Dynamic Testing and Deployment of a Contract Monitoring Service

Ellis Solaiman[1], Ioannis Sfyrakis[1] and Carlos Molina-Jimenez[2]
[1]*School of Computing Science, Newcastle University, Newcastle, U.K.*
[2]*Computer Laboratory, University of Cambridge, Cambridge, U.K.*
{*ellis.solaiman, i.sfyrakis*}*@ncl.ac.uk, carlos.molina@cl.cam.ac.uk*

Keywords: Service Agreement, Electronic Contract, Service Monitoring, Model Checking, Automated Testing, Service Oriented Computing, Cloud Computing.

Abstract: Internet and cloud based services involve electronic interactions that are normally regulated using service agreements (SA). Once an agreement between business partners is in place, a service can be monitored and/or enforced using an SA equivalent electronic contract. Because of the dynamic nature of such Internet and cloud based relationships, the rapidity at which electronic contracts are constructed, verified for correctness, tested, and deployed is an extremely important factor. This paper describes a model checker based framework for supporting the automated testing and deployment of electronic contracts. The central components of the framework are a contract monitoring service called the *Contract Compliance Checker* (CCC), the SPIN model checker, and EPROMELA, a language developed specifically for modeling electronic contracts. We describe how SPIN can be used to automatically generate execution sequences from an EPROMELA model of a contract, and how such sequences can then be used to test the correctness of the model equivalent electronic contract deployed to the CCC.

1 INTRODUCTION

Internet and cloud computing advances have made it possible for businesses to provide infrastructure and software services to their business partners and to their customers at affordable costs. Before such business relationships can commence, legal service agreements (SA) need to be negotiated and agreed. Legal agreements, explicitly define the permissible actions of the interacting parties, thus providing a legal basis for the resolution of any disputes. A Legal agreement can also be used as a guide for developing an electronic contract (Molina-Jimenez et al., 2003).

The main purpose of an electronic contract is to regulate (monitor and/or enforce) electronic service exchanges between the contracted parties, making sure that business participants adhere to the SA in place, and that performed actions comply with various message timing and sequencing constraints. Electronic contracts are not confined to the business domain, and can also be used for example to monitor/enforce SAs between the components of distributed systems in the cloud and/or the "Internet of Things".

Constructing an electronic contract that is correct (free from conflicts, and which correctly represents the requirements of the original legal document), is a challenging and time consuming task. Cloud based business relationships can be both complex and of a highly dynamic nature (Molina-Jimenez et al., 2011) therefore it is important that the process of converting a legal document into an electronic contract that is correct, is automated as much as possible. Previous work towards this goal has been extensive, and has covered problems such as electronic contract representation and modeling (Strano et al., 2008), and contract model verification (Solaiman et al., 2003) (Abdelsadiq et al., 2011). Naturally, ensuring that a model of an electronic contract is correct, does not guarantee that the electronic contract itself is also correct. In this paper, we focus on the challenge of ensuring that an electronic contract acts correctly at run time, and that modifications and/or corrections that need to be made to the rule base of the electronic contract can be applied quickly. To this end, we develop a model checker based framework to support automatic electronic contract deployment and testing.

The central component of our framework is the *contract compliance checker (CCC)* (Fig. 1) (Strano et al., 2009) (Molina-Jimenez et al., 2012), which essentially is the System Under Test (SUT). The CCC is an independent contract monitoring service that when provided with an executable specification of a con-

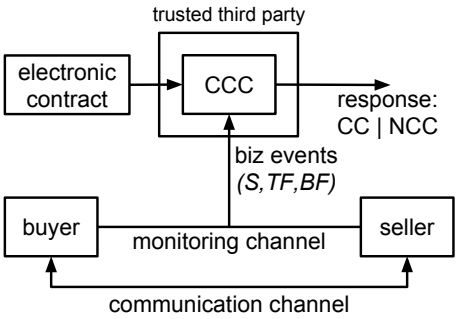

Figure 1: The CCC deployed as a contract monitor.

tract, can be deployed by the contracted parties or by a third party. The CCC is able to observe and log relevant interaction events, which it processes to determine whether the actions of the business partners are consistent with respect to the rights, obligations, and prohibitions declared in the original legal contract. Namely, the CCC declares interaction events as either contract compliant (*CC*) or non contract compliant (*NCC*).

As can be seen in Fig 1, business partners use a communication channel for exchanging their business messages. In addition they use a monitoring channel for notifying events of interest to the CCC. Notably, the figure shows that the CCC can cope with exceptions and failures, observing events that have been declared by the interacting parties as either *S (successful), TF (technical failure), or BF (business failure).*

The ability of the CCC to correctly declare interaction events as (*CC*) or (*NCC*) relies on an executable contract that has been specified correctly. Our goal is to provide a framework that enables; rapid testing of a deployed executable contract, and rapid update of the contract rules when testing detects errors. To do so, one must be able to exhaustively supply the CCC with execution sequences that it would be expected to observe during runtime. Our approach is to resort to model checker based testing.

Previous research into the area of model checker based testing of electronic contracts, (Abdelsadiq et al., 2010), describes the basic idea: construct a behavioural model of the SUT and validate the behaviour using a model checker. Such a validated model can then be used for generating executable test cases for the SUT.

The model checking tool we use is SPIN (Holzmann, 2003), a tool originally designed for the verification of communication protocols. SPIN's input language, Promela, provides constructs for modeling communication concepts such as messages, channels, and basic data types that include bit, bye, arrays. etc. Using these basic constructs alone for modeling electronic contracts, at a sufficiently high level of abstraction, is extremely challenging. This in turn makes the process of generating accurate execution sequences required for testing the CCC difficult.

To address these challenges, a fundamental component of our testing framework is EPROMELA, a high level language developed specifically for modeling electronic contracts (Abdelsadiq et al., 2011). EPROMELA extends Promela with constructs for expressing core electronic contract concepts contained in the CCC, thus enabling the construction of a contract model at a level of abstraction that is equivalent to the actual electronic contract.

This paper makes two contributions: 1) we describe the architecture of the CCC service, highlighting architectural improvements we have made that allow for dynamic update of rules coded in an electronic contract specification. 2) we describe how SPIN, and EPROMELA, can be instrumented with the aid of appropriate automation and message parsing tools, to produce business events that can accurately test the executable contract deployed within the CCC service.

