Specifying Business Processes with Azzurra

Fabiano Dalpiaz[1], Amit K. Chopra[2], Giulia Canobbio[3], Nicola Zeni[3], Paolo Giorgini[3], and John Mylopoulos[3]

[1] University of Toronto, Canada – dalpiaz@cs.toronto.edu
[2] Lancaster University, United Kingdom – a.chopra1@lancaster.ac.uk
[3] University of Trento, Italy – g.canobbio@gmail.com, {nicola.zeni,pgiorgio,jm}@disi.unitn.it

Abstract. A business process is above all else a social interaction among multiple participants. Business process modeling languages support the description of business processes in operational terms as collections of interleaved activities conducted by human and software agents. However, such descriptions do not capture adequately the richness of social interaction among participants. To address this deficiency, we propose Azzurra, a specification language for modeling and executing business processes. Azzurra is founded on social concepts, such as roles, agents and commitments among them, and Azzurra specifications are social models consisting of sets of commitments. As such, Azzurra specifications support flexible executions of business processes, and provide a semantic notion of actor accountability and business process compliance. In this paper, we present syntax and semantics of Azzurra, and we propose algorithms to determine runtime compliance with an Azzurra social specification.

Keywords: social specifications; business processes; compliance

1 Introduction

Specification languages enable describing systems at an abstract level that hides away implementation details. As such, they have been found very useful for building early models of a system that are readily analyzable to determine its properties before it has actually been built. Unsurprisingly, there are dozens of specification languages for software (e.g., Z and VDM), hardware, concurrent processes, interfaces, and more.

We are interested in developing a *specification language for business processes*. As such, our language should abstract away implementation details as well as support the creation of *social models*, since business processes are social phenomena consisting of social interactions among multiple participants. For example, an order fulfillment business process is enacted through the social interactions among supply order managers, client liaisons, customers, warehousemen, shippers, etc.

Existing business process modeling languages (e.g., BPMN, BPEL, workflow nets, π-calculus) describe processes in terms of interleaved activities conducted by human and software agents. These languages are procedural in that they prescribe the execution of activities over time in accordance with a rigid control-flow. van der Aalst rightly observes [20] that it is not easy to specify flexible workflows in these languages. While

one can model explicitly all the variants of the process, the specification becomes unwieldy and unmanageable, too complex to be extensible or even comprehensible [7, 14]. Declarative workflow languages go a step forward by expressing only the essential temporal precedence constraints between activities [20].

Our diagnosis for the flexibility issue is that business process modeling languages are grounded on computer system process concepts, rather than social ones. Thus, they require the wrong kind of detail by focusing on *how* a business process is to be enacted, rather than *what* it is intended to achieve and *who* is accountable for it.

With an eye on hiding away implementation details to enable flexible executions of processes, this paper goes beyond declarative workflows. We adopt the notion of *social commitments* [16] among actors in a business process as the fundamental business process abstraction. Commitments, like expressions in temporal logic, are declarative. However, unlike expressions in temporal logic, commitments are also a *high-level social abstraction* that the participants in a business process manipulate. Commitments explicitly capture the social responsibilities of actors towards each other.

Building on top of commitments [16] and commitment protocols [22], we propose Azzurra, a specification language for business processes that relies upon *social primitives*. Our contributions over the literature (more details in Section 6) are as follows:

- We propose an expressive language for specifying business processes as commitment protocols. The language includes business primitives such as delegations, deadlines, and constraints such as for role adoption. Azzurra defines the notion of initiation and termination of a protocol, and it supports protocol cross-references.
- We introduce a graphical notation to support designers in modeling the main elements in a business process specified in Azzurra. This notation is supported by a prototype modeling tool built on top of Eclipse.
- We provide algorithms to determine whether a set of observed events complies with an Azzurra protocol specification. The implementation of these algorithms is in Java and in the Drools rule engine language. Noncompliance can be dealt with by an *enactment engine* that is able to enact compensation tactics.

The rest of the paper is structured as follows. Section 2 reviews related work. Section 3 presents our research baseline. Section 4 presents syntax and semantics of the Azzurra language. Section 5 presents algorithms to detect compliance with an Azzurra specification. Section 6 discusses the distinguishing features of Azzurra for business process specification. Section 7 presents conclusions and future work.

2 Related work

We review the literature in cooperative work, business process modeling, commitment protocols, choreographies and service-orientation, business artifacts, and compliance with obligations. Due to space limitations, we discuss few examples per category.

Cooperative work. Commitments are a main abstraction in the influential *Coordinator* [8] system for tracking activities in an organization. This work inspires our approach. However, as observed by Singh [17], the Coordinator takes a procedural approach to model conversations, which is less flexible than commitment protocols.

Business process modeling. Most approaches in this category relate a set of activities through a control flow (e.g., BPMN, BPEL, workflow nets). These languages have a different purpose from Azzurra: they define operational details to support process execution, as opposed to specifying a process. Declarative workflows [20] are less rigid, for they rely on precedence constraints. These models emphasize the activities to carry out, and treat actors responsibility as a secondary concept. On the other hand, commitments are contractual relationships between couples of actors.

Commitment protocols. Desai et al. [6] use commitments and protocols as design abstractions for business processes. Such work inspires the REGULA framework [12], which introduces temporal operators to represent more expressive commitments and reasoning about them. Robinson and Purao [13] propose a framework for specifying and monitoring cross-organizational business processes that relies upon commitments enriched with a rich temporal logics. Azzurra's novelty includes: (i) advanced primitives for expressing business patterns such as separation of duties, compensations, workload limits, as well as supporting the life-cycle of protocol instances, from initiation to termination; (ii) a graphical notation to visualize the main elements of a protocol; and (iii) algorithms to determine compliance of observed behavior with a specification.

