

Newcastle University e-prints

Date deposited: 16th April 2012

Version of file: Author final

Peer Review Status: Unknown

Citation for item:

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)*. 2010, Shanghai, China: IEEE.

Further information on publisher website:

<http://ieeexplore.ieee.org>

Publisher's copyright statement:

© IEEE, 2010

'Authors and/or their employers shall have the right to post the accepted version of IEEE-copyrighted articles on their own personal servers or the servers of their institutions or employers without permission from IEEE.'

The definitive version of this article is available at:

<http://dx.doi.org/10.1109/CEC.2010.35>

Always use the definitive version when citing.

Use Policy:

The full-text may be used and/or reproduced and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not for profit purposes provided that:

- A full bibliographic reference is made to the original source
- A link is made to the metadata record in Newcastle E-prints
- The full text is not changed in any way.

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

**Robinson Library, University of Newcastle upon Tyne, Newcastle upon Tyne.
NE1 7RU. Tel. 0191 222 6000**

On Model Checker Based Testing of Electronic Contracting Systems

Abubkr Abdelsadiq, Carlos Molina–Jimenez and Santosh Shrivastava
School of Computing Science
Newcastle University
Newcastle upon Tyne, NE1 7RU, UK
{abubkr.abdelsadiq, carlos.molina, santosh.shrivastava}@ncl.ac.uk

Abstract—The paper investigates the suitability of model checker based testing techniques for contract monitoring and enforcing services. In particular a contract monitoring service called *Contract Compliance Checker (CCC)* is considered as the system under test. The CCC is provided with an executable specification of the contract in force and is able to determine whether the actions of the business partners are consistent with respect to the contract. Contractual interactions can give rise to highly complex execution traces, and it is quite unrealistic to assume that such traces can be produced manually for testing purposes. The paper describes how a model checker can be used effectively at design time to validate the consistency of the contractual clauses and later, to produce test case validation sequences to test the correctness of the actual implementation.

Keywords—electronic contracts; automated testing; model checking;

I. INTRODUCTION

We consider electronic contracting systems intended to provide mechanisms for regulating contractual interactions. By regulation we mean *monitoring* and/or *enforcement* of business-to-business (B2B) interactions to ensure that the business partners are performing actions that are compliant with the electronic contract in force, detecting violations of contract clauses and determining liability, enforcing sanctions or given allowances and facilitating dispute resolution, by providing audit trail of business interactions [1], [2].

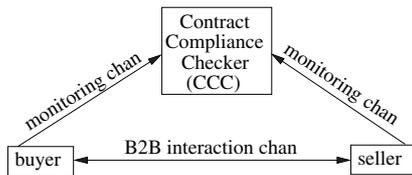


Figure 1. Contract regulated interaction.

We primarily focus on the terms and conditions of B2B legal contracts concerned with purchase orders fulfilment, supply chain management etc., rather than service level agreements (SLAs) that specify quality of service, such as bandwidth and response time (although we believe that the ideas of this paper can be extended to cover SLAs as well). Within this context, we consider an independent, third party contract monitoring service called *Contract Compliance Checker (CCC)*. The CCC (see Fig. 1 which depicts the

logical communication paths between business partners and the CCC) is provided with an executable specification of the contract in force; it is able to observe and log the relevant B2B interaction events which it processes to determine whether the actions of the business partners are consistent with respect to the contract.

Naturally, it is important to ensure that the CCC itself acts correctly. This in turn implies validating the correctness of the executable contract. Contract clauses can (and do) contain a variety of constraints to do with timing and relative ordering of operations together with exceptional clauses that specify what happens if the primary clause is violated (breached); these factors make them quite hard to validate. To this end we are interested in developing high-level testing tools for contract monitoring and enforcing services, such as the CCC. Our investigations, reported here, describe how a model checker can be used effectively for automated generation of test cases. Although model checker based testing techniques have received wide attention in the software engineering community [3]–[5], their use in testing of contract monitoring and enforcing services has received little attention.

The basic idea behind model checker based testing is simple and elegant: construct a behaviour model of the system under test (SUT) and validate the behaviour using a model checker (e.g., use Promela language for constructing the model and verify using SPIN, [6]). Such a validated model can then be used for generating executable test cases for the SUT; the model also acts as an oracle, since it also indicates the expected outputs the SUT should produce under given conditions. A principal challenge here is the construction of a model that is sufficiently small (abstract, simple) to enable, as far as possible, exhaustive checking (full validation) by the model checker; at the same time, the model should be realistic enough to be able to generate test cases that exercise the SUT.

The SUT in our case is the rule based CCC service that we have implemented that relies on the Drools rule engine [7] for rule management. For the CCC, we have also implemented a contract specification language called *EROP* (for Events, Rights, Obligations and Prohibitions), that provides constructs to specify what rights, obligation and prohibitions become active and inactive after the occurrence of events

related to the execution of business operations [8], [9].

