Events

A **business event** is an object (data structure) with several attributes that contain data about the execution of a business operation. The business events we use in the model have four attributes:

\[
\text{bevent ::= name, exoutcome, initiator, ts}
\]

- **name**: Either the name of an operation or the name of a right, obligation or prohibition whose deadline has overrun. **exoutcome** is the composite outcome from the execution of the operation or a timeout notification (when the deadline is overrun). **initiator** is the name of the role player that happens to be the initiator of the operation. Finally, **ts** is the time stamp of the execution.

As stated in Section 2.5, when the execution of an operation \( bo_i \) takes place, it will always produce one out of four possible composite outcomes as shown in Fig. 5.

![Fig. 5. Composite execution outcomes of an operation.](image)

Consequently, the value of **exoutcome** will be **InitF**, **Success**, **BizFail** or **TecFail**. For example, the execution of the operation **SubPay** stipulated in clause C5 of our running example, will produce the bevent \( \text{(SubPay,InitF,buyer,ts)} \) or \( \text{(SubPay,Success,buyer,ts)} \) or \( \text{(SubPay,BizFail,buyer,ts)} \) or \( \text{(SubPay,TecFail,buyer,ts)}. \)

We also use bevents to notify the expiry of deadlines that constrain rights, obligations and prohibitions. These business events are generated internally by the timer of the CCC (see Fig. 7). In these situations the value of the attribute **name** is the name of the right, obligation, or prohibition that timed out, whereas the value of the attribute **exoutcome** is **TO** and indicates a time out. Take clause C5 as an example; a failure to honour the obligation **SubPay** by executing the operation **SubPay** within the seven day deadline will produce the bevent \( \text{(SubPay,TO,buyer,–)}. \)

Take clause C2 as a second example; a failure to honour the obligation **RespondPO** within the 24 hours deadline by executing either the operations **AcceptPO** or **RejectPO** within this deadline, will produce the bevent \( \text{(RespondPO,TO,seller,–)}. \)

### 3.6 Current ROP sets

As a contractual interaction progresses, the rights, obligations and prohibitions associated with role players typically change, with some new ones coming in scope and some old ones going out of scope. We use \( R_{rp}, O_{rp} \) and \( P_{rp} \) to represent, respectively, the sets of rights, obligations and prohibitions currently assigned to a role player \( rp \). In our example, \( R_B, O_B \) and \( P_B \) represent, respectively, the buyer’s set of rights, obligations and prohibitions currently in force. Similarly, \( R_S, O_S \) and \( P_S \) represent, respectively, the seller’s set of rights, obligations and prohibitions currently in force. The three buyer’s and seller’s sets are represented, respectively, by \( ROP_B \) and \( ROP_S \). We refer to the six sets collectively by ROP sets and call them the **current set of rights, obligations and prohibitions**. We use the membership operations \( r_i \in ROPsets \), \( o_i \in ROPsets \) and \( p_i \in ROPsets \) to determine, respectively, if right \( r_i \), obligation \( o_i \) or prohibition \( p_i \) are in the ROP sets.

### 3.7 Matching of rights, obligations and prohibitions

We use the operator \( \vdash \) to specify a **match operation** between a given \( bo_i \) and the ROP sets, which evaluates to true (T) or false (F). Intuitively speaking, a match operation determines if the execution of a given operation is currently expected within the ROP sets and likely to be declared contract–compliant after evaluating additional conditions.

Given a valid business operation \( bo_i \in B \) and \( R_{rp} = \{r_1, \ldots, r_m\}, m \geq 1 \), \( bo_i \vdash R_{rp} = T \) only if \( bo_i \in r_j, 1 \leq j \leq m \). Match operations \( bo_i \vdash O_{rp} \) and \( bo_i \vdash P_{rp} \) are defined in a similar manner. Collectively, we say that a \( bo_i \vdash ROP_{rp} = T \) if \( bo_i \) matches one of the ROP sets of the role player. It should be understood that a match operation against an empty set always evaluates to false.

### 3.8 Contract violation

A terminated contractual interaction is classified as **normally terminated** if there are no pending obligations (all the obligations have been fulfilled). In the buyer, seller scenario this would mean that \( O_B = \emptyset \) and \( O_S = \emptyset \). On the other hand, a **contract violation** occurs if the termination leaves one or more unfulfilled obligations. Note that contract violation is defined based on the final, terminated state of the contractual interaction and is distinct from any violation of an obligation that could occur during an interaction; such a violation normally leads to sanctions coming in force and if these are honoured, then the contractual interaction could still end normally (this is discussed further in section 4.2).

### 3.9 Contract–compliant operations

As we discuss later, the CCC maintains some essential state information about the ongoing interaction in the ROP sets and updates that information only when contract–compliant operations are executed. This is the basis by which the CCC monitors an interaction and determines whether the final state is normal or abnormal (contract–violation has occurred). To determine if the
execution of a given $b_{o_1}$ is contract–compliant or not, the CCC needs to analyse three requirements.

1) $b_{o_1} \in B$.
2) $b_{o_1} \vdash \text{ROP}_{r_{op}}$.
3) Observance of the three constraints discussed in Section 2.3.

Four possible alternatives can emerge from the analysis of the above requirements.

