Model Checking Correctness Properties of Electronic Contracts

Ellis Solaiman, Carlos Molina-Jimenez, and Santosh Shrivastava

School of Computing Science, University of Newcastle upon Tyne,
Newcastle upon Tyne, NE1 7RU, England
{Ellis.Solaiman,Carlos.Molina,Santosh.Shrivastava}@ncl.ac.uk

Abstract. Converting a conventional contract into an electronic equivalent is not trivial. The difficulties are caused by the ambiguities that the original human-oriented text is likely to contain. In order to detect and remove these ambiguities the contract needs to be described in a mathematically precise notation before the description can be subjected to rigorous analysis. This paper identifies and discusses a list of correctness requirements that a typical executable business contract should satisfy. Next the paper shows how relevant parts of standard conventional contracts can be described by means of Finite State Machines (FSMs). Such a description can then be subjected to model checking. The paper demonstrates this using Promela language and the Spin validator.

Keywords: Contract, electronic contract, finite state machine, contract representation, contract enforcement, model-checking, validation, correctness requirements, safety and liveness properties.

1 Introduction

A *conventional contract* is a paper document written in English or other natural language that stipulates that two or more signatory parties agree to observe the clauses stipulated in the document. An *executable contract (x-contract)* is the electronic version of a conventional contract that can be enacted by a contract management system to enforce what the English text contract stipulates. The purpose of both conventional and electronic contracts is the same: enforcement of the rights and obligations of the contracting parties. However, there is a crucial difference between the two kinds of contract. A conventional contract is human oriented. Thus, it is likely to contain ambiguities in the text that are detected and interpreted by humans when the contract is performed; whereas an x-contract is computer oriented; consequently, it tolerates no inconsistencies. According to our findings, contract inconsistencies can be categorized into two groups. (i) Internal enterprise policies that conflict with contract clauses. (ii) Inconsistencies in the clauses of the contract. In our view, and to gain in simplicity, these two issues can be treated separately. In this paper we address the second issue.

We have observed that inconsistencies in the clauses of conventional contracts are normal rather than exceptional, for this reason the logical consistency of a conven-

M.E. Orlowska et al. (Eds.): ICSOC 2003, LNCS 2910, pp. 303–318, 2003.
© Springer-Verlag Berlin Heidelberg 2003

tional contract should be proven by some means before implementing it as an executable contract.

The question that we attempt to answer in this paper is what are the correctness requirements that a typical contract should satisfy and how can they be validated? The paper is organised as follows: In Section 2 we discuss the differences between our approach to validating contracts and related research work. In Section 3, we provide a list of what we consider the most common correctness requirements for business contracts and classify them into conventional safety and liveness properties. In Section 4 we briefly discuss our contract model which is based on finite state machines. In Section 5 we illustrate with examples how Spin can be used for validating correctness requirements. Finally, we draw some conclusions in Section 6.

2 Related Work

In this section we will summarise the essential ideas behind three works that we consider to be close to the research work of this paper.

In the work of Milosevic et. al. [1] [2] a contract is informally defined as a set of policy statements that specify constraints in terms of permissions, prohibitions and obligations for roles involved in the contract. A role (precisely, a role player) is an entity (for example a human being, machine, program, etc.) that can perform an action. Formally, each policy statement is specified in deontic logic constraints [3].Thus each deontic constraint precisely defines the permissions, prohibitions, obligations, actions, and temporal and non-temporal conditions that a role needs to fulfil to satisfy an expected behaviour.

For example, a constraint can formally specify that, “Bob is obliged to deliver a box of chocolates to Alice’s desk every weekday except on Wednesdays for three years, between 9 and 9:15 am, commencing on the 1st of Jan 2004”. The expressiveness of deontic notation allows the contract designer to verify temporal and deontic inconsistencies in the contract. The authors of this approach argue that it is possible to build verification software to visually show that, Bob’s obligations do not overlap or conflict. Such verification mechanisms would easily detect a conflicting situation where Bob has to deliver a box of chocolates to Alice’s desk and to Claire’s who works miles away from Alice’s desk. Similarly, the verifier would detect that Bob is not obliged and prohibited to deliver chocolates to Alice during the same period of time.

Another research work of relevance to ours is the EDEE system. EDEE provides a framework for representing, storing and enforcing business contracts [4]. In EDEE a contract is informally conceived as a set of provisions. In legal parlance, a provision is an arrangement in a legal document, thus in EDEE a provision specifies an obligation, prohibition, privilege or power (rights). An example of a provision is “Alice is obliged to pay Bob 20 cents before 1st Jan 2004”. Central to EDEE is the concept of occurrence. An occurrence is a time-delimited relationship between entities. It can be regarded as a participant-occurrence-role triple that contain the name of the participants of the occurrence, the name of the occurrence and the name of the roles involved in the occurrence. An example of an occurrence that involves Alice (the payer)

and Bob (the payee) is "Alice is paying Bob 20 cents on 31st Dec 2003." The formal specification of a contract in EDEE is obtained by translating the set of informal provisions derived from the clauses of the contract into a set of formal occurrences. Another basic concept in EDEE is query. A query is a request for items satisfying certain criteria (for example, "Payments performed by Alice before 31st Dec 2003"). At implementation level, the occurrences representing the contract provisions are stored together with queries and new occurrences in an occurrence store in SQL views.

