Multiagent and Grid Systems – An International Journal 6 (2010) 165–190
DOI 10.3233/MGS-2010-0147
IOS Press

On the integration of declarative choreographies and Commitment-based agent societies into the SCIFF logic programming framework

Federico Chesani[a], Paola Mello[a,*], Marco Montali[a], Sergio Storari[b] and Paolo Torroni[a]
[a]*DEIS – University of Bologna, viale Risorgimento 2, 40136 – Bologna, Italy*
[b]*ENDIF – University of Ferrara, Via Saragat 1, 44100 Ferrara, Italy*

Abstract. The definition of choreography specification languages for Service Oriented Systems poses important challenges. Mainstream approaches tend to focus on procedural aspects, leading to over-constrained and over-specified models. Because of such a drawback, declarative languages are gaining popularity as a better way to model service choreographies. A similar issue was met in the Multi-Agent Systems domain, where declarative approaches based on social semantics have been used to capture the nature of agent interaction without over-constraining their behaviour.

In this work, we present an integrated framework capable to cover the entire cycle of specification and verification of choreographies, by mixing approaches coming from the Service Oriented Computing and Multi-Agent Systems research domains. $\mathcal{S}$CIFF is the underlying logic programming framework for modelling and verifying interaction in open systems. The use of $\mathcal{S}$CIFF brings us two main advantages: (1) it allows us to capture within a single framework different aspects of a choreography, ranging from constraints on the flow of messages to effects and commitments resulting from their exchange; (2) it provides an operational model that can be exploited to perform a variety of verification tasks.

Keywords: Service choreographies, multi-agent systems, commitments, DecSerFlow, $\mathcal{S}$CIFF framework, event calculus

1. Introduction

The Service Oriented paradigm and the related technologies for implementing and interconnecting basic services are reaching a good level of maturity and a widespread adoption. Nevertheless, modeling service interaction from a global viewpoint, i.e., representing *service choreographies*, is still an open challenge [7]. In an inter-organizational context, choreographies define a "global view", which inherently crosses organizational boundaries and should be independent from the perspective of the individual participants. Each organization perceives a choreography as a public contract which specifies the rules for a sound collaboration of all parties. Business practices suggest to disclose private policies and rules only up to the strictly necessary extent, and no more. Hence, participants tend to hide their business policies, i.e., they say as little as possible about how a collaboration within a choreography is concretely

*Corresponding author: Paola Mello, DEIS, viale Risorgimento 2, 40136 – Bologna, Italy. Tel.: +39 0512093818; Fax: +39 0512093073; E-mail: paola.mello@unibo.it.

ISSN 1574-1702/10/$27.50 © 2010 – IOS Press and the authors. All rights reserved

Table 1
Some similarities between Multi-Agent and Service-Oriented systems

	MAS	SOA
interacting agents	autonomous heterogeneous agents	autonomous heterougeneous services
communication	communicative acts	messages
local view of interaction	(public) policies	behavioral interfaces
global view of interaction	interaction protocols	choreographies
Examples of Procedural Approaches	AUML, FSA	WS-BPEL, BPMN, WS-CDL
Examples of Declarative Approaches	Commitments, $\mathcal{S}$CIFF	DecSerFlow

carried out. Declarative approaches allow to specify the "rules of the game" without addressing the specific business policies, which should not be directly addressed at the choreography level.

In a intra-organizational context, choreographies are adopted to develop and document the internal organization and data-flow between the various components. Here, choreographies allow to manage the system complexity, thus supporting interoperability, and they provide a global view that managers can use to define business rules and local processes. Again, declarative approaches facilitate interoperability, since they abstract away from implementation details, and permit to easily define business rules for constraining business activities, and their properties.

The current major proposals for modeling service interaction are WS-BPEL [1] and WS-CDL [36]. As pointed out in [7,35], they are both procedural languages that lead to defining rigid control-flow and message-flow relationships, and to over-specifying and over-constraining the models. This often causes the modeler to miss the real focus of the choreography, and to rule out some otherwise perfectly acceptable interaction modalities.

A declarative approach to Service Oriented Architectures (SOA) choreography specification, in contrast to procedural approaches, has been proposed in [16], where van der Aalst and Pesic present DecSerFlow, a "truly declarative" graphical language for the specification of service flows. DecSerFlow adopts a more general and high-level view of service specification, using policies and business rules. DecSerFlow can be used to give incomplete and non-procedural specification of services, better focusing on the definition of (minimal) sets of constraints that capture the essence of a successful interaction. DecSerFlow uses a graphical representation of concepts that are semantically characterized in terms of Linear Temporal Logic (LTL).

The issue about what information is essential to define a global (or "social") view of interaction has been, and still is matter of discussion in another scientific community. In Multi-Agent Systems (MAS) research, agent interaction can be specified by posing suitable constraints on sequences of messages (or "communicative acts"). The concept of choreography corresponds to that of agent *interaction protocol*. It should not come as a surprise that Multi-Agent and Service-Oriented systems share many similarities [5, 19], and that similar solutions have been proposed (see Table 1).

As in the SOA setting, also in the MAS field several proposals have been made for modeling the interaction, from procedural approaches like AUML [8], to more declarative ones like social commitments [38] and the $\mathcal{S}$CIFF framework [2]. In [38], Yolum and Singh propose to adopt the notion of *commitment* to provide a semantics to the interaction protocols: an agent (the debtor) makes a commitment to another agent (the creditor) to bring about a certain property. They gives a meaning to the exchanged messages in terms of their impact on commitments, which in turn capture and handle mutual obligations between the interacting entities.

In [2] a different approach is proposed, which adopts the $\mathcal{S}$CIFF framework and Abductive Logic Programming (ALP) to represent the interaction protocols, in terms of rules linking observed events and

(future) expected behaviour. Noticeably, some of these approaches have been recently applied to the problems of business process adaptability [17] and of protocol composition [23].

We strongly believe that the SOA and the MAS research areas can mutually benefit from each other. In particular, we claim that a solution to the common issue of modeling choreographies can be found in the conjunct use of DecSerFlow and social commitments. Both the approaches are declarative, hence already overcoming the limits of procedural approaches. Moreover, they address different modeling aspects, being DecSerFlow focused on the execution flow while commitments capture the states of affairs and their associated properties and mutual obligations.

In this paper, we show how these two different approaches can be reconciled by means of the $\mathcal{S}$CIFF framework, acting as a unifying reference framework. The advantages of using $\mathcal{S}$CIFF comprise also a higher expressivity (e.g., in $\mathcal{S}$CIFF it is possible to model deadlines), and, above all, the possibility of performing various verification tasks (thanks to the $\mathcal{S}$CIFF operational counterpart, the $\mathcal{S}$CIFF Proof Procedure).

The paper is organized as follows: in Section 2 we motivate this work, by introducing a running example and discussing its many features (Subsection 2.1); in the same section we discuss also the limits of procedural approaches (Subsection 2.2) and motivate our choice of adopting DecSerFlow commitments and $\mathcal{S}$CIFF (Subsection 2.3). Sections 3 and 4 show how, respectively, the DecSerFlow language and the commitment approach can be fruitfully adopted for constraining the execution flow, and for representing state of affairs and mutual obligations between roles. Section 5 then describes how both proposals can be integrated by mapping them to the $\mathcal{S}$CIFF framework, briefly sketching the verification techniques enabled by such a mapping. Related Works and Conclusions follow.

2. Motivation

Let us consider a choreography that aims to define/constrain the behaviour of some players when submitting an “order” for a set of items. The choreography envisages three different roles: a *customer* which interacts with a *seller* to place an order of a set of items, and a *warehouse* which could participate to the interaction by communicating to the seller if it is able (or not) to ship the ordered items. We distinguish different interactions among many services that follow the choreography by assuming that there is a unique identifier (frequently called *instance identifier*) for each interaction: e.g., in this choreography we can assume that all the executed activities related to the same interaction are identified by means of an order number, intuitively referring to the concept of “order”. Each interaction among services that aim to follow the choreography is referred as a *choreography instance*.

Example 1. (*The Customer-Seller-Warehouse Scenario*)

The customer makes up an order by choosing one or more items from the seller list. Before committing an order, it is always possible to cancel it; in this case, the user cannot choose other items within the same instance anymore, and the choreography instance terminates. After having committed an order, the customer expects a positive or negative answer from the seller. The seller could freely decide whether to confirm or refuse customer’s order, but sometimes it has also to consider the opinion of the warehouse about the shipment. In particular:

- *the seller can confirm the order only if the warehouse has previously confirmed the shipment;*
- *if the warehouse states that it is unable to execute the shipment, then the seller should refuse* (*or have refused*) *the order.*

When the order is committed, in case of a confirmation from the seller the order becomes established, and the customer is in charge to pay for it. Conversely, when the seller receives the payment for an established order, she must deliver a single corresponding receipt. However, the seller is free to change her mind at any time: she can refuse an already established order. Clearly, if the payment has already been done, this exceptional situation must be handled by the seller, which is committed to refund the customer.

The way this example choreography is described is inherently declarative. It does not fix the control flow of the involved services, nor how they should exchange messages in order to accomplish the choreographic strategic goal. Rather, it focuses on a more abstract level, trying to capture the essential of the interaction by adopting a global and open perspective, not driven by implementation needs.

