A FORMAL APPROACH SUPPORTING THE SPECIFICATION AND VERIFICATION OF BUSINESS CONVERSATION REQUIREMENTS

Alessandra Cherubini  
Dipartimento di Matematica, Politecnico di Milano  
Piazza Leonardo da Vinci 32, Italy

Enzo Colombo, Chiara Francalanci, Paola Spoletini  
Dipartimento di Elettronica e Informazione, Politecnico di Milano  
Via Ponzio 34/5, Italy

ABSTRACT
How to verify adaptable business conversations during the requirements engineering process is still an open issue, especially when conversation requirements are translated into a network of cooperating e-services. Model checking is a suitable technique to address this problem. However, since requirements engineers typically adopt less theoretical models, model checking is given little attention. In this paper, we propose a statecharts-like model that supports the specification of business conversation requirements. These requirements are expressed as quality of service (QoS) and structural properties related to the flow of actions implemented by each cooperating partner. This flow is partly standard and partly exceptional. The exceptional flow specifies how conversations should adapt their behavior as a consequence of violations of QoS constraints. Moreover, the paper shows that this specification can be formally defined through a predicative time-bounded automaton which, in turn, can be translated into Promela, the input language of the SPIN model checker. A case study is discussed to show the application of the model.

KEYWORDS
Requirements engineering process, business conversations, service-oriented modeling, testing and verification.

1. INTRODUCTION

Over the years, companies along the same value chain have designed their software applications according to different technological standards and, they have implemented several architectures to support interoperability among their heterogeneous software systems in order to automate their business relationships. In this context, ebXML has been recently proposed as the final solution to the problem of interoperability providing a specification language and an architecture shifting the logic of integration from information to service exchange [EbXML Project].

However, ebXML does not support flexible inter-organizational conversations. Therefore, organizations cannot readily adapt their relationships according to changes in their operative environment. For example, a previous agreement cannot be re-negotiated during the execution of a collaborative activity and a partner cannot be automatically replaced when cooperating goals are not fulfilled. A primary objective of service-oriented architectures (SOA) is to overcome these limitations [SOC Manifesto 2002, Vispo Project].

The present work is focused on modeling requirements of adaptable business conversations then implemented according to a service-oriented approach. In particular, our work is concerned on business conversation requirements, i.e. all the aspects of the description of business conversations that are related to the strategy and the rationale of cooperating organizations (the “why” and the “what”), and that precede and motivate the actual definition of control and coordination mechanisms among interacting parties (the “how”) [Baresi et al. 2004, EbXML Project]. In particular, a business conversation describes the relationships among
cooperating organizations according to a global, neutral perspective, in terms of valid control and coordination mechanisms. A business conversation is usually public, since it specifies the common rules defining a valid interaction among distributed business processes.

Most of the research work on service-oriented modeling of business conversations has been done at a design level of abstraction [Bultan et al. 2003]. For example, Bultan et al. studied a framework supporting the specification of the global behavior of interacting e-services. Under this framework, individual e-services communicate asynchronously and each service maintains a queue for incoming messages. Other contributions proposed models at a higher level of abstraction mainly oriented to the development of architectures supporting e-services automatic discovery and composition [Wombacher and Mahleko 2002, Mecella and Pernici 2001]. However, little attention has been given to the modeling of business conversations during the requirements engineering process [Baresi et al. 2004] that is, on the contrary, the primary goal of the present work.

Traditionally, the requirements engineering process is organized into early- and late-requirements analysis [Castro et al. 2002]. Early requirements analysis is concerned with the understanding of the problem by studying organizations and their static dependencies. Late requirements analysis should describe the conversation with its relevant functionalities, qualities and conceptual behavior over time also providing tools to support requirements verification. In a previous work [Colombo et al. 2004], a model supporting the specification of business conversations during early requirements analysis was studied. The present work is instead concerned to support the automatic verification of business conversations requirements during the late-requirements engineering process. We note that the automatic verification of properties on a formal model of requirements is an important aspect of the requirements engineering process, since it guarantees that designers do not receive inconsistent specification that could result into an ill-designed software system.

In general, several formal approaches have been proposed in the field of requirements engineering to deal with the problem of verification. For example, the KAOS framework provides a formal approach for analyzing and transforming goals into software requirements [Darimond et al 1998]. However, it relies on theorem proving as opposed to model checking techniques. Hence, KAOS cannot provide counterexamples useful to support modelers in their iterative process of requirements refinement. Moreover, in the Tropos project, a linear-time temporal logic based on the \( i^* \) social model is proposed. Automatic verification is also supported through symbolic model checking [Fuxman et al. 2004]. However, a social representation of requirements is inadequate to model data and control flows typical of business conversations.

The closest research to our work has been presented in [Janssen et al. 1998] where a technique to verify AMBER specifications through the SPIN model checker is studied. The AMBER language has been proposed to support Business Process Engineering (BPR) and it has the typical semantics of Event-based Process Chains (EPCs) [van der Aalst 1999]. On the contrary, our proposal is based on statechart semantics and embeds concepts to represent compensations and quality of service (QoS) requirements in an explicit way consistently with the definition of business conversation. Moreover, this paper proposes a different theoretical model underlying the Promela representation of business requirements. In [Janssen et al. 1998] parallelism is not actually supported since the Büchi automaton [Holtzmann 2003] representing the AMBER specification prefixes an order or explicitly enumerates the possible combinations of states whenever parallelism is required. On the contrary, our technique is based on the adoption of an exponentially more concise alternating automaton [Vardi 1994] and it deals explicitly with time.