- A business operation that fails to satisfy the first requirement is called an unknown business operation (as defined in Section 3.3) and treated as non-contract–compliant without further analysis.
- A business operation that satisfies the first requirement but fails to satisfy the second is called a mismatched business operation and treated as non-contract–compliant without further analysis.
- A business operation that satisfies the first and second requirements but fails to satisfy the third is called an out of context business operation and treated as non-contract–compliant.
- A business operation that satisfies all the three requirements is called a contract-compliant business operation.

For example, in clause C1 of our running example, the buyer would generate an out of context execution if he starts the contractual interaction by executing a SubPO on a weekend; this execution would satisfy the first and second requirements but not the third. Note that the execution of a non-contract–compliant operation does not necessarily lead to a contract violation.

### 3.10 CCC as a reactive system

The significance of the ROP sets in our model is that they allow us to abstract the behaviour of the CCC as that of a conventional reactive system [9]. As a reactive system, the CCC remains in a given state $s_i$ waiting for the arrival of events. When a bevent $e_j$ arrives and determined to represent a contract–compliant operation, the system enters state $s_j$. The state space is determined by the set of valid operations $B = \{b_{o_1}, \ldots, b_{o_n}\}$. For each partner, a given $b_{o_1}$ can belong to at most one of its three ROP sets at any time, so, in a two party interaction, the state space of the reactive system will be bounded by $2^{6n}$. However, because of the constraints imposed on the executions of business operations by a given contract, the state space is expected to be much less than $2^{6n}$.

We will use the clauses of our running example to illustrate this idea. In the figure, $\emptyset$ represents the empty set; $e$, $c$, $T$, $Inv$, $TO$ and $Sub$ stand for bevent, condition, true, invoice, timeout, and submit, respectively. Notice that for simplicity, we show only the name and exoutcome attributes of the business events. Likewise, we show only bevents with Success and TO execution outcomes of contract compliant operations; that is, we do not show bevents with InitF, TecFail or BizFail execution outcomes; neither we show bevents related to unknown and out of context executions. In the discussion, C1, C2, C3, etc., refer to the clauses of our running example.

![Fig. 6. The CCC as a reactive system.](image)
$O_S$, thus, it appears as pending obligation in $s_4$.

State $s_2$ corresponds to C2 and C7.1: From $s_1$ the interaction progresses to the acceptable final state $s_2$ when the seller successfully executes the $\text{RejectPO}$ operation within the 24 hours deadline constraint. The obligation $\text{RespondPO}$ has been removed from $O_S$ as it has been fulfilled by the execution of the operation $\text{RejectPO}$.

State $s_3$ corresponds to C3: From $s_1$ the interaction progresses to the abnormal final state $s_3$ if the seller fails (for example, due to $\text{BizFail}$ or $\text{TecFail}$ reasons) to honour his $\text{RespondPO}$ obligation before the expiry of the 24 hours deadline. $\text{RespondPO}$ is left in $O_S$ as pending obligation.

State $s_6$ corresponds to C5: From $s_4$ the interaction progresses to $s_6$ when the seller successfully executes $\text{SubInv}$ operation within the 24 hours deadline constraint. The obligation $\text{SubInv}$ has been removed from $O_S$ as it has been fulfilled by the execution of the $\text{SubInv}$ operation and the obligation $\text{SubPay}$ has been added to $O_B$, consequently, it appears as pending obligation in this state.

State $s_5$ corresponds to C4: From $s_4$ the interaction progresses to $s_5$ when the seller fails to successfully execute the operation $\text{SubInv}$ within the 24 hours deadline to fulfil his $\text{SubInv}$ obligation. The obligation $\text{SubInv}$ is left pending in $O_S$.

State $s_8$ corresponds to C6: From $s_6$ the interaction progresses to $s_8$ when the buyer successfully executes the $\text{SubPay}$ operation within the seven day deadline. The obligation $\text{SubPay}$ has been removed from $O_B$ as it has been fulfilled by the execution of the operation $\text{SubPay}$ and the obligation $\text{Delivery}$ has been added to $O_S$, thus it appears as pending obligation in $s_8$.

State $s_7$ corresponds to C5: From $s_6$ the interaction progresses to the abnormal final state $s_7$ when the seven day timeout to execute the operation $\text{SubPay}$ expires leaving the obligation $\text{SubPay}$ pending in $O_B$.

State $s_{10}$ corresponds to C7.2: From $s_8$ the interaction progresses to acceptable final state $s_{10}$ when the seller successfully executes the $\text{Delivery}$ operation within the 7 day deadline. Notice that in $s_{10}$ the $\text{ROP}$ sets are left empty (no pending rights, obligations or prohibitions).

State $s_9$ corresponds to C6: From $s_8$ the interaction progresses to the abnormal final state $s_9$ when the seven day timeout to successfully execute the operation $\text{Delivery}$ expires. The $\text{Delivery}$ obligation is left pending in $O_S$.

### 3.11 Architecture

The overall architecture of the CCC is based on the ECA (Event Condition Action) paradigm and shown in Fig. 7.

The $\text{ROP}$ sets store the current set of rights obligations and prohibitions of the buyer and seller.

The $\text{bevent logger}$ is a permanent storage for keeping records about all the event processed by the CCC.