2.1. Explicit and implicit constraints

The natural language description of Example 1 directly suggests a list of several, explicit constraints on the activities execution, like for example:

- time-ordered relationships among activities ("*after* having committed an order, the customer expects a positive or negative answer");
- cardinality constraints ("the seller must deliver a *single* corresponding receipt");
- negative relationships, to express also what is forbidden during the choreography execution ("the user *cannot* choose other items [. . .] anymore")
- non-deterministic/opaque choices as well as relationships among unordered activities (e.g., the seller can refuse independently from the warehouse answer).

It is worth noting that negative information, as far as we are concerned, is not addressed by current proposals: they adopt a procedural-oriented control flow approach making the implicit assumption that all that is not explicitly modeled is forbidden. As pointed out in [16], the impossibility of expressing negative relationships forces the modeler to explicitly enumerate all the allowed possibilities, introducing ambiguous decision points. This often leads to over-constrain the model, forbidding possible executions which actually correctly realize the intended choreography (see [13] for a discussion).

Example 1 contains also constraints that are not stated explicitly in the description of the choreography, but that must be addressed at design time. Some of them are inherent to the execution of the activities. For example, it is assumed that either an order is committed or is canceled, or that either a refuse shipment activity or a confirm shipment activity are executed, but not both.

Then, there are constraints that are not related to the control flow, but rather they involve the participants, the role they are playing and the reached state of affairs. Roughly speaking, such constraints typically capture the obligations and commitments which relate interacting roles when certain circumstances/states of affairs have been reached during the interaction, independently of control flow. As clearly pointed out in the payment phase of Example 1, they can identify exceptional cases and define how the corresponding cancelations/compensations should be handled. E.g., Example 1 states that if the seller refuses an already paid order, it is committed to refund the customer. Such a constraint is not directly bound to the execution flow among activities, but it is instead more focused on mutual obligations between roles, by considering reached state of affairs as well as *effects* of activities execution.

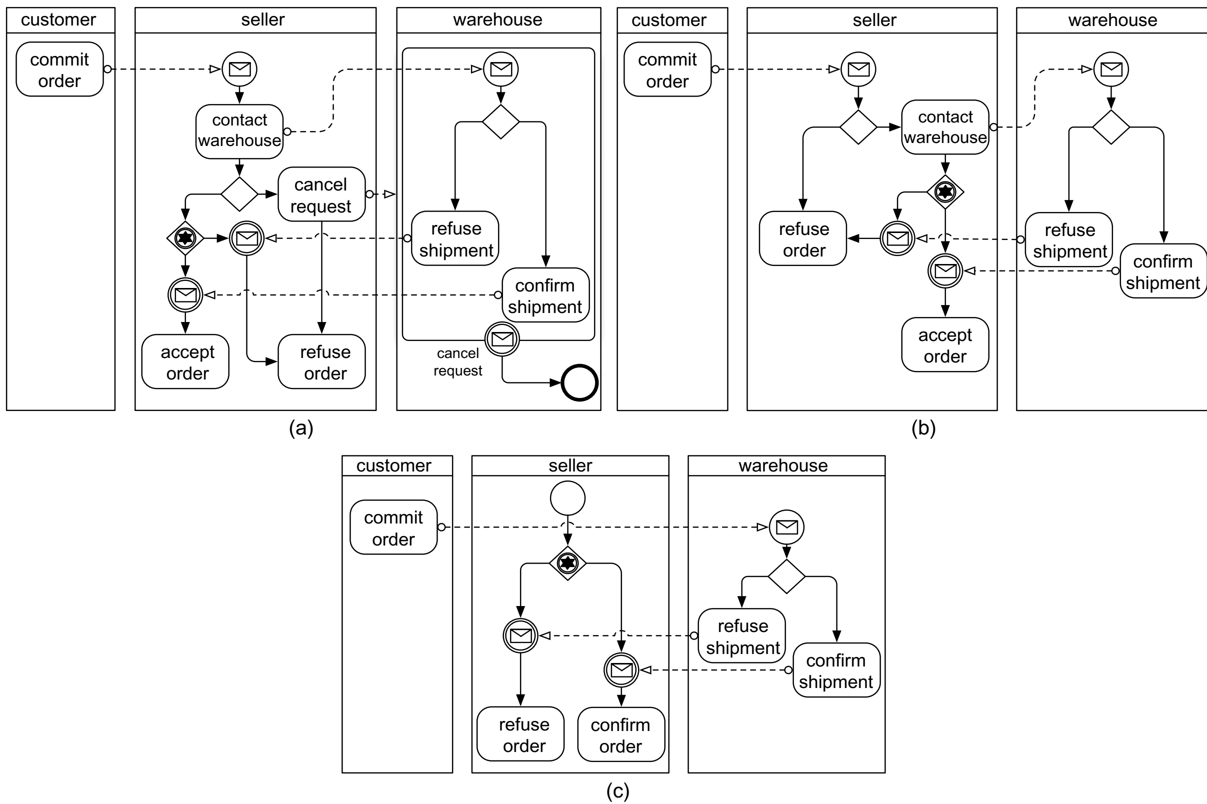

Fig. 1. Three different possible realizations of the acceptance phase in BPMN.

2.2. Procedural approaches, over-specifications, and the need for declarativeness

Avoiding over-specifications is a key issue when modeling choreographies. Instead of strictly specifying one of the possible behaviors which is able to respect the choreography, the aim of the modeler should be the identification of the minimal set of constraints that correctly regulates the interaction, achieving a trade-off between the specification of what is forbidden/expected and what is allowed.

An interesting example which clearly shows such issue is the order acceptance phase described in Example 1. The aim of this phase is to identify when a committed order should be accepted or rejected by the seller, taking into account (in some cases) the warehouse too. At a choreographic level, the coupling between seller and warehouse and between customer and warehouse is reduced at a minimum. First of all, when and how the warehouse is contacted is not specified; furthermore, there could be different choreography executions in which the warehouse is not contacted at all. An execution in which the seller autonomously decides to reject the order, without asking warehouse's opinion, is clearly accepted by the choreography; the case in which the warehouse refuses the shipment without observing the committed order (e.g., because it is overloaded) is implicitly envisaged too.

The over-specification problem arises if we try to model the acceptance phase by using one of the current proposed languages for choreographies. Figure 1 shows three different over-specified possible realizations of the acceptance phase by adopting BPMN [37] collaborative models.

Figure 1(a) shows a choreography where, after having received order's commitment, the seller contacts the warehouse in order to know if it can ship the order or not. Then, if the seller evaluates that, due to

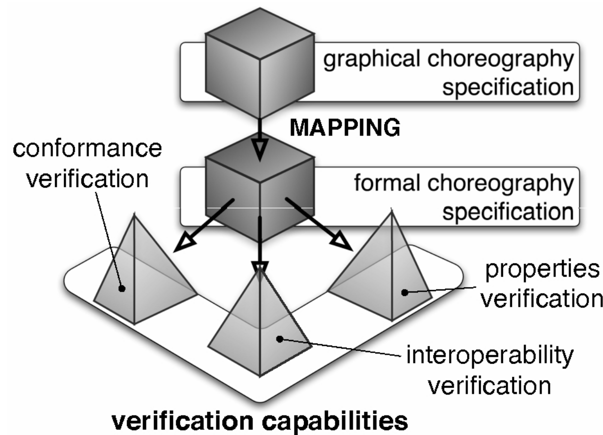

Fig. 2. An integrated framework for the specification and verification of service choreographies.

a private policy, it is in any case unable to confirm the order, it will send a message to the warehouse in order to stop the processing of its decision; otherwise, the seller will confirm or refuse the order by considering warehouse's answer. In Fig. 1(b), instead, we find that the seller firstly evaluates its internal policies, and contacts the warehouse only if the choreography prescribe to do so (i.e. only if it would accept the order; in this case, receiving an answer from the warehouse is a mandatory requirement). Finally, Fig. 1(c) shows a different message flow from customer's side, and envisages a seller who does not apply any private choice, but simply forwards what has been decided by the warehouse.

The three diagrams show that approaching the choreography specification task by directly adopting a typical control/message flow perspective leads to pointlessly complicate the model, because the required user abstractions are not properly reflected in the underlying modeling languages. We think that such a perspective should be matter of a second phase, in which the choreographic model is grounded on a set of service behavioral interfaces, to be developed from scratch or selected from an already existing repository.

2.3. DecSerFlow commitments and $\mathcal{S}CIFF$

To model choreographies by reflecting the required abstractions and flexibility, we propose to integrate the following approaches coming from the Service Oriented Computing and Multi-Agent Systems research areas (see Fig. 2):

– DecSerFlow [16], as graphical language for specifying constraints on the flow of activities. DecSerFlow is chosen because its graphical constructs can be suitably used to specify the "contractual" nature of choreographies in an intuitive and declarative fashion.
– Social commitments [34], as a way to capture relationships between interacting entities in an elegant and flexible way. While DecSerFlow is used to constrain the choreography execution flow, commitments are introduced to link activities and their effects with the mutual obligations that hold among interacting entities. Hence, commitments can be used to explicitly characterize the *states of affairs* reached during the interaction, which are left implicit by DecSerFlow; this, in turn, makes commitments a suitable approach to handle situations where exceptions must be explicitly identified and handled with corresponding compensations (see Section 4.3).