Choreographies and service orientation. Choreographies specify the flow of messages among autonomous actors. van der Aalst [19] advocates choreographies for modeling cross-organizational business processes. Khalaf [11] shows how to map the RosettaNet PIPs business protocols to abstract BPEL processes. Choreographies are used to model the interactions among web services. Decker et al. [4] extend BPEL with choreography-related constructs. WS-CDL [21] and BPMN 2.0 both support the specification of choreographies. Benatallah et al. [1] propose a transition-based conversation model to conceptualize web service conversations. Unlike choreographies, Azzurra captures the meaning of interaction as commitments among actors.

Business artifacts. Bhattacharya et al. [2] take artifacts such as purchase orders, invoices, and so on, as the center of their universe and define workflows that essentially represent the lifecycle of these artifacts. Modeling with commitments is orthogonal, as it stresses the accountability of participants. Keller et al.'s [10] event-driven business chains are an alternative to activity-based processes: they are centered on events that trigger functions performed by organizational units. Our approach considers high-level events that update the commitments of actors. An interesting future direction is to investigate the joint usage of commitments and business artifacts.

Compliance with obligations. Ghose and Koliadis [9] annotate business processes with constraints about their execution (e.g., to represent normative compliance). They observe that business process models focus on the syntax, and not on the meaning. Sadiq et al. [15] enrich business process models with obligations that an enterprise must fulfill in order to remain compliant. While a commitment is a social relation between actors, these types of obligations are technical constraints on information system design.

3 Baseline: commitments, protocols, and roles

A social commitment [16], formally $\mathsf{c}(\mathsf{x},\mathsf{y},\mathsf{p},\mathsf{q})$, is a promise with contractual validity made by an agent x (debtor) to another agent y (creditor) that, if proposition p is brought

about (antecedent), then proposition q will be brought about (consequent). If p is equal to truth ($\top$), the commitment is unconditional; otherwise, it is conditional.

Commitments are social abstractions, as they carry a social meaning (they are contracts), and they evolve independently of the internal design of communicating agents. Commitments carry responsibility on part of the agent who makes the commitment.

Commitments evolve on the basis of the interactions among agents through message exchange. Messages have a meaning in terms of *commitments operations*: (i) *creation*: the debtor commits to the creditor that the consequent will be brought about; (ii) *cancellation*: the debtor cancels a previously made commitment (the creditor is informed of a commitment violation); (iii) *release*: the creditor releases the debtor from a previous commitment; (iv) *delegation*: the debtor delegates the commitment to a third party; and (v) *assignment*: the creditor assigns its credit to another actor.

Moreover, *declare* operations let an agent inform another that a certain proposition has been brought about (e.g., the book has been sent). *Declare* operations enable the evolution of commitments. A commitment is *detached* when the debtor is informed (through a *declare*) that the antecedent has been brought about, and the commitment becomes unconditional. A commitment is discharged/fulfilled, when the creditor is informed that the consequent has been brought about.

Figure 1 shows how the state of a commitment evolves depending on performed operations. The statechart includes additional concepts that are part of Azzurra: (i) timeout: if expired, the commitment is violated by the debtor; (ii) commitment manipulation operations (cancel, release, delegate, assign) may result in violations, depending on the permissions of creditor and debtor; and (iii) delegation and assignment come in two versions, depending on the retainment or transfer of responsibility and credit.

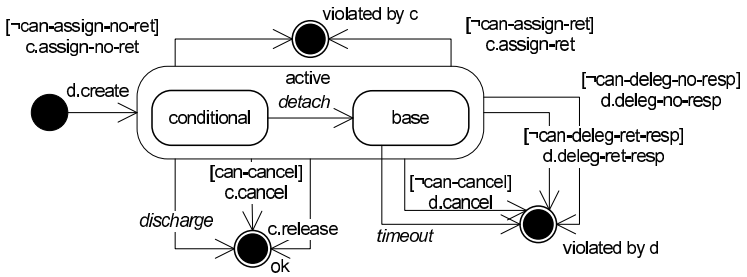

Fig. 1. Runtime semantics for a commitment instance made by debtor d to creditor c

We adopt a version of commitments [12] where antecedent and consequent are expressed in propositional logic extended with a temporal precedence operator "·". Thus, $(p \wedge q) \cdot r$ means that p and q occur (in any order) before r occurs.

Commitments can be abstracted to the class level to define an interaction (business) *protocol* between roles [3, 5]. For instance, given roles R_1 and R_2, a protocol may include a commitment class such as $C(R_1, R_2, P, Q)$. Thus, an agent playing role R_1 is expected to create instances of this commitment (to agents playing R_2). The propo-

sitions in such commitment will be instantiated too: if P is "Book sent", a possible instance p is "copy 123 of book Dracula sent".

4 Syntax and semantics of Azzurra

We present the EBNF syntax of Azzurra and its runtime semantics. The syntax is presented in Table 1 and illustrated in Table 2 through the fracture treatment scenario from the literature [20]. Fig 2 shows a graphical notation for visualizing the main elements of an Azzurra specification. The semantics is explained textually while describing the EBNF syntax.

Notational conventions. We denote classes—roles, the commitment class symbol, protocol and commitment identifiers, propositions—via strings with a leading capital letter, and instances—agents, commitment instances, the commitment instance symbol, proposition instances—via strings with a leading lowercase letter.

