High Level Model Checker Based Testing of Electronic Contracts

Ellis Solaiman[1](✉), Ioannis Sfyrakis[1], and Carlos Molina-Jimenez[2]

[1] School of Computing Science, Newcastle University, Newcastle upon Tyne, UK
{ellis.solaiman,i.sfyrakis}@ncl.ac.uk
[2] Computer Laboratory, University of Cambridge, Cambridge, UK
carlos.molina@cl.cam.ac.uk

Abstract. Within cloud and Internet-based collaborative settings, a business contract (service agreement) is a specification that describes permissible interactions between partners. Specifically, a business contract stipulates what operations the business partners have the rights, obligations or prohibitions to execute; it also specifies when the operations are to be executed and in which order. The main purpose of an electronic contract is to regulate (monitor and/or enforce) electronic service exchanges between the contracted parties, making sure that participants adhere to the service agreement in place. Because of the dynamic nature of 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 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 coupled with *EPROMELA*, a high-level language developed specifically for modeling electronic contracts, and the *LTL Manager*; a graphical tool developed in order to aid with the specification of correctness properties in *Linear Temporal Logic (LTL)*. We describe how the *LTL Manager* can used to create a repository of common contract related LTL templates, which then can be easily selected and parameterized by the contract designer. We also 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*.

Keywords: Service agreement · Electronic contract · Service monitoring · Model checking · Automated testing · Service oriented computing · Cloud computing

1 Introduction

The context of this paper is Internet and cloud-based interactions conducted between two or more business partners. Such relationships are normally preceded

© Springer International Publishing Switzerland 2016
M. Helfert et al. (Eds.): CLOSER 2015, CCIS 581, pp. 193–215, 2016.
DOI: 10.1007/978-3-319-29582-4_11

by the negotiation and signing of business contracts also known as legal service agreements (SA). 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 [1].

An electronic contract is an executable version of the service agreement, and its main purpose is to regulate (monitor and/or enforce) electronic service exchanges between the contracted parties, checking 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".

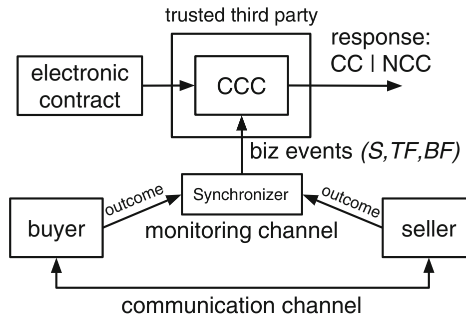

Fig. 1. The CCC deployed as a contract monitor.

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 [2]. Therefore, it is important that the process of converting a legal document into an electronic contract that is correct is automated. Previous work towards this goal has been extensive and has covered problems such as electronic contract representation and modeling [3], and contract verification [4,5]. 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 testing 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 high-level 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) [6,7], which together with the deployed electronic contract is our System Under Test (SUT). The CCC is an independent contract monitoring service that when provided with an executable specification of a contract,

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 [8] describes the basic idea: construct a behavioral model of the SUT and validate the behavior 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 [9], 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 and in any consistently standard fashion, is almost impossible. This in turn makes the process of generating accurate execution sequences required for testing the CCC difficult. Another difficulty is that specifying the contract correctness requirements is not easy. The contract designer needs to master both Promela, the input language of SPIN, and LTL (Linear Temporal Logic), the language for expressing correctness properties [10]. It is widely acknowledged that LTL is a powerful language for expressing correctness properties. Yet it has proven to be hard to master for non–experts in temporal logic. For instance, the LTL syntax traditionally accepted by SPIN is low level and based on the basic temporal logic operators (!, [], <>, etc.), which results in LTL formula that are not easy to read or write. In addition, the semantics of LTL formula are very subtle; thus writing an LTL formula that captures the intended correctness requirement within a Promela model is particularly challenging and error prone.