– $\mathcal{S}$CIFF [2], as an underlying unifying framework. While DecSerFlow is directly mapped to $\mathcal{S}$CIFF [11,28], for representing commitments we use their corresponding formalization into Event Calculus [38] ($\mathcal{EC}$ for short), which in turn can be specified in $\mathcal{S}$CIFF taking advantage from its reactivity features. The advantage of this mapping is twofold: from a declarative point of view, $\mathcal{S}$CIFF is used to provide a uniform underlying semantics to DecSerFlow and social commitments; from an operational point of view, the $\mathcal{S}$CIFF proof procedure can be exploited to execute a variety of verification tasks. A third advantage, which is not in the scope of the paper, is related to the expressiveness of $\mathcal{S}$CIFF; indeed, $\mathcal{S}$CIFF is able to model and constrain data, and in particular the activities execution times. The interested reader may refer to [11,28] to obtain more details about how $\mathcal{S}$CIFF is used to extend DecSerFlow with quantitative temporal constraints, such as deadlines.

3. Modeling choreographies using DecSerFlow

In [16], van der Aalst and Pesic propose DecSerFlow, a declarative language for modeling service flows. Besides declarativeness, its advantages rely on its appealing graphical appearance, its extensibility and its formal semantics given by means of Linear Temporal Logic (LTL).

Modeling service specifications in DecSerFlow starts by identifying the different involved activities (i.e., atomic logical unit of work), and then to identify constraints on their execution, à la policies/business rules. The semantics of constraints is given as LTL formulas, hence the name “formulas” to indicate DecSerFlow relationships.

DecSerFlow core relationships are grouped into three families:

– *existence formulas*, unary relationships used to constrain the cardinality of activities;
– *relation formulas*, which define (positive) relationships and dependencies between two (or more) activities;
– *negation formulas*, the negated version of relation formulas.

Typically, the terms *source* and *target* activities indicate activities linked by a relation formula or by a negation formula, where the execution of the source activity “activates” the relation and impose some constraint on the target activity. For example, a DecSerFlow relation is the response formula, that states that if activity A is executed, then activity B should be executed after. A DecSerFlow negation formula is the not coexistence, which states that if activity A is executed, then activity B cannot be executed (before or after A), and vice-versa. For a complete description of the DecSerFlow language and its underlying LTL formalization, the interested reader is referred to [16].

3.1. Modeling the Customer-Seller-Warehouse example in DecSerFlow

Table 2 shows how the different statements of Example 1 can be expressed as DecSerFlow activities and constraints in an intuitive and straightforward way.

For example, to prevent the customer committing an order that has been canceled, and to prevent her to cancel an order already committed, the not coexistence formula has been used (constraint C_2).

DecSerFlow’s responded existence relation is used to model the relationship between the refusals of shipment and order: it states that if the shipment is refused by the warehouse, the refuse order activity should be executed too, either before or after it (constraint C_7).

The response relation is a responded existence formula augmented with an ordering constraint imposing that the target activity should happen *after* the source activity; for example, constraint C_4

Table 2
Mapping the statements of the running example to DecSerFlow constraints. The **"constraint name"** column reports the constraint names for this choreography ($C_1..C_8$, as shown in Fig. 3), with the corresponding DecSerFlow name

source		constraint name	target	description (from example 1)
cancel order	C_1	negation response	choose item	in case of cancelation, the user cannot choose other items [. . .] anymore
	C_2	not coexistence	commit order	a canceled order cannot be committed (and vice-versa)
	C_3	precedence	choose item	an order is made up by at least one chosen item
commit order	C_4	response	refuse or confirm order	after having committed an order, the customer expects a positive or negative answer from the seller
	C_5	precedence	confirm shipment	the seller could confirm the order only if the warehouse has previously confirmed the shipment
	C_6	precedence	choose item	an order is made up by at least one chosen item
refuse shipment	C_7	responded existence	refuse order	if the warehouse is unable to execute the shipment, then the seller should refuse (or have refused) the order
receipt delivery	C_8	absence(1)		the seller will deliver a single receipt

states that after having executed the commit order, then a positive or negative answer from the seller is expected to be performed afterwards (when more target activities are involved, they are considered in a disjunctive manner). A precedence formula is provided too: again, a responded existence formula augmented with an ordering constraint imposing that the target activity should happen *before* the source activity (e.g. C_6, that constraints an order to be committed only if an item has been previously chosen).

To specify that only a single receipt should be delivered by the seller (i.e., constraint C_8), we use the DecSerFlow absence(1) existence formula. The absence(N) formula indeed states that the involved activity cannot be executed more than N times, i.e. constrains its cardinality between 0 and N.

For each positive relationship, DecSerFlow defines a corresponding negative version. Basically, negative relations forbid the execution of the target activity under certain conditions. E.g., the responded absence relationship (which is actually the negation of the responded existence one) states that if the source activity is executed, then the target activity is forbidden. It is worth noting that, as pointed out in [16], some negative relations are equivalent; e.g., stating that between a generic activity B and a generic activity A there is a responded absence formula is equivalent to specify that executions of A and B should not coexist (not coexistence formula between A and B) in the same execution instance. Such a negative relationship is used e.g. to model the impossibility to commit an order if it is canceled by the customer (constraint C_2).

By analyzing Example 1, we could complete the DecSerFlow diagram shown in Table 2 with other useful constraints not explicitly stated in the choreography description, in order to really model all the intended concepts of the description; the added constraints are shown in Table 3, while in Fig. 3 the whole set of constraints is shown using the DecSerFlow graphical notation.

Among the added constraints, C_{15} and C_{16} deal with the core concept of the choreography, which is actually the commitment of one order. Since the order could also be canceled, we attach an absence(1) constraint (i.e., C_{15}) to the commit order activity (to express that at most one order can be committed for each choreography instance), and bind the cancelation and the commitment with a choice DecSerFlow relation (i.e., constraint C_{16}), which states that at least one of the two connected activities has to be executed: an order should be committed or canceled.

3.2. The problem of representing states of affairs, exceptions and compensations

Let us now discuss the relation between the activities payment and deliver receipt. From the Seller viewpoint, she will send the deliver only if the payment has been previously done: this is captured by

Table 3
DecSerFlow constraints added to complete the running example. Note that some constraints are between a *source* and a *target*, while other constraints do not make such distinction and are imposed between two *targets*

source		constraint name	target	description (from example 1)
refuse order	C_9	precedence	commit order	An answer from the seller is valid only if it is performed after having committed the order
confirm order	C_{10}	precedence	commit order	
payment	C_{11}	precedence	confirm order	A valid payment should be preceded by the confirmation of the order
deliver receipt	C_{12}	precedence	payment	The receipt should be delivered only if the order has been paid
refuse order	C_{13}	negation response	confirm order	While a confirmed order can be refused, the opposite should not happen (nothing is said in the example)
target		constraint name	target	description (from example 1)
confirm shipment	C_{14}	not co-existence	refuse shipment	Possible warehouse's answers are mutually exclusive
commit order	C_{15}	cardinality 0..1		the choreography centres around the concept of a single order, which could possibly be canceled
commit order	C_{16}	choice	cancel order	

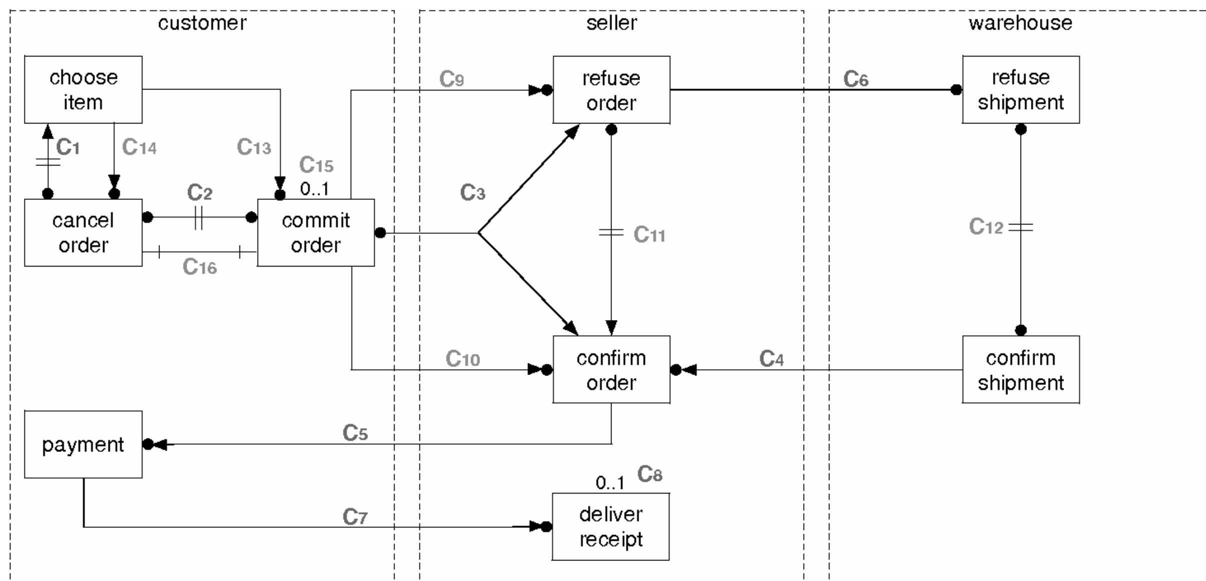

Fig. 3. DecSerFlow model of the running example. Activities are represented by means of boxes, while DecSerFlow formulas are the link between the boxes. Note that each formula/link has been labeled with an identifier C_i, in order to easy reference to them. The reader interested in the DecSerFlow notation can refer to [16].

the DecSerFlow Constraint C_{12}. However, it is not possible to put a specular constraint that imposes the delivery of the receipt as a consequence of the payment (which is a *desiderata* of the Customer). This because of the possibility, given to the Seller, to *"change her mind at any time"*, also if the payment has been already done. I.e., a situation where the Customer pays, and then the Seller decides to refuse the established order is allowed by Example 1. Moreover, the example explicitly asserts that, if this situation happens, then the Seller should give back the money. Such behaviour can not be represented by means of DecSerFlow constraints: would we try to introduce a new hypothetic refund activity, we would encounter the problem of specifying that it should be allowed only if the seller changes idea (a

refuse order activity being executed), after the order was established and after the Customer paid. Due to this impossibility, we simply let the DecSerFlow diagram unconstrained w.r.t. this part of the example.