The remainder of the paper is structured as follows: In Section 2 we describe key electronic contracting concepts with the aid of a simple example. In Section 3 we present the enhanced architecture of the CCC. Section 4 is dedicated to presenting our model checker based testing framework. In Section 5 we discuss related work. Conclusions and future work suggestions are presented in Section 6.

2 BACKGROUND

In order to elaborate key electronic contracting concepts, we present a simple scenario. Let us assume that Fig. 1 describes a relationship where two organisations, a Buyer and a Seller (a store), agree to a business contract. Below are some of its clauses:

1. *The buyer can place a **buy request** with the store to buy an item.*
2. *The store is obliged to respond with either **buy confirmation** or **buy rejection** within 3 days of receiving the buy request.*
   (a) *No response from the store within 3 days will be treated as a buy rejection.*
3. *The buyer can either **pay** or **cancel** the buy request within 7 days of receiving a confirmation.*
   (a) *No response from the buyer within 7 days will be treated as a cancellation.*

The clauses of such a legal agreement should take into consideration all relevant business operations (shown

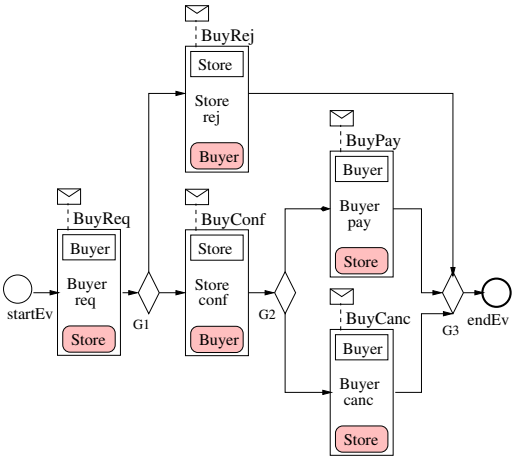

Figure 2: Correct choreography of contract example.

in bold in the contract text). A contract distinguishes operations as *Rights*, *Obligations*, *Prohibitions* (the *ROP* set). A *Right* is an operation that a party is allowed to perform under certain conditions, an *Obligation* is an operation that a party is expected to do under certain conditions, and a *Prohibition* is an operation that a party is not allowed to do under certain conditions.

To support our discussion, we will use a graphical representation of the contract written in BPMN (Business Process Management Notation) choreography language (OMG, 2011) (see Fig. 2).

The figure involves five activities, each resulting in a message (*BuyReq*, *BuyRej*, *BuyConf*, *BuyPay*, *BuyCanc*) being sent from a sender (shown as a white label in each activity), to a receiver (shown as a shaded label). These messages correspond to the five business operations (buy request, buy reject, buy confirmation, buy payment, buy cancellation) shown in bold in the English text of the contract. The diamonds in the figure are gateways. The figure includes two exclusive fork gateways (G1 and G2) and a single exclusive merge gateway (G3).

The choreography specification describes, from a global perspective, all permissible message sequences that can be exchanged between the partners, and is used by the interacting parties for two purposes: i) designing and implementing their individual parts of the business process; and ii) it is also very useful as a guide for developing the electronic contract.

The electronic contract designer is able to use the legal contract and choreography, to accurately identify and extract the ROPs attributed to the business partners, and to specify the rules which operate on the ROP set (Molina-Jimenez and Shrivastava, 2013). Rule implementation requires an appropriate specification language; contract rules written for the CCC monitoring service are currently realised using the Drools Rule Language (DRL) (RedHat, 2013).

An example of a rule that deals with receipt of a *buy request* event by the CCC, written using Drools can be seen below. Line 5 checks that the *buyRequest* operation is a right that the buyer is currently allowed to perform. If so then *buyRequest* is declared by the CCC as contract compliant (line 13). This operation is also removed from the buyer's ROP set (line 8), meaning that the buyer no longer has a right to perform this operation. At lines 10 and 11, the seller is given an obligation to perform one of 2 operations: *buyConfirm*, or *buyReject*.

```
1 rule "Buy Request Received"
2 //Verify type of event, originator, and
    responder
3 when
4  $e: Event(type=="BUYREQ",
    originator=="buyer", responder=="store",
    status=="success")
5  eval(ropBuyer.matchesRights(buyRequest))
6 then
7  //Remove buyer's right to place other Buy
    Requests
8  ropBuyer.removeRight(buyRequest, seller);
9  //Add seller's obligation to either accept
    or reject order
10 BusinessOperation[] bos = {buyConfirm,
    buyReject};
11 ropSeller.addObligation("React To Buy
    Request", bos, buyer, 60,2);
12 System.out.println("* Buy Request Received
    rule triggered");
13 responder.setContractCompliant(true)
14 end
```

Each of the activities declared in the choreography of Fig. 2 has a rule such as the one shown above. Typically, for each activity in a choreography, each business partner can have several rights, obligations, and prohibitions in force.

Once an electronic contract specification has been completed, it can be loaded into the CCC for deployment and testing. As operations are executed, and events are received by the CCC, rights, obligations and prohibitions are granted to and revoked.

The CCC processes each event to determine if it is contract compliant *(CC)* or none contract compliant *(NCC)*. The execution of a business operation (observed from the outcome event) is said to be *CC* if it satisfies the following three conditions and is said to be *NCC* if it does not: 1) it matches an operation within the set of business operations expected by the CCC, 2) it matches the ROP set of its role player (meaning, the role player that performed the operation has a right/obligation/prohibition to perform that particular operation), and 3) it satisfies the constraints stipulated in the contractual clauses. An example of a

constraint is the seven day deadline in clause 3 of the contract discussed earlier.

We also consider that the execution of a given sequence of operations is *NCC* if it includes one or more operations that are flagged by the CCC as *NCC*. A sequence of operations is also known as an *execution sequence* or *execution trace*, and drives the choreography from its initial state to a final state.