To address these challenges, we explore the development of a high level modeling and deployment framework. A fundamental component of our testing framework is *EPROMELA*, a high level language developed specifically for modeling electronic contracts [5]. *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. In addition, we have developed the *LTL Manager* [11], a graphical tool and a repository that can be populated by LTL experts with LTL templates (LTL formula with abstract variables) of typical correctness properties required for electronic contracts, together with their English language descriptions. These LTL templates can then be selected and parameterized by contract designers in order to produce LTLs that are specific to their requirements. The LTL properties are then mechanically included in the *EPROMELA* models and presented to SPIN for verification.

The overall contribution of this paper is to describe how *SPIN*, *EPROMELA*, and the *LTL Manager* can be instrumented with the aid of appropriate automation and message parsing tools, to automatically produce business events that can accurately test the executable electronic contract deployed within the CCC service.

The remainder of the paper is structured as follows: In Sect. 2 we describe key electronic contracting concepts with the aid of a simple example. Section 3 is dedicated to presenting our model checker based testing framework and its constituent tools. In Sect. 4 we present research work that is related to ours. Conclusions and future directions are discussed in Sect. 5.

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 in bold in the contract text). A business contract specifies a well defined list of business operations. A business operation is a business activity which the participants are able to perform under certain conditions. In the CCC, business operations are used to formally define the vocabulary (alphabet) of the interaction. We use $B = \{bo_1, \ldots, bo_n\}$ to represent all the valid business operations in the contract. The buyer and seller are regarded as role players interested in executing the operations is a shared fashion. The set of valid role players is represented by $RP = \{rp_1, \ldots, rp_n\}$.

The execution of each business process generates an individual outcome event which is passed to the synchronizer shown in Fig. 1 through the monitor channel. The synchronizer integrates the pair of individual outcomes from each side into

a single business event. This business event is sent to the CCC. As a monitor, the responsibility of the CCC is to determine whether a given event presented to it represents the notification of a contract compliant operation *CC*, or a none contract compliant operation *NCC*. To be able to make this determination, the CCC keeps track of the state of interaction as a *Finite State Machine (FSM)* with states being determined by enabling and disabling the current rights, obligations and prohibitions of the role players in force.

2.1 ROP Ontology

A contract distinguishes operations as *Rights*, *Obligations*, and *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.

We define an individual right r_i, obligation o_i or prohibition p_i as a set of operations where: $r_i \subseteq B$, $o_i \subseteq B$, and $p_i \subseteq B$. For a particular role player RP; $R_{rp} = \{r_1,\ldots,r_n\}$; $O_{rp} = \{o_1,\ldots,o_n\}$; and $P_{rp} = \{p_1,\ldots,p_n\}$, represent the sets of rights, obligations, and prohibitions currently assigned to the role player RP respectively. The sets of rights, obligations, and prohibitions of an RP are represented as ROP_{rp}.

2.2 Choreography of Interaction

To support our discussion, we will use a graphical representation of the contract written in *BPMN (Business Process Management Notation)* choreography language [12] (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.

2.3 Electronic Contracts

The electronic contract designer is able to use the legal contract and choreography in order to accurately identify and extract the *ROP* set attributed to the business partners, and to specify the rules which operate on the *ROP* set [13].

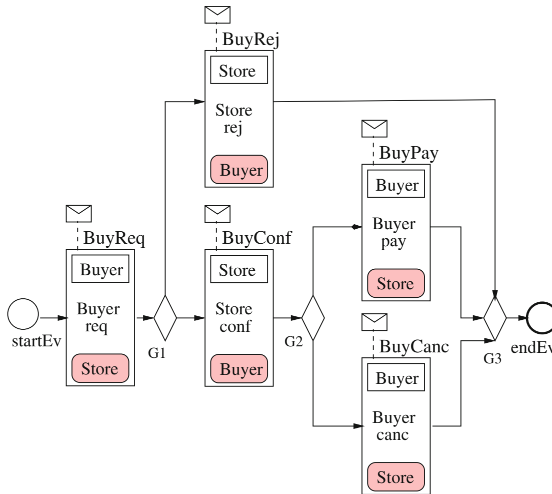

Fig. 2. Correct choreography of contract example.

