COMPUTING SCIENCE

A High–Level Model–Checking Tool for Verifying Electronic Contracts

Abubkr Abdelsadiq, Carlos Molina–Jimenez and Santosh Shrivastava

TECHNICAL REPORT SERIES

No. CS-TR-1279 September 2011

TECHNICAL REPORT SERIES

No. CS-TR-1279 September, 2011

A High–Level Model–Checking Tool for Verifying Electronic Contracts

A. Abdelsadiq, C. Molina–Jimenez and S. Shrivastava

Abstract

An electronic contracting system intended for monitoring and/or enforcement of business-to-business interactions to ensure that they comply with the rights, obligations and prohibitions stipulated in contract clauses requires a machine interpretable specification of the relevant parts of the legal contract in force. Within this context, Event Condition Action (ECA) rules are widely used for representing contracts. Naturally, it is important to verify the correctness properties of such a contract before its deployment. To this end, the paper adopts the use of model-checking techniques. A high-level model--checking tool has been developed that enables a designer to encode a contract for model checking directly as ECA rules in terms of contract entities: business operations, role players with their rights, obligations and prohibitions. This not only simplifies the task of model building but also, the designer can specify the correctness requirements, in linear temporal logic, directly in terms of the contract entities. The tool has been implemented by extending the PROMELA language of the SPIN model checker.

Printed and published by Newcastle University,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

AABDELSADIQ,A., MOLINA–JIMENEZ, C., SHRIVASTAVA, S.

A High–Level Model–Checking Tool for Verifying Electronic Contracts
[By] A. Abdelsadiq, C. Molina–Jimenez, S. Shrivastava
Newcastle upon Tyne: Newcastle University: Computing Science, 2011.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1279)

Added entries

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1279

Abstract

An electronic contracting system intended for monitoring and/or enforcement of business-to-business interactions to ensure that they comply with the rights, obligations and prohibitions stipulated in contract clauses requires a machine interpretable specification of the relevant parts of the legal contract in force. Within this context, Event Condition Action (ECA) rules are widely used for representing contracts. Naturally, it is important to verify the correctness properties of such a contract before its deployment. To this end, the paper adopts the use of model-checking techniques. A high-level model--checking tool has been developed that enables a designer to encode a contract for model checking directly as ECA rules in terms of contract entities: business operations, role players with their rights, obligations and prohibitions. This not only simplifies the task of model building but also, the designer can specify the correctness requirements, in linear temporal logic, directly in terms of the contract entities. The tool has been implemented by extending the PROMELA language of the SPIN model checker.

About the authors

Abubkr Abdelsadiq is a final year PhD student at the School of Computing Science of Newcastle University, UK. His current research interest is focused on verification and testing of electronic contracts using model-checking tools and techniques.

Carlos Molina-Jimenez received his PhD in the School of Computing Science at the University of Newcastle upon Tyne in 2000 for work on anonymous interactions in the Internet. He is currently a Research Associate in the School of Computing Science at the University of Newcastle upon Tyne where he is a member of the Distributed Systems Research Group. He is working on the EPSRC funded research project on Information Coordination and Sharing in Virtual Enterprises where he has been responsible for developing the Architectural Concepts of Virtual Organisations, Trust Management and Electronic Contracting.

Professor Santosh Shrivastava was appointed Professor of Computing Science, University of Newcastle upon Tyne in 1986. He received his Ph.D. in computer science from Cambridge University in 1975. His research interests are in the areas of computer networking, middleware and fault tolerant distributed computing. The emphasis of his work has been on the development of concepts, tools and techniques for constructing distributed fault-tolerant systems that make use of standard, commodity hardware and software components. Current focus of his work is on middleware for supporting inter-organization services where issues of trust, security, fault tolerance and ensuring compliance to service contracts are of great importance as are the problems posed by scalability, service composition, orchestration and performance evaluation in highly dynamic settings. Professor Shrivastava sits on programme committees of many international conferences/symposi. He is a member of IFIP WG6.11 on Electronic commerce - communication systems, and sits on the advisory board of Arjuna technologies Ltd.

Suggested keywords

ELECTRONIC CONTRACTS
CONTRACT SPECIFICATION
CONTRACT MONITORING/COMPLIANCE
ECA-RULES
MODEL CHECKING BASED VERIFICATION
PROMELA/SPIN

A High–Level Model–Checking Tool for Verifying Electronic Contracts

Abubkr Abdelsadiq
School of Computing Science,
Newcastle University
Newcastle upon Tyne, NE1 7RU, UK
Email: abubkr.abdelsadiq@ncl.ac.uk

Carlos Molina–Jimenez
School of Computing Science,
Newcastle University
Newcastle upon Tyne, NE1 7RU, UK
Email: carlos.molina@ncl.ac.uk

Santosh Shrivastava
School of Computing Science,
Newcastle University
Newcastle upon Tyne, NE1 7RU, UK
Email: santosh.shrivastava@ncl.ac.uk

***Abstract*—An electronic contracting system intended for monitoring and/or enforcement of business-to-business interactions to ensure that they comply with the rights, obligations and prohibitions stipulated in contract clauses requires a machine interpretable specification of the relevant parts of the legal contract in force. Within this context, Event Condition Action (ECA) rules are widely used for representing contracts. Naturally, it is important to verify the correctness properties of such a contract before its deployment. To this end, the paper adopts the use of model-checking techniques. A high-level model–checking tool has been developed that enables a designer to encode a contract for model checking directly as ECA rules in terms of contract entities: business operations, role players with their rights, obligations and prohibitions. This not only simplifies the task of model building but also, the designer can specify the correctness requirements, in linear temporal logic, directly in terms of the contract entities. The tool has been implemented by extending the PROMELA language of the SPIN model checker.**

I. Introduction

Fulfilling a given business function (e.g., 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. Compliance checking can be automated with the help of electronic contracting systems that can also be used for detecting contract violations, facilitating dispute resolution and determining liability by providing an audit trail of business interactions. An electronic contracting system will require a machine interpretable specification of relevant parts of the legal contract in force. It is important to verify the correctness properties of such an electronic contract before its deployment. 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. There is thus a strong case for developing tools for contract verification.

