A Toolkit for model checking of electronic contracts

Thesis by

Abubkr A. Abdelsadiq

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

Newcastle University

Newcastle upon Tyne, UK

January 2013

Abstract

In the business world, contracts are used to regulate business interactions between trading parties. In this context, an electronic contracting systems can be used to monitor business–to–business interactions to ensure that they comply with the rights (permissions), obligations and prohibitions stipulated in contract clauses. Such an electronic contracting system will require an executable version of the contract (e-contract) for compliance checking. It is important to be verify the correctness properties of an e-contract before deploying it for compliance checking. Model checkers are widely used for automatic verification of concurrent systems. However, such tools for e-contracts with means for expressing directly and intuitively key concepts that appear recurrently in contracts, such as executions of business operations, granting (cancellation, suspension, fulfilment, violation, etc.) of rights, obligations and prohibitions to role players are not yet available.

This thesis rectifies the situation by developing a high-level e-contract verification toolkit using the Spin model checker. A formal Contractual Business-To-Business interaction (CB2B) model based on the concepts of contract compliance checking developed earlier at Newcastle university has been constructed. Further, Promela, the input language of the Spin model checker, has been extended in a manner that enables specification of contract clauses in terms of contract entities: role players, business operations, rights, obligations and prohibitions. A given contract can now be expressed using extended Promela as a set of declarations and a set of Event-Condition-Action rules. In addition, the designer can specify the correctness requirements to be verified in Linear-Temporal-Logic directly in terms of the contract entities. A notable feature is that the CB2B model automatically checks for contract independent properties: properties that must hold for all contracts. For example, at run time, a contract should not simultaneously grant a role player a right to perform an operation and also prohibit it. Thus, the toolkit hides much of the intricate details of dealing with Promela processes communicating through channels and enables a designer to build verifiable abstract models directly in terms of contract entities.

The usefulness of the toolkit is demonstrated by trying out a number of contract examples used by researchers working on contract verification. The thesis also shows how the toolkit can be used for generating test cases for testing an implemented system.

Acknowledgment

It is a pleasure to express my sincere gratitude and respect to my supervisor Prof. Santosh Shrivastava. I would like to thank him for his support and assistance that made this thesis possible. His advice has been invaluable, and his patience in helping me clarify and express ideas and plans has been very influential during my study. My gratitude is also extended to Dr. Carlos Molina-Jimenez for his advice, and for his willingness for deep discussions and provide feedback during our meetings. I am very grateful to the Systems group in the computing department at Newcastle University for giving me the opportunity to attend their talks and learn from them.

Lastly, and most importantly, I am infinitely grateful to my family, my wife Samira, my son Mohamed and my little daughter Monia for their unconditional love, endless patience and encouragement when it was most required. I also thank my grandparents Salem and Rahma who always encouraged me to pursue my ambitions and supported me throughout my life.

Declaration

I declare that this thesis is my own work and it has not been previously submitted, either by me or by anyone else, for a degree or diploma at any educational institute, school or university. To the best of my knowledge, this thesis does not contain any previously published work, except where the person's work has been cited and included in the list of references. Some of the material in this thesis has been published in conference proceedings, below is the list of publication.

1. A. Abdelsadiq, C. Molina-Jimenez, and S. Shrivastava. A high-level model–checking tool for verifying service agreements. In: Proceedings of the 6th IEEE International Symposium on Service Oriented System Engineering (SOSE 2011). Irvine, CA, Nov. 2011, pp. 297-304.

2. A. Abdelsadiq, C. Molina-Jimenez, and S. Shrivastava. On Model Checker Based Testing of Electronic Contracting Systems. In: Proceedings of the 12th IEEE Conf. on Commerce and Enterprise Computing (CEC 2010). Shanghai, Dec. 2010, pp. 88-95.

Contents

List of Figures

Chapter 1

Introduction

Fulfilling a given business function such as order processing electronically requires business partners to exchange electronic business documents and to act on them. Naturally, the exchanges and actions undertaken need to comply with the business agreement (contract) currently in force between the partners. An agreed on contract normally specifies the obligations, permissions (rights) and prohibitions that the signatories should be held responsible for and states the actions or penalties that may be taken when any of the stated agreements are not met.

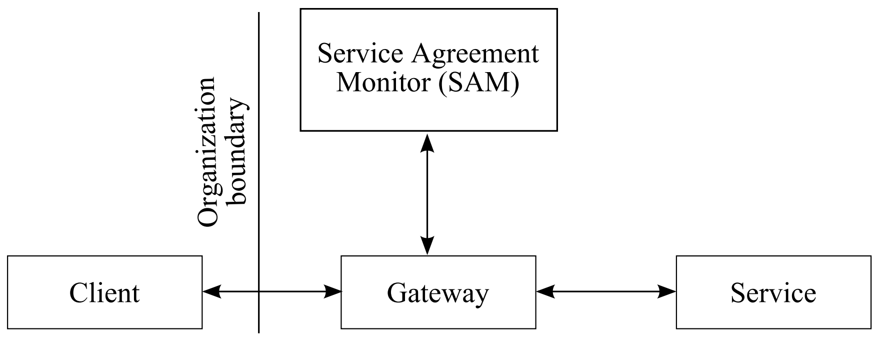


Figure 1.1: Service agreement monitoring

Contract compliance checking can be automated with the help of electronic contracting systems that can be used for detecting violations, facilitating dispute resolution and determining liability by providing an audit trail of business interactions.

Figure 1.1 shows an example of a contract system where a service provider uses a SAM for monitoring the service agreement configuration. Essentially, the service agreement monitor will require machine interpretable specifications of relevant parts of the service agreement. The provider uses a service agreement monitor loaded with an executable version of the service agreement (e-contract). The gateway acts as a check-point that can determine whether or not the client's actions are contract compliant as informed by the service agreement monitor. Similarly, it can prevent the use of the service in cases of agreement violations.

However, the intended meaning of contract clauses expressed in a natural language can be remarkably hard to capture and represent in a rigorous and concise manner for computer processing. Consider a hypothetical service agreement signed between a client and a provider of storage services, with the following clauses:

1. The *Client* is entitled to *use* the service in normal quota mode of 100 GB or exceeded quota mode of 120 GB.
2. *Clients* that exhaust their normal quota are obliged to either:
   (a) *Submit* a single payment of £5 within the next 72 hours.
   (b) *Bring* the quota back to normal within the next 72 hours by deleting sufficient number of files.
3. Violation of clause 2 will give the *Service Provider* the right of suspension.

A contract could contain inconsistencies in the form of conflicting clauses. A typical type of contract conflict occurs when a contract participant is simultaneously obliged and forbidden to perform the same action, or is both permitted and forbidden to perform the same action. Recalling the above contract clause, it will be considered as a conflict if actions such as 'use', 'bring quota back', or 'suspend' are characterised as permitted and prohibited or obliged and prohibited at the same time. The need to analyse and reason about contracts is therefore extremely important. Thus, there is a strong case for developing tools for contract validation and verification.

The clauses of a natural language contract include a combination of role players, business operations, rights, obligations and prohibitions, as well as their temporal aspects. Furthermore, the language clauses also specify exceptional behaviours arising from deviation from the original obligations (the required actions in case an obligation is not fulfilled) and of prohibitions (the required actions in case a prohibited action is performed). The actions that come into force where exceptional behaviours take place can be regarded as reparations and are known respectively as contrary-to-duties (CTDs) and contrary-to-prohibitions (CTPs).

A key requirement of an e-contract language is to capture the main elements of natural language contracts such as role players, business operations, rights, obligations and prohibitions and allow their representation in a machine interpretable notation. For this purpose, several notations for the specification of contractual clauses have been suggested in the literature. For example, some have chosen to encode contract clauses as event–condition–action (ECA) rules because of their widespread usage in the business world for representing business agreements. Others have chosen to use a logic-based solution, and the spectrum of contract notation types is reviewed in Chapter 2 below. However, inconsistencies can be accidently introduced into an executable contact, which might cause the monitoring facilities of contract compliance or contract enactment services to malfunction, as well as possibly leading to conflict between the contract parties.

Model checking is a widely accepted and useful technique that is actively applied for finding bugs in software and hardware designs. The essential idea of model checking is depicted in Figure 1.2. Typically, model checkers build a reachability graph from finite state descriptions of the system and search for violations of the properties such as temporal specifications under investigation by exhaustively exploring the reachable state space. The key element of model checking is that, if there are one or more execution traces or sequences in the system under analysis that do not satisfy a given temporal formula, then at least one of the offending execution-traces will be returned as a counter-example which can be used to trace the source of the error. If no counterexamples are returned, then one can claim that all executions of the system satisfy the properties prescribed in the temporal formula.

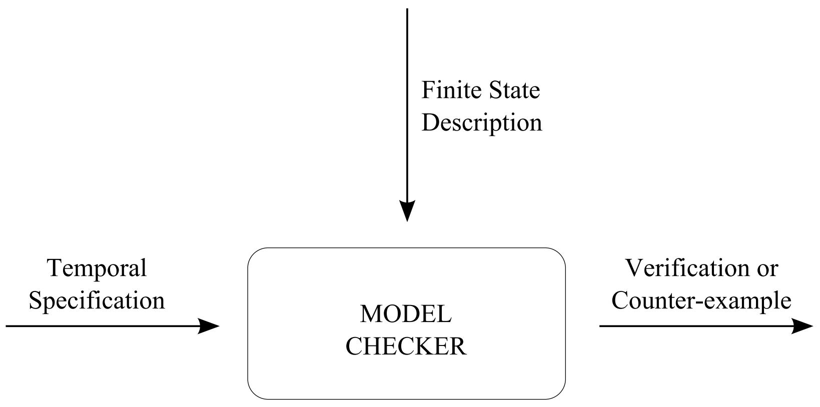


Figure 1.2: Model Checking

In order to simplify the mechanical verification of e-contract models with general purpose model checking tools, it is highly desirable that the verification tools are supported with means for directly and intuitively expressing key concepts that usually appear in contracts, such as the execution of business operations, or the granting, cancellation, suspension, fullfilment, or violation of rights, obligations and prohibitions to role players. Unfortunately model checking tools with these highly desirable constructs are not available yet. A possible but daunting and time consuming way to address the problem is to build such a tool from scratch. A more pragmatic alternative is to build such a tool using an existing model checker. As most of these were designed for validation of distributed applications such as communication protocols, so they will need enhancements with contract-specific constructs.

1.1 Research tasks and objectives

The goal of this research is to develop high level model checking toolkit for electronic contracts. We build on existing work on contract compliance checking; and a well–known Spin model checker. In this work, we developed a formal model called CB2B (for Contractual Business–to–Business interactions) based on the concepts of EROP (for events, rights, obligations and prohibitions) model for contract compliance checking developed earlier at Newcastle university (discussed in Chapter 3). The EROP model defines an architecture of Contract Compliance Checker (CCC) that observes the interactions between the business partners and forms an interpretation of their outcome.

We build on the facilities of Spin model checker in the verification and validation of e-contracts. The contract compliance checker CCC of EROP is modelled as a reactive system, converted into Promela and validated with Spin to observe properties of interest which are regarded as safety and liveness properties. The verification facilities such as simple assertions and verification of temporal logic formulae, offered by Spin, are exploited at large to reason about certain properties of models of EROP language contracts.

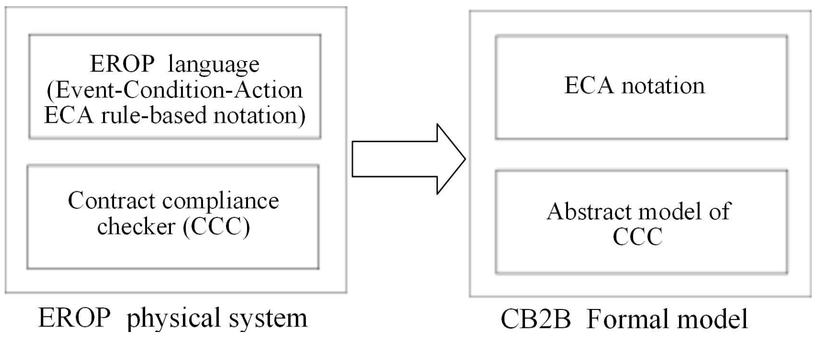
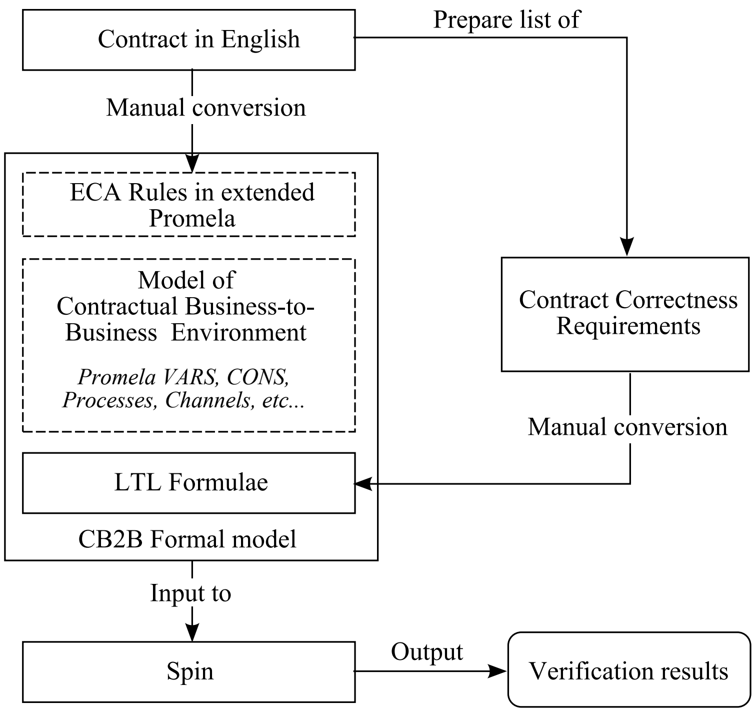

EROP language (Event-Condition-Action ECA rule-based notation)
Contract compliance checker (CCC)
ECA notation
Abstract model of CCC
EROP physical system
CB2B Formal model

Figure 1.3: A formal model based on EROP system

The toolkit enables one to: (i) specify the contract as ECA rules that can be automatically interpreted by a general model checking tool; (ii) automatically validate contract independent properties which are those properties that must hold for all contracts; (iii) check for contract dependent properties which are specific to individual contracts.



Contract in English
Prepare list of
Manual conversion
ECA Rules in extended Promela
Model of Contractual Business-to-Business Environment
Promela VARS, CONS, Processes, Channels, etc...
Contract Correctness Requirements
Manual conversion
LTL Formulae
CB2B Formal model
Input to
Spin
Output
Verification results

Figure 1.4: Contract model checking framework.

The design process supported by a verification tool is illustrated in Figure 1.4. The general purpose model checker Spin and its input language Promela are used with some extensions, for the verification of contracts. The following assumptions are made. Firstly, the contract has been negotiated by the contracting parties and drawn up in English or another natural language. Secondly, the designer manually converts the clauses of the English contract into ECA rules written in the extended Promela language, and these rules are executed by the contractual business–to–business (CB2B) interaction model as discussed in detail in Chapter 4. In parallel, the designer manually prepares a list of contract correctness requirements deduced from the contract clauses and expresses them in Linear Temporal Logic (LTL) formulae. Thirdly, the designer inputs the CB2B model together with the rules and the

LTL formulae into the Spin verifier and runs it to output verification results.

The CB2B model of the tool hides much of the intricate details of the construction of interacting processes over communication channels. It enables a designer to re-use essential components (shown in dashed boxes in Figure 1.4) to encode a contract for model checking directly as ECA rules in terms of the contract entities of business operations, and role players with their rights, obligations and prohibitions. In other words, the CB2B model offers an executable environment for the e–contract rules, and thus the executable behaviour of the contract can be exercised and tested before its deployment. Another key feature is that the verified contract model can be used for the generation of test cases to validate the actual implementation of the contract for specific properties. The tool is evaluated for the verification of several contract examples used by researchers working on contract verification.

1.2 Thesis overview

Chapter 2 presents background information that is relevant to contract representation and analysis tools. Different contract states are discussed that have to be considered in contract representation notations, and different contract analysis tools which employ model checking techniques for the validation and verification of contracts are examined.

Chapter 3 discusses basic concepts of contract compliance checker using the EROP model. These concepts underpin the e-contract model checking framework with CB2B model. The internal structure of the CCC is described and the process of contract compliance checking is demonstrated with an illustrative example. It is also discussed how the contract compliance checker CCC is modelled as a reactive system, along with how contract correctness requirements are specified as safety and liveness properties and verified with the Spin model checker.

Chapter 4 presents the contractual business–to–business formal model (CB2B) that essentially models a contract compliance checker system inspired by the CCC system. This also explained how the input language Promela of the Spin model checker is extended to specify different elements of the contract such as role player, business operation, rights, obligations and prohibitions. Finally, the representation of a contract model and its verification process are demonstrated in a number of simple examples.

Chapter 5 presents a number of case studies. The developed tool is used for the representation and verification of several contracts in which the contract rules are derived from natural language contracts, these then modeled as ECA rules written in extended Promela and verified with the Spin model checker. Errors are injected into the models to see how they would be detected using the verification procedure. Finally, Chapter 6 discusses future research directions and gives the conclusions of the present work.

Chapter 2

Background and related work

This Chapter presents general background information about contracts, and concentrates on business contracts that are used to regulate electronic business-to-business interactions. The relevant literature of contract representation approaches, contract conflicts and contract analysis tools is reviewed. The range of notations which aim to represent contract clauses are examined from different perspectives, and then different approaches to the representation of contracts are exemplified, including classic logic based notations [10], deontic logic based contract languages [45], visual notations [14] and other representations based on finite-state-machines (FSM) [49, 51].

The next section briefly reviews electronic contracts in the world of computing and describes their functional requirements. In this work we focus on the terms and conditions of business–to–business (B2B) legal contracts concerned with purchase orders fulfilment, supply chain management etc., rather than service level agreements (SLAs) that specify quality of service, such as bandwidth and response. Subsequent sections then review the notations used for representing contracts and their temporal aspects as well as contrary-to-duties (CTDs) and contrary-to-prohibitions (CTPs) [45].

2.1 Functional aspects of a contract

The functional aspects of a contract are determined by constraints imposed concerning rights, obligations and prohibitions and temporal aspects specifying deadlines associated with them. For instance, in terms of its functional requirements, some entities of the storage service contract shown earlier (Page 2) are role players (service provider, client), business operations (use, exceed and suspend), rights(use up to 100 GB), obligations (suspend when maximum quota 120 GB is exceeded).

Figure 2.1 depicts a general view of the main entities comprising a business contract. Informal descriptions of the functional requirements of a business contract are given below.

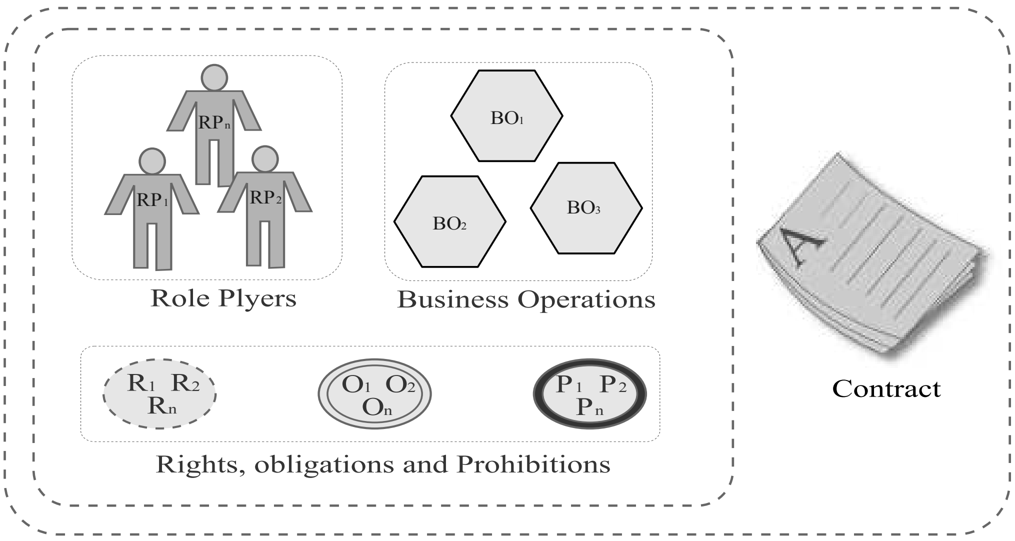


Figure 2.1: Contract entities

- Role player - an agent employed by one of the interacting parties, that takes on and plays a role defined in the contract. For example, client and service provider are instances of role players.

- 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 (e.g. [13]). Business operations constitute the vocabulary of a business contract, for example use, bring Quota back, pay and suspend are examples of business operations.

- Right or permission - is a business operation that a role player is allowed to execute. For example, the client has the right to use 10 gigabyte of the storage service.
- Obligation - is a business operation that a role player must execute, or face the penalty of being sanctioned. An example of an obligation is to pay for an invoice within 72 hours after receiving it.
- Prohibition - a business operation that a role player must not execute, or face the penalty of being sanctioned. An example of a prohibition, it is not allowed to exceed the maximum quota of storage service.

2.2 Machine-readable contracts

In order to implement machine-readable business contracts or e-contracts, it is necessary that the unstructured contract document is transformed into rigorous contract representation. This is an important phase, since it can help e-contract developers to analyse the complex and hard-to-detect interdependencies between the contract clauses. Moreover, the contract rules can easily be transformed into programming languages and processed by e-contract systems such as compliance monitoring services. This section first introduces the general issues concerning the representation of contracts in a machine-readable notation, and reviews the work that has been done by many researchers to develop different notations aiming for contract representation.

The following client/provider contract example will be modeled using different contract notations that will be reviewed in this Chapter, it is inspired by previous research [10] that discusses different contract representations down to the system–state level of granularity.

> A testing service provider (TSP) will do the testing job and deliver the testing-report to the client within 2 days from receiving the testing-job order. The client, in turn, will make payment within 5 days from the date of receiving the report. If the TSP does not deliver the report on time, then a fixed amount is to be deducted from the price for each day for delay for up to 3 days. If the client does not produce payment on time, then a fixed amount is to be added to the price due for each day of delay for up to 3 days.

In a machine–operated contractual business world, the constraints of a contract similar to the above are written in an executable format. So, it is required for such systems that different contract states have to be determined during the contract execution. For example, an acceptable state of contractual business exchange is a state in which contract parties have fulfilled their obligations, whereas a tolerable state is a state in which contract party's violations can be endured with sanctions, and finally an unacceptable state is a violation of the contract constraints [10].

As a result, different properties of the contract in force can be analysed, and e–contract systems can keep track of the evolution of a contract execution from its initial state to the current state of business exchange. From a business perspective, this is important for auditing purposes, likewise; from a technical perspective it is important in order to invoke the appropriate contract clause or its exception in the appropriate state, so that business evolves as expected.

2.2.1 Formal contract language (FCL)

The formal contract language (FCL) [10] adapts notations found in modal languages such as modal action logic which syntactically define the state space of business exchange. The proposed language is a high level representation of contract states and transitions; it is capable of representing the temporal elements of states and transitions in a contract. Each state in FCL corresponds to descriptions of the contractual business exchange. The propositions considered in the FCL language are factual and normative propositions. The factual propositions concern the properties according to which the business exchange is conducted in the real world, such as *testing-orders have been submitted* or the *customer's debts have been cleared*; whereas normative propositions state the permissions, obligations and prohibitions that describe the deontic status of business participants, for example *the customer is forbidden to cancel orders after 24 hours*, or the *TSP has the right to cancel orders at any time.* The language uses first order logic for the factual propositions and employs deontic modalities for normative propositions.

A simple notation have been used in FCL to model business e-contracts. The normative propositions can take the form such as $\Delta_A X$. Here Δ_A is a deontic operator (O for obligation, P for permission) imposed on a contract party A, and X itself is a factual or normative proposition, read as: the contract party A has a legal relation Δ to bring about X. For example, if a clause states that: the TSP is obliged to Submit the *Testing Report* to the *Customer*, the FCL would specify this obligation as: O_{TSP} *Testing Report* - representing the obligation O_{TSP} that the TSP must submit the *Testing Report* or he will be considered to be violating the agreement.

Transitions in FCL correspond to actions by contract parties which alter the current state of the contractual business exchange. Labels of the form $A : X$ denote that a contract party A has performed the appropriate action.Similarly, labels such as *not* $A : X$ denote that it is not the case that the contract party A has performed the appropriate actions. In each state the transitions advance by evaluating both the active regulatory elements of the contract, such as permissions, obligations and prohibitions and the actions performed by the contract party responsible.

2.2.1.1 FCL expressions and axioms

FCL language expressions of the form $[\tau]P$ denote that P necessarily holds in all states that are reachable following transition τ, which is the label for a transition. In a similar way, expressions such as $\neg[\tau]\neg P$ denote that P possibly holds in states that are reachable following transition τ. An expression such as $\langle\tau\rangle\mathsf{T}$ hold at a state S, when there is a transition labelled τ out of S, where T is the constant symbol for true; $[\ ]P$ denotes that P is true in all states. The FCL can also specify temporal elements of the contract. For example, $(A : X)^t$ denotes that contract party A needs to conduct an action about X at time t, or $(\Delta_A X)^t$ denotes that agent A has a pending Δ about X at t, where t may specify absolute or relative temporal relations such as (before(t_1), after (t_2) or between(t_1 , t_2)).

Axioms can also be added to contract representations written in FCL. For example, there could be an axiom which states that in any state of the business exchange where a contract party bears an obligation it is always possible for him to fulfill the pending obligation. Similarly, at any state another axiom could state that the obligation can possibly be violated. Axioms of this type could be respectively specified as follows, where A ranges over contract parties and X ranges over states of the business exchange:

$[\ ](O_A X \rightarrow \langle A \ : \ X\rangle \ \mathsf{T})$

$[\ ](O_A X \rightarrow \langle \mathit{not}\ A \ : \ X\rangle \ \mathsf{T})$

2.2.1.2 FCL contract example:

For a contract example specified in this FCL language, recall the TSP/client contract introduced in the previous section. The initial state S_0 in a business exchange regulated by this contract can be characterised by:

$O_{TSP} TestingReport$

$\langle TSP : TestingReport \rangle \top$

$\langle \text{not } TSP : TestingReport \rangle \top$

$O_{TSP} TestingReport$ denotes an obligation on the TSP to deliver the *Testing Report* to the client. The TSP can discharge this obligation $\langle TSP : TestingReport \rangle \top$ and deliver the $TestingReport$ to the $Customer$ or he can choose not to do so $\langle not\ TSP : TestingReport \rangle \top$. The business exchange in state S_1 which follows the transition labeled ⟨TSP : TestingReport⟩ $\top$ would be characterised by:

$TestingReport$

$\neg O_{TSP} TestingReport$

$O_{Customer} Pay$

$\langle Customer : Pay \rangle \top$

$\langle \text{not } Customer : Pay \rangle \top$

In this representation of the contract, Pay stands for payment is made by the customer, and the normative proposition $\neg O_{TSP} TestingReport$ asserts that the TSP's obligation to submit the testing-report is discharged. As shown in the first obligation on the TSP at S_0 the expressions $\langle Customer : Pay \rangle \top$ and $\langle \text{not } Customer : Pay \rangle \top$ provide two options for the $Customer$ to Pay or *not to Pay* respectively. A partial representation of the contract is characterised by:

$$initial \left\{ \begin{array}{l} O_{TSP} TstRpt \\ \neg TstRpt \\ \neg Pay \end{array} \right\}$$

where TSP refers to the Testing Service Provider, and *TstRpt* implies the submission of *Testing Report*.

(1)$[\ ](O_{TSP}TstRpt \rightarrow [TSP\ :\ TstRpt](TstRpt \wedge O_{Customer}Pay \wedge \neg O_{TSP}TstRpt))$

(2)$[\ ](O_{TSP}TstRpt \rightarrow [not\ TSP\ :\ TstRpt](\neg TstRpt \wedge \neg O_{Customer}Pay \wedge$

$\neg O_{TSP}TstRpt \wedge O_{TSP}TstRpt' \wedge tolerable))$

(3)$[\ ](O_{Customer}Pay \rightarrow [Customer\ :\ Pay](Pay \wedge \neg O_{Customer}Pay \wedge final))$

(4)$[\ ](O_{Customer}Pay \rightarrow [not\ Customer\ :\ Pay](\neg Pay \wedge \neg O_{Customer}Pay \wedge$

$O_{Customer}Pay' \wedge tolerable))$

(5)$[\ ](O_{TSP}TstRpt' \rightarrow [TSP\ :\ TstRpt'](TstRpt' \wedge \neg O_{Customer}TstRpt' \wedge$

$O_{TSP}Pay''))$

(6)$[\ ](O_{TSP}TstRpt' \rightarrow [not\ TSP\ :\ TstRpt'](\neg TstRpt' \wedge unacceptable))$

continue.........

The initial state of this contract is interpreted as follows. The contract starts with an obligation on TSP to submit the *TstRep* ($O_{TSP}TstRep$), the *TstRep* has not been delivered yet ($\neg TstRep$), and still no payments have been made ($\neg Pay$). The predicates *initial*, *final*, *tolerable* and *unacceptable* in some of the expressions determine the specific states of the contractual business exchange. In order to illustrate the tolerable state for example, we consider clause (2) and (6) in the FCL contract above. In clause (2), the contract tolerates failure to the obligation to submit testing report $O_{TSP}TstRpt$ by another obligation $O_{TSP}TstRpt'$. In the second obligation $O_{TSP}TstRpt'$ the contract party TSP is given a chance to fulfil his obligations (maybe with sanctions), whereas in clause (6) the second violation to submit the report will be considered as unacceptable state and might cause to terminate business exchange between the contract parties. The contract language FCL introduced here highlights important aspects of the executable notation of e-contracts, it has shown important different states that need be considered by all e–contract representation notations.

2.2.2 The contract language (CL)

Contract language (CL) [19, 45] is another high level language for the representation of business e-contracts. It is a combination of deontic, dynamic and temporal logics for the representation of obligations, permissions and prohibitions and their temporal aspects. The language can also specify exceptional behaviours arising from the violation of the contract's obligations or prohibitions.

2.2.2.1 CL syntax

The language of CL defines a contract specific syntax. The developers of CL resemble the syntax of deontic logic to model the contract clauses. Basically, the contract clauses C_O, C_P and C_F denote respectively obligation, permission, and prohibition clauses. The language also gives mean to specify exceptional behaviours arising from the violation of obligations and of prohibitions. These respectively known as Contrary-to-duties (CTD) and contrary to prohibitions (CTP). CL contract are written using the following syntax [19]:

$C := C_O \mid C_P \mid C_F \mid \mathrm{C} \wedge \mathrm{C} \mid [\beta]\mathrm{C} \mid \top \mid \bot$

$C_O := O_C(\alpha) \mid C_O \oplus C_O$

$C_P := P(\alpha) \mid C_P \oplus C_P$

$C_F := F_C(\alpha)$

$\alpha := 0 \mid 1 \mid \overline{a} \mid a \mid \alpha \;\&\; \alpha \mid \alpha ; \alpha \mid \alpha + \alpha$

$\beta := \epsilon \mid 0 \mid 1 \mid \overline{a} \mid a \mid \beta \;\&\; \beta \mid \beta \;;\; \beta \mid \beta + \beta \mid \beta^*$

A contract clause written in CL language is specified as an obligation C_O, permission C_P or prohibition C_F. The conjunction of two clauses is $C \wedge C$, which is intuitively understood as both clauses have to be satisfied. A clause can be preceded by the dynamic logic square brackets $[\beta]C$ where β is the condition which precedes the clause C. This is interpreted as, if action β is performed then the contract C must be executed; if β is not performed, the contract is trivially satisfied. The symbols $\top$

and $\perp$ refer to a trivially satisfied contract and impossible contract respectively; the former is a contract that is satisfied by any sequence of actions whereas the later is the contract that cannot be satisfied with any sequence of actions. Note that, instead of $F_{\perp}(\alpha)$ or $O_{\perp}(\alpha)$, a trivially satisfied contract clause can be simply written as $F(\alpha)$ or $O(\alpha)$ to denote obligations without violations and prohibitions without violations respectively. An obligation $O_C(\alpha)$ must be satisfied by an implicitly responsible contract party, or in case of violation a reparation C (exception) has to be executed. An obligation can also be specified as an exclusive disjunction of two other obligation clauses ($C_O \oplus C_O$) to satisfy only one of them. The forbidden action $F_C(\alpha)$ means that the specified action is forbidden and in case of violation where the action is performed then a reparation C has to be executed. With CL different operators can be used to construct action expressions from basic ones:

- (&) stands for concurrent actions.
- (;) stands for actions to occur in sequence.
- (+) stands for a choice between actions.
- * is the Kleene star.
- The complement of an action ($\bar{\cdot}$) such as a is $\overline{a}$.

Consider this clause from an Internet service provision contract: when the bandwidth limit is *(e)xtended*, the client has two obligations to choose from: either to *(p)ay* or *(d)elay* and *(n)otify*, in case of violation the price must be paid twice. This would be represented in CL as: $[e]O_{O(p;p)}(p + d\&n)$

The basic actions of this contract are $A = \{e, p, n, d\}$. Executing this expression is conditional upon the action e. If it occurs, there is an obligatory choice between p or $d\&n$ is activated. If violated, the reparation $O(p; p)$ is another obligation for paying at double the price, where the double occurrence of p implies payment twice. The CL language expression shown above does not offer reparation for the double payment obligation ($O(p;p)$). This means that if the latter obligation is violated then the whole contract is violated.

2.2.2.2 CL contract example

In this example, the TSP/client contract from the previous sections can be written with CL syntax [19] as follows:

$1 - [ContractStart]\mathbf{O}_{\mathbf{O}_{[2Days](TSPSubmitTstRep)}}(TSPSubmitTstRep)$

$2 - [TSPSubmitTstRep]\mathbf{O}_{\mathbf{O}[5Days](CPayForTstRep)}(CPayForTstRep)$

$3 - [\overline{TSPSubmitTstRep}]\mathbf{O}[3Days](TSPSubmitTstRep')$

$4 - [\overline{TSPSubmitTstRep'}]\mathbf{O}(TSPPayFine)$

$5 - [\overline{CPayForTstRep}]\mathbf{O}[5Days](CPayForTstRep')$

$6 - [\overline{CPayForTstRep'}]\mathbf{O}(CPayFine)$

The CL contract lines from 1-6 specify the obligations on contract parties (*TSP* and *client*) and reparations for their violations. A commencement date is assumed represented by the action *ContractStart*. The first line provides a reparation clause to the action *TSPSubmitTstRep*, that the TSP submitted testing report to the client, which is to tolerate the TSP's failure to submit the report so that he can re-submit within 2 days ($\mathbf{O}_{([2Days](TSPSubmitTstRep))}$). The second line is conditional upon what appears in the square brackets *[TSPSubmitTstRep]*. If the condition is satisfied then the client is supplied with an obligation to pay for the testing service (*CPayForTstRep*); and in case of failure, it is given a reparatory choice ***O****[5Days](CPayForTstRep)* which is conditional on an incremental fine within five days. The fourth and sixth lines specify obligations to pay fines when the clauses in lines 3 or 5 are violated.

