Establishing Conformance Between Contracts and Choreographies

Carlos Molina–Jimenez and Santosh Shrivastava

TECHNICAL REPORT SERIES

No. CS-TR-1383 April 2013

TECHNICAL REPORT SERIES

No. CS-TR-1383 April, 2013

Establishing Conformance Between Contracts and Choreographies

C. Molina–Jimenez and S. Shrivastava

Abstract

In a business-to-business collaborative setting, a choreography and a business contract (service agreement) are two specifications that describe permissible interactions between partners from different view points, emphasising different aspects. A choreography specification is a description, from a global perspective, of all permissible message exchange sequences between the partners. A business contract on the other hand specifies what operations the business partners have the rights, obligations or prohibitions to execute; it also stipulates when the operations are to be executed and in which order. It is naturally important to make sure that message exchanges as encoded in a given choreography conform to (are in accordance with) the contract between the partners. In other words, make sure that any message interaction permitted in the choreography will not cause a breach of the contract. The paper develops the concept of conformance between a contract and a choreography assuming that they can be modelled by Finite Automata. This approach opens the way for automatically establishing conformance by using model checking techniques.

Bibliographical details

MOLINA-JIMENEZ, C., SHRIVASTAVA, S.

Establishing Conformance Between Contracts and Choreographies
[By] C. Molina-Jimenez, S. Shrivastava.

Newcastle upon Tyne: Newcastle University: Computing Science, 2013.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1383)

Added entries

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1383

Abstract

In a business-to-business collaborative setting, a choreography and a business contract (service agreement) are two specifications that describe permissible interactions between partners from different view points, emphasising different aspects. A choreography specification is a description, from a global perspective, of all permissible message exchange sequences between the partners. A business contract on the other hand specifies what operations the business partners have the rights, obligations or prohibitions to execute; it also stipulates when the operations are to be executed and in which order. It is naturally important to make sure that message exchanges as encoded in a given choreography conform to (are in accordance with) the contract between the partners. In other words, make sure that any message interaction permitted in the choreography will not cause a breach of the contract. The paper develops the concept of conformance between a contract and a choreography assuming that they can be modelled by Finite Automata. This approach opens the way for automatically establishing conformance by using model checking techniques.

About the authors

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

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

Suggested keywords

CONTRACT COMPLIANCE CHECKING
CHOREOGRAPHIES
BUSINESS PROCESSES
SERVICE AGREEMENTS

Establishing Conformance Between Contracts and Choreographies

Carlos Molina–Jimenez
School of Computing Science
Newcastle University, UK
Carlos.Molina@ncl.ac.uk

Santosh Shrivastava
School of Computing Science
Newcastle University, UK
Santosh.Shrivastava@ncl.ac.uk

***Abstract*—In a business-to-business collaborative setting, a choreography and a business contract (service agreement) are two specifications that describe permissible interactions between partners from different view points, emphasising different aspects. A choreography specification is a description, from a global perspective, of all permissible message exchange sequences between the partners. A business contract on the other hand specifies what operations the business partners have the rights, obligations or prohibitions to execute; it also stipulates when the operations are to be executed and in which order. It is naturally important to make sure that message exchanges as encoded in a given choreography conform to (are in accordance with) the contract between the partners. In other words, make sure that any message interaction permitted in the choreography will not cause a breach of the contract. The paper develops the concept of conformance between a contract and a choreography assuming that they can be modelled by Finite Automata. This approach opens the way for automatically establishing conformance by using model checking techniques.**

***Keywords*-Contract compliance checking, choreographies, business processes, service agreements.**

I. Introduction

The context of this paper is Business to Business (B2B) interactions conducted over the Internet between two or more business partners. As in any commercial undertaking, partner interactions will be underpinned by a *business contract*, that we will also refer to here as a *service agreement*. A contract/service agreement specifies, among other things, what business operations the partners are permitted, obliged and prohibited to execute. Fulfilment of some business function (e.g., order fulfilment) stated in the clauses of a contract requires partners to exercise their rights and/or obligations and this in turn requires them to send business messages to each other for the exchange of electronic business documents and to act on them. This activity can be viewed as the business partners taking part in the execution of a shared business process (also called *public* or *cross–organizational* business process), where each partner is responsible for performing their part in the process. The design and implementation of individual business process components and their coordination that make up the overall cross-organizational business process is greatly aided by the availability of a *choreography* specification that describes, from a global perspective, of all permissible message exchange sequences between the partners.

Contract and choreography specifications describe permissible interactions between partners from different view points, emphasising different aspects. This is reflected in the fact that each has its own set of notations, design, verification and validation tools. It is important to make sure that message exchanges as encoded in a given choreography conform to (are in accordance with) the contract. This will make sure that the choreography is contract–compliant (so any message interaction permitted in the choreography will not cause a breach of the contract). In addition to contract–compliance, it would be desirable to be able to establish that the choreography is not restrictive (that is, it does not exclude certain interactions that are permissible in the contract). Thus, our aim is to establish automatically whether all the behaviours permissible in a choreography are also permissible in the corresponding contract and vice versa, that all the behaviours permissible in a contract are also permissible in the corresponding choreography. Establishing such conformance is clearly important, but has received little attention in the literature so far.

One of the obstacles to overcome in conformance establishment is bridging the semantic gap that exists between the two view points. We elaborate on this observation with the help of Fig. 1 that depicts a contract monitoring system capable of observing B2B interactions and determining whether they are compliant with the contract.

The figure shows a contract monitoring service called *Contract Compliance Checker (CCC)* that is deployed by the contractual parties (a buyer and a seller in this particular case) to monitor their B2B interactions at run time. The CCC is provided with an executable contract specification (derived from the natural language description of the business contract, as depicted by the dotted arrowed line) and instrumented to observe significant messaging events, referred to here as business events (*biz events*) produced from the interaction between the two parties and analyse them. Strictly speaking a CCC consists of an executable contract plus ancillary software and data (for example event logs and authentication mechanisms). Yet for brevity we will abstract these ancillary parts away and focus on the executable contract and refer to it as the CCC.

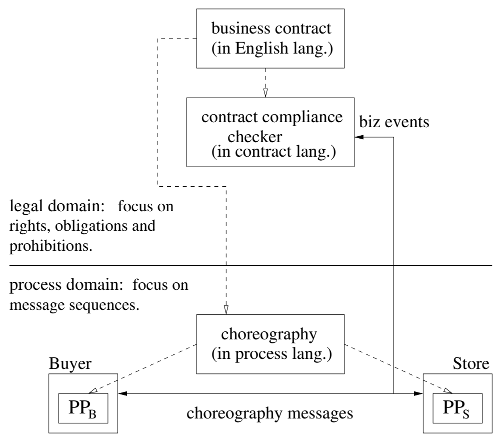

Figure 1. Business and process domains.

The business contract also forms the basis for deriving a choreography specification. This specification in turn is used for deriving public processes of each partner (PP_b and PP_s of buyer and seller respectively).

The CCC specification should enable reasoning about the observance of rights, obligations and prohibitions. Business contracts are expressed using concepts drawn from the *legal business domain*. The focus is on specifying what parties are involved in the business relationship (role players) and what business operations (actions) they are (or are not) expected to execute. These requirements are expressed as normative statements that include a list of rights, obligations, prohibitions, and contrary–to–duty–obligations, that the business partners are expected to observe. Consequently, the CCC specification notation should offer easy to use means for encoding statements like *Obligation to pay is imposed on the buyer* and that *The buyer's right to submit purchase orders is suspended until he fulfills all his pending obligations* and so forth. In this respect, event–condition–action (ECA) rule based languages have found wide acceptance.

Turning our attention to a choreography, the focus is on specifying the business interactions at message level, that is, on determining the permissible message sequences that the business partners are expected to exchange to achieve their business goals. The specification should enable reasoning about safety and liveness properties of the process it represents, such as *never deliver the goods before payment* and *for returned goods, money is eventually refunded*, respectively. In this respect, concepts and notations from the domain of *business process modelling* seem most appropriate. A good example is the Business Process Model and Notation, BPMN [1] that is widely used for specifying business processes and choreographies.

In Fig. 1, the horizontal line represents the conceptual separation between the domain of the CCC and choreography, where different formalisms are used for expressing them. This separation represents the semantic gap by which we mean that concepts that are primary to one of the domains are not necessarily primary to the other domain.

This semantic gap is reflected in the constructs of the languages used for specifying CCC and choreography. BPMN for example, does not offer constructs to explicitly express that the execution of a given task resulted in the fulfillment of a pending obligation. Conversely, this statement, can be expressed elegantly in EROP—a rule based language specifically designed for CCC [2].

This paper develops the concept of conformance between a contract and a choreography by assuming that they can be modelled by Finite Automaton (FA) that accept languages over the same alphabet. Business events (biz events, Fig. 1) form the common alphabet (common vocabulary) and enable us to bridge the gap. We show that by carefully defining the alphabet and specification approaches, we can reason about choreographies described using (a restricted but highly practical subset of) the BPMN notation and CCC described using event-condition-action rules. A noteworthy feature is that we are able to cope with failures and exceptions that any practical specification technique —for contract or for choreography— must take into account. We show that our approach can be used for automatically establishing conformance using model checking techniques (techniques that are widely used for automatic verification of reactive systems).