The $\text{bevent queue}$ is a queue that stores bevents until they are removed for processing by the relevance engine.

The $\text{contract compliance checker}$ keeps track of deadlines associated to each right, obligation and prohibition stored in the $\text{ROP}$ sets. Deadlines are set and reset (set/reset deadline TO) by the relevance engine. When a deadline expires, a $\text{timeout bevent}$ is sent to the filter $\text{mism. bo}$.

Bevents that arrive from the monitoring channel pass through two filtering mechanisms before they can be stored in the $\text{bevent queue}$. The filter for unknown operations ($\text{filter unkn. bo}$) is responsible for filtering out bevents that correspond to unknown business operations. Such bevents are sent to the bevent logger for auditing. The filter for mismatched operations ($\text{filter mism. bo}$) is responsible for filtering out bevents that correspond to mismatched business operations and for sending them to the bevent logger. As shown in the figure, timeout bevents are also examined by the filter for mismatched operations before they can reach the bevent queue.

A $\text{timeout bevent}$ should be filtered out if the corresponding $bo_i$ is in the bevent queue or is currently being processed by the relevance engine; similarly, a bevent arriving from the monitoring channel should be filtered out if a corresponding timeout bevent is in the bevent queue or is currently being processed by the relevance engine. In summary, the intention of the filtering mechanisms is to detect and discard bevents.
that correspond to non-contract compliant operations as early as possible; thus the relevance engine deals only with bevents that are likely to be contract-compliant.

The job of the relevance engine is to remove the bevent from the head of the bevent queue, perform a rule matching of the bevent against the rule base and trigger relevant rules. Notice that, in accordance with the definitions stated in Section 3.9, bevents that trigger rules correspond to contract-compliant business operations. Again, records about the bevent are stored in the bevent logger.

4 USES OF THE MODEL

4.1 Contract specification

The CCC model with ROP sets provides a relatively easy way of constructing ECA rules to check contract compliance. To illustrate we will convert the clauses of our running example into ECA rules and explain how the rules are used by the CCC to manipulate the ROP sets. In the simplest case, clauses and rules correspond one-to-one, yet in general, the mapping is N→N.

We will use the following acronyms: S: Success, BF: BizFail, TF: TecFail, c: business event, d: day, OB: buyer’s set of obligations, OS: seller’s set of obligations, SubPO: submission of PO, AcceptPO: acceptance of PO, RejectPO: rejection of PO, SubInv: submission of invoice, SubPay: submission of payment, Delivery: delivery of goods. From the contract, we can derive RP = \{buyer, seller\}, and B = \{SubPO, AcceptPO, RejectPO, SubInv, SubPay, Delivery\}.

Our ECA rules are expressed as follows:
\[ e \equiv (\text{bevent}, \{c_1\}, \ldots, \{c_m\}) \rightarrow \{a_1\}, \ldots, \{a_m\} \] where \(c_1, \ldots, c_m\) are mutually exclusive conditions for compliance checking and \(a_1, \ldots, a_m\) are actions. When the event \text{bevent} arrives and condition \(c_i\) holds, action \(a_i\) is executed. In general conditions and actions can be composite in that they might consist of several primitive conditions and actions, respectively. Also, conditions that always evaluate to true can be omitted; this results in simple rules of the form \(e \equiv \text{bevent} \rightarrow a\).

Rules are organised and commented (#) with Fig. 6 in mind. Notice that except for \(s_0\), we show only the rules whose execution result in ROP sets manipulation and therefore, in a change of state. For instance, the rules for processing the bevents (AcceptPO, BizFail), (AcceptPO, TecFail) and (AcceptPO, InitFail) which correspond to state \(s_1\) are not shown as they are of little interest since they have an empty right hand side (no actions taken). To appreciate the job of the relevance engine (see Section 3.11), we indicate what matching (\(\triangleright\)) needs to be performed to make a rule potentially executable.

# State \(s_0\) corresponds to clause C1:
# SPO-S: potentially executable when exec of oper. SubPO
# produces S and SubPO \(\triangleright\) RB = T
# brings CCC to intermediate state \(s_1\)
\[ e \equiv (\text{name} = \text{SubPO}, \text{exoutcome} = \text{S}, \text{initiator} = \text{buyer}, \text{initiator} = \text{buyer}, \text{initiator} = \text{buyer}) \]
\[ \{\text{e.ts} \in \{\text{Mon, ..., Fri}\} \land \text{e.ts} \in [9, 17]\} \rightarrow \{\text{R}_B = \text{SubPO}, \text{O}_S = \text{RespondPO}, \text{O}_T = 24\text{h}\}\]

# SPO_BF: potentially executable when exec of oper. SubPO
# produces BF and SubPO \(\triangleright\) RB = T, CCC remains in \(s_0\)
\[ e \equiv (\text{name} = \text{SubPO}, \text{exoutcome} = \text{BF}, \text{initiator} = \text{buyer}, \text{initiator} = \text{buyer}) \]
\[ \{\text{e.ts} \in \{\text{Mon, ..., Fri}\} \land \text{e.ts} \in [9, 17]\} \rightarrow \{\text{\}}\]