To ease the introduction of basic concepts, our legal contract example and corresponding choreography of Fig. 2, deal with successful outcome events only. However, a contract monitoring service such as the CCC should also be able to observe outcome events that include exceptional circumstances (Molina-Jimenez et al., 2009). Therefore, following the ebXML standard (OASIS, 2006), we assume that at the end of a business conversation, each party independently declares an execution outcome event from the set {*Success(S)*, *BizFail(BF)*, *TecFail(TF)*} as shown in Fig. 1. *Success* events model successful execution outcomes. *TecFail* models protocol related failures detected at the middleware level, such as a late, or a syntactically incorrect message. *BizFail* models semantic errors in a message detected at the business level, e.g., the credit card details extracted from the received payment document are incorrect.

Adding exceptional outcome events to the CCC's set of observable events, naturally means that the CCC has to monitor a much larger number of execution sequences. The task of generating these in order to test the CCC effectively is extremely challenging, and strengthens the case for needing to automate the testing process. Before moving on to our testing framework, let us first take a look at the new architecture of the CCC.

3 ARCHITECTURE OF THE CONTRACT COMPLIANCE CHECKER (CCC)

The overall architecture of the CCC is shown in Fig. 3. It consists of two layers: The *CCC Engine* (The *Logical Layer*), and the new addition is the *CCC Service* (The *Presentation Layer*). The *CCC Engine* is responsible for processing business events and for determining whether they are contract compliant or not. The *CCC Service* is an interface to the *CCC Engine*, it is used for delivering business events to the CCC, and for collecting the corresponding responses. In addition, the *CCC Service* can be used by the rule administrator for loading and editing the rules that represent the contract. The functionality of the architecture is as follows: A business event is received through the *monitoring channel* as an XML document that includes the names of the participants, the business operation, and its outcome from the set: *(Success, BizFail, TecFail)*:

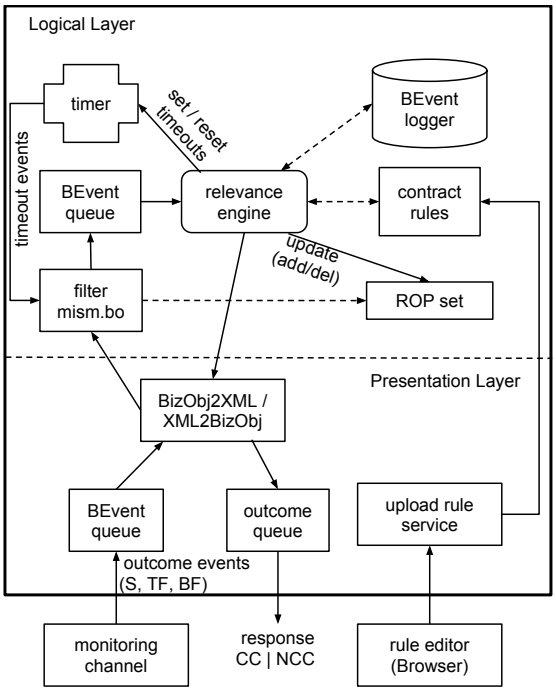

Figure 3: Architecture of the contract compliance checker.

```
<event>
  <originator>buyer</originator>
  <responder>seller</responder>
  <type>BuyReq</type>
  <status>success</status>
</event>
```

The event shown here is produced as a result of the implementation of a conversation synchronization protocol between the interacting parties. The protocol guarantees mutually agreed conversation outcomes. It is the responsibility of the interacting partners to apply the protocol. A detailed discussed can be found in (Molina-Jimenez et al., 2007). The XML document representing the business event is passed to the *BEvent queue*. Business events are retrieved, and converted using the *xml2BizObj/BizObj2xml Convertor* from their XML format into business event objects. Events are then passed to the *CCC Engine*. The *filter mism.bo*, discards mismatched business events that are not among the permitted events defined within the ROP set. Business events that pass this filter are inserted into the *BEvent queue*. All deadlines are set and reset by the *relevance engine*, and enforced by the *timer*. Timeout events are added to the *filter mism.bo* as required by the contract, and are examined by the

filter to decide if *bevents* are mismatched. For example, receiving a *buy confirmation* event from the store after the 3 day deadline has elapsed, will be treated as mismatched. The *relevance engine* removes a business event from the head of the *bevent queue* and compares it to the rules stored in the *contract rules*. Rules that match the *bevent* under examination are triggered to determine if their conditions are satisfied. The actions of the rules whose conditions are satisfied are executed, and this may alter (*add/del*) the current state of the *ROP sets*. For our example in Fig. 2, a rule triggered by a *BuyConf* business event and finds its conditions satisfied will delete (disable) the store's right to execute another *BuyConf* business operation, and delete the store's right to execute a *BuyRej* operation. The rule also will add (impose) an obligation on the buyer to either initiate the execution of a *BuyPay*, or initiate the execution of *BuyCanc*. The *bevent* is then stored in the *BEvent logger* as a record for any future dispute resolution. The *relevance engine* eventually declares the business event either *CC* or *NCC* and produces a response as a business object, which is sent out to the *Presentation Layer*. The business object passes through the *xml2BizObj/BizObj2xml Converter*, where it is serialized into an XML message of the following format:

```
<result>
  <contractcompliant> true|false
   </contractcompliant>
</result>
```

The *xml2BizObj/BizObj2xml Converter* inserts the response into the *outcome queue*, which can be accessed by the contracted parties. The *Presentation Layer* allows a "rule manager" to update the *contract rules* at run time. For this purpose, rules can be edited using the *rule editor (in a browser)* and sent to the *rule upload service* as a conventional RESTful POST operation. The *rule upload service* is responsible for producing a *drl* (Drools) file (for example *new–rules.drl*) from the payload of the POST operation, and for uploading it to the *CCC Logical Layer* to replace the *Contract Rules*.

The *CCC Logical Layer* is implemented using JBoss's Drools rules Engine (version 6.1 as of the year 2014) (RedHat, 2013). The Drools rules engine powers the decision making capabilities of the *relevance engine*. The *relevance engine*, acts as a wrapper for the Drools rule engine and its responsibilities include the initialisation of the contract, as well as the addition and processing of events received from the *Presentation Layer*.

The *Presentation layer*, a new addition to the CCC, exposes the CCC as a RESTful web service. Its aim is to enable the exchange of XML event messages between the CCC and the contracted clients, and to ease the editing and update of the contract rules (these were previously hard coded). The *Presentation Layer* is implemented using the JBoss Enterprise Application Platform (EAP), (RedHat, 2014). The *BEvent queue* and the *outcome queue*, are implemented using JBoss's HornetQ (a message oriented middleware layer), and using the Java Message Service (JMS) API. A Message Driven Bean (MDB) receives business events from HornetQ and passes them to the *XML2BizObj/BizObj2XML* converter, which is implemented using Java. The *upload rule service* is part of the Drools Workbench– a web authoring and rules management application.