Rule implementation requires an appropriate specification language; contract rules written for the CCC monitoring service are currently realized using the Drools Rule Language [14].

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. As operations are executed, and events are received by the CCC; rights, obligations, and prohibitions are granted to and revoked as specified by the rules. Therefore within the CCC, a right, obligation or prohibition can be in one of two states only: *inactive* or *active*.

Drools as a language for specifying electronic contracts is verbose, and not as declarative and readable as would be ideal. A much more suitable tool is *EROP* a language that we developed precisely for the specification of electronic contracts. *EROP (for Events, Rights, Obligations, and Prohibitions)* was first introduced in [6], and we have just completed a tool for automatically translating EROP to Drools. The *EROP to Drools Translator* has been developed using Java, and *ANTLR* [15]. The translator takes as input an *EROP* file and outputs a Drools file containing the contract rules. An example of an *EROP* to Drools conversion is shown in Fig. 3. A detailed description of the EROP language can be found in [6].

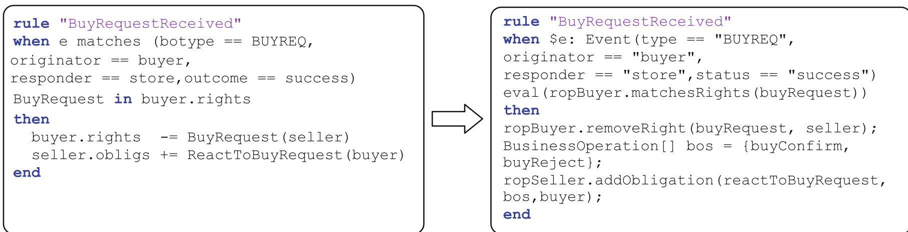

Fig. 3. EROP to Drools conversion.