II. Motivating Example

For the sake of illustration, we will use parts of a simple contract between a buyer and store. Although only a hypothetical contract, it contains realistic business statements that can help elaborate our arguments.

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

Current industrial practice makes use of contracts implicitly in designing choreographies. Several researchers have suggested explicit use of contracts in deriving choreographies and/or business processes of partners. This is a topic of ongoing research (see Section VI on related work). SAVARA is a good example of an industrial framework for choreography design [3]. It provides graphical tools for building a choreography in BPMN 2.0 notation [1]. There are tools for exercising (simulating) the choreography with message sequences to check whether a given sequence is a valid execution trace of the choreography. SAVARA also has

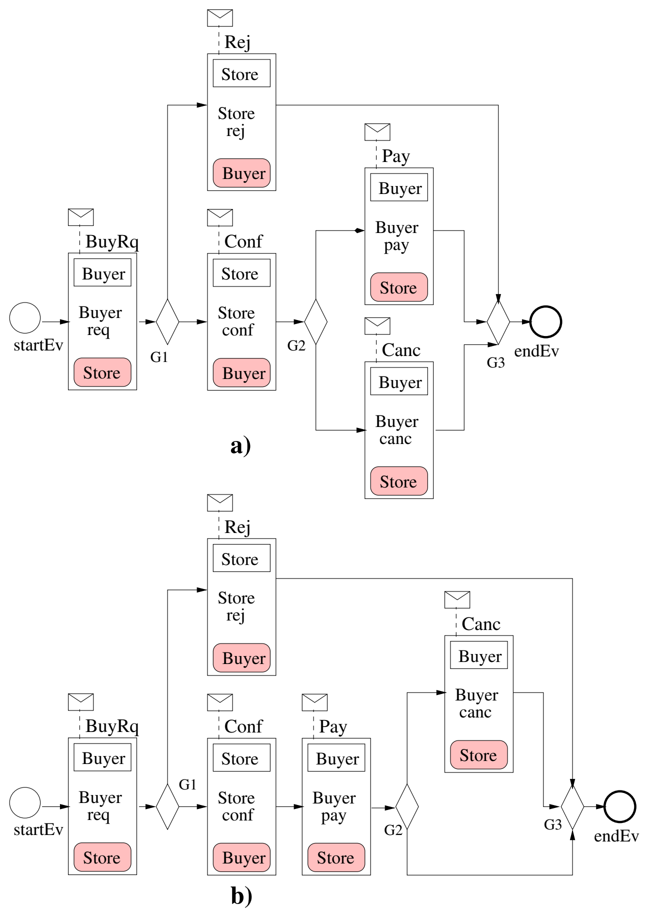

Figure 2. Choreography examples.

tools for automatically deriving partner business processes (expressed in BPEL) from a given choreography.

We show two interpretations of the contract in Fig. 2, where we use BPMN 2.0 notation. To keep matters simple, we omit details of coping with expiries of 3 and 7 days deadlines (clauses 2–a and 3–a) from these diagrams.

We will explain here only the constructs that we use in the figure. Circles are used for representing events, thus *startEv* and *endEv* represent, respectively, the start and end events of the process. The executions of activities are represented by boxes that specify the names of the activities, participants and messages. The figure includes five activities called *Buyer req, Store rej, Store conf, Buyer pay* and *Buyer canc*. They represent the activities indicated in bold in the contract. The names of the participants are specified inside bands of different colours. The sender's in a white band and the receiver in a shaded band. The figure includes five messages, namely, *BuyRq, Rej, Conf, Pay* and *Canc*. Gateways are represented by diamonds. The figure includes two exclusive fork gateways (*G1* and *G2*) and a single exclusive merge one (*G3*).

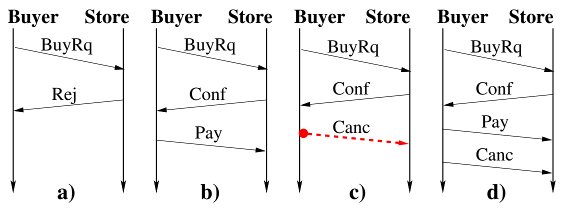

Figure 3. Testing scenarios.

We assume that choreography of Fig. 2–a is considered correct, whereas that of Fig. 2–b is incorrect. Specifically, *b)* allows cancellations (execution of *Buyer canc*) only after payments, whereas the intention is that cancellations are allowed after confirmation (alternative execution of *Buy canc* in gateway *G2* of *a)*).

As stated earlier, a framework such as SAVARA will allow checking of whether a given message sequence is a valid execution trace of a choreography. This is the principal means of testing a choreography. Fig. 3 shows four message sequences that the designers have generated manually for testing. We assume that the designers regard sequences *a)* to *c)* as representing valid executions with respect to the contract and therefore should also represent valid execution traces of a choreography. This is the case for the choreography of Fig. 2–a, but for the choreography of Fig. 2–b, sequence *c)* is invalid (message *Canc*, represented by a dashed line, is flagged as invalid). In a similar vain, sequence *d)* (which is actually invalid with respect to the contract) will be regarded as valid by the choreography of Fig. 2–b and invalid by the choreography of Fig. 2–a. This way of testing a choreography provides a very useful, but nevertheless a rather informal basis for establishing conformance. We are seeking a more rigorous approach.

III. CONTRACT AND CHOREOGRAPHY SPECIFICATIONS

In this section we elaborate how by carefully defining the alphabet and specification approaches, we can use FA to model choreographies described using (a restricted but highly practical subset of) the BPMN notation and CCC described using event-condition-action rules.

A. Common alphabet

The alphabet is the set of business events representing the outcome of executing *business operations*. A business operation represents a primitive interaction between two partners, involving exchange of one business message (containing a business document) for a specific, well defined function (e.g., *buy request*, *invoice notification*, *verify that a customer credit card is valid and can be used as a form of payment for the amount requested*, etc.). In general, an operation could involve exchange of more than one business message, but for the sake of simplicity, we restrict ourselves to just one. RosettaNet [4] is a good example of a widely used industry standard that has standardised a number of *partner interface processes* (PIPs). A PIP corresponds to a business operation, and an 'action message' of a PIP corresponds to a business message. Arbitrarily complex multi–party interactions can be built out of two partner business operations.

Taking the cue from the RosettaNet and other B2B standards, such as ebXML [5], we note that a business operation needs to be supported by a fairly sophisticated messaging protocol, as business messages usually have timing and validity constraints: a received document is accepted for processing by the receiver only if the document is received within the set time-out period (if applicable) and the document satisfies syntactic and semantic validity checks. Thus, once a business operation is initiated it always completes to produce a business event (*outcome event*) representing the outcome of the operation from the set $\{S, BF, TF\}$ whose elements represent respectively a *Successful* conclusion, a *Business Failure* or a *Technical Failure*. *BF* and *TF* events model the (hopefully rare) execution outcomes when, after initiating an operation, a party is unable to reach the normal end of the underlying protocol execution due to exceptional situations. *TF* models protocol related failures detected at the middleware level, such as a late, syntactically incorrect or a missing message. *BF* models semantic errors in a message detected at the business level, e.g., the goods-delivery address extracted from the message is invalid. In practical systems, any additional information regarding success or exceptions can be added (in the form of attributes) to these generic outcome events in an application specific manner. It is important to make sure that both the parties involved in a business operation reach the same conclusion regarding the outcome; a synchronisation mechanism is therefore required to make this happen (see for example [6], [7]).

We define the set $BO = \{bo_1, \ldots, bo_n\}$, $n \geq 1$, to contain all the business operations (*bo*). We use the following superscript notation to represent the three potential outcome events of executing a bo_i: $bo_i^s, bo_i^{bf}, bo_i^{tf}$. On this basis, we define the alphabet (also called the vocabulary) of the interaction as the set $B = \{bo_1^s, bo_1^{bf}, bo_1^{tf}, \ldots, bo_n^s, bo_n^{bf}, bo_n^{tf}\}$ that contains all the potential outcome events of all $bo_i \in BO$. We note that in certain situations, analysts might be interested in considering just successful outcomes for some business operations, in which case *B* will contain only *s* events for these operations. Similarly, in some other situations, it might be appropriate not to distinguish between *bf* and *tf* events and consider them just as business failure events, in which case *B* can be defined to contain just *s* and *bf* events for those operations.

B. Contract compliance checker

Our contact specification technique is based on the concept of contract compliance checker (CCC) explained at large in [2]. Here we present only a brief summary of basic concepts to help the reader follow our arguments.

The natural language text of a contract stipulates the rights (something that a party is allowed to do), obligations (something that a party is expected to do) and prohibitions (something that a party is not expected to do unless it is prepared to be penalised) of the parties. Contract clauses also stipulate when, in what order and by whom the operations are to be executed. If a contract is intended for electronic implementation, as is the case here, then it is important to ensure that it contains clauses that specify what to do in case messaging related failures are encountered [8]. For the sake of illustration, we show a simple modification to the contract discussed earlier, by adding clause 4 for failure handling that allows for a finite number of retries if technical or business failures are encountered (the actual number of retries will normally be a configuration parameter).

....

4) Failure handling: if even after repeated attempts, an operation does not succeed, then the contractual interaction shall be declared terminated.