We describe how the CCC can be modelled as a reactive system, converted into Promela and validated with Spin to observe properties of interest which are regarded as safety (something bad will never happen) and liveness (something good will eventually happen) properties. We are able to check properties specific to a given contract, e.g., deadline extensions are granted exactly as stated in the clauses and so forth. We are able to build the CCC model at a sufficiently high level of abstraction to keep the state explosion problem under control, and yet produce test sequences that are realistic enough for the SUT. We describe how the implemented version of the CCC was instrumented to accept test sequences and produce outputs. Overall, we have found model checker based testing techniques to be a very effective way of automating the testing of contracting systems.

The paper is structured as follows: section two contains the background material, including a sample contract that will be used as the running example. This is followed by section three that describes the architecture of the CCC; model checking and test case generation is described in section four; section five describes how the implemented version of the CCC can be tested. Related work is described in section six and concluding remarks, including directions for further work are presented in section seven.

II. BACKGROUND

Let us have a look at a hypothetical example of a contract written in legal English. Although this contract is not comprehensive (for example, invoicing is not stipulated), it does contain clauses of considerable degree of complexity to serve as a running example. Notice that exceptional clauses are executed to impose sanctions or grant allowances to the offender, depending on the problem that cause the violation of the primary clause. Our EROP language is particularly suited to the specification of such exceptional (or contingency) clauses that come in force when the delivery obligation stated in the 'primary clause' is not fulfilled (breach or violation of the contract).

As we have argued elsewhere [9], exceptional clauses in electronic contracts should be structured appropriately to take account of messaging problems. Our study of B2B messaging standards such as ebXML [10], RosettaNet [11], BizTalk [12] suggests that at the highest level of specification (e.g., legal English), such problems can be referred to as business problems (problems caused by semantic errors in business messages, preventing their processing) and technical problems (problems caused by faults in networks and hardware/software components).

1. OFFERS AND PURCHASE ORDERS

- 1.1 The seller is entitled to send an offer to the buyer.
- 1.2 The buyer has the right to use its sole discretion to ignore an offer or respond to it by submitting a corresponding purchase order.
- 1.3 Failure to respond to the offer within 10

days shall complete the contractual transaction.

2. DISCOUNTS

- 2.1 The seller agrees to grant 15% discount to purchase orders submitted within 7 days of the receipt of the offer.
- 2.2 A purchase order submitted after 7 days (but not exceeding 10 days) will be processed but granted no discount unless clauses 4.1 or 4.2 apply.
- 2.3 Purchase orders submitted after 10 days will not be processed online.

3. PAYMENT

- 3.1 The buyer is obliged to submit payment within 5 days of sending the purchase order.
- 3.2 Payments made after 5 days will incur 10% fine and, if submitted, not considered for online processing, unless clause 4.3 applies.

4. DELAYED PURCHASE ORDERS AND PAYMENTS

- 4.1 A delayed purchase order due to business reasons shall be granted only 10% discount.
- 4.2 A delayed purchase order due to technical problems shall be granted 15% discount.
- 4.3 Failure to meet a payment deadline due to business or technical reasons will grant:
 - 4.3.1 a payment deadline extension of 5 days to the buyer.
 - 4.3.2 right of purchase order cancellation to the seller.

5. CANCELLATIONS AND REFUNDS

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

6. NUMBER OF FAILURES

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

We summarise the overall development process that is described in detail in the rest of the paper. The starting point is a contract in a natural language, such as the one just considered. From this, an executable version in EROP is developed and installed in the CCC which then becomes the SUT. For the same natural language contract, a Promela model of the CCC is developed and validated with respect to contract specific correctness requirements (such as termination in acceptable final states, checking that deadline extensions have been granted exactly as stated in the clauses, refund has taken place properly and so forth). This validated model is then used for generating test cases and applying them to the SUT as discussed subsequently.

We assume that interaction between partners takes place through a well defined set of primitive *business operations* such as *purchase order submission*, *invoice notification*, and so on; each operation typically involves the transfer of one or two business documents. A business operation is implemented by a *business conversation*: a well defined message interaction protocol with stringent message timing and validity constraints (normally, a business message is accepted for processing only if it is timely and satisfies specific syntactic and semantic validity constraints). RosettaNet Partner Interface Processes and ebXML industry standards serve as good examples of such conversations. Following the ebXML specification [10], we assume that once a conversation is started, (i.e., a business operation is initiated) it always completes to produce an *execution outcome* event from the set $\{Success, BizFail, TecFail\}$ whose elements represent

respectively a successful conclusion, a business failure or a technical failure. *BizFail* and *TecFail* events model the (hopefully rare) execution outcomes when, after a successful initiation, a party is unable to reach the normal end of a conversation due to exceptional situations. *TecFail* models protocol related failures detected at the middleware level, such as a late, syntactically incorrect or a missing message. *BizFail* models semantic errors in a message detected at the business level, e.g., the goods-delivery address extracted from the business document is invalid.