The paper is organized as follows. Next section presents our formal model for late-requirements analysis of business conversations together with a set of rules supporting its translation into Promela, the input language of the SPIN model checker. Sect. 3 exemplifies the application of our approach on a small case study based on the industrial district of Matera (Italy) and Section 4 experiments the verification of a set of requirements on the model of conversation. Finally, conclusions are drawn and directions for future work are discussed.

2. FORMAL MODEL

Business conversations are modeled as a reactive system through a variant of classical statecharts, called Business Conversation Charts, in short BCC. Referring to Harel’s statecharts [Harel 1996], in the next section we describe our model pointing out the differences with the classical model. Then Sect. 2.2 formally
defines the semantics of BCCs and Sect. 2.3 presents a set of rules supporting the translation of conversations requirements into the Promela language of the SPIN model checker.

2.1 Business Conversation Charts

As in statechart, BCCs support the iterative refinement of requirements through hierarchical, AND- and OR-states. Moreover, states are connected through labeled transitions. As in statecharts transition are labeled with ECA rules, i.e. expressions of the form \{null, e/a, e[c]/ξ, e[c]/a\}, where e is the event that triggers the transition, c is a guard condition, and a is an action (ξ indicates no action). However, in BCC we label transitions with particular type of events, conditions and compensation actions, as described in Definition 1, 2 and 3.

**Definition 1** Events are only of two types \{Begin\(tl\), End\(tl\)\} where \(tl\) is a task, with the natural meaning of beginning and end of the task passed as argument.

**Definition 2** A condition is a predicate \(p\), that can be categorized in the following classes:

1. If \(p\) has the form \(\text{Achieved}(s, g)\), where \(g\) is a goal and \(s\) is an actor, it is a goal condition.
2. If \(p\) has one of the forms \(\text{Fulfilled}(a)\) or \(\text{Done}(a)\), where \(a\) is a compensation action, \(p\) is called compensation condition.
3. If \(p\) has the form (i) \([p•c],\) where • is one of \(\{≤, ≥, =, <, >\}\), \(p\) is a variable, \(c\) is a constant and the square brackets with the index \(t\) denote that \(c\) and \(p\) are of the same type \(t\) or (ii) \(\text{Received}(x, s, r)\) where \(r\) is a resource, \(s\) is an actor, \(x\) is a temporal variable and \(l\) is a temporal interval, \(p\) is a resource condition. For short, in the following we will denote the form (ii) as \(\text{Received}(l, s, r)\).
4. If \(p\) is a resource condition of the form (i) where \(t\) is a temporal type, it is a temporal condition.

The meaning of predefined conditions is the natural one suggested by their name. For example, \(\text{Received}[1,5]\text{days},\) seller, order\), is a resource condition stating that an order is expected by a seller within 1 to 5 days. Moreover, a resource condition constraining the size of a wheel is defined as \([\text{wheel-diameter}=4]\text{cm}\), where \(\text{wheel-diameter}\) is a resource variable, 4 is the constant value and \(\text{cm}\) is the type. An example of temporal condition constraining process lead-time is defined as \([\text{lead-time}>10]\text{days}\), where lead time is the temporal variable, 10 is the constant value and \(\text{days}\) is the type.

**Definition 3** Actions are either compensation actions or the flow-control action reset_history. Compensation actions are classified in delay, re-execute, re-negotiate and re-transact action classes and are shown in Table 1. Actions can be composed by means of the logical operators ¬, ∧, ∨, and the operator Sequence. When actions are composed with ∨, the engine can select the function to be enacted either randomly or according to lower-level implementation criteria (e.g. improvement of performance). The Sequence operator involves the execution of a finite number of compensation actions in a sequence. However, compensation stops at the first successful compensation action in the sequence. For example, compensation actions within the re-negotiate class can be executed only within the negotiation phase.

<table>
<thead>
<tr>
<th>Class</th>
<th>Action</th>
<th>Description</th>
<th>Semantics and usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay</td>
<td>Wait((x);) (x \in X)</td>
<td>Wait while (x) is verified.</td>
<td>There is not an urgency around the execution of a task pending for a resource. This scenario applies either when lead-time is not mandatory or when this task should be executed in parallel with a temporally longer flow.</td>
</tr>
<tr>
<td></td>
<td>Waitfor((x, r)); (X \subseteq X, r \in R)</td>
<td>Wait for information resource (r) while (x) is verified.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Delay((x, s, t)); (X \subseteq X, s \in S, t \in TL)</td>
<td>Delay the execution of task (t) by actor (s) while (x) is verified.</td>
<td>Adds a delay to a task. Accordingly, the residence time of the state representing this task is incremented by the delay. This scenario applies when penalties on the execution of a task are implemented (e.g. delay of payment)</td>
</tr>
<tr>
<td>Informative</td>
<td>Urge((x, s, r)); (X \subseteq X, s \in S, r \in R)</td>
<td>Urge resource (r) from actor (s) while (x) is verified.</td>
<td>There is an urgency for a resource. This resource feeds a strategic task that without specific information cannot run.</td>
</tr>
</tbody>
</table>

469
Re-negotiate

Relax(x, s, y); x ∈ X, s ∈ S, y ∈ X,

Actor x is asked to relax a previous constraint y while x is verified.