Business operations invoked by the contractual parties are seen as occurrences intercepted and passed through the occurrence store where they are analysed to see if they satisfy the contractual occurrences associated with the operations. EDEE has been provided with some means for detecting contract inconsistencies. To detect overlap between queries (a set of occurrences being both prohibited and permitted, a set of occurrences being obliged and prohibited, etc.) the authors of EDEE rely on a locally implemented coverage-checking algorithms.

Of relevance to our research is also the Ponder language [5]. Ponder is a declarative language that permits the specification of policies for managing a distributed system or contractual service level agreements between business partners. Ponder specifies policies in terms of obligations, permissions and prohibitions and provides means for defining roles and relationships. To detect and prevent policy conflicts such as conflict for a given resource or overlapping of duties, Ponder's notation permits the specification of semantic constraints that limit the applicability of a given policy in accordance with person playing the role, time, or state of the system.

A common pattern of the related works discussed above is that all of them rely on elaborate logical notations that include temporal constraints and role players in their parameters. The expectation is that this notation should be able to specify arbitrarily complex business contracts and detect all kind of inconsistencies. This generality is certainly desirable; however, because of the complexity of the problem it might be rather ambitious. We believe that a modular approach is more realistic for detecting contract ambiguities. For that to be possible, we need to be able to identify and isolate the different sources of possible inconsistencies in business contracts.

In our business model [6] enterprises that engage in contractual relationships are autonomous and wish to remain autonomous after signing a contract. Thus a signing enterprise has its own resources and local policies. In our view each contracting enterprise is a black box where private business processes represented as finite state machines, workflows or similar automaton, run. A private business process interacts with its external environment through the contract from time to time to influence the course of the shared business process. Thus, a contract is a mechanism that is conceptually located in the middle of the interacting enterprises to intercept all the contractual operations that the parties try to perform. Intercepted operations are accepted or rejected in accordance with the contract clauses and role players' authentication.

From this perspective, we can identify two fairly independent sources of contract inconsistencies:

- Internal enterprise policies conflicting with contractual clauses.
- Inconsistencies in the clauses of the contract.

It is our view that these two issues should be treated separately rather than encumbering a contract model with excessive notation (details, concepts and information) that might be extremely difficult to validate. Such a separation is not considered in the

work discussed above. In this paper we address only the second issue, that is, we are concerned only with the cooperative behaviour of business enterprises and not their internal structure.

Our approach is to represent business interactions as finite state machines. Use of finite state machines for representing such interactions has been proposed for Web services (Web service conversation language, WSCL [7]). We note that inter-organisation business interactions, PIPs (partner interaction processes) as specified in Rosettanet industrial consortium [8] can also be represented as finite state machines.

In our business model each contracting enterprise has the privilege and responsibility of verifying that its internal policies do not conflict with the clauses of the contract. Similarly, each enterprise exercises its independence to choose the roles players that would invoke operations on the contract and provide them with a proper contract role player certificate (a cryptographic key for example). Consequently, it is the responsibility of each enterprise to prevent inconsistencies with role players such as duty overlapping, duty separation, etc.

In our contract model we intentionally leave the notion of role players out of the game. However, we assume they are authenticated by the contract management system before they are allowed to perform operations of the FSMs. It can be argued that our FSM model is less expressive in comparison with the related works discussed above. However we believe that its expressiveness is good enough for modeling a wide variety of business interactions. Our model is simple. Thanks to this simplicity we can rely on widely used of-the-shelf model checkers like Spin [9] to validate general safety and liveness properties of contracts, relatively easily. We have to admit that so far he have modeled static contracts (contracts whose clauses do not change once the contract is signed), it remains to be seen whether we can use the same paradigm for describing complex contracts where the clauses change and the signing parties join and leave while the contract is in execution. This is a topic for further research.

3 Common Correctness Requirements

Knowing the correctness requirements of an x-contract at design time is crucial as an x-contract can be proven correct only with respect to a specific list of correctness requirements. It is sensible to think, that different contract users would be interested in being assured of the correctness of different parts of a given contract. On the other hand, the parts of a contract that more likely contain logical inconsistencies vary from contract to contract. Because of this, it is too ambitious to intend to identify a complete list of correctness requirements for business contracts. However, it is possible to provide a list of fairly standard correctness requirements and to generalise them. The list provided below, is the result of analyzing several traditional business contracts. Hopefully, this generalisation will help designers of x-contracts reason about correctness requirements of x-contracts in terms of conventional and well understood terminology such as correct termination, deadlocks, etc. In the following list CR stands for correctness requirement:

CR1: Correct commencement: An x-contract should start its execution in a well-defined initial state on a specific date or when something happens. This correctness requirement is a special case and cannot be guaranteed by the x-contract itself but by the human being or system (software or hardware) that triggers the execution of the x-contract.