Example 1 (Fracture treatment). The patient is initially examined by a specialist. In case of a fracture, x-rays are requested. Specific treatments are applied depending on the diagnosis: sling, fixation, surgery, and cast. The handling doctor is responsible for choosing. Cast and fixation are mutually exclusive. If no fracture is found, a sling has to be made. Patients who undergo surgery are advised to execute rehabilitation. Medications can be provided and additional x-rays can be performed if needed. □

Protocol signature (1,3). A protocol (1) has an identifier p_{id} and a set of parameters (3): a "key" variable that is the unique identifier for the instances of that protocol, and a set of agent variables (two or more) associated with specific roles. Protocol designers are responsible for choosing a meaningful key for the protocol. The agent variables indicate those agents that play certain roles when a protocol is instantiated. The semantics of protocol instantiation is explained later in this section.
Example. In the treatment protocol in Table 2, the protocol name is Treatment, the key is the hospitalization number hospnr, the agent variables are patient p and specialist sp.

Protocol body (2). It includes a set of typed agent variables (their type is a role), a set of commitment classes, a set of protocol refinements (optional), and a knowledge base that defines semantic relations between atomic propositions (optional).
Example. In Table 2, there are five agent variables, including rc (a rehab centre) and ra (a radiologist), nine commitments (C_1–C_9), and two commitment refinements.

Commitments (5,6). The core of a protocol (5) includes commitment classes and their refinements. A commitment in Azzurra (6) extends the semantics presented in our baseline in different ways. First, we introduce the notion of a strong commitment (C^*), where the debtor commits to bring about the consequent only after the antecedent has occurred. Second, given that commitments belong in a specific Azzurra protocol, every state of affairs appearing in the antecedent and consequent of a commitment (e.g., Examined, Diagnosed) has an implicit parameter, i.e., the key of the protocol. This parameter enables relating the commitments instances that apply to the same protocol instance (e.g., examined(121) and diagnosed(234) refer to two different protocol instances, each concerning a specific patient hospitalization). Third, Azzurra enriches the

Table 1. EBNF syntax of Azzurra; terminals in bold, non-terminals in italics

$$
\begin{array}{rll}
prot \rightarrow & \textbf{protocol}\ p_{id}\ \textbf{(}params\textbf{)}\ \{ & (1)\\
& \quad [\textbf{ag-variables:}\ vars] & \\
& \quad \textbf{commitments:}\ comms\ crefn^{*} & \\
& \quad [\textbf{refinements:}\ (id : refn)^{*}] & \\
& \quad [\textbf{kb:}\ domain^{+}]\ \} & (2)\\
params \rightarrow & \textbf{key}\ v\textbf{,}\ v\ \textbf{:}\ role\ (\textbf{,}\ v\ \textbf{:}\ role)^{+} & (3)\\
vars \rightarrow & v\ \textbf{:}\ role\ (\textbf{,}\ v\ \textbf{:}\ role)^{*}\textbf{;} & (4)\\
comms \rightarrow & (\textbf{init}\ \twoheadrightarrow [_{\leq time}]\ comm\ \textbf{final;})^{+}\ (ev\ [[prec]] \twoheadrightarrow [_{\leq time}]\ comm\textbf{;})^{*} & (5)\\
comm \rightarrow & id : \textbf{C[*]}(v, v, prop, prop) & (6)\\
crefn \rightarrow & \textbf{deadline}(id, time)\ |\ \textbf{can-deleg-ret-resp}(id)\ |\ \textbf{can-deleg-no-resp}(id)\ | & \\
& \textbf{can-assign-ret-cred}(id)\ |\ \textbf{can-assign-no-cred}(id)\ |\ \textbf{can-cancel}(id) & (7)\\
refn \rightarrow & \textbf{max-per-role}(role, nr)\ |\ \textbf{max-of-class}(role, id, nr)\ |\ \textbf{role-confl}(role, role) & \\
& \textbf{comm-role-confl}(role, id, id)\ |\ \textbf{sep-duties}(id, id)\ | & (8)\\
prec \rightarrow & atom\ |\ cstate\ |\ pstate\ |\ prec\ op\ prec\ |\ \neg prec\ |\ (prec) & (9)\\
prop \rightarrow & atom\ |\ cstate\ |\ pstate\ |\ prop\ op\ prop\ |\ (prop) & (10)\\
op \rightarrow & \wedge\ |\ \vee\ |\ \oplus\ |\ \cdot & (11)\\
cstate \rightarrow & \textbf{create}(id)\ |\ \textbf{deleg-no-resp}(id\ [\textbf{to}\ v])\ |\ \textbf{deleg-ret-resp}(id\ [\textbf{to}\ v])\ |\ \textbf{fulfil}(id)\ | & \\
& \textbf{cancel}(id)\ |\ \textbf{expire}(id)\ |\ \textbf{release}(id)\ |\ \textbf{assign-ret-cred}(id\ [\textbf{to}\ v])\ | & \\
& \textbf{assign-no-cred}(id\ [\textbf{to}\ v]) & (12)\\
pstate \rightarrow & \textbf{init-p}(p_{id}\ (\textbf{,}\ v = v)^{*})\ |\ \textbf{fulfil-p}(p_{id}\ (\textbf{,}\ v = v)^{*}) & (13)\\
ev \rightarrow & \textbf{init}\ |\ atom\ |\ cstate\ |\ pstate & (14)\\
atom \rightarrow & \top\ |\ \bot\ |\ staffairs\ [(v\ (\textbf{,}\ v)^{*})] & (15)\\
domain \rightarrow & \textbf{implies}(staffairs\textbf{,}\ staffairs)\ |\ \textbf{mut-excl}(staffairs(\textbf{,}\ staffairs)^{+}) & (16)
\end{array}
$$

syntax of commitments with triggers and creation deadlines. A trigger—the expression before the $\twoheadrightarrow$ symbol—is an event that determines that a commitment shall be created. Triggers may have an associated precondition—$[prec]$ in (5)—that indicates that, when the event occurs, the commitment shall be created only if the precondition evaluates to true. A deadline ($\leq time$) specifies that the commitment has to be created within a certain amount of time after the trigger event fires off. Finally, Azzurra supports two special types of commitments that relate to protocol instantiation and termination:

- *Initial commitments* are created when a protocol is instantiated. Their trigger is "init", an event that occurs when a protocol is instantiated. Debtor and creditor of initial commitments shall be agent variables in the parameters of the protocol. This way, initial commitments are created between couples of agents (debtor and creditor do not refer to unassigned agent variables).
- *Final commitments*: every protocol must contain at least one final commitment. A protocol instance terminates successfully when any of the final commitments is fulfilled, while it terminates unsuccessfully if all final commitments are violated (e.g., cancelled by the debtor). Final commitments are also initial. When a protocol terminates, all debtors of active commitments are released from their responsibility.