In general, DecSerFlow is a powerful and flexible framework for constraining the execution flow, but it cannot deal with such special cases because it has no explicit notion of the *state of affairs* reached during interaction. Modeling the states of affairs is important, especially when exceptional situations must be identified and *compensated*[1] to bring back the overall process in a consistent state. The Customer's refund of Example 1 is a typical example of compensation.

While at static time verification is carried out by considering only all the "normal" cases, abstracting away from errors and compensations, at run-time these exceptional possibilities must be taken into account (especially in an *open* setting). In the next section, we describe how they can be suitably modeled by considering the *effects* of activities and the mutual obligations that hold between the interacting parties, thus completing the DecSerFlow diagram in its unconstrained part.

4. Modeling choreographies using commitments

Social commitments [34] are a widely recognized framework for modeling interaction in open and heterogeneous Multi-Agent Systems, ensuring flexibility and allowing interacting agents to exploit opportunities. Instead of focusing on the flow of messages through which interaction reveals itself, commitments capture the mutual obligations which constrain interacting roles to bring about a certain property of condition. Two types of commitments are introduced:

– a *base-level* (or *unconditional*) commitment $\mathsf{c}(X, Y, P)$ means that a debtor agent X is committed to a creditor agent Y to bring about a property/condition P;
– a *conditional* commitment $\mathsf{cc}(X, Y, P, Q)$ means that if a condition P holds, then X becomes committed towards Y about Q.

Commitments are initiated, manipulated and resolved by interacting agents when their actions lead to a certain state of affairs. E.g., a conditional commitment is transformed into a base-level one if the reached state of affairs satisfy the condition P. Conditions and effects are in turn satisfied and generated by agents when they perform activities or exchange messages.

4.1. Manipulation of commitments

We briefly recall the six basic operations introduced in [34] to manipulate commitments:

– $\mathsf{create}(E, X, C, T)$. The debtor agent X establishes commitment C by generating event E at time T;
– $\mathsf{discharge}(E, X, C, T)$. The debtor agent X successfully accomplishes commitment C by generating E at time T;
– $\mathsf{cancel}(E, X, C, T)$. If the debtor agent X generates E at time T the commitment C is canceled, often being replaced by a new commitment to compensate the cancelation;
– $\mathsf{release}(E, Y, C, T)$. When generating E at time T, the creditor agent Y releases the debtor from the obligation of resolving C;

[1]Compensation is a mechanism introduced in the Service Oriented Computing to manage exceptional circumstances that could invalidate the actual process/service execution.

Table 4
The reactive Event Calculus ($\mathcal{REC}$) ontology

happens(Ev, T)	Event Ev happens at time T
holds(F, T_i, T_f)	Fluent F begins to hold from time T_i and persists to hold until time T_f
not_holds(F, T_i, T_f)	Fluent F ceases to hold at time T_i and persists to not hold until time T_f
holdsat(F, T)	Fluent F holds at time T
not_holdsat(F, T)	Fluent F does not hold at time T
initially(F)	Fluent F holds from the initial time
initiates(Ev, F, T)	Event Ev can initiate fluent F at time T; this means that if F does not hold at time T, it is declipped by the happening of Ev at that time
terminates(Ev, F, T)	Event Ev can terminate fluent F at time T; if F holds at time T, it is clipped by the happening of Ev at that time

- $\mathsf{assign}(E, Y, Z, C, T)$. Through the generation of event E at time T, the creditor agent Y transfers the commitment to a new creditor Z, making Z the new beneficiary of C;
- $\mathsf{delegate}(E, X, Z, C, T)$. With event E at time T, the debtor agent X transfers the responsibility of carrying out C to a new debtor Z.

In [38], Singh and Yolum exploit Event Calculus ($\mathcal{EC}$) to specify and manipulate social commitments. Table 4 summarizes the main elements of the $\mathcal{EC}$ theory which will be formalized in Section 5.3.

In the $\mathcal{EC}$ framework, messages exchange and activities execution are modeled by means of occurring events, whereas commitments, conditions and effects caused by events are represented with *fluents*, which holds inside certain time intervals and are initiated/terminated by occuring events. In [34], several axioms are introduced to define the operations on commitments, linking them to the $\mathcal{EC}$ predicates shown in Table 4. To cite an example, we report the two axioms related to the discharge operation. They specify that a commitment is terminated by an event which discharges it; an event has in turn the ability of discharging a commitment associated to condition P if its generation initiates P (i.e. leads P to hold):

$$terminates(E, \mathsf{c}(X, Y, P), T) \leftarrow \mathsf{discharge}(E, X, \mathsf{c}(X, Y, P), T).$$

$$\mathsf{discharge}(E, X, \mathsf{c}(X, Y, P), T) \leftarrow initiates(E, P, T).$$

In a concrete case, such axioms may be enriched by specifying further conditions (e.g. by constraining the debtor and creditor roles, or by introducing data-related aspects), to precisely define the state of affairs which leads a certain commitment to be created, discharged, canceled and so on. Such an expressive power makes commitments a suitable framework to declaratively model complex interaction patterns, such as exceptions handling [24].

In the following sections, we will address such issues in the context of our running example. In particular, as clearly pointed out in Example 1, the concluding part of the choreography, which deals with the payment and receipt delivery, is focused on mutual obligations between the Customer and Seller roles, taking also into account exceptional cases and corresponding compensations. Such obligations can be easily captured in terms of social commitments between the interacting roles, leading to the possibility of manipulating them in a flexible and declarative way.

4.2. Customer-Seller-Warehouse example: modeling the effects of activities

Typically, commitments are established when a certain state of affairs has been reached inside the choreography. As already pointed out, by exploiting $\mathcal{EC}$ the global state can be characterized by means of fluents currently or previously holding. Fluents, in turn, are used to represents conditional or base-level commitments, and to represent properties and effects caused by messages exchange.

First of all, we identify effects and properties caused by the execution of activities. For easy of comprehension, we do not capture all the effects, but only those we are interested in. Following the example, we say that an order becomes established when the seller confirmes it; such a property is terminated if the seller changes her mind and subsequently refuses the order. Furthermore, activities pay and deliver receipt respectively cause the order to be paid and a corresponding receipt to be delivered. We represent all these conditions by means of Prolog clauses:

$$initiates(confirmOrder(S,C), orderEstablished, T) \leftarrow role(S, seller), role(C, customer).$$

$$terminates(refuseOrder(S,C), orderEstablished, T) \leftarrow role(S, seller), role(C, customer).$$

$$initiates(pay(C,S), orderPaid, T) \leftarrow role(C, customer), role(S, seller).$$

$$initiates(deliverReceipt(S,C), receiptDelivered, T) \leftarrow role(S, seller), role(C, customer).$$

Note that we capture the concepts of sender/receiver (and also commitment debtor and creditor) by only referring to their corresponding choreography role. We model the fact that entity X plays role R by using a Prolog predicate $role(X, R)$. When the choreography is grounded on a specific setting, i.e. interacting roles are played by concrete services, we can simply specify which service covers a certain role by enlisting a set of Prolog facts $role$ (e.g., $role(http://soap.example.com/seller.wsdl, seller)$ states that the service available from the given URL plays the role of seller). These facts will be part of a Knowledge Base given as input, together with DecSerFlow constraints and commitments, to the $\mathcal{S}$CIFF proof procedure (see Section 5.1).

4.3. Customer-Seller-Warehouse example: modeling commitments and compensations

Now we identify some commitments envisaged in Example 1. Two of them are conditional commitments capturing the mutual obligations between Customer and Seller respectively about order payment and delivery. On the one hand, the Customer is committed to pay for the order when she has committed it, provided that the order has been effectively established; on the other hand, the Seller is committed to deliver a receipt for the order when she has confirmed it, provided that it has been paid. We formalize these two obligations by means of conditional commitments and by identifying activities which lead to their creation:

$$\begin{aligned}&\mathsf{create}(commitOrder(C,S), C,\\&\quad \mathsf{cc}(C, S, orderEstablished, orderPaid), T) \leftarrow role(C, customer), role(S, seller).\\&\mathsf{create}(confirmOrder(S,C), S,\\&\quad \mathsf{cc}(S, C, orderPaid, recDelivered), T) \leftarrow role(S, seller), role(C, customer).\end{aligned}$$

The next two rules, are instead introduced to model in an intuitive and elegant manner the payment phase, taking into account the exceptional situation where the Seller refuses a previously accepted order, which has been already paid by the Customer. Should this exception situation occur, some compensation activity must be executed to refund the Customer, bringing back the overall process in a consistent state.