CR2: **Correct termination:** An x-contract should reach a well-defined termination state on a specific date or when something happens. For example, the x-contract terminates on the 31st of Dec 2005 or the x-contract terminates when the purchaser delivers 500 cars.

CR3: **Attainability**: Each and every state within an x-contract should be attainable, i.e. executable at least in one of the execution paths of the x-contract.

CR4: **Freedom from deadlocks:** An x-contract should never enter a situation in which no further progress is possible. For example, an x-contract should not make a supplier wait for a payment before sending an item to the purchaser while the purchaser is waiting for the item before sending the payment to the supplier.

CR5: **Partial correctness:** If an x-contract begins its execution with a precondition true then, the x-contract will never terminate (normally or abnormally) with the precondition false, regardless of the path followed by the x-contract from the initial to its final state. For example, if the amount of money borrowed by a customer from a bank is *Debt= 0* at the beginning of the x-contract, the x-contract cannot be closed unless *Debt=0*.

CR6: **Invariant:** If an x-contract begins its execution with a precondition true then, the precondition should remain true for the whole duration of the contract. A slight variation of this correctness requirement would be a requirement that the precondition remains true only or at least during certain parts of the execution of the x-contract. To mention an example we can think that an x-contract between a banker and a customer stipulates that the amount of money borrowed by the customer should never exceed the customer's credit limit.

CR7: **Occurrence or accessibility:** A given activity should be performed by an x-contract at least once no matter what execution path the x-contract performs. A slight variation of this requirement is one that demands that a certain activity should be performed infinitely often. For example, an x-contract between a bank and a customer should guarantee that the customer will receive bank statements at least once a month.

CR8: **Precedence:** An x-contract can perform a certain activity only if a given condition is satisfied. For example, the lend period of a book in the possession of a student should not be extended unless the waiting list for the book is empty.

CR9: **Absence of livelocks:** The execution of an x-contract should not loop infinitely through a sequence of steps that has been identified as undesirable, presumably because the sequence produces undesirable output or no output at all. For example, an x-contract between an auctioneer and a group of bidders should not allow one of the bidders to place his bids infinitely often and leave the rest of the bidders bid-starving. This correctness requirement is also known as *fairness* or *absence of individual starvation*.

CR10: **Responsiveness:** The request for a service will be answered before a finite amount of time. For example, an x-contract should guarantee that a buyer responds to every offer from a client in less than five days.

CR11: **Absence of unsolicited responses**: An x-contract should not allow a contractual party to send unsolicited responses. For example, an x-contract between a banker and a customer should not allow the banker to send unsolicited advertisement to the customer.

3.1 Model-Based Validation of Correctness Requirements

Model-based validation is widely used for validating correctness requirements. This approach relies on the use of software tools that are known as model checkers. The core idea behind this approach is to use model-checking algorithms to determine [Spin-Book-chapter11], whether the contract model (a finite state transition system) satisfies a list of correctness requirements. The correctness requirements are specified as safety and liveness properties translated into temporal logics or regular expressions. We discuss safety and liveness properties thoroughly in the Section 3.2. Model-based validation is a compromise between bare-eye inspection and mathematical proof and works well for distributed applications of moderate complexity. For this reason from here on we will focus our discussion on model-based validation and leave bare-eye inspection and mathematical proof aside.

3.2 Safety and Liveness Properties

Informally we can define safety and liveness properties as follows: a *property* is a quality of a programme that holds true for every possible execution of the program.
Properties are expressed as statements. A *safety property* is a statement that claims that something will not happen. In other words, a safety property is a claim that a programme will never perform a given activity (for example, send $message_j$ before $message_i$) presumably, because the activity is bad, that is, undesirable. Similarly, a *liveness property* is a statement that claims that something will eventually happen. In other words, a liveness property dictates that a programme will eventually perform a given activity (for example, send the sequence of messages $message_i$, $message_j$, $message_k$), presumably because the activity is good and desirable.

On the ground of our own experience with x-contract validation we argue that most, if not all, correctness requirements of traditional business contracts can be readily expressed either as safety or liveness properties. With the intention of giving the designer of an electronic contract some guidance about the kind of correctness requirement he/she is faced with, we will classify into safety and liveness properties the list of typical correctness requirements of electronic business contracts provided in Section 3:

- Safety properties: attainability, partial correctness, invariant, deadlocks, precedence, absence of unsolicited responses.
- Liveness properties: correct termination, occurrence, livelocks, responsiveness.

We are aware that it has been shown that not all correctness requirements can be readily classified as either safety or liveness property [10]. Fortunately, it has been formally proven that any correctness property can be represented as the intersection of a safety property and a liveness property [11]. The idea behind our approach is that a

complex correctness requirement demanded by a signing party can always be expressed as a combination of a number of the basic correctness requirements listed in Section 3.

4 Representation of Contracts by Means of FSMs

A contract can be represented as a set of FSMs, one for each of the contracting parties that interact with each other. Conceptually, we can assume that a FSM is located within each contracting party and that these FSMs communicate with each other through communication channels. Each entry in a contract is called a *term* or a *clause*. The clauses of a contract stipulate how the signing parties are expected to behave. In other words, they list the rights and obligations of each signing party. The rights and obligations stipulated in a contract can be abstracted and grouped into a set of Rights (R) and a set of Obligations (O). The sets R and O can be mapped into the events and the operations that the x-contract involves.