4 MODEL CHECKER BASED TESTING

To claim categorically that the CCC functions correctly, we need to test that it can correctly identify contract compliant and non-contract compliant executions of sequences and their constituent business operations. To this end, one needs to be able produce sequences of operations that are known to be contract compliant, and also produce sequences that include both contract compliant and non contract compliant operations.The challenge here is the production of such sequences.

4.1 Testing Framework

Fig. 4 shows the main elements of our testing framework. Squares with smooth corners represent humans involved in the design process. Tools are represented by solid squares with sharp corners, and dashed squares represent data.

As stated earlier, a central component of our testing framework is the SPIN model checker. SPIN models are constructed using Promela, and specifically using EPROMELA, a modeling language developed precisely for modelling electronic contracts (Abdelsadiq et al., 2011). EPROMELA is essentially a high level tool that extends Promela with constructs for expressing core electronic contract concepts contained in the CCC. Correctness properties that an EPROMELA model is expected to satisfy, can be expressed by the model designer using Linear Temporal Logic (LTL). When provided with a model of the contract and appropriate LTL properties, SPIN is able to verify the correctness of the model with respect to those properties. With the aid of tools for message parsing and automation, SPIN also can

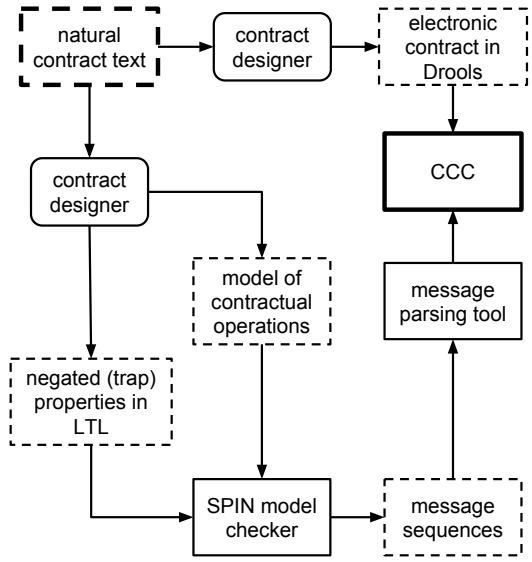

Figure 4: Model Checker based testing framework.

be instrumented to generate message sequences that can be used to test the ability of the CCC to detect contract compliant and non contract compliant message sequences, a process that we will describe next. Model checker based sequence generation follows these steps:

1. The designer constructs an abstract model of the System Under Test (SUT), and verifies that the model is correct in that it satisfies the correctness properties of interest.
2. The verified abstract model is used for generating execution sequences. This is done by presenting the verification tool with the verified abstract model, together with a negated correctness requirement in LTL (a trap property), and then challenging the verification tool to find and produce counter examples that violate the LTL.
3. Each counter example contains an execution sequence that can be extracted with the aid of a message parsing tool.

4.2 Example

We begin by building an EPROMELA model of our example contract presented in the Background section. To ease the task of parsing the counter examples of interest, we include within the EPROMELA model print statements that produce the required XML events. The end of each execution sequence is marked using a *reset message*).

4.2.1 Model Construction and Verification

Below is a section of our EPROMELA contract model, which includes the rule that deals with the *BUYRREQ* operation of Fig. 2. Each of the operations for the choreography in Fig. 2 has a rule which updates the status of the ROP set belonging the participants as they transition from state to state.

```
1 RULE(BUYREQ)
2 {
3 WHEN::EVENT(BUYREQ,
    IS_R(BUYREQ,BUYER),SC(BUYREQ))->{
4   SET_X(BUYREQ,BUYER);
5   atomic{
6   printf("<originator>buyer</originator>");
7   printf("<responder>store</responder>");
8   printf("<type>BUYREQ</type>");
9   printf("<status>success</status>");
10 }
11   SET_R(BUYREQ,0);
12   SET_O(BUYREJ,1);
13   SET_O(BUYCONF,1);
14   RD(BUYREQ,BUYER,CCR,CO);
15 }
16 END(BUYREQ);
```

Line 3 of the model deals with receiving a successful buy request event `SC(BUYREQ)`. `IS_R(BUYREQ,BUYER)` is a guard that checks if the *BUYER* has a right to perform the *BUYREQ* operation. If so, then `SET_X(BUYREQ,BUYER)` declares that this operation has been executed, and the buyer's right to execute *BUYREQ* is removed at line 11. The rule then sets an obligation to the Store to execute either *BUYREJ* or *BUYCONF* (lines 12 - 13). At line 6 we introduce the print statements required for parsing the generated execution sequences. The print statements produce XML events in the format expected by the CCC.

Each of the operations *BUYREQ, BUYREJ, BUYCONF, BUYPAR, BUYCANC*, has a rule such as the one above. Events are generated using an EPROMELA *Event Generator* module, which models the interaction between the business partners. Events exercise rules such as the one above through another EPROMELA module known simply as the *Rule Manager*. For a full description, see (Abdelsadiq et al., 2011). When the entire EPROMELA model has been constructed, SPIN can be used to verify that the model is free from any inconsistencies. Common correctness properties such as absence of deadlocks and reachability of states, can easily be checked using SPIN's configuration options. Checking for contract specific correctness properties however, requires the application of Linear Temporal Logic (LTL) formulae. Typical correctness properties of the electronic contracting domain are those that express mutual ex-

clusion of rights, obligations, and prohibitions; for example the requirement that the execution of a given operation (such as making a purchase order) is never simultaneously obliged and prohibited. Thanks to the contract constructs offered by EPROMELA, this correctness requirement can be elegantly and intuitively expressed in LTL as follows:

```
[]!(IS_O(BUYREQ, BUYER) && IS_P(BUYREQ, BUYER))
```

Where `[]` is the LTL `always` operator. `!` is the universal `not`, `IS_O(BUYREQ, BUYER)` returns true if the `BUYREQ` operation is currently obliged and `IS_P(BUYREQ, BUYER)` returns true if the `BUYREQ` operation is currently prohibited. Instructing SPIN to run through the EPROMELA model using this LTL, will drive SPIN to find any examples that violate this property. If such an example is found then it is presented as a counter example to the designer, who must then correct the model.