The agent variables corresponding to debtor and creditor prescribe that:

Table 2. Azzurra protocol for the fracture treatment scenario

protocol Treatment (**key** hospnr, p : Patient, sp : Specialist) {
ag-variables: rc : RehabCentre, ra : Radiologist, or : Orthopedist, su : Surgeon, nu : Nurse;
commitments:
init $\twoheadrightarrow$ C_1 : C(sp, p, $\top$, Examined · Diagnosed · Dehospitalized) final
NoXRayNeeded $\twoheadrightarrow$ C_2 : C(or, sp, $\top$, SlingMade)
XRayRequested $\twoheadrightarrow$ C_3 : C(ra, sp, $\top$, XRayPerformed)
XRayRequested $\twoheadrightarrow$ C_4 : C^*(sp, ra, XRayPerformed, FractAssessed)
FractAssessed $\twoheadrightarrow$ C_5 : C(or, sp, $\top$, ((Fixated$\oplus$Plastered) $\vee$ fulfil(C_6) $\vee$ SlingMade))
FractAssessed $\twoheadrightarrow_{\leq 2h}$ C_6 : C^*(su, or, SurgeryRequested, Operated)
Operated [$\neg$Refused] $\twoheadrightarrow$ C_7 : C(nu, p, $\top$, RcChosen(rc))
RcChosen(rc) $\twoheadrightarrow$ C_8 : C(rc, p, $\top$, fulfil-p(RehabGiven, key=hospnr, pat-id=p, ref-sp=sp))
MedPrescribed(m) $\twoheadrightarrow$ C_9 : C(nu, sp, $\top$, MedApplied(m))
can-deleg-no-resp(C_3)
deadline(C_2, 2h)
refinements:
role-confl(Radiologist,Orthopedist)
kb:
implies(XRayRequested, Diagnosed)
implies(NoXRayNeeded, Diagnosed)
implies(MedPrescribed(m), Diagnosed)
mutExcl(XRayRequested, NoXRayNeeded) }

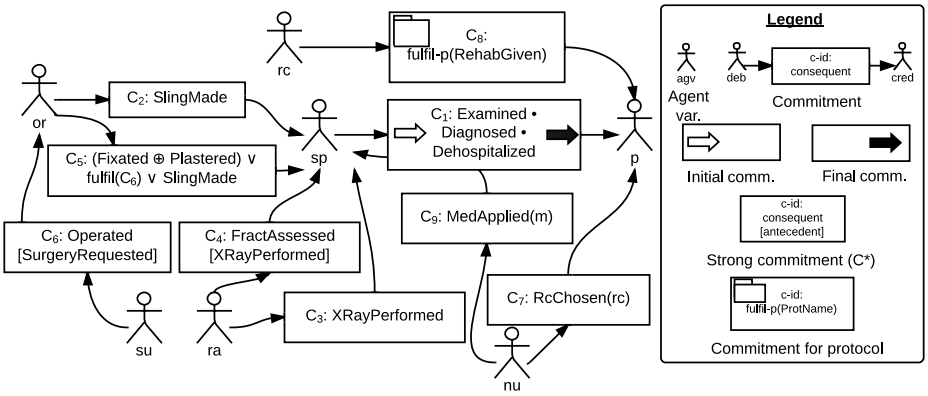

Fig. 2. Graphical representation for the protocol in Table 2

- if an agent a is assigned to the agent variable, a shall be debtor (or creditor);
- if the agent variable is unassigned, any agent a' can be debtor (or creditor), and a' is assigned to the agent variable by participating in the commitment.

Example. In Table 2, C_1 is the only final commitment and the only initial commitment. The protocol has two agent variable parameters (p and sp), which are the debtor and the creditor of C_1. When an instance of the protocol is created, with agent frank assigned to sp and agent mel assigned to p, an instance c_1 of C_1 shall be created with debtor frank and creditor mel. When c_1 is fulfilled (the patient is examined, then diagnosed, and

finally dehospitalized), the protocol instance terminates successfully. If c_1 is violated, the protocol terminates unsuccessfully. The triggered commitment C_2 is instantiated only if x-rays are not needed, and it specifies that an orthopedist has to commit to sp to make a sling. C_4 shows strong commitments: a specialist commits to assess the fracture only after x-rays have been performed.

Agent variables (2,4). We support agent variables that are unassigned when the protocol is instantiated. They are assigned when an instance of a commitment where they appear is created, and, as an additional effect, the assigned agent adopts the specified role in the protocol instance. Azzurra supports assign-once variables: once an agent is assigned, no other agent can be assigned to that variable.
Example. In Table 2, agent variables exists for a rehab centre, a radiologist, an orthopedist, a surgeon, and a nurse. Actual agents will be assigned to these variables as the protocol evolves, i.e., when commitments are created. For example, an orthopedist will be assigned to or as soon as she creates an instance of C_2.