Fig. 1, shows the graphical representation of x-contracts we use in this paper, where *e* and *o* stand for event and operation, respectively (a null operation will be represented by ε). Thus *e* are business events, and *o* are business operations.
Any event can be triggered by a decision taken internally within the enterprise in which the event is to be performed (for example the purchaser exercising the right of deciding to send a purchase order), or by an operation performed externally within another enterprise (for example when the supplier wants to offer a new item to the purchaser).

The lines between the finite state machines in Fig. 1 indicate events being triggered by external operations. For example the event *p* was triggered at the purchaser's side when the Supplier exercised the right of performing operation *O1*.
The supplier's FSM will allow the supplier to execute only the operations he has the right to execute and nothing else. Likewise, the FSM enforces the supplier to execute the operations he has the obligation to execute. The purchaser's FSM works in a similar way.

For more details on representing contracts as FSMs, we refer the reader to [6].

5 Validation of Correctness Requirements with Spin

Spin is a model checker that has gained a wide acceptance. Spin validates safety and liveness properties of models coded in the Pomela modelling language. The Spin toolkit is freely available, and includes a simulator and a validator.

5.1 Spin Verification Tools

Spin comes with a graphical user interface called XSpin which can be used to edit Promela code, and to run the simulator and the validator.

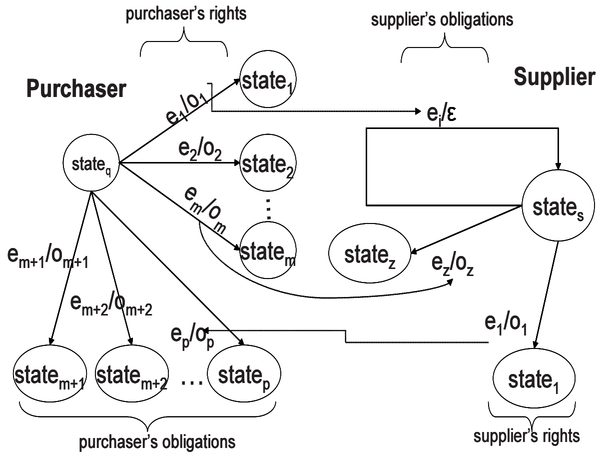

Fig. 1. Contractual rights and obligations represented with FSMs.

The Spin Simulator

The Simulator runs through a single sequence of reachable states (path or routes) of the model coded in Promela. The designer can choose a specific path for the simulator to run through, or can leave the simulator to run through a random path. The simulator will test a specific path for some safety correctness requirements; freedom from deadlocks (CR4), unspecified receptions (which covers CR11), and unattainable states (CR3).

The Spin Validator

The Validator is used for validating the correctness requirements of *Promela* code (the verification model). It generates and inspects all the states and paths of the system that are reachable from the initial state. The Spin validator lists a number of correctness properties that the designer can choose from to validate the correctness of its model. Spin's correctness properties are very similar to the contractual correctness requirements that we listed in Section 3. Consequently we have found that Spin's validator can be used to successfully validate contract correctness requirements.

To validate a contract model, we run the validator against each of the desired correctness requirements. The validator will highlight any paths through the model that have errors. The designer can then use the Simulator to run through the erroneous path, and trace the point at which the error originated.

In this section, we present an example of a contract (Fig. 2) for the supply of e-goods between a *Supplier* and a *Purchaser.* The contract at a first glance looks correct. We will use Spin to verify whether the contract satisfies some of the correctness requirements listed in Section 3, and therefore discovering any inconsistencies within the contract.

The contract clauses that we would like to verify are the following:

2. Offer

2.1 The supplier may use his discretion to send offers to the purchaser.

2.2 The purchaser is entitled to accept or reject the offer, but he shall notify his decision to the supplier.

3 Commencement and completion

3.1 The contract shall start immediately upon signature.

3.2 The purchaser and the supplier shall terminate the x-contract immediately after reaching a deal for buying an item.

This deed of agreement is entered into as of the effective date identified below.
Between
[Name] of [Address] (To be known as the (Supplier)), and [Name] of [Address] (To be knows as the (Purchaser)).
Whereas
(Supplier) desires to enter into an agreement to supply (Purchaser) with [Item].
Now it is hereby agreed that (Supplier) and (Purchaser) shall enter into an agreement subject to the following terms and conditions:
1. Definitions and Interpretations
1.1 Price, Dollars or $ is a reference to the currency of the [Country].
1.2 All information (purchase order, payment, notifications, etc.), is to be sent electronically.
1.3 This agreement is governed by [Country] law and the parties hereby agree to submit to the jurisdiction of the Courts of the [Country] with respect to this agreement.
2. Offer
2.1 The supplier may use his discretion to send offers to the purchaser.
2.3 The purchaser is entitled to accept or reject the offer, but he shall notify his decision to the supplier.
3. Commencement and completion
3.1 The contract shall start immediately upon signature.
3.2 The purchaser and the supplier shall terminate the x-contract immediately after reaching a deal for buying an item.
4. Disputes
*4.1 (Supplier) and (Purchaser) **shall** attempt to settle all disputes, claims or controversies arising under or in connection with the agreement through consultation and negotiations in good faith and a spirit of mutual cooperation.*
*4.2(Supplier) and (Purchaser) **shall** provide electronic evidences about breaches of the e-contract.*
4.3 This method of determination of any dispute is without prejudice to the right of any party to have the matter judicially determined by a [Country] Court of competent jurisdiction.
5 Amendment
*5.1 This agreement **may** only be amended in writing signed by or on behalf of both parties.*
E-SIGNATURES
In witness whereof (Supplier) and (Purchaser) have caused this agreement to be entered into by their duly authorized representatives as of the effective date written below.
Effective date of this agreement: [day] of [month] [year]