2.4 Contract Compliance Within the CCC Monitor

The overall architecture of the CCC is described in detail in [11]. The CCC processes each event to determine if it is contract compliant *(CC)* or none contract compliant *(NCC)*. A business event is received by the CCC as an XML document that includes the names of the participants, the business operation, and its outcome from the set: *(Success, BizFail, TecFail)*:

```
<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 [16]. The CCC inserts the response into an outcome queue, which can be accessed by the contracted parties. The response is of the format:

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

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) $bo_i \in BO$; the business operation matches an operation within the set of business operations expected by the CCC,
(2) $bo_i \vdash ROP_{rp}$; the business operation 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). By "match", we mean that for a valid business operation bo_i, and a particular role player's *ROP* set; ROP_{rp} where: $R_{rp} = \{r_1, \ldots, r_m\}$, $O_{rp} = \{o_1, \ldots, o_m\}$, $P_{rp} = \{p_1, \ldots, p_m\}$, and $m \geq 1$, their relationship should be that: $bo_i \in r_j$ or $bo_i \in o_j$ or $bo_i \in p_j$, where $1 \leq j \leq m$.
(3) the business operation must also satisfy 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.

2.5 Exception Handling

The 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 [17]. Therefore, following the ebXML standard [18], 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.

3 Model Checker Based Testing Framework

To be able to claim that an electronic contract within the CCC is correct and conflict free, 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.

Figure 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. The framework has been updated with 2 new tools since our work in [19] with the addition of the *LTL Manager*, and the *EROP to Drools Translator*.

Electronic contract models are constructed using EPROMELA, a modeling language we developed specifically for modeling electronic contracts [5]. EPROMELA is essentially a high-level tool that extends SPIN's modeling language 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), which is not an easy task. The *LTL Manager* is a tool we have developed in order to help the contract designer with expressing correctness properties using 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 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) using EPROMELA, 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.

3.1 EPROMELA Interaction Model

An abstract view of EPROMELA components is shown in Fig. 5, which essentially models the system depicted in Fig. 1. The Business Event Generator (BEG)

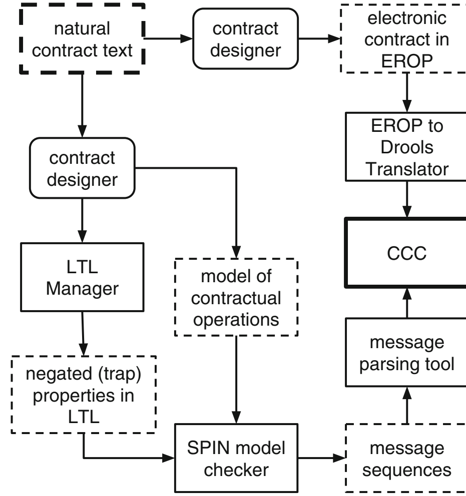

Fig. 4. Model Checker based testing framework.

generates events that are simulations of events generated by the interacting parties; for example a payment event placed by the buyer. The Contract Rules Manager (CRM) together with the ROP sets and the ECA rules (rule base) represent the CCC. The CRM is responsible for including rules as needed. The BEG and CMR communicate by two uni-directional channels (BEG2R and R2BEG). The contract rules are composed in a separate file. The ROP sets contain information about the rights, obligations, and prohibitions currently in force. For a full description, see [5]. The rule base contains a rule for each business event representing the outcome of an operation execution. So for a business operation such as "submit purchase order" there will be 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 execution behavior of the interaction model shown in Fig. 5 is as follows: (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, and executes the coded action if the associated conditions are 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.

3.2 Model Construction and Verification

Below is an example of a rule within of our EPROMELA contract model. The rule deals with the *BUYRREQ* operation of Fig. 2. Each of the operations for

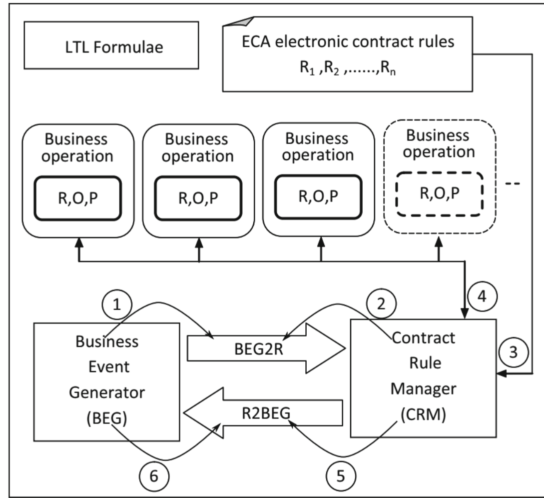

Fig. 5. EPROMELA interaction model.

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. Notice that we include within the rule, print statements that produce XML events. These are XML events that will eventually be extracted and used to automatically test the electronic contract deployed in the CCC. The end of each execution sequence is marked using a *reset* message.

```
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.

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) formula. Typical correctness properties of the electronic contracting domain are those that express mutual exclusion 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.

3.3 The LTL Manager

As discussed earlier, *Linear Temporal Logic (LTL)*, which we use for specifying contract correctness requirements, is not easy to master. In order to deal with this challenge, we have developed the *LTL manager*, a graphical interface that can be used by contract designers to include correctness properties within their EPROMELA models. The *LTL manager* offers the capability of editing LTL templates (LTL formula with abstract variables), and stores them in a database. The database is a repository of typical contract LTL formula that can be populated by LTL experts. Once the LTL repository has been populated, a contract designer can retrieve an LTL template of interest, parameterize, and include it in an EPROMELA model. The SPIN model checker is invoked from the *LTL manager* by the designer. It takes EPROMELA models augmented with LTL correctness properties and verifies whether the LTLs are satisfied or violated. Details of how to download the LTL Manager can be found in [19].

Using the tool (see Fig. 6): (a) the LTL expert specifies and adds to the template repository, common LTL templates that are of interest to contract designers. This needs to be done in natural language (*Description* box), and in LTL syntax (*Formula* box). (b) the contract designer can then load the LTLs from the database, select, and parameterize those templates of interest. As can be seen in Fig. 6, the `@V1@ @V2@ @V3@ @V4@` variables are LTL propositional symbols that can be parameterized. The tool offers a drop–down list that has all six operations (*BuyReq,*

Fig. 6. Using the LTL Manager to (a) create LTL templates and (b) parametrize them.