We assume that the business partners have been instrumented to generate a *business event* upon the execution of a business operation. Each such event contains information that includes the termination status (*Success*, *BizFail* or *TecFail*), name of the operation, the timestamp and other attributes to classify the operation further (for example, for a payment operation, what discount is used). The monitoring channel delivers these events to the CCC exactly once in temporal order; these events are logged at the CCC.

Informally, an *execution sequence or execution trace* is a sequence of business operations executed by the business partners that drive the interaction from its initial to a final state. The CCC will have a record of the trace in its event log. The CCC can examine a trace and determine whether an event, representing a business operation is contract compliant or not.

In these sequences we use the following notation: *OFFER*—offer submission, *PO7DAY*—purchase order submitted within seven days, *PO10DAY*—purchase order submitted within ten days, *POCNL*—purchase order cancellation, *PAY0DSC*—payment with 0% discount, *PAY15DSC*—payment with 15% discount within deadline extension, *RFND*—refund; likewise, we append *S*, *BF*, *TF* or *TO* to the name of the operation, to indicate, respectively, that the execution produced success, business failure, technical failure or that the time out to complete the execution expired. Finally, we use the notation $e_i \rightarrow e_j$ to indicate that event e_i precedes event e_j . For example, $PO7DAY_S \rightarrow PAY15DSC_{TF}$ means that the successful execution of a purchase order submitted within seven days was followed by the execution of a payment entitled to 15% discount that, unfortunately, completed in a technical failure.

In the following execution trace $OFFER_S \rightarrow PO7DAY_S \rightarrow PAY15DSC_S$ where everything goes smoothly (no exceptional clauses involved) from the submission of the offer to the submission of the payment with 15% discount, all business operations are contract compliant (a valid trace), and a correctly functioning CCC should indicate that. Whereas, in the trace $OFFER_S \rightarrow PO10DAY_S \rightarrow PAY15DSC_S$ the payment operation with 15% discount is not contract compliant (only purchase order submitted within 7 days are entitled to 15% discount).

Here is a significantly more complicated trace with all contract compliant operations, that indicates

money refund: $OFFER_{BF} \rightarrow OFFER_S \rightarrow PO7DAY_{TF} \rightarrow PO7DAY_{TO} \rightarrow PO10DAY_{BF} \rightarrow PO10DAY_S \rightarrow PAY15DSC_{BF} \rightarrow PAY15DSC_{TO} \rightarrow PAY15DSC_{EX}_{BF} \rightarrow POCNL_{BF} \rightarrow POCNL_S \rightarrow PAY15DSC_{EX}_S \rightarrow RFND_{BF} \rightarrow RFND_S$.

In general, a valid contractual interaction can give rise to highly complex execution traces, and it is quite unrealistic to assume that such a traces can be produced manually for testing purposes. Clearly there is a need for high-level testing tools. Such tools are needed for instance, at design time to validate the consistency of the contractual clauses and later, to produce test case validation sequences to test the correctness of the actual implementation.

III. THE CONTRACT COMPLIANCE CHECKER

We will first develop the notion of a contract compliant operation and then describe how the CCC checks for contract compliance. Let the set $B = \{bo_1, \dots, bo_n\}$ of business operations contain all the primitive business operations stipulated in the contract. We say that a given bo_i is a *valid business operation* only if $bo_i \in B$. Informally, a right is something that a business partner is allowed to do; an obligation is something that a business partner is expected to do unless they wish to take the risk of being penalized; finally, a prohibition is something that a business partner is not expected to do unless they are prepared to be penalized. As the contractual interaction progresses and operations are executed, rights, obligations and prohibitions are granted and revoked to parties. This idea is at the heart of the functionality of the CCC. We call *ROP sets*, the sets of rights, obligations and prohibitions currently in force for the participants, from the perspective of the CCC.

A business operation performed by a partner is said to be *contract compliant* if it satisfies the following three requirements: (i) the operation is valid; (ii) it is in the ROP set of the partner (a *matched business operation*); and (iii) it satisfies the constraints stipulated in the contractual clauses.

A business operation that fails to satisfy the first requirement is called an *unknown business operation* and declared non-contract-compliant without further analysis. A business operation that satisfies the first but not the second requirement is called a *mismatched business operation* and declared non-contract-compliant without further analysis. A matched business operation that fails to satisfy the third requirement is called an *out of context business operation* and declared non-contract-compliant.

The overall architecture of the CCC is shown in Fig. 2; it is built on an Event Condition Action (ECA) mechanism. As stated earlier, business events (*bevents*) are supplied by the business partners to the CCC through the monitoring channel and carry information on the undertaken business operations.