[E-signature]	*[E-signature]*
[Person]	*[Person]*
[Role]	*[Role]*

E-address for Notices:

[E-address]	*[E-address]*

Fig. 2. A contract between a purchaser and a supplier for the purchase of goods.

From these contract clauses, we can extract the sets of rights and obligations for the Purchaser and the Supplier and express them in terms of operations for FSMs. The sets of rights and obligations stipulated in this contract look as follows:

Purchaser's rights:

R_1^P : SendAccepted -- right to accept offers.

R_2^P : SendRejected -- right to reject offers.

Purchaser's obligations:

O_1^P : StartEcontract -- obligation to start the x-contract.

O_2^P : SendAccepted or SendRejected -- obligation to reply to offers.

O_3^P : EndEcontract -- obligation to terminate the x-contract.

Supplier's rights:

R_1^s : SendOffer -- right to send offers.

Supplier's obligations:

O_1^S : StartEcontract -- obligation to start the x-contract.

O_2^S : EndEcontract -- obligation to terminate the x-contract.

Fig. 3 shows how the sets R and O are mapped into FSMs.

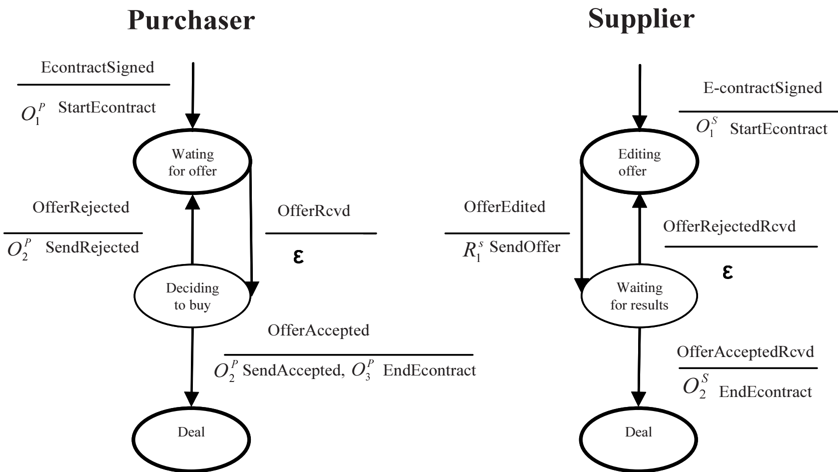

Fig. 3. Representation of a contract for the purchase of goods, with FSMs.

To validate our contract clauses we have to convert the FSM shown into the modeling language *Promela* first. The result of this conversion is shown in Fig. 4.

5.2 X-Contract Verification

Safety Properties

Safety properties can be categorized into, *general* safety properties that must hold true for any x-contract (CR3: Attainability, CR4: Freedom from deadlocks, CR11: Absence of unsolicited responses), and *specific* safety properties that must hold true only if so required by the contracting parties for the specific requirements of a certain x-contract (CR5: Partial correctness, CR6: Invariant, and CR8: Precedence).

Running the Spin validator under its default settings will check for *general* safety properties. Validation of the remaining *specific* safety properties can be done by in-

serting "Assertions" within the Promela code. Running the Spin validator under its default settings gives us the results shown in Fig. 5.

```
/*Verification Model for the Contract Finite State Machines*/
/*in their initial ambiguous state*/

#define              MA   20 /*Maximum acceptable offer*/
#define  OA    1   /*Offer accepted */
#define  OR    0   /*Offer rejected*/

mtype = {Offer, Response}
chan S2P = [1] of {mtype, int};
chan P2S = [1] of {mtype, byte};

proctype Supplier() /***Suppliers FSM***/
{
  int offerValue;
 byte responseValue; /*OA or OR*/
  SupEContractSigned:
  EditingOffer:
if
    :: offerValue = 30; /* An offer that is too high > MA*/
    :: offerValue = 20; /* < MA */
    :: offerValue = 10; /* < MA */
 fi;
 if
    :: S2P!Offer(offerValue) -> goto WaitingForResults;
    :: skip /*Taking into account the possiblity that*/
 fi;      /*the supplier might not send anything */
  WaitingForResults:
 P2S ? Response(responseValue);
  if
    :: (responseValue == OR) -> goto EditingOffer;
    :: (responseValue == OA) -> goto Deal;
fi;
 Deal:
 printf("\n\n Supplier: Deal \n\n");
 end:
 printf("\n\n Supplier: End \n\n");
}

proctype Purchaser() /***Purchasers FSM***/
{
 int  offerValue;
 PurEContractSigned:
 WaitingForOffer:
 S2P ? Offer(offerValue) ->
 DecidingToBuy:
 if
    ::(offerValue>MA)-> P2S!Response(OR);
          goto WaitingForOffer;
    :: else -> P2S ! Response (OA); goto Deal;
 fi;
 Deal:
 printf("\n\n Purchaser: Deal\n\n");
 end:
 printf("\n\n Purchaser: End\n\n");
}

init
{
 run Supplier();
 run Purchaser();
}
```