Tighten(x, s, y); x ∈ X, s ∈ S, y ∈ X,

Actor x is asked to tighten a previous constraint y while x is verified.

Delete(x, s, y); X ∈ X, s ∈ S, y ∈ X,

Actor x is asked to delete a previous constraint y while x is verified.

Re-execute

Re-execute(x, s, tl); x ∈ X, s ∈ S, tl ∈ K

Actor x re-executes task tl while x is verified.

Require the re-execution of a task. It is typically performed when a mandatory requirement is not fulfilled.

Re-execute from(x, s, tl); x ∈ X, s ∈ S, tl ∈ K

Actor x re-executes from task tl while x is verified.

Require the re-execution of a hierarchical task from a well-defined state. It is typically performed when only part of an already executed process must be re-executed.

Skip(x, s, tl); x ∈ X, s ∈ S, tl ∈ K

Actor x skips task tl while x is verified.

Require to skip a task over the run of a business conversation.

Re-transact

Delegate(x, s, tl); x ∈ X, s ∈ S, tl ∈ K

Task tl is delegated to a different actor to be identified through matchmaking for re-execution while x is verified.

Delegates a task to another actor within the community. This compensation is typically performed before aborting the current instance of the conversation.

An other main difference is in the label of the states. In statecharts, states can be labeled with ECA rules, while the transition labels of the BCC are defined as in Definition 4.

Definition 4. A state label lq is a 3-uple lq = <tl, s, x> where tl is an operating task, s is the actor implementing the task and x is a temporal condition. The initial state q0 is not provided of a state label. Final state labels are modeled as <{commit, abort}, ∅, ∅>.

The third difference with statecharts is in the managing of time. In BCC we do not specify time on state transitions, but state transitions result timed iff they are labeled with a time consuming action. Overall, a business conversation either commits or is aborted in a finite time period.

The last main difference is in the description of the History. While in statecharts the history is described as connectors, in BCC it is a label of the states, defined as the capability of a hierarchical state to store its internal evolution as a consequence of an interleaved transition, i.e. a transition from any sub-state to an external destination state. And- and Or-states does not admit interleaving.

2.2 Formal Semantics

In this section we formalize the semantics of BCC, through an intermediate automaton representation, called Predicative Time-bounded Automaton (PTA). A PTA provides both existential and universal branching modalities. Quite naturally, ∧ is used to denote universality, while ∨ denotes existentiality. These automata represent a useful tool since they can be exponentially more concise than non-deterministic automata, and are well suited for dealing with logic formulae [Chandra 1981, Vardi 1994]. The use of both the transition modalities is necessary to represent the AND-states in a concise way, avoiding the combinatorial explosion that arises when explicitly enumerating all the possibilities of the AND-state parallelism.

Let

1. \( \Pi = \{ \tau, \pi_1, \pi_2, \ldots, \pi_n \} \) be a finite set of types where \( \tau \) stands for time,
2. \( x_1, x_2, \ldots, x_n \) be a finite set of variables such that \( \sigma(x_i) \in \Pi \), where \( \sigma \) is a function providing the type associated with \( x_i \),
3. \( \land, \lor, \neg \) be logical connectives,
4. \( \mathcal{P} = \{ P^n | P \in \Pi \} \), where \( I \) is a finite set, be the set of predicate symbols where the superscript \( n \) is the arity of the predicate name \( P \), \( P \) contains at least Done, Fulfilled, End and Begin of arity 1, Achieved, ≥, ≤, =, > and < of arity 2 and Received of arity 3.
5. \( P^n(x_1, x_2, \ldots, x_n) \) be an atomic formula, where \( \sigma(P^n) = \langle \sigma(x_1), \sigma(x_2), \ldots, \sigma(x_n) \rangle \in \Pi^n \) provides the prototype of \( P^n \).
Definition 5. Let $C$ be a set of symbols representing compensation actions, $S$ a set of symbols representing actors, $K$ a set of symbols representing tasks and $P$ the set of predicates symbols. A predicative time-bounded automaton on $C, S, K$ is a 7-uple: $A = <X, \Sigma, Q, q_0, B, \tau, F>$ with $\Sigma = \Sigma_C \cup \Sigma_Q \cup \Sigma_A$ where:

1. $X$ is a set of temporal conditions of the form $}\mathit{p}\mathit{\cdot}\mathit{c}$\rangle, $\mathit{e}\mathit{-}\mathit{\langle} \mathit{\leq}, \mathit{\geq}, \mathit{\prec}, \mathit{\succ} \mathit{\rangle} \in \mathit{P}$ where $p \in X$ is a temporal variable with values in $N$ and $c \in N$ is a constant. Any conjunction of temporal conditions through logical connectives is also a temporal condition.
2. $\Sigma_C$ is formed by predefined tuples obtained as in Table 2 from Received, Begin, End, Done, Fulfilled and Achieved and their Boolean combinations.
3. $\Sigma_Q = \langle q, s, x \rangle$ is a set of triples where $q \in C$ is a symbol representing a compensation action, $s \in S$ is a symbol representing an action and $x \in X$ is a temporal condition.
4. $\Sigma_Q = \langle q, s, x \rangle$ is a set of triples where $q \in K$ is a symbol representing a task, $s \in S$ is a symbol representing an actor and $x \in X$ is a temporal condition.
5. $Q$ is a finite set of states.
6. $q_0 \in Q$ is the initial state of the automaton.
7. $B = \{b_1, b_2, \ldots, b_n\}$ is a set of counter variables, shared among the states of the automaton in order to store its evolution (history).
8. $\tau: Q \times \Sigma_C \times \Sigma_Q \times B \rightarrow Q \times \Sigma_A \times B$ is a transition function.
9. $F \subseteq Q$ is the set of final states of the automaton.