The *ROP sets* stores the current set of rights obligations and prohibitions of the buyer and seller. The *bevent log*-

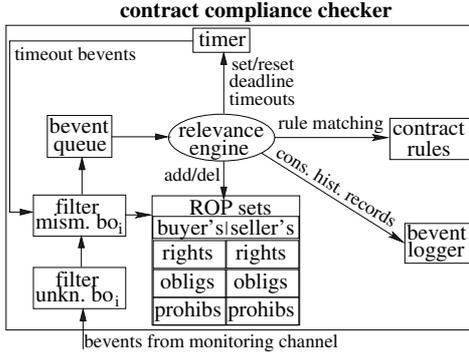


Figure 2. The architecture of the CCC.

ger is a permanent storage for keeping records about all the event processed by the CCC. The *bevent queue* is a queue that stores business events until they are removed for processing by the relevance engine. The *contract rules* is the rule base repository and contains a list of ECA rules that describe the contract in force. The events in these rules are the bevents; whereas their conditions correspond to the constraints imposed on the execution of business operations; for instance, in our running example, clause 3.1 states that payment must be performed within five days of sending the purchase order. Some conditions such as number of failures (see clause 6.1) are related to the history of the interaction, consequently, their verification involves consultation of the historical records (*cons. hist. records*) kept by the bevent logger. The actions in the rules include the operations *add* and *delete* (*add/del*) executed against the ROP sets to add and delete rights, obligations and prohibitions. The effect of an action is the update of the state of the ROP sets after the occurrence of a bevent.

The *timer* keeps track of deadlines associated to each right, obligation and prohibition stored in the ROP sets. Deadlines are set and reset *set/reset deadline timeouts* by the relevance engine. When a deadline expires, a *timeout bevent* is sent to the *filter mism. bo_i*.

Bevents that arrive from the monitoring channel pass through two filtering mechanisms before they can be stored in the bevent queue. The *filter unkn. bo_i* is responsible for filtering out bevents that correspond to unknown business operations. The *filter mism. bo_i* is responsible for filtering out bevents that correspond to mismatched business operations. As shown in the figure, timeout bevents are also examined by the *filter mism. bo_i* before they can reach the bevent queue. A timeout bevent should be filtered out if the corresponding *bo_i* is in the the bevent queue or is currently being processed by the relevance engine. The intention of the filtering mechanisms is to detect and discard bevents that correspond to non-contract compliant operations as early as possible; thus the relevance engine deals only with bevents that are likely to be contract compliant.

The *Relevance Engine* analyses queued events and triggers any relevant rules among those it holds in its contract rules base, following this algorithm: (i) Fetch the first event *e* from the bevent queue; (ii) Identify the relevant rules for *e*; and (iii) For each relevant rule *r*, execute the actions listed. The main action here is the updating (addition and deletion of rights, obligations and prohibitions) of the current state of the ROP sets.

The design and implementation of the CCC and the associated EROP language are described in [8]. That paper also illustrates how the clauses of a contract such as the one under consideration here would be coded using EROP, so we will show only a sample, clause 2.1: if a purchase order submission is successfully performed (event is *PO7DAY_S*) within seven days (event timestamp is less than seven days), then the buyer's obligation includes making a payment with 15% discount.

```
#buyer submits PO within 7d and gets
# 15% discount
when e is PO7DAY_S && PO7DAY_S in buyer.rights
    && orig==buyer && e.ts<7d
then buyer.obligs+=PAY15DSC
end
```

IV. MODEL CHECKING AND TEST CASE GENERATION

Spin is a freely available, mature and well documented model checker designed to validate correctness properties of asynchronous process systems. It validates abstract models written in Promela language against safety and liveness correctness claims that can be expressed as basic assertions and Linear Temporal Logic (LTL) formulae [6]. In response to the detection of a violation of a given property, Spin produces counterexamples that show how the property was violated. A *counterexample* is actually an execution trace from the initial state of the model down to the state where the violation of the property takes place. Spin can be instructed to include in its counterexamples various features of interest involved in the execution trace, such as the sequence of Promela statements executed as well and the messages sent and received. The facility that model checkers like Spin offer to generate counterexamples can be exploited to automatically generate test cases for the SUT. A *test case* is to be understood as a set of values needed to instantiate the input variables of the SUT for a given run. Similarly a *test suite* is a set of test cases meant to test a specific correctness requirement.

The basic idea of using model-checkers for generating test cases is explained at large in [4], [13] and other papers; briefly it involves four steps: 1) Build an abstract model of the SUT; 2) Formulate the property of interest (for example, that the SUT progresses from state S_i to state S_j when presented with input e_i) in LTL; 3) Negate the LTL, present it to the model checker and challenge it to execute the abstract model to show that the negated LTL claim can be violated;

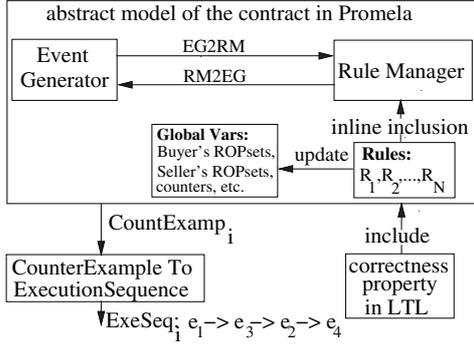


Figure 3. Promela model used for test cases generation.

as a result, the model checker produces counterexamples that include transitions from S_i to S_j when e_i is provided. As explained in [14], counterexamples produced by model checkers contain abstract parameters that are meaningful to the abstract model but meaningless to the SUT; thus they serve only as raw data to produce actual test cases that can be fed into the SUT to exercise a run.