First of all, if Seller's commitment about the receipt delivery is pending when the Seller refuses the order, then the commitment becomes canceled: the Seller will not be in charge to deliver the receipt anymore. However, if the Customer has already paid, i.e. the *orderPaid* fluent holds at the refusal time, then the cancelation of the commitment must be compensated by refunding the Customer.

The compensation is again modeled in terms of a commitment from the Seller to the Customer about payment's refund.

$$\mathsf{cancel}(refuseOrder(S,C), S, \mathsf{c}(S,C,recDelivered), T) \leftarrow role(S, seller), role(C, customer).$$

$$\mathsf{create}(refuseOrder(S,C), S, \mathsf{c}(S,C,refDelivered), T) \leftarrow role(S, seller), role(C, customer), holdsat(orderPaid, T).$$

As described above, establishment and payment of the order are linked to performed activities by a cause/effect relationship; for example, an order becomes concretely established when the seller confirms it. By explicitly modeling effects and separating them from their causes, we guarantee openness and flexibility of the choreography: commitments are satisfied because certain properties have been brought about by interacting entities, not because specific messages have been exchanged. Hence, new ways to interact can be found (being in any case compliant with the flow constraints imposed by the DecSerFlow model), if they still lead to bring about the desired properties.

For example, the compensation commitment, modeling that the Customer must be refunded, does not consider a specific refunding activity, but simply states that a $refDelivered$ effect (the effect of refunding the Customer) must be produced in some way. However, it is necessary to identify at least one activity whose effect is to bring to life the fluent $refDelivered$, such as for example a refundByBankTransfer activity. The risk otherwise would be to instantiate a commitment without having the possibility of discharging (satisfying) it, hence making the choreography specification unsatisfiable for this specific case.

It is worth noting that, even if commitments' manipulation implicitly constrains the execution of activities,[2] modeling it by means of flow constraints would not be the right choice. The main reason is methodological: commitments do not directly refer to events and activities, but on conditions and states of affairs, without specifying how this conditions/states have been reached. Reducing commitments to constraints on the flow of activities would lead to pointlessly complicate the model, being in some cases even impossible to be expressed (as we have seen in Section 3.2 for the payment phase of our running example).

5. Mapping DecSerFlow and commitments to the $\mathcal{S}$CIFF framework

We now present the $\mathcal{S}$CIFF framework, sketching how it faces the problem of specifying interaction in open and heterogeneous systems with a declarative flavor. We then show how both DecSerFlow and social commitments can be represented in this framework, hence proposing $\mathcal{S}$CIFF as an underlying unifying language. While the mapping of DecSerFlow is directly given by means of $\mathcal{S}$CIFF rules, social commitments are supported because of their underlying representation in Event Calculus [38], exploiting the possibility of enconding a form of reactive Event Calculus in $\mathcal{S}$CIFF.

5.1. The $\mathcal{S}$CIFF approach to Agent Interaction Protocols

The $\mathcal{S}$CIFF [2] language was originally introduced for the specification of global interaction protocols in open agent societies. It does not make any assumption about participants internals, but instead focuses

[2] Indeed, a created base-level commitment must be discharged, canceled, released or delegated sometime in the future.

on the observable and relevant events which occur within the society at run-time. To let the user decide which are the relevant events inside the considered domain, the $\mathcal{S}$CIFF language completely abstracts from the problem of deciding "what is an event". In the SOA context an event can be either the invocation of a service or the exchange of a message.

$\mathcal{S}$CIFF adopts an explicit notion of time, and models the occurrence of an event Ev at a certain time T as $\mathbf{H}\ (Ev, T)$, where Ev is a logic programming term and T is an integer, representing the discrete time point at which the event happened (the bold **H** stand for "Happened"). The set of all the events that have happened during a protocol execution constitutes its interaction log.

Beside the explicit representation of what has already happened, $\mathcal{S}$CIFF introduces the concept of "what" is expected to happen, and "when". The notion of expectation plays a key role when defining interaction protocols, choreographies, and more in general any dynamically evolving process: it is quite natural, in fact, to think of such processes in terms of rules of the form "if A happened, then B should be expected to happen, under certain conditions". $\mathcal{S}$CIFF pays particular attention to the openness of interaction: interacting peers are not completely constrained, but they enjoy some freedom. This means that the prohibition of a certain event should be explicitly expressed in the model and this is the reason why $\mathcal{S}$CIFF supports also the concept of negative expectations (i.e. of what is expected not to happen). Positive expectations about events come with form $\mathbf{E}\ (Ev, T)$, where Ev and T could be variables, or they could be grounded to a particular (partially specified) term or value respectively. Constraints (à la Constraint Logic Programming, CLP) as well as Prolog predicates can be specified over the variables; this makes it possible to define conditions and restrictions on the data associated to events. Moreover, in $\mathcal{S}$CIFF the time is explicitly modeled, differently from other mainstream approaches such as temporal logics, which have an abstract notion of time (as a sequence of events); as a consequence, constraints and predicates can also be associated to the temporal variables associated to happened events or expectations, supporting the possibility of modeling quantitative temporal constraints. For example, attaching the constraint $T > 10$ on the above expectation means that the expectation is about an event to happen at a time greater than 10. Negative expectations about events come with form $\mathbf{EN}(Ev, T)$; just to give an intuition, variables used inside negative expectations are universally quantified: writing $\mathbf{EN}(Ev, T) \wedge T > 10$ means that Ev is forbidden at any time which is greater than 10. Positive and negative expectations present strong similarities with deontic logic; indeed, $\mathcal{S}$CIFF has been exploited to express the deontic operators [3].

Social Integrity Constraints are forward rules used to link happened events and expectations in order to define the declarative rules which regulate the course of interaction, i.e. to model the interaction protocol. They come as rules of the form $body head$, where $body$ can contain (a conjunction of) happened events and expectations, and $head$ can contain (a disjunction of conjunctions of) positive and negative expectations. For example, to model that "if a customer sends the payment to the seller, then the seller should answer delivering the corresponding receipt, within 24 hours" we could use the following Integrity Constraint:

$$\mathbf{H}(pay(C, S, Order), T_p) \rightarrow \mathbf{E}(deliver(S, C, receipt(Order, Id)), T_d)$$
$$\wedge T_d > T_p \wedge T_d < T_p + 24.$$

With a slight extension of the $\mathcal{S}$CIFF language, in the $head$ of a Social Integrity Constraint we also allow the introduction of happened events: the intended meaning is that whenever the body is evaluated as true, then the events specified in the $head$ happen. I.e., the $\mathcal{S}$CIFF proof procedure simulate the happening of the events introduced in the $head$ by automatically generating the corresponding $\mathbf{H}(Ev, T)$.

$\mathcal{S}$CIFF accepts also a (Prolog) knowledge base, where the user can define all the pieces of knowledge which are independent from the interaction, and which are used inside Integrity Constraints to put

conditions on the involved variables. In this way, $\mathcal{S}$CIFF reconciles forward, abductive reasoning with backward, goal-oriented reasoning: while Integrity Constraints are interpreted in a forward, reactive way (whenever the body of a constraint becomes true, then also its head must be true), predicates are interpreted in a classical Prolog setting (i.e., resolved with a backward-chaining strategy).

The $\mathcal{S}$CIFF semantics is based on Abductive Logic Programming [20], a powerful mechanism for hypothetical reasoning in the presence of incomplete knowledge, that is handled by labeling some pieces of information as "abducibles". Abducibles can be viewed as possible hypotheses which can be assumed, provided that they are consistent with the current knowledge base. The abduction process is typically applied when looking for an explanation about some observation. Starting from some observed facts, possible causes are hypothesised (they are abduced). Then it is possible to confirm the hypotheses by performing some additional observation: for example, the scientist postulates some theory, and then develops new experiments to confirm (or disconfirm) such theory. To constrain the way hypotheses are made, Integrity Constraints are added, linking observed facts and hypotheses (abducibles) that can be made starting from that observed facts.

In $\mathcal{S}$CIFF an interaction specification (i.e. the set of rules regulating the allowed possible interactions) is treated an Abductive Logic Program, where Integrity Constraints define the interaction protocols, and positive/negative expectations are considered as abducibles. The operational counterpart of the language, namely the $\mathcal{S}$CIFF proof procedure, is indeed able to verify compliance of a set of interacting entities w.r.t. the considered protocol: it hypothesizes positive (resp. negative) expectations and then performs an *hypothesis confirmation* step checking whether a matching happened event actually exists (resp. does not exist). For a detailed description of the $\mathcal{S}$CIFF language, as well as its declarative semantics and the corresponding proof procedure, the interested reader is referred to [2].

5.2. Expressing DecSerFlow concepts in $\mathcal{S}$CIFF

Each DecSerFlow constraint can be mapped to one or more $\mathcal{S}$CIFF Integrity Constraints. We assume activity executions to be atomic, i.e., we represent each execution with a single event. It would be possible, of course, to adopt a representation based on a couple of events, distinguishing the start of the activity execution and its completion. Our approach can cope with such mapping, provided some axioms would properly be modified. However, for the sake of comprehension, in this paper we do not explore such possibility.