4.2.2 Generating the Test Sequences

Once the contract model has been verified for required correctness properties, it can be used as an oracle for producing sequences that can test the electronic contract. Test sequence generation is very similar to verification in that we make use of LTL properties. We can (as described in the previous section) instruct SPIN to find undesirable examples of sequences that violate a desirable property. But we also need to to be able to instruct SPIN to find desirable sequences that violate a non-desirable property. The latter is done by negating a desirable LTL property converting it into a *trap* property.

As a very simple example, let us instruct SPIN to generate all sequences of messages that end with a *BUYREJECT* operation. The LTL formulae required for this task is as follows:

```
!<>IS_X(BUYREJ,STORE)
```

Where `< >` is the LTL `eventually` operator. The formulae states that the model will `not eventually` reach a state where `BUYREJ` is executed. SPIN can now be instrumented to show all sequences that do end with `BUYREJ`. From the command line we apply the following steps (*CorrectChore* is the name if the file that contains the EPROMELA model):

1. `% spin -a CorrectChore` is used for generating the verifier source code in C.
2. `% cc -o pan pan.c` is used for compiling the verifier.
3. `% ./pan -a -e -c100` instructs SPIN to produce all the counter examples (trail files) that it can find, which violate the trap property. By default SPIN produces the first one it finds and stops. The `-c100` parameter instructs SPIN to generate the first 100 counter examples it finds. The number of counter examples requested needs to be above the actual number of counter examples that SPIN could possibly find. This number can be determine by the designer using trial and error.
4. `spin -tN -s -r -B CorrectChore` converts the N^{th} trail file into a text file that includes the XML messages involved in the execution sequence.

Given the potentially large number of trail files that can be produced by SPIN, it is advisable to mechanise the process. We use a simple shell script for this purpose. The following text represents the contents of one of the trail files produced by the Linux shell script. To ease readability, we have removed some irrelevant lines.

```
2: proc 0 (Buyer) line 35 "CorrectChore" Sent
    BuyReq,1
3: proc 1 (Store) line 71 "CorrectChore" Recv
    BuyReq,1

<originator>buyer</originator>
<responder>store</responder>
<type>BUYREQ</type>
<status>success</status>

5: proc 1 (Store) line 114 "CorrectChore"
    Sent BuyRej,1
6: proc 0 (Buyer) line 049 "CorrectChore"
    Recv BuyRej,1

<originator>store</originator>
<responder>buyer</responder>
<type>BUYREJ</type>
<status>success</status>

<originator>reset</originator>
<responder>reset</responder>
<type>reset</type>
<status>reset</status>
```

The execution sequence shown above includes a *BUYREQ* message sent from the buyer to the store, followed by *BUYREJ* sent by the store to the buyer. The status element indicates the outcome of the execution of the operation. The status in this example accounts only for successful execution outcomes (No exceptional circumstances such as technical failures are assumed), consequently, the content of this element is always *success*. The last message is the *reset* message, which we artificially include to mark the end of the sequence.

As can be appreciated from this example, the files produced by SPIN and the shell script need parsing in order to extract the XML tagged messages.

4.2.3 Sequence Parsing

Our parser is built using Python. It extracts all the XML tagged messages from a given sequence and stores each message as an individual XML file. The parser achieves this by creating a recursive grammar that describes the precise structure of the business events inside a sequence. As seen in the code segment below in lines 2 - 5, we first define the XML tags we want to find.

```
1 #define grammar for sequence file
2 tagOriginator = pyp.Literal("<originator>")
    + pyp.Word(pyp.alphas) +
    pyp.Literal("</originator>")
3 tagResponder = pyp.Literal("<responder>") +
    pyp.Word(pyp.alphas) +
    pyp.Literal("</responder>")
4 tagType = pyp.Literal("<type>") +
    pyp.Word(pyp.alphas) +
    pyp.Literal("</type>")
5 tagStatus = pyp.Literal("<status>") +
    pyp.Word(pyp.alphas) +
    pyp.Literal("</status>")
6 lineString = tagOriginator | tagResponder |
    tagType | tagStatus
```

The parser reads a file containing a message sequence, and searches for matches against each line according to the following rule in line 6: *If there is a line that includes a tag definition of either the originator, responder, type, or status, then the match is successful.* If the parser finds a match, then it performs the following actions: i) the parser creates a new folder with the name of the sequence, ii) it extracts the XML part that is matched according to the above rule, iii) a new XML file is created that includes the extracted business event. Thus, the folder *ExeSeq1–xml* for the sequence shown above will contain three XML files because the sequences contains three messages, namely $BUYREQ \rightarrow BUYREJ \rightarrow reset$.

4.2.4 Testing the Electronic Contract

After loading and initialising the CCC with the rules that encode the electronic contract, we can proceed with sending each of the execution sequences to the *BEvent queue*. Responses are collected from the *outcome queue* (see Fig. 3). The following lines show the results of testing the execution sequence $BUYREQ \rightarrow BUYREJ \rightarrow reset$:

```
1 filename: event1.xml
2 -Begin Request to CCC service-
3 BusinessEvent [originator=buyer,
    responder=store, type=BUYREQ,
    status=success]
4 -End Request to CCC service-
5
6 -Begin Response from CCC service-
7 <result>
8 <contractCompliant>true</contractCompliant>
9 </result>
10-End Response from CCC service-
11
12 filename: event2.xml
13 -Begin Request to CCC service-
14 BusinessEvent [originator=store,
    responder=buyer, type=BUYREJ,
    status=success]
15 -End Request to CCC service-
16
17 -Begin Response from CCC service-
18 <result>
19 <contractCompliant>true
   </contractCompliant>
20 </result>
21 -End Response from CCC service-
22
23 filename: event3.xml
24 -Begin Request to CCC service-
25 BusinessEvent [originator=reset,
    responder=reset, type=reset,
    status=reset]
26 -End Request to CCC service-
27 -Begin Response from CCC service-
28 <result>
29 <contractCompliant>true
   </contractCompliant>
30 </result>
31 -End Response from CCC service-
```

The operations (*BUYREQ* and *BUYREJ*) included in the sequence, are declared contract compliant by the CCC indicating that the contract rules have been coded correctly with respect to the LTL property in Section 4.2.2. The first operation is sent to the CCC in line 3, and its response `<contractCompliant>true` is shown at line 8. Similarly, *BUYREJ* operation is sent to the CCC at line 14, and its response `<contractCompliant>true` can be seen at line 19.