An schematic view of the Promela model that we use for the generation of test cases is shown in Fig. 3. The top component (*abstract model of the contract in Promela*) represents the abstract model of the CCC with the contract. The goal of the model is to capture the behaviour of the CCC (precisely its state transitions) as its clauses (regarded as ECA rules) are executed. The current state is determined by the business partners' rights, obligations and prohibitions currently pending. The CCC can be regarded as a typical *reactive system* [15], [16] with the general form of behaviour: when event e occurs in state A , if condition c is true at that time, the system transfers to state B .

This model was first presented in [17] where its internal details are discussed. It was used to validate the logical consistency (for example, absence of deadlocks, rule conflicts and redundancy) of the contractual clauses expressed as safety and liveness properties. Thus regarding its logical consistency, we assume that the model is correct.

We present here only a brief summary of its functionality. The *Event Generator* (EG) models the interaction between the two business partners in the sense that it generates the business events that they produce (for example, offer successfully submitted: $OFFER_S$, purchase order submission operation performed within seven days suffered a business failure: $PO7D_{BF}$ etc.). These events are sent to the *Rule Manager* (RM) through the $EG2RM$ channel. The *Rule Manager* (RM) models the rule engine as it is responsible for processing the events and selecting potential matching rules. R_1, R_2, \dots, R_N represent the contractual clauses modeled as ECA-rules. Relevant rules are loaded by the Promela *inline inclusion* statement. The main action of a rule is an update (add/delete rights, obligations and prohibitions) of the

buyer's and seller's ROPsets; in addition, the action might also change the state of other global variables like counters which can be used as conditions in the ECA rules. The *buyer's and seller's ROPsets* are bit-vectors that represent the current right, obligations and prohibitions of the business partners; they are global variables. The status (contract compliant or non-contract compliant) of the execution of a given event is notified to the EG through the $RM2EG$ channel.

A. Generation of test cases

The components outside the model of the contract are extensions to support the generation of tests cases. As shown in the figure, to produce a counterexample, the designer needs to provide the abstract model of the contract with a correctness property expressed in LTL. The *CounterExample To ExecutionSequence* component is a filtering mechanism used for filtering out irrelevant information from counterexamples. The current implementation is a Java programme rich in regular expressions.

The length of the sequences and the status of their events (unknown, matched, mismatched, out of context and contract compliant) depend on two factors: i) the constraints imposed by the contractual clauses that might dictate, for example, to terminate the contract if more than N unknown or mismatched events are generated by the business partners; ii) the configuration of the event generator which can be tuned to generate sequences that fall into one of four possible categories:

- 1) Sequences that can include events of all possible status: unknown, matched and mismatched, non-contract compliant, contract-compliant. This is the most general case. If the contract dictates that the contract should be terminated upon the detection of the third unknown event, these sequences will include at most three unknown events.
- 2) Sequences with events corresponding to execution of matched and mismatched business operations. These sequences model the situation when the business partners intentionally or accidentally ignore the current status of the interaction and select to execute any business operation—not necessarily in the ROP sets—from within the set B . The event generator can be easily tuned not to generate unknown business events.
- 3) Sequences with events corresponding to execution of matched business operations only. This alternative models the situation when the business partners are aware of the current content of the ROP sets and execute only operations that correspond to pending rights, obligations and prohibitions, but that do not necessarily satisfy the conditions on the ECA-rules. To generate these sequences the designer has to tune the event generator to consult the ROP sets before generating its next event. This explains why the ROPsets

are modelled by global variables in the abstract model.

- 4) Sequences with events corresponding to the execution of contract compliant operations only. This is the most restricted alternative and models the situation where the business partners have accurate information about their rights and obligations and any constraints on the business operations (in the model this means knowledge of current state of the ROPsets and other global variables used in the conditions of the ECA rules). To generate these sequences the event generator has to be tuned to consult the current state of the ROPsets and the global variables, before it generates the next event.

It is worth clarifying that different sequences of events are used to test different properties of the SUT. We would be most interested in testing the behaviour of the CCC when presented with sequences of type 3 and 4; the justification is that we assume that unknown and mismatched events are removed by the filtering mechanisms (see Fig. 2) before they reach the rule engine (and it is relatively easy to test the adequacy of these filtering mechanisms). For type 4 sequence, this means the generated test cases may include events corresponding to the execution of contract compliant business operations (where the status of each business operation could be: successful, business failure, technical failure) and timeout events, corresponding to non-execution of business operations. Note that, as long as they are within the constraints expressed in the contract, the events with technical or business failures or timeouts are treated as contract compliant.