2.2.3 Contract-oriented diagram (C-O diagram)

Contract-oriented diagram (C-O diagram) [34] introduces an approach for the specification of e-contracts in a more user friendly way. It is based on visual models to enhance the perception of knowledge, and the intuitive understanding, reading and maintenance of electronic contract specifications. This visual model defines a hierarchical tree diagram used to specify contract clauses.

2.2.3.1 C-O diagram visual elements

Figure 2.2 depicts the basic element of a C-O diagram called a box. This box corresponds to a contract clause with four fields in order to specify the normative aspects: legal relations of the contract P, reparation R, conditions g and time restrictions tr.

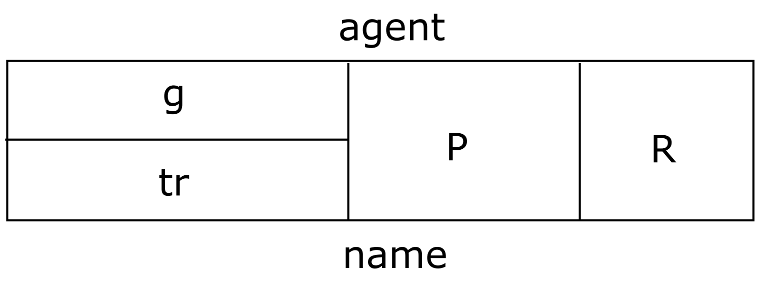


Figure 2.2: C-O diagram box structure

Each box can be identified by a name and agent, or contract party. The name uniquely describes the intended clause and references the box from other clauses. The *agent* indicates who is the actor behind the action to be performed. The left-hand side of the box specifies the conditions and restrictions. The guard (g) specifies the conditions under which the contract clause must be taken into account. The time restriction (tr) specifies the time frame in which the contract clause must be satisfied. The propositional content (P), in the centre, is the main field of the box, and it is used to specify the normative aspects of obligations, permissions and prohibitions that in relation to actions, and/or the actions themselves. The other field in these boxes, on the right-hand side, is the reparation (R). This reparation, if specified by the contract clause, is another contract that must be satisfied in cases where the main norm is not satisfied, considering that the clause is eventually satisfied if this reparation is satisfied.

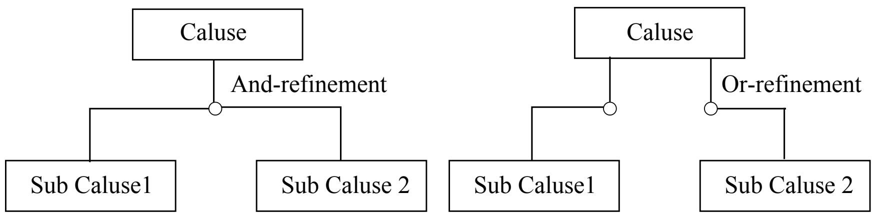


Figure 2.3: AND/OR refinements

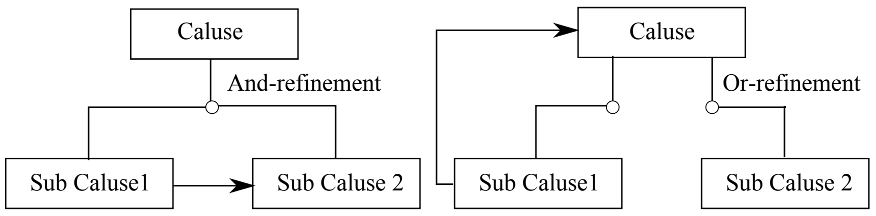


Figure 2.4: SEQ refinement and repetition in C-O diagrams

2.2.3.2 Refining C-O diagrams

The basic elements of a C-O diagram can be refined by using *AND,OR,SEQ* refinements (see Figures 2.3 and 2.4). Intuitively *AND-refinement* and *OR-refinement* imply respectively both of the contract subclauses or only one of them must be satisfied. The SEQ refinement is used to specify a temporal relationship of a sequence between the contract subclauses. Note that C-O diagrams implement a hierarchical structure in which a parent clause is only considered fulfilled if its subclauses are fulfilled. In this way, a hierarchical tree is built with the clauses defined by the contract, where the leaf clauses correspond to clauses that cannot be divided into further subclauses. Repetition in C-O diagrams is represented as an arrow going from a subclause to one of its ancestor clauses or to itself, as shown in Figure 2.4. This means the repetitive application of all the subclauses of the target clause after satisfying the source subclause. It is worth mentioning that, instead of the formal contract notations shown in the previous subsections, the C-O diagram visual model is introduced in order to provide an alternative approach for those untrained final users who want to use formal representations of contracts.

2.2.3.3 An example using C-O diagrams

Our running example can be specified using this visual notation to illustrate the compound action, propositional content (P), guard (g) and reparation (R).

Clause	Agent	Modality	Action	Reparation
C_1	TSP	Obligation (O_1)	Submit testing report within 2 days (a_1)	TSP pays penalty for each day delay (max 3 days) (r_1)
C_2	Client	Obligation (O_2)	Make payment within 5days after receiving the report (a_2)	Client pays penalty for each day delay (max 3 days) (r_2)

Table 2.1: Norms of the TSP/client contract example

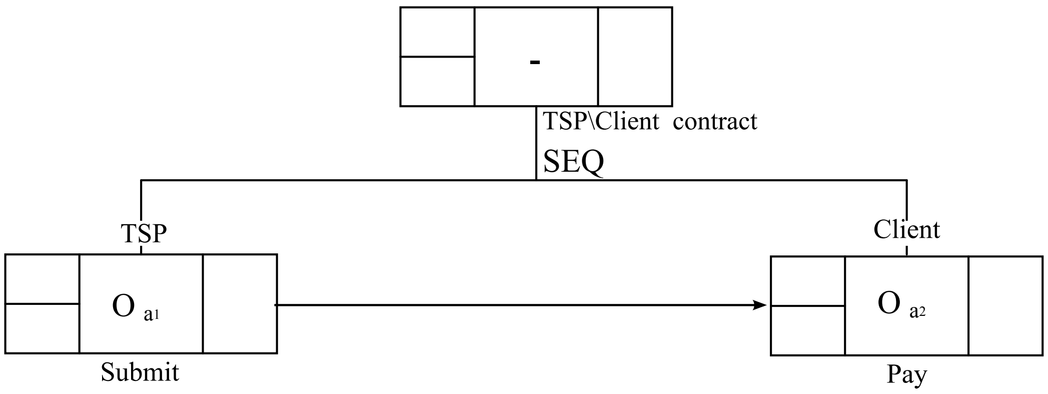
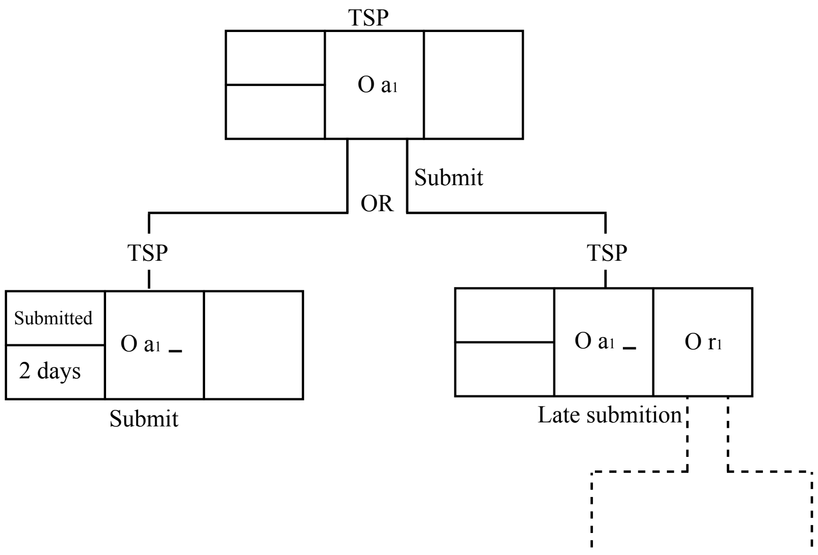
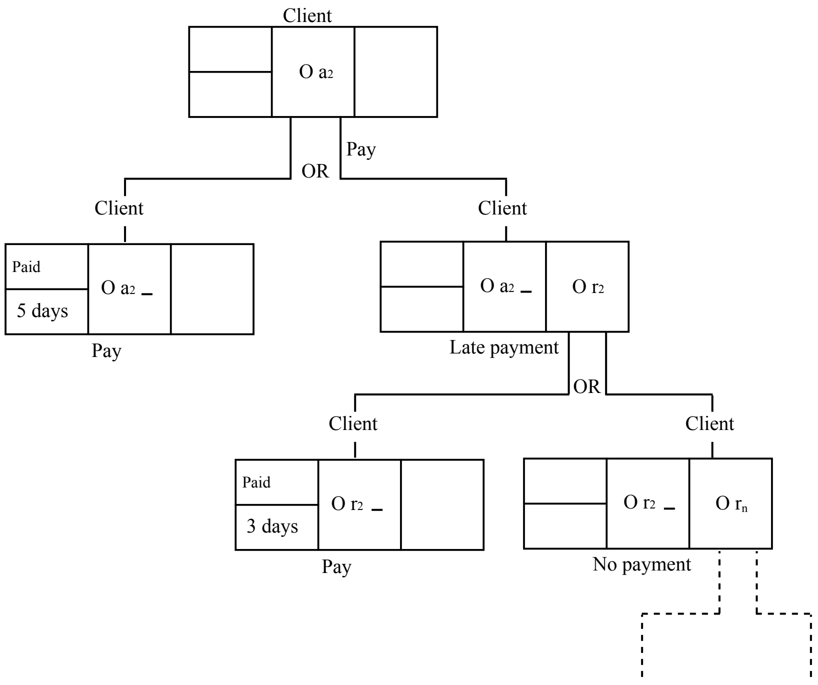


Figure 2.5: Top level C-O diagram of TSP/client contract



(a)



(b)

Figure 2.6: (a) Decomposition of clause Submit (b) Decomposition of clause Pay and the reparation clause Late payment.

Figure 2.5 shows the top level of the C-O digram; it shows a sequence (SEQ) relationship between the contract obligations *Submit* and *Pay* (see Table.2.1)). The decomposition of clause *Pay* into subclauses can be seen in Figure 2.6(b). Using the 'OR' operator in this decomposition implies that one of the two obligations can be chosen; a *client* can pay within 5 days so that the obligation (O_{a2}) is discharged, or, as a reparation, the *Late payment* clause would handle this deviation and add the agreed upon penalty amount. The hierarchical representation of the contract using C-O digram has increased the perception of the the CL language discussed in the previous section.

Through the visual notation in the C-O Digram, a contract developer can recognise some of the conflicts in contract and identify them visually. For example, two boxes in a C-O diagram intend to represent two obligations may reveal contradictory actions imposed on the contract parties. For example a contract party is permitted and forbidden to do the same action, obliged and forbidden to do another action, etc. However, contract conflict detection with C-O diagrams is still a manual process relying on human capabilities.

2.2.4 X-Contract language

In the X-Contract language [49], contracts are modelled as finite state machines (FSMs) [24], with one FSM for each contracting party. With X-Contracts relevant parts of standard conventional contracts can be described by means of FSMs. Such a description becomes quite suitable for model checking [49]. The following example shows how business contract clauses are represented as X-Contracts:

1. Offer
    (a) The supplier may use his discretion to send offers to the purchaser.
    (b) The purchaser is entitled to accept or reject the offer, but he shall notify his decision to the supplier.
2. Commencement and completion
    (a) The contract shall start immediately upon signature.
    (b) The purchaser and the supplier shall terminate the contract immediately after reaching a deal for buying an item.

From the contract in English text, the sets of rights and obligations for the purchaser and the supplier are extracted and then expressed in terms of operations for FSMs. What follows is the set of rights and obligations extracted from the contract document:

- Purchaser's rights
    - SendAccepted (right to accept offers)
    - SendRejected (right to reject offers)
- Purchaser's obligations
    - StartEcontract (obligation to start the contract)
    - SendAccepted or SendRejected (obligation to reply to offers)
    - EndEcontract (obligation to terminate the contract)
- Supplier's rights
    - SendOffer (right to send offers)
- Supplier's obligations
    - StartEcontract (obligation to start the contract),
    - EndEcontract (obligation to terminate the contract)

Figure 2.7 shows how the sets rights and obligations are mapped into FSMs. For the validation of X-Contracts with Spin model checker, the X-Contract shown in this example is converted manually into Promela and then verified automatically. With regard to our research, two important observations follow from this example:

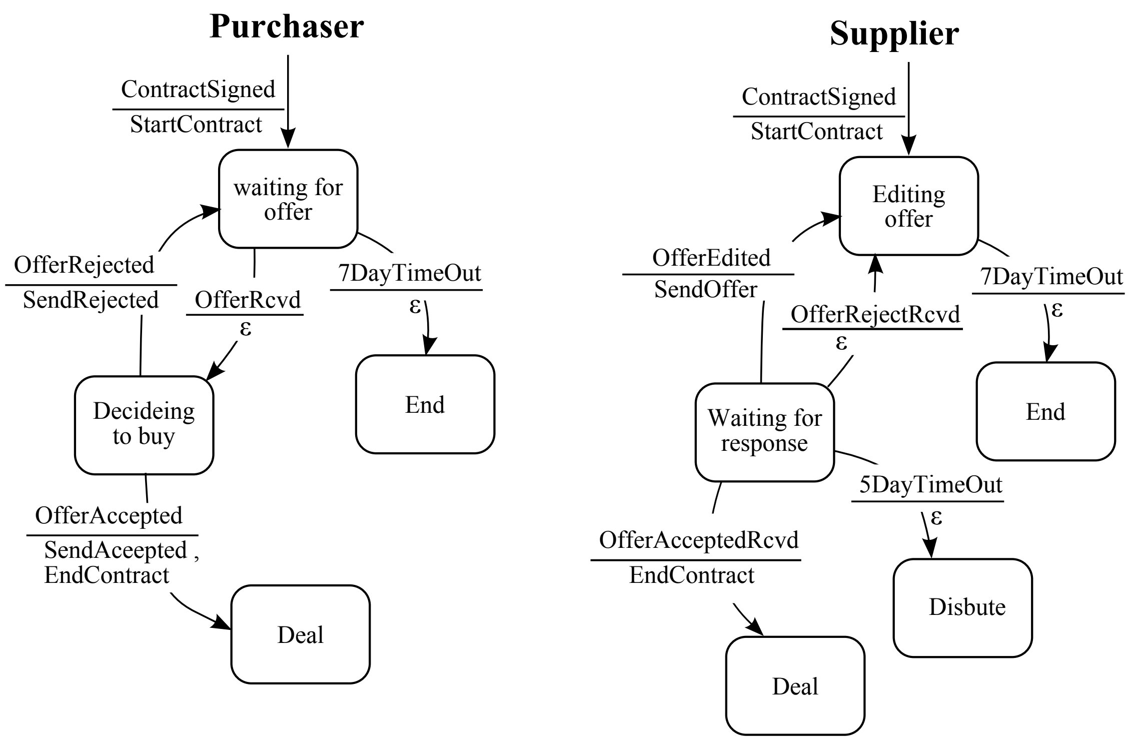


Figure 2.7: Buyer and Seller Contract FSM

(i) it shows that the challenging task of converting contracts written in the English language into mathematically rigorous notation such as FSMs is achievable; (ii) this suggests that the analysis of contract correctness requirements can be performed mechanically, for example, via model checking techniques. However, a limitation of FSMs in modeling contracts is that although FSMs can capture a 'change' of the contract, but cannot elegantly describe it in terms of its internal data, or the changes in this data that occur after the execution of each business operation in the state transition diagram. In other words, the expressive power of FSMs mainly lies in modelling the control part of the contract.

2.2.5 EROP contract language

The language of EROP [50] standing for events, rights, obligations and prohibitions. Unlike the contract languages discussed so far, a contract model specified in EROP can easily be transformed into an executable e-contract notation and monitored or enacted by the appropriate e-contract systems. The EROP language can be used for describing business contracts with event-condition-action rules (ECA) that explicitly manipulate the partners rights, obligations and prohibitions. ECA rules are then used to monitor contract compliance [50]. The concepts thast underpin this language are described in Chapter 3 as we will be using them in our work as well. The main constructs of the EROP language are the declaration sections and the contract rule base. In the declaration section EROP defines data types such *BusinessOperation*, *RolePlayer* and allow contract rights, obligations and prohibitions to be initialised. The ECA rules of EROP are expressed as follows:

$e \equiv$*(Business event)*, $c_1 \rightarrow a_1; ...; c_m \rightarrow a_m$; where $c_1, ..., c_m$ are mutually exclusive conditions for compliance checking and $a_1, ..., a_m$ are actions. When the business event 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. Also, conditions that always evaluate as true can be omitted; this results in simpler rules of the form $e \equiv (Business\ event) \rightarrow a$.

2.2.5.1 Buyer/Seller contract

A hypothetical EROP contract [38] between buyer and seller is shown below, in which clause C1 grants the buyer a right; and clause C2 imposes an obligation on the seller. Clause C7 is an example of clauses that take into account problems caused by infrastructure level problems; such problems can be referred to as business problems, indicating problems caused by semantic errors in business messages which prevent their processing, and technical problems such as problems caused by faults in networks and hardware/software components [38, 50].

- C1 - The buyer has the right to submit a Purchase Order (right), between 9 am and 5 pm, Mon to Fri.

- 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 abnormal contract termination with offline 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
  - C7.1 - The seller rejects the purchase order, or
  - C7.2 - The seller successfully delivers the goods.

2.2.5.2 Modelling Buyer/Seller contract in EROP

Here we show how the previous contract can be modelled in EROP rule base notation. This is a manual process in which the user of the language identifies the role players of the contract and the business operations subject to regulation by the contract rules. The following set of the contract rules are derived from clauses C3, C4, C6, C7 and C8 of the above contract. The declaration section contract also shows *rule players* and *business operations* as declarations of EROP abstract data types, a complete reference to EROP can be found in [50].

```
Contract Declaration:

RolePlyer buyer, seller;
BusinessOperation POSubmission, Invoice, Payment, POCancellation, Refund;
BusinessOperation GoodsDelivery, POAcceptance, PORejection;
```

The following contract rules are derived from clauses C3 and C4 of the above contract:

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

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

In the following, Rule 6 is derived from C6 and C7, while Rule 8 is derived from clause C8.

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

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

The contract rules R3, R4, R6 and R8 show samples of the EROP language representation contracts. The rules are distinguished by unique names and each rule is triggered by an event 'e' of a particular type. For example R3 identifies contract rule that handles a notification of purchase order acceptance (POAcceptance). The rule's header (When part) is encoded to monitor business events 'e' of type POAcceptance, which is an event expected to be originated by the seller and responded to it by the buyer. The rule also encoded to check, in the seller's ROP set, that the seller is obliged to respond to buyer or not. Thus, the rule header is guarded by the occurrence of the

POAcceptance business event as well as the other fields shown in the 'When' part of the rule. The consequences part of the rule (Then part) is encoded to discharge the obligation on the seller to respond to the buyer and adds a new obligation on her to send the invoice within 24 hours.

The implementation of the other rules R4, R6 and R8 is not different from R3 that discussed. Small additions to R6 and R8 is that in R6 the rule is encoded to handle business failures (BizFail) and in R8 the rule is encoded to cope with no more that 3 business failures and technical failures all together. Rule 8 shows as a consequence of exceeding the threshold of failures would cause contract termination. Note that the declaration part of EROP language shown above the rules is used to declare the business operations used in the contract, role players and any other variables needed to encode the contract rules

2.3 Contract conflicts

The previous sections have shown how contracts found in the business world may serve as a basis for defining machine-oriented contracts (e-contracts). Ideally the electronic representation of contracts should be shown to be contradiction-free. This is a necessary requirement prior to implementing an executable version of contract for compliance monitoring or enactment.

The types of conflicts arising in systems where the behaviour of participants is regulated by norms are classified in [23]. For example, conflicts in a system of norms can arise when a pair of norms have an *opposite subject* and compliance with one norm causes conflict with another. This conflict is identified by legal philosophers using the *impossibility-of-joint-compliance* test; other researchers resolve this conflict through an *obedience statement* for a norm, which is basically a proposition stating what the subject to a norm can do if he obeys the norm so that the conflict can be detected when two norms are logically inconsistent. Another conflicts are *clash* or *collide* [23], when an agent is in a situation where its actions or omissions result in the violation of an obligation, or result in the violation of a permission.

In fact, many of these conflicts can be encountered in business contract specifications. It can be argued that electronic contract rules that are consistent with standard deontic logic SDL [5], the logic of permissions, obligation and prohibitions, may not pose any *direct* inconsistencies for a contract party's actions, so that the contract party will never find himself in a situation where he will be obliged to perform and at the same time prohibited from performing an action or bringing it about. However, *indirectly* inconsistent situations cannot be avoided even with the SDL language. For instance, it may be the case that it is obligatory for a client to pay a provider upon receipt of ordered goods, and simultaneously it may be prohibited for the client to pay using credit cards for some reason, yet a credit card may be the only way for the client to perform payment.

A previous study has identified four different causes of conflicts in contracts [18]. The first two are being both obliged and forbidden to perform the same action (e.g., $O(a) \wedge F(a)$), and being both permitted and forbidden to perform the same action (e.g., $P(a) \wedge F(a)$). The other two kinds of conflicts correspond to obligations to perform contradictory actions (e.g., $O(a) \wedge O(b)$ with $a\#b$), and permissions and obligations concerning contradictory actions (e.g., $P(a) \wedge O(b)$ with $a\#b$), where $a\#b$ means that actions 'a' and 'b' are mutually exclusive.

Conflicts could be encountered in business contracts when more than one rule appears to be applicable [26]; consider the following example:

Rule A - if customer returns the purchased e-ticket for any reason, within 7 days, then the purchase amount, minus a 10% penalty fee, will be refunded.

Rule B - if customer returns the purchased e-ticket because the flight is canceled by the seller (travel agent), before the due date, then the full purchase amount will be refunded.

A conflict would occur if the ticket is canceled by the seller in the overlapping period between rule A and rule B within 7 days from the ticket issuing date, so both rules are applicable. This conflict between rules A and B can be solved through the use of priority rule as shown below:

Priority rule (A and B) - if both rule A and rule B are applicable, then rule B must be applied, so the full purchase amount will be refunded.

The priority rule applies in situations when the customer returns the e-ticket within 7 days because the flight is canceled. In this case rule A and B conflict by each other. This conflict can be easily resolved by the priority rule which gives higher priority to rule B than rule A.

2.4 Analysis of contract conflicts

Contract analysis is a process aimed at detecting potential errors that can be introduced by inconsistent contract clauses. As discussed above, contracts could contain inconsistencies in the form of conflicting clauses. For instance, two or more clauses could conflict or be inconsistent if in certain conditions one of the rules is triggered by an event but the actions defined in the body of the rules contradict some other rules. Different tools have been used for the verification and analysis of contracts, most of which are model checkers. For example the Spin model checker is used for the verification of safety and liveness properties of e-contracts [36]. The use of Spin has been demonstrated for validating an ECA rule based contract compliance checker that monitors the interactions between business partners. The same Spin model checker has also been used for the verification of Web service specifications modeled as synchronous interactions [21]. Other model checkers such as MCMAS [3], NuSMV [7], VeriSoft [22] and CLAN [19] have also been successfully used for the verification of contracts. For example, the specification of contract regulated service compositions written in WS-BPEL are translated into timed automata which are automatically verified by the MCMAS model-checker [4]. The NuSMV model checker is also used to verify the properties of contracts written in CL contract language [41]. The VeriSoft model checker has been suggested for the validation of business processes [53]. Petri nets [39] are also used to model the behaviour of the participants of a contractual protocol, where a process view of agreements between parties means that a contract is modelled as it evolves over time in terms of actions or more general events that effect changes in its state [9]. CLAN is another model checker that is implemented for performing the automatic analysis of conflicting clauses written in CL language. This is not a general model checker however, but was designed and implemented only for the verification of contracts written in CL language.

We have seen that different model checkers have been successfully used for the analysis of contract properties. Next we discuss the main principles of model checking using Spin model checker. Notice that, apart from the contract–dedicated model checker CLAN, the principles of model checking with Spin are quite similar to the other model–checker introduced in this section.

2.4.1 Principles of model–checking (using Spin)

Spin [27] is a freely available, mature and well documented model checker designed to validate the correctness properties of asynchronous process systems. It validates abstract models written in the Promela language against safety and liveness correctness claims that can be expressed as basic assertions and linear temporal logic (LTL) formulae [28]. Recall that safety is concerned with a program not reaching a bad state and that liveness is concerned with a program eventually reaching a good state [33]. In response to the detection of a violation of a given property, Spin produces counterexamples that show how the property was violated. A counterexample is actually an execution trace from the initial state of the model through to the state where the violation of the property takes place. Figure 2.8 depicts the Spin structure: in a Promela model describing the behaviour of a system, and an LTL property expressing a correctness property that the Promela model is expected to observe are presented to Spin for analysis. Spin produces a counterexample if the LTL is violated, otherwise, it outputs nothing.

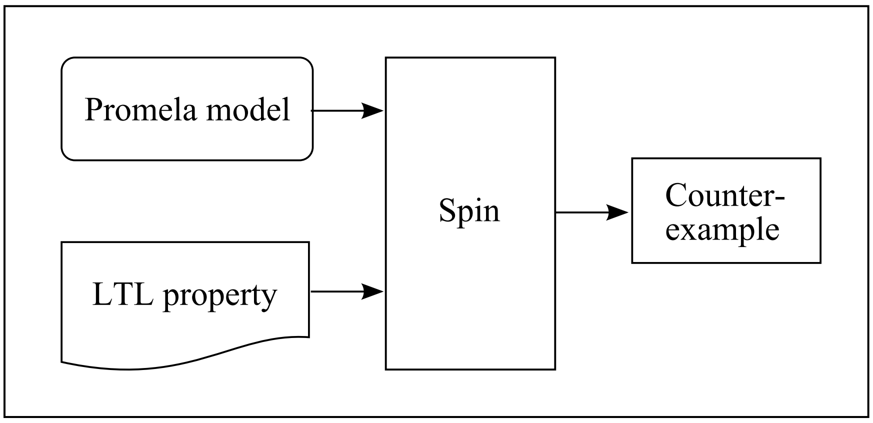


Figure 2.8: Spin structure

Promela is a process modeling language, which can generally be seen as a fairly standard imperative programming language although it does have some novel features. Its intended use is for describing concurrent systems. It is a non-deterministic, multi-process language which incorporates guarded commands and communication channels. Processes are global objects that represent the concurrent entities of the distributed system. Message channels and variables can be declared either globally or locally within a process. Processes specify behavior, channels and global variables define the environment in which the processes run. Given a program in Promela, Spin verifies the model for correctness by performing random or iterative simulations of the modeled system's execution, or it can perform an exhaustive verification of the generated system's state space. During simulations and verifications, Spin looks for the absence of deadlocks, unspecified receptions, and unexecutable (dead) code. The verifier can also be used to prove the correctness of system invariants and it can find non-progress execution cycles. Finally, it supports the verification of linear time temporal constraints by formulating them in temporal logic formulae. Each model can be verified with Spin under different types of assumptions about the environment. Once the correctness of a model has been established, that fact can be used in the construction and verification of subsequent models.

2.4.2 Analyzing x-contracts with Spin

Analyzing x-contracts with the Spin model checker relies on model-checking algorithms to determine whether or not a contract model satisfies a list of correctness requirements. The validation process following a widely used model-based validation approach for validating such requirements, including correct commencement, precedence, and the absence of livelocks as defined in a previous study [49]. With the Spin model checker, the correctness requirements of an x-contract are specified as safety and liveness properties through linear temporal logic. Informally, a safety property is a statement which claims that something bad will not happen, and a liveness property is a statement which claims that something good will eventually happen. Furthermore, it can be argued that most correctness requirements in traditional business contracts can readily be expressed either as safety or liveness properties [49].

For the verification of the safety and liveness properties of a contract, an x-contract has to be derived and represented as a set of FSMs, one for each of the contracting parties that interact with each other. Conceptually this assumes that an FSM is located within each contracting party and that these FSMs communicate with each other through communication channels.

In order to use the Spin model checker for the verification of x-contracts, the first step is to convert the FSMs into Promela language. After that, Spin can by default verify general safety properties that must hold true for any x-contract, for example deadlocks. The validation of the remaining specific safety properties is done through simple assertions inserted within the Promela code. Through these steps, the validation of x-contracts with Spin follows the classical model checking technique of software systems.

2.4.3 Analyzing EROP contracts with Spin

Spin has been used for the validation of EROP contracts specified as a set of ECA rules. In a practical approach, an abstracted model of a contract compliance checker (CCC) [51] system has been modelled as a reactive system, converted into Promela and validated with Spin to observe the properties of interest which are regarded as safety and liveness properties. This checks the consistency of the ECA rules and the behaviour of the CCC in response to both valid, or contractually compliant or invalid events erroneously supplied by the parties. Other properties considered are contract specific properties, such as deadline extensions granted exactly as stated in the clauses and so forth.

In has been suggested that a large proportion of errors in ECA rules, such as rule redundancy, submed rules or conflicting rules can be detected by conventional model checkers [36]. With Spin, general types of potential errors that might be accidentally introduced into ECA contract rules can be detected as violations of safety and liveness properties, such as deadlocks, non-progress cycles, unexpected messages and incorrect final states. With default settings, Spin has been used to validate these requirements and for subtle errors linear temporal logic (LTL) expressions have been suggested to verify rule conflicts.

2.4.4 Analyzing CL contracts with CLAN

Section 2.3.3 has described the contract language CL and demonstrated through an example how it is used to specify the permissions, obligations and prohibitions of contract clauses. A CLAN tool is used for the automatic analysis of conflicting clauses in contracts written in the CL, and it enables the automatic analysis of the contract for normative conflicts as well as the automatic generation of a monitor for the contract [19].

CLAN has been implemented to verify the correctness of specific contracts written in CL language. In particular CLAN considers the contract conflicts omitted in the previous studies discussed in sections 2.4.1 and 2.4.2 using a different approach based on deontic logic. For conflict analysis, the tool performs exhaustive search on clauses written in CL language to detect conflicts that arise for four reasons [19]: (i) obligation and prohibition on the same action; (ii) permission and prohibition on the same action; (iii) obligation to perform mutually exclusive actions; and (iv) permission and obligation to perform mutually exclusive actions.

2.5 Discussion

Model-checking tools have been successfully used by different researchers to reason about contracts. The success model checking techniques for contracts arises from their ability to operate on two specifications: the operational specifications of system behaviour expressed as state-machines, Petri nets, for examples and the declarative specification of behavioural requirements expressed in temporal logic [9]. However, a disadvantage of these approaches is that contracts written in languages such as FCL, x-contract and EROP still need to be re-written in the input language of the model checker in order to be model checked. The only tool reviewed here that accepts direct contract language is CLAN, which has been demonstrated in modelling a number of contracts. However, the CL language itself is unable to express complex contract clauses; for example, it is not possible to express basic arithmetic statements or declare role players. Other special purpose tools for reasoning about policies for conflicts for instance are also suggested in [12, 32, 38]

Several types of notation for the specification of contractual clauses and their analysis tools have been reviewed in this chapter, others have been published [4, 6, 13, 31, 42, 47]. The main conclusion that can be drawn is that many of the existing rigorous contract notations lack expressive power and are not very intuitively clear for non-experts. The developers of these languages have not considered the close collaboration required between domain experts such as software engineers and business managers. An exception to this statement would be the EROP language, whose developers have chosen to encode contract clauses as ECA rules because of the widespread usage of such rules in the business world for representing business agreements. The language offers a more user friendly notation likely to be familiar to non-experts who might frequently want to discuss contract terms and conditions with software engineers tasked to implement or analyse contracts. But although these features of the EROP language are appealing, it lacks the soundness of other languages grounded in well-established logics.

Moreover, subtle inconsistencies can be introduced via the ECA representation of contract clauses. Examples of these inconsistencies are situations where there is no rule to deal with payment or a situation where the rule that deals with the payment is triggered twice and the client is double charged. Likewise, the rules might allow conflicting situations where the client is simultaneously obliged to and prohibited from submitting payment; or where the client is simultaneously permitted to and prohibited from submitting payment. It is argued here that these logical errors should be uncovered at the design stage and if possible corrected, in order to prevent confronting policy managers with flawed rules that are likely to hinder proper interactions between clients and providers.

2.6 Approach taken

Model checking is a widely accepted solution and is adopted in this study to develop a tool to verify the correctness properties of contracts. However, as stated earlier verification tools with means for directly and intuitively expressing key concepts that often appear in contracts, such as executions of business operations, the granting, cancellation, suspension, fullfilment, or violation of rights, obligations and prohibitions for role players, are still lacking. We base our work on EROP model [38] and the EROP language [50]. We develop an abstract version of the EROP language that is directly verifiable using a general model checking tool Spin. The main advantage of this approach are:

- Large class of business contracts can be represented as a set of ECA rules and then verified with the Spin model checker.
- The ECA rules of the abstract version of EROP can be translated relatively easily into executable concrete versions of EROP that run with a rule engine (Drools [11]).
- Instead of the natural language contract, the ECA rules may be used as intermediate representations of the contract in communications between software engineering team and business managers.

The next Chapter describes the EROP system [38] in more detail and defines the practical contract compliance checking process by observing that business operations executed by a role players must abide by rights, obligations and prohibitions, together with any additional constraints, as stipulated in contract clauses. The Chapter then explains how the EROP system is modelled as a reactive system, and how contract correctness requirements are specified as safety and liveness properties and verified with the Spin model checker. Following that, our toolkit based on EROP model and EROP language will then be introduced and discussed in Chapter 4.

Chapter 3

The EROP model

The EROP language [50] was introduced in the previous Chapter. Here the underlying model (contract compliance checker CCC) [38] that underpins the EROP language is discussed in more detail; we describe its main components and its executable behaviour for monitoring contracts. The CCC model supports the representation and monitoring of contracts, and is composed of an ontology and an architecture. It observes the interactions between business partners, forms an interpretation of their outcome, and checks their contractual compliance by matching executed operations with their sets of rights, obligations, and prohibitions, and by reacting accordingly to them.