BuyRej, BuyConf, BuyPay, BuyCan) included in the choreography of Fig. 2. The designer selects the desired parameters as shown in Fig. 6, and the *LTL Manager* automatically creates the correct LTLs. After the LTL pattern has been parameterized in the previous step, the designer can now simply validate the model by pressing the *Add* button, and then the *Validate button* on the next screen (not shown here). The results of the validation are then displayed to the designer. In this case, both LTLs are satisfied by the validation model; consequently, SPIN displays *errors: 0*. If on the other hand, the designer adds an LTL property that cannot be satisfied by the model; for example (<> `BuyPay`) (all execution paths must eventually result in BuyPay to be executed), SPIN signals that the formula is violated, and displays *errors: 1*. In addition, SPIN creates a trail file in the working folder that can be used by the designer to trace the source of the error within the model.

3.4 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 instruct SPIN to find undesirable examples of

sequences that violate a desirable property. But we also need 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 formula required for this task is: `!<>IS_X(BUYREJ,STORE)` where `< >` is the LTL `eventually` operator. The formula 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 Nth 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 mechanize 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 to extract the XML tagged messages.

3.5 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 contain three messages, namely $BUYREQ \rightarrow BUYREJ \rightarrow reset$.

3.6 Testing the Electronic Contract

After loading and initializing 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. 1). 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 Sect. 3.4. 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.

3.7 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 (CC) operations only. An EPROMELA model can be tuned to generate sequences which include unknown and none contract compliant (NCC) business events using the EPROMELA *Event Generator* module mentioned under Sect. 3.1. Thus, we can alter the EPROMELA model to follow any variation of the choreography shown in Fig. 2. For example, the modified choreography of Fig. 7 does not correctly reflect the original text contract.

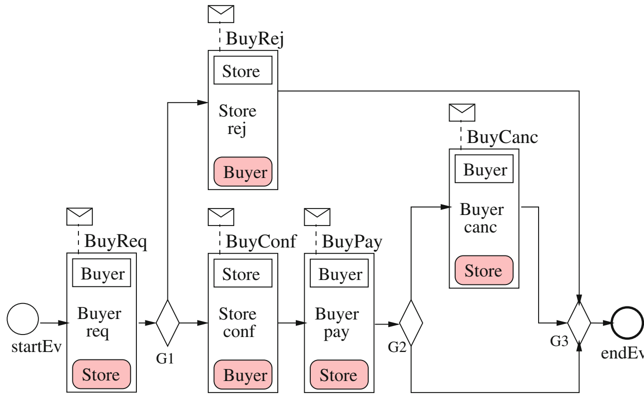

Fig. 7. Incorrect choreography of contract example.