This paper describes a model-checking based tool intended for electronic contracts encoded using Event Condition Action (ECA) rules. We chose ECA rules because of their wide spread usage in the business world for representing business agreements. Our tool supports a very traditional approach to system building: a model of the system is first constructed and verified against a set of key correctness requirements; the model then provides the basis for the actual implementation of the system.

The challenge is to build a model that is simple enough to be reasoned about by humans and amenable to verification by automated tools, yet at the same time, the model should approximate the intended actual system sufficiently closely to the extent that the task of building the actual system satisfying the requirements, given the model, becomes a relatively straightforward process. The constructed system must be validated to make sure that it does indeed meet the requirements that the model satisfied. Here again, the verification techniques used for the model should provide useful inputs for the generation of test cases for exercising the constructed system.

We make use of a model checking technique that is widely used for automatic verification of reactive systems. With this technique, a model is constructed as a set of interacting state machines and the model checker generates all possible states of the model and checks that specified properties hold in each state. From our experience, we have learnt that contract designers need a model–checking tool that hides much of the intricate details of constructing state machine models and permits modelling of contract clauses directly in terms of basic contract concepts of business operations, role players, rights, obligations and prohibitions. For instance, the tool should readily and intuitively allow to express that *the role player buyer is currently obliged to execute business operation payment*. Equally important, the tool should offer a notation to express correctness requirements directly using essential contract concepts and to verify them. For instance, it should allow, the designer to express that a *the role player buyer is always obliged to execute payment operation for a given item, exactly once*.

We have realised that 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 (and the subject of exploration in this paper) is to build such a tool using an existing model–checking tool originally designed for validation of distributed applications such as communication protocols and enhancing it with contract–specific constructs.

In this work, we take the pragmatic alternative and explore the possibility of taking advantage of the facilities of the SPIN model checker [1], [2] in the verification and validation of business contracts. We discuss the implementation of a tool that is based on the extending the standard PROMELA (language of SPIN) with the concept of business operation and operators to manipulate it. For example, in extended PROMELA, the designer can include in his model operations like *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 abstracted as safety and liveness properties and expressed in LTL (Linear Temporal Logics) formulae. This feature is of particular interest to our work because, as discussed in [21], a large class of correctness requirements of electronic contracts can be abstracted as safety and liveness properties. This suggests that they can be expressed in LTL formulae and verified by conventional model–checkers like SPIN.

The justification for extending PROMELA, as opposite to using the standard version, is that SPIN was originally designed for verification of communications protocols. It was designed to verify models written in PROMELA language against correcteness properties (safety and liveness) expressed in standard LTL formulae. Thus PROMELA provides constructs for modelling essential communication concepts such as messages, channels and operations for sending and receiving messages from channels. Not surprisingly, standard PROMELA offers only a few and very basic built–in data types that include bit, byte and array together with basic operations to manipulate them.

In principle and as demonstrated by our previous works ([6], [13], [21]), PROMELA's basic contructs are sufficient for building models of electronic contracts. However, the encoding of contract clauses in standard PROMELA, quickly turns into an arduous task that engages the designer in language details that distract him or her from the logical aspects of the model. Thus it is worth exploring other alternatives.

The design process supported by our verification tool is illustrated in Fig.1. We assume that (i) the contract has been negotiated by the contracting parties and drawn up in English (or other natural) language. (ii) The designer manually converts the English text into an abstract model written in our extended PROMELA language. In parallel, the designer manually prepares a list of contract correctness requirements (deduced from the contract clauses) into LTL formulae. (iii) The designer inputs both the CB2B model and the LTL formulae into the SPIN verifier and runs it to output verification results.

The CB2B model (Contractual Business To Business interaction model) is constructed using the concepts of contract compliance checking that we have developed earlier, and described in [3]. It contains the clauses of the contract expressed as ECA rules written in PROMELA. The concepts discussed in [3] also underpin the rule based contract specification language called EROP (for Events, Rights, Obligations and

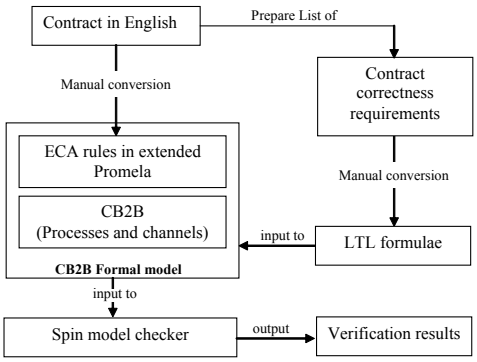


Fig. 1. Contract model checking framework.

Prohibitions) and a contract compliance checking service for contracts written in EROP [4], [5]. Once a contract has been verified using our tool, the ECA rules from the CB2B model can be translated relatively easily into their EROP counterpart (indeed, this process can be automated, although we have not yet implemented this step). We have therefore a systematic way of generating a machine interpretable contract. In a separate paper we have described how the model checker can be used for generating test cases for the EROP system [6]. Thus our tool, built using a very widely available model checker, provides a comprehensive framework for contract verification and validation. In this paper we describe the structure of the CB2B model and illustrate its use by taking a hypothetical internet service provision contract that has been used by other researchers for illustrating contract verification concepts and techniques [7]. The next section describes the salient aspects of contract compliance checking; in Section III we will describe the CB2B model and its use for contract verification.