# SPO_TF: potentially executable when exec of oper. SubPO
# produces TF and SubPO \(\triangleright\) RB = T, CCC remains in \(s_0\)
\[ e \equiv (\text{name} = \text{SubPO}, \text{exoutcome} = \text{TF}, \text{initiator} = \text{buyer}, \text{initiator} = \text{buyer}) \]
\[ \{\text{e.ts} \in \{\text{Mon, ..., Fri}\} \land \text{e.ts} \in [9, 17]\} \rightarrow \{\text{\}}\]

# SPO_InitF: potentially executable when exec of SubPO
# produces InitF and SubPO \(\triangleright\) RB = T, CCC remains in \(s_0\)
\[ e \equiv (\text{name} = \text{SubPO}, \text{exoutcome} = \text{InitF}, \text{initiator} = \text{buyer}, \text{initiator} = \text{buyer}) \]
\[ \{\text{e.ts} \in \{\text{Mon, ..., Fri}\} \land \text{e.ts} \in [9, 17]\} \rightarrow \{\text{\}}\]

# State \(s_1\) corresponds to clause C2:
# APO-S: potentially executable when exec of oper. AcceptPO
# produces S and AcceptPO \(\triangleright\) O_S = T
# brings CCC to intermediate state \(s_4\)
\[ e \equiv (\text{name} = \text{AcceptPO}, \text{exoutcome} = \text{S}, \text{initiator} = \text{seller}, \text{initiator} = \text{seller}) \]
\[ \{\text{\}} \rightarrow \{\text{O}_S = \text{RespondPO}, \text{O}_S = \text{SubInv}, \text{O}_T = 24\text{h}\}\]

# RPO-S: potentially executable when exec of oper. RejectPO
# produces S and RejectPO \(\triangleright\) O_S = T
# brings CCC to normal final state \(s_2\)
\[ e \equiv (\text{name} = \text{RejectPO}, \text{exoutcome} = \text{S}, \text{initiator} = \text{seller}, \text{initiator} = \text{seller}) \]
\[ \{\text{\}} \rightarrow \{\text{O}_S = \text{RespondPO}\} \]

# State \(s_2\) corresponds to clause C4:
# SLInv-S: potentially executable when exec of oper. SubInv
# produces S and SubInv \(\triangleright\) O_S = T
# brings CCC intermediate state \(s_6\)
\[ e \equiv (\text{name} = \text{SubInv}, \text{exoutcome} = \text{S}, \text{initiator} = \text{seller}) \]
\[ \{\text{\}} \rightarrow \{\text{O}_S = \text{SubInv}, \text{O}_S = \text{SubPay}, \text{O}_T = 24\text{h}\}\]

# State \(s_4\) corresponds to clause C5:
# SPay-S: potentially executable when exec of oper. SubPay
# produces S and SubPay \(\triangleright\) O_S = T
# brings CCC intermediate state \(s_8\)
\[ e \equiv (\text{name} = \text{SubPay}, \text{exoutcome} = \text{S}, \text{initiator} = \text{buyer}) \]
\[ \{\text{\}} \rightarrow \{\text{O}_S = \text{SubPay}, \text{O}_S = \text{Delivery}, \text{O}_T = 24\text{h}\}\]

# State \(s_2\) corresponds to clause C6:
# Dlvry-S: potentially executable when exec of oper. Delivery
# produces S and Delivery \(\triangleright\) O_S = T
# brings CCC to normal final state \(s_{10}\)
\[ e \equiv (\text{name} = \text{Delivery}, \text{exoutcome} = \text{S}, \text{initiator} = \text{seller}) \]
\[ \{\text{\}} \rightarrow \{\text{O}_S = \text{Delivery}, \text{O}_S = \text{Delivery}, \text{O}_T = 24\text{h}\}\]
4.2 Exception Handling Support

Exceptional clauses normally specify sanctions which are obligations that come in force when the primary obligations are violated (not fulfilled). In electronic contracting, it is particularly important to distinguish violations caused by infrastructure level problems: situations that arise primarily because of the inherently distributed nature of the underlying computations from those that are not and are mostly human/organisation related.

Take a clause $C_5$ as an example and imagine that the buyer fails to provide the payment before the stipulated seven day deadline. It makes sense to distinguish cases where the missing or delayed payment is owing to some infrastructure related problem (say a network breakdown) from cases where no such problems existed (so probably the buyer was just late or deliberately avoiding payment); ideally, a sanction (such as a fine) should not be imposed on the buyer in the former case, instead actions such as extending the deadline should be undertaken. Since failures to successfully execute operations are reported to the CCC as business events, it is possible to handle exceptional situations at rule level very conveniently and for the CCC to clearly decide what sanctions to impose in different situations, and to monitor that they are applied correctly [10].

As it is, our buyer–seller contract has no provision for handling potential extraordinary circumstances that can prevent the buyer from fulfilling his payment obligations. To address this issue we suggest that contracts include, together with primary clauses, contingency ones as shown in the new version of clause $C_5$:

- $C_5$: The buyer has the obligation to pay the due amount within seven days of receiving the invoice.
  - $C_5.1$: The buyer shall be granted a seven day extension to honour his payment obligation if he missed his deadline due to technical or business failures. At the same time, the seller is granted the right to cancel the purchase order.
  - $C_5.2$: Under normal circumstances (no failures), no deadline extensions to pay shall be granted, thus pending payments and potential disputes shall be sorted out off-line.
  - $C_5.3$: The seller is obliged to refund payments received after cancellations.
  - $C_5.4$: The buyer and seller are obliged to stop the transaction upon the detection of three failures to submit payment and solve potential conflicts off-line.