Business partners exercise their rights, obligations and prohibitions by executing their corresponding business operations. The events are observed by the CCC at the granularity of outcome events, delivered to the CCC exactly once in temporal order and logged. Each event contains the termination status (*S*, *BF* or *TF*), name of the operation, the timestamp and might contain additional attributes.

As operations are executed, rights, obligations and prohibitions are granted to and revoked from business partners. In general at a given moment, each party can have several rights, several obligations and several prohibitions in force. This idea is at the heart of the functionality of the CCC that is observing outcome events of business operations. With each participant (role player), we associate a *ROP set*, the set of Rights, Obligations and Prohibitions currently in force.

For the CCC, the execution of a business operation bo_i (observed from the outcome event) is said to be *contract compliant* if it satisfies the following three requirements and is said to be *non–contract–compliant* if it does not:

- C1) $bo_i \in BO$;
- C2) it matches the ROP set of its role player (meaning, the role player has a right/obligation/prohibition to perform that operation);
- C3) it satisfies the constraints stipulated in the contractual clauses.

An example of a constraint (mentioned in C3 above) is the seven day deadline in clause 3 of the contract discussed earlier.

The significance of the ROP sets in our model is that they allow to abstract the behaviour of the CCC as that of a reactive system [9], a finite automaton, with $m+1$ states $S = \{s_0, \ldots, s_m\}$ where each state s_i represents the current state of the ROP sets. As a reactive system, the CCC remains in a given state s_i awaiting the arrival of events. When such an event represents the execution of a contract–compliant operation, the CCC executes an action and progresses to state s_j. No state changes occur or actions are executed when the event represents the execution of a non-contract–compliant operation. The main action executed consists in updating the ROP sets: rights, obligations and prohibitions from state s_i are disabled and those that determine state s_j are enabled. The salient feature of this state–centric model is that it is intellectually manageable as there are well understood formal methods and software tools (such as model checkers) that can help reason about the correctness of both the model and its implementation. Furthermore, the CCC can be directly implemented as a Event Condition Action (ECA) rule based system [2]. We refer the reader to [10] that describes an implementation of the CCC and the associated EROP rule language.

An instance of a complete contractual interaction is indicated by a non empty sequence of outcome events, that progresses the state of the ROP sets from initial to terminated final states. Such a sequence will be defined as *contract–compliant execution sequence* if all the constituent business operations are contract–compliant and the ROP sets in the final state indicate that there are no pending obligations (all the obligations have been fulfilled, so no contract violation has occurred). Astute readers will have guessed by now that our objective is to ascertain that a choreography generates only contract compliant execution sequences.

C. Choreography

For choreography specification, we have chosen the RosettaNet Methodology for Creating Choreographies [11], [12] which is a restricted version of BPMN 2.0 [1], yet it offers the right abstractions and simplifications (business operation with normal and exceptional outcomes, synchronised outcomes and so on) for modelling B2B interactions at the level of the process domain (see Fig. 1). The simplicity of this notation allows us to build choreographies that can be modelled as FA with edges labelled with symbols from the B alphabet.

The choreographies of Fig. 2 depict only the normal execution paths. We now include failures; consider the specific case of dealing with the contract with failure handling clause (section III-B). The new version of the choreography of Fig. 2–a that takes failures into account is shown in Fig. 4. Here we assume that a business operation either succeeds or generates a business failure exception and a failed operation is retried once.

In the figure, *S* and *TF* stands for Success and Technical Failure, respectively. Similarly, *rqTF*, *rjTF*, *coTF*, *paTF* and *caTF* represent counters that count the number of failed executions of the operations, *BuyRq*, *Rej*, *Conf*, *Pay* and *Canc*, respectively. In the same order, $N > 0$ is a bound on the total number of executions of an operation; for this particular example we set $N = 2$ (one retry allowed).

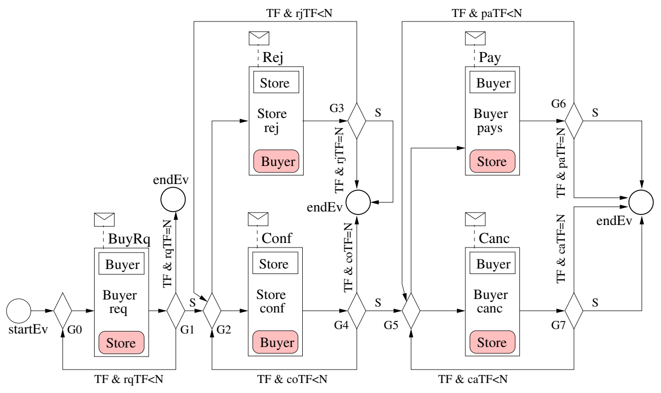

Figure 4. Choreography with failure support.

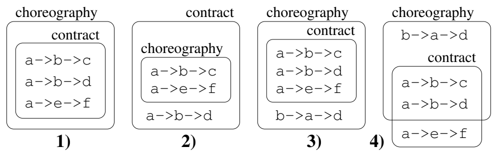

Figure 5. Discrepancies between contracts and their choreographies.

In this diagram, the execution of each activity leads to a gateway with three outgoing arrows. Let us look at the execution of activity *BuyRq* to explain the idea. A successful (*S*) execution of *BuyRq* leads to the normal execution of the contract, namely to *G2*. Alternatively, if the execution completes in *TF* and the number of failed execution ($rqTF$) of the *BuyRq* operation is less than *N*, the execution is tried again. However, if the outcome is *TF* and it has already failed $N-1$ times, the contractual interaction is terminated. Failure handling with the remaining of the activities is similar, except that the split gateways *G2* and *G5* introduce more additional alternative execution paths.

IV. Conformance

A. Informal treatment

Let us assume that $B = \{a, b, c, d, e, f\}$ is the alphabet of a given contract and choreography. We use the symbol $\rightarrow$ to denote the *happened before relation*, thus $a \rightarrow b$ denotes that a happened before b.

Let us assume that the set $contract = \{a \rightarrow b \rightarrow c, a \rightarrow b \rightarrow d, a \rightarrow e \rightarrow f\}$, contains all the sequences that are contract compliant. Consequently, the sequence $\{b \rightarrow a \rightarrow d\}$ is non–contract compliant. Finally, let us assume that the set *choreography* contains all the execution sequences that the choreography can generate. Naturally, different choreographies will generate different *choreography* sets. Four sets of choreographies are shown in Fig. 5.

- **conformance:** Case *1)* represents *conformance*, $contract = choreography$, which means that the choreography generates all the contract compliant sequences accounted by the contract and nothing else.

- **weak conformance:** Represented by case *2)*, $choreography \subset contract$. The corresponding interpretation is that the choreography fails to generate one or more of the contract compliant sequences ($a \rightarrow b \rightarrow d$). We call this situation *weak conformance* because the contract is never violated but some of the contract compliant sequences are never generated. Depending on the particular application, this situation might be acceptable.
- **non–conformance:** Represented by cases *3), 4)*. For case *3)*, $contract \subset choreography$; the choreography generates absolutely all the contract compliant sequences accounted by the contract. Regretfully, the choreography also generates one or more non–contract compliant sequences like $b \rightarrow a \rightarrow d$. The choreography of 4) suffers from a combination of errors of case *2)* and *3)*.

For completeness, it is worth mentioning that we excluded from this discussion the situation where the $contract \cap choreography = \emptyset$ on the basis that it is an unlikely situation.

B. Formal treatment

Let A_{cho} and A_{con} be Finite Automata that accept languages over the same alphabet $B = \{bo_1^s, bo_1^{bf}, bo_1^{tf}, \ldots, bo_n^s, bo_n^{bf}, bo_n^{tf}\}$. A_{cho} represents a choreography and A_{con} represents a contract; similarly, let us define $L_{cho} \subseteq B^*$ as the language accepted by A_{cho} and $L_{con} \subseteq B^*$ as the language accepted by A_{con}.

We say that a choreography *conforms* to a contract if and only if the languages accepted by their FA are equivalent, that is, $L_{con} = L_{cho}$. A choreography is *weakly conformant* to a contract if $L_{cho} \subset L_{con}$.

Determination of language equivalence is a well understood problem that can be addressed by different approaches. For instance, if the specification of the two FAs is provided we can determine equivalence by analysis of their state space. Alternatively and in the absence of their specification one can regard the FA as black boxes and determine equivalence by analysis of the sequences that they accept. We take the second alternative.

In this manner, a contract conformant choreography for our contract example (Section II) would be represented by an automaton that accepts a language defined as $L_{con} = \{BuyRq \rightarrow Rej, BuyRq \rightarrow Conf \rightarrow Pay, BuyRq \rightarrow Conf \rightarrow Canc\}$. Note that, as this example only concerns normal (successful) executions, we have dropped the s superscript. An examination of their execution sequences would reveal that this requirement holds for Fig. 2–a but not for Fig. 2–b.

As explained at large in Section V, execution sequences can be generated automatically with the assistance of software tools such as model–checkers. Let us assume the availability of such tools. The general idea behind using these tools is sketched in Fig. 6.

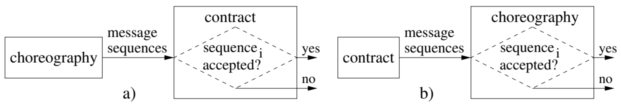

Figure 6. Contract–choreography conformance checking.