II. Contract Compliance Checking

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

We consider an independent, third party contract monitoring service called *Contract Compliance Checker* (CCC). The CCC (see Fig. 2 which depicts the logical communication paths between business partners and the CCC) is provided with an executable specification of the contract in force; it is able to observe and log the relevant business–to–business (B2B) interaction events which it processes to determine whether the actions of the business partners are consistent with respect to the contract.

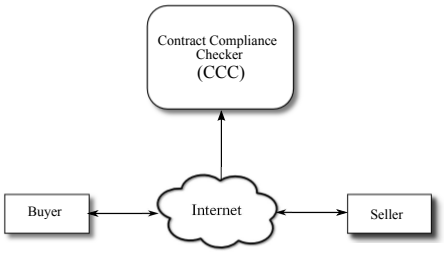


Fig. 2. Contract compliance checking.

We assume that 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 two business documents. A business operation would normally be implemented by a *business conversation*: a well defined message interaction protocol with stringent message timing and validity constraints (normally, a business message is accepted for processing only if it is timely and satisfies specific syntactic and semantic validity constraints). RosettaNet Partner Interface Processes and ebXML industry standards serve as good examples of such conversations [8], [9]. Following the ebXML specification [9], we assume 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. *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 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.

Failure outcomes play an important role in making electronic contracts tolerant against infrastructure level problems, as they provide a way of incorporating specific exceptional clauses to deal with them [10]. For example, an exceptional payment clause might be along the lines: ”failure to meet a payment deadline due to business or technical reasons will grant 5 days extension to the buyer”. Another example: ”If the total number of business and technical failures exceed an agreed bound, then online processing will be terminated”.

As indicated by the *business events* label of Fig.2, we assume that the CCC is able to observe B2B interactions at the granularity of outcome events of business operations. Each such event contains information that includes the termination status (*Success*, *BizFail* or *TecFail*), name of the operation, the timestamp and other attributes to classify the operation further (for example, the role player performing the operation). The monitoring channel delivers these events to the CCC exactly once in temporal order; these events are logged at the CCC.

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 is observing outcome events of business operations. With each participant, also termed a role player, we associate a *ROP set*, the set of Rights, Obligations and Prohibitions currently in force. We use the set $B = \{bo_1, \ldots, bo_n\}$ of business operations to specify all the primitive business operations stipulated in a contract.

For the CCC, the execution of a 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, 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* 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 an 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 (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 (all the obligations 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 any violation (non fulfllment) of an obligation that could occur during an interaction; such a violation normally leads to sanctions coming in force and if these are honoured, then the contractual interaction could still end normally.[1]

Elsewhere we have argued that the concepts presented above form a sound basis for constructing contract compliance checking systems and rule based contract languages [3] . Basically, a CCC (see Fig. 2) has an event queue for storing incoming events that represent execution outcomes of business operations. It executes the following algorithm:

1) Fetch the first event e from the Event Queue;

[1]Sanctions are obligations that come in force when the primary obligations are violated.

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

In a practical implementation of a CCC, testing for conditions C1 and C2 can be made a standard part of event processing in step 1, so the 'condition' part of a rule in step 3 need only be concerned with checking for C3. We use a simple example (taken from [11]) to illustrate how rules are constructed; the example will also motivate the reader for the need for verification.

The significance of the ROP sets is that they allow us to abstract the behaviour of the CCC as a reactive system [3]. As a reactive system, the CCC remains in a given state waiting for the arrival of events. When an event arrives that represents a contract compliant operation, the CCC changes its state, otherwise the event is flagged as non-contract compliant and no state change occurs. Thus, a CCC with a correctly coded contract rules will have the property that a contract compliant business operation is never flagged as non-contract compliant.

Let us consider a very simple contract fragment involving just one role player who is repeatedly performing operations, 'a' followed by 'b' or 'c' under a constraint stated as follows: *there is an obligation to choose between doing 'b' or 'c' after 'a', and a prohibition on doing 'b' if 'b' has been performed*. For the sake of simplicity we assume that operations always terminate successfully (there are no *BizFail* and *TecFail*). The pseudo code of the rules is shown in Fig. 3. There are three rules corresponding to events representing the execution of the corresponding business operations.

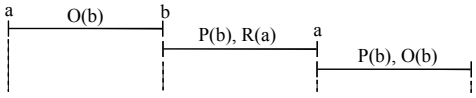
```
ContractRule "a"
{
   if IS_R(a){
      SET_R(a, FALSE);
      Choice{
          :: SET_O(b, TRUE);
          :: SET_O(c, TRUE);
       }
}
```

```
ContractRule "b"
{
   If IS_O(b){
      SET_O(b, FALSE);
      SET_P(b, TRUE);
      SET_R(a, TRUE);
   }
   if IS_P(b){
      SET_R(a, TRUE);
    }
}
```

```
ContractRule "c"
{
   if IS_O(c) {
      SET_O(c, FALSE);
      SET_R(a, TRUE);
   }
}
```

Fig. 3. Simple contract rules example.

Let us see how the ROP set is manipulated in the rule for 'a': check that there is a right to perform 'a' (IS_R(a) returns TRUE), and if so, that right is now removed (as 'a' has been performed) and obligation to perform either 'b' or 'c' (chosen non–deterministically) is inserted. In the rules for 'b' and 'c', the right to perform 'a' is inserted again. At a first glance, these rules seem to be an accurate representation of the contract. We would like to make sure that the rules satisfy the following requirement: *there should be no simultaneous obligation and prohibition on executing operation 'b'*. It turns out that our rules do not meet this requirement. Fig. $4(a)$ 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'.