4.2.5 Testing None Compliant Events

A model that has been verified, will by default generate test sequences with events corresponding to the execution of contract compliant operations only. An EPROMELA model can be tuned to generate sequences which include unknown and noncompliant business events using the EPROMELA *Event Generator* module mentioned under Section 4.2.1. Thus we can alter the EPROMELA model to follow any variation of the choreography shown in Fig. 2. For example the modified choreography of figure Fig. 5 does not correctly reflect the original text contract.

The particularity of this diagram is that it produces contract compliant sequences such as $BuyReq \rightarrow BuyRej$. In addition, it produces non–contract compliant sequences, for instance it allows for cancellation after payment which is not stipulated in the orig-

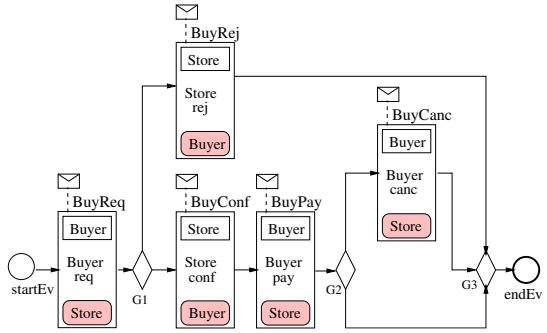

Figure 5: Incorrect choreography of contract example.

inal contract. Consequently, the execution of *BuyCanc* within the sequence *BuyReq* → *BuyConf* → *BuyPay* → *BuyCanc* should be declared non contract compliant by the CCC. The following text shows the results of the execution of the non–contract compliant sequence discussed above. The first 2 events *BUYREQ*, *BUYCONF*, were declared contract compliant by the CCC as expected. To save space we only show the outcome of the 2 events of relevance in this example (*BUYPAY* followed by *BUYCANC*):

```
1 filename: event3.xml
2 -Begin Request to CCC service-
3 BusinessEvent [originator=buyer,
    responder=store, type=BUYPAY,
    status=success]
4 -End Request to CCC service-
5
6 -Begin Response from CCC service-
7 <result>
8  <contractCompliant> true
    </contractCompliant>
9 </result>
10 -End Response from CCC service-
11
12 filename: event4.xml
13 -Begin Request to CCC service-
14 BusinessEvent [originator=buyer,
    responder=store, type=BUYCANC,
    status=success]
15 -End Request to CCC service-
16
17 -Begin Response from CCC service-
18 <result>
19  <contractCompliant> false
    </contractCompliant>
20 </result>
21 -End Response from CCC service-
```

The process *BUYPAY* is contract compliant (lines 3 and 8). The execution of *BUYCANC* at line 14 and the corresponding response received at line 19 indicates that the CCC has declared *BUYCANC* non–contract compliant. This is the desired behaviour from the CCC, as it has detected that this sequence of events is not consistent with the contract.

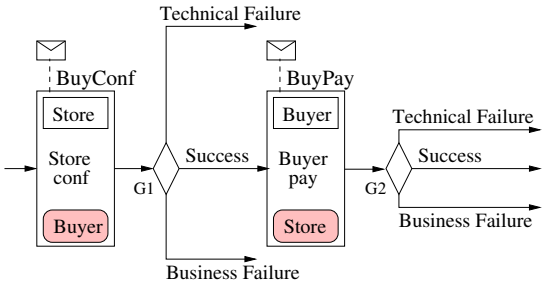

Figure 6: Execution model with success and failures.

4.3 Accounting for Exceptional Outcome Events

The contract example we have used so far assumes that the execution of operations always succeeds; it does not account for potential failures. More realistic examples would include the execution of activities as shown in Fig. 6, which account for successful and failed outcomes.

As discussed in Section 2, and following the ebXML standard (OASIS, 2006), we would like to be able to detect two types of failures; business failures, and technical failures. To this end, the EPROMELA modeling language has been designed with the ability to deal with these 2 types of failures. As an example of an electronic contract that can handle exceptional outcomes, we add the following clause to our original contract to account for potential semantic errors (*business failures*) in the execution of any operation:

4. *Failure handling: if after 2 attempts, an operation is not performed correctly, then the contractual interaction shall be declared terminated.*

Fig. 7 shows a partial choreography representation of the contract for three out of its five tasks only (for readability). The contract allows for a finite number of retries if business failures are encountered. The actual number of retries will normally be a configuration parameter. In the figure, *S* and *BF* stands for Success and Business Failure, respectively. Similarly, *rqBF*, *rjBF*, and *coBF*, represent counters that keep track of the number of failed executions of the operations; *BUYREQ*, *BUYREJ*, and *BUYCONF*, respectively. *N* represents an arbitrary integer that in our example allows for two failure execution ($N = 2$). The execution of each activity leads to a gateway with three outgoing arrows. As an example, at *BUYREQ*, a successful (*S*) execution leads to the normal execution of the contract, namely to *G2*. Alternatively, if the execution completes in *BF* and the number of failed executions *rqBF* of the *BUYREQ* operation is less than *N*, the execution is tried again. However, if the outcome is *BF* and it has already failed *N* times, the contractual

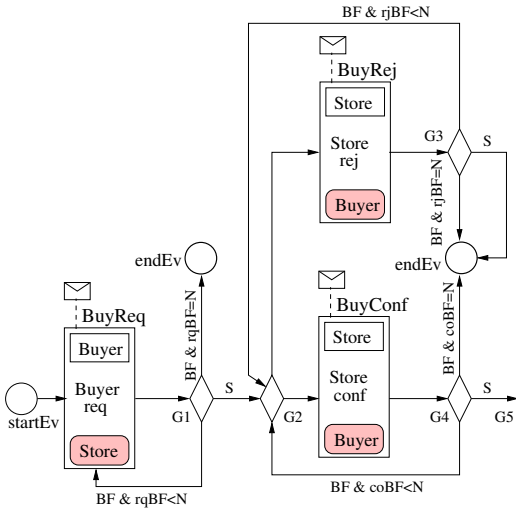

Figure 7: Contract example that accounts for failures.