Commitment refinements (7). A deadline commits the debtor to bring about the consequent within a certain time after the antecedent occurs. The debtor can be authorized to delegate the commitment, either retaining (can-deleg-ret-resp) or releasing (can-deleg-no-resp) her responsibility. The creditor, similarly, can be authorized to assign the commitment, either retaining (can-assign-ret-cred) or releasing (can-assign-no-cred) her credit. The debtor can be authorized to cancel her commitment (can-cancel).
Example. In Table 2, the radiologist can delegate instances of C_3, possibly to a colleague, without retaining responsibility. Without such authorization, delegations would correspond to a violation on part of the radiologist.

Protocol refinements (8). They constrain the agents that participate in a protocol instance. The maximum number of concurrent commitments for an agent playing a certain role can be limited (max-per-role), as well as the number of instances of a commitment class that an agent can make (max-of-class). Role conflicts (role-confl) prescribe that an agent cannot play two roles in the same protocol instance. Separation of duties (sep-duties) implies that an agent cannot be debtor in instances of two commitment classes, and it can be restricted to agents playing a specific role (comm-role-confl).
Example. A role-confl refinement specifies that the same agent cannot play both radiologist and orthopedist, because their roles are incompatible.

Preconditions, propositions, and triggers (9–15). Azzurra supports different types of preconditions (9) and propositions types (10): atomic (atom), commitment states (cstate), protocol states (pstate), binary operators, and so on. Negations can be used in preconditions only. If used as consequent in a commitment, the commitment would be that "an event will never occur"; at any point in time, one cannot claim that such commitment is fulfilled, because the event may occur in the future. The binary operators (11) are conjunction ($\wedge$), disjunction ($\vee$), exclusive disjunction ($\oplus$), and temporal precedence ($\cdot$). Atomic propositions (15) can be truth ($\top$), falsity ($\bot$), or states of affairs (e.g. FractAssessed). States of affairs may be parametric and, thus, have multiple instances. For example, MedPrescribed(med-id) has an instance for each medication the patient is given. The state of a protocol instance evolves because of the occurrence of

events (14), as they trigger new commitment instances and change the state of existing commitment instances. Three event types are supported:

- An atomic proposition becomes true. This includes the occurrence of a state of affairs (e.g., the patient is diagnosed).
- The state of a commitment instance changes (see clause (12) below).
- The state of another protocol instance changes, i.e., it is instantiated (init-p) or fulfilled (fulfil-p). Optionally, one can specify constraints on the protocol instance parameters, e.g., to impose a certain key or that a specific agent in the current protocol instance shall be assigned to an agent parameter in the referenced protocol.

Example. The consequent of commitment C_5 tells that the commitment is fulfilled if an instance of Fixated or Plastered occurs (but not both), an instance of C_6 is fulfilled, or an instance of SlingMade occurs. The consequent of C_8 indicates that a successful instance of the protocol RehabGiven is expected, with the constraints that the patient identifier parameter (pat-id) corresponds to the patient in the instance of Treatment, and that the reference specialist (ref-sp) corresponds to the specialist who is responsible for the current patient hospitalization.

Commitment states (12). Propositions and may denote that a commitment is in or has changed to a specific state (as described in Figure 1). Given a commitment class id:

- create(id): an instance of id is created;
- deleg-no-resp(id [to v]): an instance of id is delegated (to agent v) without retaining responsibility;
- deleg-ret-resp(id [to v]): an instance of id is delegated (to v); the delegator keeps responsibility;
- fulfil(id): an instance of id is fulfilled;
- cancel(id): an instance of id is canceled;
- expire(id): an instance of id has expired;
- release(id): an instance of id is released;
- assign-ret-cred(id [to v]): an instance of id is assigned (to v) retaining the credit;
- assign-no-cred(id [to v]): id is assigned, but the assignor does not retain the credit.

Knowledge base (16). It specifies semantic relationships, i.e., implications and mutual exclusions, between states of affairs. These relationships belong to the shared vocabulary of the participants in a protocol.
Example. Three states of affairs imply a diagnosis: XRayRequested, NoXRayNeeded, and MedPrescribed. XRayRequested is mutually exclusive with NoXRayNeeded.

5 Runtime compliance with Azzurra protocols

The semantics of Azzurra specifications enables determining whether the actors participating in a *protocol instance* are compliant with the specification. We assume that the messages that the actors exchange within the context of protocol execution are observable by a monitoring infrastructure. Compliance checking compares the *observed* behavior from occurred events and the *expected* behavior as indicated by the protocol specifications. We present two algorithms that enable determining compliance:

1. Algorithm 1 (EnactProtocols) determines how an event updates the state of existing protocol instances and of the commitment instances therein. We call this activity *enactment* of a protocol. The output constitutes the *expected* behavior.
2. Algorithm 2 (CheckCompliance) checks whether an occurred event violates the specification of a protocol instance. This corresponds to verifying if expected commitments are not created/fulfilled, if disallowed commitment operations are performed, and if protocol constraints (e.g., maximum roles per agent) are violated.

In our algorithms, we assume that the occurring events are associated with a specific protocol instance (there is no ambiguity about which protocol instance they refer to). Events are processed sequentially by dequeuing a first-in first-out queue of events. When the algorithms invoke the ENQUEUE function, an event is added to such queue.