(a)

```
ContractRule "a"
{
   if IS_R(a){
      SET_R(a, FALSE);
      Choice{
          :: if not IS_P(b){
                  SET_O(b, TRUE);
                  }
           :: SET_O(c, TRUE);
       }
    }
}
```

(b)

Fig. 4. (a) inconsistent assignment of prohibition and obligation (b) rule 'a' modified.

Our encoding of rule for 'a' is not quite right. The corrected rule is shown in Fig.$4(b)$; we make sure now that the obligation to perform 'b' is inserted only when there is no prohibition on it.

III. CB2B Model

A. Introduction

In our earlier work, we used PROMELA directly to model CCC and the rules [13]. We quickly realised that with pure PROMELA, encoding of contract rules can be an arduous task. Lack of data types other than the built-in types *bit*, *byte*, *array*, etc., is seen as a serious restriction to use the PROMELA as a specification language instead of protocol modeling language [12]. However, the language can be extended to add user defined data types. The *typedef* construct can be used to define new data types 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*. We also implemented a set of operations on the *BIS_OP* data type to maintain information about ROP sets. We added these enhancements at user level, thus they do not impact the SPIN source code.

These additional facilities, not only simplifies the task of writing rules, but also, help the designer specify the correctness requirements, in linear temporal logic, directly in terms of the contract entities. This design together with the various language notations are discussed in the next subsection. Here we give a flavour by showing how the rules for the simple example discussed earlier will look like in extended PROMELA (see Fig.5). 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*". We are showing the corrected version of rule 'a' here; following

```
RULE(a)
{
 if
    :: IS_R(a)->
       SET_R(a,0);
       if
          :: IIS_P(b)->
             SET_O(b,1);
          :: SET_O(c,1);
       fi;
       RULE_DECI(CCR,CON);
    :: else
       RULE_DECI(NCC,CON);
fi;
}
```

```
RULE(b)
{
  if :: IS_O(b) ->
        SET_O(b,0);
        SET_P(b,1);
        SET_R(a,1);
        RULE_DECI(CCO,CON);
     :: IS_P(b)->
        SET_R(a,1);
        RULE_DECI(CCP,CON);
     :: else
        RULE_DECI(NCC,CON);
  fi;
}
```

```
RULE(c)
{
  if :: IS_O(c)->
        SET_O(c,0);
        SET_R(a,1);
        RULE_DECI(CCO,CON);
     ::else
        RULE_DECI(NCC,CON);
  fi;
}
```

Fig. 5. Simple contract rules in extended PROMELA.

PROMELA, the choice between *oblig(b)* or *oblig(c)* in rule 'a' will be chosen non–deterministically.

We can see that these rules closely resemble the pseudo code discussed earlier. Each rule ends with a decision (sent to the event generator) indicating whether the operation is contract compliant (RULE_DECI(CCR,CON)) or non–contract compliant (RULE_DECI(NCC,CON)); CON is short for continue, indicating that event generation should continue, CCR, CCO and CCP that appear in the rules are short for contract compliant right, obligation and prohibition respectively, and NCC is short for non–contract compliant.

The requirement that *there should be no simultaneous obligation and prohibition on executing operation 'b'* is written in LTL as:

```
[](not( IS_O(b) && IS_P(b) ))
```

That is, *always it is not possible to be obliged and prohibited on executing business operation 'b'*. If we had coded rule 'a' incorrectly as mentioned earlier, model checking would have revealed the error.

B. Model implementation

An abstract view of our framework is shown in Fig. 6. The key component of the figure is the CB2B model which essentially models the system depicted in Fig. 2. The LTL formulae are the correctness requirements that the designer wishes to validate as discussed in Fig. 1.

The Business Event Generator (BEG) represents the buyer and seller, precisely, their interaction over the Internet, thus it is responsible for generating business events, for example, payment placed by buyer. The Contract Rules Manager (CRM) together with the ROP sets and the ECA rules (rule base) represents the CCC. The CRM is responsible for including rules as needed. The BEG and CMR are communicated by two uni-directional channels (BEG2R and R2BEG). The contract rules are composed in a separate file and offered to CRM via the usual *#include* mechanism. The ROP sets contain information about the rights, obligations and prohibitions currently in force.

The rule base contains a rule for each business event be_i representing the outcome of an operation execution; so for a business operation say, 'submit purchase order' there will be

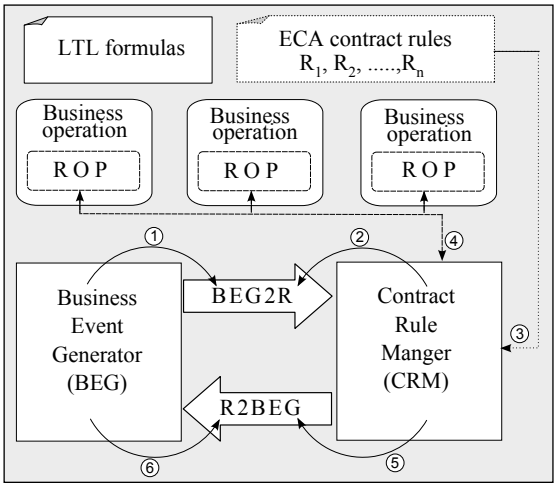


Fig. 6. CB2B formal model.

a rule for the operation terminating successfully (S), and optionally (depending on whether the contact has clauses dealing with failure outcomes) a rule for the operation terminating in a technical failure (TF) and one for the operation terminating in a business failure (BF).