The mapping of these clauses into rules is shown below. In this setting, two additional operations $\text{CancelPO}$ and $\text{Refund}$ are included, consequently $B = \{ \text{SubPO}, \text{AcceptPO}, \text{RejectPO}, \text{SubInv}, \text{SubPay}, \text{Delivery}, \text{CancelPO}, \text{Refund} \}$. We organise rules with Fig. 6 in mind and show the rules to handle the four potential bevents that can be generated from the execution (or time out expiry) of the $\text{SubPay}$ operation that takes place in state $s_0$.

A Boolean variable $\text{PayExt}$, initially False, records if the payment deadline has been extended or not. Rule $\text{SubPay}_S$ uses $\text{PayExt}$ to determine if the payment has been made under normal circumstances ($\text{PayExt} = F$) or under a deadline extension ($\text{PayExt} = T$). If under extension, the query happened($\text{CancelPO}, S$), which queries the event log, is used to check whether a successful execution of $\text{CancelPO}$ operation took place before the successful execution of the $\text{SubPay}$ operation. If the response to the query is affirmative, the rule imposes the obligation $\text{Refund}$ on the seller to encourage him to return the payment, otherwise, the rule imposes the obligation $\text{Delivery}$ on the seller.

### State $s_0$ corresponds to clause $C_5$:
- $\text{SPay}_S$: potentially executable when exec of oper. $\text{SubPay}$ produces $\# T$ and $\text{SubPay} \vdash \mathcal{O}_B = T$.
- $\text{PayExt} = F | T$: pay within normal | extended deadline, respec.
  - $e \equiv (\text{name} = \text{SubPay}, \text{exoutcome} = S, \text{initiator} = \text{buyer}, -)$,
    $\{ \text{PayExt} = F \} \rightarrow \{ \mathcal{O}_B = (\text{SubPay}), \mathcal{O}_S = (\text{Delivery}, \text{7d}) \}$
  - $\text{Deadline extended and seller has not cancelled PO}$
    $\{ \text{PayExt} = T \} \land \text{happened}(\text{CancelPO}, S) \rightarrow \{ \mathcal{O}_B = (\text{SubPay}), \mathcal{O}_S = (\text{Delivery}, \text{7d}) \}$
  - $\text{Deadline extended and seller has cancelled PO}$
    $\{ \text{PayExt} = T \land \text{happened}(\text{CancelPO}, S) \rightarrow \{ \mathcal{O}_B = (\text{SubPay}), \mathcal{O}_S = (\text{Refund}) \}$

### State $s_0'$ corresponds to clause $C_5$:
- $\text{SPay}_S$: potentially executable when exec of oper. $\text{SubPay}$ produces $\# F$ and $\text{SubPay} \vdash \mathcal{O}_B = T$.
  - $e \equiv (\text{name} = \text{SubPay}, \text{exoutcome} = S, \text{initiator} = \text{buyer}, -)$,
    $\{ \text{PayExt} = F \} \rightarrow \{ \mathcal{O}_B = (\text{SubPay}), \mathcal{O}_S = (\text{Delivery}, \text{7d}) \}$
  - $\text{Deadline extended and seller has not cancelled PO}$
    $\{ \text{PayExt} = T \} \land \text{happened}(\text{CancelPO}, S) \rightarrow \{ \mathcal{O}_B = (\text{SubPay}), \mathcal{O}_S = (\text{Refund}) \}$

### State $s_0''$ corresponds to clause $C_5$:
- $\text{SPay}_S$: potentially executable when exec of oper. $\text{SubPay}$ produces $\# T$ and $\text{SubPay} \vdash \mathcal{O}_B = T$
  - $e \equiv (\text{name} = \text{SubPay}, \text{exoutcome} = T, \text{initiator} = \text{buyer}, -)$,
    $\{ \text{NumFail} < 3 \} \rightarrow \{ \mathcal{O}_B = (\text{SubPay}) \}$
  - $\text{Deadline extended and seller has cancelled PO}$
    $\{ \text{PayExt} = T \land \text{happened}(\text{CancelPO}, S) \rightarrow \{ \mathcal{O}_B = (\text{SubPay}) \}$

### State $s_0'''$ corresponds to clause $C_5$:
- $\text{SPay}_S$: potentially executable when exec of oper. $\text{SubPay}$ produces $\# F$ and $\text{SubPay} \vdash \mathcal{O}_B = T$
  - $e \equiv (\text{name} = \text{SubPay}, \text{exoutcome} = T, \text{initiator} = \text{buyer}, -)$,
    $\{ \text{NumFail} < 3 \} \rightarrow \{ \mathcal{O}_B = (\text{SubPay}) \}$
  - $\text{Deadline extended and seller has cancelled PO}$
    $\{ \text{PayExt} = T \land \text{happened}(\text{CancelPO}, S) \rightarrow \{ \mathcal{O}_B = (\text{SubPay}) \}$