To illustrate, we consider the generation of type 4 sequences, where we use the following interpretation of clause 6.1: for online processing to continue, each business operation can suffer at most one failure (business failure or technical failure). Let us say that the designer is interested in testing that the CCC behaves correctly when clause 2.1 is executed; precisely, he would like to test that the buyer gets 15% discount when he successfully submits his purchase order within seven days. Such a requirement can be tested by a test suite composed of execution sequences that include the two operations under examination, namely, purchase order submitted within seven days (PO7DAY) and payment with 15% discount (PAY15DSC). We generate such a test suite with the assistance of the abstract model of Fig. 3. Firstly, we regard the correctness requirement as a liveness property that stays that when a PO7DAY is successfully executed, successful operation PAY15DSC will eventually follow. Secondly, we express the liveness property in LTL as $\Box PO7DAY_S \rightarrow \langle \rangle PAY15DSC_S$, where \Box , \rightarrow and $\langle \rangle$ are the conventional LTL operators, *always*, *implication* and *eventually*, respectively. Thirdly, we present the model with the negation of the LTL and instruct it to run.

The result is a set of 60 counterexamples including redundant ones. Notice that there are techniques to reduce

both the number of counter examples and their length [18]. In this order, the sequence $OFFER_S \rightarrow PO7DAY_S \rightarrow PAY15DSC_S$ represents the shortest execution from the test suit, whereas the sequence

$OFFER_{BF} \rightarrow OFFER_S \rightarrow PO7DAY_{TF} \rightarrow PO7DAY_S \rightarrow PAY15DSC_{BF} \rightarrow PAY15DSC_S$, is one of the longest.

Similarly, negation of the following LTL formula $\Box PO10DAY_S \rightarrow \langle \rangle PAY10DSC_S$ can be presented to the model to generate test cases for clause 4.1 (namely, a delayed purchase order due to business reasons shall be granted only 10% discount). Here is one of the longest sequences generated; in all these sequences, a $PO7DAY_{TO}$ event is preceded by a $PO7DAY_{BF}$ event and succeeded by a $PAY10DSC_S$ event (this behaviour is guaranteed by the validated model). In the sequence shown, offer and discount payment operations suffer failures (first and seventh events, respectively) but eventually succeed (second and last events, respectively). $OFFER_{BF} \rightarrow OFFER_S \rightarrow PO7DAY_{BF} \rightarrow PO7DAY_{TO} \rightarrow PO10DAY_{BF} \rightarrow PO10DAY_S \rightarrow PAY10DSC_{BF} \rightarrow PAY10DSC_S$.

V. TESTING

Since we assume that the abstract model is correct, it is reasonable to expect that a correctly functioning CCC should consume (accept) every single type 4 execution sequence produced by the model. Likewise, since the events are actually state transition events, the execution of such a given sequence should drive the SUT from its initial state to one of its final states that matches the state of the model that consumes the same sequence.

We instrumented the SUT to accept the traces produced by the model. We wrote a Java application that produced concrete events from the abstract ones coming from *CounterExample To Execution Sequence* module. At this stage the generated events become ready to be imported by the SUT. We note here that timeout events need special attention. When a timeout event is encountered in the sequence, its presence is used to ensure that the corresponding deadline in the time module expires straight away. To ascertain what state the SUT is in, we also instrumented it to reveal the contents of its ROP sets. This way we are able to check whether the ROP sets of the SUT match those of the model as state transitions occur: a mismatch indicating a flaw in the SUT. We are thus able to automate the testing of the CCC.

As an example, we show the behaviour of the SUT when the following fragment is presented as input: $OFFER_{TF} \rightarrow OFFER_S \rightarrow PAY7DAY_{TO}$

The SUT goes through correct state transitions. We begin with the initial state where the seller's ROP set indicates that it has the right to make an offer, the buyer's ROP set is empty. The first attempt at offer fails; after a successful offer event is encountered, the seller's ROP set becomes empty

and the buyer is given a right to submit a purchase order within seven days. However, no such operation is performed, and the seven day timeout event occurs, and the buyer is given a right to submit the purchase order within 10 days:

```
Type: init, Status: S
Seller ROP set:{ROPEntity-BO Type:OFFER, ROP Type:Right}
Buyer ROP set :{Empty}
Type: OFFER, Status: TF
Seller ROP set:{ROPEntity-BO Type:OFFER, ROP Type:Right}
Buyer ROP set :{Empty }
Type: OFFER, Status: S
Seller ROP set:{Empty }
Buyer ROP set:{ROPEntity-BO Type:P07D, ROP Type:Right }
Type: P07D, Status: TO
Seller ROP set:{Empty }
Buyer ROP set:{ROPEntity-BO Type:P010D ROP Type:Right }
```