In *a)*, all the choreography sequences (one at a time) are fed to the contract which outputs *yes* if all of the sequences were accepted; it outputs *no* if at least one of the sequences was not accepted. Table I summarises what can be deduced from such an experiment, referring to the four cases of Fig. 5. A *yes* will indicate that the choreography is at least weakly conformant; a *no* will indicate non–conformance.

A complementary experiment is conducted in *b)*: all the contract sequences (one at a time) are fed to the choreography which outputs *yes* if all of the sequences were accepted and *no* if at least one of the sequences was not accepted. Table II summarises what can be deduced from such an experiment. A *yes* indicates that the choreography accepts all the contract compliant sequences, but there is no assurance that non–contract compliant sequences will be rejected by the choreography; a *no* indicates the possibility of either non–conformance or weak-conformance, but the experiment cannot ascertain beyond this.

outcome	interpretation
yes	case 1 or 2
no	case 3 or 4

Table I
SEE FIG.6 A)

outcome	interpretation
yes	case 1 or 3
no	case 2 or 4

Table II
SEE FIG.6 B)

To determine categorically if a given choreography conforms to its contract, the designer needs tools for generating all execution sequences for both choreography and contract and perform set comparison for equality.

V. A TOOL FRAMEWORK

Based on the concepts developed earlier, in this section we discuss a model checker based framework: our aim is to verify at the design stage that the behaviour of the two models conform to each other before proceeding to the implementation stage. The verification includes two stages: independent verification and combined verification. Firstly, the two models are verified independently to guarantee that they satisfy certain correctness requirements specific to their domains (e.g., for choreography, verify that it is *realizable*, see later). In the second stage, the behaviour of the previously verified models are contrasted against each other by means of comparison of their execution sequences. We use model checkers for generating counter examples from where we extract execution sequences.

Fig. 7 shows the framework that we are constructing based around the SPIN model checker [13]. Its input language for model building is PROMELA and the list of correctness properties that the model is expected to satisfy are expressed using Linear Temporal Logics (LTL). Tools are represented by solid squares with sharp corners; squares with smooth corners represent humans and dashed boxes represent data. In the figure, parts *a)* and *b)* refer to the tool sets for CCC and choreography, respectively.

At the time of writing (April 2013), we have completed building model checking tools for the CCC and BPMN choreographies [14], [15]. For the CCC, we have extended PROMELA (*EPROMELA*) with constructs for expressing the core contractual concepts as embodied in the CCC described in Section III-B to provide a high level model checking tool. Once a model is declared correct it can be used for generating sequences (*message sequences*). The sequence generation technique involves in presenting the model checker with a trap property expressed in LTL (*trap properties in LTL*, Fig. 7). As a response, the model checker produces counter examples from where one can extract the message sequences. The models can also be used for the generation of test cases for exercising the constructed system to validate that the implementation actually satisfies the correctness requirements that the models do ([16] describes how CCC is tested). Effort to integrate these tools within the SAVARA project is also under way. We will explain now the functionality of the two tool sets included in the framework.

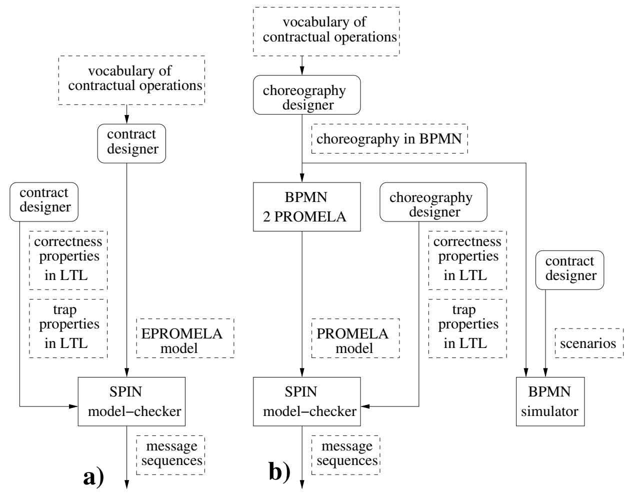

Figure 7. Tools for validating logical consistency and conformance.

A. Part a)

The tool shown in part *a*) of the figure is in effect a contract–validation tool for reasoning with the help of the SPIN model checker about contract models written in the EPROMELA language. Thanks to the additional features that we have added to EPROMELA, we can include contract related concepts in validation models. For instance, we can include `SET_O(PAY,1)`, where O stand for obligation, in validation models, to impose an obligation to pay (on a buyer for example). In the same order, to query if the obligation to pay is still pending, one can say `IS_O(PAY,BUYER)`.

The conversion of the contract in English into EPROMELA is done manually and involves the conversion of contract clauses into ECA rules written in EPROMELA language. The basic approach is quite straightforward: there is a rule for each outcome event of a business operation (or equivalently, a rule for each business operation, with logic for all the outcome events for that operation). The rule manipulates the ROP set by granting/removing rights/obligations etc., as implied by the contract clauses.

If a contract compliance system is to be deployed, then the verified EPROMELA model of the CCC can be used as the basis for producing the actual implementation for which a candidate language is the EROP contract language [2], [10]. As EROP is an ECA language designed according to the concepts as embodied in the CCC, this process is relatively straightforward. We show below some relevant parts of the EPROMELA implementation of the contract of Section II but with failure support. The complete code is shown in Appendix A. We assume the same vocabulary as the choreography of Fig. 4, so we deal with success and technical failure events for each business operation, and assume that a failed operation is retried once. Again, for the sake of simplicity, we omit dealing with the expiries of 3 and 7 days deadlines (clauses 2–a and 3–a). The code consists of two components. The first one (*BuyerStoreContract*) defines the vocabulary of business operations and implements the generation of the business events from that vocabulary whereas the second one (*Rules*) implements the rules that react to the business events.

```
/* Programme name: BuyerStoreContract */
... ... ...
#define TRUE  1
#define FALSE 0
#define AbnContractEnd (abncoend==TRUE)

/* var for occurrences of executions
 * with S and TF outcomes
 */
bool abncoend=FALSE;
bool ReqFailBefore=FALSE;
bool RejFailBefore=FALSE;
bool ConfFailBefore=FALSE;
bool PayFailBefore=FALSE;
bool CancFailBefore=FALSE;

/* declaration 2 role players involved */
RolePlayer(BUYER,STORE);
... ... ...
/* 5 operations involved in contract */
BIS_OP(BUYREQ);
BIS_OP(BUYREJ);
BIS_OP(BUYCONF);
BIS_OP(BUYPAY);
BIS_OP(BUYCANC);

/* Ex. of contract specific correctness
 * requirements.
 * p0: "if the buyer has an oblig to pay, he
```

```
30  * will eventually, pay or cancel unless
31  * the contract terminates abnormally".
32  * Meaning of LTL variables:
33  * AbnContractEnd: Abnormal contract end
34  * IS_O(BUYPAY,BUYER): is buyer's oblig to pay
35  *                     pending?
36  * IS_X(BUYPAY,BUYER): has buyer executed buypay
37  *                     operation?
38  *
39  */
40  /* ltl p0 { [](!IS_O(BUYPAY,BUYER)) ||
41   * <>(IS_O(BUYPAY,BUYER) -> <> (IS_X(BUYPAY,BUYER)
42   * || IS_X(BUYCANC,BUYER) || AbnContractEnd))  }
43   */
44
45 /* Ex of trap LTLs for generating exec sequences */
46  * p1: generates exec seq including successful
47  *     executions of BUYREJ
48  * p2: generates exec seq including  successful
49  *     executions of BUYPAY
50  * p3: generates exec seq including successful
51  *     executions of BUYCANC
52  */
53 /* ltl p1 { !<>IS_X(BUYREJ,STORE)  } */
54 /* ltl p2 { !<>IS_X(BUYPAY,BUYER)  } */
55 /* ltl p3 { !<>IS_X(BUYCANC,BUYER) } */
56
57 /* Business Event Generator */
58 proctype BEG()
59 {
60  BEGIN_INIT:
61  {
62  /* Define the initial state of the rights (R),
63   * obligations (O) and prohibitions (P) of the 2
64   * role players following:
65   * INIT(OperName, RolePlayerName, R,O,P)
66   * 1 means granted, 0 means not granted
67   * In initial state buyer has been granted the
68   * right to execute BUYREQ. No other R,O,P are
69   * granted to buyer or store */
70   INIT(BUYREQ,  BUYER, 1,0,0);
71   INIT(BUYREJ,  STORE, 0,0,0);
72   INIT(BUYCONF, STORE, 0,0,0);
73   INIT(BUYPAY,  BUYER, 0,0,0);
74   INIT(BUYCANC, BUYER, 0,0,0);
75  }
76  END_INIT:
77
78  /* generation of business events.
79   * For each of the 5 operations, 2 possible exec
80   * are modelled: exec with S and exec with TF */
81  end:do
82  :: B_E(BUYER, BUYREQ,  S);
83  :: B_E(BUYER, BUYREQ,  TF);
84  :: B_E(STORE, BUYREJ,  S);
85  :: B_E(STORE, BUYREJ,  TF);
86  :: B_E(STORE, BUYCONF, S);
87  :: B_E(STORE, BUYCONF, TF);
88  :: B_E(BUYER, BUYPAY,  S);
89  :: B_E(BUYER, BUYPAY,  TF);
90  :: B_E(BUYER, BUYCANC, S);
91  :: B_E(BUYER, BUYCANC, TF);
92  od;
93 }
94 ... ... ...
```

```
 1 /* Programme name: Rules */
 2 ... ... ...
 3 /* Rule triggered by buyreq executions
 4  * initiated by the buyer and completed
 5  * either in success or technical failure.
 6  */
 7 RULE(BUYREQ)
 8 {
 9  WHEN::EVENT(BUYREQ,IS_R(BUYREQ,BUYER),SC(BUYREQ))
10   /* handle buyreq with success outcome */
11   ->{ SET_X(BUYREQ,BUYER);/*BUYREQ successfully */
12                           /*executed           */
13       SET_R(BUYREQ,0);  /* remove right         */
14       SET_O(BUYREJ,1);  /* impose obligation    */
15       SET_O(BUYCONF,1); /* impose obligation     */
16       RD(BUYREQ,BUYER,CCR,CO); /* rule notifies  */
17      }/* S exec of contract compliant BUYREQ     */
18
19  /* handle buyreq with technical failure outcome */
20  ::EVENT(BUYREQ,IS_R(BUYREQ,BUYER),TF(BUYREQ))
21   ->{
22     if /* 1st notification of buyreq with TF      */
23     :: (ReqFailBefore==FALSE)->ReqFailBefore=TRUE;
24      RD(BUYREQ,BUYER,CCR,CO); /* rule notifies    */
25        /* TF exec of contract compliant BUYREQ    */
26
27     /* 2nd notification of buyreq with TF         */
28     :: (ReqFailBefore==TRUE) -> abncoend=TRUE;
29      SET_R(BUYREQ,0); /* remove right */
30      RD(BUYREQ,BUYER,CCR,CND);/*abnormal cont end */
31      /* rule notifies TF exec of BUYREQ and pre-  */
32      /* maturaly terminates contract after 2 TF   */
33     fi
34     }
35  END(BUYREQ);
36 }
37 ... ... ...
```