For example, the fact that a customer C has committed the order to seller S can be represented by an event of the type $commitOrder(C, S)$.[3]

For a complete mapping, the interested reader is referred to [11,28]; in the remainder of this section, we will illustrate some mappings grounding the presentation on the running example.

The body of the Integrity Constraint which maps a relation or a negation formula contains an happened event which corresponds to the formula's source (each DecSerFlow relation is triggered when its source activity is performed). The head instead contains a positive expectation in the case the relation is a positive one (an event representing an activity execution will be expected to happen); in the case the relation is a negative one, the head will contain a negative expectation (an event representing an activity execution will be prohibited).

Furhermore, since $\mathcal{S}$CIFF adopts an explicit notion of time, DecSerFlow concepts such as *after* and *before* orderings among activities are modeled by constraining the involved execution times. Therefore,

[3]Note that the current version of DecSerFlow does not support activities-related data, and therefore neither the sender and receiver of a message; these concepts can instead be seamlessly modeled in $\mathcal{S}$CIFF.

Table 5
Mapping of the simple DecSerFlow relation formulas in $\mathcal{S}$CIFF

	formula		$\mathcal{S}$CIFF Integrity Constraint
C7	responded existence		$\mathbf{H}(refuseShipment(W,S),T_s)$
		$\rightarrow$	$\mathbf{E}(refuseOrder(S,C),T_r)$.
C4	response		$\mathbf{H}(commitOrder(C,S),T_c)$
		$\rightarrow$	$\mathbf{E}(refuseOrder(S,C),T_r) \wedge T_r > T_c$
		$\vee$	$\mathbf{E}(acceptOrder(S,C),T_a) \wedge T_a > T_c$
C9	precedence		$\mathbf{H}(refuseOrder(S,C),T_r)$
		$\rightarrow$	$\mathbf{E}(commitOrder(C,S),T_c) \wedge T_c < T_r$

responded existence, response and precedence constraints are represented in $\mathcal{S}$CIFF by using similar rules, different only by the involved time constraints; this is pointed out in Table 5, where C, S and W respectively identify Customer, Seller and Warehouse.[4] Constraint C_4 in Table 5 also shows that branching formulas are mapped to rules containing a disjunction in the head.

Negation formulas are mapped exactly in the same way, but by using negative expectations instead of positive ones. For example, constraint C_{13}, which is a negation response stating that after having refused an order it is not possible to confirm it, is represented as a response, but by forbidding the target activity:

$$\mathbf{H}(refuseOrder(S,C),T_r) \rightarrow \mathbf{EN}(confirmOrder(S,C),Tc) \wedge T_c > T_r.$$

Some DecSerFlow formulas, namely the choice and existence N ones, are translated to $\mathcal{S}$CIFF in a slight different way. In particular, their mapping do not have a triggering part but simply generates a set of expectations. Therefore, they define, in some sense, the initial goal of the choreography: the corresponding expectations are generated independently from the interaction. For example, the choice constraint C_{16} of Example 1 states that, in any case, the order must be either canceled or committed by the customer, and is mapped to the following Integrity Constraint:

$$\rightarrow \mathbf{E}(cancelOrder(C,S),T_{canc}) \vee \mathbf{E}(confirmOrder(C,S),T_{conf}).$$

5.3. Modeling social commitments in $\mathcal{S}$CIFF

To integrate social commitments in $\mathcal{S}$CIFF, we rely on the formalization given by Singh and Yolum in [38], which is based on Event Calculus ($\mathcal{EC}$).

We start from showing how $\mathcal{EC}$ can be expressed in $\mathcal{S}$CIFF in an intuitive and elegant way. Being this calculus expressive enough to model the commitments theory, we can then build upon the previous results. The basic predicates of the calculus here proposed are listed in Table 4, where events and fluents are terms and times are integer (CLP) variables.

The main distinctive feature of our implementation is that it is reactive: fluents are initiated and terminated by dynamically occurring events. Therefore, we will call the proposed axiomatization "reactive Event Calculus" ($\mathcal{REC}$). In the classical $\mathcal{EC}$ setting two main reasoning tasks can be carried out: *narrative verification* [21], exploiting the calculus in a deductive manner, to check whether a certain fluent holds given a set of facts, and *planning* [33], using abduction to simulate narratives of the systems, trying to

[4]For the sake of simplicity, we do not report in the formalization the management of roles, which can be seamlessly added as in commitments (see Section 4.3).

produce a possible execution which satisfies the requirements. Such verifications are respectively carried out a posteriori (after execution), and a priori (before execution). The use of $\mathcal{EC}$ to monitor an ongoing execution, and to check if it complies with the choreography (run-time monitoring and verification), has been little exploited so far, mainly due to a lack of suitable underlying reasoning tools.

Our axiomatization is inspired by the *Cached Event Calculus* ($\mathcal{CEC}$) of Chittaro and Montanari [12]. The main distinctive feature of $\mathcal{CEC}$ is that it "caches" the outcome of the reasoning process carried out until the current time. In particular, $\mathcal{CEC}$ computes and stores fluents' *maximum validity intervals*, which are the maximum intervals in which fluents hold, according to the already acquired events. The set of cached validity intervals is then extended/revised as new events occur.

While in [12] the authors argue that such a caching provides a great improvement w.r.t. the computational complexity of the calculus, we exploit it to provide run-time monitoring and verification capabilities. The main distinctive feature of our axiomatization is that the holds/3 predicate, which is used to represent maximum validity intervals, is treated as abducible; $\mathcal{S}$CIFF is therefore employed for reacting to events which initiate or terminate fluents, by coherently generating the corresponding holds predicates and handling their extreme time points.

Since the concept of happened event is already present in $\mathcal{S}$CIFF as a first-class object, the happens/2 predicates is simply mapped as follows:

$$happens(Ev, T) \leftarrow \mathbf{H}(Ev, T). \tag{1}$$

We make use of two special events *clip*(*Fluent*) and *declip*(*Fluent*), which are internally generated events, not explicitly contained in the execution trace under study. They model the clipping and declipping of fluents, and are used to suitably characterize the notion of maximum validity interval, as shown in the two following axioms.

Axiom 1. (*Extreme points consistency*) *The start and completion time of a maximum validity interval are associated to the declipping and clipping of the corresponding fluent. If fluent F holds starting from time T_i, then at time T_i it has been declipped:*

$$\mathbf{holds}(T_i, F, T_f) \rightarrow \mathbf{E}(declip(F), T_i). \tag{2}$$

If fluent F holds until time T_f, then at time T_f it will be clipped:

$$\mathbf{holds}(T_i, F, T_f) \rightarrow \mathbf{E}(clip(F), T_f). \tag{3}$$

Axiom 2. (*Persistence consistency*) *If a fluent F holds inside a time interval which spans from time T_i to time T_f, then it is not possible to clip F between T_i and T_f.*

$$\mathbf{holds}(T_i, F, T_f) \rightarrow \mathbf{EN}(clip(F), T) \wedge T > T_i \wedge T < T_f. \tag{4}$$

Thanks to these axioms, the start time T_i of holds is properly connected to a clipping event, and the completion time T_f is bound to the nearest time, after T_i, at which the fluent is declipped.

We now capture the situation in which a certain fluent is declipped, consequently starting to hold. Two different cases may arise. The first one deals with the initial status of fluents.

Axiom 3. (*Initial status of fluents*) *If fluent F initially holds, a corresponding declipping* (*clipping resp.*) *event is generated at the initial time* (*i.e., at time point 0*).

$$initially(F) \rightarrow \mathbf{H}(declip(F), 0). \tag{5}$$

The second one deals with the case in which an event capable of initiating a certain fluent indeed happens.

Axiom 4. (*Fluents initiation*) *When an event Ev capable to initiate a fluent F happens at time T, and at this time F does not hold, then F is declipped.*

$$\mathbf{H}(Ev, T) \wedge not_holdsat(F, T) \wedge initiates(Ev, F, T) \rightarrow \mathbf{H}(declip(F), T). \quad (6)$$

As in the case of fluents initiation, two situations may arise when dealing with the clipping of fluents. The first situation is the one in which an event terminating a certain fluent occurs.

Axiom 5. (*Fluents termination*) *When an event Ev capable to terminate a fluent F happens at time T, and at this time F holds, then F is clipped.*

$$\mathbf{H}(Ev, T) \wedge holdsat(F, T) \wedge terminates(Ev, F, T) \rightarrow \mathbf{H}(clip(F), T). \quad (7)$$

The other situation deals with the final status of fluents. In $\mathcal{S}$CIFF, a special *complete* event is used to state that the execution is completed and no further event will occur. Such a special event is used by $\mathcal{S}$CIFF to evaluate still pending expectations,[5] deciding if they are violated or not; for example, a positive expectation which has not yet been fulfilled by a corresponding happening event is considered violated. W.r.t. $\mathcal{REC}$, we use the *complete* event to clip all the fluents which still hold (indeed, all these fluents still have a corresponding expectation about a clipping event).

Axiom 6. (*Final clipping of fluents*) *All fluents are terminated by the special complete event.*

$$terminates(complete, F). \quad (8)$$

The last axiom defines how the happening of a *declip* event affects the corresponding fluent (provided that such an event has been generated before the *complete*), abducing that the fluent begins to hold.