Suppose there is a flaw in the SUT: say the rule that deals with the timeout of purchase order within seven days fails to grant the buyer the right to submit the purchase order within 10 days, then the ROP sets will be empty and will not correspond to that of the model (the mismatch is detected).

```
Type: P07D, Status: TO
Seller ROP set: {Empty }
Buyer ROP set: {Empty }
```

VI. RELATED WORK

Since the focus of our research is on functional testing, we found standard LTL convenient for expressing correctness requirement which are normally expressed as regular properties. The limitations of standard LTL to describe correctness requirements used in structural testing, such as pre and post conditions of procedure calls are pointed out in [19]. We are also aware that not all correctness requirements can be expressed in standard LTL; for instance, to express that a certain property is true at every n^{th} state of the computation, some extensions to standard LTL are needed [20]. Similarly, [21] points out the limitation of standard LTL to express (in a single LTL formula) constraints that span more than one execution trace such as those that involve modified condition/decision coverage.

The use of model-checking tools in combination with LTL for checking whether the specification of a given business process satisfies a set of compliance rules (for example, *Bank transactions larger than 10 thousand Euros must be approved by the general manager.*) is suggested in [22]; however, the focus of this work is only on static checking, whereas in ours, the interest is on run-time testing of the actual system implementation to discover potential errors in the implementations of both, the CCC and contract clauses.

A salient feature of the CCC that we suggest is that it is state-centric where states are determined by the sets of pending (active) rights, obligation and prohibitions associated to contractual clauses. Thus it allows reasoning in terms of what rights, obligations and prohibitions should or should not be fulfilled and what intermediate states the application will go through to reach a given critical state; the

advantages of being able to reason about individual states and contractual clauses are also pointed out in [23].

We are aware of some of the disadvantages of model-based testing; for instance, the usefulness of its test cases heavily depends on the skills and creativity of the technical person (not necessarily the same individual) that builds the abstract model of the SUT and formulate the correctness requirement in LTL; a practical guidance into the use of model-based testing with its pros and cons is presented in [24]. A potential problem with finite state machine (FSM) based models is the limitations of FSMs to distinguish between disjunctive and conjunctive choices in [25].

The use of model-checking tools like Spin and LTL for checking correctness properties of asynchronous systems such as Web services and contractual interactions has also been explored in [26], [27]. In [26] they discuss how WSAT (Web Service Analysis Tool) can be used to generate a Promela specification of a web service composition whose correctness properties can be formulated in LTL and verified with Spin. A more recent paper [28], explores the use of interface grammars to perform contract compliance run-time testing of web service interfaces regulated by contractual specifications such as Amazon Web Services.

VII. CONCLUDING REMARKS

We have seen that contractual interactions can give rise to highly complex execution patterns, and it is quite unrealistic to assume that these can be produced manually for testing purposes. Testing tool support is therefore needed at design time to validate the consistency of the contractual clauses and later, to produce test case validation sequences to test the correctness of the actual implementation. We have investigated what form this support should take.

Overall, we have found model checker based testing techniques to be a very effective way of automating the testing of contracting systems. We are able to build the CCC model at a sufficiently high level of abstraction to keep the state explosion problem under control, and yet produce test sequences that are realistic enough for the SUT. We are able to generate a range of test cases that can include events of all possible status(unknown, matched and mismatched, non-contract compliant, contract-compliant) and test cases with events corresponding to the execution of contract compliant operations only. We described how the CCC can be instrumented to respond to these test cases and indicate if its behaviour deviates from that of the model. The knowledge gained from this work can be used for building suitable testing tools for integrated development environment for electronic contracting systems. This is left as an item for future work.

ACKNOWLEDGMENT

The second author was funded in part by UK Engineering and Physical Sciences Research Council (EPSRC), Platform

Grant No. EP/D037743/1 and is currently with Arjuna Technologies Ltd under Knowledge Transfer Secondments award EP/H500332/1. Massimo Strano was responsible for the implementation of the CCC and the EROP language.