Fig. 4. A contract coded in Promela.

```
Verification Output

pan: invalid endstate (at depth 11)
pan: wrote pan_in.trail
(Spin Version 4.0.1 -- 7 January 2003)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never-claim             - (none specified)
        assertion violations    - (disabled by -A flag)
        cycle checks            - (disabled by -DSAFETY)
        invalid endstates       +

State-vector 44 byte, depth reached 23, errors: 1
```

Fig. 5. Output of the Spin validator.

Spin has detected an error in our verification model. "*invalid endstate (at depth 11)*". The fourth line in Fig. 5 indicates that the Spin validator stops the verification

process before completion because it detects an error in the model. XSpin saves the path where the error is detected. To trace the point at which the error occurred we can instruct XSpin to run the simulator through the offending path. The results of this simulation are shown in Fig. 6.

```
Simulation Output
preparing trail, please wait...done
  1:    proc  0 (:init:) line  78 "pan_in" (state 1)    [(run Supplier())]
  2:    proc  1 (Supplier) line  26 "pan_in" (state 1)  [offerValue = 30]
  3:    proc  0 (:init:) line  79 "pan_in" (state 2)    [(run Purchaser())]
  4:    proc  1 (Supplier) line  32 "pan_in" (state -)  [values: 1!Offer,30]
  4:    proc  1 (Supplier) line  32 "pan_in" (state 6)  [S2P!Offer,offerValue]
  5:    proc  2 (Purchaser) line  60 "pan_in" (state -) [values: 1?Offer,30]
  5:    proc  2 (Purchaser) line  60 "pan_in" (state 1) [S2P?Offer,offerValue]
  6:    proc  2 (Purchaser) line  65 "pan_in" (state 2) [((offerValue>20))]
  7:    proc  2 (Purchaser) line  65 "pan_in" (state -) [values: 2!Response,0]
  7:    proc  2 (Purchaser) line  65 "pan_in" (state 3) [P2S!Response,0]
  8:    proc  1 (Supplier) line  37 "pan_in" (state -)  [values: 2?Response,0]
  8:    proc  1 (Supplier) line  37 "pan_in" (state 11)
[P2S?Response,responseValue]
  9:    proc  1 (Supplier) line  40 "pan_in" (state 12) [((responseValue==0))]
 10:    proc  1 (Supplier) line  27 "pan_in" (state 2)  [offerValue = 20]
 11:    proc  1 (Supplier) line  33 "pan_in" (state 8)  [(1)]
spin: trail ends after 12 steps
#processes: 3
 12:    proc  2 (Purchaser) line  60 "pan_in" (state 1)
 12:    proc  1 (Supplier) line  37 "pan_in" (state 11)
 12:    proc  0 (:init:) line  80 "pan_in" (state 3)
3 processes created
Exit-Status 0
Single Step | Suspend | Save in: | Clear | Cancel
```

Fig. 6. Spin output showing and erroneous path.

After step 10, the Supplier was expected to send an offer to the Purchaser, but the Simulator does not show this occurring. A closer look at step 11 reveals that the trail ended after the simulator went through line 33 of the *Promela* verification model:

```
31  if
32  :: S2P!Offer(offerValue) -> goto WaitingForResults;
33  :: skip /*Taking into account the possibility that*/
34  fi;      /*the supplier might not send anything */
```

Line 33 represents the fact that the *Supplier* might choose not to send the offer to the *Purchaser* for whatever reason. Fig. 5 also shows that the simulator detects problems in lines 60, and 37:

```
59 WaitingForOffer:
60 S2P ? Offer(offerValue) ->

36 WaitingForResults:
37 P2S ? Response(responseValue);
```

No *offerValue* was received by the *Purchaser* process, and subsequently, no *responseValue* was received by the *Supplier* process. The finite state machines of the *Supplier* and the *Purchaser* fall into a deadlock situation.

A possible solution to avoid this undesirable situation is to make use of the *Promela* "*timeout"* statement. This statement allows a process to abort and not wait indefinitely for a condition that can no longer become true such as the one we just encountered:

```
59  WaitingForOffer:
60  if
61    ::S2P ? Offer(offerValue)
62    ::timeout -> goto end
63  fi;
```

We can run the validator as many times as necessary, and after ensuring the correctness of the *general* safety requirement, we can use the validator to check for some *specific* safety correctness requirements. For example, we would like to check that the invariant "The price offered by the *Supplier* should not be accepted by the Purchaser if the price exceeds an agreed price *P"* holds (see CR6 in Section 3). To guarantee this invariant we can insert an assertion of the form *assert(offerValue<=P)* at the required check points in the verification model. We then set and run the validator to check for assertions. The validator does not signal any errors, so we know that the invariant we specified holds true.