Axiom 7. (*Events effects*) *The happening of a declip(F) event causes fluent F starting to hold from time T.*

$$\mathbf{H}(declip(F), T) \wedge T < T_\infty \rightarrow \mathbf{holds}(T, F, T_f) \wedge T_f > T. \quad (9)$$

where T_∞ is the time at which the complete event happens.

To integrate social commitments and DecSerFlow by exploiting $\mathcal{S}$CIFF, it is now sufficient to define in the $\mathcal{S}$CIFF Knowledge Base the concrete commitments involved in the choreography under study, deciding how to link their corresponding operations to specific events; in our running example, these are the clauses introduced in Sections 4.2 and 4.3.

[5] An expectation is *pending* if it has not yet been fulfilled. Positive expectations are fulfilled when a corresponding matching event occurs, whereas negative expectations are fulfilled when no corresponding matching event happens.

5.4. Choreographies verification with $\mathcal{S}$CIFF

Having shown how both DecSerFlow and commitments can be modeled inside the $\mathcal{S}$CIFF framework, we now sketch its verification capabilities, which range from compliance checking at run-time to static detection of modeling conflicts. We also show how $\mathcal{S}$CIFF is able to monitor the trace produced by an instance of the choreography described in Example 1.

Compliance verification. Compliance verification aims at verifying whether a set of concrete services interact without undermining the choreography rules of engagement (and by correctly handling the involved commitments). The $\mathcal{S}$CIFF proof procedure [2] is able to verify compliance *at run-time*, by monitoring exchanged messages, or *a-posteriori*, by checking already completed execution traces. Roughly speaking, compliance is verified by means of the following steps: (i) occurring events are collected; (ii) choreography rules are applied to generate expectations; (iii) expectations are verified w.r.t. the actual behavior, checking whether they are actually *fulfilled* by the interacting entitites. Fulfillment states that each positive (negative resp.) expectation should (should not resp.) have a corresponding happened event. If it is not the case, the expectation is considered as violated.

Static verification. Static verification is an a-priori reasoning process which aims to check the consistency of a choreography. Following the work of Pesic et al. [29], we mainly tackle two kinds of consistency verification: *conflicts detection*, which aims to check consistency of the whole model, i.e. to ensure that at least one execution is accepted; *discovery of dead activities*, which aims to find local inconsistencies inside a model, namely to discover the set of activities that cannot be executed at all. In order to detect conflicts, a generative version of $\mathcal{S}$CIFF, called g-$\mathcal{S}$CIFF, is exploited; g-$\mathcal{S}$CIFF is capable to automatically transform expectations into happened events, trying to simulate execution traces which satisfy model's constraints.[6]

Interoperability verification. Interoperability verification mainly focus on evaluating whether a concrete local behavioural interface can play a role in a choreography without undermining local and global rules of engagement. Because DecSerFlow models are declarative and loosely-coupled, the only form of interoperability currently addressed is an *existential* one. That is, a global and a local model are composed by simply joining their constraints and then g-$\mathcal{S}$CIFF is used to detect conflicts and dead activities on the overall model. If such an overall model is conflict-free, then at least one execution exists such that both the choreography and service constraints are satisfied.

Choreography identification with process mining. Following the work presented in [22], it is possible to mine a DecSerFlow choreography specification starting from a set of already collected execution traces, previously labeled as compliant or not. This is particularly useful in situations where collaborating processes already exist and where different partners have a local perception of the interaction, but they want to discover a global view of the collaboration. This task is identified in [15] as *choreography identification*.

An Example: run-time compliance verification with $\mathcal{S}$CIFF. Thanks to the integration of social commitments and DecSerFlow presented in this paper, we can exploit the $\mathcal{S}$CIFF proof procedure not only to verify compliance of interacting services w.r.t. a DecSerFlow choreography, but also to monitor the status of involved commitments.

Run-time verification is a fundamental aspect in the context of service choreographies. Indeed, in a dynamic and open environment interacting services may not be known in advance. Even if the behavioural interface of a service is known, verifying (at static time) that it conforms to the choreography specification

[6] For more information, consult http://www.lia.deis.unibo.it/research/climb.

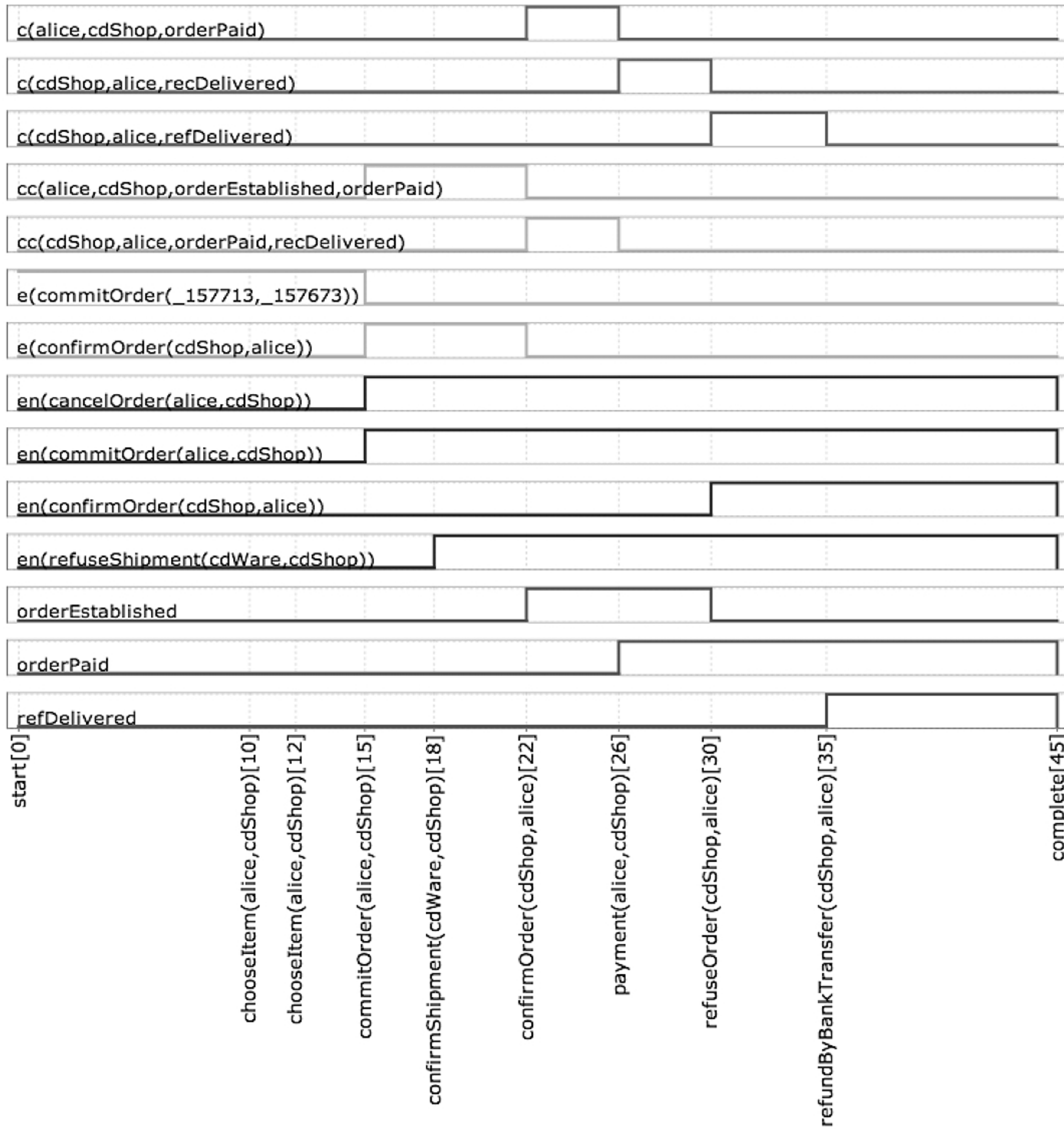

Fig. 4. Report produced by $\mathcal{S}$CIFF when monitoring an instance of the choreography described in Example 1; the monitored trace is shown in the bottom part of the figure.

does not guarantee that the real implementation of the service will behave correctly: services could be hostile or there could be a mismatch between the behavioural interface and its underlying implementation.

Figure 4 shows the result of the compliance verification carried out by $\mathcal{S}$CIFF on a specific instance of Example 1. $\mathcal{REC}$ and the commitments theory proposed in [38] have been fully implemented in $\mathcal{S}$CIFF; therefore, to model the commitments involved in the running example it is sufficient to specify the clauses described in Sections 4.2 and 4.3 and declare which concrete services play the three choreography roles. In the considered instance, we have $role(alice, customer)$, $role(cdShop, seller)$

and $role(cdWare, warehouse)$. The $\mathcal{S}$CIFF translation of the DecSerFlow choreography shown in Fig. 3 has been instead produced manually following the mapping presented in [28]; we are working on a model-driven approach which exploits *model transformation* techniques to automatically generate a $\mathcal{S}$CIFF specification starting from a graphical DecSerFlow diagram.

During the run-time verification, $\mathcal{S}$CIFF generates an XML report having one entry for each acquired happened event. Each entry stores the list of currently holding fluents and currently pending expectations. The XML report is then automatically analyzed by a JAVA program which produces a graphical chart summarizing fluents' and expectations' trends. $\mathcal{S}$CIFF required 720 ms to verify the considered instance (and produce the corresponding XML report).