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

1) BEG generates event be_i and sends it through the BEG2R channel;
2) CRM reads be_i from the BEG2R channel;
3) CMR includes the contract rule R_i corresponding to be_i;
4) R_i checks be_i against the ROP sets (condition C2), and executes the action if the associated condition, C3, is satisfied;
5) R_i sends its decision about be_i (either contract compliant or non–contract–compliant) through the R2BEG channel;
6) BEG extracts the decision from the R2BEG channel and resumes its event generation process.

In our framework, a contract is specified by declaring a set of business operations, role players and rules and some global variables necessary for recording some aspect of contract execution that might be required by rules or to express LTL formulas (e.g, payment made, goods not delivered).

Table I summarises the operations defined on the *BIS_OP* abstract data type. We use $boName$ to indicate business operation name. A given instance of an operation records in its associated ROP set whether a role player has the right, obligation or prohibition to perform that operation. There are 'SET' methods to grant/remove a right, obligation or prohibition and 'IS' methods to test whether a role player has the right etc. SET_X method is used to record that the operation has been executed; that status can be checked by the IS_X method. These two methods are useful for 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 [10] for more examples).

Name	Description
BIS_OP(boName)	Declares BIS_OP of type typedef with the fields:*name, index, RolePlayer, right, obligation, prohibition, execution status.* Ex. - BIS_OP (offer)
SET_R(boName,1)	Gives (removes, when second parameter is 0) right to execute boName. Ex.- SET_R(offer,1)
SET_O(boName,1)	Assigns (removes, when second parameter is 0) obligation to execute boName. Ex.- SET_O (pay, 1)
SET_P(boName,1)	Sets (removes, when second parameter is 0) prohibition to execute boName. Ex.- SET_P(cancel,1)
IS_R(boName,RolePlayer)	Returns 1 if RolePlayer has permission to excute boName, 0 otherwise. Ex.- IS_R(offer, Seller)
IS_O(boName, RolePlayer)	Returns 1 if RolePlayer is obliged to excute boName, 0 otherwise. Ex.- IS_O (pay, Seller)
IS_P(boName, RolePlayer)	Returns 1 if RolePlayer is prohibited to execute boName, 0 otherwise. Ex.- IS_P (cancel, Buyer)
SET_X(boName)	Sets execution status of boName to 1. The default value is 0. Ex.- SET_X (offer) means offer has been executed.
IS_X(boName)	Returns 1 if boName has been executed. Ex.- IS_X(offer)
INIT(boName,R,O,P,RolePlayer)	Initiates BIS_OP with RolePlayer, (R)ight, (O)bligation or (P)rohibition. Ex.- INIT(offer,1,0,0,Seller)- means that Seller is given the right to excute offer.

TABLE I
BIS_O OPERATIONS LIST.

Name	Description
CONTRACT(boName,status)	Retrieves the execution status (S,BF,TF) of boName. Blocks if the expected result is not available. Ex.-CONTRACT(offer,S) returns 1 if offer has been executed with status S.
B_E(boName,status)	Sends an event with status (S,BF or TF) to the Rule Manager if boName is currently a right, obligation or prohibition of a role player. Ex.- B_E(offer,S).
RULE_DECI(m1,m2)	Used by rules to notify the BEG about the outcome (m1) of the execution of an operation and a request (m2) to generate the next event. m1 is CCR, CCO or CCP (contract compliance right, contract compliance obligation or contract compliance prohibition, respectively. Alternatively, m1 is NCCR, NCCO or NCCP (non–contract compliance right, non–contract compliance obligation or non–contract compliance prohibition, respectively. m2 is either CON or CND (continue or contract ended, respectively). Ex.- RULE_DEC(CCR,CON).
`RULE(boName,[status]){ Statements }`	The format of a business rule. boName is a business event. status is optional an equal to S, BF or TF.
`SYNCH(boName){ Statements } HCNYS(boName)`	The statements are executed if boName has been executed before. Used within a rule to do some actions only if boName has been executed.

TABLE II
CB2B OPERATIONS LIST.

Table 2 shows the operations and control structures that have been implemented as syntactic sugar; these supplement PROMELA constructs such as statement sequencing, atomic sequencing, concurrent execution, case selection, repetition and unconditional jumps. There use will become clear in the next section where we present a complete example.

We note that the following cyclic construct provides a powerful way for the BEG process to generate business events:

```
do
  :: B_E(a,S);
  :: B_E(a,TF);
  :: B_E(b,S);
  :: B_E(c,S);
od
```

This construct in a given iteration will consider the currently executable operations (see the explanation of B_E(boName,status)) and non–deterministically select one event of that operation to be sent on the BEG2R channel.

Recall that in the earlier section dealing with contract compliance, we had categorized business operations as *valid* (satisfying condition C1), *matched* (satisfying conditions C1 and C2) and *contract compliant* (satisfying conditions C1 and C2 and C3). BEG can be programmed to generate any combination of these. From the point of view of model checking, a particularly important case is generation of sequences of events corresponding to the execution of contract compliant operations only. This considerably reduces the size of the state space for exploration yet enables checking that rules are responding correctly to contract compliant business operations (e.g., 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 matched or contract compliant operations and to remove any errors. Then extend the state space of exploration by reprogramming the BEG to generate events of valid operations. This aspect is discussed further subsequently.