Liveness Properties

Unlike safety properties, there are no *general* liveness properties. All liveness properties are *specific* to the requirements of the contracting parties depending on the purposes of a specific contract. To validate liveness properties (correct termination, occurrence or accessibility, livelocks, responsiveness) we can insert specifically designed labels such as "accept" labels that check for livelocks, "progress" labels that check for progress states, and temporal claims, in the Promela code.

As an example, in our x-contract we would not desire a situation where the supplier infinitely often makes undesirable offers. That is we do not want livelock (CR9) in the x-contract. We can insert an accept label in line 20 as follows:

```
17  EditingOffer:
18  if
19   :: offerValue = 30;
20 acceptOfferTooHigh: skip  /* An offer that is too
                                          high > MA*/
21   :: offerValue = 20; /* < MA */
22   :: offerValue = 10; /* < MA */
23  fi;
```

We can now set the validator verification parameters to detect "livelock". The output results show that the search stops after detecting an error. A simulator run would show that the problem occurs after the Supplier makes an offer with *offerValue=30*. The output shows that we have an undesirable situation where the *Supplier* can make unacceptable offers infinitely. There are many possible solutions to this problem, one would be for example to limit the Supplier to *N<=10* offers.

Following testing the x-contract model against the desired correctness properties, and removal of detected ambiguities, the verification model and therefore the x-contract must be modified accordingly. For our example, the finite state machines are modified as can be seen in Fig. 7.

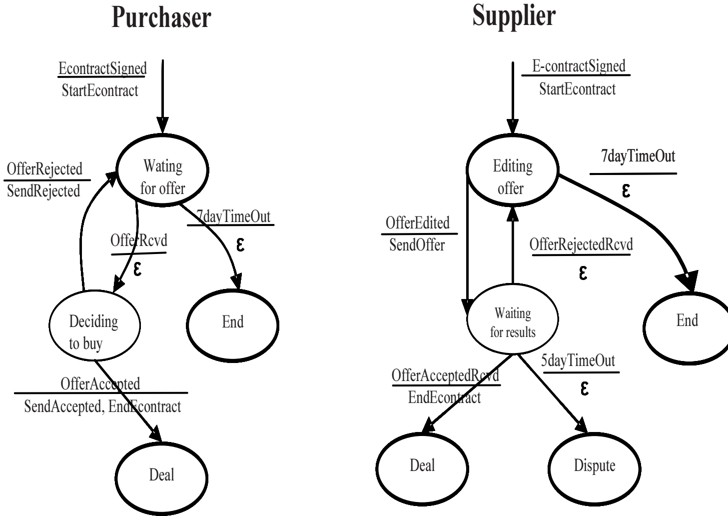

Fig. 7. Representation of a contract with FSMs (revised version of Fig. 3).

After the corresponding modifications the contract clauses look as follows:

2 Offer

2.1 The supplier may use his discretion to send offers to the purchaser.

2.2 If no offer is sent within seven days after the signature of the x-contract, or after the latest rejected offer, the x-contract shall be terminated.

2.3 The purchaser is entitled to accept or reject the offer, but he shall notify his decision to the supplier within five days after the receipt of the offer.

3 Commencement and completion

3.1 The contract shall start immediately upon signature.

3.2 The purchaser and the supplier shall terminate the x-contract immediately after reaching a deal for buying an item.

Complex Correctness Requirements

A very useful facility provided by Spin is the verification of “temporal claims”. Temporal claims can be used to express complex correctness requirements. This facility is very useful as transactions between parties to a contract may need to run in a certain sequence, and/or under certain conditions.

As a separate example let us consider the Promela code of Fig. 8 which describes a complaint handling state machine. We want the validator to check the requirement that a complaint about the quality of the goods must not be sent by the *Purchaser* before the goods are received from the *Supplier*. The verification model to express possible scenarios is shown in Fig. 8.

In *XSpin*, verification of temporal claims is done using the Linear Temporal Logic (LTL) Manager. We are claiming the following: It is invariantly true that following the placement of an order a complaint should not be received before the order is received. This is expressed in Linear Temporal Logic as follows:
[] (placeOrder -> !complaintRecd U orderRecd).

We can enter this formula into the LTL Manager (Fig. 8) and then run the validator. As expected, the validator detects that our claim is false. As it can be proved with the validator, it is enough to remove the lines in the Promela model that gives the Purchaser the option to complain before receiving the order, to make our claim hold true.