It is also discussed here that the CCC can be modelled as a reactive system, and different properties of a contract then can be verified as safety and liveness requirements using model checking techniques. The main concepts of EROP, its contract compliance checker CCC and its input language (EROP) are underpinning our toolkit for e-contract model checking that will be introduced in Chapter 4.

3.1 The Contract Compliance Checker

The contract compliance checker (CCC) is provided with an executable specification of the contract in force and monitors business events related to the actions of contract participants with the contract clauses specified in the rule-based and event-driven EROP language. The language is used in a similar fashion to contracts in natural language, where contractual clauses are expressed as business rules which are conditional statements associating events and conditions to lists of actions altering the rights, obligations and prohibitions of the contract participants.

3.1.1 Monitoring business events

The function of the CCC is to maintain a record of the observed events and act as an arbiter to help provide answers to queries regarding fulfilling the contract obligations. Figure 3.1 below depicts the logical communication paths between business partners and the CCC. The interaction between partners takes place through a well defined set of primitive business operations such as *purchase order submission*, *invoice notification*, and so on; each operation typically involves the transfer of one or more business documents.

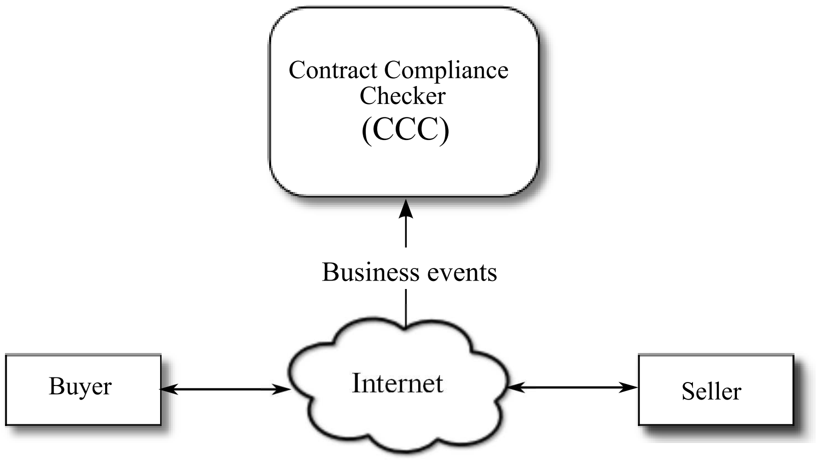


Figure 3.1: Contract regulated interactions

A business operation would normally be implemented by a business conversation. This is a well defined message interaction protocol with stringent message timing and validity constraints where, normally a business message is accepted for processing only if it is timely and satisfies specific syntactic and semantic validity constraints. RosettaNet Partner Interface Processes and ebXML industry standards serve as good examples of such conversations [13, 30]. Following the ebXML specification [13], once a conversation is started, or in other words a business operation is initiated, it is always completed to produce an execution outcome event from the set {*Success, BizFail, TecFail*} whose elements represent respectively a successful conclusion, a business failure or a technical failure. BizFail and TecFail events model the hopefully rare execution outcomes when, after a successful initiation, a party is unable to reach the normal end of a conversation due to exceptional situations. TecFail models protocol related failures detected at the middleware level, such as a late, syntactically incorrect, or missing message. BizFail models semantic errors in a message detected at the business level, such as invalid address for the delivery of goods extracted from the business document.

Failure outcomes play an important role in making electronic contracts tolerant against infrastructure level problems, since they provide a way of incorporating specific exceptional clauses to deal with them [37]. For example, an exceptional payment clause might read along the lines of : failure to meet a payment deadline due to business or technical reasons will grant 5 days extension to the buyer. Another example would be: If the total number of business and technical failures exceed an agreed bound, then online processing will be terminated.

It is assumed that the CCC is able to observe B2B interactions at the level of granularity of the outcome events of business operations. The events are delivered to the CCC exactly once in temporal order; these events are logged by the CCC. Each such event contains information that includes the termination status (Success, BizFail or TecFail), name of the operation, the timestamp and other attributes so that the operation can be further classified' for example, in terms of role player performing the operation.

Business partners exercise their contractual rights, obligations and prohibitions by executing their corresponding business operations. As operations are executed, rights, obligations and prohibitions are granted to and revoked from business partners. In general at a given moment, each business partner can have several rights, several obligations and several prohibitions, in force. This idea is at the heart of the functionality of the CCC that in observing the outcome events of business operations. With each participant also termed a role player, a *ROP set* is associated with the set of rights, obligations and prohibitions currently in force. The set $B = \{bo_1, \ldots, bo_n\}$ of business operations is used to specify all the primitive business operations stipulated in a contract.

3.1.2 Observing compliance with contract rules

The execution of business operation bo_i is said to be contract compliant if it satisfies the following three requirements, and is said to be non–contract compliant if it does not:

- C1) $bo_i \in B$;
- C2) it matches the ROP set of its role player, meaning that the role player has a right/obligation/prohibition to perform that operation;
- C3) it satisfies the constraints stipulated in the contractual clauses.

A business operation that meets the first requirement is termed valid, or else it is termed an unknown business operation. A valid business operation that satisfies the second requirement is termed matched, otherwise it is termed a mismatched business operation; a matched business operation that does not meet the third requirement is termed an out-of-context business operation. Consider the example contract clause: the buyer is obliged to submit payment within 5 days of sending the purchase order. A payment operation performed by the buyer within 5 days, which is a constraint, will be contract compliant, whereas the operation performed after 5 days will be out of context.

A terminated contractual interaction is classed as normally terminated if there are no pending obligations because they have been fulfilled. 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 the violation or non fulfilment of an obligation that could occur during an interaction; such violations normally lead to sanctions coming into force and, if these are honoured, then the contractual interaction could still end normally. The concepts presented in this section form a sound basis for constructing contract compliance checking systems and rule based contract languages [38].

3.2 The architecture of the CCC

The overall architecture of the CCC as shown in Figure 3.2 is based on an event condition action (ECA) mechanism. As stated in the previous Section, business events (b-events) are supplied by the business partners to the CCC through the business conversation monitoring channel and carry information about the business operations executed.

3.2.1 Main components

The main components of the CCC architecture (Figure 3.2) are the *Event Queue*, the *Time Keeper*, the *Event Logger*, *Relevance Engine* and the Contract Rule Repository.

- The *ROP set* - stores the current set of rights obligations and prohibitions of the buyer and seller.
- The *b-event logger* - a permanent storage for keeping records about all the events that arrive to the CCC.
- The *b-event queue* - stores business events until they are removed for processing by the relevance engine.
- The *contract rule repository* - contains a list of ECA rules that describe the contract in force.
- The *timer keeper* - keeps track of deadlines associated with each right, obligation and prohibition stored in the ROP sets. Deadlines are set or reset (set/reset) by the relevance engine. When a deadline expires, a timeout b-event is sent to the b-event queue.
- The *relevance engine* - the relevance engine analyses queued events and triggers any relevant rules among those it holds in its contract rules base.

Notice that, in this structure the events in contract rules are the b-events; whereas their conditions correspond to the constraints imposed on the execution of business operations. For instance, a contract clause states that payment must be performed within five days of sending the purchase order. Some conditions, such as the number of failures, are related to the history of the interaction, and consequently their verification involves consultation of the historical records (cons. hist. records) kept by the b-event logger. The actions in the rules include the operations add and delete (add/del) executed against the ROP sets to add and delete rights, obligations and prohibitions. The effect of an action is the updating of the state of the ROP sets after the occurrence of a b-event.

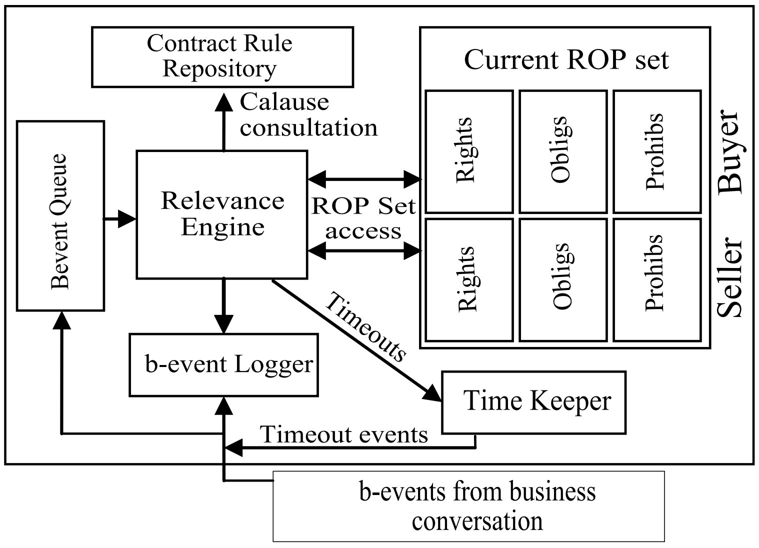


Figure 3.2: The architecture of the CCC

3.2.2 Contract analysis algorithm

A core function of the CCC is to determine if the logged b–events are contract compliant or not. For this, the relevance engine analyses queued events and triggers any relevant rules among those it holds in its contract rules base, following this algorithm:

1. Fetch the first event e from the event queue;
2. Identify the relevant rule for e;
3. For the selected rule r, if conditions C1, C2 and C3 are satisfied (and thus the operation is contract compliant), execute the actions listed in the body of the rule. The main action here is updating such as by the addition and deletion of rights, obligations and prohibitions of the current state of the ROP sets; then return to step 1.

The design and implementation of the CCC and the associated EROP language have been described in a previous study [50], which also illustrates how the clauses of a contract would be coded using EROP. Section 2.2.6 has also shown how the clauses of a natural language contract are written in EROP language.

3.3 The CCC as a reactive system

Before discussing how the CCC can be modelled as a reactive system, the nature of a reactive system is first described and its main properties discussed. A reactive system is a system that responds (reacts) to external events. The need to specify and reason about properties of reactive system motivated the development of temporal logic, which is an extension of the classical logic, specifically adding operators related to time [20]. The temporal logic contains operators such as '$\bigcirc$' (i.e. the next moment in time), '$\Box$' (at every future moment), and '$\Diamond$' (at some future moment) to extend classical logic. These additional operators help designers express correctness properties (e.g. safety and liveness) that they wish to verify against a given reactive system.

3.3.1 Safety properties

The safety properties of a system are claims that something 'bad' will not happen [40]. For example, a safety property dictates that a given activity will never be performed most probably, because it is bad and undesirable. Examples of bad thing in electronic contracts are actions that if happen lead to conflicts. Hence, typical safety properties of contracts are statement claiming that actions that bring the contractual parties into a conflict states never happen. Such properties can be described using temporal formulae of the form '$\Box\neg(...)$' or always not (...). For instance, the following properties P_1, P_2 and P_3 are typical safety properties of contracts, specified in the temporal logic LTL:

$P_1 : \Box\ \neg(pay\_on\_time \wedge penalised)$

$P_2 : \Box\ \neg(obliged\_to\_pay \wedge prohibited\_to\_pay)$

$P_3 : \Box\ \neg(paid \wedge \bigcirc\ obliged\_to\_pay)$

Bad things can be represented by an assertion P_i which is evaluated to be true in exactly those states in which the bad condition holds true. For a safety property to be true, $\neg P_i$ must be an invariant at any moment during the contractual business exchange. From the discussion discussion of contract conflicts in Chapter 2, it is not difficult to realise that a large of contract conflicts can be expressed in terms of safety properties (e.g. P_2), which if violated it would signal inconsistent specifications in the contract rules. The verification of safety properties can be achieved through a search of the system state space. If the invariant is violated by a specification of the system and accordingly concerns the safety property, then there will be a finite behaviour that which shows the invariant violation.

3.3.2 Liveness properties

A liveness property captures the fact that something 'good' must eventually occur [40]. In other words, a liveness property states that a given activity will eventually, at some point in the future be performed, seemingly because it is desirable and good for it to happen. Properties of this type are usually described by temporal formulae of the form '$\Diamond(...)$' i.e *eventually* (...). For instance, the following Q1 and Q2 assertions are liveness properties and can be specified in temporal logic LTL. They claim that something will eventually occur during the system execution.

$Q_1 : \Diamond \text{ (paid} \wedge \text{payment} = 3.0)$

$Q_2 : \Box \text{ (purhase\_order\_cancelled} \implies \Diamond(\text{money\_refunded}))$

In contract applications, examples of good things are actions that when they happen do not lead to conflicts of any type. In order to explain this property, consider the refund business operation (property Q_2) of a contractual business exchange: a customer is penalised or prevented from performing refund by, for instance, not permitting or prohibiting the customer to execute refund operation by erroneous rule. In a situation like this, a sensible form of liveness is associated with the guarantee of a response to the business operation requested.

3.3.3 The CCC reactive system explained

As discussed earlier in relation to the architecture of the CCC, the significance of the ROP sets in the EROP model is that they allow the CCC to be abstracted as a conventional reactive system [38]. Thus, its correctness requirements can be expressed as safety and liveness properties. As a reactive system, the CCC remains in a given state s_i waiting for the arrival of events. When a business event e_j arrives and is determined to represent a contract–compliant operation, the system enters state s_j. The state space heavily depends on the set of valid operations $B = bo_1, ..., bo_n$. For each partner, a given bo_i can belong to at most one of its three ROP sets at any time. This can be illustrated using the following hypothetical contract example from [38] which includes a few rights, obligations, and prohibitions.

- *Cl1*: The buyer has the right to submit a Purchase Order (right), between 9 am and 5 pm, Mon to Fri.
- *Cl2*: The seller has the obligation to respond to the purchase order with acceptance or rejection within 24 hours (obligation).
- *Cl3*: The seller's failure to respond to the Purchase Order within the 24 hours deadline will result in abnormal contract termination with offline settlement.
- *Cl4*: 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).
- *Cl6*: The seller is obliged to deliver the goods within seven days of receiving payment (obligation).
- *Cl7*: This contractual transaction terminates when either
  - *Cl7.1*: The seller rejects the purchase order, or
  - *Cl7.2*: The seller successfully delivers the goods.

In Figure 3.3, Ø represents the empty set; e, c, T, Inv, TO and Sub stand for business event, condition, true, invoice, timeout, and submit, respectively. Notice that, for simplicity, only the name and execution outcome attributes of the business

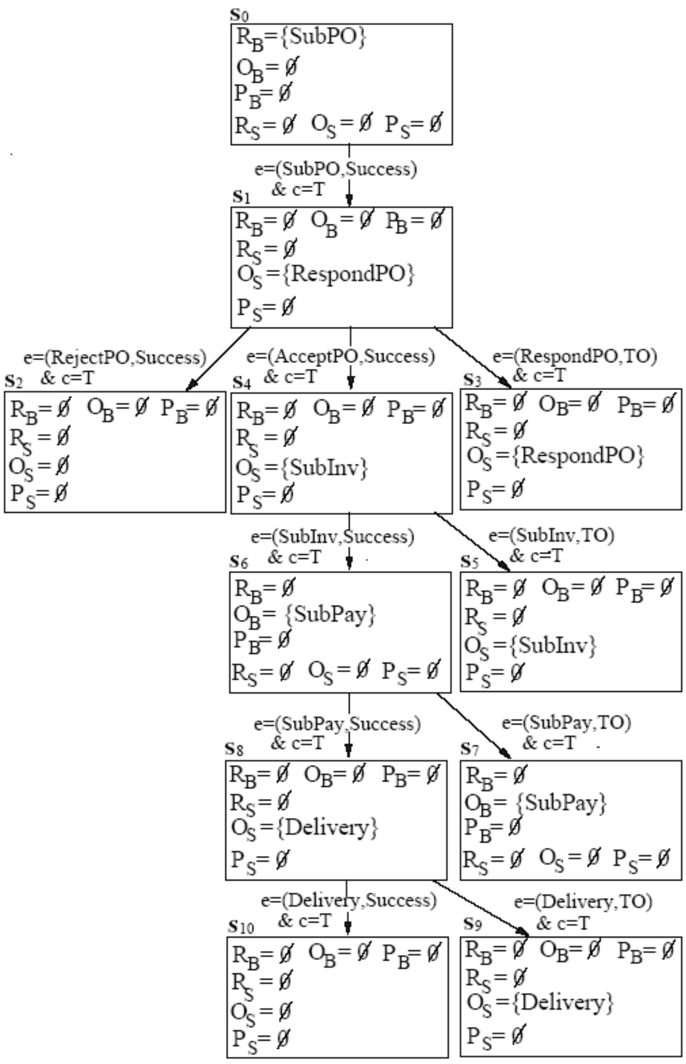


Figure 3.3: CCC as reactive system

events are shown in this illustration. Likewise, only the business events with Success and TO execution outcomes from contract compliant operations are shown. Business events with neither *InitF* ,*TecFail* or *BizFail* execution outcomes nor those related to unknown and out-of-context executions will be shown. In the discussion, Cl1, Cl2, Cl3, and so on refer to the clauses of our contract example.

State s_0 corresponds to Cl1: The state s_0 corresponds to the start of the interaction where the seller has the right to submit a *PO*. Once started, the interaction will complete either normally resulting in one of the acceptable final states with no pending obligations or abnormally in a nonacceptable state with obligations still pending thus indicating a contract violation.

State s_1 corresponds to Cl2: From s_0 the interaction enters s_1 when a business event $e = (SubPO, Success)$ is notified and the required conditions (c = T) hold. The right *SubPO* has been removed from R_B as it has been exercised by the execution of the *SubPO* operation. The obligation *RespondPO* is added to O_S.

State s_4 corresponds to Cl4: From s_1 the interaction progresses to s_4 when the seller successfully executes the *AcceptPO* operation within the 24 hours deadline. The obligation *RespondPO* has been removed from O_S since it has been fulfilled by the execution of the *AcceptPO* operation and the obligation *SubInv* has been added to O_S; thus, it appears as a pending obligation in s_4.

State s_2 corresponds to Cl2 and Cl7.1: From s_1 the interaction progresses to the acceptable final state s2 when the seller successfully executes the *RejectPO* operation within the 24 hours deadline constraint. The obligation RespondPO has been removed from O_S since it has been fulfilled by the execution of the operation *RejectPO*.

State s_3 corresponds to Cl3: From s_1 the interaction progresses to the abnormal final state s_3 if the seller fails, for example, due to *BizFail* or *TecFail* reasons to honour his *RespondPO* obligation before the expiry of the 24 hours deadline. *RespondPO* is left in O_S as a pending obligation.

State s_6 corresponds to Cl5: From s_4 the interaction progresses to s_6 when the seller successfully executes *SubInv* operation within the 24 hours deadline constraint. The obligation *SubInv* has been removed from O_S since it has been fulfilled by the execution of the *SubInv* operation and the obligation *SubPay* has been added to O_B; consequently, it appears as a pending obligation in this state.

State s_5 corresponds to Cl4: From s_4 the interaction progresses to the abnormal final state s_5 when the buyer fails to successfully execute the operation *SubInv* within the 24 hours deadline to fulfil his SubInv obligation. The obligation *SubInv* is left pending in O_S.

State s_8 corresponds to Cl6: From s_6 the interaction progresses to s_8 when the buyer successfully executes the *SubPay* operation within the seven day deadline. The obligation *SubPay* has been removed from O_B since it has been fulfilled by the execution of the operation *SubPay*, and the obligation Delivery has been added to O_S; thus it appears as a pending obligation in s_8.

State s_7 corresponds to Cl5: From s_6 the interaction progresses to the abnormal final state s_7 when the seven day timeout to execute the operation *SubPay* expires, leaving the obligation *SubPay* pending in O_B.

State s_{10} corresponds to Cl7.2: From s_8 the interaction progresses to the acceptable final state s_{10} when the seller successfully executes the Delivery operation within the 7 day deadline. Notice that in s_{10} the ROP sets are left empty, with no pending rights, obligations or prohibitions.

State s_9 corresponds to Cl6: From s_8 the interaction progresses to the abnormal final state s_9 when the seven day timeout to successfully execute the operation Delivery expires. The Delivery obligation is left pending in O_S.

3.4 Conflicts in EROP contracts

Representing natural language contracts in the EROP language provides a better structured set of contract clauses than those natural language representations. However, this elegant representation of contract clauses as ECA rules still requires the careful analysis of consistency between the contract rules. Conflicts that lead to inconsistent contract rules could have been inherited from the original contract clauses or might arise during the manual conversion of natural language contract into ECA rules. In either case, the analysis of these conflicts is advisable before producing an executable version of the contract. The advantage of using EROP language is that it produces a high level representation as ECA rules of the contract clauses in such a way as to be familiar to both business managers and software engineers.

To this end, and for the purpose of the automatic analysis of contract clauses modeled as ECA rules (as in the EROP language), two main types of conflicts are considered here: contract-independent and contract-dependent conflicts. The former identify contract properties that must satisfied by all contracts. For example, at run time, a contract should not simultaneously grant a role player a right to perform an operation and simultaneusly impose a prohibition on it. For instance, S_0 in Figure 3.3 shows that the buyer is permitted to submit a purchase order, however, an erroneously configured rule may simultaneously prohibits him from doing so.

In the latter conflicts, properties are considered that may occur in specific contracts. For example, consider Cl2 clause from the contract shown in Section 3.3.3: The buyer has the obligation to pay the due amount within seven days of receiving the invoice, a conflict may occur here if the seller requests payment after the buyer rejects the purchase order. A possible erroneous contract execution which could lead to this conflict is: $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_6$ (see Figure 3.3). Both contract–dependent and contract–independent types of conflicts are discussed in more detail below.

3.4.1 Contract-independent conflicts

These conflicts are considered to be contract–independent because they are specific to individual contracts. Such conflicts may appear in contract clauses that specify deontic properties of business operations such as rights, obligations and prohibitions. These deontic properties can be treated as three mutually exclusive values. This means that, at each point in time a business operation can only have one restricted property, such as right, obligation or prohibition during an execution trace of the contract. In fact, this assumption is rooted in SDL logic [5], and can help us to capture a set of complex and hard-to-detect conflicts in contract clauses. These conflicts are hard to detect in contract couched in natural language because they are informally specified, structureless, scattered between the contract clauses, or sometimes indirectly elicited from the contract language. The SDL defines an inter-dependency between the rights(permissions), obligations and prohibitions. Consider this example from [23]; if obligation is taken as a primitive, permission and prohibition are defined as follows:

- It is permitted that A: *Permission(CP, A)* $\equiv \neg$*Obligation(CP,* $\neg A$*)*
- It is prohibited that A: *Prohibition(CP, A)* $\equiv \neg$ Permission (CP, A) $\equiv$ Obligation (Agent,$\neg A$)

where A denotes to either a specific action of a Contract Party (CP) or a system state. These definitions capture the common intuition that what is obligatory must be permitted, and that what is prohibited can neither be permitted nor obligatory [23]. It follows that a wide range of contract conflicts may occur in a *contract execution trace* (cet_i) with one or more *business operation* $bo_i \in BO$ from the set of all business operations. The existence of a conflict can be formally captured using the following expression: $bo_i \vdash \langle R(bo_i) \wedge O(bo_i)\rangle \,||\, \langle R(bo_i) \wedge P(bo_i)\rangle \,||\, \langle O(bo_i) \wedge P(bo_i)\rangle$, where:

$\langle R(bo_i) \wedge O(bo_i)\rangle$ is a simultaneous right and obligation to bo_i

$\langle R(bo_i) \wedge P(bo_i)\rangle$ is a simultaneous right to and prohibition on bo_i

$\langle P(bo_i) \wedge O(bo_i)\rangle$ is a simultaneous prohibition on and obligation to bo_i

Let us use cet_i to refer to an arbitrary contract execution trace, where an execution trace cet_i, refers to the execution of business operations for one or more contract participant (also called a computation or a run) from the set $CET = \{cet_1, cet_2, ..., cet_n\}$; $n \geq 1$ of contract execution traces.

Each cet_i includes a subset of business operations $bo_i \in BO$, and each bo_i has a deontic property P which characterises the contractual business operation as right, obligation or prohibition, and its values are mostly updated by the contract rules in response to actions of the contract parties. The initial state and final state of an execution trace cet_i are $bo_0, bo_f \in BO$ respectively, where the former (bo_0) is a business operation which starts the contract execution trace cet_i, and bo_f the business operation which concludes the contract execution trace. This envisioning of contract execution suggests that the problem of detection of contract independent conflicts is about ensuring that each cet_i is conflict-free. Moreover, if a conflict is captured, the responsible contract rule causing that conflict can be identified by means of analysis of the conflicting execution tract cet_i. The problem then can be solved by fixing the contradictory clauses.

3.4.2 Contract-dependent conflicts

Contract-dependent conflicts may occur when specific contract-dependent correctness requirements are invalid or lead to contradictions between contract parties. Most of the conflicting scenarios shown in Section 2.3 fall in to this category. Let us return to the Buyer/Seller contract at Section 3.3.3. We stated two crucial requirements: a) the contractual interaction always terminates in one out of several acceptable final states; b) the contractual interaction always progresses through a valid sequence of states. In particular, it is desirable to verify that 'a Seller is penalised a fixed amount of money if he fails to deliver the purchased items on time'. Likewise, verify that the operation 'reject purchase order' always takes place after the occurrence of an 'offer' operation and within the permitted period for rejection. Recall that the term conflict used to refers to an inconsistency within a single clause or two or more contract clauses that may lead to a contradiction between the contract parties.

3.5 Discussion

This Chapter discussed the main components of the contract compliance checker of EROP model and how it can be abstracted as a reactive system. As discussed in Section 2.4.2, an abstract model of the CCC has been modelled using Promela and verified using Spin model checker [36]. Thus, different properties of a business contract model have been verified using the general purpose model checking tool (Spin).

We believe that the facilities of the Spin model checker can be reused in a better way to assist in the verification and validation of business contracts. In particular, the standard Promela can be extended [48] with the concept of business operations and operators to manipulate it. For example, in extended Promela, the designer can include in his model operations such as assign obligation delivery to the seller, and express queries like 'is the buyer currently obliged to pay?' Like many other model-checkers, Spin can accept and verify correctness constraints, for example where the buyer is never obliged to and prohibited from paying, abstracted as safety and liveness properties and expressed in LTL formulae [49].

The next Chapter introduces a formal *Contractual B2B model* (CB2B) based on EROP concepts. We discuss how Spin and an extended version of Promela that adds specific constructs aimed at capturing concepts frequently found in contracts can work together as a pragmatic tool for contract model checking. Such a tool can be built using the concepts of contract compliance checking CCC described at the beginning of this Chapter (Section 3.1.1). With the appropriate level of abstraction of the contract and contractual B2B environment, contract models of manageable sizes can be built with extended Promela and verified by Spin. So that the ECA rules expressed in extended Promela can be automatically analysed using well known model checking techniques and readily available technology.

Chapter 4

CB2B formal model

The contractual business–to–business model (CB2B) considered in this chapter abstracts the behaviour of the contract compliant checker CCC of the EROP language. It is empowered with a set of operations which facilitate access to the contract elements such as role player obligations held in the system memory. In earlier work, Promela was used to model the CCC and the rules [36]. The lack of data types other than the built-in types bit, byte and array is seen as a serious restriction to the use of Promela as a specification language rather than a protocol modeling language. However, the language can be extended with user–defined data types [48]. Using its typedef construct, and the inline and/or cpp macros can be used to define operations on such new data types. On this basis, we have implemented an abstract data type extension to the standard Promela called BIS_OP. A set of operations on the BIS_OP data type have also been implemented to maintain information about ROP sets. These additional concepts not only simplify the task of writing rules, but also help the designer to specify the correctness requirements in LTL with parameters that map directly to contract concepts. Here, different aspects of the the CB2B model, its implementation and the CB2B's model checking framework are described.

4.1 CB2B model: principles of operation

In this thesis, we have used the reactive system paradigm to design and develop an abstract architecture for contractual business to business interactions. Our architecture (Figure 4.1) is general and abstract enough that it can be used for i) verifying the logical correctness of contracts regarded as ECA systems; ii) implementing them and iii) testing the correctness of their implementations. Another salient point of the architecture is its modularity which allows reuse of its components at design, implementation and testing stages.

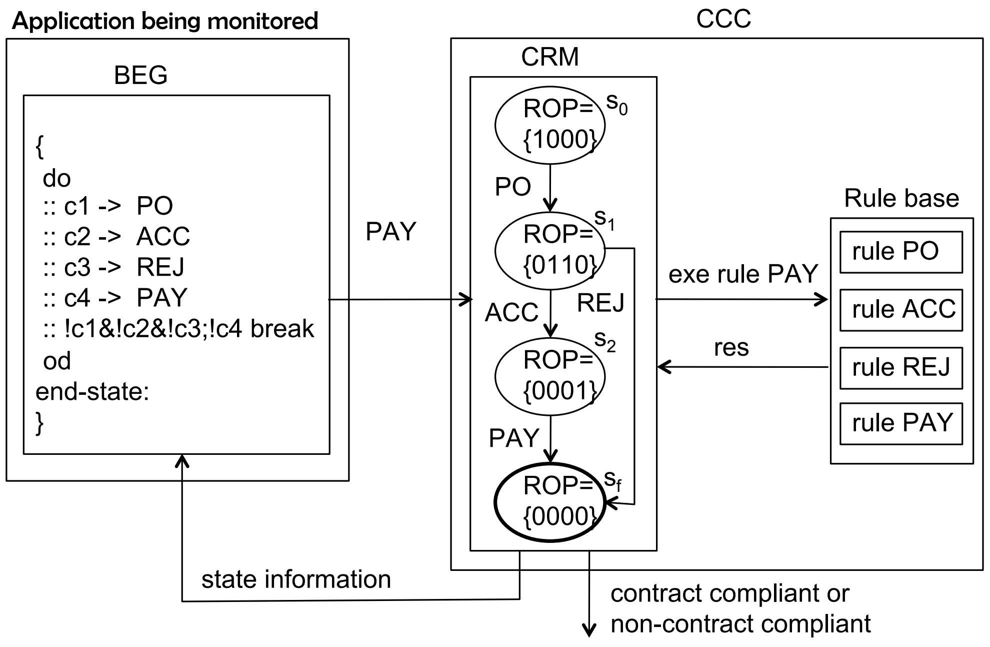


Figure 4.1: CB2B as a reactive system

To explain the concepts of CB2B model, we will use the following hypothetical contractual clauses:

1. The buyer can place a *Purchase Order (PO)* with the seller to buy an item.

2. The seller is obliged to reply with either *Accept (ACC)* or *Reject (REJ)* after receiving purchase order.

3. A rejection shall be taken as completion of the contractual interaction.

4. The buyer shall place *Payment (PAY)* after receiving an acceptance of the purchase order.

The contract example is oversimplified, for instance, it does not account for exceptional situation like unsuccessful completion of the execution of an operation. Neither does it include deadline to complete them. In the following discussions, we use the symbol $\rightarrow$ to denote the *business events chronological sequence*, thus $a \rightarrow b$ denotes that a happened before b. The main components of the CB2B model are discussed in detail next.

4.1.1 The business event generator (BEG)

We model the external environment (*application being monitored*) by a Business Event Generator (BEG) which is responsible for generating finite sequences of events that represent potential contract runs of the contract being monitored. A *contract run* (also called an execution path) is a specific execution of the contract from its initial to its final state following one of the paths encoded in the contract. For example the contract example includes two contract runs: $PO \rightarrow ACC \rightarrow PAY$ and $PO \rightarrow REJ$.

We assume that events are observed at the granularity of outcome events (see section 3.1.1). Under this assumption, monitoring of a contractual interaction based on the execution of operations from the set $B = \{bo_1, \ldots, bo_n\}$, reduces to monitoring the occurrence of business events from the set $E = \{e_1, \ldots, e_n\}$. Each e_i notifies about the execution of its corresponding bo_i and contains in addition to the name of the operation, the termination status which could be S (success), BF (business failure) or TF (technical failure), the timestamp and as many additional attributes as necessary. Since the elements of B are one–to–one mapped to the elements of E, we often use the name of the business operation (for example *pay*, *deliver*, *cancel*, etc.) to refer to the event that notifies of its execution, under the understanding that the event includes all the attributes needed by the rules for processing it.

At the heart of the BEG is an *do—od* construct with guarded (*::*) commands, that can generate n types of events, for instance, events from the set E and if needed, events that do not belong to E. In the example shown in the figure, the BEG is programmed to generate events that correspond to the four business operations (PO, ACC, REJ and PAY) included in the contract example, so it generates the events: PO, ACC, REJ and PAY. In the same order c_1, c_2, c_3 and c_4 represent conditions that evaluate to TRUE (T) or FALSE (F). The $\rightarrow$ arrow at the right of the conditions means *generate*. For example, $c_1 \rightarrow PO$ means that if c_1 evaluates to T, event PO is available for generation; conversely, $c_1 = F$ disables this possibility. Similarly c_2 evaluates to T, event ACC becomes available for generation. When two or more conditions evaluate to T one of the guarded commands is chosen non–deterministically and the corresponding event is generated. When none of the conditions evaluates to T (! represents a logical negation), the *do–od* block is abandoned ($!c_1, !c_2, !c_3, !c_4; break$) and the BEG stops at a valid end state (*end-state:*). This transition means the completion of a contract run and consequently, of an event sequence. If necessary, the BEG can be re–started to generate another sequence of events (that correspond to another contractual run), by means of bringing it manually or automatically to the beginning of the block.

The conditions can be used for constraining the type and number of events included in the sequences. At one end of the spectrum one can use conditions that always evaluate to T which is equivalent to using no conditions. At the other end of the spectrum one can use conditions that always evaluate to F. In the first situation the BEG will generate events randomly whereas in the second, no events will be generated at all. In practice, the conditions include parameters that contain information about the state of the CCC which is fed back to the BEG through the *state information* line. For example, we can place a condition that prevent a given event from being included more than one time in a given sequence. Similarly, we can use conditions to force the BEG to generate only events that are likely to be declared contract compliant by the CCC. As we will explain it later on, we use conditions at validation time to constrain the state space of the validation and focus the attention to specific properties at different stages of the design.

4.1.2 The contract rule manager (CRM)

The main component of the CCC when regarded as a reactive system (see Figure. 4.1) is a Contract Rule Manager (CRM) which is supported by a *Rule base*. The CRM is the dynamic component and is responsible for storing and managing the state of the system. In our contractual applications, it stores and keeps track of the state of the contractual interaction between the business partners. The current state of the CCC —and in particular, of the CRM— is determined by the business partners' rights, obligations and prohibitions currently pending (awaiting execution in the current state).

The formal definition of the current set of Rights, Obligations and Prohibitions (ROP) is presented in [38]. Yet to make this discussion self—contained, we will briefly discuss the concept here.