The particularity of this diagram is that it produces CC sequences such as $BuyReq \rightarrow BuyRej$. In addition, it produces NCC sequences, for instance it allows for cancellation after payment which is not stipulated in the original contract. Consequently, the execution of *BuyCanc* within the sequence $BuyReq \rightarrow BuyConf \rightarrow BuyPay \rightarrow BuyCanc$ should be declared NCC by the CCC. The following text shows the results of the execution of the NCC sequence discussed above. The first 2 events *BUYREQ*, *BUYCONF*, were declared CC 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-
```

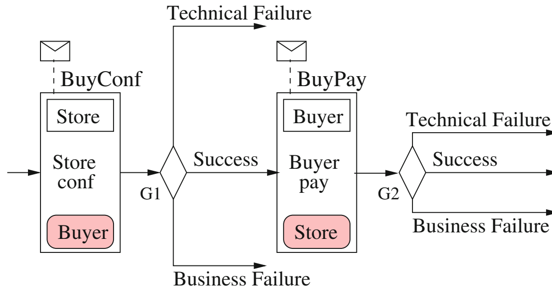

Fig. 8. Execution model with success and failures.

The process *BUYPAY* is CC (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* NCC. This is the desired behavior from the CCC, as it has detected that this sequence of events is not consistent with the contract.

3.8 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. 8, which account for successful and failed outcomes. As discussed in Sect. 2, and following the ebXML standard [18], 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.*

Below we show how an exception such as a business failure of the *BUYREQ* operation can be intuitively and naturally modeled using EPROMELA. The rule for *BUYREQ* described in Sect. 3.2 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 [17].

4 Related Work

Research work on the monitoring of cross-organizational interactions between parties was pioneered by Minsky [20] with work on Law Governed Interaction (LGI). The notion of rights, obligations, and prohibitions was introduced in [21]. A useful summary of various issues involved in contract management is provided in [22].

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 [23], where extensions to LTL are suggested.

Naturally, 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. Also, it is difficult to ensure complete coverage of all possible system behaviors during testing with manually specified LTL properties. Therefore, it is desirable to be able to systematically create complete test suites according to some test objective [24]. Research work in [25] proposes 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, as can be done using our *LTL Manager*.

The advantages and disadvantages of model checker based testing are discussed in [26] where the author provides a practical guide. Although model checker based testing techniques have been studied widely in the software engineering community [27–29], 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 [8], 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 in any standard manner is almost 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 can leverage the capabilities of a domain specific modeling language such as *EPROMELA*, which was developed specifically for modeling electronic contracts.

5 Conclusion and Future Work

Cloud and Internet based interactions between business partners can 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 a service such as the *Contract Compliance Checker (CCC)* is difficult and cannot be achieved manually. 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 formula 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.

The *LTL Manager* presented in Sect. 3.3, enables the creation and description of common contract related correctness requirements as LTL templates, which are stored in an *LTL repository*. The choreography designer can use the *LTL manager* to augment an EPROMELA model with LTL correctness properties that result from the parameterization of the LTL templates. The EPROMELA

model can then be presented to the SPIN model checker for verification and for generating test sequences.

There are a number of future research directions which we are currently exploring. 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. Also an important item for future work is to conduct experiments to determine how the presented testing framework performs as the number of possible events increases.

An issue that requires further exploration, is the development of mechanisms to aid with establishing conformance between electronic contracts and business choreographies [13]. Additional research work is required to extend such mechanisms to business functions involving more than two parties [30].

In addition to the *EROP* to Drools translator presented in Sect. 2.3, we would also 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

1. Molina-Jimenez, C., Shrivastava, S., Solaiman, E., Warne, J.: Contract representation for run-time monitoring and enforcement. In: 2003 IEEE International Conference on E-Commerce (CEC 2003). IEEE (2003)
2. Molina-Jimenez, C., Shrivastava, S., Wheater, S.: An architecture for negotiation and enforcement of resource usage policies. In: IEEE International Conference on Service Oriented Computing and Applications (SOCA). IEEE (2011)