REFERENCES

- [1] P. R. Krishna and K. Karlapalem, "Electronic contracts," *IEEE Internet Computing*, pp. 60–68, July–Aug 2008.
- [2] A. Arenas, M. Wilson, S. Crompton, D. Cojocarasu, T. Mahler, and L. Schubert, "Bridging the gap between legal and technical contracts," *IEEE Internet Computing*, vol. 12, no. 2, pp. 13–19, Mar/Apr 2008.
- [3] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. Morgan–Kaufmann, 2006.
- [4] G. Fraser, P. Ammann, and F. Wotawa, "Testing with model checkers: A survey," *Journal for Software Testing, Verification and Reliability*, vol. 19, no. 3, pp. 215–261, 2008.
- [5] M. Pezzé and M. Young, *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, 2008.
- [6] G. J. Holzmann, *Design and Validation of Computer Protocols*. Prentice Hall, 1991.
- [7] JBoss, "Drools," <http://www.jboss.org/drools/>.
- [8] M. Strano, C. Molina-Jimenez, and S. Shrivastava, "A rule-based notation to specify executable electronic contracts," in *Proc. Int'l Symp. RuleML 2008 (RuleML'08)*. Springer, LNCS vol. 5321, 2008, pp. 81–88.
- [9] C. Molina-Jimenez, S. Shrivastava, and M. Strano, "Exception handling in electronic contracting," in *Proc. 11th IEEE Conf. on Commerce and Enterprise Computing (CEC'09)*. Jul 20–23, Vienna, Austria: IEEE CS, 2009, pp. 65–73.
- [10] OASIS, "ebxml business process specification schema technical specification v2.0.4, OASIS standard, 21 dec." 2006. [Online]. Available: <http://docs.oasis-open.org/ebxmlbp/2.0.4/OS/spec/ebxmlbp-v2.0.4-Spec-os-en.pdf>
- [11] RosettaNet, "Rosettanet implementation framework: Core specification. version v02.00.01," Revised: 6 March 2002. [Online]. Available: <http://www.rosettanet.org/>
- [12] M. Corporation, "BizTalk Framework 2.0: Document and Message Specification," Dec 2000. [Online]. Available: www.microsoft.com/biztalk/techinfo/biztalkframework20.doc
- [13] S. Rayadurgam and M. P. Heimdahl, "Test–sequence generation from formal requirement models," in *Proc. of the 6th IEEE Int'l Symposium on High Assurance Systems Engineering (HASE01)*. IEEE CS, 2001, pp. 23–31.
- [14] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, , and T. Stauner, "One evaluation of model-based testing and its automation," in *Proc. 27th Int'l Conf. on Software Engineering (ICSE'05)*. St. Louis, MO, US: ACM, 2005, pp. 392–401.
- [15] D. Harel and A. Pnueli, "On the development of reactive systems," *Logics and Models of Concurrent Systems*, vol. NATO ASI Series, F13, 1985.
- [16] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, pp. 231–274, 1987.
- [17] C. Molina-Jimenez and S. Shrivastava, "Model checking correctness properties of a middleware service for contract compliance," in *In Proc. of the 4th Int'l Workshop on Middleware for Service Oriented Computing (MW4SOC'09)*. Nov. 30, Urbana–Champaign, USA: ACM digital library, 2009, pp. 13–18.
- [18] Z. Micskei and I. Majzik, "Model-based automatic test generation for event-driven embedded systems using model checkers," in *Proc. Int'l Conf. Dependability of Computer Systems (DepCos–RELCOMEX'06)*. IEEE CS, 2006, pp. 191–198.
- [19] R. Alur, "Trends and challenges in algorithmic software verification," in *First IFIP Int'l Conf. Verified Software: Theories, Tools, Experiments (VSTTE 2005)*. Zurich, Switzerland, Oct. 10–13: Springer, LNCS 4171, 2005.
- [20] A. Galton, *Temporal Logics and Computer Science: An Overview*. Academic Press, 1987, ch. 1, pp. 27–48.
- [21] S. Rayadurgam and M. P. Heimdahl, "Generating mc/dc adequate test sequences through model checking," in *Proc. 28th Annual NASA Goddard Software Engineering Workshop (SEW'03)*, 2003, pp. 91–96.
- [22] Y. Liu, S. Müller, and K. Xu, "A static compliance–checking framework for business process models," *IBM Systems Journal*, vol. 46, no. 2, pp. 335–361, 2007.
- [23] M. Jakob, S. Miles, M. Pěchouček, and M. Luck, "Case studies for contract-based systems," in *Proc. 7th Int'l Conf. on Autonomous Agents and Multiagent Systems (AAMAS'08)*, 2008, pp. 55–62.
- [24] I. K. El-Far, "Enjoying the perks of model-based testing," in *Proc. of the Software Testing, Analysis, and Review Conference (STARWEST 2001)*, Oct/Nov, 2001, pp. —.
- [25] A. Wombacher, P. Fankhauser, and B. Mahleko, "Matching for business processes," in *Proc. IEEE Conf. on E–Commerce (CEC'03)*. IEEE SC, 2003, pp. 7–11.
- [26] X. Fu, T. Bultan, and J. Su, "Wsat: A tool for formal analysis of web services," in *Proc. 16th Int'l Conf. on Computer Aided Verification (CAV 2004)*, L. 3114, Ed., 2004, pp. 510–514.
- [27] N. Desai, Z. Cheng, A. K. Chopra, and M. P. Singh, "Toward verification of commitment protocols and their compositions," in *AAMAS '07: Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*. New York, NY, USA: ACM, 2007, pp. 1–3.
- [28] S. Hallé, T. Bultan, G. Hughes, M. Alkhalaf, and R. Villemaire, "Runtime verification of web service interface contracts," *Computer*, vol. 43, no. 3, pp. 59–66, Mar. 2010.