### State $s_0''''$ corresponds to clause $C_5$:
- $\text{SPay}_S$: potentially executable when exec of oper. $\text{SubPay}$ produces $\# T$ and $\text{SubPay} \vdash \mathcal{O}_B = T$
  - $e \equiv (\text{name} = \text{SubPay}, \text{exoutcome} = T, \text{initiator} = \text{buyer}, -)$,
    $\{ \text{NumFail} < 3 \} \rightarrow \{ \mathcal{O}_B = (\text{SubPay}) \}$
  - $\text{Deadline extended and seller has cancelled PO}$
    $\{ \text{PayExt} = T \land \text{happened}(\text{CancelPO}, S) \rightarrow \{ \mathcal{O}_B = (\text{SubPay}) \}$

To implement clause 5.4, a variable $\text{NumFail}$, initialised to zero, is used to record the number of business and technical failures (here we count an initiation failure also as a part of a technical failure); this variable is incre-
mented in rules $SPay\text{ InitF}$, $SPay\text{ BF}$ and $SPay\text{ TF}$; when $NumFail$ exceeds 3, these rules execute an Exit command to record final, abnormal state. A seven day deadline extension to honour $SubPay$ obligation is granted to the buyer by rule $SPay\text{ TO}$ when the obligation is timed out and there are records of payment attempt that resulted in business or technical failures; $PayExtd$ is set to true to indicate this. The rule also grants the seller the right to execute a $CancelPO$ operation to cancel the PO.

Fig. 8 shows four possible timelines of the execution of $SubPay$ operation. In a) the execution of the $SubPay$ operation succeeds in the first attempt within the seven day deadline (7d). In b), the buyer fails once due to a BizFail, so a 7d deadline extension is granted to him, and the right to cancel the PO is granted to the seller. The buyer succeeds in his second attempt to pay —($SubPayS$) while the seller decides not execute $CancelPO$. In c), the operation $SubPay$ fails three times —a ($SubPayTF$) followed by two ($SubPayBF$) — without cancellation from the seller, so the business transaction is stopped at $NumFail = 3$. In d) the payment succeeds in the second attempt —($SubPayS$) while the seller successfully exercises his right to execute $CancelPO$ after the buyer’s first attempt to pay fails —($SubPayBF$) —; if the operations of execution $SubPay$ and $CancelPO$ overlap, it is possible that (as shown in the figure) the event ($SubPayS$) is processed after ($CancelPOS$); consequently, the seller needs to execute a $Refund$ operation to return the money, in the figure this operation is executed successfully —($RefundS$).

![Fig. 8. Deadline extension due to failures.](image)

4.3 Contract Enforcement

The CCC, which has been assumed so far as a passive observer, can be enhanced to act as a contract enforcer that ensures that a business operation is executed only if it is contract–compliant in accordance with the contractual clauses. For this functionality, the enforcer will need to intercept the initiation that precede the execution of operations (see Fig. 4) and prevent initiation of operations that are unknown, mismatched or out of context (as defined in Section 3.9). The degree of control exercised can be a matter of design choice; for example, as discussed in [11] the enforcer can be made proactive whereby it can remind partners of their obligations well before the deadlines expire. Our CCC provides a very efficient basis for implementing such a proactive enforcer (which can also be seen as a contract compliant business process coordinator). Whenever the ROP sets change, the enhanced CCC can inform the partners of the change, thereby ensuring that the partners have a mutually consistent knowledge of their respective sets.

5 IMPLEMENTATION

The ECA notation we have used in our illustrative examples hints at the type of contract representation language that our model can easily support. We have indeed designed such a language [12] called EROP (for Events, Rights, Obligations and Prohibitions) for the CCC, that provides constructs to specify what rights, obligation and prohibitions become active and inactive after the occurrence of events related to the execution of business operations. The core components of the CCC (relevance engine, contract rules, event queue, event logger and timer) have been implemented. The service relies on the JBoss Rules engine, also known as Drools [13], for the decision capabilities of the relevance engine and for rule management. Additional Java components for Drools implement the functionality required for the manipulation of ROP sets, historical queries and timer management, using Java statements within an augmented version of the Drools rule language. These implementation details are described in [14]. This implementation and the EROP language serve to illustrate the practical basis of the model presented in this paper.

6 RELATED WORK

Contract monitoring and/or enforcement at run time has been addressed by several researchers. One of the earliest works in this direction is on LGI (Law-Governed Interaction) [15]. LGI is a ‘law enforcer’ that regulates the interaction between two (e.g., buyer and seller) or more autonomous and distributed agents linked by a communication network. A controller instrumented with the law (e.g., contractual clauses) is placed between each agent and the network to intercept and filter out incoming or outgoing messages that are incompatible with the law, keep the agent’s state in synchrony with other agents, verify certain conditions, and execute relevant actions to enforce the law imposed on the agent. LGI system architecture is peer–to–peer in the sense that each participant is required to run an instance of LGI, whereas we have examined compliance checking from the view point of a ‘third party’. Further, unlike our work, timing and message validity constraints that are an essential part of B2B messaging are not considered in LGI.