IV. Case study

We will illustrate our model checking framework by taking a hypothetical internet service provision contract that has been used by other researchers for illustrating contract verification concepts and techniques [7]. We consider two parameters of the service: high, low, denoting the client's Internet traffic. We abstract away several technical details on how the Internet traffic is measured, and consider the following part of the contract:

1) Whenever the Internet traffic is *high* then the client must pay x$ immediately, or the client must notify the service provider by sending an e-mail specifying that he will pay later.
2) In case the client delays the payment, after notification he must immediately lower the Internet traffic to *low* level, and pay later $2 * x\$$
3) if the client does not lower the Internet traffic immediately, then the client will pay $3 * x\$$
4) The provider is forbidden to cancel the contract without previous written notification by normal post and by e-mail.
5) The provider is obliged to provide the services as stipulated in the contract, and according to the law regulating Internet services.

Normal condition is that the Internet traffic generated by the client is low; if it goes to high, clause 1 comes into effect. We model check beginning with the initial condition of Internet traffic is high. Fig. 7 and Fig. 6 show the contract encoding necessary for model checking. Referring to Fig. 7, we begin by declaring some global variables, such as Payment, declaring two role players etc. We define a number of business operations; their intended functions should be clear from their names (so, PAY1 refers to x$ payment, PAY3 refers to $3*x\$$ payment, DELAY refers to the client deciding to delay payment and so forth).

At this point we can write some LTL formulas that we would like the ECA rules of the contract to satisfy. Here are some essential LTL properties P1, P2 and P3:

```
P1: {[]((IS_X(DELAY) && IS_X(NOTIFY)) ->
     <> (IS_X(LOWER) || IS_X(NOT_LOWER)))}

P2: {[] (IS_X(LOWER) -> <> IS_X(PAY2)) &&
     [](IS_X(NOT_LOWER) -> <> IS_X(PAY3))}

P3: {[]((IS_X(SEND)&& IS_X(WRITE)) -> <> (IS_X(CANCEL)))}
```

Note that we are able express directly in terms of the contract entities of the CB2B model. P1 states that if the client performs DELAY and NOTIFY then the traffic is lowered or not lowered. P2 captures the property that if the traffic is lowered then eventually PAY2 is performed, but if the traffic is not lowered, then eventually PAY3 is performed. P3 captures the property that if the ISP cancels the service provision, then he must have informed earlier by post (WRITE) and email (SEND).

In the BEG process, we initialise the global variables, and the business operations. So, Internet traffic is high, client is initially obliged to make PAY1 payment and so forth. The event generation loop has been programmed to generate matched events (in this study, we assume that all operations terminate successfully – there are no TF or BF). Note that we need to preserve causality as well, so NOTIFY is generated after DELAY.

Fig. 8 has all the required rules. The three payment rules are simple and follow the same logic. If the client decides to delay payment, he is obliged to lower the traffic *after* notification (he has also the right to not lower the traffic, but in that case he will be required to pay $3 * x\$$). The rules for DELAY and NOTIFY have been programmed accordingly; note the use of 'synch' construct in these rules.

We used SPIN to verify that all the LTL formulas stated here are satisfied by these contract rules. Other contract-independent properties such as freedom from deadlock are checked by SPIN by default.

```
#include "BizOperation.h"
#include "setting.h"
#include "icrules.h"

byte Payment;
bool INTERNET_HIGH;
bool INTERNET_LOW;
RuleMessage(S,TF,BF);
RolePlayer(CLIENT,ISP);

BIS_OP(PAY1);
BIS_OP(PAY2);
BIS_OP(PAY3);
BIS_OP(DELAY);
BIS_OP(NOTIFY);
BIS_OP(LOWER);
BIS_OP(NOT_LOWER);
BIS_OP(CANCEL);
BIS_OP(SEND);
BIS_OP(WRITE);

proctype BEG()
{
  contract:
  INTERNET_HIGH = TRUE;
  INTERNET_LOW  = FALSE;

  START();

  INIT(PAY1,0,1,0,CLIENT);
  INIT(PAY2,0,0,0,CLIENT);
  INIT(PAY3,0,0,0,CLIENT);
  INIT(DELAY,0,1,0,CLIENT);
  INIT(NOTIFY,0,1,0,CLIENT);
  INIT(LOWER,0,0,0,CLIENT);
  INIT(NOT_LOWER,0,0,0,CLIENT);
  INIT(SEND,1,0,0,ISP);
  INIT(WRITE,1,0,0,ISP);
  INIT(CANCEL,0,0,1,ISP);

 do
   ::
    if
     :: B_E(PAY1,S) ;
     :: B_E(DELAY,S) -> B_E(NOTIFY,S);
     if
        :: B_E(LOWER,S)
           -> B_E(PAY2,S) ;
        :: B_E(NOT_LOWER,S)
           -> B_E(PAY3,S) ;
     fi;
     :: B_E(SEND,S) -> B_E(WRITE,S)
       -> B_E(CANCEL,S);
    fi;
 od;
}

proctype CRM()
{
 do
     :: CONTRACT(PAY1,S);
     :: CONTRACT(PAY2,S);
     :: CONTRACT(PAY3,S);
     :: CONTRACT(DELAY,S);
     :: CONTRACT(NOTIFY,S);
     :: CONTRACT(LOWER,S);
     :: CONTRACT(NOT_LOWER,S);
     :: CONTRACT(CANCEL,S);
     :: CONTRACT(SEND,S);
     :: CONTRACT(WRITE,S);
 od;
}

init
 {
  atomic
  {
    run BEG();
    run CRM();
  }
}
```