We now informally show the semantics of each component of BCC through the PTA. State and transition labels, events and guard conditions are represented in the PTA, using a renaming function $f$. In particular, synchronous events and resource, compensation and goal conditions are represented into a tuples according to Table 2.

Table 2. Mapping of formulas into conditions.

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Formula</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compensation</td>
<td>$\neg \mathit{Done}(a)$</td>
<td>$&lt;\mathit{action_name}=0, s \in S, x \in X&gt;$</td>
</tr>
<tr>
<td></td>
<td>$\mathit{Done}(a)$</td>
<td>$&lt;\mathit{action_name}=1, s \in S, x \in X&gt;$</td>
</tr>
<tr>
<td></td>
<td>$\mathit{Done}(a)$, $\neg \mathit{Fulfilled}(a)$</td>
<td>$&lt;\mathit{action_name}=2, s \in S, x \in X&gt;$</td>
</tr>
<tr>
<td></td>
<td>$\mathit{Done}(a)$, $\mathit{Fulfilled}(a)$</td>
<td>$&lt;\mathit{action_name}=3, s \in S, x \in X&gt;$</td>
</tr>
<tr>
<td>Goal conditions</td>
<td>$\neg \mathit{Achieved}(s, g)$</td>
<td>$&lt;\mathit{goal_name}=0, s \in S&gt;$</td>
</tr>
<tr>
<td></td>
<td>$\mathit{Achieved}(s, g)$</td>
<td>$&lt;\mathit{goal_name}=1, s \in S&gt;$</td>
</tr>
<tr>
<td>Resource conditions</td>
<td>$\neg \mathit{Received}(s, r)$</td>
<td>$&lt;\mathit{Resource_name}=0, s \in S, x \in X&gt;$</td>
</tr>
<tr>
<td></td>
<td>$\mathit{Received}(s, r)$</td>
<td>$&lt;\mathit{Resource_name}=1, s \in S, x \in X&gt;$</td>
</tr>
<tr>
<td>Synchronous events</td>
<td>$\mathit{Begin}(t)$</td>
<td>$&lt;\mathit{Task_name}=0, \mathit{null}, \mathit{null}&gt;$</td>
</tr>
<tr>
<td></td>
<td>$\mathit{End}(t)$</td>
<td>$&lt;\mathit{Task_name}=1, \mathit{null}, \mathit{null}&gt;$</td>
</tr>
</tbody>
</table>

Notice that $x \in X$ is elicited from the time consuming compensation action. Moreover, compensation actions are directly mapped into $<q, s, x> \in \Sigma_A$. This function associates each compensation action with both a temporal and a conditional condition elicited from the compensation action itself (see Table 3, row 1).

When the BCC does not include any complex state, i.e. hierarchical, AND- and OR-states, it is naturally reduced to the PTA, otherwise we need to specify accurately the reduction of BCC in PTA.

Hierarchical states are represented as shown in Table 3, row 2. In details, if $q$ is a simple hierarchical state and $Q_q = \{q_1, q_2, \ldots, q_n\}$ the set of its sub-states, then this hierarchy is represented through a dummy state $q$ and two edges $(q, q_1)$ and $(q, q_n)$. The transition from the dummy state $q$ to $q_i$ has as guard the condition $b=0$ with $b \in B$. Moreover, the transition from $q_i$ to $q$ contains the assignment $b:=n$. Note that before the first visit to $q$, $b$ is initialized to zero, meaning that the first visit to a hierarchical state is treated as a visit to the first of its sub-states (see Table 3 row 2, transition). Moreover, the set of transitions connecting the sub-states of $q$ are preserved during the translation and the value of the history variable $b$ is incremented during each transition. The transition from $q$ to the next state is augmented with the guard $b=n$ to guarantee that the sub-automaton is all visited and with the assignment $b=0$ to enable the visit of the sub-automaton if the hierarchical state has to be visited an other time. We note that a sub-automaton is simply a statechart embedded in a complex state.
The variables in the set B are mainly used to implement history. If a hierarchical state q is marked with history, its corresponding dummy state must be connected with all states in Qr. These transitions are triggered by b∈B according to the following transition function: τ(q, null, null, b=i-1)≡(q, null, b=i), ∀i∈[1,n] and the transition from q to the next state does not contain the assignment that reset b. For example, let us suppose that after visiting q2 (b=1), the automaton takes a transition that leaves state q. This transition increments the value of b. The symbol formalizing the value of b when an interleaved transition is taken is ⊙ (see Table 2, row 2, history). If, afterwards, the automaton takes a transition directed to q, this transition is treated as a transition directed to q3, since b=2.

AND-states are complex states representing parallelism. Without loss of generality, let us show how the PTA represents AND-states considering an AND-state composed by two automata A1 and A2, with the set of states \(Q = \{q_1, q_2, \ldots, q_n\}\) and \(Q' = \{q_1', q_2', \ldots, q_m'\}\) respectively. An AND-state is represented through a dummy state q connected with q1 and q1' through a universal branch [Chandra 1981, Vardi 1994]. This branch is taken under the condition that history variables b' and b'' associated with A1 and A2 are both equal.