Algorithm 1 Enacting protocol instances based on an occurred event

EnactProtocols(Event ev, ProtInst $[\,]\,\mathcal{P}$)

1 **if** $ev = \textit{init-p}(\textit{p-id}, \textit{key}, \textit{par}_1 = \textit{ag}_1, \ldots, \textit{par}_n = \textit{ag}_n)$
2 **then** $p \leftarrow$ CreateProtInstance($\textit{p-id}, \textit{key}, \textit{ag}_1, \ldots, \textit{ag}_n$)
3 $\mathcal{P}$.add(p)
4 **for each** $\textit{init} \twoheadrightarrow_{\leq t} C_i : C(Db, Cd, Ant, Cons) \in p.spec$
5 **do** p.addCommI(C_i, p.valOf(Db), p.valOf(Cd), $Ant, Cons$, now $+ t$, nil)
6 ProtInst $p \leftarrow$ getProtInstForEvent(ev)
7 **if** $Ev[Prec] \twoheadrightarrow_{\leq t} C_i : C(Db, Cd, Ant, Cons) \in p.spec \ \wedge \ p.kb \vdash prec(p.key)$
8 **then** p.addCommI(C_i, p.valOf(Db), p.valOf(Cd), $Ant, Cons$, now $+ t, ev.args$)
9 **if** $ev = \textit{create}(db, cd, c)$
10 **then** p.addCommInstance(c)
11 CommClass $cc \leftarrow p$.classOf(c)
12 **if** p.valOf($cc.deb$) $=$ nil **then** p.assign($cc.deb, db$)
13 **if** p.valOf($cc.cred$) $=$ nil **then** p.assign($cc.cred, cd$)
14 **if** $ev = \textit{cancel}(db, cd, c) \ \wedge \ c \in p$ **then** p.remove(c)
15 **if** $ev = \textit{release}(cd, db, c) \ \wedge \ c \in p$ **then** p.remove(c)
16 **if** $ev = \textit{deleg-no-resp}(db, db_2, c) \ \wedge \ c \in p$ **then** $c.db \leftarrow db_2$
17 **if** $ev = \textit{deleg-ret-resp}(db, db_2, c) \ \wedge \ c \in p$
18 **then** p.addCommI($c.id, db_2, c.cd, c.ant, c.cons, c.args$)
19 **if** $ev = \textit{assign-no-cred}(cd, cd_2, c) \ \wedge \ c \in p$ **then** $cj.cd \leftarrow cd_2$
20 **if** $ev = \textit{assign-ret-cred}(cd, cd_2, c) \ \wedge \ c \in p$
21 **then** p.addCommI($c.id, c.db, cd_2, c.ant, c.cons, c.args$)
22 **for each** CommInst $c \in p$
23 **do** $c =$ residuateAntCons(c, ev)
24 **if** $c.state =$ `fulfilled` **then** enqueue($\textit{fulfill}(c)$)
25 **if** $\exists c \in p.\textit{finalcomminsts}$ s.t. $\textit{fulfill}(c) \in evts$
26 **then for each** CommInst $cj \in p$ **do** release($cj.db, cj.id$)
27 $p.state \leftarrow$ `fulfilled`
28 enqueue($\textit{fulfill-p}(p.id, p.key)$)
29 **if** $\forall c \in p.\textit{finalcomminsts} \ . \ c.state =$ `violated`
30 **then for each** CommInst $cj \in p$ **do** release($cj.db, cj.id$)
31 $p.state \leftarrow$ `failed`
32 enqueue($\textit{failure-p}(p.id, p.key)$)
33 enqueueAll(getImpliedFrom($p.spec, ev$))

Algorithm 1 enacts a set of protocol instances and the commitments therein contained. The algorithm depends on the type of the processed event *ev*:

- *Protocol instantiation* (lines 1–5): a new protocol instance is created (class, key, and arguments are taken from the event), and added to the protocol instances $\mathcal{P}$ (lines 1–3). An instance of every initial commitment in the protocol specification is created (lines 4–5): debtor and creditor are set by retrieving the agents that are assigned to the agent variables in the commitment class. If specified, a creation deadline is set by adding the creation timeout to the current time (NOW). The following events types refer to the protocol instance that relates to the event (line 6).
- *Commitment trigger* (lines 7–8): commitment instances are created whenever an instance of the trigger event occurs, if the optional precondition holds (the knowledge base of the protocol instance, inferred from all occurred events, entails it).
- *Commitment creation* (lines 9–13): the commitment instance is added to the protocol instance (line 9). If the agent variables for the debtor and the creditor of the corresponding commitment class are still unassigned, their value is assigned to the debtor and the creditor of the commitment instance (lines 11–13).
- *Commitment updates* (lines 14–21): *cancel* and *release* operations imply the removal of the commitment instance from the protocol (lines 14–15). Delegation without retaining responsibility transfers the responsibility to another debtor (line 16), while delegation with retaining responsibility creates a second commitment instance (lines 17–18). Assignments of credit are treated similarly (lines 19–21).

The antecedent and consequent of commitment instances in the protocol instance are residuated [18]: they are updated based on the fact that a new event has occurred (lines 22–23). If a commitment's consequent is '$p \cdot q$', and event 'p' occurs, the consequent becomes 'q'. If 'q' occurs before 'p', the status of the commitment switches to violated. If a commitment is fulfilled, a corresponding event is enqueued.

Lines 25–32 handle protocol termination. Success (lines 25–28) occurs if a final commitment is fulfilled: active commitment instances are released, the protocol instance is set to fulfilled, and a corresponding event is enqueued. Failure (lines 29–32) occurs if all final commitment instances are violated. Finally, all the events that are implied from the *ev* via implies relationships are enqueued for processing (line 33).