3. Strano, M., Molina-Jimenez, C., Shrivastava, S.: A rule-based notation to specify executable electronic contracts. In: Bassiliades, N., Governatori, G., Paschke, A. (eds.) RuleML 2008. LNCS, vol. 5321, pp. 81–88. Springer, Heidelberg (2008)
4. Solaiman, E., Molina-Jiménez, C., Shrivastav, S.: Model checking correctness properties of electronic contracts. In: Orlowska, M.E., Weerawarana, S., Papazoglou, M.P., Yang, J. (eds.) ICSOC 2003. LNCS, vol. 2910, pp. 303–318. Springer, Heidelberg (2003)
5. Abdelsadiq, A., Molina-Jimenez, C., Shrivastava, S.: A high level model checking tool for verifying service agreements. In: The 6th IEEE International Symposium on Service-Oriented System Engineering (SOSE 2011). IEEE (2011)
6. Strano, M., Molina-Jimenez, C., Shrivastava, S.: Implementing a rule-based contract compliance checker. In: Godart, C., Gronau, N., Sharma, S., Canals, G. (eds.) I3E 2009. IFIP AICT, vol. 305, pp. 96–111. Springer, Heidelberg (2009)
7. Molina-Jimenez, C., Shrivastava, S., Strano, M.: A model for checking contractual compliance of business interactions. IEEE Trans. Serv. Comput. **5**(2), 276–289 (2012)
8. Abdelsadiq, A., Molina-Jimenez, C., Shrivastava, S.: On model checker based testing of electronic contracting systems. In: IEEE International Conference on Commerce and Enterprise Computing (CEC 2010). IEEE (2010)

9. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison Wesley Professional, Boston (2003)
10. Pnueli, A.: The temporal logic of programs. In: Proceedings of 18th Annual Symposium on Foundations of Computer Science (FOCS 1977), pp. 46–57 (1977)
11. Solaiman, E., Sun, W., Molina-Jimenez, C.: A tool for the automatic verification of bpmn choreographies. In: IEEE 12th International Conference on Services Computing (SCC). IEEE (2015)
12. OMG: Documents associated with business process model and notation (bpmn) version 2.0 (2011). http://www.omg.org/spec/BPMN/2.0/
13. Molina-Jimenez, C., Shrivastava, S.: Establishing conformance between contracts and choreographies. In: 15th IEEE Conference on Business Informatics (CBI), IEEE Computer Society, Vienna, Austria. IEEE (2013)
14. RedHat: Drools (2013). http://www.drools.org/
15. Parr, T.: The Definitive ANTLR 4 Reference, January 2013
16. Molina-Jimenez, C., Shrivastava, S., Cook, N.: Implementing business conversations with consistency guarantees using message-oriented middleware. In: IEEE 11th International Enterprise Computing Conference (EDOC 2007), pp. 51–62 (2007)
17. Molina-Jimenez, C., Shrivastava, S., Strano, M.: Exception handling in electronic contracting. In: IEEE Conference on Commerce and Enterprise Computing (CEC). IEEE, Vienna, Austria (2009)
18. OASIS: ebXML Business Process Specification Schema Technical Specification v2.0.4. http://docs.oasis-open.org/ebxml-bp/2.0.4/OS/spec/ebxmlbp-v2.0.4-Spec-os-en.pdf (2006)
19. Solaiman, E., Sfyrakis, I., Molina-Jimenez, C.: Dynamic testing and deployment of a contract monitoring service. In: 5th International Conference on Cloud Computing and Services Science. SCITEPRESS (2015)
20. Ungureanu, V., Minsky, N.H.: Establishing business rules for inter-enterprise electronic commerce. In: Herlihy, M.P. (ed.) DISC 2000. LNCS, vol. 1914, pp. 179–193. Springer, Heidelberg (2000)
21. Ludwig, H., Stolze, M.: Simple obligation and right model (SORM) - for the runtime management of electronic service contracts. In: Bussler, C.J., Fensel, D., Orlowska, M.E., Yang, J. (eds.) WES 2003. LNCS, vol. 3095, pp. 62–76. Springer, Heidelberg (2004)
22. Hvitved, T.: A survey of formal languages for contracts. In: Fourth Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS 2010) (2010)
23. Galton, A.: Temporal Logics and Computer Science: an Overview. Academic Press, Cambridge (1987). Chap. 1
24. Fraser, G., Wotawa, F., Ammann, P.: Testing with model checkers: a survey, pp. 215–261. Verification and Reliability, Software Testing (2009)
25. van der Aalst, W.M.P., Pesic, M.: DecSerFlow: towards a truly declarative service flow language. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 1–23. Springer, Heidelberg (2006)
26. El-Far, I.K.: Enjoying the perks of model-based testing. In: Proceedings of the Software Testing, Analysis, and Review Conference (STARWEST 2001) (2001)
27. Utting, M., Legeard, B.: Practical Model-Based Testing: a Tools Approach. Morgan-Kaufmann, Burlington (2006)
28. Pezze, M., Young, M.: Software Testing and Analysis: Process. Wiley, Principles and Techniques, New York (2008)

29. Torsel, A.M.: 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 (2013)
30. Shrivastava, S., Little, M.: Designing atomic business functions with distributed control. In: 17th IEEE Conference on Business Informatics (CBI 2015). IEEE (2015)