### Table 3. Semantics of a BCC's transitions and hierarchical states expressed through a PTA

<table>
<thead>
<tr>
<th>Graphical Representation</th>
<th>Topic</th>
<th>Business Conversation Chart</th>
<th>Predicative Time-bounded Automaton</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transition labels</td>
<td></td>
<td>τ(q1, q2, q3, q4) = q5;</td>
<td>(q4, q5, q6) = q6;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>τ(q4, q5, q6, q7) = q8;</td>
<td>(q6, q7, q8) = q8;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>τ(q4, q5, q6, q7) = q8;</td>
<td>(q6, q7, q8) = q8;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>τ(q4, q5, q6, q7) = q8;</td>
<td>(q6, q7, q8) = q8;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>τ(q4, q5, q6, q7) = q8;</td>
<td>(q6, q7, q8) = q8;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>τ(q4, q5, q6, q7) = q8;</td>
<td>(q6, q7, q8) = q8;</td>
</tr>
</tbody>
</table>

### Table 4. Semantics of a BCC's And- and Or-states expressed through a PTA

<table>
<thead>
<tr>
<th>Graphical Representation</th>
<th>Topic</th>
<th>Business Conversation Chart</th>
<th>Predicative Time-bounded Automaton</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND-state</td>
<td></td>
<td>τ(q1, q2, q3) = q4;</td>
<td>(q1, q2, q3) = q4;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>τ(q1, q2, q3) = q4;</td>
<td>(q1, q2, q3) = q4;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>τ(q1, q2, q3) = q4;</td>
<td>(q1, q2, q3) = q4;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>τ(q1, q2, q3) = q4;</td>
<td>(q1, q2, q3) = q4;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>τ(q1, q2, q3) = q4;</td>
<td>(q1, q2, q3) = q4;</td>
</tr>
</tbody>
</table>

472
to zero. Moreover, $q''_m$ is connected with dummy state $q$ through a transition that has as assignment $b''=m$, while a new state $q''_{m+1}$ is added to $A_2$ and $q''_{m+1}$ is connected to it through a transition that contains the assignment $b''=m$. Then the transition from $q$ to the next state is augmented with the guard $b''=m \land b''=m$. In this way, when the universal branch is taken, there are as many copies of the PTA as the number of the automata composing the AND-state. Since one of the copy of the PTA forms a loop with $q$, when the sub-automaton that composes the loops finishes its computation, it is back in $q$. But, because of the guard of the transition to leave $q$, it needs that all the automata finish their computation to leave the AND-state.

The OR-state represents non-determinism. In order to explain its behavior, we use, without loss of generality, an OR-state composed by two sub-automata (see Table 5, row 2). The OR-state is represented with a new state $q$, that is connected with $q''_1$ and $q''_2$ through an existential branch [Chandra 1981, Vardi 1994]. The final states of all the automata are connected with $q$ and the transition to leave $q$ has as additional guard a condition that guarantee that at least one of the sub-automata that compose the OR-state is executed.

Finally, we need to define the semantics of interleaved transitions and self-loops directed to a hierarchical state. Interleaving is represented through a transition that takes into account that before leaving a sub-state, the corresponding history variable must be updated (see Table 5, row 1). On the other hand, a self-loop directed to a hierarchical state is modeled as a self-loop on either the first or the last sub-state according to the type of event. In particular, if the event is Begin($t$), the self-loop is mapped onto the first sub-state, while, if the event is End($t$), the self-loop is mapped onto the last sub-state. This rule is sound since transitions are always synchronous (see Table 5, row 2).

### 2.3 Translation into Promela

To verify PTA, that are the formal representation of BCC, we use the model checker SPIN [Holtzmann 2003], representing PTA in Promela, SPIN input language. Traditionally Promela represents Büchi automata, while PTAs allow also alternation; hence we provide the rules to simulate PTAs with Promela. Note that our PTAs are finite state machines that recognize finite strings. However, an extension to infinite strings is easily developable. This characteristic allows a simplification of the corresponding Promela code, as discussed in the following. The strings recognized by our automata are a function of external events (e.g. a violation of a resource condition). Thus, in order to validate an automaton, the set of external events is generated exhaustively. We use the following translation rules:

1. Each state of a PTA is mapped onto a communicating SPIN process. Each process can be active or idle (i.e. all processes are running, but we can identify active and idle processes with flags). Processes are connected through a communication net that simulates the transitions of the automaton. Hence, a PTA is a network of communicating active processes where only one process is allowed to be active, except for automata within an AND-state. Note that a theoretical model based on alternating states is necessary to simulate the behaviour of AND-states, without explicitly enumerating all the possible combinations of the sub-states of an AND-state.

The OR-state represents non-determinism. In order to explain its behavior, we use, without loss of generality, an OR-state composed by two sub-automata (see Table 5, row 2). The OR-state is represented with a dummy state $q$, that is connected with $q''_1$ and $q''_2$ through an existential branch [Chandra 1981, Vardi 1994]. The final states of all the automata are connected with $q$ and the transition to leave $q$ has as additional guard a condition that guarantee that at least one of the sub-automata that compose the OR-state is executed.