In the program component *BuyerStoreContract*, five business operations are named (lines 21–25). We follow the convention of using the same names as the action messages depicted in the choreography (see Fig. 4). Lines 82–91 define the set *B* of business events.

In the *Rules* program component, we show the rule that deals with *BUYREQ*. Thus, after receiving a notification of a *BUYREQ* with a S outcome (line 9), the rule removes the right of the buyer to execute *BUYREQ* and assigns an obligation to execute *BUYREJ* or *BUYCONF* (lines 13–15). In contrast, upon receiving a notification of a *BUYREQ* with a TF outcome (line 20), the rule verifies if the operation has failed before (line 28). If it has, it calls for an early termination of the contractual interaction (line 30); otherwise, it registers the occurrence of the technical failure (line 23) but does not alter the state of the ROP set or terminates the contract; in this manner, the operation can be tried one more time.

The EPROMELA model is presented to *SPIN* together with a list of correctness properties written in LTL. Correctness properties of interest here are those that include concepts from the business domain such as rights, obligations and prohibitions expressed in the normative statements of the contract. Typical correctness properties of this domain are those that express mutual exclusion of rights, obligations and prohibitions. For example, a requirements that the execution of a given operation (for example, payment) is never simultaneously obliged and prohibited. Thanks to the contract constructs offered by EPROMELA, this correctness property can be elegantly and intuitively expressed in LTL as follows:

```
[] not(IS_O(BUYPAY, BUYER) &&
 IS_P(BUYPAY, BUYER))
```

Where `IS_O(BUYPAY, BUYER)` returns true if, for the buyer, the payment operation is currently obliged and `IS_P(BUYPAY, BUYER)` returns true if the payment operation is currently prohibited; `[]` and `&&` are the *always* and *and* LTL operators. This correctness property is a typical example of a *contract independent* property that is expected from all contracts. The EPROMELA model therefore automatically checks for such a property, therefore the designer does not need to explicitly specify it.

Contract dependent properties must of course be specified. Again, the designer is expected to express them in LTL formulae that include constructs (for example, `IS_X(BUYPAY,BUYER)`) offered by EPROMELA. An example of such LTLs is $p0$ shown in lines (40–42) of the *BuyerStoreContract* code, that can be activated after removing the comment delimiters. This LTL states that once an obligation to pay is imposed on the buyer, he either pays or cancels unless the contract terminates abnormally (*AbnContractEnd*) after exhausting the allowed number of retries due to technical failures.

Once the model is declared correct (it satisfies all the contract dependent and independent properties), it can be used for generating sequences (*message sequences*) to verify contract to choreography conformance as suggested in Fig. 6–b. These sequences can also be used for exercising a BPMN tool, say from SAVARA, as mentioned in Section II. As discussed in [16], the sequence generation technique involves in presenting the model checker with a trap property expressed in LTL (*trap properties in LTL*, Fig. 7). As a response, the model checker produces counter examples from where one can extract the message sequences.

An example of a trap property that can be used for generating execution sequences that include the successful execution of the *BUYREJ* operation is $p1$ shown in line 53 of the *BuyerStoreContract*. The smallest execution sequence produced by $p1$ is shown next (without its XML tags): $BUYREQ^S \rightarrow BUYREJ^S$

This sequence lead to the successful (*S*) execution of a *BUYREQ* followed by a successful execution of *BUYREJ*. Similarly, $p2$ (line 54) and $p3$ (line 55) trap LTLs can be used for generating execution sequences that lead, respectively, to the successful execution of the *BUYPAY* and *BUYCANC* operations. Naturally, one can use a combination of $p1$, $p2$ and $p3$ to generate all the execution sequences that lead to the successful executions of *BUYREJ*, *BUYPAY* and *BUYCANC* in a single run of SPIN. It is worth mentioning that the use of built–in EPROMELA constructs, makes $p1$–$p3$ intuitive. Significantly more complex are execution sequences that include both, executions of operations that complete successfully (*S*) and in technical failures (*TF*). We show one of them next. Recall that the contract stipulates that a failed execution of an operation can be tried only one more time.

$BUYREQ^{TF} \rightarrow BUYREQ^S \rightarrow BUYREJ^{TF} \rightarrow BUYCONF^{TF} \rightarrow BUYCONF^S \rightarrow BUYPAY^{TF} \rightarrow BUYPAY^S$.

The execution sequence shows a contractual run where the buyer executes a buy request operation that completes in technical failure. Next the buyer executes the operation again —this time it completes successfully. The third event in the sequence represents the store's execution of a buy reject that completes in technical failure. The fourth event shows that after failing to execute the buy reject operation, the store abandons it and executes a buy confirmation operation that also completes in technical failure. Next the store tries again the execution of the buy confirmation operation —this times it completes successfully. The last two events show that the buyer executes the buy pay operation twice—the first time it completes in technical failure, but completes successfully the second time.

B. Part b)

The choreography side of the tool framework (Fig. 7–b) is similar in spirit to the contract side (Fig. 7–a). The *choreography designer* uses the *vocabulary of contractual operations* for constructing BPMN choreography following the conventions set in the RosettaNet BPMN specification [11]. In the figure, we suggest that the *BPMN2PROMELA* tool can be used by designers for converting the a BPMN choreography into an abstract model written in standard PROMELA (*PROMELA model*) and augmented with LTL formulae that express correctness properties. Standard PROMELA is a convenient abstract language here (as opposite to EPROMELA) because the core concepts of a choreography are messages and activities—concepts that can be elegantly modeled in PROMELA. Correctness properties of interest here involve messages and activities (as opposite to rights, obligations and prohibitions). For example, assume that c, p and n stand respectively, for execution of activities *Store conf*, *Buyer pay* and *Buyer canc*. Then a correctness property stating that always a confirmation message is eventually followed by either payment or cancellation can be expressed in LTL as:

$[](c \rightarrow <> (p||n))$

where $||$ is the conventional *or* LTL operator. Observe that this LTL formula expresses constrains on message sequences which are central parameters to choreographies. An important property of a choreography is that it should be *realizable*: it should be implementable by a set of distributed peers. Solutions to this problem which can be utilised within this type of framework have been suggested by other researchers (see for example [23]). Work on fully developing the choreography side of the tool framework is currently in progress.

Like in Fig. 7–a, once the *PROMELA model* is declared correct, it can be challenged with trap properties to generate message sequences to verify contract to choreography conformance as suggested in Fig. 6–a.

At the time of writing (April 2013), we have completed version 1.1 of the BPMN2PROMELA tool which can convert BPMN diagrams into PROMELA and include LTLs. The current version does not support the execution of activities that can produce more that one outcome (success, business failure or technical failure). An example of the code that it produces from the BPMN choreography of Fig. 2–a) is shown in Appendix B.

VI. Related work

A review of contract languages based on different formalisms ranging from modal logics to ECA rules is presented in [18]. In parallel, a great variety of choreography languages have also been suggested [19], [20] with focus on modelling different aspects of choreography processes. In [12] the author argues that existing choreography languages are too general and consequently not entirely satisfactory for modelling B2B integration (B2Bi). In particular, the author criticises the excess of constructs offered by BPMN and its lack of semantics for modelling B2Bi choreographies. He suggests the use of a restricted version of BPMN notation [11]. This notation accounts for features that are within the scope of our interest. For instance, it accounts for potential exceptional execution outcomes and assumes the existence of underlying synchronisation mechanisms to keep the interacting parties aligned during their interactions.