For a formal definition of the current set of *Rights*, *Obligations* and *Prohibitions* (ROP) (see Section 4.3.1), we use boolean variables to represent the rights, obligations and prohibitions of the contractual parties. These variables are turned *1* and *0*, *ON* and *OFF*, respectively, as the contractual interaction progresses. *ON* indicates pending (in force, enabled, etc.), whereas *OFF* indicate disabled (revoked, etc.). Let us return to our contract example which includes, in particular to the clause states: a buyer's right to execute *PO*, a seller's obligation to execute either *ACC* or *REJ* and a buyer's obligation to execute *PAY*.

We can use a vector ($ROP = \{0000\}$) of four boolean variables to represent the right and the three obligations, where the left most bit represents the buyer's right the to execute PO, the second bit represents the seller's obligation to execute ACC, the third bit represents the seller's obligation to execute REJ and the fourth bit represents the buyer's obligation to execute PAY.

Thus $ROP = \{1000\}$ indicates that a right to execute PO is currently granted, $ROP = \{0110\}$ indicates that obligation to execute either ACC or REJ is currently pending, $ROP = \{0001\}$ indicates that an obligation to execute PAY is currently pending, $ROP = \{0000\}$ indicates that no rights, obligations or prohibitions are currently pending. As shown in Figure. 4.1, we use this ROP variable to determine the states of the CRM as the contractual interaction progresses. Thus in the initial state s_0 the buyer's right to execute PO is granted. The CRM progresses to state s_1, presumably when the buyer executes PO where the right to execute PO is disabled and the obligation to execute ACC or REJ is pending. From state s_1 the CRM might progresses to state s_2, presumably when the seller's excutes ACC, where the obligation to execute PAY is pending. Alternatively, from state s_1 the CRM might progress to state s_f, presumably, when the seller to reject the purchase order executes REJ. From state s_2 the CRM progresses to state presumably when the buyer executes PAY, where no rights, obligations or prohibitions are left pending.

4.1.3 The CB2B model rule base

The rule base of CB2B model is a file that contain the list of ECA rules (e.g. *rule PO*, *rule ACC*, *rule REJ* and PAY) that model the contractual clauses. In the architecture of CB2B model, events from the set E and rules from the rule base are in one to one correspondence. Thus for each individual event e_i there is an individual rule R_i that contains the logics to handle it. In our particular example, the rule base contains a rule called PO to handle the event PO produced by the BEG, a rule called ACC to handle the event ACC, a rule called REJ to handle the event REJ, and a rule called PAY to handle the event PAY.

Upon request from the CRM, an individual rule can be executed (for example *rule* PAY) to produce a response (res) that indicates whether the event under examiation correspond to a contract compliant business operation or not; equally important, the execution of a rule alters the current state of the set of rights, obligations and prohibitions and consequently, the current state of the CRM.

4.2 CB2B model implementation in Promela

The CB2B model is constructed with Promela (the Spin input language) using two processes BEG and CRM and two uni-directional channels BEG2R and R2BEG (see Figure 4.2). The business event generator (BEG) process represents the external world. The contract rules manager (CRM) process together with the ROP sets and the ECA rules represent the CCC. The contract rules are composed in a separate file and offered to CRM via the usual *#include* mechanism. There is rule for each business event be_i representing the outcome the execution of an operation. So for a business operation say, 'submit purchase order', there will be rule for the operation terminating successfully (S), or optionally, depending on whether the contact has clauses dealing with failure outcomes, for the operation terminating in a technical failure (TF) or in a business failure (BF).

4.2.1 Execution cycle

The executable behaviour of the CB2B Promela model can be seen as the following set of read and write process operations:

1. The BEG process writes the generated event be_i in the input channel BEG2R;

2. The CRM reads be_i from the input channel; and

3. The contract rule R_i corresponding to be_i event is activated;

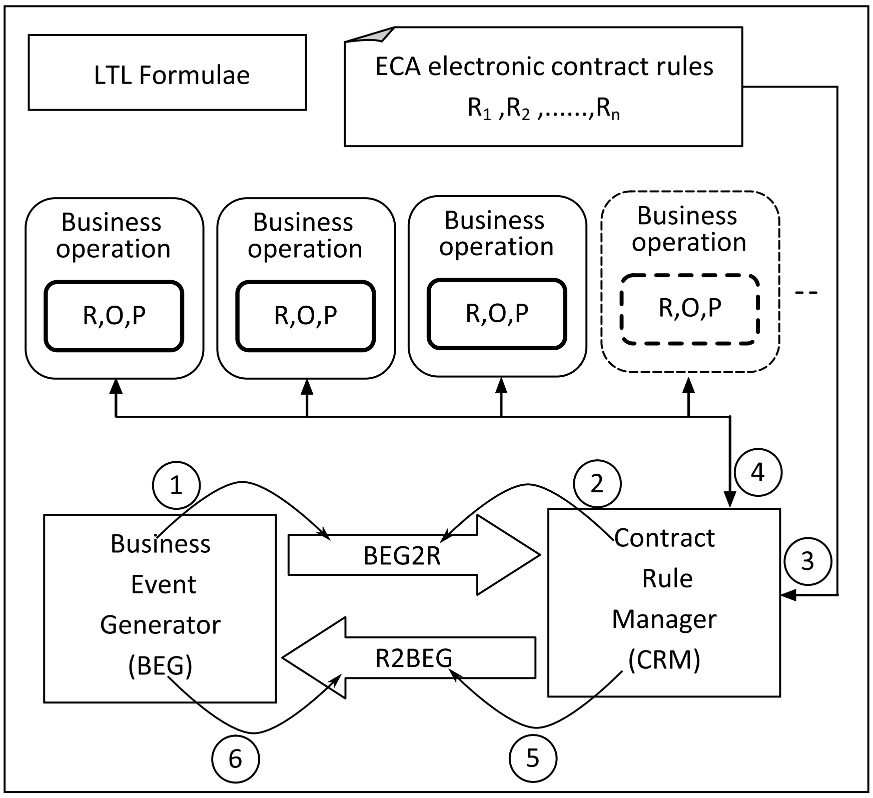


Figure 4.2: CB2B formal model.

4. The R_i checks be_i against the ROP sets (condition C2), and executes the action if the associated condition, C3, is satisfied;
5. The R_i writes the decision made regarding contract compliance in the output channel R2BEG;
6. BEG process extracts the decision from the output channel to resume the event generation process

With the CB2B model and extended Promela, a contract is specified by declaring a set of business operations, role players, rules and global variables necessary for recording some aspect of contract execution that might be required by the rules or LTL formulae. At the core of the extension made is the abstract data type Business Operation (BIS_OP). Each instance of Business Operation is always associated with a ROP set and supports a list of operations (methods) such as SET_R(a, 0) and

IS_R(a) to respectively grant the right to execute operation a to a role player and to query if the right is enabled. In addition, operations and control structures have been implemented as syntactic sugar to supplement Promela constructs such as statement sequencing, atomic sequencing, concurrent execution and case selection.

4.2.2 Key features

The CB2B model characterised by unique features facilitating the composition of contract models as a set of ECA rules and allow their verification using model checking techniques. Below, we list the salient features of CB2B model that facilitate model checking of contracts using Spin:

1. Supports an expressive high-level ECA rule notation [1]. It is a *rule-based contract language* aims to capture and represent the natural language contract in more rigorous and concise manner suitable for both human and computer processing.

2. Facilitates the construction of LTL formulae such as *always Prohibited (*CP_i*,* A_i*)*, which means action A_i of a contract party (CP_i) must remain prohibited in every state during the contractual business exchange. Similarly other temporal properties can be used, such as *next*, *eventually* and *until*.

3. The ability to specify contrary-to-duties (CTDs) and contrary-to-prohibition(CTPs) where the former specify what obligation is to be demanded when the original obligation is not fulfilled, and the later specify what penalty is to be applied in cases where a prohibition is violated.

4. The automatic detection of contract-independent conflicts. This is a key feature of the CB2B model which is enabled through embedded assertions that automatically assert contract-independent properties that must hold for all contracts.

We discussed the contract-independent conflicts in Section 3.2.2. The macros responsible for the assertion of the contract–independent properties can found in Appendix A.

4.2.3 Example

This section uses a simple example to illustrate how ECA rules are coded in a CB2B model. Later sections show the full implementation and verification of certain properties of this example. Although it is simple and small, the example reveals important problems linked to the manual translation of contracts into a machine-oriented format (in this case ECA rules). The contract says: There is an obligation to choose between doing 'b' or 'c' after 'a', and a prohibition on doing 'b' if 'b' has been performed.

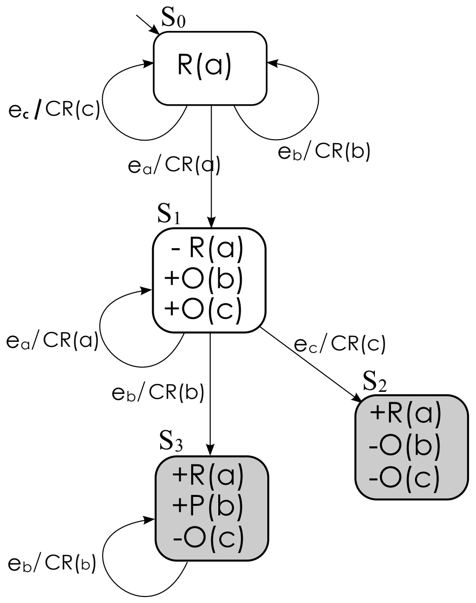
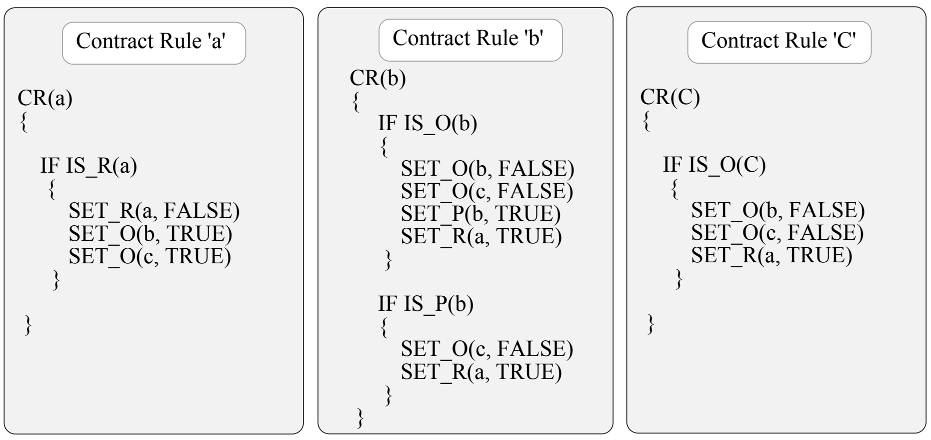


Figure 4.3: State transitions in the executable contract model

Figure 4.3 shows that each state is characterised by the contract entities (R)ights, (O)bligations and (P)rohibitions that become active or inactive after each state transition. The plus '+' and the minus '-' are used to show the effect of a business event Be_i on the contract status where '+' and '-' model the granting and revoking of a contract rights, obligations and prohibitions. The contract transitions occur when business events are generated, ROP sets evaluated and any other conditions might have been obtained during the business exchange are held to be true in the transition's source state. In Figure 4.3 this conditional transition is shown as $\mathrm{be}_i/\mathrm{CR}(\mathrm{be}_i)$. Figure 4.4 shows the pseudo–code of contract rules a, b and c that are manually extracted from the contract text described above.



Contract Rule 'a'

```
CR(a)
{

  IF IS_R(a)
   {
     SET_R(a, FALSE)
     SET_O(b, TRUE)
     SET_O(c, TRUE)
   }

}
```

Contract Rule 'b'

```
CR(b)
{
   IF IS_O(b)
   {
      SET_O(b, FALSE)
      SET_O(c, FALSE)
      SET_P(b, TRUE)
      SET_R(a, TRUE)
   }

   IF IS_P(b)
   {
      SET_O(c, FALSE)
      SET_R(a, TRUE)
   }
}
```

Contract Rule 'C'

```
CR(C)
{

  IF IS_O(C)
   {
      SET_O(b, FALSE)
      SET_O(c, FALSE)
      SET_R(a, TRUE)
   }

}
```

Figure 4.4: ECA Contract rules

The example illustrates how the CB2B model for monitoring the contract compliance reacts to stimulus of business events be_i and activates the corresponding contract rule CR_i to update the contract status, where updates will alter the current ROP set values. The vocabulary of this contract example consist of (a, b, c). The set of business events in this example are be_a,be_b,be_c, and the occurrence of each business event be_i activates a contract rule CR_i in which a specific behaviour of a contract party is encapsulated. For example, in S_0 the model is initialised with the *right* to execute 'a', which is shown as R(a) in s_0), and thus the CB2B model would react to the business event be_a by activating the contract rule CR(a) and then evaluate any other conditions which may hold in S_0 to change the contract status to S_1. Similarly the CB2B model reacts to 'b' and 'c' and consult CR(b) and CR(c) but the system state does not change because none of them were rights, obligations or prohibitions.

A contractual trace_i, which is a finite sequences of business events be_i for which the contract rules CR_i are evaluated and the ROP set is updated, is considered to be concluded if the eventual objective of the contract in force is reached; Figure 4.3 concluded contract states are S_1 and S_2. For example, $\mathrm{trace}_1 : a \rightarrow c$ implies that if business event be_a is fired in S_0 and the contract rule $\mathrm{CR}(\mathrm{Bo}_a)$ is activated then two obligations O(b) and O(c) become active in state S_1. This means that either O(b) or O(c) can be fulfilled from S_1 forwards. Thus, the emergence of an event corresponding to any of these would discharge them both. So in trace_1 if a contract party has chosen the action 'c'), ignoring the two obligations O(b) and O(c) in S_1, this is considered a violation of the whole contract and may promote legal action. Gaining the right to perform 'a' again, R(a) is added in S_2 is similar to the initial state of the contract, and thus the contract compliance checking process is restarted from S_0 to monitor future business events. Other possible contractual traces could be: trace1: a, b, c, a, b, trace2: a, b, b, b, b and trace3: a, b, a. In fact, real contractual traces would be more complex than those shown here, and may include many reparations of the original obligations.

4.2.4 Verification of contract properties

The verification of contract properties with the support of the CB2B model can be achieved through LTL formulae or assertion statements. Different contract properties can be formalised using temporal logic formulae and directly verified with the Spin model checker. With Spin version 6, an LTL formula is specified globally with the following syntax:

```
ltl [ name ] '{'  formula '}'
```

For example, LTL formulae can be written as follows:

```
ltl p1 { []<> p }
ltl p2 { always eventually p }
```

Spin offers both symbolic (p1) and textual (p2) alternatives to specify the temporal properties of an LTL formula. The LTL formulae specified in p1 and p2 are identical, they both mean that 'p' will happen at least once in the future. With regard to the verification of contract properties using CB2B model, with the textual form one could write the following formula:

```
ltl p3 { always not(obliged(b,seller) && prohibited(b,seller))}
```

That is, it will always not be accepted for a seller to be simultaneously obliged to and prohibited from doing the same action 'b'. Recall that properties of this kind are generic and categorised as contract independent properties, as discussed in Chapter 2.

The contract-dependent properties are different, being specific to each contract. For instance, consider the following property in p4:

```
ltl p4 {always(done(b,seller) implies eventually always
                                      (prohibited(b,seller)))}
```

That is, whenever an action 'b' is executed, it always implies that eventually 'b' will be prohibited. In cases of violation, the contract rules as well as the contract document will be revisited again to resolve the problem. In some cases a contract developer contacts business managers to clarify the ambiguity of the contract.

Verification through assertions is a simple verification tool offered by Spin. The language Promela is supplemented with a construct called the assert statement. Statements of the form assert(boolean condition) are always executable. If the boolean condition holds, then the statement has no effect. If, however, this condition does not hold, then the statement will produce an error report during the execution of verification with Spin.

Given a CB2B model and a set of contract rules written in extended Promela, in describing a system comprising BEG and CRM processes we can use Spin for two purposes. Firstly, it can execute the CB2B model via ('*Spin program_name*'); that is, performing on a run of the contract, carrying out random choices where necessary. Alternatively, it can generate a structure representing 'all' of the possible runs of the system (via '*Spin -a program_name*'); acting effectively as a generating automaton describing all possible behaviours of the program.

4.3 The CB2B model notation

This section introduces a high level notation to represent a contract as ECA rules and to specify its basic entities such as business operations, business events and role players in order to allow the verification of certain properties of e-contracts using model checking techniques. The primitives that we have added to Promela in this notation are at user level only without any changes to the Spin input language, and can be used to build contract models at different levels of abstraction for the purpose of exploring specific properties of the system.

4.3.1 Mapping contract entities into Promela

Programs written in Promela basically consist of processes, message channels, and variables. Processes are global objects that represent the concurrent entities of the distributed system. Message channels and variables can be declared either globally or locally within a process. Processes specify behaviour, and channels and global variables define the environment in which the processes run. Promela models can be analyzed with the Spin model checker to verify that the modeled system produces the desired behaviour.

In order to map the contract entities to Promela, the language can be extended to add user defined data types and to offer a high level notation to configure contract entities. A key feature of the high-level notation is that it hides many of the intricate details of the construction of state machine models using pure Promela. It enables a designer to directly encode a contract for model checking as ECA rules in terms of the contract entities of business operations and role players with their rights, obligations and prohibitions. This facilitates the process of the verification of certain properties of a contract using model checking techniques. We will explain next the new constructs and extension we added to Promela to facilitate contract model checkering with Spin and the CB2B model.

4.3.1.1 Role players

A role player is an entity that participates in the execution of business operations on behalf of the contracting business parties. We extend Promela with a *RolePlayer* construct, so the designer can use it to declare *RolePlayer(RolePlayerName1, RolePlayerName2, ...)*. For example the statement *RolePlayer(Buyer, Seller)* declares a Buyer and a Seller role players. Using our macros extensions, the declaration of role player is translated automatically into the Promela basic *mtype* data type, as well as, a declaration of a *Bitvector*. The *Bitvector* construct (explained below) is initialised with each role player declaration. It holds a history of the execution status (1 or 0) of the business operations performed by that role player during the contract run. The macros mapping this high level representation of *RolePlayer* into Promela basic data types are:

```
=================================================================
*                  Counts the number of arguments               *
*       TAKES BETWEEN 1-10 ARGUMENTS (gcc's testsuite)          *
*                                                               *
=================================================================
#define gnu_count(y...)   _gnu_count1 ( , ##y)
#define _gnu_count1(y...) _gnu_count2 (y,10,9,8,7,6,5,4,3,2,1,0)
#define _gnu_count2(_,x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,n,ys...) n

=================================================================
*   Macros for role player - (declare up to three role players)  *
=================================================================
#define RolePlayer(types...) _ROLE_PLAYER_N(gnu_count(types),types)
#define _ROLE_PLAYER_N(n,types...) _ROLE_PLAYER(n,types)
#define _ROLE_PLAYER(n,types...) _ROLE_PLAYER_##n(types)
#define _ROLE_PLAYER_0()
#define _ROLE_PLAYER_1(a) mtype={a} \
BITV_32 a##exTrace;
#define _ROLE_PLAYER_2(a,b) mtype={a,b} \
BITV_32 a##exTrace; \
BITV_32 b##exTrace;
#define _ROLE_PLAYER_3(a,b,c) mtype={a,b,c} \
BITV_32 a##exTrace; \
BITV_32 b##exTrace; \
BITV_32 c##exTrace;
```

The *Bitvector* in the *RolePlayer* declaration is an unsigned piece of memory (see [48] for the full implementation of the *Bitvector*), where each bit can be individually set, reset and tested. Promela's built-in integer types have been used to represent the bit vectors: *byte* (max 8 bits), *short* (max 16 bits) and *int* (max 32 bits) . To show an example, the following is a declaration of a *Bitvector* of 32 bits named *Buyer_Execution_Trace*:

```
BITV_32 Buyer_Execution_Trace;
```

After this declaration, the 32 bits of the *Buyer_Execution_Trace's* Bitvector can be manipulated in different ways. The following list shows the set of operations that can be used with a *Bitvector*:

```
SET_ALL_0(Buyer_Execution_Trace) to set all the vector bits with zero
SET_ALL_1(Buyer_Execution_Trace) to set the vector bits with one
SET_1(Buyer_Execution_Trace,i) to set the vector bit in index i with one
SET_0(Buyer_Execution_Trace,i) to set the vector bit in index i with zero
```

Below is a list of the macros that define operations RESET (rolePlayer), SET_X (boName,rolePlayer) and IS_X (boName,rolePlayer) that built using the *Bitvector* operations.

```
==================================================================
*    Macros to set, test or reset role player execution trace    *
```

```
=====================================================================
#define SET_X(name,rolePlayer) \
SET_1(rolePlayer##exTrace, name##_##bo.id)

#define IS_X(name, rolePlayer) \
IS_1(rolePlayer##exTrace, name##_##bo.id)

#define RESET(rolePlayer) \
SET_ALL_0(rolePlayer##exTrace);
```

The macro *RolePlayer(RolePlayerName1)* concatenates the parameter *RolePlayerName1* (of *mtype* data type) with the suffix *exTrace* and declares it as a global *Bitvector*. For example, declaration such as *RolePlayer(SELLER, BUYER)* declares Seller and Buyer of Promela *mtype* data type and two bit vectors *SELLERexTrace* and *BUYERexTRace* respectively. The *RolePlayer(RolePlayerName1)* macro use the concatenation operator ## to affix each role player name with 'exTrace' to distinguish it. In a contract model, each role player is initialised with a set of business operations (see next section). During the contract execution these business operations can be manipulated in different ways. For example, if the contract declares 'Cancel' business operation, then operation such as SET_X(Cancel, Seller) will alter the execution trace of the seller and use the seller's *Bitvector* to flag the business operation Cancel as executed. In fact, what happens is that a command such as SET_X(Cancel, Seller) issues the Bitvector command SET_1(executionTraceSeller, Cancel_bo.id).

The other commands IS_X(name, rolePlayer) and RESET(rolePlayer) operate in the same manner, and both make use of bitvector operations. The former can be issued as IS_1(exTraceSeler, Cancel_bo.id) to assert that the business operation has been executed, whereas the later is used to reset the whole execution trace of the Seller to zeros. The latter normally performed when a contract goal has been reached in particular run.

4.3.1.2 Business operations

Business operations are the activities (for example, purchase order, payment, cancellation, delivery) that contractual parties are expected to execute to observe their rights, obligations and prohibitions. The construct *BIS_OP (name)* can be used for declaring a business operation. For example, the statements *BIS_OP(PO)*; *BIS_OP(ACC)*; *BIS_OP(PAY)* declare three business operations, purchase order, accept and payment, respectively. A contract *Business operation* declared as BIS_OP(name) is built upon the *typdef* Promela data type. Within its construct it defines the basic properties such as name, id, status, right, obligation and prohibition, and a set of operations (performed by macros) to manipulate the BIS_OP(name). Notice that the business operations declared with the construct BIS_OP(name) are declared globally to allow all of the running processes access to its data items. Next we show a macro that declares a business operation as Promela *typedef* datatype, along with, a business event of Promela *mtype* datatype.

```
==================================================
*            BIS_OP abstract data type          *
==================================================
typedef BIS_OP
{
 byte name;
 byte role_pl;
 bool right;
 bool oblig;
 bool prohib;
 bool executed;
 byte id;
 byte status;
}
==============================================
*             BIS_OP(name) macro           *
==============================================
#define BIS_OP(name) BIS_OP name##_##bo; \
mtype= { name }
```

4.3.1.3 Right

A *right* is 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. For example, in a Buyer-Seller scenario, the right to submit a purchase order is a right with no deadline. The rights set is all the rights granted to a role player at a given time. Rights are dynamic, they can be added and removed before the business transaction is concluded. The following macros can be used to set, reset or inquire a right property of business operation. It can be noted that macros SET_R(bo,r), SET_O(bo,o) and SET_P(bo,p) use Promela *assertion* method to assert that, updates in the ROP set during the contract execution do not cause contract–independent conflicts, that we discussed in Chapter 3, Section 3.3.1.

```
====================================
*    SET_R(bo,r) inline and macro  *
====================================
 inline SET_R(bo,r)
 {
   bo.right=r;
   /*Assert for contract-independent conflicts*/
   assert(!(bo.right==1 && bo.oblig==1));
   assert(!(bo.right==1 && bo.prohib==1));
 }
 ======================
 *        Set Right    *
 *         macro       *
 ======================
 #define SET_R(name,r) \
 SET_R(name##_##bo,r)

 =====================
 *    Inquire Right  *
 *       macro       *
 =====================
 #define IS_R(name, rp) \
 name##_##bo.right == 1 && name##_##bo.role_pl == rp
```

4.3.1.4 Obligation

An obligation is a business operation that a role player must execute, or face the penalty of being sanctioned. An example of an obligation is the obligation to pay an invoice within seven days that is imposed on the buyer after the seller sends an invoice. The obligations set is the set of all the obligations granted to a role player. Obligations are dynamic, they can be added and removed as the business exchange progresses. The following macros can be used to set, reset or inquire an obligation property of business operation.

```
/********************************/
/* SET_O(bo,o) inline and macro */
/********************************/
 inline SET_O(bo,o)
 {
   bo.oblig=o;
   /*Assert for contract-independent conflicts*/
   assert(!(bo.oblig ==1 && bo.prohib ==1));
   assert(!(bo.oblig ==1 && bo.right ==1));
 }
 #define SET_O(name,o) \
 SET_O(name##_##bo,o)

 =======================
 * Inquire Obligation  *
 *       macro         *
 =======================
 #define IS_O(name, rp) \
 name##_##bo.oblig == 1 && name##_##bo.role_pl == rp
```

4.3.1.5 Prohibition

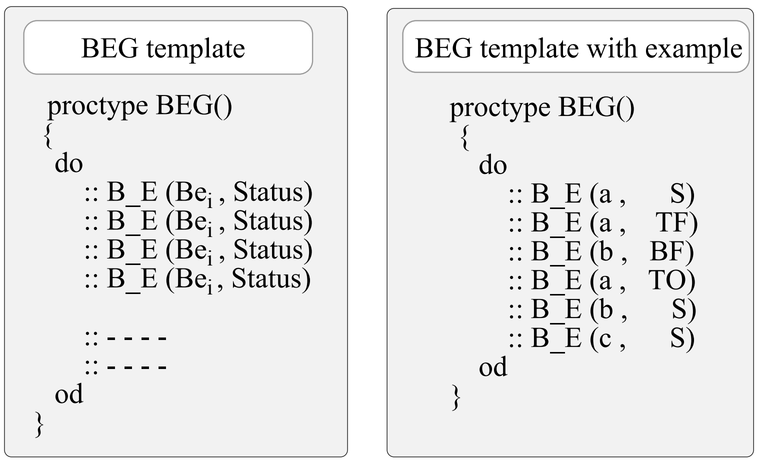
A prohibition is a business operation that a role player must not execute, or face the a penalty of being sanctioned. An example of a prohibition is that the contract forbids the Seller from canceling a purchase order. The prohibitions set is the set of all the prohibitions granted to a role player. Prohibitions are dynamic: they can be added and removed as the business exchange progresses. The following macros can be used to set, reset or inquire a prohibition property of business operation.

```
=======================================
*      SET_P(bo,p) inline and macro   *
=======================================
 inline SET_P(bo,p)
 {
   bo.prohib=p;

   /*Assert for contract-independent conflicts*/
   assert(!(bo.prohib ==1 && bo.right ==1));
   assert(!(bo.prohib ==1 && bo.oblig ==1));
 }
 #define SET_P(name,p) \
 SET_P(name##_##bo,p)

 =======================
 * Inquire Prohibition *
 *       macro         *
 =======================
 #define IS_P(name, rp) \
 name##_##bo.prohib == 1 && name##_##bo.role_pl == rp
```

4.3.1.6 Business events

The business event is a message carrying information about something happening during the business exchange. In the CB2B system, events model the execution of business operations which take place over a given time interval. In our extension, these are declared within the *BIS_OP*(*name*) macro. Once a BIS_OP is declared, a business event with the same name is declared as well. During the execution of a contract model, the business events are generated and sent by the BEG to the CRM. The BEG and CRM are implemented as a standard PROMELA processes. The structure and functionality of the BEG is contract independent. Thus we have designed the BEG template shown below that can be reused after parametrisation. Figure 4.5 describes a cyclic construct of the BEG process. This construct is used by CB2B model for generating events from the business event generator process (BEG). It provides a powerful way of generating business events nondeterministically. Thus, it precisely models the generation of business events as it takes place in actual B2B interactions.

This construct executes repeatedly. With each execution, it nondeterministically selects one out of the N (four in this example) business event within the do–od block. The events


BEG template
BEG template with example

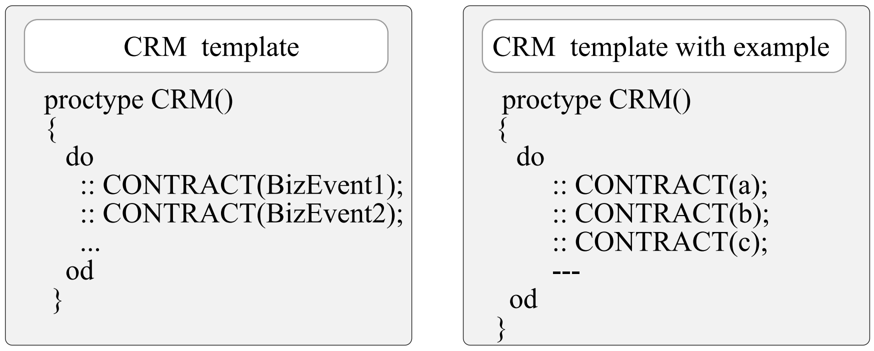
```
proctype BEG()
{
 do
    :: B_E (Be_i , Status)
    :: B_E (Be_i , Status)
    :: B_E (Be_i , Status)
    :: B_E (Be_i , Status)

    :: - - - -
    :: - - - -
 od
}
```

```
proctype BEG()
 {
  do
    :: B_E (a ,     S)
    :: B_E (a ,    TF)
    :: B_E (b ,    BF)
    :: B_E (a ,    TO)
    :: B_E (b ,     S)
    :: B_E (c ,     S)
  od
}
```

Figure 4.5: Business event generator template

represent the execution of business operations a, b, and c and their execution outcomes. For instance, the event B_E(a, S) represents the execution of operation 'a' with a success (S) outcome. Similarly B_E(a, TF) represents the execution of a but with a technical failure (TF) outcome. The event B_E(c,BF) models the execution of 'c' with a business failure (BF) outcome. The business event generator construct is programmed to generate any combination of the business events which correspond to business operations characterised as a contract compliant business events or non-contract compliant business events. However, by default it only generates the contract compliant business events. From the point of view of model checking, a particularly important case is the generation of sequences of events corresponding only to the execution of contract compliant operations. This considerably reduces the size of the state space for exploration, yet still enables checking that rules are responding correctly to contract compliant business operations (by not flagging them as non-contract compliant). A recommended way of model checking would be first to verify the rules using the restricted state space of contract compliant operations and to remove uncovered errors. Then the state space of exploration can be extended by releasing the restriction on BEG to generate all combinations of business events.

All the business event generated by the BEG are handled by the the Contract Rule Manager (CRM) process. The CRM is responsible for calling individual rules for executions as dictated by the arrival of business events. Like the BEG, the CRM is implemented as a conventional PROMELA process. Its structure and functionality is contract independent. Thus we have designed the CRM template shown below that can be reused after parametrisation.


CRM template
CRM template with example

```
proctype CRM()
{
  do
    :: CONTRACT(BizEvent1);
    :: CONTRACT(BizEvent2);
    ...
  od
}
```

```
proctype CRM()
{
  do
      :: CONTRACT(a);
      :: CONTRACT(b);
      :: CONTRACT(c);
      ---
  od
}
```

Figure 4.6: Contract rule manager template

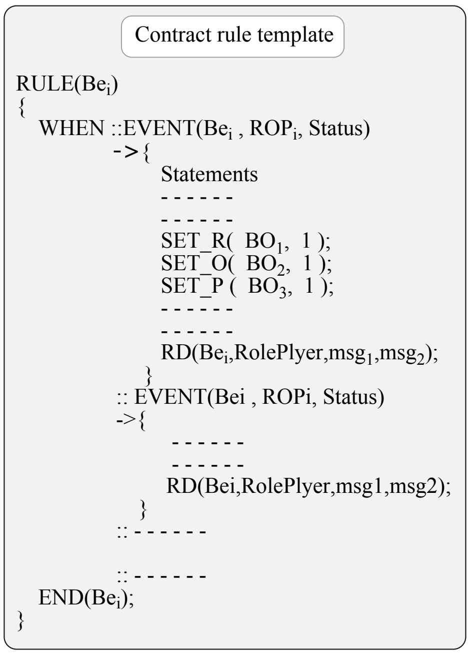
BizEvent1 and BizEvent2 are names of business events received from the BEG. As many events as needed can be included in the do–od block. For example, the CRM template can be parameterised as follows to tune it to a contract that involves four operations: a, b, c.

4.3.2 Contract rule template

Figure 4.7 shows the rule template of CB2B model. Since in our model rules and events are in one to one correspondence, Be$_i$ is the name of the rule and of the event expected to trigger it. Among other parameters, an event always indicate the status of the execution which is *success (S)*, *Business Failure (BF)* or *Technical Failure (TF)*.

To account for different execution outcomes of the operation represented by the event, the body of the rule (statements within the WHEN....END(Bei) keywords) is composed of one or more (normally three) independently executable blocks. Each of them is guarded by an EVENT(Be$_i$, ROP$_i$, Status) that evaluates to either *true* or *false*. When the event Be$_i$ triggers the rule, one or none of the blocks executes. In either case the rule produces a Rule Decision (RD) that is notified to the contract rule manager process, indicating that Be$_i$ was declared contract compliant or non contract compliant by the rule.

When none of the executable blocks executes, the rule decision is NCC (Non Contract Compliant). However, when one of the blocks is executed, the rule decision is CC (Con-


Contract rule template

```
RULE(Be_i)
{
  WHEN ::EVENT(Be_i , ROP_i, Status)
            ->{
                 Statements
                 - - - - - -
                 - - - - - -
                 SET_R(  BO_1,  1 );
                 SET_O(  BO_2,  1 );
                 SET_P (  BO_3,  1 );
                 - - - - - -
                 - - - - - -
                 RD(Be_i,RolePlyer,msg_1,msg_2);
               }
            :: EVENT(Bei , ROPi, Status)
            ->{
                  - - - - - -
                  - - - - - -
                 RD(Bei,RolePlyer,msg1,msg2);
              }
            :: - - - - - -

            :: - - - - - -
  END(Be_i);
}
```

Figure 4.7: CB2B Rule template

tract Compliant), unless other conditions inside the executable block are considered. For example, when an event Be$_i$ arrives and its corresponding operation is currently in ROP_i and its status is *success* then the execution block guarded by EVENT(Be$_i$, ROP_i, success) executes; however, if the operation that corresponds to the event is currently in the ROP_i, but the status of the event is *BusinessFailure*, the block guarded by EVENT(Be_i, ROP_i, BusinessFailure) executes instead. As it will be seen in Chapter 5, the ECA rule template and the set of operations that have been defined on the BIS_OP(Be$_i$) abstract data type (refer to Table 4.1 and Table 4.2) can be used to implement large class of business contracts. Thus, various business contracts, especially those can be represented by EROP language(see Section 2.2.6), can be relatively easy modelled using our CB2B model notation.