Fig. 7. Initialisation, BEG and CRM.

```
RULE(PAY1)
{
 if
    ::IS_O(PAY1,CLIENT)->
        SET_O(PAY1,0);
        Payment = 1;
        RULE_DECI(CCO,CND);
    :: else
    RULE_DECI(NCC,CON);
 fi;
}
RULE(PAY2)
{
 if
    ::IS_O(PAY2,CLIENT)->
        SET_O(PAY2,0);
        Payment = 2;
        RULE_DECI(CCO,CND);
    :: else
    RULE_DECI(NCC,CON);
 fi;
}
RULE(PAY3)
{
 if
    ::IS_O(PAY3,CLIENT)->
        SET_O(PAY3,0);
        Payment = 3;
        RULE_DECI(CCO,CND);
    :: else
    RULE_DECI(NCC,CON);
 fi;
}
RULE(DELAY)
{
if
  ::IS_O(DELAY,CLIENT)->
    d_step
    {
       SET_O(DELAY,0);
       SET_O(PAY1,0);
          synch(NOTIFY)
            -> SET_O(LOWER,1);
            -> SET_R(NOT_LOWER,1);
          hcnys(NOTIFY)
     }
     RULE_DECI(CCO,CON);
   ::else
    RULE_DECI(NCC,CON);
fi;
}
```

```
RULE(NOTIFY)
{
if
  ::IS_O(NOTIFY,CLIENT)->
    d_step
    {
     SET_O(NOTIFY,0);
     SET_O(PAY1,0);
        synch(DELAY)
          -> SET_O(LOWER,1);
          -> SET_R(NOT_LOWER,1);
        hcnys(DELAY)
    }
    RULE_DECI(CCO,CON);
  ::else
    RULE_DECI(NCC,CON);
fi;
}

RULE(LOWER)
{
if
    :: IS_O(LOWER,CLIENT)->
       d_step
       {
           SET_O(LOWER,0);
           SET_R(NOT_LOWER,0);
           INTERNET_HIGH=FALSE;
           INTERNET_LOW =TRUE;
           SET_O(PAY2,1);
       }

        RULE_DECI(CCR,CON);
    :: else
    RULE_DECI(NCC,CON);
fi;
}

RULE(NOT_LOWER)
{
 if
    :: IS_R(NOT_LOWER,CLIENT)->
        d_step
        {
            SET_R(NOT_LOWER,0);
            SET_O(LOWER,0);
            SET_O(PAY3,1);
        }
        RULE_DECI(CCR,CON);
    :: else
        RULE_DECI(NCC,CON);
fi;
```

```
RULE(WRITE)
{
if
  ::IS_R(WRITE,ISP)->
    d_step
    {
     SET_R(WRITE,0);
     synch(SEND)
        -> SET_P(CANCEL,0);
        -> SET_R(CANCEL,1);
     hcnys(SEND);
    }
    RULE_DECI(CCR,CON);
  :: else
    RULE_DECI(NCC,CON);
fi;
}

RULE(SEND)
{
if
  ::IS_R(SEND,ISP)->
    d_step
    {
     SET_R(SEND,0);
        synch(WRITE)
          -> SET_P(CANCEL,0);
          -> SET_R(CANCEL,1);
        hcnys(WRITE);
    }
    RULE_DECI(CCR,CON)
  :: else
    RULE_DECI(NCC,CON);
 fi;

}

RULE(CANCEL)
{
if
  ::IS_R(CANCEL,ISP)->
    SET_R(CANCEL,0);
    RULE_DECI(CCR,CNL)
  ::IS_P(CANCEL,ISP)->
    RULE_DECI(CCP,CON)
  ::else
    RULE_DECI(NCC,CON);
fi;
}
```

Fig. 8. Contract rules.

V. Evaluation of the Model

The case study is intended to demonstrate that the CB2B model provides primitives for encoding a contract for model checking directly as ECA rules in terms of contract entities: business operations, the role players with their rights, obligations and prohibitions. These primitives that we have added to PROMELA can be used to build contract models at different levels of abstractions aimed at exploring specific properties of the system. Abstraction is routinely used by designers to reduce: a) the size of the state space of the model, b) its complexity, or c) both [14].

The case study demonstrates how we use *restriction* to reduce the size of the state space and the complexity of the model (restriction called *slicing* in [15]). The size of the state space and the 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. Let us use M^r, A^r and A^r_s to refer to the restricted model, its automaton and state space, respectively. Notice that in M^r we have strengthened the guards in the BEG so that the BEG generates only *matched events*, that is, events that satisfy conditions *C1* and *C2*. 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, easy to reason about and amenable to exhaustive verification in no time; it is focused on the exploration of an specific part of the state space A_s; 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.

Once we have confidence in the correctness of the contract, one very useful function the model can serve is for generating executable test cases for testing an implemented system. In a separate paper we discuss this way of testing electronic contracts [6].

VI. Related work

Number of papers on the subject of formal specification contracts have been published [16], [17], [18], [19]. These approaches consist of formal languages which would be hard to use by business process developers. Other works like [4], [20] aim to provide notations that are more suited to the needs of business process developers, but of necessity sacrifice the rigour of the formal approaches. Our approach is somewhere in between the two, and provides a practical tool for the specification and verification of contracts. To our knowledge, few works have specifically considered the problem of model checking of contracts [21], [13] [22], [7], [11].

In [21], automata are used to model the behaviours of the contract participants. [13] uses PROMELA directly to model CCC and the rules; the present paper takes that approach further by extending PROMELA with primitives for representing contracts in a high level manner. In [7] the NuSMV model checker is used to model and verify the properties of contracts written in the contract language *CL* [19]. In [11] a tool called CLAN is implemented for performing automatic analysis of conflicting clauses written in *CL* language. The natural language document specifying the contract is converted into *CL* first, and then analysed using the CLAN tool.

In our work, we rely on an existing, well known model checking tool (SPIN). The notations of the CB2B model provide a relatively simple way of writing contracts in ECA , a notation that can be used by business process developers and others not familiar with model checking and formal specification notations. The CB2B model hides away the technical details of the processes and the communication channels and treats a contract simply as a set of ECA rules. The ECA rules from the CB2B model can be translated relatively easy into executable specifications, such as that supported by the EROP language.