Algorithm 2 raises errors whenever an event violates a constraint in the specification of a protocol instance. Line 2 handles mutual exclusion constraints (mut-excl): if the occurred event happens, and the knowledge base entails a conflicting state of affairs, an error is raised. Lines 3–14 examine all commitment instances, and raise errors when different commitment constraints are violated: expired creation deadline ($\leq t$), expired fulfillment deadline (deadline), disallowed delegation with retained responsibility (deleg-ret-resp), disallowed delegation without responsibility (deleg-no-resp), disallowed assignment retaining credit (assign-ret-cred), disallowed assignment without credit retainment (assign-no-cred), and disallowed cancellation (cancel). Lines 11–14 raise errors if the event is the creation of a commitment, but the debtor or the creditor is not the expected one. For instance, if a commitment class has debtor agent variable agv_1, agent "john" is already assigned to agv_1, and a commitment instance for that class is created with debtor "mike", an error is raised. Lines 15–29 detect violations of protocol refinement constraints, such as max-per-role and sep-duties.

Algorithm 2 Checking compliance with a protocol instance

```
CHECKCOMPLIANCE(Event ev, ProtInst p)
 1  ProtSpec sp ← p.spec
 2  if mut-excl(St_i, Ev) ∈ sp ∧ st_i(p.key) ∈ p.kb then ERROR(p, mut-excl(st_i, ev))
 3  for each CommInst c ∈ p
 4  do if NOW > c.creatDeadline ∧ !c.created then ERROR(p, create-timeout(c))
 5     if NOW > c.fulfilDeadline ∧ !c.fulfilled then ERROR(p, fulfill-timeout(c))
 6     if ev = deleg-ret-resp(db_1, db_2, c) ∧ !c.canDelRet then ERROR(p, del-ret(c))
 7     if ev = deleg-no-resp(db_1, db_2, c) ∧ !c.canDelNoR then ERROR(p, del-no-r(c))
 8     if ev = assign-ret-cred(cd_1, cd_2, c) ∧ !c.canAssgnR then ERROR(p, assign-ret(c))
 9     if ev = assign-no-cred(cd_1, cd_2, c) ∧ !c.canAssgnNoC then ERROR(p, assign-no-r(c))
10     if ev = cancel(db, c) ∧ !c.canCancel then ERROR(p, cancel(c))
11     if ev = create(db, cd, c)
12       then CommClass cc ← p.CLASSOF(c)
13            if NIL ≠ p.VALOF(cc.deb) ≠ db then ERROR(p, wrong-deb(c, db))
14            if NIL ≠ p.VALOF(cc.cred) ≠ cd then ERROR(p, wrong-cred(c, cd))
15 for each AgentVar agv ∈ p.agent-vars
16 do Role rl ← agv.role
17    Agent ag ← p.GETASSIGNEDAGENT(agv)
18    if ag = NIL then break
19    if max-per-role(rl, n) ∈ sp ∧ |p.COMMWITHDEB(ag)| > n
20      then ERROR(p, max-per-role(ag, rl))
21    for each CommClass cc ∈ sp.comms
22    do if max-of-class(rl, cc.id, n) ∈ sp ∧ |p.COMMOFTYPEWITHDEB(cc, ag)| > X
23         then ERROR(p, max-of-class(ag, rl, cc.id, X))
24    if role-confl(rl, rl_2) ∈ sp ∧ p.PLAYS(ag, rl_2) then ERROR(p, role-confl(ag, rl, rl_2))
25    if comm-role-confl(rl, C_1, C_2) ∈ sp ∧ p.PLAYS(ag, rl_2) ∧ p.DEBFORBOTH(ag, C_1, C_2)
26      then ERROR(p, comm-role-confl(ag, rl, C_1, C_2))
27 for each ag ∈ p.GETALLPARTICIPANTS()
28 do if sep-duties(C_1, C_2) ∈ sp ∧ p.DEBFORBOTH(ag, C_1, C_2)
29      then ERROR(p, sep-duties(ag, C_1, C_2))
```

Enactment engines. A protocol execution is an exchange of messages among the agents communicating their progress in fulfilling their commitments. An organization can support the execution of its protocols through an *enactment engine*, a system that can *compensate* to noncompliance with the specifications is detected. What it should do and when it should intervene depends on organizational requirements.

Implementation. We implemented our algorithms in a prototype Java tool that uses the Drools rule engine. Drools is part of the JBoss suite and it can efficiently handle large-scale scenarios. Our tool is currently working offline on textual event traces. We are working towards an version of the tool with event listeners and a graphical interface.

6 Advantages of Azzurra for business process specification

We show how different features of Azzurra make it better suited than existing approaches from the literature for the specification of business processes.

Flexibility. Instead of prescribing a specific course of action, Azzurra specifies correctness criteria. There are multiple possible executions that would comply with an Azzurra

specification. Azzurra is flexible because it is a declarative and social specification language. Being *declarative* (like, e.g., Declare [20]), Azzurra abstracts away operational details on how processes are carried out (those details are expressed by workflow languages). Azzurra protocols are specified in terms of actors accountability (who is responsible for what to whom): this *social* perspective reflects how business processes are executed in reality. The social nature of Azzurra is evident in the graphical notation (Figure 2), which shows how actors are related by commitments.

Executable language and compliance. Azzurra has an executable semantics in terms of creation and fulfillment of commitments. In Section 4, we build on the basic semantics of commitments (Figure 1) to define the meaning of more sophisticated primitives. Algorithms 1 and 2 apply the semantics and enable determining if the events inferred by actors interaction complies with process specifications in Azzurra.

Expressiveness and business primitives. Unlike existing approaches for modeling commitments protocols (e.g., [22, 5, 12]), Azzurra is an expressive language that includes business primitives for specifying business processes accurately. For example, Azzurra includes deadlines for commitment creation and fulfillment, agent variables to make an agent responsible for multiple commitments (e.g., in Figure 2, specialist sp is responsible for both examination and fracture assessment), authorizations about delegation/assignment/cancellation, role conflicts, separation of duties, workload limits.