The idea of providing electronic contracts with discretionary mechanisms to react (as opposite to prevent) to contract violations in accordance with their causes (accidental, due to force majeure, intentional, etc.) is discussed in [16]; however, the focus of this work is on settlement of potential disputes after the occurrence contract violations related to non-functional requirements.
Essential concepts underlying contract monitoring are presented in [17] from the perspective of a model driven approach. The paper presents a metamodel level discussion on a variety of topics, including sub-contracting, simultaneous execution of several interleaving contract instances, nested executions, multiple monitoring, and so on. Our work can be seen as a concrete instance that broadly conforms to some of the metamodels of the paper. In our concrete model we address various practical issues concerned with partner state alignment, deadlines and fault tolerance that are not discussed in the metamodels.

Approaches to contract specification based on the use of Deontic Logic has been suggested by many authors (for example, [18], [19], [20], [21]). With this approach policy statements are specified in a Deontic Logic notation to define the permissions, prohibitions, obligations, actions, and temporal and non-temporal conditions that a role player needs to fulfill to satisfy a contract. Such an approach provides a way for a contract designer to verify (with the assistance of software tools) that the contract is free from temporal and Deontic inconsistencies. A limitation is that standard Deontic Logic is static in the sense that it cannot describe permissions, obligations and prohibitions that become and cease to be in effect depending on the occurrence of time and other events. There is ongoing work exploring the possibility of enhancing Deontic Logic with additional logical constructs to overcome its limitations, for instance, to incorporate constructs from Modal Logic, Temporal Logic, Logic of Action or from their combinations ( [22], [23]). We have found the ECA based approach of our model more intuitive to use.

We acknowledge that we are not the first authors to suggest an event-centric approach (also called, history-based) to model contracts. In [24], for instance, an event-centric mechanism is used to monitor contractual service level agreements. Relevant to our work are their activate and deactivate operations applied to contractual norms (ECA rules). These operations produce a similar effect as our addition and deletion on ROP sets, except that our addition and deletion operations are applied to individual rights, obligations and prohibitions rather than to the whole rule. The actual computation model is event calculus-based, so it offers the user built-in primitives for querying at run time what norms are active and inactive. In this respect our model bears a strong similarity; the examination of the buyer’s and seller’s current sets of rights, obligations and prohibitions in our model, would produce similar results. Another ECA based contract enforcement is discussed in [25]. The central idea here is to mediate the interaction between each pair of communication objects by an enforcing Synchronization Point (SP). This SP is basically an ECA engine responsible for receiving events, evaluating conditions and executing actions. A feature of this work is the explicit modelling of the minimum and maximum number of occurrences of events (e.g. maximum number of attempts to pay by credit card). However, the rule notation suffers somewhat as it relies on rules cross-referring each other; in the same order, given a rule, it is not possible to know if it represents a right, obligation or prohibitions by examining it. Exception handling issues are not addressed in that paper.

Our work has been influenced by research reported in [26] where the authors suggest an obligation and right based model to design electronic contracts that can be monitored at runtime. Like in our work, the authors here consider that some rights and obligations become in force dynamically as the contract is executed; hence they distinguish between background (always in force) and state-based (in force only in the current state) rights and obligations; like us, they rely on add and remove operations to modify the set of state-based right and obligations. Heimdhal [27] is an obligation enforcer focused on the enforcement of resource usage policies such as “No execution should last more than one second”. It is designed under the assumption that all actions are allowed and (if necessary) compensatory; likewise all resources are assumed to be preemptive (e.g., “Abort a given job after 3 sec of execution to free CPU”). Consequently, unlike our work, the notion of rights and prohibitions are not of concern in Heimdhal. The architecture of the CCC bears some similarities to Heimdhal’s, except that pending obligations in Heimdhal are recorded in a history log whereas in our work we express them explicitly in our ROP sets.

Conformance checking of process behaviour against choreography specification is discussed in [28]. The goal of this work is to verify if a given sequence of messages (or business activities) extracted from a log file created during a business interaction conforms to the expected behaviour of the parties as specified (in BPEL for example) in the business process. The main idea here is to build a Petri Net model of the business process and then to replay the sequence of messages against it to see if the sequence of messages is possible according to the model; that is, if such a sequence can be generated by the model. We regard this work as complimentary to ours.

A general overview of the four phases (information, pre-contracting, contracting and enactments) that an electronic contracting process involves is presented in [29]. With respect to this taxonomy, our work focuses on the enactment phase and in particular, in the monitoring and control activity; our intention is to provide a concrete solution to monitor business activity and collect historical records that, if necessary, can assist in offline evaluation of contract compliance and in dispute resolution.

The use of events and rules in business process monitoring is also discussed in [30]. The interest here is in monitoring business situations as seen from a single party. The authors did not consider the possibility of conflicting outcomes; for example, where an order is shipped in time but rejected by the shopper because
he considers that it did not arrive in time. In [31] an event-driven-architecture for cross-organisational business processes is discussed; events are used to model normal and exceptional outcomes; however, exceptional outcomes covers only what we call business failures, that is, technical failures are not addressed. The need of exception handling in Web service composition is recognised in [32]. However, the computation model here is client–server whereas we consider buyer and seller in peer-to-peer relationship. Exception handling in workflow executions is discussed in [33]. Again, we depart from this work as we focus on peer-to-peer interactions. Exception handling is central to the contract enforcers discussed in [34] and [35]; however, the exceptions of interest in these works are undesirable business situations (e.g., bounced check, payment deadline ignored, etc.) that emerge due to business reasons; in this order, actions executed in response to events signalling the missing of obligation deadlines are regarded as raised exceptions; these works do not consider exceptions due to technical reasons.