Concerns about the lack of mechanical tools and guidelines for checking compatibility between business contracts and their corresponding business processes are raised in [21]. The authors discuss a methodology for mechanically determining whether a choreography of a business process expressed in BPMN, conforms to its contract expressed in FCL (a Deontic Logic based language). Like ours, their approach is based on the comparison of execution sequences produced by the choreography and the contract. However, to produce choreography sequences, they suggest mapping the BPMN choreography onto an event pattern language; in contrast, we suggest that the choreography sequences can be produced by using a model checker like SPIN.

Conformance checking of the behaviour of processes to their specification is studied in [22]. The goal is to systematically verify whether a given service (a node) that interacts with others to compose a global process sends the expected messages as dictated by the specification (for example, a BPEL process). To solve the problem, the BPEL process is converted into Petri net model and traces of messages produced by the actual implementation of the BPEL process are collected in a log. The Petri net is presented with traces from the log (one at a time) to determine if they correspond to valid execution paths of the Petri net model. Though the techniques used in this work are similar to ours, our focus of attention is at a higher level of abstraction: choreography to contract conformance rather than conformance of local processes to their specifications.

Conformance of choreography, but with focus on implementation, is studied in [23]. In this work the implementation is produced automatically (by means of projection) from the choreography; consequently, the goal is to produce *realizable choreographies* that by definition will project conformant implementations. Like in our work, these authors use software tools (model checkers) for sequence generation and comparison. However, the focus of our work is on a higher level of abstraction, namely on conformance of the choreography to the business contract that it represents. Consequently, individual validations of the contract and choreography is not enough to declare each other's conformance; this is why we suggest cross–verification of message sequences produced by the contract and choreography.

VII. Concluding Remarks

We developed the concept of conformance assuming that contracts and choreographies can be modelled by Finite Automaton (FA) that accept languages over the same alphabet. We showed that by carefully selecting the alphabet and specification approaches, we can consider choreographies described using (a restricted but highly practical subset of) the BPMN notation and contract described using event–condition–action rules. A noteworthy feature is that we are able to cope with failures and exceptions that any practical specification technique for contract or for choreography must take into account. We described a model checker based tool framework for conformance checking that can form the basis for building contract compliance checkers as well as contract compliant business processes.

Acknowledgment

The first author was funded by EPSRC grant KTS-EP/H500332/1. Discussions with Garry Brown from Red Hat concerning the tool framework and SAVARA and constructive comments from Andreas Schoenberger have been useful.

References

[1] OMG, "Documents associated with business process model and notation (bpmn) version 2.0," http://www.omg.org/spec/BPMN/2.0, Jan 2011.

[2] C. Molina-Jimenez, S. Shrivastava, and M. Strano, "A model for checking contractual compliance of business interactions," *IEEE Trans. on Service Computing*, vol. PP, no. 99, 2011.

[3] Jboss, "Savara and testable architecture," http://www.jboss.org/savara, 2012.

[4] RosettaNet, "Rosettanet member home page," Nov 2011. [Online]. Available: http://www.rosettanet.org/

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

[6] C. Molina-Jimenez and S. Shrivastava, "Maintaining Consistency between Loosely Coupled Services in the Presence of Timing Constraints and Validation Errors," in *Proc. 4th IEEE European Conf. on Web Services (ECOWS'06)*. IEEE CS, 2006, pp. 148–160.

[7] C. Molina-Jimenez, S. Shrivastava, and N. Cook, "Implementing business conversations with consistency guarantees using message-oriented middleware," in *Proc. 11th IEEE Int'l Enterprise Computing Conf. (EDOC'07)*. IEEE CS, 2007, pp. 51–62.

[8] 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.

[9] D. Harel and A. Pnueli, "On the development of reactive systems," *Logics and Models of Concurrent Systems*, vol. NATO ASI Series, F13, 1985.

[10] M. Strano, C. Molina-Jimenez, and S. Shrivastava, "Implementing a rule–based contract compliance checker," in *Proc. 9th IFIP Conf. on e-Business, e-Services, and e-Society (I3E'2009)*. Nancy, France: Springer, 2009, pp. 96–111.

[11] RosettaNet, "Rosettanet methodology for creating choreographies," 27 July 2011 2012, version Identifier: R11.00.00A. [Online]. Available: http://www.rosettanet.org/

[12] A. Schönberger, "Visualizing b2bi choreographies," in *Proc. IEEE Int'l Conf. on Service-Oriented Computing and Applications (SOCA'11)*, 2011, pp. 1–8.

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

[14] A. Abdelsadiq, C. Molina-Jimenez, and S. Shrivastava, "A high–level model–checking tool for verifying service agreements," in *Proc. 6th IEEE Int'l Symposium on Service–Oriented System Engineering (SOSE'2011)*, 2011, pp. 297–304.

[15] C. Molina-Jimenez and W. Sun(Jim), "A tool for automatic verification of bpmn choreographies," School of Computing Science, Newcastle Univ. UK, Tech. Rep. CS-TR-, Feb. 2013.

[16] A. Abdelsadiq, C. Molina-Jimenez, and S. Shrivastava, "On model checker based testing of electronic contracting systems," in *12th IEEE Int'l Conf. on Commerce and Enterprise Computing(CEC'10)*, 2010, pp. 88–95.

[17] C. Molina-Jimenez and S. Shrivastava, "Establishing conformance between contracts and choreographies," School of Computing Science, Newcastle Univ. UK, Tech. Rep. CS-TR-, Feb. 2013.

[18] T. Hvitved, "A survey of formal languages for contracts," in *Fourth Workshop on Formal Languages and Analysis of Contract–Oriented Software (FLACOS'10)*, 2010, pp. 29–32.

[19] G. Decker, O. Kopp, and A. Barros, "An introduction to service choreographies," *Information Technology*, vol. 50, no. 2, pp. 122–127, 2008.

[20] A. Schönberger, "Do we need a refined choreography notion?" in *Proc. 3rd Central–European Workshop on Services and their Composition, (ZEUS'11)*, vol. 705. CEUR-WS.org, 2011.

[21] G. Governatori, Z. Milosevic, and S. Sadiq, "Compliance checking between business processes and business contracts," in *Proc. 10th IEEE Int'l Enterprise Distributed Object Computing Conf. (EDOC'06)*. IEEE computer society, 2006, pp. 221–232.

[22] W. M. P. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, and E. Verbeek, "Conformance checking of service behavior," *Transactions on Internet Technology*, vol. 8, no. 3, pp. 13:1–13:30, May 2008.

[23] G. Salaün, T. Bultan, and N. Roohi, "Realizability of choreographies using process algebra encodings," *IEEE Transactions on Services Computing*, vol. preprint, no. 10.1109 TSC.2011.9, 2011.