interaction is terminated. Failure handling with the remaining activities is similar, except that gateways *G2* and *G5* introduce additional alternative execution paths. For instance, after failing to complete successfully *BUYREJ* at the first attempt, the initiator is allowed to choose *BUYREJ* again or alternatively can execute *BUYCONF*. Below we show how an exception such as the business failure of the *BUYREQ* operation described above can be intuitively and naturally modeled using EPROMELA. The rule for *BUYREQ* described in Section 4.2.1 can be easily enhanced as follows:

```
1 /*handle failure outcome event*/
2::EVENT(BUYREQ,IS_R(BUYREQ,BUYER),
    BF(BUYREQ))->{
3 atomic{
4 printf("<originator>buyer</originator>");
5 printf("<responder>store</responder>");
6 printf("<type>BUYREQ</type>");
7 printf("<status>bizfail</status>");
8 }
9 if /*1st notification of BF*/
10 ::(ReqFailBefore==NO)->
    ReqFailBefore=YES;
11 printf("First BUYREQ-BF");
12 RD(BUYREQ,BUYER,CCR,CO);
13 /*2nd notification of BF*/
14 ::(ReqFailBefore==YES)->
    abncoend=TRUE;
15 printf("Last BUYREQ-BF");
16 SET_R(BUYREQ,0);
17 atomic{
19 printf("<originator>reset</originator>");
20 printf("<responder>reset</responder>");
21 printf("<type>reset</type>");
22 printf("<status>reset</status>");
23 RD(BUYREQ,BUYER,NCCR,CND); /*abnormal
    contract end*/
```

The model can now also handle *BUYREQ* events that result in *BF* outcomes (line 2). If a failed event is received, then the rule checks if a failure of this kind has happened before. If not (line 10), then this first failure is registered, and contract execution is allowed to continue (line 12). On the other hand, if this is the second time *BUYREQ* has been received with a *BF* outcome then the rule terminates contract interaction at line 23.

The EPROMELA model includes rules like the one described above for dealing with each of the 5 business events shown in bold in our contract example. After the model has been verified using SPIN, the electronic contract deployed to the CCC can be tested, in combination with the testing framework described previously, using much more realistic execution sequences that include exceptions. A detailed description of how exceptions are handled in the CCC can be found in (Molina-Jimenez et al., 2009).

5 RELATED WORK

Research work on the monitoring of cross-organizational interactions between parties was pioneered by Minsky (Ungureanu and Minsky, 2000) with work on Law Governed Interaction (LGI). The notion of rights, obligations and prohibitions was introduced in (Ludwig and Stolze, 2003). A useful summary about various issues involved in contract management is provided in (Hvitved, 2010).

Linear Temporal Logic (LTL) is a powerful tool for specifying correctness properties in a model whether it is for verifying the correctness of the model, or for the generation of test sequences. However not all correctness properties can be expressed using LTL; for example it is not possible to specify that a particular property will hold for every 3rd or 4th state of the system. Such limitations are discussed in (Galton, 1987), where extensions to LTL are suggested. In addition, despite the advantages of model checker based testing, it does have its disadvantages; building a model of the SUT and describing the required LTL properties relies heavily on the skills of the technical person who must also be intimately familiar with the SUT. The advantages and disadvantages of model checker based testing are discussed in (El-Far, 2001) where the author provides a practical guide. Naturally it is difficult to ensure complete coverage of all possible system behaviors during testing with manually specified LTL properties. Therefore, it is extremely desirable to be able to systematically create complete test suites according to some test objective (Fraser et al., 2009). (Van der Aalst and Pesic, 2006) propose to automate the task of specifying LTL

properties by means of a graphical language (DecSerFlow) that is then mapped into LTL formulas. Using this language, the designer can specify a set of common or frequent correctness requirements.

It is worth noting at this point that we are in the process of developing an *LTL Manager* component for our testing framework. The *LTL Manager* includes a repository that can be populated with templates of LTL formulae for common contract related properties that must be satisfied by all contracts. For example that a business operation is not simultaneously prohibited and obliged at the same time.

Although model checker based testing techniques have been studied widely in the software engineering community (Utting and Legeard, 2006) (Pezze and Young, 2008) (Torsel, 2013), their use in the testing of a contract monitoring service has received little attention. The principles of model checker based testing of electronic contracts are investigated previously by us in (Abdelsadiq et al., 2010), however contract models in this work are built using Promela, the basic input language of SPIN. Attempting to predict how a designer would use basic Promela to model a contract is an impossible task, which makes developing tools for automating the testing process extremely difficult. An important contribution of this paper is that we highlight the benefits of developing a tool based framework that has been tailored specifically to leverage the capabilities of a domain specific modelling tool such as EPROMELA (Abdelsadiq et al., 2011), which was developed specifically for modeling electronic contracts.

6 CONCLUSION AND FUTURE WORK

Ensuring the correct functionality of an electronic monitoring service such as the *contract compliance checker (CCC)*, becomes more difficult as the number of execution sequences that the CCC is expected to observe increase. We have seen that cloud and Internet based interactions between business partners can indeed be extremely complex, and this is especially true when exceptional outcome events from these interactions are taken into consideration. Reproducing such complex exchanges in order to test the correct functionality of the CCC is difficult and cannot be achieved manually. In addition, it is extremely important to be able to correct and update the rule base of the monitoring service rapidly so that interruptions to the deployed service are reduced as much as possible.

In order to address these issues, we have presented a model checker based framework that includes tools to automate the testing process. By using the SPIN model checker in combination with EPROMELA, a high level modeling language designed specifically for modeling electronic contracts, we can build verified models that accurately resemble the System Under Test (SUT) with relative ease. By using appropriate LTL formulae within an EPROMELA model, we can instrument SPIN to automatically produce contract compliant, and none contract compliant execution sequences that are capable of exhaustively testing the correct operation of the CCC. In addition, we have presented a new and enhanced architecture and implementation of the CCC, with an additional *Presentation Layer* that exposes the CCC as a web service. An important feature is that the *Presentation Layer* includes a Drools editing and upload service that enables dynamic update of the electronic contract rules at runtime.