4.3.3 Manipulating contract status

A contract designer composes the set of ECA rules to manipulates the contact status through the set of macros discussed here in this section. For full implementations of the macros introduced here refer to Appendix A.

Initiation of rights, obligations and prohibitions: Each operation is associated (expected to be executed by) to a role player as a right, obligation or prohibition. The initial state of the rights, obligations and prohibitions imposed on a given operation is declared with the help of the construct *INIT(BizOper, RolePlayer, R,O,P)*. *BizOper* is the name of operation. *RolePlayer* is the name of the role player associated to the operation. *R,O,P* are bits (set to either *0* or *1*) that represent, a right, obligation or prohibition, respectively. For example, *INIT(PO, BUYER, 1,0,0)* declares that the buyer has the right to execute operation *PO*; similarly, *INIT(OFFER, SELLER, 0,0,1)* declares tht the seller is prohibited from executing the OFFER operation.

Grant/revoke rights, obligations and prohibitions: As the contractual interaction progresses, rights, obligations and prohibitions are granted (SET) and revoked (RESET). The operations *SET_R(BizOper, boolvalue)*, *SET_O (BizOper, boolvalue)*, *SET_P(BizOper, boolvalue)* can be used, respectively, for setting/ resetting rights, obligations and prohibitions. *BizOper* is the name of a business operations whereas *boolvalue* is *1* or *0*. For example, the statement *SET_R(PO, 0);* revokes the right to execute operation *PO*. Similarly, the statement *SET_O(PAY, 1);* imposes and obligation to execute a *PAY* operation. Subsequently, when the obligation is fulfilled, the designer can include the statement *SET_O(PAY, 0);* to remove the obligation to pay.

Inquire status of rights, obligations and prohibitions: The operations *IS_R (BizOper, RolePlayer)*, *IS_O(BizOper, RolePlayer)*, *IS_P(BizOper, RolePlayer)* return either *T* of *F* and can be used for inquring about the status rights, obligations and prohibitions. *BIZOper* is the business operation of interest whereas *RolePlayer* is the role player associated to the right, obligation or prohibition. For example, to enquire if the buyer has currently the right to execute a purchase order operation we can use *IS_R(PO,BUYER)* which will return either T if he has or F if he has not. Similarly, the execution of the statement *IS_O(REFUND, SELLER)* will return T if the seller is currently obliged to execute a *REFUND* operation.

Set/Reset execution status of operations: The execution status of operations is set to F in the initial state, by default. Subsequently, we can use the operation *SET_X(BizOper, RolePlayer)* to set the status of the operation *BizOper* to T. For example, the statement *SET_X(PO, BUYER);* can be used to register that the buyer has executed the operation *PO*.

Enquire about execution of operations: The operations *IS_X(BizOper, RolePlayer)* which returns either T of F can be used for inquiring if the operation *BizOper* has been executed at least once by the role player *RolePlayer*. For example, the statement *IS_X(PAY, BUYER)* will return F if the buyer has not executed yet the operation *PAY*; otherwise, it will return F.

4.4 CB2B model operations

Table 4.1 summarises the operations defined for the *BIS_OP* abstract data type. The parameter *boName* is used for the business operation name. There are 'SET' methods to grant/remove a right, obligation or prohibition and 'IS' methods to test whether a role player has a right, obligation or prohibition restriction with respect to a business operation. The SET_X method is used to record that the operation has been executed; that status can be checked by the IS_X method. These methods are useful when implementing rules for clauses such as 'send a reminder if the payment has not been made' or 'extend payment deadline if a technical failure has occurred for payment' (see [37] for more examples).

Name	**Description**
BIS_OP(Be_i)	Declares *business operations* of Promela type *typedef* with the fields: *name, id, RolePlayer, right, obligation, prohibition, execution status.* The rule field *name* takes the same value of the *business event* name (Be_i). Ex. - BIS_OP (offer)
SET_R(Be_i, 1)	Gives (removes, when second parameter is 0) right to execute the business operation of Be_i. Ex.- SET_R(offer, 1)
SET_O(Be_i, 1)	Assigns (removes, when second parameter is 0) obligation to execute the business operation of Be_i. Ex.- SET_O (pay, 1)
SET_P(Be_i, 1)	Sets (removes, when second parameter is 0) prohibition to execute the business operation of Be_i. Ex.- SET_P(cancel, 1)
IS_R(Be_i, RolePlayer)	Returns 1 if RolePlayer has permission to excute the business operation of Be_i, 0 otherwise. Ex.- IS_R(offer, Seller)
IS_O(Be_i, RolePlayer)	Returns 1 if RolePlayer is obliged to excute the business operation of Be_i, 0 otherwise. Ex.- IS_O (pay, Seller)
IS_P(Be_i, RolePlayer)	Returns 1 if RolePlayer is prohibited to execute the business operation of Be_i, 0 otherwise. Ex.- IS_P (cancel, Buyer)
SET_X(Be_i, RolePlayer)	Sets execution status of the business operation of Be_i to 1. The default value is 0. Ex.- SET_X (offer, Seller) means offer has been executed by the Seller.
IS_X(Be_i, RolePlayer)	Returns 1 if Be_i has been executed by the RolePlyaer.
INIT(Be_i, RolePlayer,R,O,P)	Initialise BIS_OP(Be_i) with (R)ight, (O)bligation or (P)rohibition and add it to a RolePlayer. Ex.- INIT(offer, Seller,1,0,0)- The Seller is given the *right* to execute an offer.

Table 4.1: BIS_OP operations list.

Table 4.2 shows the operations and control structures that have been implemented as syntactic sugar; these supplement Promela constructs for purposes such as statement sequencing, atomic sequencing, concurrent execution, case selection, repetition and unconditional jumps. With this high level notation built upon the Promela basic data types, the business operations defined in the contract model are declared globally with BIS_OP(Be_i). Then an expression such as INIT(Be_i, $\mathrm{RolePlayer}_i$,0,1,0) associates the Be_i with the $\mathrm{RolePlayer}_i$, the last three bits denote that a business operation Be_i can be initialised as a right, obligation or prohibition respectively (since the second bit is set to one, this example is initialised with an *obligation*). Similarly, an operation such as SET_O(Be_i,0) discharges the obligation imposed on $\mathrm{RolePlayer}_i$. Finally, an obligation on Be_i associated with the $\mathrm{RolePlayer}_i$ denotes that it is obligatory for the $\mathrm{RolePlayer}_i$ bearing this obligation to perform the action Be_i. In the

Appendix A we list a full implementation of the macros we developed in this work.

Name	**Description**
CONTRACT(Be_i)	Includes the contract Rule_i into the contract rule manager (CRM) based on the received business event Be_i. CONTRACT(Be_i) blocks until the the business event Be_i is generated. Ex.- CONTRACT(offer) becomes executable if the business event *offer* is received, and if executed, then the contract rule *Rule(offer)* is loaded to handle the event *offer*.
EVENT(Be_i, ROP_i, Status)	Blocks until the business event (Be_i) occurs, the ROP entity (ROP_i) (*right*, *obligation* or *prohibition*) and its execution status are evaluated to true. Ex. EVENT(Refund, IS_O(Refund, CLIENT), SC(Refund)).
B_E($\mathrm{RolePlyer}_i$, Be_i, Status)	Sends *business event* Be_i of a business operation belong to $\mathrm{RolePlyer}_i$ with *status* (S, BF, TF or TO) to the Contract Rule Manager(CRM). Ex.- B_E(Seller, offer, S).
SC(Be_i), BF(Be_i), TF(Be_i), TO(Be_i)	Returns 1 if a business event Be_i submitted is flagged with success (S), business failure (BF), technical failure (TF) or timeout (TO) execution status respectively, 0 otherwise. Ex.- SC(Refund), BF(Refund), TF(Refund), TO(Refund).
RD(Be_i, RolePlayer, m1, m2)	The *(R)ule (D)ecision* (RD) notifies about the outcome of the rule execution. The parameter *m1* could be a CCR,CCO or CCP(that is a *(C)ontract (C)ompliant (R)ight, (O)bligation or (P)rohibition*, respectively) or alternatively a *NCC ((N)ot (C)ontract (C)ompliant)*. The parameter *m2* is either *CON* or *CND* (continue contract or contract ended respectively). *Ex.- RD(offer, Seller, CCO, CON).*

SYN(Be_i){
 Statements
 − − −−
 − − −−
 }
 NYS(Be_i)
	Used within the contract rules blocks to synchronise with the business operation of Be_i. The statements within SYN..NYS body are executed when the history of the execution status of Be_i is evaluated to true.

Table 4.2: CB2B operations list.

4.5 Contract model checking with CB2B model

The CB2B model is supported by the verification framework shown in Figure 4.8. It can be initialized with a set of ECA rules, where the rules have been coded using our extended Promela language. Three assumptions are made; firstly that the contract has been negotiated by the contracting parties and drawn up in English (or other natural) language. Secondly, it is assumed that the designer manually converts the clauses of the English contract into ECA rules written in extended Promela language which are executed by the processes and channels of the CB2B model. In parallel, the designer manually prepares a list of contract correctness requirements deduced from the contract clauses into LTL formulae. Finally, the designer inputs the CB2B model together with the rules and the LTL formulae into the Spin verifier and runs it to output verification results.

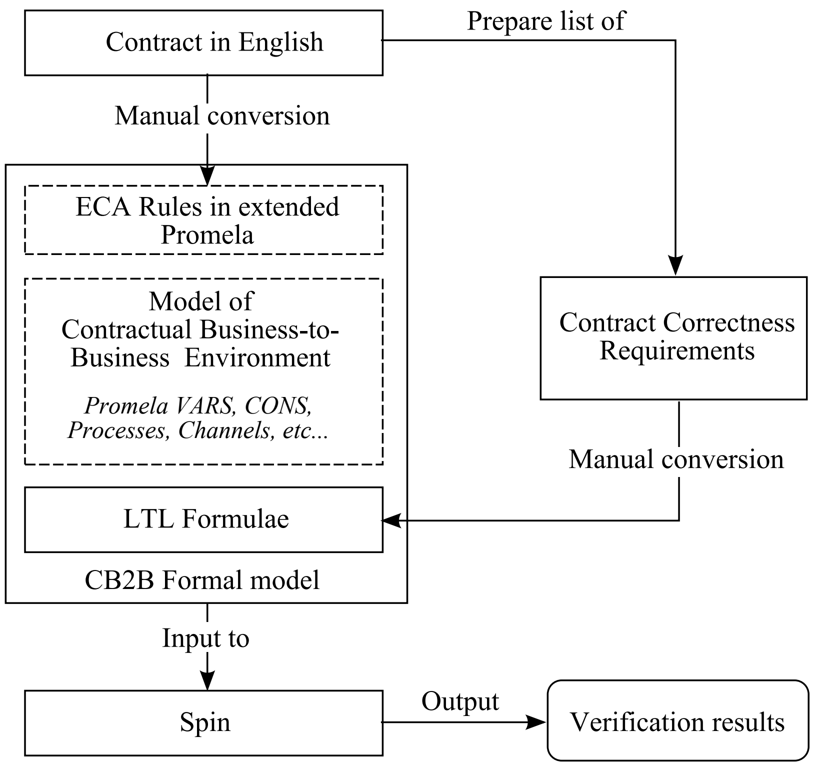
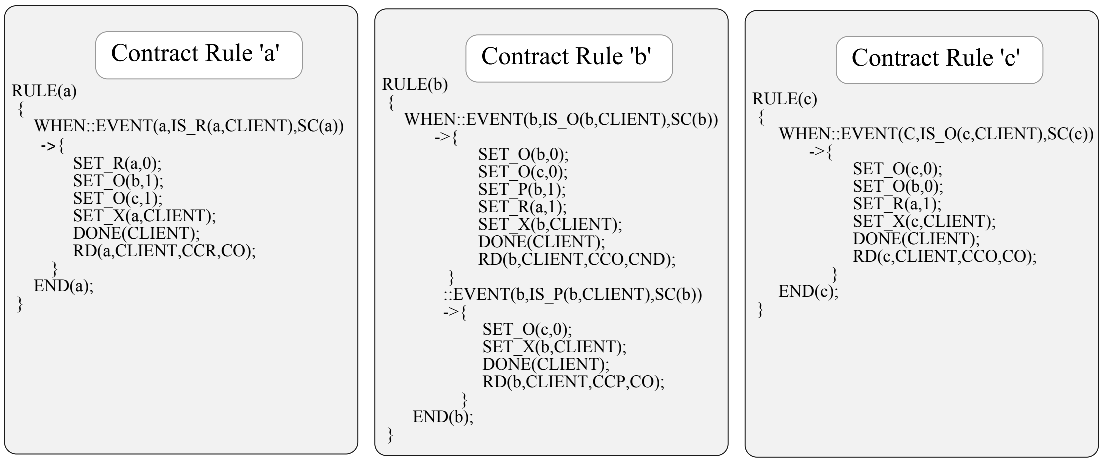

Contract in English
Prepare list of
Manual conversion
ECA Rules in extended Promela
Model of Contractual Business-to-Business Environment
Promela VARS, CONS, Processes, Channels, etc...
Contract Correctness Requirements
Manual conversion
LTL Formulae
CB2B Formal model
Input to
Spin
Output
Verification results

Figure 4.8: Contract model checking framework.

It is worth mentioning that electronic contracts are likely to experience modifications, for example after re–negotiations, during their life-cycle. However, modifications of the contract usually lead to several problems in terms of conflicts and deadlock/livelock freedom. A reliable solution to these frequent updates of the contract clauses consists of the application of model checking techniques in order to verify if specific properties of the monitorable contract are preserved by any changes which may be caused by introducing new clauses. It is advisable to expose each new version of the contract to the procedure described in Figure 4.8, possibly with some short-cuts taken depending on the severity of the updates. The next subsection revisiting the contract example discussed throughout this chapter and model it with the notation discussed above, and then verify it with the CB2B verification framework.

4.5.1 Contract example re-visited

Revisiting the contract example discussed earlier in section 4.2.3 helps to show how the different elements of the contract such as rights, obligations and prohibitions as well as the set of contract rules, will be specified in extended Promela (see Figure 4.9 for the contract rules) and then verified using the CB2B model. Recall that the contract fragment is 'there is an obligation to choose between doing 'b' or 'c' after 'a', and a prohibition on doing 'b' if 'b' has been performed. Applying our verification framework to this example reveals a contract defect that is not easily detected in the original text or during the conversion into ECA rules.


Contract Rule 'a'
Contract Rule 'b'
Contract Rule 'c'

```
RULE(a)
 {
   WHEN::EVENT(a,IS_R(a,CLIENT),SC(a))
    ->{
        SET_R(a,0);
        SET_O(b,1);
        SET_O(c,1);
        SET_X(a,CLIENT);
        DONE(CLIENT);
        RD(a,CLIENT,CCR,CO);
      }
   END(a);
 }
```

```
RULE(b)
 {
   WHEN::EVENT(b,IS_O(b,CLIENT),SC(b))
       ->{
            SET_O(b,0);
            SET_O(c,0);
            SET_P(b,1);
            SET_R(a,1);
            SET_X(b,CLIENT);
            DONE(CLIENT);
            RD(b,CLIENT,CCO,CND);
         }
        ::EVENT(b,IS_P(b,CLIENT),SC(b))
        ->{
            SET_O(c,0);
            SET_X(b,CLIENT);
            DONE(CLIENT);
            RD(b,CLIENT,CCP,CO);
          }
    END(b);
 }
```

```
RULE(c)
 {
   WHEN::EVENT(C,IS_O(c,CLIENT),SC(c))
       ->{
            SET_O(c,0);
            SET_O(b,0);
            SET_R(a,1);
            SET_X(c,CLIENT);
            DONE(CLIENT);
            RD(c,CLIENT,CCO,CO);
          }
   END(c);
 }
```

Figure 4.9: Contract rules in extended Promela

We can see that these rules closely resemble the pseudo-code discussed earlier (Section 4.2.3). A comparison of these rules against their equivalent in standard Promela would reveal that these rules are far more readable, compact and intuitively clear. The reason for this is that they take advantage of our new contract concepts. For instance, for Rule(c), a query to check if a role player is obliged to execute operation c can be coded intuitively as a single line: IS_O(c). Likewise in Rule(b), a single line, SET P(b, 1), is enough to prohibit a role player from executing operation b. Each rule ends with a decision such as RD(a,CLIENT,CCR,CO) sent to the event generator indicating whether the operation is a contract compliant right, obligation or prohibition (CCR, CCO or CCP respectively) or noncontract compliant (NCC); where CO is short for continue, indicating that event generation should continue.

The complete verification process of this contract with the *CB2B* verification framework follows the next steps below:

1. Extract the contract entities from the text and re-write the contract as a set of ECA rules using the CB2B rule template (Figure 4.9).
2. Prepare the list of requirements as LTL formulae (e.g. P1 below).
3. Run Spin model checker for exhaustive verification and observe the output - in case of violations the Spin model checker returns a counterexample.

```
ltl P1 {[]((IS_X(b,CLIENT)-> [](IS_P(b,CLIENT))))}
```

That is, always after its first execution, a role player client is prohibited to perform the operation '*b*' again forever. Recall that the model can detect the contract independent-conflicts discussed in 3.2.1 automatically. Thus, writing properties such as P2 below to verify that it is always not possible to be simultaneously obliged on and prohibited to execute the business operation 'b' is redundant:

```
ltl P2 {[](not( IS_O(b,CLIENT) && IS_P(b,CLIENT) ))}
```

In fact, as discussed earlier, properties of this type such as P2 can be detected through assertions that are implemented as part of the CB2B operations SET_R(), SET_O() and SET_P(). Thus, Spin would automatically examine them after each operation execution, and in case of violation is detected the counterexample is returned.

The verification of property P1 passed the test; this means that '*b*' will be prohibited after its first execution forever. It is important to test such properties of the contract before its deployment in a contract monitoring service. This is because the real implementation of a rule such as '*b*' may impose sanctions, termination or legal action whenever the contract rule is violated, which in turn may badly affect the business exchange and lead to disputes if the decisions made by monitoring service were not accurate.

The verification of the contract independent properties, or for properties such as P2, did not pass the test. Spin complained and returned a counterexample after the execution $trace : a \rightarrow b \rightarrow a$. During the contract compliance checking, the ROP set is manipulated in the rule for 'a': checking that there is a right to perform 'a' (IS_R(a) returns True), and, if so, that right is now removed since 'a' has been performed and an obligation to perform 'b' or 'c' is inserted, only one of them will be chosen non–deterministically. In the rules for 'b' and 'c', the right to perform 'a' is inserted again. At first glance, these rules seem to be an accurate representation of the contract. However, it turns out that the rules do not meet this requirement formalised in P1 which informally states that there should be no simultaneous obligation and prohibition on executing the operation 'b'.

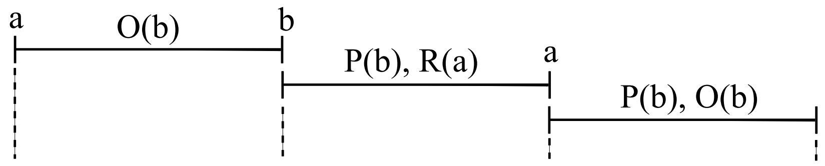
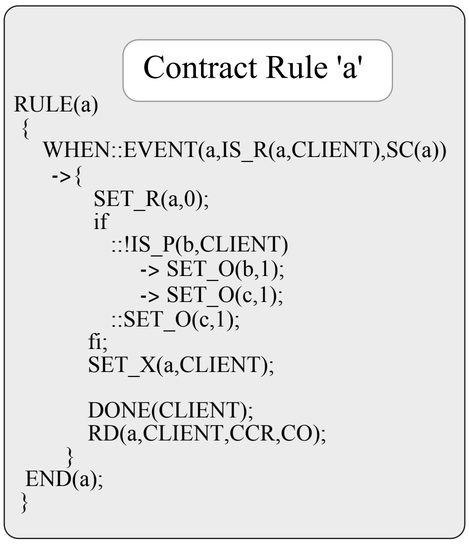


Figure 4.10: Inconsistent assignment of prohibition and obligation

Figure 4.10 shows the membership of the ROP set for one particular execution: operations 'a' followed by 'b' followed by 'a', and in the second execution of 'a', 'b' is chosen again; now there is obligation as well as prohibition to perform 'b'. The corrected version is shown in Figure 4.11 where the obligation to perform operation b is enabled only when there is no prohibition on it.

An advantage of the CB2B verification framework is that the contract is specified as a set of ECA rules. As a result, once a contract has been verified, the ECA rules


Contract Rule 'a'

```
RULE(a)
 {
   WHEN::EVENT(a,IS_R(a,CLIENT),SC(a))
    ->{
         SET_R(a,0);
         if
           ::!IS_P(b,CLIENT)
               -> SET_O(b,1);
               -> SET_O(c,1);
           ::SET_O(c,1);
        fi;
        SET_X(a,CLIENT);

        DONE(CLIENT);
        RD(a,CLIENT,CCR,CO);
      }
 END(a);
 }
```

Figure 4.11: Correct version of contract rule 'a'

from the CB2B formal model can be translated relatively easily into their executable counterpart, such as EROP rules. We now therefore have a systematic way of generating a machine interpretable contract, although it has not yet been implemented. Another key point is that the rules are specified in a separate document in a high level notation based on the contract entities to simplify the close collaboration required between domain experts such as software engineers and business practitioners through the intuitively understandable format of the contract's ECA rules. In fact, decision to adopt the verification tools in the business world should consider these usability issues.

4.5.1.1 Complete code of the contract example

```
/* Import required files */
#include "setting.h"
#include "BizOperation.h"

/* import contract rules*/
#include "contract_rules.h"

/*Declare global variables*/
byte RefundAmount;
```

```
RolePlayer(CLIENT);
RuleMessage(S,BF,TF,TO);
BIS_OP(START);
BIS_OP(a);
BIS_OP(b);
BIS_OP(c);

/* Define LTL formulae */
/*ltl X {[]((IS_X(b,CLIENT)-> [](IS_P(b,CLIENT))))}*/

/* Business Event Generator process (BEG)*/
proctype BEG()
{
  BEGIN_INIT:
  {
     INIT(a,CLIENT,1,0,0);
     INIT(c,CLIENT,0,0,0);
     INIT(b,CLIENT,0,0,0);
  }
  END_INIT:

  /* GENERATING BUSINESS EVENTS */
  do
     :: B_E(CLIENT,a,S);
     :: B_E(CLIENT,b,S);
     :: B_E(CLIENT,c,S);
  od;
}

/* CONTRACT RULE MANAGER process (CRM)*/
proctype CRM()
{
  do
     :: CONTRACT(a);
     :: CONTRACT(b);
     :: CONTRACT(c);
  od;
}
/* Initialise model processes */
init
{
  atomic
  {
   run BEG();
   run CRM();
  }
}
```

4.5.1.2 Rule–base of the contract example

The contract ECA rules saved in a separate file (contract_rules.h), and imported to the CB2B model using the command *#include* "contract_rules.h" as shown in the previous section.

```
 RULE(a)
 {
   WHEN::EVENT(a,IS_R(a,CLIENT),SC(a))
       ->{
          SET_R(a,0);
          if
            ::!IS_P(b,CLIENT)
              -> SET_O(b,1);
              -> SET_O(c,1);
            ::SET_O(c,1);
          fi;
          SET_X(a,CLIENT);
          DONE(CLIENT);
          RD(a,CLIENT,CCR,CO);
         }
END(a);
}
RULE(b)
{
    WHEN::EVENT(b,IS_O(b,CLIENT),SC(b))
        ->{
           SET_O(b,0);
           SET_O(c,0);
           SET_P(b,1);
           SET_R(a,1);

           SET_X(b,CLIENT);
           DONE(CLIENT);
           RD(b,CLIENT,CCO,CND);
           }
        ::EVENT(b,IS_P(b,CLIENT),SC(b))
    ->{
           SET_O(c,0);
           SET_X(b,CLIENT);
           DONE(CLIENT);
           RD(b,CLIENT,CCP,CO);
          }
END(b);
}
  RULE(c)
  {
      printf("Cancel business operation");
      WHEN::EVENT(C,IS_O(c,CLIENT),SC(c))
      ->{
         SET_O(c,0);
         SET_O(b,0);
         SET_R(a,1);
         SET_X(c,CLIENT);
         DONE(CLIENT);
         RD(c,CLIENT,CCO,CO);
        }
      END(CANCEL);
  }
```

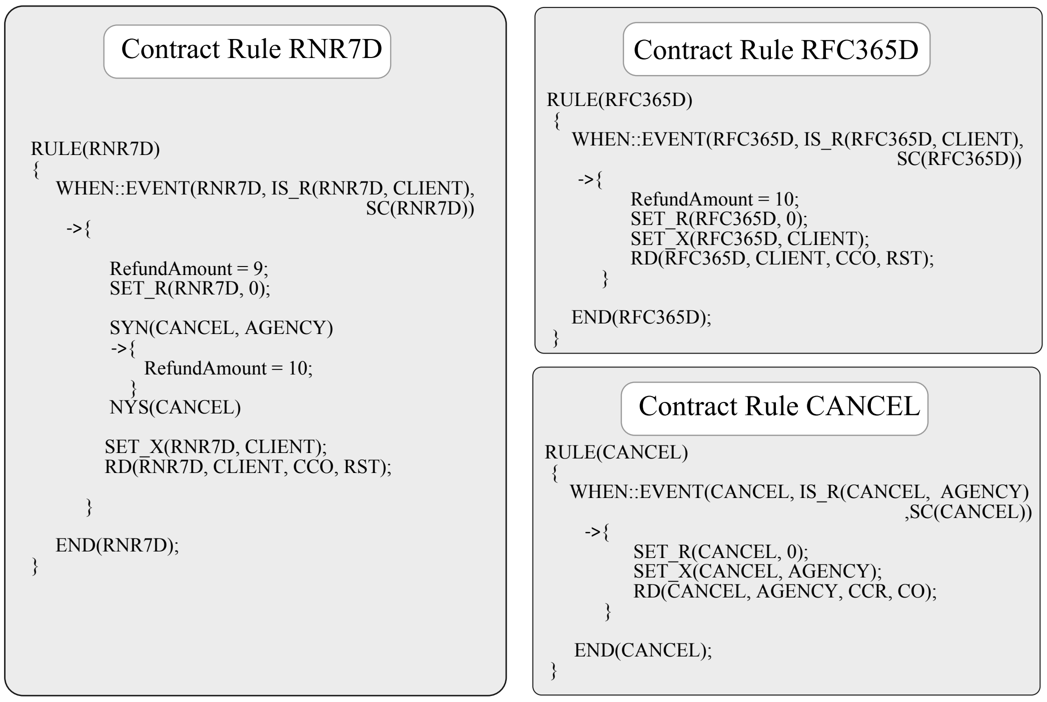
In this contract example we have shown the full implementation of the CB2B contract model. We have shown the global declaration part, where the contract global variables and rules are accessed and manipulated by CB2B processes. The code listing above also shows how the model imports the contract rules (from the contract_rules.h), the set of macros extensions (from the BizOperation.h) and setting.h (model channels, messages etc.). The extension .h given to the files is not compulsory, Spin will accept other extensions such as .c .pml .txt etc.

4.5.2 Contract example with priority rules conflict

A well known class of inconsistency that impact contracts is conflicts at level that – if not addressed – result in conflicts at rule implementation level. A conflict happens when two or more contradictory actions (operations) appear to be in force simultaneously. Consider the following example (discussed earlier in Section 3.2.1):

- Rule A - if customer returns the purchased e-ticket for any reason, within 7 days, then the purchase amount, minus a 10% penalty fee, will be refunded.
- Rule B - if customer returns the purchased e-ticket because the flight was canceled by the seller (travel agent), before the due date (up to one year), then the full purchase amount will be refunded.

A conflict would occur if the ticket is canceled by the seller in the overlapping period between rule A and rule B, within 7 days of the ticket issuing date, so both rules are applicable.



Contract Rule RNR7D

```
RULE(RNR7D)
{
    WHEN::EVENT(RNR7D, IS_R(RNR7D, CLIENT),
                                    SC(RNR7D))
      ->{

            RefundAmount = 9;
            SET_R(RNR7D, 0);

            SYN(CANCEL, AGENCY)
            ->{
                RefundAmount = 10;
              }
            NYS(CANCEL)

            SET_X(RNR7D, CLIENT);
            RD(RNR7D, CLIENT, CCO, RST);

        }

    END(RNR7D);
}
```

Contract Rule RFC365D

```
RULE(RFC365D)
{
    WHEN::EVENT(RFC365D, IS_R(RFC365D, CLIENT),
                                    SC(RFC365D))
      ->{
            RefundAmount = 10;
            SET_R(RFC365D, 0);
            SET_X(RFC365D, CLIENT);
            RD(RFC365D, CLIENT, CCO, RST);
        }

    END(RFC365D);
}
```

Contract Rule CANCEL

```
RULE(CANCEL)
{
    WHEN::EVENT(CANCEL, IS_R(CANCEL,  AGENCY)
                                    ,SC(CANCEL))
      ->{
            SET_R(CANCEL, 0);
            SET_X(CANCEL, AGENCY);
            RD(CANCEL, AGENCY, CCR, CO);
        }

    END(CANCEL);
}
```

Figure 4.12: E-ticket refund contract rules

This contract needs to consider both the client's refund and the travel agency's cancellation business operations. Two contract rules are defined to handle the refund operations. Rule(RNR7D) for refund for no reason (RNR) within seven days (7D), this could happen if the buyer changed his mind about the flight, and (RFC365D) for refund for canceled e-ticket (RFC) within one year (365D), this normally happens when the Seller cancels the flight. Furthermore, Rule(CANCEL) is added for handling the cancellation business event (CANCEL) from the travel agent. The contract is initialised with right to the client to perform RNR7D and RFC365D, as well as a right to the travel agent to cancel the ticket at any time. Figure 4.12 above shows rule implementation to this contract using extended Promela. Each rule can be triggered by an event from the set of business events (RNR7D, RFC365D and CANCEL). By default, the set of the contract rules respond only to the contract compliant business events. In response to such events, the contract status is updated; for example rights, obligation or prohibitions may be applied, or permissible operations might be prohibited. Note that the block SYN, NYS in Rule(RNR7D) is to synchronise with cancel business operation (hence the name SYN() is used). Basically, SYN(CANCEL, AGENCY) checks for the execution history of cancel business operation. If it has been found executed, the rule would guarantee full refund to the client. By default, the client would be penalised 10% as a cancelation for no reason penalty. Figure 4.13, shows different possible timelines of the execution of the refund operation.

Figures 4.13(a) and 4.13(b) respectively show that the business event RNR7D is permitted within 7 Days, and the business event RFC365D is permitted within the whole year. In both timelines no cancellation business events have been witnessed. Figure 4.13(c), shows that the seller has the right to cancel the flight ticket for the whole year, whereas the grey arrows in a,b and c show different points in time in which the business events RNR7D, RFC365D and CANCEL may occur. Figure 4.13(d), shows possible scenario when both of the contract rules RNR7D and RFC365D can be executed within the first 7 after the commencement of the contract date. In such a situation, the buyer might be penalised if he return the flight ticket for no reason within 7 Days, however, the full refund amount must be returned as the seller has already canceled the flight.

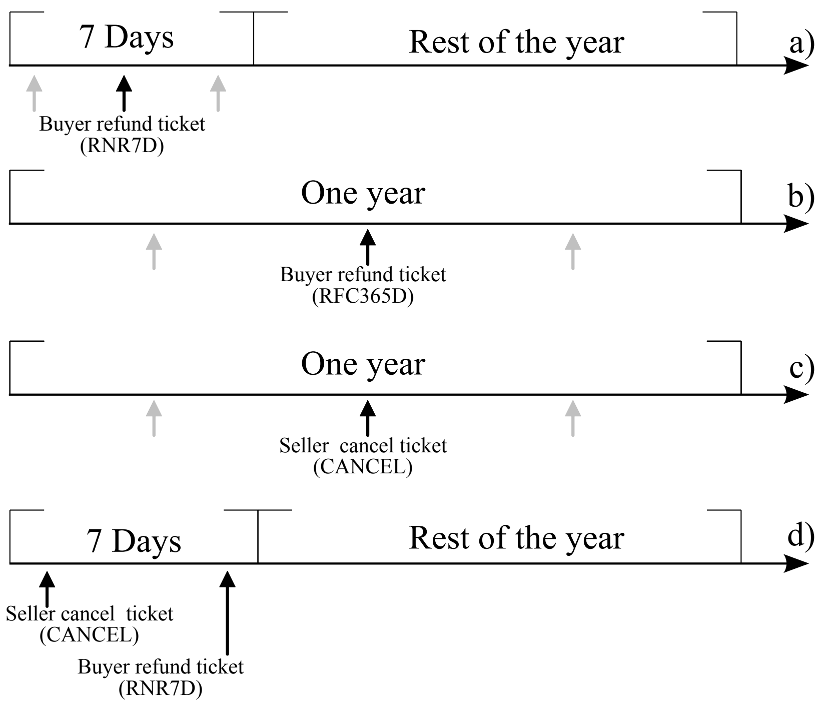


Figure 4.13: Priority rule conflict illustrated with timeline

In order to verify that the above contract rules consider the discussed possible conflict, Spin is first executed with its default settings with the feature for assertion violations detection is enabled to verify the contract-independent conflicts. Then for the contract-dependent conflicts, the following two formulae P1 and P2 can be verified. P1 verifies that whenever RNR7D and CANCEL are executed, the *RefundAmount* must be paid in full, whereas P2 verifies that whenever RNR7D is executed and ticket is not canceled, then the Client must be penalised.

```
ltl P1{[]((IS_X(RNR7D,CLIENT)&&IS_X(CANCEL,AGENCY))->
                                    (RefundAmount==10))}

ltl P2{[]((IS_X(RNR7D,CLIENT)&& not(IS_X(CANCEL,AGENCY)))->
                                    (RefundAmount==9))}
```

The current implementation of the contract does not complain about P1 and P2; the verification ended with no problems. This would ensure that there will be no case when a client refund the ticket within 7 Days for no reason while the ticket is has already been cancelled and refunded £9 instead of £10. Injecting the Rule(RNR7D) with an error, such as changing the refund amount to 10 or any other value, and verify for P2 will cause Spin to complain and return counterexample instantly after receiving the RNR7D business event and execute the body of Rule(RNR7D).