Finally, we need to define the semantics of interleaved transitions and self-loops directed to a hierarchical state. Interleaving is represented through a transition that takes into account that before leaving a sub-state, the corresponding history variable must be updated (see Table 5, row 1). On the other hand, a self-loop directed to a hierarchical state is modeled as a self-loop on either the first or the last sub-state according to the type of event. In particular, if the event is Begin($t$), the self-loop is mapped onto the first sub-state, while, if the event is End($t$), the self-loop is mapped onto the last sub-state. This rule is sound since transitions are always synchronous (see Table 5, row 2).

<table>
<thead>
<tr>
<th>Graphical Representation</th>
<th>Topic</th>
<th>Business Conversation Chart</th>
<th>Predicative Time-bounded Automation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interleaving</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Self-loops and hierarchical states</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2. A SPIN process is provided of a unique identification to guarantee a non ambiguous communication among processes. Each process is organized into different parts to simulate the different behaviors of the state depending on the transitions that activate it: (i) a main body representing the first transition into the state, corresponding to the standard behavior and (ii) a set of parts representing any other feedback or self-loop that formalizes the recovery behavior.

3. Whenever it is necessary to obtain data from the environment, a possible scenario is randomly generated for the evaluation of temporal, goal, resource, and compensation conditions.

4. A process becomes idle when it sends a message to the next working process. The next working process corresponds to the destination state of the triggered transition in the PTA.

5. Time is modeled as a global natural variable, since both the residence time of states and the time consumed by compensation actions are finite (see $\Sigma_\alpha, \Sigma_\omega$ in Sect. 2.2).

6. The residence time of a state is randomly generated according to the time requirements specified by $x \in X$. The global counter variable is updated accordingly. Random generation also considers the violation of a time requirement.

7. A set of local counter variables is implemented to model the residence time of a multiple-state (see Table 4). The residence time of a multiple-state is processed as the maximum residence time among all states. Finally, a condition checking the violation of the upper bound associated with a state residence time is implemented.

8. The time consumed by a compensation action is randomly generated within the destination state according to the time requirements specified by $x \in X$. The global counter variable is updated accordingly. Note that this strategy is consistent with the definition provided for transition function $\tau^\prime$. Finally, a condition checking the failure of a compensation action is also considered.

9. The failure of a compensation action can force the PTA into an abort state. This behavior is modeled as a SPIN process broadcasting a termination message to all processes belonging to the network. Then, the broadcaster terminates.

Notice that the mechanism explained at point 1 and 2 can be optimized using a single process, called main process, with internal labels to represent all the states that do not represent an and state, while all the sub-automata in a universal branch are represented with autonomous processes communicating with the main process using the mechanism explained at point 1.

3. EXEMPLIFICATION

This section presents a case study based on the Italian district of Matera. The case study involves a sofa producer and one of its supplier. Typically, sofa producers “buy” operating activities from business partners within the district and perform final assembly. Moreover, the sofa producer monitors part of the internal business process of its supplier requiring to be informed when violation occurs. In particular, the buyer monitors the cost of backbones and the frequency of errors during production. Moreover, the sofa producer requires a backbones strength (minimum weight tolerated) greater then 10 kg/m².

Basically, the business conversation is organized into three sequential macro-states: negotiation, execution and post-settlement [Colombo et al. 2004]. Execution is a sequence of two states corresponding to the “order scheduling” and the “component production” task, respectively. Post-settlement is organized as a sequence of three states: “backbone quality control”, an AND-state modeling parallel activities performed by both the buyer and the seller and a “delivery” task. In particular, after quality control, while the buyer executes the payment, the supplier stores backbones in a warehouse and plans their delivery. Payment is performed after quality control since a violation of the requirement on backbone strength can cause the failure of the conversation. Moreover, because of the type of machines involved in the production process, the supplier cannot produce backbones greater of a given size. In the following we provide a detailed discussion of the conversation modeled through a BCC.

Before performing the “order scheduling” task, the supplier controls its mandatory requirement on backbone size. If backbone size does not comply with the supplier’s requirements, the correct value of the parameter is urged through compensation action $\text{Urge}(0,3]$, buyer, order.size). At the end of the “order scheduling” task, the unitary cost of raw materials is checked and, if violated, the sofa producer (buyer) is notified.

At the end of the order scheduling task, the automaton should move to the next state even if the requirement on backbone cost is violated and the compensation action is not fulfilled. If cost is violated, the supplier must notify the violation as a first attempt to resolve the exception. Since the requirement on
backbone size is mandatory, the transition is enabled under the operating condition order.size ≤ 1.5 m² (we can always transform the relative numbers into natural numbers). This condition is formalized as follows:

\[
\text{End(Order Scheduling)} \left( [(\neg \text{cost} > 100 \text{ euro/unit}) \land \text{order.size} \leq 1.5 \text{ m}^2 )] \land \text{Notify(null, buyer, cost)} \right) \lor \\
\text{End(Order Scheduling)} \left( \text{Done(Notify(null, buyer, cost))} \land \text{order.size} \leq 1.5 \text{ m}^2 \right)
\]

Note that, in order to obtain a deterministic and finite evolution of the business conversation, the self-loop enabling the notification of the cost violation must embed the guard condition \( \neg \text{Done(Notify(<null, buyer, cost>))} \). Under this condition, modelers require that the self-loop enabling the notification of the cost violation is taken once and only once per business conversation.