Exception handling. The Azzurra language natively allows specifying compensations to exceptions: a commitment can be triggered by the failure of another: $\mathsf{failed}(\mathsf{C}_1) \twoheadrightarrow \mathsf{C}_2 : \mathsf{C}(\ldots)$ means that an instance of C_2 shall be created when an instance of C_1 has failed. Additionally, *enactment engines* can be to put in place by an organization to support compensation tactics that are not included in the specification. Once noncompliance is detected (by Algorithm 2), an enactment engine could intervene by finding an alternative agent in the organization (e.g., a different specialist to diagnose the patient).

7 Conclusions

We have presented Azzurra, a specification language for business processes based on *social models*, which relate interacting actors through social commitments. Azzurra comprises business primitives to facilitate the specification of business processes, such as delegation, deadlines, and role adoption constraints.

In addition to syntax and semantics of Azzurra, we have proposed a graphical notation to visualize the main elements of an Azzurra specification, and algorithms for checking if the interaction among a set of actors complies with protocol specifications.

This paper opens the doors to further work on social specifications of business processes, including (i) developing an enactment engine that supports remedies to noncompliance; (ii) empirical validation of Azzurra; (iii) improving the graphical notation; and (iv) investigating the joint usage of Azzurra and business process modeling languages.

References

1. B. Benatallah, F. Casati, F. Toumani, and R. Hamadi. Conceptual Modeling of Web Service Conversations. In *Proc. of CAiSE*, pages 449–467, 2003.

2. K. Bhattacharya, C. Gerede, R. Hull, R. Liu, and J. Su. Towards Formal Analysis of Artifact-Centric Business Process Models. In *Proc. of BPM*, pages 288–304, 2007.
3. A. K. Chopra, F. Dalpiaz, P. Giorgini, and J. Mylopoulos. Modeling and Reasoning about Service-Oriented Applications via Goals and Commitments. In *Proc. of CAiSE*, pages 113–128, 2010.
4. G. Decker, O. Kopp, F. Leymann, and M. Weske. Interacting Services: From Specification to Execution. *Data and Knowledge Engineering*, 68(10):946–972, 2009.
5. N. Desai, A. K. Chopra, and M. P. Singh. Amoeba: A Methodology for Modeling and Evolution of Cross-Organizational Business Processes. *ACM Transactions on Software Engineering and Methodology*, 19(2), 2010.
6. N. Desai, A. U. Mallya, A. K. Chopra, and M. P. Singh. Interaction Protocols as Design Abstractions for Business Processes. *IEEE Transactions on Software Engineering*, 31(12):1015–1027, 2005.
7. K. Figl and R. Laue. Cognitive Complexity in Business Process Modeling. In *Proc. of CAiSE*, volume 6741 of *LNCS*, pages 452–466, 2011.
8. F. Flores, M. Graves, B. Hartfield, and T. Winograd. Computer Systems and the Design of Organizational Interaction. *ACM Transactions on Information Systems*, 6:153–172, 1988.
9. A. K. Ghose and G. Koliadis. Auditing Business Process Compliance. In *Proc. of ICSOC*, pages 169–180. Springer, 2007.
10. G. Keller, M. Nüttgens, and A.-W. Scheer. Semantische Prozessmodellierung auf der Grundlage "Ereignisgesteuerter Prozessketten (EPK)". *Veröffentlichungen des Instituts für Wirtschaftsinformatik*, 89, 1992.
11. R. Khalaf. From RosettaNet PIPs to BPEL Processes: A Three Level Approach for Business Protocols. *Data and Knowledge Engineering*, 61(1):23 – 38, 2007.
12. E. Marengo, M. Baldoni, C. Baroglio, A. K. Chopra, V. Patti, and M. P. Singh. Commitments with Regulations: Reasoning about Safety and Control in REGULA. In *Proc. of AAMAS 2011*, pages 467–474, 2011.
13. W. N. Robinson and S. Purao. Specifying and Monitoring Interactions and Commitments in Open Business Processes. *IEEE Software*, 26(2):72–79, 2009.
14. M. La Rosa, P. Wohed, J. Mendling, A. H. M. ter Hofstede, H. A. Reijers, , and W. M. P. van der Aalst. Managing Process Model Complexity Via Abstract Syntax Modifications. *IEEE Transactions on Industrial Informatics*, 7(4):614–629, 2011.
15. S. Sadiq, G. Governatori, and K. Namiri. Modeling Control Objectives for Business Process Compliance. In *Proc. of BPM*, pages 149–164, 2007.
16. M. P. Singh. An Ontology for Commitments in Multiagent Systems: Toward a Unification of Normative Concepts. *Artificial Intelligence and Law*, 7(1):97–113, 1999.
17. M. P. Singh. A Social Semantics for Agent Communication Languages. In *Issues in Agent Communication*, pages 31–45. Springer, 2000.
18. M. P. Singh. Distributed Enactment of Multiagent Workflows: Temporal Logic for Web Service Composition. In *Proc. of AAMAS*, pages 907–914, 2003.
19. W. M. P. van der Aalst, M. Dumas, A. H. M. ter Hofstede, N. Russell, H. M. W. Verbeek, and P. Wohed. Life After BPEL? In *Formal Techniques for Computer Systems and Business Processes*, volume 3670 of *LNCS*, pages 35–50. Springer, 2005.
20. W. M. P. van der Aalst, M. Pesic, and H. Schonenberg. Declarative Workflows: Balancing between Flexibility and Support. *Computer Science-Research and Development*, 23(2):99–113, 2009.
21. WS-CDL. Web Services Choreography Description Language Version 1.0, November 2005. www.w3.org/TR/ws-cdl-10/.
22. P. Yolum and M. P. Singh. Flexible Protocol Specification and Execution: Applying Event Calculus Planning using Commitments. In *Proc. of AAMAS*, pages 527–534, 2002.