4.6 The use of CB2B model for testing

The basic idea behind model checker based testing is simple and elegant: construct a behaviour model of the system under test (SUT) and validate the behaviour using a model checker (e.g., use Promela language for constructing the model and verify using Spin, [27]). Such a validated model can then be used for generating executable test cases for the SUT; the model also acts as an oracle, since it also indicates the expected outputs the SUT should produce under given conditions. A principal challenge here is the construction of a model that is sufficiently small (abstract, simple) to enable, as far as possible, exhaustive checking (full validation) by the model checker; at the same time, the model should be realistic enough to be able to generate test cases that exercise the SUT. Different techniques have been proposed in order to force a model checker to create traces suitable as test cases. The traces are generated as counterexamples for property violations; such traces are known as *witness* traces, which serve to illustrate that a system property is satisfied. Model checker based testing techniques have received wide attention in the software engineering community see [25, 43, 46, 52].

4.6.1 Spin based test–case generation tool

A schematic view of test case synthesis tools such TGV [8] for the generation of test cases is shown in bold in Figure 4.14. Then, given a reference model, and some criterion, a set of test cases [54] is produced. The reference model is intended to represent the behaviour of the system under test. The role of the synthesis tool (SPIN) is to select test cases from behaviours of the system specification. Thus, a second input which is a criterion or test purpose is required. This criterion is aimed to precisely match the system functionalities to be tested. The output of the tool is a set of test cases describing the behaviours of the system under test along with verdicts associated with those behaviours.

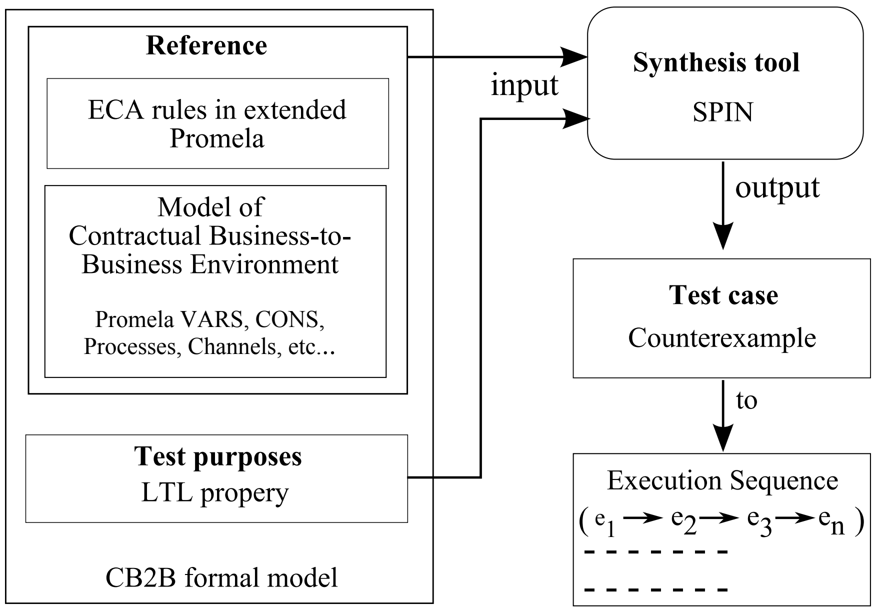


Figure 4.14: General structure of test case synthesis tool using CB2B model

4.6.2 Test–case generation steps

Generating test cases with the CB2B model and the general Spin model checker involves three steps. Firstly, an abstract *reference model* of the SUT is built. Then the *test purpose* concerning the property of interest such as when the SUT progresses from state S_i to state S_j after being presented with input e_i is formed as LTL formula. Thirdly, the LTL is negated and presented to the model checker (*the synthesis tool*) with the challenge to execute the abstract model to show that the negated LTL claim can be violated. As a result, the model checker then produces witnesses that include transitions from S_i to S_j when e_i is provided. As explained in [44], counterexamples (or witnesses) produced by model checkers contain abstract parameters that are meaningful to the abstract model but meaningless to the SUT. Thus they serve only as raw data to produce executable test cases that can be fed into the SUT to exercise a run. We note that a verified CB2B model (the reference) will by default generate sequences with events corresponding to the execution of contract–compliant operations only. However, the model can be tuned to generate sequences which include unknown and non–compliant business events; different categories of non–compliant business events have been discussed in Chapter 3. Once a test–case is generated, a test verdict of pass or fail is then associated with it, so it can be used to test the correctness of behaviours of the actual implementation of the contract rules.

4.6.3 Limitations of Spin counterexamples

Model–checker based approaches, for test–case generation, offer many advantages: They are fully automated and flexible. However, because model–checkers in general were not originally designed as test–case generation tools, there are many limitations for using them for this purpose. In this section we list the main limitation of using Spin for generating test–cases. We note that the list of limitations discussed below might be applicable to a large class of model checkers not only Spin.

- The enormous state space of finite–state models of practical software specifications often leads to the state explosion problem: Spin model checker might run out of memory or time before it can analyze the complete state space.
- Because Spin does a depth–first search (DFS) of the state–machine model, it produces very long counterexamples (which means very long test–cases). We note that Spin offers switch which finds the shortest counterexample, this might be used to find found a test sequence with the shortest possible length.

- A large percentage of the test-cases produced using Spin model checker are redundant. Various heuristics approaches suggested to tackle the redundancy of test–cases (see for example [15–17]).
- Finally, a major weakness of this approach is the reliance of this method on a manual translation of the specification to the model–checker input language, which requires some skill and ingenuity. In our case, we tried to tackle this problem by developing an intuitive high–level notation (by extending Promela, the input language of Spin model checker) to model the contract rules.

The B2B contractual interactions can give rise to highly complex execution patterns, and it is quite unrealistic to assume that these can be produced manually for testing purposes. Testing tool support is therefore would help at design time to validate the consistency of the contractual clauses and later, to produce test case validation sequences to test the correctness of the actual implementation. This Section has briefly illustrated the idea of using the CB2B model and Spin model checker to generate test–cases for electronic contracts. An earlier version of the CB2B model [2] has been used to demonstrate this tool for test–case generation while testing an electronic contract system [51].

4.7 Evaluation of CB2B system state

As discussed with respect to the executable behaviour of the CB2B model in Section 4.2.1, the contract passes through several states according to several transitions. Figure 4.3 depicts some possible transitions that can be generated from or lead to other states (S_i, S_{i-1} and S_{i+1}). In this context a transition can be observed each time a business event Be_i arrives from the business event generator (BEG) to the contract rule manager (CRM). In terms of Promela language, this involves *write* and *read* operations to a global channel between two asynchronous processes. In fact, if the business event written to the channel between the two processes is conforms to the ROP entities of rights, obligations and prohibitions, a transition is generated ($S_i \rightarrow S_{i+1}$); otherwise the state S_i is maintained until another business event occurs. The set of reachable states from an initial state through consecutive sequence of transitions is called a reachability graph (or RG for short). Formally, that is a 3-tuple (S_0, T_r, S), where S_0 is the set of initial states, T_r is the set of transition relations, and S is a set of states. The RG is a representation of the system's behaviour. It facilitates the discovery of relations and anomalies in system behaviour. Hence, starting from the initial state, a RG is constructed with all the reachable states that are attainable from the former through all possible subsequent transitions.

Fundamentally, in order to derive the system state, Spin translates each process template into a finite automaton. The global behavior of the concurrent system is obtained by computing an asynchronous interleaving product of automata, with one automaton per asynchronous process behavior. The resulting global system behaviour is itself again represented by an automaton. This interleaving product is often referred to as the state space of the system, and, because it can easily be represented as a graph, it is also commonly referred to as the global reachability graph [29].

In the worst case, the global reachability graph has the size of the Cartesian product of all component systems. Although in practice size of global reachability never approaches this worst case, the reachable portion of the Cartesian product can also easily become prohibitively expensive to construct exhaustively. Spin is supported by a number of complexity management techniques which have been developed to combat this problem [29].

Sometimes it is not enough to rely on Spin techniques to combat the complexity problem of Promela models. Different techniques can be employed, and here a technique called *restriction* is used to reduce the size of the state space and the complexity of the model. This restriction called *slicing* [35]. The size of the state space and its complexity is reduced by removing states, transitions and strengthening guards in the model. The resulting model is a smaller model that covers only a subset of the state space of the original model. M^r, A^r and A^r_s may be used to refer to the restricted model, its automaton and state space, respectively. Notice that in M^r the guards in the BEG have been strengthened so that it generates only matched events; that is, events that satisfy conditions *C1* and *C2* as discussed Section 3.1. In other words, the model is suitable for exploring the behaviour of the rules when they are triggered by matched events. A more general model M with automaton A and state space A_s can be built by removing the guards in the BEG of M^r so that the business event generator can provide the rules with events that satisfy condition $C1$ but not necessarily $C2$. Model M can be used to explore the behaviour of the rules when triggered by any event. Since $A^r_s \subset A_s$, all the behaviour of M^r is covered by M. The motivation for using M^r at this early stage of the design is that A^r_s is small; in other words, it is easy to reason about and amenable to exhaustive and rapid verification. It focuses on the exploration of a specific part of the state space A_s, where the designer can use M^r to prove the absence of errors in M^r and claim that M is free from those errors as well.

Next the effect of the restriction technique on contract model which models the contract example presented in Page (74) is shown. The exhaustive verification of the contract model is run with the restriction technique discussed above as well as without it. Then the results are compared in the following:

```
===================================================================
*         Run with Spin default options with restriction          *
===================================================================
State-vector 68 byte, depth reached 100, errors: 0
     225 states, stored
       9 states, matched
     234 transitions (= stored+matched)
       1 atomic steps
hash conflicts:         0 (resolved)
```

```
Stats on memory usage (in Megabytes):
   0.018 equivalent memory usage for states (stored*(State-vector +
   overhead))
   0.251 actual memory usage for states (unsuccessful compression:
   1394.52%)
   state-vector as stored = 1155 byte + 16 byte overhead
   2.539 total actual memory usage
=========================[End Of Spin report]======================


===================================================================
*       Run with Spin default options with out restriction        *
===================================================================

State-vector 68 byte, depth reached 103, errors: 0
     515 states, stored
      75 states, matched
     590 transitions (= stored+matched)
       1 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
   0.041 equivalent memory usage for states (stored*(State-vector +
   overhead))
   0.252 actual memory usage for states (unsuccessful compression:
   609.93%)
   state-vector as stored = 496 byte + 16 byte overhead
   2.539 total actual memory usage

========================[End Of Spin report]=======================
```

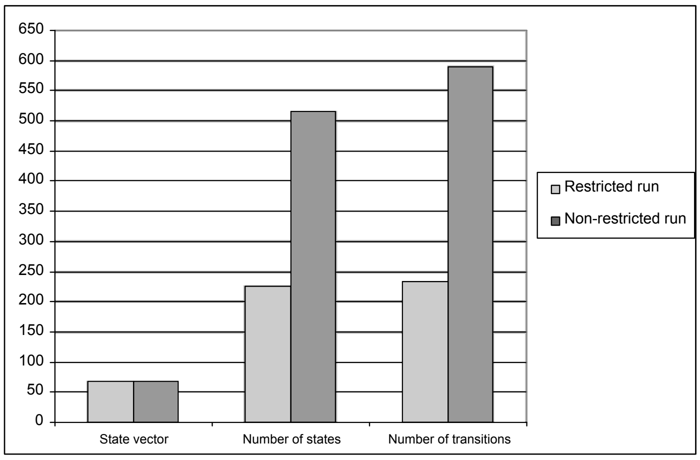
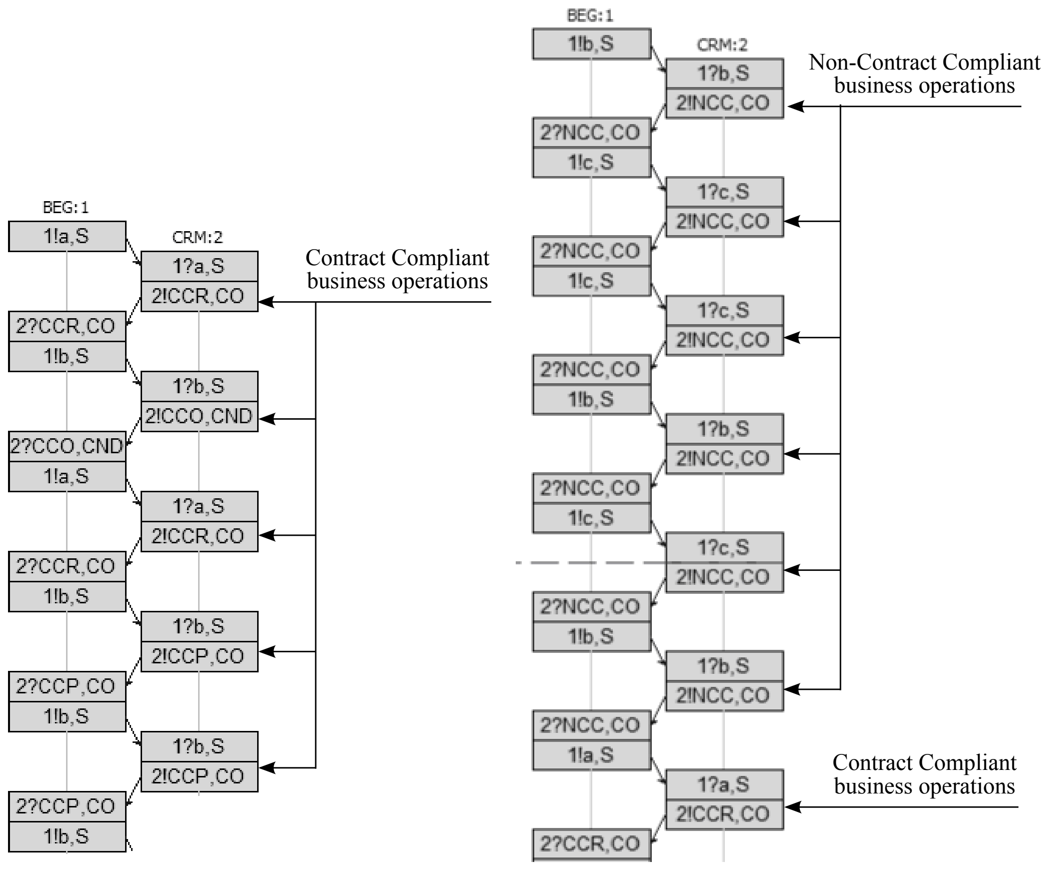


Figure 4.15: Comparison between restricted and non-restricted model runs.

From the two Spin reports and the summary comparison shown in 4.15, it can be seen that without restriction the number of states and transitions is comparatively larger. For illustration, Figure 4.16 is used to show two random contract runs with the same random seed number. In Figure 4.16a, the CB2B model only responds to the contract compliant business operations, thus the execution trace does s not include message of NCC type (i.e the rule reports the business event is Non-Contract Compliant); the messages were CCR (Contract Compliant Right), CCO (Contract Compliant Obligation) or CCP (Contract Compliant Prohibition). However, Figure 4.16b, shows that the first contract compliant business operation witnessed by the event (a, s) at the bottom of the sequence after number of executions included $b, s \rightarrow c, s \rightarrow c, s \rightarrow b, s$ (where 's' indicates the business operation is successful).



(a) - Restricted execution (b) - Non restricted execution

Figure 4.16: Random run of restricted and non-restricted contract executions

4.8 Discussion

This Chapter has described different aspects of the *CB2B* formal model and its use for the verification of contracts. The *CB2B* model hides much of the intricate details of the construction of interacting state machines and enables a designer to encode a contract for model checking directly as ECA rules in terms of the contract entities of business operations, and role players with rights, obligations and prohibitions. Equally importantly, the designer can specify the correctness requirements, in linear temporal logic, directly in terms of the contract entities. The primitives that added to Promela are at user level without any changes to the Spin source code and can be used to build contract models at different levels of abstraction aimed at exploring specific properties of the system.

What distinguishes the high level notation of the e-contract developed in this study is the executable behaviour offered by the CB2B model. Unlike previous work which has only considered the representation of the contract norms, this approach provides an executable notation on which the capabilities of the Spin model checker can be directly exploited for the verification of the contract model.

For example, a contract designer can use the interactive mode of the Spin model checker and verify the execution of the described business events in any order and observe the execution output. Alternatively, a simulation that takes one execution path randomly is also available in Spin. More importantly, a designer can perform an exhaustive verification that would discover potential deadlocks or livelocks in the contract rules. In fact, with our approach to contract modelling the whole features of the Spin model checker become available to the contract designer. Another very useful function of our CB2B model is that it can be used for generating executable test cases for testing actual implementation of contracts, this is discussed in Section 4.6.

Chapter 5

Case studies

This chapter uses the tool for representation and verification of number of e-contracts that that has been derived from natural language descriptions, modelled as ECA rules. It is shown that our representation of a contract model captures the contract's essential elements of business operations, role players, rights, obligations and prohibitions, and it alsi handles CTDs and CTPs. With the help of CB2B model Spin automatically detects contract-independent conflicts as discussed in Chapter 3 and contract-dependent conflicts as discussed in Chapter 2 and Chapter 3. These can be specified with relative ease in terms of contract elements as safety and liveness requirements and then verified with the Spin model checker.

5.1 Internet provision contract

The example used is an Internet provision contract between an Internet service provider (ISP) and a client, where the ISP gives the client contractual access to the Internet. The original contract in this example has been used before [45] to illustrate contract language *CL*. Two parameters for the Internet service in the contract are considered: *high* and *low*, which denote the volume of the client's Internet traffic.

1. Whenever Internet traffic is high then the client must pay \$ X or the client must notify the service provider by sending an e-mail specifying that he will pay later within 24 hours.
2. In case the client delays the payment, after notification he must lower Internet traffic within 24 hours to the low level, and pay \$ $2.X$.
3. If the client does not lower the Internet traffic within 24 hours, then the client will have to pay \$ $3.X$.
4. The provider does not have permission to cancel the contract without previous written notification by normal post and by e-mail.

Step 1 - Declaration Part

```
/*Declaration of some variables*/
  bool INTERNET_HIGH;

/*Declaration of Role Players*/
  RolePlayer(CLIENT,ISP);

/*Declaration of Business Operation*/
  BIS_OP(PAY24H);   /*Client - Pay within 24 hours*/
  BIS_OP(PAY48H);   /*Client - Pay within 48 hours*/
  BIS_OP(DE_NO_48H);/*Client - Delay note within 48 hours*/
  BIS_OP(LOWER24H); /*Client - Lower within 24 hours*/
  BIS_OP(SEND);     /*ISP - Send cancellation email*/
  BIS_OP(WRITE);    /*ISP - Write cancellation letter*/
  BIS_OP(CANCEL);   /*ISP - Cancel Internet service*/
```

Step 2 - Contract initialisation

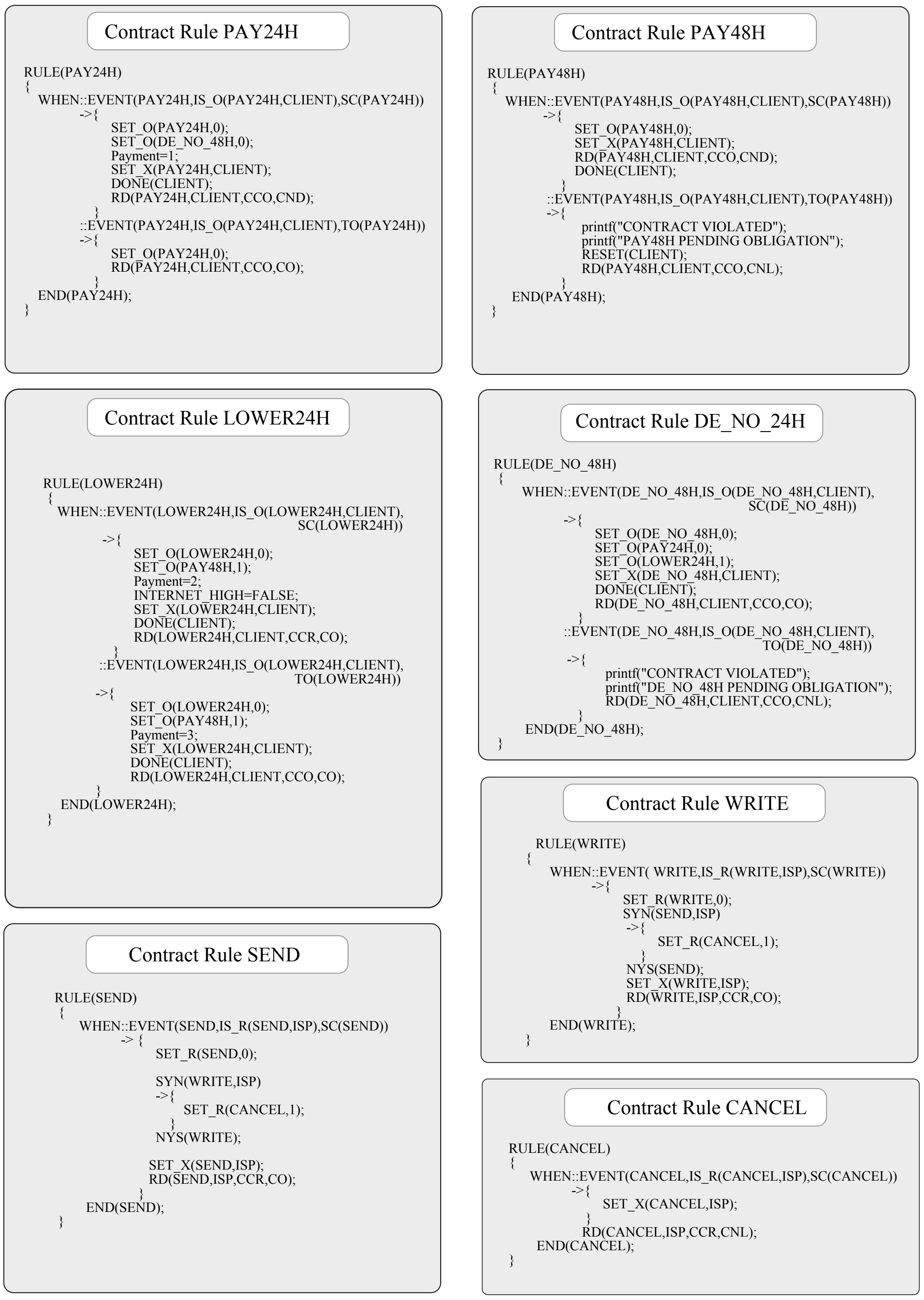
```
  RESET(CLIENT);
  RESET(ISP);
  INIT(PAY24H,CLIENT,0,1,0);
  INIT(PAY48H,CLIENT,0,0,0);
  INIT(DE_NO_48H,CLIENT,0,1,0);
  INIT(SEND,ISP,1,0,0);
  INIT(WRITE,ISP,1,0,0);
  INIT(CANCEL,ISP,0,0,0);
  INIT(LOWER24H,CLIENT,0,0,0);
```

Step 3 - Deriving contract rules

The natural language of the Internet provision contract is manually converted into the following ECA rules:

1. Rule(PAY24H) - Client pays within 24 hours.
2. Rule(PAY48H) - Client pays within 48 hours.
3. Rule(DE_NO_48H) - Client delays payment and sends delay notification within 48 hours.
4. Rule(LOWER24H) - Client lowers Internet traffic within 24 hours after his decision to delay payment.
5. Rule(SEND) - ISP sends cancellation email to the Client.
6. Rule(WRITE) - ISP writes cancellation letter to the Client.
7. Rule(Cancel) - ISP cancels the Internet service.

Figure 5.1 shows how the Internet provision contract rules are implemented as a set of ECA rules with the extended Promela. For example, the implementation of Rule(PAY24H) allows it to respond to business events with success or timeout status, as discussed business events and their status in detail in Chapter 3. Thus, when Rule(PAY24H) is executed within 24 hours, and the business event (PAY24H) is successfully received (i.e. with status 'S'), the expression Event(PAY24H, IS_O(PAY24H, CLIENT), SC(PAY24H)) is evaluated to be TRUE. In such a case the obligation is considered fulfilled and reset with the operation (SET_O(PAY24H,0)). Also, the rule decision RD(PAY24H, CLIENT, CCO, CND) states that: the business event PAY24H generated by the CLIENT implies that the business operation is a contract-compliant obligation (CCO), and the contract execution trace for the client is ended (CND). Otherwise, the expression Event(PAY24H, IS_O(PAY24H, CLIENT) is evaluated as FALSE, and the rule decision is considered non-contract- compliant (NCC) business operation.


Contract Rule PAY24H

```
RULE(PAY24H)
{
  WHEN::EVENT(PAY24H,IS_O(PAY24H,CLIENT),SC(PAY24H))
        ->{
             SET_O(PAY24H,0);
             SET_O(DE_NO_48H,0);
             Payment=1;
             SET_X(PAY24H,CLIENT);
             DONE(CLIENT);
             RD(PAY24H,CLIENT,CCO,CND);
          }
        ::EVENT(PAY24H,IS_O(PAY24H,CLIENT),TO(PAY24H))
        ->{
             SET_O(PAY24H,0);
             RD(PAY24H,CLIENT,CCO,CO);
          }
  END(PAY24H);
}
```

Contract Rule PAY48H

```
RULE(PAY48H)
 {
   WHEN::EVENT(PAY48H,IS_O(PAY48H,CLIENT),SC(PAY48H))
         ->{
              SET_O(PAY48H,0);
              SET_X(PAY48H,CLIENT);
              RD(PAY48H,CLIENT,CCO,CND);
              DONE(CLIENT);
           }
          ::EVENT(PAY48H,IS_O(PAY48H,CLIENT),TO(PAY48H))
          ->{
               printf("CONTRACT VIOLATED");
               printf("PAY48H PENDING OBLIGATION");
               RESET(CLIENT);
               RD(PAY48H,CLIENT,CCO,CNL);
           }
    END(PAY48H);
 }
```

Contract Rule LOWER24H

```
RULE(LOWER24H)
 {
   WHEN::EVENT(LOWER24H,IS_O(LOWER24H,CLIENT),
                                          SC(LOWER24H))
          ->{
               SET_O(LOWER24H,0);
               SET_O(PAY48H,1);
               Payment=2;
               INTERNET_HIGH=FALSE;
               SET_X(LOWER24H,CLIENT);
               DONE(CLIENT);
               RD(LOWER24H,CLIENT,CCR,CO);
            }
         ::EVENT(LOWER24H,IS_O(LOWER24H,CLIENT),
                                          TO(LOWER24H))
         ->{
              SET_O(LOWER24H,0);
              SET_O(PAY48H,1);
              Payment=3;
              SET_X(LOWER24H,CLIENT);
              DONE(CLIENT);
              RD(LOWER24H,CLIENT,CCO,CO);
           }
    END(LOWER24H);
 }
```

Contract Rule DE_NO_24H

```
RULE(DE_NO_48H)
 {
     WHEN::EVENT(DE_NO_48H,IS_O(DE_NO_48H,CLIENT),
                                       SC(DE_NO_48H))
          ->{
               SET_O(DE_NO_48H,0);
               SET_O(PAY24H,0);
               SET_O(LOWER24H,1);
               SET_X(DE_NO_48H,CLIENT);
               DONE(CLIENT);
               RD(DE_NO_48H,CLIENT,CCO,CO);
            }
          ::EVENT(DE_NO_48H,IS_O(DE_NO_48H,CLIENT),
                                         TO(DE_NO_48H))
           ->{
                printf("CONTRACT VIOLATED");
                printf("DE_NO_48H PENDING OBLIGATION");
                RD(DE_NO_48H,CLIENT,CCO,CNL);
            }
     END(DE_NO_48H);
 }
```

Contract Rule WRITE

```
RULE(WRITE)
 {
   WHEN::EVENT( WRITE,IS_R(WRITE,ISP),SC(WRITE))
         ->{
              SET_R(WRITE,0);
              SYN(SEND,ISP)
               ->{
                    SET_R(CANCEL,1);
                 }
               NYS(SEND);
               SET_X(WRITE,ISP);
               RD(WRITE,ISP,CCR,CO);
           }
   END(WRITE);
 }
```

Contract Rule SEND

```
RULE(SEND)
 {
     WHEN::EVENT(SEND,IS_R(SEND,ISP),SC(SEND))
            -> {
                 SET_R(SEND,0);

                 SYN(WRITE,ISP)
                 ->{
                      SET_R(CANCEL,1);
                   }
                 NYS(WRITE);

                SET_X(SEND,ISP);
                RD(SEND,ISP,CCR,CO);
               }
      END(SEND);
 }
```

Contract Rule CANCEL

```
RULE(CANCEL)
 {
    WHEN::EVENT(CANCEL,IS_R(CANCEL,ISP),SC(CANCEL))
          ->{
               SET_X(CANCEL,ISP);
            }
            RD(CANCEL,ISP,CCR,CNL);
     END(CANCEL);
 }
```

Figure 5.1: Internet provision contract rule set

Step 4 - Prepare LTL Formulas

As an example, three LTL formulae are listed below P1, P2 and P3 to verify arbitrary contract-dependent properties. In P1 it is verified that, whenever the business operation of lower traffic within 24 hours is executed, and the business event LOWER24H corresponding to it is successfully received, the payment penalty to pay double the original price is applied. The LTL formula in P2 verifies that, whenever the business operation to lower traffic within 24 hours times out, the payment penalty to pay triple the original price is applied. Finally, P3 verifies that the ISP will be granted the right to cancel after he sent an email and wrote a cancellation letter to inform the client.

```
ltl P1 {[]((((IS_X(LOWER24H,CLIENT)&&SC(LOWER24H)) -> (Payment==2 ))))}
ltl P2 {[]((((IS_X(LOWER24H,CLIENT)&&TO(LOWER24H)) -> (Payment==3 ))))}
ltl P3 {[](((IS_X(SEND,ISP) && (IS_X(WRITE,ISP)))->(IS_R(CANCEL,ISP))))}
```

Step 5 - Verification

After the previous steps 1-4, the contract is ready for verification using the Spin model checker. The set of contract rules shown in Figure 5.1 is tested and found to satisfy the contract-independent properties, as well as the contract dependent properties specified in P_1, P_2 and P_3 above. In fact, we have now reached the final specifications of the set of rules after a number of verification runs and many refinements.

Figure 5.2 shows a random contract run consisting of a number of business operations shared between the contract parties Client and ISP and dictated by the contract in force. As shown in step 2 earlier, the contract is initialised with the client is being obliged to submit payment, or to delay the payment and send delay notification to the ISP. Simultaneously, the ISP is permitted to terminate the Internet service for whatever reason if he sends an email and writes a cancellation letter to the Client in advance.

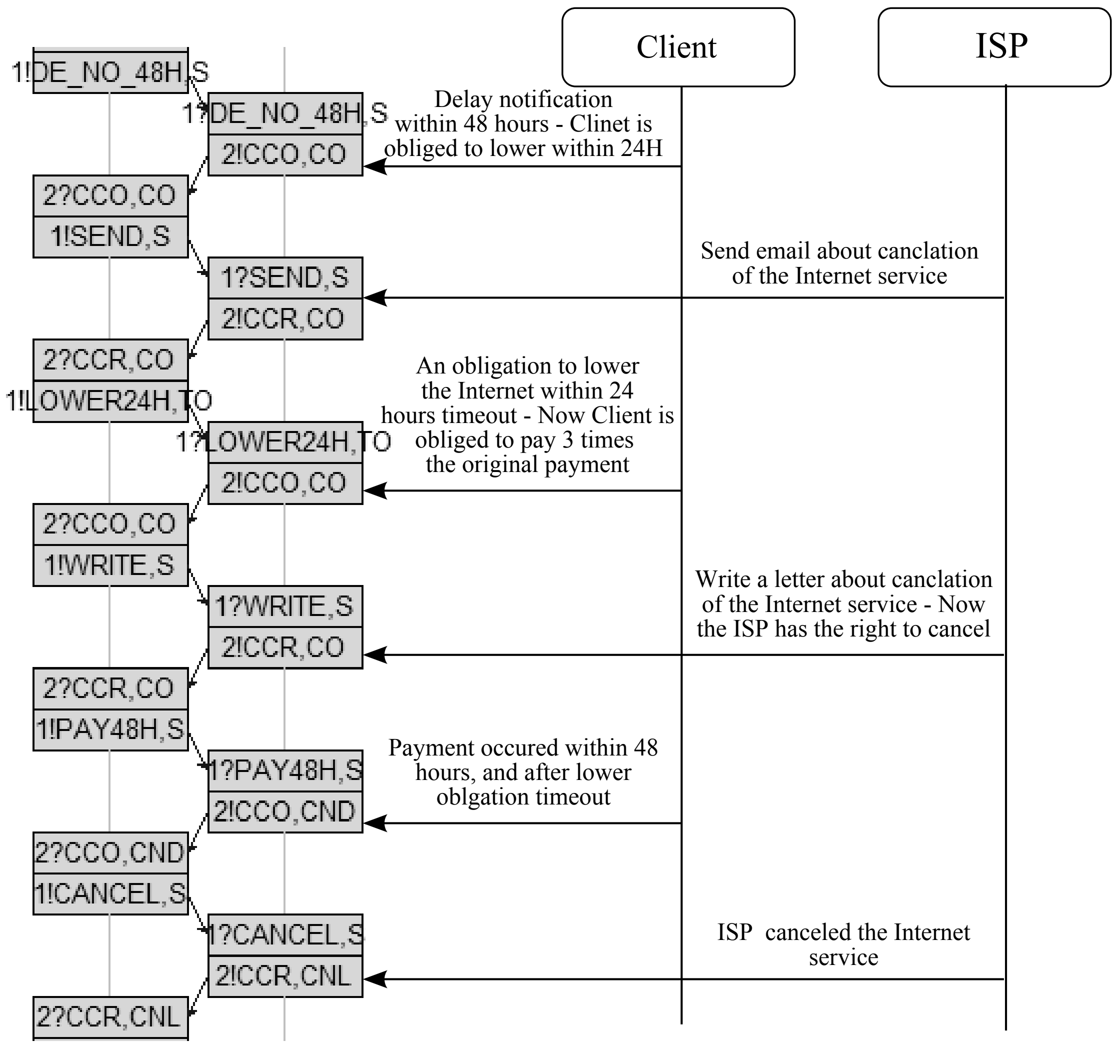

Client
ISP
1!DE_NO_48H,S
1?DE_NO_48H,S
2!CCO,CO
Delay notification within 48 hours - Clinet is obliged to lower within 24H
2?CCO,CO
1!SEND,S
1?SEND,S
2!CCR,CO
Send email about canclation of the Internet service
2?CCR,CO
1!LOWER24H,TO
1?LOWER24H,TO
An obligation to lower the Internet within 24 hours timeout - Now Client is obliged to pay 3 times the original payment
2!CCO,CO
2?CCO,CO
1!WRITE,S
1?WRITE,S
2!CCR,CO
Write a letter about canclation of the Internet service - Now the ISP has the right to cancel
2?CCR,CO
1!PAY48H,S
1?PAY48H,S
2!CCO,CND
Payment occured within 48 hours, and after lower oblgation timeout
2?CCO,CND
1!CANCEL,S
1?CANCEL,S
2!CCR,CNL
ISP canceled the Internet service
2?CCR,CNL

Figure 5.2: Internet provision contract run with Spin model checker

The random business operations of the Client participating in the contract run as depicted in Figure 5.2 can be shown in the following sequence:

$$(DE\_NO\_48H, S) \rightarrow (LOWER24H, TO) \rightarrow (PAY48, S).$$

This run starts when the Client has decided to delay and notify the ISP about his delayed payment within 48 hours as required in the contract rule (DE_NO_48H). Having done that, the client becomes obliged to lower traffic and pay double the price. However, the business event *LOWER24H,TO* implies that the Client has failed to fulfill his obligation to lower traffic within 24 hours. When the obligation to lower times out, the client must pay triple the original price.