There are a number of future research directions which we are currently exploring. Drools, the language we use for specifying electronic contracts is verbose, and not as declarative and readable as would be ideal. We have developed a contract specification language called EROP (for Events, Rights, Obligations, and Prohibitions) (Strano et al., 2009), and are in the process of completing a tool for translating EROP to Drools. Also we would like to enhance the CCC, which currently acts as a passive monitor, with the capability to act as a contract enforcer. The aim of a contract enforcement service would be to ensure that an operation is executed only if it is contract compliant.

An important item for future work is to conduct experiments in order to determine how the presented testing framework performs as the number of possible events increases. The verification and testing of the CCC can be further automated in several ways; in addition to the development of the *LTL Manager* component discussed in Section 5, we would like to create a translation tool that can produce an EPROMELA model from an electronic contract specification written in EROP automatically. This would reduce the risk of introducing unwanted errors into the contract model during construction. We believe that this goal is achievable because of the semantic similarities between EPROMELA and the electronic contracting concepts within the CCC.

REFERENCES

Abdelsadiq, A., Molina-Jimenez, C., and Shrivastava, S. (2010). On model checker based testing of electronic contracting systems. In *IEEE International Confer-*

ence on Commerce and Enterprise Computing (CEC 2010). IEEE.

Abdelsadiq, A., Molina-Jimenez, C., and Shrivastava, S. (2011). A high level model checking tool for verifying service agreements. In *The 6th IEEE International Symposium on Service-Oriented System Engineering (SOSE 2011)*. IEEE.

El-Far, I. K. (2001). Enjoying the perks of model-based testing. In *Proc. of the Software Testing, Analysis, and Review Conference (STARWEST 2001)*.

Fraser, G., Wotawa, F., and Ammann, P. (2009). Testing with model checkers: A survey. *Software Testing, Verification and Reliability*, pages 215–261.

Galton, A. (1987). Temporal logics and computer science: An overview. *Academic Press*, pages ch. 1, pp. 2748.

Holzmann, G. J. (2003). *The Spin model checker: primer and reference manual*. AddisonWesley Professional.

Hvitved, T. (2010). A survey of formal languages for contracts. In *n Fourth Workshop on Formal Languages and Analysis of ContractOriented Software (FLACOS10)*.

Ludwig, H. and Stolze, M. (2003). Simple obligation and right model (sorm)-for the runtime management of electronic service contracts. In *2nd Intl Workshop on Web Services, eBusiness, and the Semantic Web (WES03) LNCS*, volume 3095, pages 62–76.

Molina-Jimenez, C. and Shrivastava, S. (2013). Establishing conformance between contracts and choreographies. In *15th IEEE Conference on Business Informatics (CBI). 2013, Vienna, Austria: IEEE Computer Society*. IEEE.

Molina-Jimenez, C., Shrivastava, S., and Cook, N. (2007). Implementing business conversations with consistency guarantees using message-oriented middleware. In *IEEE 11th Intl Enterprise Computing Conf. (EDOC 07)*, pages 51–62.

Molina-Jimenez, C., Shrivastava, S., Solaiman, E., and Warne, J. (2003). Contract representation for runtime monitoring and enforcement. In *2003 IEEE International Conference on E-Commerce (CEC 2003)*. IEEE.

Molina-Jimenez, C., Shrivastava, S., and Strano, M. (2009). Exception handling in electronic contracting. In *IEEE Conference on Commerce and Enterprise Computing (CEC). 2009, Vienna, Austria*. IEEE.

Molina-Jimenez, C., Shrivastava, S., and Strano, M. (2012). A model for checking contractual compliance of business interactions. *IEEE TRANSACTIONS ON SERVICES COMPUTING*, 5(2):276–289.

Molina-Jimenez, C., Shrivastava, S., and Wheater, S. (2011). An architecture for negotiation and enforcement of resource usage policies. In *IEEE International Conference on Service Oriented Computing & Applications (SOCA)*. IEEE.

OASIS (2006). *ebXML Business Process Specification Schema Technical Specification v2.0.4*. Available: http://docs.oasis-open.org/ebxml-bp/2.0.4/OS/spec/ebxmlbp-v2.0.4-Spec-os-en.pdf.

OMG (2011). *Documents associated with business process model and notation (bpmn) version 2.0*. http://www.omg.org/spec/BPMN/2.0/.

Pezze, M. and Young, M. (2008). *Software Testing and Analysis: Process, Principles and Techniques.* Wiley.

RedHat (2013). *"Drools"*. http://www.drools.org/.

RedHat (2014). *JBoss Enterprise Application Platform v 6.3*. http://www.redhat.com/en/technologies/jboss-middleware/application-platform.

Solaiman, E., Molina-Jimenez, C., and Shrivastava, S. (2003). Model checking correctness properties of electronic contracts. In *International Conference on Service Oriented Computing (ICSOC03)*. Springer.

Strano, M., Molina-Jimenez, C., and Shrivastava, S. (2008). A rule-based notation to specify executable electronic contracts. In *Rule Representation, Interchange and Reasoning on the Web: International Symposium (RuleML)*. Springer-Verlag.

Strano, M., Molina-Jimenez, C., and Shrivastava, S. (2009). Implementing a rule-based contract compliance checker. In *Software Services for e-Business and e-Society: 9th IFIP WG 6.1 Conference on e-Business, e-Services and e-Society (I3E)*. Springer.

Torsel, A.-M. (2013). A testing tool for web applications using a domain-specific modelling language and the nusmv model checker. In *IEEE Sixth International Conference on Software Testing, Verification and Validation*.

Ungureanu, V. and Minsky, N. H. (2000). Establishing business rules for interenterprise electronic commerce. In *14th International Symposium on Distributed Computing (DISC00)*, pages 179–193.

Utting, M. and Legeard, B. (2006). *Practical Model-Based Testing: A Tools Approach*. MorganKaufmann.

Van der Aalst, W. and Pesic, M. (2006). Decserflow: Towards a truly declarative service flow language. In *Bravetti M, Nunez M, Zavattaro G (eds) International Conference on Web Services and Formal Methods (WS-FM 2006)*, volume 4184, pages 1–23. Lecture Notes in Computer Science Springer-Verlag.