Appendix A. EPROMELA Code of Buyer–Seller Contract

```
/*
 * Carlos Molina-Jimenez, 18 Apr 2013, Ncl Uni, UK
 *                       Carlos.Molina@ncl.ac.uk
 * BuyerStoreContract.pml: EPROMELA code of a
 * contract between a buyer and store. This model
 * is meant to correctly implement the English
 * contract of Fig 4 of this technical report, thus
 * it accounts for technical failures.
 *
 * To run this code you need
 * 1) Spin Version 6.1.0 or a more recent one.
 * 2) The macros setting.h and BizOperation.h
 *    vector.lpr and for.h deployed in your
 *    working folder.
 * 3) rules.h in your working folder.
 *
 * 4) Edit BuyerStoreContract.pml to comment
 * and uncomment the LTL provided
 * in the code as needed. Keep in mind that
 * Spin can verify only a single LTL at a time.
 *
 * 5) To run the code from Linux type:
 * % spin -a BuyerStoreContract.pml
 * % cc -o pan pan.c
 * % pan -a
 *
 * Notation used in this code:
 * S-success, BF-business failure, TF-technical
 * failure, TO--timeout, exec--execution
 */

#include "setting.h"          /* macro definition */
#include "BizOperation.h" /* macro definition */
#include "rules.h"              /* ECA rule code    */

#define TRUE  1
#define FALSE 0
#define YES   1
#define NO    0

#define AbnContractEnd (abncoend==TRUE)

/* var for recording occurrences of executions
 * with S and BF outcomes
 */
bool abncoend=FALSE;
bool ReqFailBefore=NO;
bool PayFailBefore=NO;

/* declaration of the 2 role players involved */
RolePlayer(BUYER,STORE);

/* account for S,BF,TF,TO execution outcome,
 * in this ex, we use only S and BF */
RuleMessage(S,BF,TF,TO);

/* 5 operations are involved in the contract */
BIS_OP(BUYREQ);
BIS_OP(BUYREJ);
BIS_OP(BUYCONF);
BIS_OP(BUYPAY);
BIS_OP(BUYCANC);


/*
 * LTLs for expressing, mutual exclusion of
 * obligations and prohibitions which are
 * checked by default
 */

/*
 * When the buyer is obliged to pay the obligation
 * remains pending until the buyer pays or cancels
 * or the contract terminates abnormally.
 */
/* ltl p0 { [](!IS_O(BUYPAY,BUYER)) ||
   <>(IS_O(BUYPAY,BUYER) -> <> (IS_X(BUYPAY,BUYER)
   || IS_X(BUYCANC,BUYER) || AbnContractEnd))  } */


/*
 * trap LTL for generating execution sequences
 */
/*
 * the following LTL can be used for generation
 * sequences that include the execution of BUYREJ
 * including the simplest one: BUYREQ(S)-> BUYREJ(S).
 */
/* ltl p1 { !<>IS_X(BUYREJ,STORE) } */

/*
 * The following LTL can be used for generating
 * exe sequences that include the successful
 * and bizfail execution of operations that
 * eventually complete in a successful execution
 * of BUYPAY
 */
/* ltl p2 { !<>IS_X(BUYPAY,BUYER) } */

/*
 * The following LTL can be used for generating
 * exe sequences that include the successful
 * and tecfail execution of operations that
 * eventually complete in a successful execution
 * of BUYCANC
 */
ltl p3 { !<>IS_X(BUYCANC,BUYER) }



/* Business Event Generator */
proctype BEG()
{
 BEGIN_INIT:
 {
 /* Define the initial state of the rights (R),
  * obligations (O) and prohibitions (P) of the 2
  * role players following:
  * INIT(OperName, RolePlayerName, R,O,P)
  * 1 means granted, 0 means not granted
  * In initial state buyer has been granted the
  * right to execute BUYREQ. No other R,O,P are
  * granted to buyer or store */
  DONE(BUYER);
  DONE(STORE);

  INIT(BUYREQ,  BUYER, 1,0,0);
  INIT(BUYREJ,  STORE, 0,0,0);
  INIT(BUYCONF, STORE, 0,0,0);
  INIT(BUYPAY,  BUYER, 0,0,0);
  INIT(BUYCANC, BUYER, 0,0,0);
 }
 END_INIT:

 /* generation of business events.
  * For each of the 5 operations, 2 possible exec
  * are modelled: exec with S and exec with TF */
 end:do
 :: B_E(BUYER, BUYREQ,  S);
 :: B_E(BUYER, BUYREQ,  TF);

 :: B_E(STORE, BUYREJ,  S);
 :: B_E(STORE, BUYREJ,  TF);

 :: B_E(STORE, BUYCONF, S);
 :: B_E(STORE, BUYCONF, TF);

 :: B_E(BUYER, BUYPAY,  S);
 :: B_E(BUYER, BUYPAY,  TF);

 :: B_E(BUYER, BUYCANC, S);
 :: B_E(BUYER, BUYCANC, TF);
 od;
}

/* contract rule manager: it uses the rules.h
 * file declared in the inline definition.
 * It retrieves and includes the rule (one at a
 * time) needed to respond to the event under
 * process
 */
proctype CRM()
{
 printf("CONTRACT RULE MANAGER");
 end:do
  :: CONTRACT(BUYREQ);  /* include RULE(BUYREQ) */
  :: CONTRACT(BUYREJ);  /* include RULE(BUYREJ) */
  :: CONTRACT(BUYCONF);
  :: CONTRACT(BUYPAY);
  :: CONTRACT(BUYCANC);
 od;
}

init
{
  atomic /* start exec of BRG and CRM */
  {
   run BEG(); run CRM();
  }
}
```

```
/*
 * Carlos Molina J., 17 Apr 2013, Ncl Uni, UK
 *
 * rules.h: EPROMELA code of the ECA rules that
 * implement a contract between a buyer and store.
 * The code prints out messages with xml like
 * tags which can be used for signaling out
 * messages when the model is used for generating
 * execution sequences.
 *
 * Notation used in this code: cont-contract,
 * SC-success, TF-technical failure,
 * tecfail- technical failure.
 */

/* Rule triggered by buyreq executions initiated
 * by the buyer and completed either in success or
 * technical failure.
 */
RULE(BUYREQ)
{
 WHEN::EVENT(BUYREQ,IS_R(BUYREQ,BUYER),SC(BUYREQ))
  /* handle buyreq with success outcome */
  ->{ SET_X(BUYREQ,BUYER);
     atomic{
     printf("\n\n");
     printf("<originator>buyer</originator>\n");
     printf("<responder>store</responder>\n");
     printf("<type>BUYREQ</type>\n");
     printf("<status>success</status>\n");
     printf("\n\n")
     }
     SET_R(BUYREQ,0);
     SET_O(BUYREJ,1);
     SET_O(BUYCONF,1);
     RD(BUYREQ,BUYER,CCR,CO);
    }
 /* handle buyreq with technical failure outcome */
 ::EVENT(BUYREQ,IS_R(BUYREQ,BUYER),TF(BUYREQ))
```

```
  ->{
     atomic{
     printf("\n\n");
     printf("<originator>buyer</originator>\n");
     printf("<responder>store</responder>\n");
     printf("<type>BUYREQ</type>\n");
     printf("<status>tecfail</status>\n");
     printf("\n\n")
     }
    if /* 1st notification of buyreq with TF */
    :: (ReqFailBefore==NO) ->ReqFailBefore=YES;
     printf("First BUYREQ-TechnicalFailure");
     RD(BUYREQ,BUYER,CCR,CO);

    /* 2nd notification of buyreq with TF */
    :: (ReqFailBefore==YES) -> abncoend=TRUE;
     printf("Last BUYREQ-TechnicalFailure");
     SET_R(BUYREQ,0);
     atomic{
     printf("\n\n");
     printf("<originator>reset</originator>\n");
     printf("<responder>reset</responder>\n");
     printf("<type>reset</type>\n");
     printf("<status>reset</status>\n");
     printf("\n\n")}

     RD(BUYREQ,BUYER,CCR,CND);/*abnormal cont end*/
    fi
    }
 END(BUYREQ);
}



/* Rule triggered by buyrej executions initiated
 * by the store and completed either in success or
 * technical failure.
 */
RULE(BUYREJ)
{
 /* handle buyrej with success outcome */
 WHEN::EVENT(BUYREJ,IS_O(BUYREJ,STORE),SC(BUYREJ))
  ->{ SET_X(BUYREJ,STORE);
     atomic{
     printf("\n\n");
     printf("<originator>store</originator>\n");
     printf("<responder>buyer</responder>\n");
     printf("<type>BUYREJ</type>\n");
     printf("<status>success</status>\n");
     printf("\n\n")
     }
     SET_O(BUYREJ,0);
     SET_O(BUYCONF,0);
     atomic{
     printf("\n\n");
     printf("<originator>reset</originator>\n");
     printf("<responder>reset</responder>\n");
     printf("<type>reset</type>\n");
     printf("<status>reset</status>\n");
     printf("\n\n")}

     RD(BUYREJ,STORE,CCO,CND);
    }
 /* handle buyrej with technical failure outcome */
 ::EVENT(BUYREJ,IS_O(BUYREJ,STORE),TF(BUYREJ))
  ->{
     atomic{
     printf("\n\n");
     printf("<originator>store</originator>\n");
     printf("<responder>buyer</responder>\n");
     printf("<type>BUYREJ</type>\n");
     printf("<status>tecfail</status>\n");
     printf("\n\n")
     }
    if /* 1st notification of buyrej with TF */
    :: (RejFailBefore==NO) ->RejFailBefore=YES;
     printf("First BUYREJ-TechnicalFailure");
     RD(BUYREJ,STORE,CCO,CO);

    /* 2nd notification of buyrej with TF */
    :: (RejFailBefore==YES) -> abncoend=TRUE;
     printf("Last BUYREJ-TechnicalFailure");
     SET_O(BUYREJ,0);
     SET_O(BUYCONF,0);
     atomic{
     printf("\n\n");
     printf("<originator>reset</originator>\n");
     printf("<responder>reset</responder>\n");
     printf("<type>reset</type>\n");
     printf("<status>reset</status>\n");
     printf("\n\n")}

     RD(BUYREJ,STORE,CCO,CND);/*abnormal cont end*/
    fi
    }
 END(BUYREJ);
}



/* Rule triggered by buyconf executions initiated
 * by the store and completed either in success or
 * technical failure.
 */
RULE(BUYCONF)
{
 /* handle buyconf with success outcome */
 WHEN::EVENT(BUYCONF,IS_O(BUYCONF,STORE),SC(BUYCONF))
  ->{ SET_X(BUYCONF,STORE);
     atomic{
     printf("\n\n");
     printf("<originator>store</originator>\n");
     printf("<responder>buyer</responder>\n");
     printf("<type>BUYCONF</type>\n");
     printf("<status>success</status>\n");
     printf("\n\n")
     }
     SET_O(BUYREJ,0);
     SET_O(BUYCONF,0);
     SET_O(BUYPAY,1);
     SET_O(BUYCANC,1);
     RD(BUYCONF,STORE,CCO,CO);
    }
 /* handle buyconf with technical failure outcome */
 ::EVENT(BUYCONF,IS_O(BUYCONF,STORE),TF(BUYCONF))
  ->{
     atomic{
     printf("\n\n");
     printf("<originator>store</originator>\n");
     printf("<responder>buyer</responder>\n");
     printf("<type>BUYCONF</type>\n");
     printf("<status>tecfail</status>\n");
     printf("\n\n")
     }
    if /* 1st notification of buyconf with TF */
    :: (ConfFailBefore==NO) ->ConfFailBefore=YES;
     printf("First BUYCONF-TechnicalFailure");
     RD(BUYCONF,STORE,CCO,CO);

    /* 2nd notification of buyconf with TF */
    :: (ConfFailBefore==YES) -> abncoend=TRUE;
     printf("Last BUYCONF-TechnicalFailure");
     SET_O(BUYREJ,0);
     SET_O(BUYCONF,0);
     atomic{
     printf("\n\n");
     printf("<originator>reset</originator>\n");
     printf("<responder>reset</responder>\n");
     printf("<type>reset</type>\n");
     printf("<status>reset</status>\n");
     printf("\n\n")}

     RD(BUYCONF,STORE,CCO,CND);/*abnormal cont end*/
    fi
    }
 END(BUYCONF);
}



/* Rule triggered by buypay executions initiated
```

```
 * by the buyer and completed either in success or
 * technical failure.
 */
RULE(BUYPAY)
{
 printf("BUYPAY rule (first lines) \n");
 /* handle buypay with success outcome */
 WHEN::EVENT(BUYPAY,IS_O(BUYPAY,BUYER),SC(BUYPAY))
  ->{SET_X(BUYPAY,BUYER);
     atomic{
     printf("\n\n");
     printf("<originator>buyer</originator>\n");
     printf("<responder>store</responder>\n");
     printf("<type>BUYPAY</type>\n");
     printf("<status>success</status>\n");
     printf("\n\n")
     }
     SET_O(BUYPAY,0);
     SET_O(BUYCANC,0);

     atomic{
     printf("\n\n");
     printf("<originator>reset</originator>\n");
     printf("<responder>reset</responder>\n");
     printf("<type>reset</type>\n");
     printf("<status>reset</status>\n");
     printf("\n\n")}

     RD(BUYPAY,BUYER,CCO,CND);/*ideal cont end*/
    }
 /* handle buypay with technical failure outcome */
 ::EVENT(BUYPAY,IS_O(BUYPAY,BUYER),TF(BUYPAY))
  ->{
     atomic{
     printf("\n\n");
     printf("<originator>buyer</originator>\n");
     printf("<responder>store</responder>\n");
     printf("<type>BUYPAY</type>\n");
     printf("<status>tecfail</status>\n");
     printf("\n\n")
     }
    if /* 1st notification of buypay with TF */
    :: (PayFailBefore==NO) ->PayFailBefore=YES;
     printf("First BUYPAY-TechnicalFailure");
     RD(BUYPAY,BUYER,CCO,CO);

    /* 2nd notification of buypay with TF */
    :: (PayFailBefore==YES) -> abncoend=TRUE;
     printf("Last BUYPAY-TechnicalFailure");

     SET_O(BUYPAY,0);
     SET_O(BUYCANC,0);
     atomic{
     printf("\n\n");
     printf("<originator>reset</originator>\n");
     printf("<responder>reset</responder>\n");
     printf("<type>reset</type>\n");
     printf("<status>reset</status>\n");
     printf("\n\n")}

     RD(BUYPAY,BUYER,CCO,CND);/*abnormal cont end*/
    fi
    }
 END(BUYPAY);
}



/* Rule triggered by buycanc executions initiated
 * by the buyer and completed either in success or
 * technical failure.
 */
RULE(BUYCANC)
{
 /* handle buycanc with success outcome */
 WHEN::EVENT(BUYCANC,IS_O(BUYCANC,BUYER),SC(BUYCANC))
  ->{ SET_X(BUYCANC,BUYER);
     atomic{
     printf("\n\n");
     printf("<originator>buyer</originator>\n");
     printf("<responder>store</responder>\n");
     printf("<type>BUYCANC</type>\n");
     printf("<status>success</status>\n");
     printf("\n\n")
     }

     SET_O(BUYPAY,0);
     SET_O(BUYCANC,0);
     atomic{
     printf("\n\n");
     printf("<originator>reset</originator>\n");
     printf("<responder>reset</responder>\n");
     printf("<type>reset</type>\n");
     printf("<status>reset</status>\n");
     printf("\n\n")}

     RD(BUYCANC,BUYER,CCO,CND);
    }
 /* handle buycanc with technical failure outcome */
 ::EVENT(BUYCANC,IS_O(BUYCANC,BUYER),TF(BUYCANC))
  ->{
     atomic{
     printf("\n\n");
     printf("<originator>buyer</originator>\n");
     printf("<responder>store</responder>\n");
     printf("<type>BUYCANC</type>\n");
     printf("<status>tecfail</status>\n");
     printf("\n\n")
     }
    if /* 1st notification of buycanc with TF */
    :: (CancFailBefore==NO) ->CancFailBefore=YES;
     printf("First BUYCANC-TechnicalFailure");
     RD(BUYCANC,BUYER,CCO,CO);

    /* 2nd notification of buycanc with TF */
    :: (CancFailBefore==YES) -> abncoend=TRUE;
     printf("Last BUYCANC-TechnicalFailure");
     SET_O(BUYPAY,0);
     SET_O(BUYCANC,0);
     atomic{
     printf("\n\n");
     printf("<originator>reset</originator>\n");
     printf("<responder>reset</responder>\n");
     printf("<type>reset</type>\n");
     printf("<status>reset</status>\n");
     printf("\n\n")}

     RD(BUYCANC,BUYER,CCO,CND);/*abnormal cont end*/
    fi
    }
 END(BUYCANC);
}
```