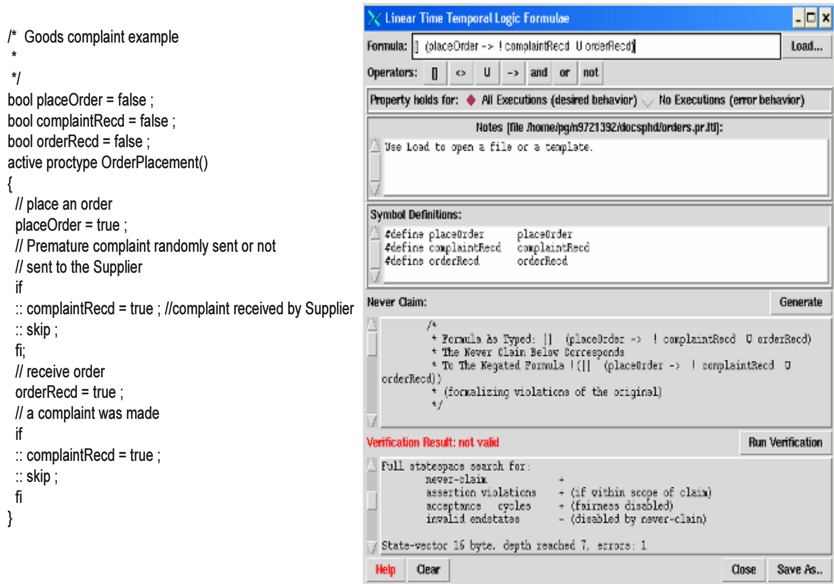

Fig. 8. Use of the LTP property manager for validating a temporal claim in a Promela model.

6 Conclusions and Further Work

This paper contains some of the results we have obtained from our work on modeling business contracts. Our thesis is that contracts are complex systems that involve contractual and private policies. Contractual policies regulate the interaction between the trading enterprises, whereas private policies regulate the interaction between the members of a trading enterprise and the contract. To simplify the problem, we propose to study the two kinds of policies separately. Thus in this paper we focused on contractual policies. We argue that conventional business contracts normally contain ambiguities that should be detected and eliminated before converting the contract into its electronic equivalent. To help the validator of an electronic contract, we provide a list of correctness requirements that most traditional business contracts should satisfy.

The list is not exhaustive but illustrative. To put contract correctness in the context of traditional program correctness, we mapped the list of contract correctness requirements into conventional safety and liveness properties. We also argue that FSMs are a suitable formal notation for describing conventional contracts and show how a contract described by means of FSMs can be validated using standard and readily available model checkers such as Spin. To support our arguments we illustrate the validation of two simple contracts. More complex examples are discussed in [12].

Acknowledgements. This work is part-funded by the UK EPSRC under grant GR/N35953/01: "Information Co-ordination and Sharing in Virtual Environments"; by the European Union under Project IST-2001-34069: "TAPAS (Trusted and QoS-Aware Provision of Application Services)"; and by the UK DTI e-Science programme under project "GridMist".

References

1. Andrew Goodchild, Charles Herring and Zoran Milosevic, Business Contract for B2B, In Proceedings of the CAISE00 Workshop on Infrastructure for Dynamic Business-to-Business Service Outsourcing; Stockholm, June 5–6, 2000.
2. Z. Milosevic and R.G. Dromey, On Expressing and Monitoring Behaviour in Contracts, In proceedings of the 6th International Enterprise Distributed Object Computing Conference (EDOC2000), Lausanne, Switzerland, Sep. 17–20, 2002.
3. Olivera Marjanovic and Zoran Milosevic, Towards Formal Modeling of e-Contracts, In Proceedings of the 5th International Enterprise Distributed Object Computing Conference (EDOC 2001), 4–7 Sep. 2001, Seattle, WA, USA, IEEE Computer Society 2001.
4. Abrahams A.S., Eyers D.M., and Bacon J.M. "Mechanical Consistency Analysis for Business Contracts and Policies". Proc 5th International Conference on Electronic Commerce Research (ICECR5), Montreal, Canada, 23–27 October 2002.
5. E. Lupu, M Sloman, N. Dulay, N. Damianou, Ponder: Realising Enterprise Viewpoint Concepts, In proceedings of the 4th International Enterprise Distributed Object Computing Conference (EDOC2000), Makuhari, Japan, 25–28 Sep. 2000.
6. Carlos Molina-Jimenez, Santosh Shrivastava, Ellis Solaiman and John Warne, "Contract Representation for Run-time Monitoring and Enforcement", Proc. IEEE Int. Conf. on E-Commerce (CEC-2003), Newport Beach, California, June 2003.
7. Web Service Conversation Language (WSCL) 1.0 (http://www.w3.org/TR/wscl10/).
8. Rosettanet implementation framework: core specification, V2, Jan 2000. http://rosettanet.org.
9. Gerard J. Holzmann, The Model Checker Spin, IEEE Transactions on Software Engineering, Vol. 23, N. 5, May, 1997.
10. Gleb Naumovich and Lori A. Clarke, Classifying Properties: An Alternative to the Safety-Liveness Classification, In Proceedings of the Eighth International Symposium on the Fundations of Software Engineering, Nov. 2000.
11. B. Alpern and F.B. Schneider, Defining liveness, Information Processing Letters, Vol. 21, N. 4, Oct, 1985.
12. Ellis Solaiman, University of Newcastle upon Tyne, PhD Theses (In preparation).