VII. Concluding Remarks

We have described a model-checking based tool implemented by extending Promela. The CB2B model of the tool hides much of the intricate details of constructing interacting state machines and enables a designer to encode a contract for model checking directly as ECA rules in terms of contract entities: business operations, role players with their rights, obligations and prohibitions. Equally important, the designer can specify the correctness requirements, in linear temporal logic, directly in terms of the contract entities. The primitives that we have added to PROMELA are at user level (that is, without any changes to the SPIN source code) and can be used to build contract models at different levels of abstractions aimed at exploring specific properties of the system. In our model, business operations are categorized as valid (satisfying condition C1), matched (satisfying conditions C1 and C2) and contract compliant (satisfying conditions C1 and C2 and C3). The model's event generator, BEG, can be programmed to generate any combination of these. From the point of view of model checking, a particularly important case would be generation of sequences of events corresponding to the execution of matched operations only. This considerably reduces the size of the state space for exploration yet enables checking that rules are responding correctly to contract compliant business operations. One can then extend the state space of exploration, for example by reprogramming the BEG to generate events satisfying condition $C1$ but not necessarily $C2$. CB2B model provides convenient way of performing such experiments.

References

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

[2] M. Ben-Ari, *Principles of the Spin Model Checker*. Springer, 2008.

[3] C. Molina-Jimenez, S. Shrivastava, and M. Strano, "A model for checking contractual compliance of business interactions," *IEEE Transactions on Services Computing*, vol. Preprint, no. 99, 2011.

[4] M. Strano, C. Molina-Jimenez, and S. Shrivastava, "A rule-based notation to specify executable electronic contracts," in *Proc. Int'l Symposium on Rule Representation, Interchange and Reasoning on the Web*, ser. RuleML '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 81–88.

[5] ——, "Implementing a rule–based contract compliance checker," in *Proc. 9th IFIP Conf. on e-Business, e-Services, and e-Society (I3E'2009)*. Nancy, France: Springer, 2009, pp. 96–111.

[6] A. Abdelsadiq, C. Molina-Jimenez, and S. Shrivastava, "On model checker based testing of electronic contracting systems," in *Commerce and Enterprise Computing (CEC), 2010 IEEE 12th Conf. on*, nov. 2010, pp. 88 –95.

[7] G. Pace, C. Prisacariu, and G. Schneider, "Model checking contracts: a case study," in *Proc. 5th Int'l Conf. on Automated technology for verification and analysis*, ser. ATVA'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 82–97.

[8] RosettaNet, "Implementation Framework, Version V02.00.01 – High Availability Features – Technical Recommendation," 2004. [Online]. Available: http://www.rosettanet.org/

[9] OASIS, "ebxml business process specification schema technical specification v2.0.4, OASIS standard, 21 dec." 2006. [Online]. Available: http://docs.oasis-open.org/ebxml-bp/2.0.4/OS/spec/ebxmlbp-v2.0.4-Spec-os-en.pdf

[10] 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.

[11] S. Fenech, G. J. Pace, and G. Schneider, "Clan: A tool for contract analysis and conflict discovery," in *ATVA*, 2009, pp. 90–96.

[12] T. C. Ruys, "Towards effective model checking," Ph.D. dissertation, University of Twente, Enschede, March 2001.

[13] C. Molina-Jimenez and S. Shrivastava, "Model checking correctness properties of a middleware service for contract compliance," in *Proc. of the 4th Int'l Workshop on Middleware for Service Oriented Computing (MW4SOC'09)*. Nov. 30, Urbana–Champaign, USA: ACM digital library, 2009, pp. 13–18.

[14] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, P. Petit, L. Petrucci, P. Schnoebelen, and P. McKenzie, *Systems and Software Verification*. Springer, 1999.

[15] L. I. Millett and T. Teitelbaum, "Slicing promela and its applications to model checking, simulation, and protocol understanding," in *Proc. Spin98 Workshop*, 1998, pp. 1–9.

[16] M. Bravetti and G. Zavattaro, "Towards a unifying theory for choreography conformance and contract compliance," in *Pre-proceedings of 6th Symposium on Software Composition*. Springer, 2007, pp. 34–50.

[17] M. G. Buscemi and U. Montanari, "Cc-pi: A constraint-based language for specifying service level agreements," in *ESOP, volume 4421 of LNCS*. Springer, 2007, pp. 18–32.

[18] H. Davulcu, M. Kifer, and I. V. Ramakrishnan, "Ctr-s: A logic for specifying contracts in semantic web services," 2004.

[19] C. Prisacariu and G. Schneider, "A formal language for electronic contracts," in *Proc. 9th IFIP WG 6.1 Int'l Conf. on Formal methods for open object-based distributed systems (FMOODS'07)*, 2007, pp. 174–189.

[20] E. Martínez, G. Díaz, M. E. Cambronero, and G. Schneider, "A model for visual specification of e-contracts," in *Proc. IEEE Int'l Conf. on Services Computing (SCC'10)*, Washington, DC, USA, 2010, pp. 1–8.

[21] E. Solaiman, C. Molina-jimenez, and S. Shrivastava, "Model checking correctness properties of electronic contracts," in *Proc. Int'l Conf. on Service Oriented Computing (ICSOC'03)*. Springer-Verlag, 2003, pp. 303–318.

[22] A. Daskalopulu, "Model checking contractual protocols," in *13th Annual Conf., Frontiers in Artificial Intelligence and Applications Series*. IOS Press, 2000, pp. 35–47.