Appendix B.
Code of Buyer–Seller Contract Produced by BPMN2PROMELA

```
/*
 * Carlos Molina-Jimenez, 18 Apr 2013, Ncl Uni, UK
 * Carlos.Molina@ncl.ac.uk
 *
 * PROMELA code with an LTL formula included
 * of the choreography shown in Fig 2-a.
 * 1) The code was produced automatically
 * by the BPMN2PROMELA given the xml description
 * of the BPMN choreography diagram.
 * 2) The LTL used in this example can be
 *    classified as a response: "if confirmation
 *    happens, it will be eventually followed by
 *    either payment or cancellation". It was
 *    included mechanically into the PROMELA
 *    code by the BPMN2PROMELA tool from a
 *    template.
 * 3) The PROMELA code with the LTL included
 *    were presented to Spin (which is integrated
 *    to the BPME2PROMELA tool) for validation.
 *    As expected, the Spin output shows that the
 *    LTL is satisfied by the model.
 *
 * It is worth clarifying that apart from the
 * deletion of some of the lines produced by
 * Spin, the PROMELA code and the never claim
 * of the LTL are shown as produced by the tool.
 */

#define TRUE 1
#define FALSE 0
bool BuyConfRcv = FALSE;
#define a (BuyConfRcv == TRUE)
bool BuyPayRcv = FALSE;
#define b (BuyPayRcv == TRUE)
bool BuyRejRcv = FALSE;
#define c (BuyRejRcv == TRUE)
bool BuyCancRcv = FALSE;
#define d (BuyCancRcv == TRUE)
bool BuyReqRcv = FALSE;
#define e (BuyReqRcv == TRUE)
/*Potential outcomes from each operation*/
mtype = {BuyConf, BuyRej, BuyPay, BuyCanc, BuyReq};


/*All channels in the diagram*/
chan Buyer2Store = [0] of {mtype, byte};
chan Store2Buyer = [0] of {mtype, byte};


/*All processes involved in the choreography*/
proctype Buyer() {

Buyer2Store ! BuyReq(1);
if
::
atomic {Store2Buyer ? BuyRej(_); BuyRejRcv = TRUE;
}

::
atomic {Store2Buyer ? BuyConf(_); BuyConfRcv = TRUE;
}
if
if
::
Buyer2Store ! BuyPay(1);

::
Buyer2Store ! BuyCanc(1);

fi;

fi;

}



proctype Store() {

atomic {Buyer2Store ? BuyReq(_); BuyReqRcv = TRUE;
}
if
::
Store2Buyer ! BuyRej(1);

::
Store2Buyer ! BuyConf(1);
if
::
atomic {Buyer2Store ? BuyPay(_); BuyPayRcv = TRUE;
}

::
atomic {Buyer2Store ? BuyCanc(_); BuyCancRcv = TRUE;
}

fi;

fi;

}



init {
atomic {run Buyer();
run Store();
}
}



/*** LTL to satisfy ***/
never  {    /* !([] (a -> <> (b || d))) */
T0_init:
        if
        :: (! ((b || d)) && (a)) -> goto accept_S4
        :: (1) -> goto T0_init
        fi;
accept_S4:
        if
        :: (! ((b || d))) -> goto accept_S4
        fi;
}


/*
 * Spin validation outputs
 */
(Spin Version 6.2.3 -- 24 October 2012)
        + Partial Order Reduction

Full statespace search for:
        never claim           + (never_0)
        assertion violations - (disabled by -A flag)
        cycle checks         - (disabled by -DSAFETY)
        invalid end states   - (disabled by -E flag)

State-vector 60 byte, depth reached 19, errors: 0
       18 states, stored
        3 states, matched
        21 transitions (= stored+matched)

```