The behavior of the model as a consequence of backbone quality control is formalized in the post-settlement phase. If the requirement on backbone strength is violated, the supplier must notify the violation and it can propose the negotiation of backbone cost in order to avoid the re-execution of component production. If, after the bargaining process, an agreement is not reached, the supplier must provide a new lot of backbones to the sofa producer thus re-executing component production. The corresponding ECA is:

\[
\text{End(Backbone quality control)} \left( \text{minimum weight} < 10 \text{ Kg/unit} \land \neg \text{Done(Re-execute task from(component production))} \right) \\
\text{Sequence(Notify(null, buyer, minimum weight) \land Relax([2,5], seller, cost)); Re-execute task from(component production))}
\]

Note that the “Relax” and “Re-execute task from” compensation actions belong to different compensation classes (see Table 1, Sect. 2.1). The “Relax” compensation must be performed during negotiation, while “Re-execute task from” is performed during execution. Accordingly, the corresponding ECA rule should be re-written into two rules: the first should be associated with the transition from post-settlement to negotiation, while the second should be associated with the transition from negotiation to execution. Therefore, the transition label from post-settlement to negotiation is formalized as follows:

\[
\text{End(Backbone quality control)} \left( \text{minimum weight} < 10 \text{ Kg/unit} \land \neg \text{Done(Re-execute task from(component production))} \right) \\
\text{Notify(null, buyer, minimum weight) \land Relax([2,5], buyer, cost)).}
\]

The transition label from negotiation to post-settlement is composed of two ECAs. The first ECA formalizes the behavior of the automaton if an agreement is reached. At the end of the negotiation phase, the transition to execution is therefore enabled if the “Relax” compensation action is fulfilled. The second ECA models the behavior of the automaton when the “Relax” compensation fails (\( \neg \text{Fulfilled(Reset([2,5], seller, cost))} \)) forcing the re-execution from component production.

\[
\text{End(Negotiation)} \left( \text{Fulfilled(Reset([2,5], buyer, cost)))} \right) \lor \\
\text{End(Negotiation)} \left( \neg \text{Fulfilled(Reset([2,5], buyer, cost)))} \right) \\
\text{Re-execute task from(component production))} \land \text{Reset_history(post-settlement))}
\]
We note that in this case, the use of history is critical. If the parties agree on the reduction of backbone cost, i.e. \text{Fulfilled}(\text{Relax}<[2,5], \text{buyer, cost}>), the business conversation must evolve into the “post-settlement” state. By marking both execution and post-settlement with history, the automaton enters the execution state and immediately leaves it because history points to the final sub-state within execution. Leaving execution, the automaton enters post-settlement, but, since the “backbone quality control” state has been already visited, history points to the following state, as required. On the other hand, marking post-settlement with history could generate a wrong behavior, since after the re-execution of “component production”, the automaton skips quality control. The effect would be that the sofa supplier receives the second lot regardless of its quality constraint and accepts to pay the full price. This behavior is corrected by specifying a “reset_history” action together with the “re-execution from component production”, as shown in Figure 1. Finally, if the “re-execution from component production” fails, the business conversation is aborted.

In the next section we verify our model of business conversation requirements testing several properties.

4. EXPERIMENTATION

This section discusses a few examples of properties that can be verified with our formal model. Properties are formalized by means of classical LTL operators [Pnueli 1981]. Two classes of properties can be verified on the BCC model formalized in Section 2:

1. \textit{Structural properties}, modeling the characteristics of a BCC. Examples of structural properties include termination, reachability of states, sequence/parallelism among tasks and actors taking responsibility for the execution of a task.


Structural and QoS properties can be either safety or liveness properties [Lamport 1987]. This section discusses a few examples of properties that can be verified with our formal model. These properties are formalized by means of classical LTL operators [Pnueli 1985].

A first verification of the model provided in Figure 1 discovered a deadlock, showing “order scheduling” as an invalid end state [Holtzmann 2003]. The analysis of the SPIN message sequence chart highlights that this deadlock condition is caused by a failure of the “Urge” compensation action. This anomalous behavior is corrected by modeling a transition from “order scheduling” to abort labeled as follows:

\{\text{Begin(Order Scheduling)|}¬\text{Fulfilled(Urge}<[0,3], \text{buyer, order.size}>))\}

In Table 6 we provide a set of properties verified by the BCC (we used a PC equipped with an Intel Pentium 1500 MHz and 256Mbyte of RAM). The formal verification guarantees that the formalization of our business conversation requirements does not present inconsistencies with modeler’s expectations.

<table>
<thead>
<tr>
<th>ID</th>
<th>Property</th>
<th>Formalization*</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>The business conversation must not embed a run requiring more than 90 days.</td>
<td>□ (time ≤ 90)</td>
</tr>
<tr>
<td>P2</td>
<td>At the end of the negotiation phase, the business conversation is aborted iff the corresponding order is not received in at most in three days.</td>
<td>□ [Agreed⇒¬(Received_order ⇔ Aborted)]</td>
</tr>
<tr>
<td>P3</td>
<td>Only exceptional flows may bring a conversation into an abort state.</td>
<td>□ [Aborted ⇒ (Size_violation ∨ Cost_violation ∨ Weight_violation)]</td>
</tr>
<tr>
<td>P4</td>
<td>The “order scheduling” task must be executed before production.</td>
<td>□ [(¬Component_production Until Order_scheduling) ∨ ¬Component_production]</td>
</tr>
<tr>
<td>P5</td>
<td>A conversation may commit even if an agreement different from the initial contract is reached.</td>
<td>□ (Commit ⇒ ¬Done_relax)**</td>
</tr>
<tr>
<td>P6</td>
<td>If a time-out is triggered, the conversation is always aborted.</td>
<td>□ (Time_out ⇒ ◊ Aborted)</td>
</tr>
<tr>
<td>P7</td>
<td>A violation of the weight parameter may be compensated through a delegated compensation.</td>
<td>□ (Weight_violation ∧ IsSeller ⇒ ◊ IsBuyer)</td>
</tr>
<tr>
<td>P8</td>
<td>During a conversation, the agreement may be renegotiated.</td>
<td>□ (¬Done_Re谈判) **</td>
</tr>
<tr>
<td>P9</td>
<td>The “order scheduling” task is executed only once.</td>
<td>□ (Order_scheduling ⇒ Order_scheduling Until (¬Order_scheduling))</td>
</tr>
</tbody>
</table>