Finally, witnessing the business event *PAY48H,S* following *LOWER24H,TO* implies that payment by the client has occurred within 48 hours after lowering the Internet traffic, which means that the contract is ended successfully without pending obligations and a late payment penalty has been applied. Note that the LTL formula specified in P_3 verifies this contract requirement is valid. In a worse scenario, the payment business operation within 48 hours may timeout, so instead of the above sequence of business events we may see the following sequence:

$$(DE\_NO\_48H, S) \rightarrow (LOWER24H, TO) \rightarrow (PAY48H, TO).$$

In such a case, the contract is considered violated and it must be terminated to resolve this matter offline. Resolving the offline obligation is modelled by restarting a new contract session from its initial state with an assumption that the pending obligation has been resolved offline.

Recalling the random contract run depicted in Figure5.2, the sequence of contract business operations for the Internet service provider (ISP) participating in the contractual business run can be shown in the following sequence:

$$\textit{(SEND,S)} \rightarrow \textit{(WRITE,S)} \rightarrow \textit{(CANCEL,S)}.$$

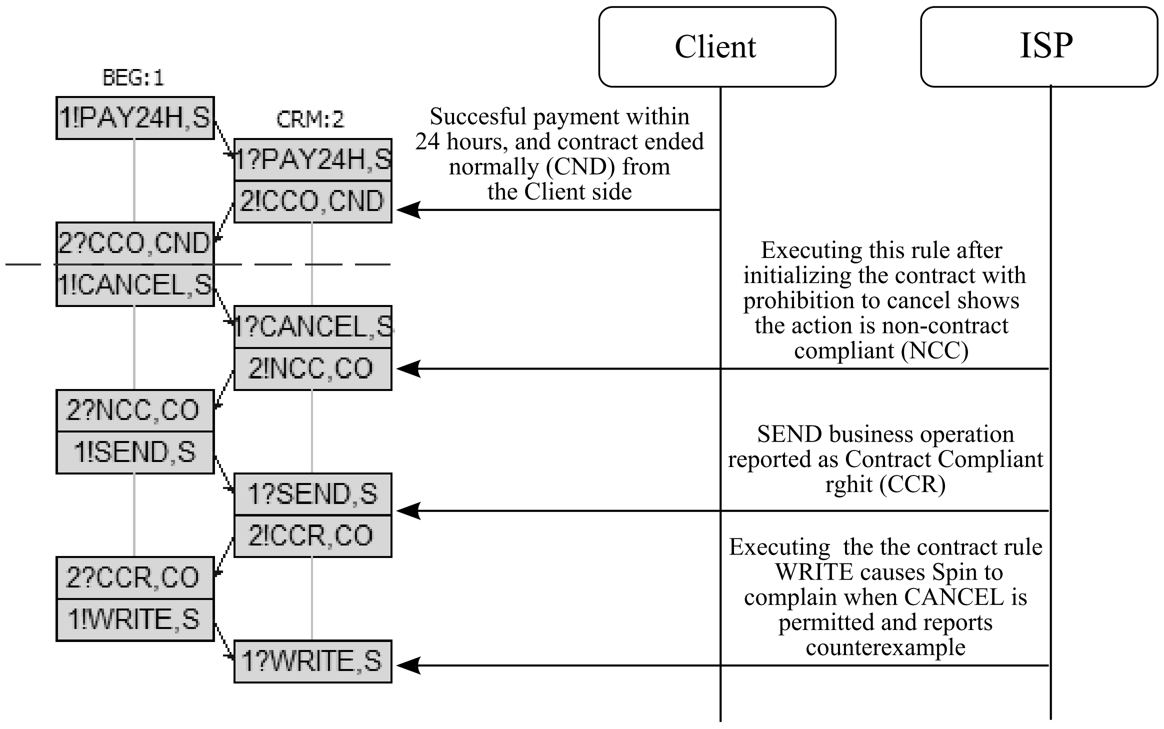
We know from clause 4 of the contract that the ISP does not have permission to cancel the Internet service unless he sent an email and wrote a cancellation letter. These contract clauses are modeled with contract rules SEND and WRITE (see Figure 5.1), and the LTL formula P_3 verifies that this property hold for all contract executions.

We can inject an error into rule SEND or rule WRITE by withdrawing from the ISP the right to cancel when both SEND and WRITE are executed. Such an error may occur as a result of omission or the bad interpretation of contract clause number 4. After this update, a verification for the property P_3 reveals a counterexample after the sequence $\mathit{SEND,S} \rightarrow \mathit{WRITE,S}$, which is the shortest path to this failure. This means that both business operations (READ and WRITE) have been executed and the ISP has not been granted the right to cancel the service. The implementation of such erroneous contract rules would means that the ISP would not be able to cancel the service and this may cause disputes between the ISP and the Client.

In another error injection example, the contract could initialize with a prohibition on the ISP to perform the cancel business operation. In this case Spin should detect the error automatically and report it as an assertion violation counterexample. Executing Spin with its default settings with the option for assertion violation detection, Spin stop its verification process and returns a violation report part of which is shown below. It can be sees that the assertion statement shows that there is a state where the contract permits the cancel business operation and prohibits it at the same time.

```
Spin: INTERNET1.c:278, Error: assertion violated
Spin: text of failed assertion:
assert(!(((CANCEL_bo.right==1)&&(CANCEL_bo.prohib==1))))
#processes: 3
```

Another key feature of verification using the CB2B model and the Spin model checker is that a contract designer can can detect that the cancel business operation has been reported as a non-contract compliant shown as a NCC message in the returned counterexample. Figure 5.3 depicts Spin's counterexample when this contract independent property has been violated. The returned counterexample also shows that the contract rule CANCEL's decision when it is triggered to handle the business event (CANCEL,S) is considers it non-contract compliant. This is because the rule



Client
ISP
BEG:1
CRM:2
1!PAY24H,S
1?PAY24H,S
2!CCO,CND
2?CCO,CND
1!CANCEL,S
1?CANCEL,S
2!NCC,CO
2?NCC,CO
1!SEND,S
1?SEND,S
2!CCR,CO
2?CCR,CO
1!WRITE,S
1?WRITE,S
Succesful payment within 24 hours, and contract ended normally (CND) from the Client side
Executing this rule after initializing the contract with prohibition to cancel shows the action is non-contract compliant (NCC)
SEND business operation reported as Contract Compliant rghit (CCR)
Executing the the contract rule WRITE causes Spin to complain when CANCEL is permitted and reports counterexample

Figure 5.3: Spin's counterexample when CANCEL is prohibited and permitted

can only handle the situation when the ISP has been permitted, or given the right to perform the CANCEL business operation. The same rule can handle prohibition as well, but so long as there is no sanction or penalties described in the contract in case of prohibition violation, it can be ignored.

5.2 Storage service consumption contract

This section presents a hypothetical contract for service agreement signed by a client agreeing on the terms and conditions provided by the storage service provider (SSP).

The contract is intended to regulate storage service consumption with some restrictions on the amount of space that can be used by a Client. This contract shows another example of the use of contracts for the management and regulation of online services that can be offered as Cloud services and consumed by different types of users.

1. The Client is entitled to use the service in normal quota mode of 100 GB or exceeded quota mode of 120 GB.
2. Clients that exhaust their normal quota are obliged to either:
   (a) Submit a single payment of £5 within the next 72 hours.
   (b) Bring the quota back to normal within the next 72 hours by deleting sufficient number of files.
3. Violation of clause 2 will give the right to the storage service provider to suspend the service.

Step 1 - Declaration Part

```
/*Declaration of Role Players*/
  RolePlayer(CLIENT);

/*Declaration of Business Operation*/
  BIS_OP(START);      /*Start the contract*/
  BIS_OP(EXC_100GB);  /*Exceeds 100 GB */
  BIS_OP(PAY5D_72H);  /*Pay 5 pounds within 72 hours*/
  BIS_OP(BRING_QB_72);/*Bring quota back within 72 hours*/
```

Step 2 - Contract initialisation

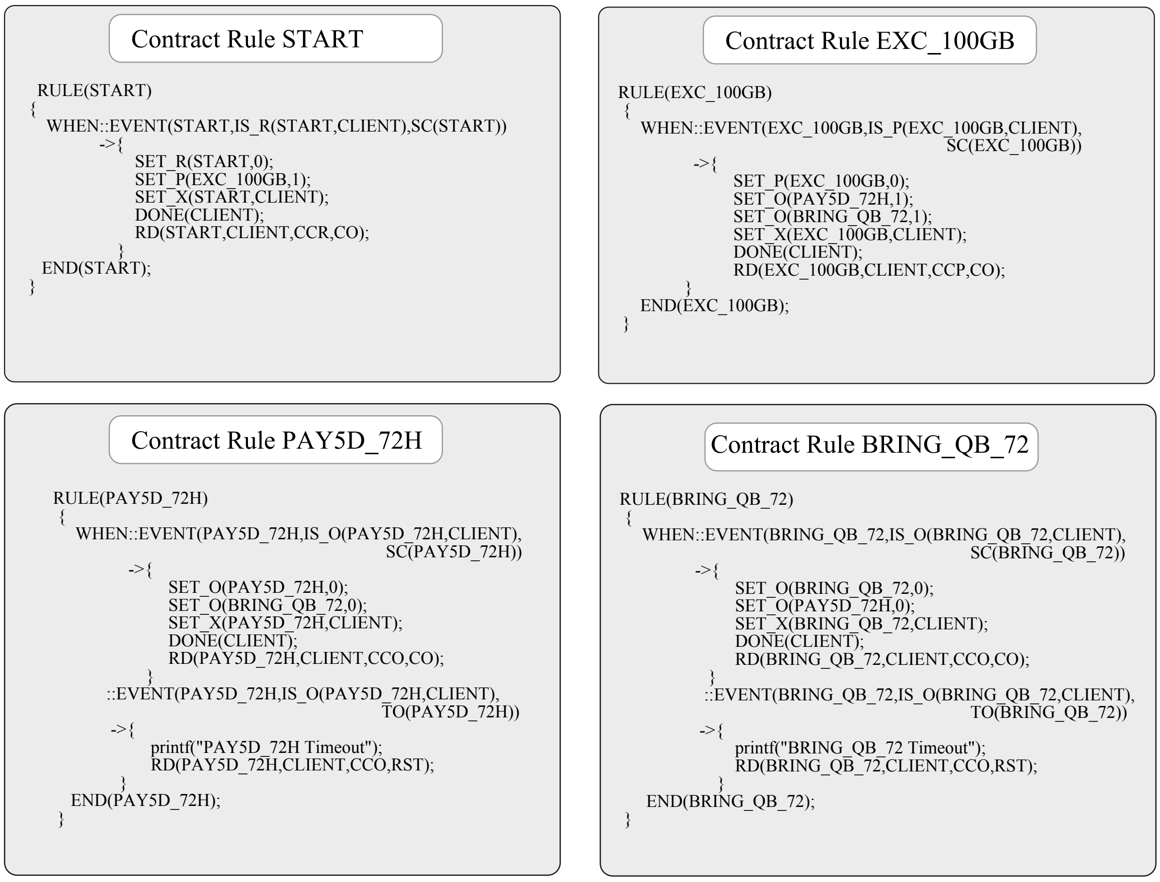
```
  RESET(CLIENT);
  INIT(START,CLIENT,1,0,0);
  INIT(EXC_100GB,CLIENT,0,0,0);
  INIT(PAY5D_72H,CLIENT,0,0,0);
  INIT(BRING_QB_72,CLIENT,0,0,0);
```

Step 3 - Deriving the contract rules

The natural language of the storage service contract is manually converted into the following ECA rules:

1. Rule(START) - Client starts storage storage service.
2. Rule(EXC_100GB) - Client exceeds 100GB.
3. Rule(PAY5D_72H) - Client pays £5 within 72 hours.
4. Rule(BRING_QB_72H) - Client brings the quota back within 72 hours.

We can see in the implementation of the rules below that, Rule(EXC_100GB) and Rule(PAY5D_72H) handle both the success and timeout of business obligations. In case of the timeout of an obligation the whole contract might be restarted.



Contract Rule START

```
RULE(START)
{
  WHEN::EVENT(START,IS_R(START,CLIENT),SC(START))
          ->{
               SET_R(START,0);
               SET_P(EXC_100GB,1);
               SET_X(START,CLIENT);
               DONE(CLIENT);
               RD(START,CLIENT,CCR,CO);
            }
  END(START);
}
```

Contract Rule EXC_100GB

```
RULE(EXC_100GB)
 {
   WHEN::EVENT(EXC_100GB,IS_P(EXC_100GB,CLIENT),
                                        SC(EXC_100GB))
          ->{
               SET_P(EXC_100GB,0);
               SET_O(PAY5D_72H,1);
               SET_O(BRING_QB_72,1);
               SET_X(EXC_100GB,CLIENT);
               DONE(CLIENT);
               RD(EXC_100GB,CLIENT,CCP,CO);
            }
   END(EXC_100GB);
 }
```

Contract Rule PAY5D_72H

```
RULE(PAY5D_72H)
 {
   WHEN::EVENT(PAY5D_72H,IS_O(PAY5D_72H,CLIENT),
                                        SC(PAY5D_72H))
          ->{
               SET_O(PAY5D_72H,0);
               SET_O(BRING_QB_72,0);
               SET_X(PAY5D_72H,CLIENT);
               DONE(CLIENT);
               RD(PAY5D_72H,CLIENT,CCO,CO);
            }
        ::EVENT(PAY5D_72H,IS_O(PAY5D_72H,CLIENT),
                                        TO(PAY5D_72H))
          ->{
               printf("PAY5D_72H Timeout");
               RD(PAY5D_72H,CLIENT,CCO,RST);
            }
   END(PAY5D_72H);
 }
```

Contract Rule BRING_QB_72

```
RULE(BRING_QB_72)
 {
   WHEN::EVENT(BRING_QB_72,IS_O(BRING_QB_72,CLIENT),
                                        SC(BRING_QB_72))
          ->{
               SET_O(BRING_QB_72,0);
               SET_O(PAY5D_72H,0);
               SET_X(BRING_QB_72,CLIENT);
               DONE(CLIENT);
               RD(BRING_QB_72,CLIENT,CCO,CO);
            }
        ::EVENT(BRING_QB_72,IS_O(BRING_QB_72,CLIENT),
                                        TO(BRING_QB_72))
          ->{
               printf("BRING_QB_72 Timeout");
               RD(BRING_QB_72,CLIENT,CCO,RST);
            }
   END(BRING_QB_72);
 }
```

Figure 5.4: Storage service contract

Step 4 - Prepare LTL formulae

As in the previous example, some of the properties of interest are specified as LTL formulae. For example, the LTL formula P1 verifies that, whenever a client starts using the storage service, he is prohibited from exceeding 100GB. The LTL formula in P2 verifies that whenever the prohibition in P1 is violated, and 100GB of storage service is exceeded, the Client must choose between paying £5, or bringing the quota back within 72 hours.

```
ltl P1 {[](IS_X(START,CLIENT)->(IS_P(EXC_100GB,CLIENT)))}
ltl P2 {[]((IS_X(EXC_100GB,CLIENT))->(IS_O(PAY5D_72H,CLIENT)&&
                                       IS_O(BRING_QB_72,CLIENT)))}
```

Step 5 - Verification

The contract properties of interest are verified in the same manner as in the previous example. Note here that clause 2 is interpreted as a prohibition to exceed 100GB of storage space (EXC_100GB). In this case a violation to this clause would cause the client to have to choose between two obligations: either he pays £5, or brings the quota back to 100 GB or less. If one of the obligations timeout, it would be considered as a violation to the whole contract. If the whole contract is violated, a new session of the contractual interaction is started.

The LTL formulae P_1 and P_2 have been verified successfully. In P_1 it is checked that the contract always prohibts exceeding 100GB once it has been started. In P_2, it is verified that, if the normal quota of 100GB is exceeded, there must be two obligations ($PAY5D\_72H$ and $BRING\_QB\_72$) and the CLINET must choose one of them. As in the previous examples, the LTL formulae do not specify the entire list of contract-dependent requirements, as only some examples are included.

5.3 Buyer/seller contract

A hypothetical example of a buyer/seller contract is considered in this section. Although this contract is not comprehensive (for example, invoicing is not stipulated), it does contain clauses of considerable degree of complexity. Also, this contract includes clauses consider BizFail and TecFail events that may encountered during the contractual business exchange. Notice that following the ebXML specifications [13] (discussed in Section 3.1), it is assumed that once a conversation is started, (i.e., a business operation is initiated) it always completes to produce an execution outcome event from the set Success, BizFail, TecFail whose elements represent respectively a successful conclusion, a business failure or a technical failure. TecFail models protocol related failures detected at the middleware level, such as a late, syntactically incorrect or a missing message. BizFail models semantic errors in a message detected at the business level, e.g., the goods-delivery address extracted from the business document is invalid.

1. Offers and purchase orders
    - 1.1 The seller is entitled to send an offer to the buyer.
    - 1.2 The buyer has the right to use its sole discretion to ignore an offer or respond to it by submitting a corresponding purchase order.
    - 1.3 Failure to respond to the offer within 10 days shall complete the contractual transaction.
2. Discounts
    - 2.1 The seller agrees to grant 15% discount to purchase orders submitted within 7 days of the receipt of the offer.
    - 2.2 A purchase order submitted after 7 days (but not exceeding 10 days) will be processed but granted no discount unless clauses 4.1 or 4.2 apply.
    - 2.3 Purchase orders submitted after 10 days will not be processed online.

3. Payment

   3.1 The buyer is obliged to submit payment within 5 days of sending the purchase order.

   3.2 Payments made after 5 days will incur a 10% fine and, if submitted, not considered for online processing, unless clause 4.3 applies.

4. Delayed purchase orders and payments

   4.1 A delayed purchase order due to business reasons shall be granted only 10% discount.

   4.2 A delayed purchase order due to technical problems shall be granted 15% discount.

   4.3 Failure to meet a payment deadline due to business or technical reasons will grant:

      4.3.1 a payment deadline extension of 5 days to the buyer.

      4.3.2 right of purchase order cancellation to the seller.

5. Cancellation and refunds

   5.1 The seller is obliged to refund payments received after cancellations.

6. Number of failures

   6.1 If the total number of business and technical failures exceed an agreed bound, then online processing will be terminated

Step 1 - Declaration Part

```
/*Declaration of some variables*/
  byte Discount;
  byte Fine;

/*Declaration of Role Players*/
  RolePlayer(SELLER, BUYER);

/*Declaration of Business Operation*/
  BIS_OP(OFFER);    /*Offer submission*/
  BIS_OP(PO7D);     /*Purchase order submitted within seven days*/
  BIS_OP(POCNL);    /*Purchase order cancellation*/
  BIS_OP(REFUND);   /*Refund payment after cancellation*/
  BIS_OP(PO10D);    /*purchase order submitted within ten days*/
  BIS_OP(PAY5DAY);  /*Payment within 5 Days*/
  BIS_OP(PAY5DEXT); /*Payment within 5 Days extension*/
```

Step 2 - Contract initialisation

```
RESET(BUYER);
RESET(SELLER);
INIT(OFFER,SELLER,1,0,0);
INIT(POCNL,SELLER,0,0,0);
INIT(REFUND,SELLER,0,0,0);
INIT(PO7D,BUYER,0,0,0);
INIT(PO10D,BUYER,0,0,0);
INIT(PAY5DAY,BUYER,0,0,0);
INIT(PAY5DEXT,BUYER,0,0,0);
```

Step 3 - Prepare LTL Formulae

```
ltl P1 {[]((IS_X(OFFER, SELLER)-> IS_R(PO7D, BUYER)))}
ltl P2 {[](((IS_X(PO10D, BUYER)&&BF(PO10D)))->(Discount==10))}
ltl P3 {[](((IS_X(PO10D, BUYER)&&TF(PO10D)))->(Discount==15))}
ltl P4 {[]<>(((IS_X(PAY5DEXT, BUYER)))->(Fine==10))}
ltl P5 {[](IS_O(REFUND, SELLER)->(IS_X(POCNL, SELLER)))}
```

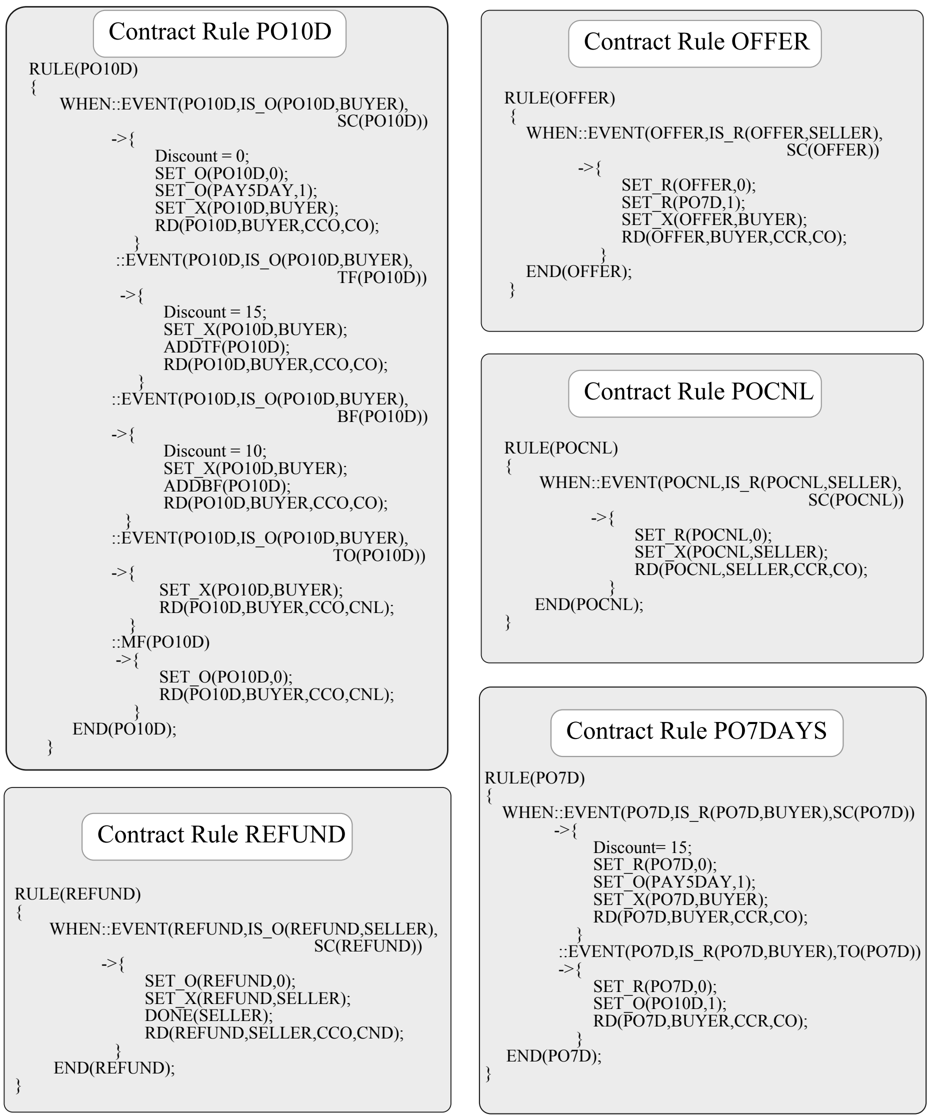
The LTL properties P1 to P5 are samples of the contract requirements in this case study. P1 verifies that the execution of an offer business operation gives the right to the buyer to submit purchase order within 7 days. P2 verifies that if purchase order within 10 days executed and business failure encountered, then the buyer becomes eligible for 10% discount.

Similarly P3 checks that a technical failure grants the buyer 15% discount. The formula in P4 checks that the contract model handles the payment when 5 days deadline is timeout; the event is infinitely often occurs, and whenever it occurs the seller must pay extra 10% fine of the total payment. Finally, P5 would check that the obligation to refund occurs only after the seller has cancelled purchase order.

Step 4 - Deriving the contract rules

The natural language of the Buyer/Seller contract is manually converted into the following ECA rules:

1. Rule(OFFER) - Seller submits an offer.
2. Rule(PO7D) - Buyer submits purchase order in 7 days.
3. Rule(POCNL) - Seller cancels purchase order.
4. Rule(PO10D) - Buyer submits purchase order in 10 days.
5. Rule(PAY5DAY) - Buyer submits payment in 5 days.
6. Rule(PAY5DEXT)- Buyer submits payment after first 5 days period is extended.


Contract Rule PO10D

```
RULE(PO10D)
{
    WHEN::EVENT(PO10D,IS_O(PO10D,BUYER),
                                      SC(PO10D))
         ->{
               Discount = 0;
               SET_O(PO10D,0);
               SET_O(PAY5DAY,1);
               SET_X(PO10D,BUYER);
               RD(PO10D,BUYER,CCO,CO);
           }
         ::EVENT(PO10D,IS_O(PO10D,BUYER),
                                      TF(PO10D))
          ->{
                Discount = 15;
                SET_X(PO10D,BUYER);
                ADDTF(PO10D);
                RD(PO10D,BUYER,CCO,CO);
            }
         ::EVENT(PO10D,IS_O(PO10D,BUYER),
                                      BF(PO10D))
         ->{
                Discount = 10;
                SET_X(PO10D,BUYER);
                ADDBF(PO10D);
                RD(PO10D,BUYER,CCO,CO);
           }
         ::EVENT(PO10D,IS_O(PO10D,BUYER),
                                      TO(PO10D))
         ->{
               SET_X(PO10D,BUYER);
               RD(PO10D,BUYER,CCO,CNL);
           }
         ::MF(PO10D)
          ->{
               SET_O(PO10D,0);
               RD(PO10D,BUYER,CCO,CNL);
            }
     END(PO10D);
  }
```

Contract Rule REFUND

```
RULE(REFUND)
{
    WHEN::EVENT(REFUND,IS_O(REFUND,SELLER),
                                      SC(REFUND))
         ->{
               SET_O(REFUND,0);
               SET_X(REFUND,SELLER);
               DONE(SELLER);
               RD(REFUND,SELLER,CCO,CND);
           }
     END(REFUND);
}
```

Contract Rule OFFER

```
RULE(OFFER)
 {
   WHEN::EVENT(OFFER,IS_R(OFFER,SELLER),
                                 SC(OFFER))
        ->{
               SET_R(OFFER,0);
               SET_R(PO7D,1);
               SET_X(OFFER,BUYER);
               RD(OFFER,BUYER,CCR,CO);
          }
   END(OFFER);
 }
```

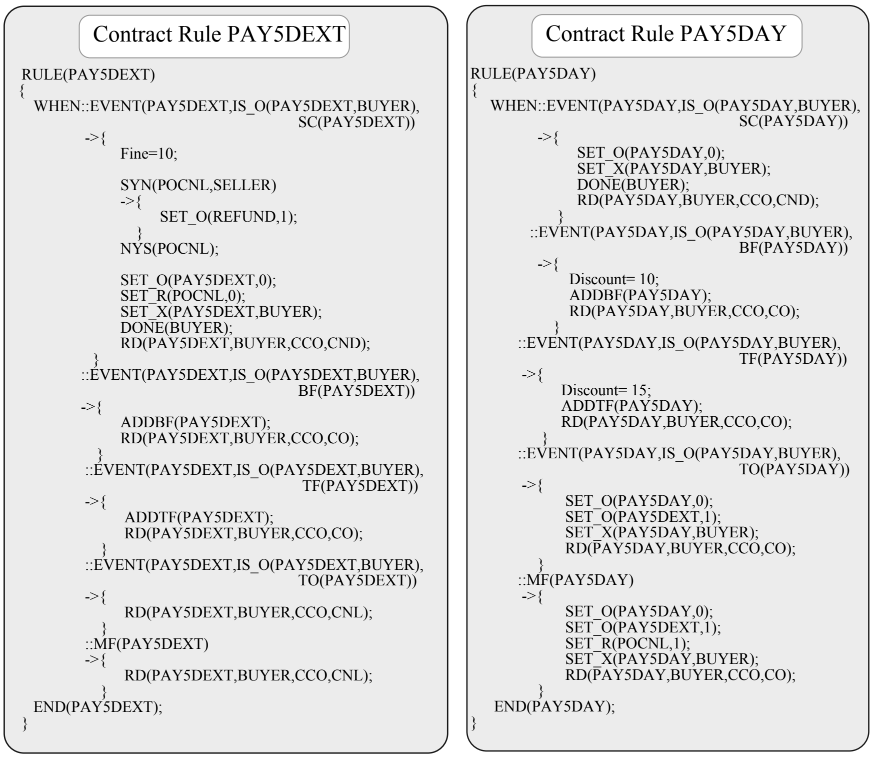
Contract Rule POCNL

```
RULE(POCNL)
{
    WHEN::EVENT(POCNL,IS_R(POCNL,SELLER),
                                  SC(POCNL))
         ->{
               SET_R(POCNL,0);
               SET_X(POCNL,SELLER);
               RD(POCNL,SELLER,CCR,CO);
           }
    END(POCNL);
}
```

Contract Rule PO7DAYS

```
RULE(PO7D)
{
  WHEN::EVENT(PO7D,IS_R(PO7D,BUYER),SC(PO7D))
       ->{
             Discount= 15;
             SET_R(PO7D,0);
             SET_O(PAY5DAY,1);
             SET_X(PO7D,BUYER);
             RD(PO7D,BUYER,CCR,CO);
         }
       ::EVENT(PO7D,IS_R(PO7D,BUYER),TO(PO7D))
       ->{
             SET_R(PO7D,0);
             SET_O(PO10D,1);
             RD(PO7D,BUYER,CCR,CO);
         }
  END(PO7D);
}
```

Figure 5.5: Buyer/Seller contract rules


Contract Rule PAY5DEXT
Contract Rule PAY5DAY

```
RULE(PAY5DEXT)
{
  WHEN::EVENT(PAY5DEXT,IS_O(PAY5DEXT,BUYER),
                                SC(PAY5DEXT))
         ->{
              Fine=10;

              SYN(POCNL,SELLER)
              ->{
                   SET_O(REFUND,1);
                 }
              NYS(POCNL);

              SET_O(PAY5DEXT,0);
              SET_R(POCNL,0);
              SET_X(PAY5DEXT,BUYER);
              DONE(BUYER);
              RD(PAY5DEXT,BUYER,CCO,CND);
           }
        ::EVENT(PAY5DEXT,IS_O(PAY5DEXT,BUYER),
                                BF(PAY5DEXT))
        ->{
              ADDBF(PAY5DEXT);
              RD(PAY5DEXT,BUYER,CCO,CO);
            }
         ::EVENT(PAY5DEXT,IS_O(PAY5DEXT,BUYER),
                                 TF(PAY5DEXT))
         ->{
               ADDTF(PAY5DEXT);
               RD(PAY5DEXT,BUYER,CCO,CO);
             }
         ::EVENT(PAY5DEXT,IS_O(PAY5DEXT,BUYER),
                                TO(PAY5DEXT))
         ->{
               RD(PAY5DEXT,BUYER,CCO,CNL);
             }
         ::MF(PAY5DEXT)
         ->{
               RD(PAY5DEXT,BUYER,CCO,CNL);
             }
  END(PAY5DEXT);
}
```

```
RULE(PAY5DAY)
{
   WHEN::EVENT(PAY5DAY,IS_O(PAY5DAY,BUYER),
                               SC(PAY5DAY))
          ->{
               SET_O(PAY5DAY,0);
               SET_X(PAY5DAY,BUYER);
               DONE(BUYER);
               RD(PAY5DAY,BUYER,CCO,CND);
             }
         ::EVENT(PAY5DAY,IS_O(PAY5DAY,BUYER),
                               BF(PAY5DAY))
          ->{
              Discount= 10;
              ADDBF(PAY5DAY);
              RD(PAY5DAY,BUYER,CCO,CO);
             }
        ::EVENT(PAY5DAY,IS_O(PAY5DAY,BUYER),
                               TF(PAY5DAY))
        ->{
             Discount= 15;
             ADDTF(PAY5DAY);
             RD(PAY5DAY,BUYER,CCO,CO);
           }
        ::EVENT(PAY5DAY,IS_O(PAY5DAY,BUYER),
                               TO(PAY5DAY))
        ->{
              SET_O(PAY5DAY,0);
              SET_O(PAY5DEXT,1);
              SET_X(PAY5DAY,BUYER);
              RD(PAY5DAY,BUYER,CCO,CO);
           }
        ::MF(PAY5DAY)
        ->{
              SET_O(PAY5DAY,0);
              SET_O(PAY5DEXT,1);
              SET_R(POCNL,1);
              SET_X(PAY5DAY,BUYER);
              RD(PAY5DAY,BUYER,CCO,CO);
           }
    END(PAY5DAY);
}
```

Figure 5.6: Buyer/Seller contract rules cont.

Unlike the previous case studies, here some of the contract rules such as Rule(PO10D), Rule(PAY5DAY) and Rule(PAY5DEXT) consider business and technical failures associated with the business events. For example, Rule(PAY5DEXT) shows how these failures are handled; ADDBF(PAY5DEXT) and ADDTF(PAY5DEXT) increment the business failures or technical failures of a contract model respectively. The model assumes a maximum of one technical or business failure only, operation MF(PAY5DEXT) becomes executable if the maximum failure is reached, so the proper action (as described in the contract) can be taken. We note that the Rule(PAY5DEXT) also handles the case when purchase orders are canceled after the buyer has submitted payment; within the SYN..NYS construct, the seller is obliged to refund if the purchase order cancellation (POCNL) has been executed. Figure 5.5 and Figure 5.6 show the ECA rules of the buyer/seller contract model.