The approaches discussed so far focus on bilateral contracts; work on multi–party contracts has been reported in [36]. A contract is regarded as a set of actions (e.g. SendInvoice) to be executed (under the observance of some occurrence order) to fulfil a set of commitments (equivalent to contract rules) established between pairs of participants in the multi–party contractual relationship. An algorithm has been developed for contract monitoring; for instance, for detecting the parties that missed executing an action that resulted in a violation of the contract. The approach used does not distinguish between rights, obligations and prohibitions: commitments cannot encode the notion of permissible, obligatory or forbidden actions. Further, as it is, the contract model does not consider exception handling aspects that we have highlighted in our model.

7 Concluding Remarks

The CCC is a neutral entity (conceptually located between the interacting parties) whose function is to observe B2B interaction events and infer from them whether the business operations these events relate to are contract compliant or not. This way it is able to track the state of the B2B interaction and determine whether the final end state is normal or abnormal (contract–violation has occurred). Business partners are required to implement business interactions according to the execution model of business operation discussed in Section 2.5. In practice this would require —in case of RosettaNet based implementation— wrapping PIP conversations with explicit outcome synchronization. We discussed how to cope with situations arising due to protocol level errors (message delays, loss), invalid business messages and deadlines in a manner that enables us to formulate business rules in a concise way for expressing contractual clauses, especially those that are concerned with stating alternative course of actions.

Notice that depending on where and how conditions are expressed and verified, the designer can take different alternatives for converting clauses into rules. The approach taken impacts the functionality and complexity of the relevance engine. In our approach (see R1), all the conditions related to event attributes (e.g., valid day to submit a purchase order) are encoded within the rule. However, the condition related to the role player (e.g., has the buyer the right to submit a purchase order?) is encoded in the ROP sets. Thus, upon receiving an event, the relevance engine needs to inspect the ROP sets and possibly, the rules. Another alternative would be to build a more sophisticated relevance engine capable of updating the ROP sets (e.g., adding a right to submit a purchase order on Monday and removing it on Friday) on the basis of information deduced from the conditions encoded in the rule.

We have argued that exceptional clauses in electronic contracts should be structured so as to take account of infrastructure level problems. Our model provides simple ways to distinguish violations caused by such problems.

We have presented the CCC architecture in terms of the constituent components for business rule specification, event management and logging in sufficient detail to enable development of practical contract representation languages and third party services for monitoring and enforcing contract compliance.

Our own language design and implementation effort, briefly mentioned here and described in more detail elsewhere ([12] and [14], respectively) serves to illustrate the practical basis of the model. Future work will involve extending the CCC to work as a contract enforcer along the lines hinted here. An interesting line of research would be to examine whether the model needs to be extended to handle long transactions. One could imagine that long transactions would be a useful way of structuring the public business process, in which case the model may well be requested to track commit and abort events, and relate them to clauses. Another avenue for future research is the verification that the CCC itself functions correctly; this in turn implies verifying the correctness of the contract and the actual implementation of the CCC. Since the CCC is a reactive system, we are able to use conventional model–checking techniques to validate the ECA–rules against both general (redundancy, conflicts, subsumed rules, unexpected events, etc.) and contract–dependent correctness requirements (e.g., deadline extensions are granted exactly as stated in the clauses) [37]. This work can be further extended in two directions: (i) the rules implemented within the validated model could form the basis for automatically generating EROP version of the contract; and (ii) the event sequences produced from model–checking runs can be used as executable test cases for testing the actual implementation of the CCC.
ACKNOWLEDGMENT

This work is funded in part by UK EPSRC Platform Grant No. EP/D037743/1, Networked Computing in Inter-organisation Settings and Knowledge Transfer Secondments award EP/H500332/1 to Arjuna Technologies, Ltd.

REFERENCES


Carlos Molina–Jimenez is currently a senior research associate at the School of Computing Science of Newcastle University, UK. His research interests lie in distributed systems and are currently focused on middleware for supporting contract-regulated business applications, policy enforcement and formal validation of business processes. He was awarded a PhD degree in computing Science from Newcastle University, UK. He is currently with Arjuna Technologies Ltd., under a Knowledge Transfer Secondments award.

Santosh Shrivastava Santosh K. Shrivastava received the PhD degree in computer science from Cambridge, in 1975. He was appointed as a professor of computer science at the University of Newcastle upon Tyne in 1986. His research interests span many areas of distributed computing, including middleware, transaction processing and fault tolerance. His current focus is on middleware and services for supporting interorganizational services, where issues of trust, security, fault tolerance, and ensuring compliance to service contracts are of great importance.

Massimo Strano received the BSc degree from La Sapienza University of Rome, Italy, in 2002, and the PhD degree from Newcastle University, UK. He is currently a Senior Software Engineer for Red Hat UK. His research interests include representation and execution of electronic contracts, contract compliance of business interactions, business to business integration and dynamic reconfiguration of business processes.