* All formulas are written using future operators since the SPIN model checker does not support past.
** These properties are verified by generating a counter-example
For example, P1 formalizes a critical requirement since the production of backbones is embedded in a broader inter-organizational process and a delay could cause the failure of the overall schedule of activities. Moreover, the satisfaction of P3 guarantees that if the counterparts comply with their contract the conversation terminates with a commit. We note that even with a small example, automatic verification discovers unintuitive properties of the conversation. For example, at a first glance P6 seems true, after verification it results false since time-outs associated with relax compensation actions does not bring the conversation into abort. However, if we re-write this property excluding this particular condition, it results true as expected (see P6 revised in Table 6). Finally, P9 specifies that when the seller is forced to re-execute its components production task the re-execution of the “order scheduling” task is not necessary.

<table>
<thead>
<tr>
<th>ID</th>
<th>Result</th>
<th>States</th>
<th>Transitions</th>
<th>Memory (MBytes)</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>true</td>
<td>87.416</td>
<td>147.145</td>
<td>9,393</td>
<td>-</td>
</tr>
<tr>
<td>P2</td>
<td>true</td>
<td>87.840</td>
<td>147.625</td>
<td>9,496</td>
<td>-</td>
</tr>
<tr>
<td>P3</td>
<td>true</td>
<td>221.988</td>
<td>375.622</td>
<td>15,537</td>
<td>-</td>
</tr>
<tr>
<td>P4</td>
<td>true</td>
<td>87.420</td>
<td>147.149</td>
<td>9,496</td>
<td>-</td>
</tr>
<tr>
<td>P5</td>
<td>true*</td>
<td>107</td>
<td>107</td>
<td>5,502</td>
<td>246</td>
</tr>
<tr>
<td>P6</td>
<td>false</td>
<td>1.166</td>
<td>1.582</td>
<td>5,502</td>
<td>360</td>
</tr>
<tr>
<td>P6 revised</td>
<td>true</td>
<td>91.400</td>
<td>159.905</td>
<td>9,598</td>
<td>-</td>
</tr>
<tr>
<td>P7</td>
<td>true</td>
<td>102.526</td>
<td>181.139</td>
<td>10,011</td>
<td>-</td>
</tr>
<tr>
<td>P8</td>
<td>true*</td>
<td>66</td>
<td>66</td>
<td>5,502</td>
<td>155</td>
</tr>
<tr>
<td>P9</td>
<td>true</td>
<td>89.026</td>
<td>155.200</td>
<td>9.496</td>
<td>-</td>
</tr>
</tbody>
</table>

* These properties have been verified through counter-examples

Table 7 shows the benchmarks associated with the verification of the properties presented in Table 6. It summarizes the result of verification, the number of states and transitions of the Büchi automaton explored for the verification and the memory required. Moreover, when properties are verified through counterexamples, we also show the number of levels in the verification tree that have been explored by SPIN before the identification of the counterexample. Finally, we note that since our model of business conversation is time-bounded, the memory needed to process the verification of a property is very small and processing time is negligible (about .4 seconds). Property requirements verification is therefore fast and easy to schedule.

5. CONCLUSION AND FUTURE WORK

This paper provides an approach supporting the specification of business conversations during late-requirements analysis. The model in this work is described through the Promela language of the SPIN model checker in order to verify structural and QoS properties. In particular, it is discussed how our model of business conversation can be formally defined through a Predicative Time-bounded Automaton and how this automaton can be described with Promela. We note that the automatic verification of properties on a formal model of requirements is an important aspect of the requirements engineering process, since it guarantees that designers do not receive inconsistent specification that could result into an ill-designed software system. Moreover, our proposal allow an efficient verification of business conversation embedding parallel activities distributed among multiple actors since the underlying model then translated into Promela is exponentially more concise of a Büchi automaton.

However, note that the theoretical model and the corresponding Promela description proposed in this paper are suitable for the verification of properties of a single business conversation under the assumption that each conversation is independent from previous conversations. This assumption would not be suitable for monitoring systems, where compensation actions can be specified as a cumulated effect of events triggered in the past. Accordingly, a monitoring system would require an unbounded-time automaton. Finally, a tool supporting the automatic translation of a BCC into Promela is currently under development. The development of a tool is necessary to promote the use of formal methods among managers and business
domain analysts. We note that the implementation of a translator does not present particular difficulties since the translation from the BCC to Promela is linear in the size of the BCC.

REFERENCES


Holzmann, G.J. (2003), The SPIN model checker, Addison Wesley.


VISPO Project www.casaccia.enea.it/vispo