This contract has been been verified using an earlier version of the CB2B model [2], and used for test case generation to test an actual implementation of a contract compliance checker. An executable version in EROP language is developed and installed in the contract compliance checker CCC [38] which then becomes the SUT. For the same natural language contract, a contract model in our extended Promela is developed (as in our buyer/seller example above) and validated with respect to contract correctness requirements (such as termination in acceptable final states, checking that deadline extensions have been granted exactly as stated in the clauses, refund has taken place properly and so forth). This validated model is then used for generating test cases and applying them to the SUT. Informally, an *execution sequence or execution trace* is a sequence of business operations executed by the business partners that drive the interaction from its initial to a final state. The system under test (CCC) implements the contract in force and can examine a trace and determine whether an event, representing a business operation is contract compliant or not. An example of execution trace is $PO7DAY_S \rightarrow PAY15DSC_{TF}$ means that the successful execution of a purchase order submitted within seven days was followed by the execution of a payment entitled to 15% discount that, unfortunately, completed in a technical failure.

Generally the notation $e_i \rightarrow e_j$ is used to indicate that event e_i *precedes* event e_j, and the name of the operation is appended with S, BF, TF or TO, to indicate, respectively, that the execution produced success, business failure, technical failure or that the time out to complete the execution expired. Since it is assumed that the abstract model is correct after its verification, it is reasonable to expect that a correctly functioning CCC should consume (accept) the execution sequences produced by the contract model. Likewise, since the events are actually state transition events, the execution of such a given sequence should drive the SUT from its initial state to one of its final states that matches the state of the model that consumes the same sequence.

The SUT is instrumented to accept the traces produced by the model. A Java application was written to produces concrete business events from the abstract ones. At this stage the generated events become ready to be imported into the SUT. Note here timeout events need special attention. When a timeout event is encountered in the sequence, its presence is used to ensure that the corresponding deadline in the time module expires straight away. To ascertain what state the SUT is in, it was also instrumented so as to reveal the contents of its ROP sets. In this way it is possible to check whether or not the ROP sets of the SUT match those of the model as state transitions occur: a mismatch indicating a flaw in the SUT. We are thus able to automate the testing of the CCC.

For example, the behaviour of the SUT can be shown when the following fragment is presented as input: $OFFER_{TF} \rightarrow OFFER_{S} \rightarrow PAY7DAY_{TO}$. The SUT goes through correct state transitions, beginning with the initial state where the seller's ROP set indicates that it has the right to make an offer, and the buyer's ROP set is empty. The first attempt at an offer fails and after a successful offer event is encountered, the seller's ROP set becomes empty and the buyer is given a right to submit a purchase order within seven days. However, no such operation is performed and the seven day timeout event occurs, and so the buyer is given a right to submit the purchase order within 10 days:

```
Type: init, Status: S
Seller ROP set:{ROPEntity-BO Type:OFFER, ROP Type:Right}
Buyer ROP set :{Empty}
Type: OFFER, Status: TF
Seller ROP set:{ROPEntity-BO Type:OFFER, ROP Type:Right}
Buyer ROP set :{Empty }
Type: OFFER, Status: S
```

```
Seller ROP set:{Empty  }
Buyer  ROP set:{ROPEntity-BO Type:PO7D, ROP Type:Right }
Type: PO7D, Status: TO
Seller ROP set:{Empty  }
Buyer  ROP set:{ROPEntity-BO Type:PO10D ROP Type:Right }
```

Suppose there is a flaw in the SUT. Say for example the rule that deals with the timeout of the purchase order within seven days fails to grant the buyer the right to submit the purchase order within 10 days. In this case the ROP sets will be empty and will not correspond to those of the model; and flaw is detected.

```
Type: PO7D, Status: TO
Seller ROP set: {Empty  }
Buyer  ROP set: {Empty  }
```

5.4 Performance issues

One of the main problems with model checking is how to restrain the state space explosion. It can be caused by the level of detail or the level of concurrency within the validation model [27]. Optimising parameters such as the number of states, state vector size, size of search stack and the verification time can considerably improve the process of model verification. The state vector in the Promela model is the set of information stored by Spin to uniquely identify the system state. It contains information on the global variables, contents of each channel, process counter and the local variables in each process. The size of the state vector represents the amount of storage space it occupies. This in turn determines the state space of the model for the set of all possible states which occur during the computation. State space size is the total space required to store the state vectors corresponding to all states. If the size of a single state vector is m bytes and there are n states in the Promela model, the state space size is $m * n$ bytes.

The contract models discussed in this Chapter executed on a HP Compaq PC with Intel(R) Core(TM)2, CPU 6300 @ 1.86 GHz 1.86 GHz and 1.98 GB of RAM. The operating system is Microsoft Windows XP Professional, Version 2001, Service Pack 3. The models verified using Spin Version 6.1.0 and iSpin version 1.0.3. All contract models experimented with the default options of iSpin with few changes. For example, the assertion violation box check–box on, the storage mode exhaustive and the search algorithm depth–first search (DFS). The next table summarises some performance metrics returned by Spin during the verification of the models. As the table shows, we have been able to verify interesting properties of contracts with reasonable size. We discussed our technique to maintain the state space of CB2B models in detail in Section 4.7.

Internet-provision contract			**Storage service contract**			**Buyer/Seller contract**		
Stored states	Memory usage	Elapsed time	Stored states	Memory usage	Elapsed time	Stored states	Memory usage	Elapsed time
29950 states	5.664 Mbyte	0.047 seconds	287 states	2.539 Mbyte	0.015 seconds	317228 states	317228 states	0.672 seconds
State-vector size			State-vector size			State-vector size		
96 byte			68 byte			152 byte		

Table 5.1: Number of states and elapsed time of the case studies

Chapter 6

Conclusions and future work

6.1 Conclusions

This thesis developed a toolkit for the model checking of electronic contracts. It has employed the facilities of the Spin model checker and its input language Promela with some extensions for the process of modelling contracts. The extensions aimed to use the language of a high-level of notation in order to facilitate the composition of a contract model as a set of ECA rules. With the high level notation of the language the user can intuitively write contract rules, thus benefiting from the well-known and widely used Spin model checker for the verification and validation of the correctness of contracts.

For contract verification, this thesis considers two major types of contract conflicts: contract–independent conflicts, which are general conflicts that may affect any contract, and contract–dependent conflicts specific to each contract. In Chapter 3, a contract compliance checker (CCC) was modelled as a reactive system, and the contract correctness requirements to detect both contract–dependent and contract–independent conflicts are specified as safety and liveness properties.

In Chapter 4, the Spin model checker and its input language Promela were used to develop a contract verification model (CB2B) based on the concepts of EROP system, its contract compliance checker and its rule–based EROP language. The CB2B is abstracting the behaviour of the contract compliant checker CCC of EROP system and empowered with a set of operations facilitates manipulation of contract elements (e.g. rights, obligations, prohibitions , etc...) held in the system memory. Furthermore, we have implemented an abstract data type extension to the standard Promela called BIS_OP. We also implemented a set of operations on the BIS_OP data type to maintain information about ROP sets. This extension, facilitated writing the contract models in a way mimicking the EROP language, however, our contracts can be automatically verified with the Spin model checker. Chapter 4 also describe a tool for test–case generation using our CB2B model.

Chapter 5 demonstrated our tool for the representation and verification of a number of contracts derived from natural language descriptions, modelled as ECA rules and input to the CB2B model for contract compliance checking. It was shown that: (i) our representation of a contract model captures the contracts essential elements of business operations, role players, rights, obligations and prohibitions and handles CTDs and CTPs; (ii) with the help of the CB2B model, Spin automatically detects contract independent conflicts (discussed in Chapter 3), (iii) contract-dependent conflicts can be specified with relative ease in terms of contract elements as safety and liveness requirements which were verified by the Spin model checker.

We believe that the CB2B model can be used as an integral tool of a framework for building contractual applications. Within this context the CB2B model could play a key role operating as i) validator of logical inconsistencies of contracts and ii) generator of sequences of executions of contractual business operations. Execution sequences describe the behaviour encoded in the contract, thus they can be used for many purposes, including testing as discussed in Section 4.4. Also, they can be used for verifying conformance of the contract, for example, against i) an upgraded version of the contract expected to be equivalent ii) a business process that is supposed to implement the contractual interaction. In both cases the problem can be solved by means of comparison of the sequences generated by the contract written in CB2B model against the sequences under question.

6.2 Future work

The work presented here can be extended to provide comprehensive support for verification and validation of contracts. Three suggested areas are:

- Translation - The current work has not discussed how contract models written in our notation are translated into a real electronic contract, say, in the EROP language. It is assumed that this translation can be accomplished manually with relative ease, but it would be nice to have a translation tool that automates this process.
- IDE contract testing tool - The knowledge gained from this work can be used for building suitable modelling and testing tools for an integrated development environment for electronic contracting systems.
- Another open issue is how to handle large number of execution sequences. Spin —the model checker used in this dissertation— is not at its best in this regard. It generates each sequence as an independent file (a counter example) that it writes onto the disk. To exacerbate the problem, Spin generates a large number of duplicate counter examples.

Appendix A

In this appendix we list the macros that have been implemented to facilitate writing ECA rules and model check them using Spin and the CB2B model. In the macros listed below we use the following abbreviations in the macros variable and parameters. The arrow → indicates the contract entity from which the abbreviations has been derived. A brief description to the macros shown here can be found in Table 4.1 and Table 4.2 in Chapter 4, Section 4.4.

```
bo      --> Bsiness Operation.
r       --> Right.
o       --> Obligation.
p       --> Prohibition.
oblig   --> Obligation.
prohib  --> Prohibition.
role_pl --> Role Player.
rp      --> Role Player.
S       --> Success.
BF      --> Business Failure.
TF      --> Technical Failure.
TO      --> Timeout.
```

A.1 CB2B model macros

```
/****************************************/
/* BIS_OP typedef definition and macro  */
/* Note: the macro declares business    */
/* operation and business event using   */
/* the same parameter name.             */
/****************************************/
 typedef BIS_OP
 {
   byte name;
   byte role_pl;
   bool right;
   bool oblig;
   bool prohib;
   byte id;
   byte status;
 }
/***********************/
/* BIS_OP(name)  macro  */
/***********************/
 #define BIS_OP(name) BIS_OP name##_##bo; \
 mtype= { name }
/************************************/
/* Setting right, obligation and     */
/* prohibition macros.               */
/*                                   */
/* Note: the macros also use the     */
/* assert command from Promela to    */
/* assert for contract-independent   */
/* conflicts discussed in the thesis */
/************************************/

/***********************/
/* SET_R(bo,r)  macro   */
/***********************/
 inline SET_R(bo,r)
 {
   bo.right=r;
   assert(!(bo.right==1 && bo.oblig==1));
   assert(!(bo.right==1 && bo.prohib==1));
 }
 #define SET_R(name,r) \
 SET_R(name##_##bo,r)
```

```
/***********************/
/* SET_O(bo,o)  macro   */
/***********************/
 inline SET_O(bo,o)
 {
   bo.oblig=o;
   assert(!(bo.oblig ==1 && bo.prohib ==1));
   assert(!(bo.oblig ==1 && bo.right ==1));
 }
 #define SET_O(name,o) \
 SET_O(name##_##bo,o)

/***********************/
/*      SET_P(bo,p)     */
/***********************/
 inline SET_P(bo,p)
 {
   bo.prohib=p;
   assert(!(bo.prohib ==1 && bo.right ==1));
   assert(!(bo.prohib ==1 && bo.oblig ==1));
 }
 #define SET_P(name,p) \
 SET_P(name##_##bo,p)

/**********************************/
/* Inquiring for right, obligation   */
/* or prohibition macros.           */
/**********************************/

/***********************/
/*  IS_R(name,rp) macro */
/***********************/
 #define IS_R(name,rp) \
name##_##bo.right==1 && name##_##bo.role_pl==rp

/***********************/
/*  IS_O(name,rp) macro */
/***********************/
#define IS_O(name,rp) \
name##_##bo.oblig==1 && name##_##bo.role_pl==rp

/***********************/
/*  IS_P(name,rp) macro */
/***********************/
```

```
#define IS_P(name,rp) \
name##_##bo.prohib==1 && name##_##bo.role_pl==rp

/************************/
/*  SET_X(name,rp) macro */
/************************/
#define SET_X(name,rp) \
SET_1(rp##exTrace,name##_##bo.id)

/************************************/
/*  SYN(bo,rp) and NYS(bo) macros      */
/*  to test the execution history of   */
/*  business operation(bo) belong a    */
/*  particular RolePlayer (rp)         */
/************************************/
#define SYN(bo,rp) if:: \
((IS_X(bo,rp)))
#define NYS(bo) :: else skip \
fi;

/**********************************************/
/*  INIT(bo,r,o,p,rp) inline and macros.          */
/*                                                */
/*  The parameters are business operation bo and  */
/*  initial right, obligation or prohibition and  */
/*  the role player.                              */
/**********************************************/

 inline INIT_WITH_5(bo,rp,r,o,p)
 {
  d_step
  {
      _counter_=_counter_+1;
      bo.role_pl=rp;
      bo.right=r;
      bo.oblig=o;
      bo.prohib=p;
      bo.id=_counter_;
  }
 }

/**********************************************/
/*  INIT(bo,rp) inline and macros.                */
/*                                                */
/*  The parameters are business operation bo,     */
/*  initial right, obligation or prohibition and   */
/*  the role player.                               */
/*************************************************/
 inline INIT_WITH_2(bo,rp)
 {
  d_step
  {
     _counter_=_counter_+1;
     bo.role_pl=rp;
     bo.executed=0;
     bo.id=_counter_;
  }
 }
/*************************************************/
/*   The macros for INIT_WITH_5(bo,rp,r,o,p) and   */
/*   INIT_WITH_2(bo,rp)                            */
/*************************************************/
 #define INIT(types...) _INIT_N(gnu_count(types),types)
 #define _INIT_N(n,types...) _INIT(n,types)
 #define _INIT(n,types...) _INIT_##n(types)
 #define _INIT_2(x,y) INIT_WITH_2( x##_##bo,y )
 #define _INIT_5(a,b,c,d,f) INIT_WITH_5( a##_##bo,b,c,d,f )

/******************************************************/
/*   Macros/inlines for different purposes for CB2B model */
/******************************************************/

/* To use WHEN instead of if as in EROP language */
#define WHEN if


/* To test if the status of business event is Succeeded */
#define SC(name) \
name##_##bo.status==S

/* To test if the status of business event is Business failure */
#define BF(name) \
name##_##bo.status==BF

/* To test if the status of business event is Technical failure */
#define TF(name) \
name##_##bo.status==TF

/* To test if the status of business event is Time-Out */
```

```
#define TO(name) \
name##_##bo.status==TO

/* To test the event conditions */
#define EVENT(name,msg1,msg2) \
((msg1==1)&&(msg2==1));

/*****************************/
/*   Rule decision inline and  */
/*   macro                     */
/*****************************/
inline RD(rp,msg1,msg2)
{
  CRM2BEG! msg1(msg2);
}
#define RD(name,rp,msg1,msg2) \
RD(rp,msg1,msg2) \

/*****************************************/
/*      Test if the business event         */
/*   is Right, Obligation or Prohibition   */
/*                                         */
/*****************************************/
#define bizEvent(name) \
name##_##bo.right==1||name##_##bo.oblig==1
                    ||name##_##bo.prohib==1

/*****************************************/
/*   Macro to set the status of business   */
/*****************************************/
inline SET_STATUS(bo,stat){
  bo.status=stat;
}
#define SET_STATUS(name,stat) \
name##_##bo.status=stat

/***************************/
/*   Macro\ inline DONE     */
/* resets global variable   */
/* _counter_ and resets     */
/* execution trace of Role  */
/* Player (rp)              */
/***************************/
inline DONE(rp)
```

```
{
 _counter_= 0;
}
#define DONE(rp) \
DONE(rp##exTrace); \
SET_ALL_0(rp##exTrace);
/**************************************/
/* When_Event(name) inline and macro, */
/* this will be used by CONTRACT(name)*/
/* macro. This considers S, TO status */
/* of business events only.           */
/**************************************/
inline When_Event(name){
 if
  ::BEG2CRM ? [name,S]  -> BEG2CRM ? _,_
  ::BEG2CRM ? [name,TO]  -> BEG2CRM ? _,_
 fi;
}
#define When_Event(name) \
When_Event(name)

/*********************************************/
/* When_Event(name, status) inline and       */
/* macro. This is a another implementation   */
/* to When_Event. It takes any event         */
/* with any status. Current implementation   */
/* of CB2B uses When_Event(name) shown above */
/*********************************************/
inline When_Event(name,status){
  BEG2CRM ? [name,status]  -> BEG2CRM ? _,_
}
#define When_Event(name,status) \
When_Event(name,status)

/**********************************/
/* CONTRACT(name) macro, this     */
/* used to block executing the    */
/* contract rules until their     */
/* corresponding business events  */
/* are emerged.                   */
/**********************************/
#define CONTRACT(name) \
When_Event(name); \
ContractRule(name)
```

```
/*****************************************/
/*   Macro to generate business events     */
/*****************************************/
#define B_E(rp,name,status) \
bizEvent(name); \
SET_STATUS(name,status); \
sendEvent(name,status); \
RuleDecision(rp);

/***********************************************/
/*      As in gcc's testsuite.This counts the       */
/*   number of arguments between 1-10              */
/***********************************************/

#define gnu_count(y...)   _gnu_count1 ( , ##y)
#define _gnu_count1(y...) _gnu_count2 (y,10,9,8,7,6,5,4,3,2,1,0)
#define _gnu_count2(_,x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,n,ys...) n

/****************************************************/
/*     This to declare messages used by the CB2B as      */
/*                  Promela mtype                        */
/****************************************************/
#define RuleMessage(types...) _TYPES_TO_MTYPE_N(gnu_count(types)
        ,types)
#define _TYPES_TO_MTYPE_N(n,types...) _TYPES_TO_MTYPE(n,types)
#define _TYPES_TO_MTYPE(n,types...) _TYPES_TO_MTYPE_##n(types)
#define _TYPES_TO_MTYPE_0()
#define _TYPES_TO_MTYPE_1(x) mtype={x}
#define _TYPES_TO_MTYPE_2(x,y) mtype={x,y}
#define _TYPES_TO_MTYPE_3(x,y,z) mtype={x,y,z}
#define _TYPES_TO_MTYPE_4(a,b,c,d) mtype{a,b,c,d}

/*********************************************/
/*  To define role player - SIMILAR TO THE ABOVE */
/*********************************************/
#define RolePlayer(types...) _ROLE_PLAYER_N(gnu_count(types)
        ,types)
#define _ROLE_PLAYER_N(n,types...) _ROLE_PLAYER(n,types)
#define _ROLE_PLAYER(n,types...) _ROLE_PLAYER_##n(types)
#define _ROLE_PLAYER_0()
#define _ROLE_PLAYER_1(a) mtype={a} \
BITV_32 a##exTrace;
#define _ROLE_PLAYER_2(a,b) mtype={a,b} \
BITV_32 a##exTrace; \
BITV_32 b##exTrace;
```

```
#define _ROLE_PLAYER_3(a,b,c) mtype={a,b,c} \
BITV_32 a##exTrace; \
BITV_32 b##exTrace; \
BITV_32 c##exTrace;
#define _ROLE_PLAYER_4(a,b,c,d) mtype={a,b,c,d} \
BITV_32 a##exTrace; \
BITV_32 b##exTrace; \
BITV_32 c##exTrace; \
BITV_32 d##exTrace;

/*******************************************************/
/* DEF_BizOperStatus to define possible status to our   */
/*        operations TAKES  BETWEEN 1-4 ARG             */
/*******************************************************/
#define DEF_BizOperStatus(types...) _TYPES_TO_STATUS_N(gnu_count
        (types) ,types)
#define _TYPES_TO_STATUS_N(n,types...) _TYPES_TO_STATUS(n
        ,types)
#define _TYPES_TO_STATUS(n,types...) _TYPES_TO_STATUS_##n
       (types)
#define _TYPES_TO_STATUS_0()
#define _TYPES_TO_STATUS_1(x) mtype={x}
#define _TYPES_TO_STATUS_2(x,y) mtype={x,y}
#define _TYPES_TO_STATUS_3(x,y,z) mtype={x,y,z}
#define _TYPES_TO_STATUS_4(a,b,c,d) mtype{a,b,c,d}

/*******************************************************/
/*         To Define contract rule as inlines           */
/*              TAKES  BETWEEN 1-4 ARG                  */
/*******************************************************/
#define RULE(types...) _TYPES_TO_INLINE_N(gnu_count
        (types),types)
#define _TYPES_TO_INLINE_N(n,types...) _TYPES_TO_INLINE(n
        ,types)
#define _TYPES_TO_INLINE(n,types...) _TYPES_TO_INLINE_##n
        (types)
#define _TYPES_TO_INLINE_0()
#define _TYPES_TO_INLINE_1(x) inline x##_##nil()
#define _TYPES_TO_INLINE_2(x,y) inline x##_##y()
#define _TYPES_TO_INLINE_3(x,y,z) inline x##_##y##_z()
#define _TYPES_TO_INLINE_4(a,b,c,d) inline a##_##b##_##c
        ##_##d()

/******************************************************/
```

```
/*  ContractRule is called by CONTRACT(name) macro     */
/*  to execute the rules correspond to the business    */
/*  correspond to the parameter name of CONTRACT(name) */
/*  macro.                                              */
/*******************************************************/
#define ContractRule(types...) _TYPES_TO_EX_INLINE_N(gnu_count
       (types),types)
#define _TYPES_TO_EX_INLINE_N(n,types...) _TYPES_TO_EX_INLINE(n
        ,types)
#define _TYPES_TO_EX_INLINE(n,types...) _TYPES_TO_EX_INLINE_##n
        (types)
#define _TYPES_TO_EX_INLINE_0()
#define _TYPES_TO_EX_INLINE_1(x) x##_##nil()
#define _TYPES_TO_EX_INLINE_2(x,y) x##_##y()
#define _TYPES_TO_EX_INLINE_3(x,y,z) x##_##y##_z()
#define _TYPES_TO_EX_INLINE_4(a,b,c,d) a##_##b##_##c##_##d()
```

A.2 External macros

```
/*
 *  From Towards Efficient Model Checking,
 *  Phd Dissertation by Theo Ruys 2001, chapter 4
 */

#define BITV_U(x,n) unsigned x : n
#define BITV_8 byte
#define BITV_16 short
#define BITV_32 int

/* this const is 0111....111 */
#define ALL_1S 2147483647

/* set bit i to 0 and 1, respectively */
#define SET_0(bv,i) bv=bv&(~(1<<i))
#define SET_1(bv,i) bv=bv|(1<<i)

/* set all bit to 0 and 1, respectively */
#define SET_ALL_0(bv) bv=0
#define SET_ALL_1(bv,n) bv=ALL_1S>>(31-n)

/* is bit i 0 or 1, respectively */
#define IS_0(bv,i) (!(bv&(1<<i)))
#define IS_1(bv,i) (bv&(1<<i))
```

```
/* Ben-Ari version from Principles of the Spin Model Checker
 * Springer 2008, p 192.
 * #define IS_1(bv,i) (bv >> i & 1)
 */
```

A.3 Global declaration for CB2B model

```
/***************************************************/
/* Global variables, channels, macros used by CB2B */
/***************************************************/
#define setting() \
#define YES           1
#define NO            0
#define TRUE          1
#define FALSE         0
#define ACCEPT        1
#define REJECT        0

/* Global variable used by INIT macro */
   byte _counter_=0;

/* Channel to send event from BEG to CRM */
   chan BEG2CRM = [1] of {mtype,mtype};

/* Channel to send rule decision from CRM to BEG */
   chan CRM2BEG = [0] of {mtype,mtype};

/* Messages used by the CB2B model for communication */
   RuleMessage(CC,CCR,CCO,CCP,NCC,CO,OCX,CNL,CND,RST);
```

References

[1] A. Abdelsadiq, C. Molina-Jimenez, and S. Shrivastava. “A high-level model-checking tool for verifying service agreements”. In: *Proceedings of the 6th IEEE International Symposium on Service Oriented System Engineering*. SOSE ’11. Irvine, CA, 2011, pp. 297 –304.

[2] A. Abdelsadiq, C. Molina-Jimenez, and S. Shrivastava. “On Model Checker Based Testing of Electronic Contracting Systems”. In: *Proceedings of the 12th IEEE Conf. on Commerce and Enterprise Computing*. CEC ’10. Shanghai, China, 2010, pp. 88 –95.

[3] L. Alessio and R. Franco. “Mcmas: A model checker for multi-agent systems”. In: *Proceedings of TACAS*. Springer Verlag, 2006, pp. 450–454.

[4] L. Alessio, Q. Hongyang, and S. Monika. “Towards Verifying Contract Regulated Service Composition”. In: *Web Services, IEEE International Conf. on* 0 (2008), pp. 254–261.

[5] B. F. Chellas. *Modal Logic: An Introduction*. Cambridge: Cambridge University Press, 1980.

[6] K.W. Chiu, S. C. Cheung, and T. Sven. “A Three-Layer Architecture for E-Contract Enforcement in an E-Service Environment”. In: *Hawaii International Conf. on System Sciences* 3 (2003), 74a.

[7] A. Cimatti et al. “NUSMV: A New Symbolic Model Verifier”. In: *Proceddings of the 11th International Conf. on Computer Aided Verification*. London, UK: Springer-Verlag, 1999, pp. 495–499.

[8] J. Claude and J. Thierry. “TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems”. In: *International Journal on Software Tools for Technology Transfer* 7.4 (2005), pp. 297–315.

[9] A. Daskalopulu. “Model Checking Contractual Protocols”. In: *CoRR* cs.SE/0106009 (2001).

[10] A. Daskalopulu and T. Maibaum. “Towards Electronic Contract Performance”. In: *Proceedings of the 12th Int. Workshop on Database and Expert Systems Applications*. DEXA ’01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 771–.

[11] *Drools*. JBoss. URL: http://www.jboss.org/drools/.

[12] N. Dunlop, J. Indulska, and K. Raymond. “Methods for Conflict Resolution in Policy-Based Management Systems”. In: *Proceedings of the 7th International Conf. on Enterprise Distributed Object Computing*. EDOC ’03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 98–.

[13] *ebXML: Business Process Spec. Spec. v2.0.4. http://docs.oasisopen.org/ ebxml-bp/2.0.4/OS/spec/ebxmlbp-v2.0.4-Spec-os-en.pdf*. 2006.

[14] Gregorio D. Enrique M., M. Emilia Cambronero, and Gerardo Schneider. “A Model for Visual Specification of E-contracts”. In: *Services Computing, IEEE International Conf. on* 0 (2010), pp. 1–8.

[15] Gordon F. and Franz W. “Creating Test-Cases Incrementally with Model-Checkers”. In: *GI Jahrestagung (2)*. 2007, pp. 381–386.

[16] Gordon F. and Franz W. “Redundancy based test-suite reduction”. In: *Proceedings of the 10th international conference on Fundamental approaches to software engineering*. FASE’07. Braga, Portugal: Springer-Verlag, 2007, pp. 291–305. ISBN: 978-3-540-71288-6.

[17] Gordon F., Franz W., and Paul A. “Issues in using model checkers for test case generation”. In: *J. Syst. Softw.* 82.9 (Sept. 2009), pp. 1403–1418. ISSN: 0164-1212.

[18] S. Fenech, G. J. Pace, and G. Schneider. “Automatic Conflict Detection on Contracts”. In: *Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing*. ICTAC ’09. Kuala Lumpur, Malaysia: Springer-Verlag, 2009, pp. 200–214.

[19] S. Fenech, G. J. Pace, and G. Schneider. “CLAN: A Tool for Contract Analysis and Conflict Discovery”. In: *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis*. ATVA ’09. Macao, China: Springer-Verlag, 2009, pp. 90–96.

[20] M. Fisher. “Temporal Logic”. In: John Wiley and Sons, Ltd, 2011, pp. 9–48.

[21] X. Fu, T. Bultan, and J. Su. “Analysis of interacting BPEL web services”. In: *Proceedings of the 13th international Conf. on World Wide Web*. WWW ’04. New York, NY, USA: ACM, 2004, pp. 621–630.

[22] Patrice G. “Model checking for programming languages using VeriSoft”. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL ’97. Paris, France: ACM, 1997, pp. 174–186.

[23] G. K. Giannikis and A. Daskalopulu. “Normative conflicts in electronic contracts”. In: *Electron. Commer. Rec. Appl.* 10.2 (2011), pp. 247–267.

[24] A. Gill. *Introduction to the Theory of Finite State Machines*. New York: McGraw-Hill, 1962.

[25] F. Gordon, W. Franz, and A. Paul. “Testing with model checkers: a survey”. In: *Softw. Test., Verif. Reliab.* 19.3 (2009), pp. 215–261.

[26] B.n N. Grosof, Y. Labrou, and H. Y. Chan. “A declarative approach to business rules in contracts: courteous logic programs in XML”. In: *Proceedings of the 1st ACM Conf. on Electronic commerce.* EC ’99. Denver, Colorado, United States: ACM, 1999, pp. 68–77.

[27] G. Holzmann. *The Spin model checker: primer and reference manual.* Addison-Wesley Professional, 2003.

[28] G. J. Holzmann. *Design and validation of computer protocols.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1991.

[29] G. J. Holzmann. “The model checker SPIN”. In: *IEEE Transactions on Software Engineering* 23 (1997), pp. 279–295.

[30] *http://www.rosettanet.org/*. 2002.

[31] P. R. Krishna, K. Karlapalem, and A. R. Dani. “From Contracts to E-Contracts: Modeling and Enactment”. In: *Inf. Technol. and Management* 6 (4 2005), pp. 363–387.

[32] E. C. Lupu and M. Sloman. “Conflicts in Policy-Based Distributed Systems Management”. In: *IEEE Trans. Softw. Eng.* 25.6 (1999), pp. 852–869.

[33] Jeff Magee and Jeff Kramer. *Concurrency - state models and Java programs.* Wiley, 1999, pp. I–XIII, 1–355.

[34] E. Martinez et al. “A Model for Visual Specification of E-contracts”. In: *Proceedings of the 2010 IEEE International Conf. on Services Computing.* SCC ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–8.

[35] L. I. Millett and T. Teitelbaum. “Issues in Slicing PROMELA and Its Applications to Model Checking, Protocol Understanding, and Simulation”. In: *STTT* 2.4 (2000), pp. 343–349.

[36] C. Molina-Jimenez and S. Shrivastava. “Model checking correctness properties of a middleware service for contract compliance”. In: *Proceedings of the 4th International Workshop on Middleware for Service Oriented Computing*. MWSOC ’09. Urbana Champaign, Illinois: ACM, 2009, pp. 13–18.

[37] 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)*. Jul 20–23, Vienna, Austria: IEEE CS, 2009, pp. 65–73.

[38] Carlos Molina-Jimenez, Santosh Shrivastava, and Massimo Strano. “A Model for Checking Contractual Compliance of Business Interactions”. In: *IEEE Transactions on Services Computing* 5.2 (2012), pp. 276–289. ISSN: 1939-1374. DOI: http://doi.ieeecomputersociety.org/10.1109/TSC.2011.37.

[39] T. Murata. “Petri Nets: Properties, Analysis and Applications.” In: *Proceedings of the IEEE* 77.4 (1989), pp. 541–580.

[40] S. Owicki and L. Lamport. “Proving Liveness Properties of Concurrent Programs”. In: *ACM Trans. Program. Lang. Syst.* 4.3 (1982), pp. 455–495. ISSN: 0164-0925.

[41] G. Pace, C. Prisacariu, and G. Schneider. “Model checking contracts: a case study”. In: *Proceedings of the 5th international Conf. on Automated technology for verification and analysis*. ATVA’07. Tokyo, Japan: Springer-Verlag, 2007, pp. 82–97.

[42] G. Paul et al. *CrossFlow: Cross-Organizational Workflow Management in Dynamic Virtual Enterprises*. Enschede, 2000.

[43] M. Pezze and M. Young. *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, 2008.

[44] A. Pretschner et al. “One evaluation of model-based testing and its automation”. In: *Proceedings of the 27th international Conf. on Software engineering*. ICSE ’05. St. Louis, MO, USA: ACM, 2005, pp. 392–401.

[45] C. Prisacariu and G. Schneider. “A formal language for electronic contracts”. In: *Proceedings of the 9th IFIP WG 6.1 international Conf. on Formal methods for open object-based distributed systems*. FMOODS’07. Paphos, Cyprus: Springer-Verlag, 2007, pp. 174–189.

[46] S. Rayadurgam and M. P. E. Heimdahl. “Test-Sequence Generation from Formal Requirement Models”. In: *The 6th IEEE International Symposium on High-Assurance Systems Engineering: Special Topic: Impact of Networking*. HASE ’01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 23–31.

[47] M. Rouached, O. Perrin, and C. Godart. “A Contract Layered Architecture for Regulating Cross-Organisational Business Processes”. English. In: *3rd International Conf. on Business Process Management - BPM 2005*. Ed. by Wil M. P. van der Aalst et al. Vol. 3649. Nancy/France: Springer-Verlag GmbH, 2005, pp. 410–415.

[48] T. C. Ruys. “Towards Effective Model Checking”. PhD thesis. Enschede: University of Twente, 2001.

[49] E. Solaiman, C. Molina-jimenez, and S. Shrivastava. “Model Checking Correctness Properties of Electronic Contracts”. In: *In proceedings of the International Conf. on Service Oriented Computing (ICSOC03)*. Springer-Verlag, 2003, pp. 303–318.

[50] M. Strano, C. Molina-Jimenez, and S. Shrivastava. “A Rule-Based Notation to Specify Executable Electronic Contracts”. In: *Proceedings of the International Symposium on Rule Representation, Interchange and Reasoning on the Web.* RuleML ’08. Orlando, Florida: Springer-Verlag, 2008, pp. 81–88.

[51] M. Strano, C. Molina-Jimenez, and S. Shrivastava. “Implementing a Rule–Based Contract Compliance Checker”. In: *Proceedings of the 9th IFIP Conf. on e-Business, e-Services, and e-Society (I3E’2009).* Nancy, France: Springer, 2009, pp. 96–111.

[52] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan–Kaufmann, 2006.

[53] W. Wang et al. “E-process design and assurance using model checking”. In: *Computer* 33.10 (2000), pp. 48 –53.

[54] M. Young and M. Pezze. *Software Testing and Analysis: Process, Principles and Techniques.* John Wiley & Sons, 2005.