The top part of Fig. 4 shows the evolution of commitments (conditional commitments are depicted in blue, and base-level commitments are depicted in light blue). Their trend is in turn determined by the evolution of fluents representing the effects of activities; they are shown in the bottom part (in violet). In the considered instance, *alice* sends an order to *cdShop*, executing the payment after having received a confirmation. As soon as the payment is emitted (i.e., at time 26), the conditional commitment relating *cdShop* with the delivery of the receipt becomes a base-level commitment. However, *cdShop* decides to refuse, at time 30, the previously confirmed order: this causes the base-level commitment about the receipt delivery to be canceled, and a corresponding compensating commitment to be created. This new commitment states that *cdShop* must refund *alice*, and it is correctly discharged at time 35. At the end of the choreography, all fluents used to represent base-level commitments do not hold, attesting that they have been correctly handled.

The center part of Fig. 4 is instead focused on the positive (green color) and negative (red color) expectations generated by the $\mathcal{S}$CIFF rules which formalizes the DecSerFlow choreography. The level of each expectation is "high" when the expectation is pending. Pending positive expectations are waiting for the occurrence of a corresponding matching event. This is why only two expectations are listed in the figure: they correspond to the positive DecSerFlow constraints referring to the future (C_4 and C_{16}); expectations related to constraints referring to the past are evaluated as soon as the constraint triggers, and are therefore never pending. It is worth noting that C_{16} does not fix the sender and receiver of the commitOrder activity: it simply states that someone playing the customer role must commit or cancel an order made up by interacting with some seller. C_4 instead put an expectation about the confirmation or refusal of the order grounding the involved parties to *cdShop* and *alice*; indeed, it states that if a customer C commits an order to a seller S, then that S must send back a corresponding answer to C.

Finally, pending negative expectations forbids the presence of any matching event; for example, when *alice* commits her order, constraints C_2 and C_{15} trigger, forbidding further executions of the commitOrder activity and forbidding the possibility of canceling the order.

6. Related works

Because the procedural nature of modeling languages can be, in some cases, an obstacle in developing choreographies and Multi-Agent Systems, in this work we have instead discussed and compared alternative approaches advocating the use of declarative approach. In this section, we briefly summarizes related works both in the field of service choreographies and Multi-Agent Systems, restricting our attention to the recently proposed approaches that advocate the use of declarative specifications.

The need for an alternative to procedural languages such as the Web Services Choreography Description Language (WS-CDL) [36] has been identified in "Let's Dance" [39,40] which is a language for modeling service interactions and their flow dependencies. Let's Dance focuses on interaction patterns and

mechanisms, while DecSerFlow mainly deals with the process perspective. In [32] the authors use an extension of the Event Calculus of Kowalski and Sergot [21] to declaratively model event based requirements specifications. In [6], the authors propose the use of a declarative, logic-based language to enrich the description of choreography specifications.

In [31], the authors tackle the problem of achieving knowledge sharing in open and large scale distributed systems. They propose a framework in which services share and execute *interaction models* specified by means of LCC, a declarative lightweight coordination calculus which presents many similarities with $\mathcal{S}$CIFF.

In the field of Multi-Agent Systems, some approaches consider MAS as open societies and model interaction protocols by declaratively constraining the possible interactions. For example, in [18] the semantics of communicative acts is defined by means of transitions on a finite state automaton which describes the concept of commitment; in our work, based on [38], we adopt instead a variant of Event Calculus to model and manipulate commitments. Here, commitments evolve in relation to events and fluents and the semantics of messages is given in terms of predicates on such events and fluents. Our formalization, thanks to the exploitation of reactivity and automatic compliance cheking of $\mathcal{S}$CIFF, is particularly suitable for monitoring and verifying such commitments.

With respect to social constraints and roles, in [4] a framework based on (Logical) Constraints, specified by using Event Calculus, is presented. $\mathcal{S}$CIFF instead captures constraints by means of forward rules, modeling time aspects with suitable constraints on finite domains, and grounds such rules on an abductive framework, where hypotheses (i.e. expectations) can be confirmed (fulfilled) or disconfirmed (violated) by the occurring events. However, in this work we have shown that Event Calculus can be suitably encoded in $\mathcal{S}$CIFF; in this way, we equip $\mathcal{S}$CIFF with a more expressive form of temporal reasoning, by exploiting at the same its abductive and reactive nature.

Recent works have pointed out the importance of adopting agent-oriented methodologies also for software engineering, with particular emphasis on requirements elicitation. One of the leading proposals in this respect is Tropos [9]. It is worth noting that declarative approaches based on social semantics have been proposed also as a way to enrich, formalize and verify Tropos models. For example, in [25] Mallya and Singh show how commitments protocols can be seamlessly introduced within the Tropos methodology, while in [10] the authors propose a mapping from Tropos to $\mathcal{S}$CIFF, to the aim of extending Tropos models with business process-oriented aspects and of realizing different forms of verification. It would be interesting to pursue this research activity in the context of service choreographies, developing an integrated methodology covering constraints on the execution flow (DecSerFlow), mutual obligations and commitments and their underlying organizational structure (Tropos).

7. Conclusions

In this work, we have proposed a conjunct use of declarative approaches coming from the Service Oriented Computing and Multi-Agent Systems research areas, to the aim of specifying and verifying service choreographies. In particular, we have chosen the DecSerFlow graphical language to declaratively constrain the choreography flow, and social commitments to capture mutual obligations between interacting roles. By means of a running example, we have shown the need of exploiting both aspects to properly deal with the choreography modeling process, pointing out their complementarity and synergies.

We have then proposed $\mathcal{S}$CIFF as an underlying integrated framework, enabling different verification capabilities. We have sketched how the different DecSerFlow concepts can be automatically mapped to $\mathcal{S}$CIFF Integrity Constraints. Furthermore, by providing a reactive form of Event Calculus in $\mathcal{S}$CIFF, we

have demonstrated how commitments can be easily accomodated into the framework, making possible to link activities with their effects and to handle exceptional situations/compensations in a flexible way.

We believe that the integration proposed in this work further attests that the Multi-Agent Systems and the Service Oriented Computing settings are closely related and can benefit from each other.

Starting from this work, we foresee several research directions:

– Automatizing the mapping of DecSerFlow to $\mathcal{S}$CIFF by means of model transformation techniques, enabling the possibility of editing DecSerFlow models, transparently obtaining the underlying formalization. A related investigation will study the properties of the mapping; indeed, each DecSerFlow constraint can be translated into $\mathcal{S}$CIFF in different ways [11], and different dimensions can be used to evaluate the "quality" of the mapping (e.g., what is the impact on the verification tasks? Is the mapping concise and readable?).
– Enhancing the commitments theory by considering an extended framework which incorporate also deadlines [26]; in this way, the capability of $\mathcal{S}$CIFF to express quantitative time constraints will be deeply exploited.
– Comparing $\mathcal{S}$CIFF and other approaches based on temporal logics. A first comparison between $\mathcal{S}$CIFF and explicit/symbolic model checking when performing static verification of DecSerFlow models is presented in [27]; we are extending the comparisons by taking into account also timed temporal logics (which enable the possibility of expressing quantitative time constraints) and SAT-based model checkers such as ZOT [30].
– Investigating the use of Event Calculus and $\mathcal{S}$CIFF for verifying WS-CDL and WS-BPEL models; indeed, Event Calculus has been exploited to express also procedural specifications [14].

Acknowledgments

We would like to thank the anonymous reviewers of this Special Issue and of the MALLOW-AWESOME'007 workshop for their valuable comments and suggestions. Furthermore, we would like to thank all the workshop participants for the fruitful and interesting discussions. This work has been partially supported by the FIRB project TOCAI.IT.

References

[1] Web services business process execution language version 2.0. OASIS Consortium, http://www.oasis-open.org, April 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf.

[2] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello and P. Torroni, Verifiable agent interaction in abductive logic programming: the SCIFF framework, *ACM Transactions on Computational Logic* **9**(4), 2008.

[3] M. Alberti, M. Gavanelli, E. Lamma, P. Mello, G. Sartor and P. Torroni, Mapping deontic operators to abductive expectations, *Computational and Mathematical Organization Theory* **12**(2–3) (October 2006), 205–225.

[4] A. Artikis, J. Pitt and M. Sergot, Animated specifications of computational societies. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, New York, NY, USA, ACM, 2002, pages 1053–1061.

[5] M. Baldoni, C. Baroglio, A. Martelli, V. Patti and C. Schifanella, Verifying the conformance of web services to global interaction protocols: A first step, in: *International Workshop on Web Services and Formal Methods, WS-FM 2005, Versailles, France, 1–3 September 2005, Proceedings*, M. Bravetti, L. Kloul and G. Zavattaro, eds, volume 3670 of *Lecture Notes in Computer Science*, Springer Verlag, 2005, pp. 257–271.

[6] M. Baldoni, C. Baroglio, A. Martelli, V. Patti and C. Schifanella, Reasoning on choreographies and capability requirements, *International Journal of Business Process Integration and Management* **2**(4) (2007), Inderscience.

[7] A. Barros, M. Dumas and P. Oaks, A critical overview of the web services choreography description language (WS-CDL), *BPTrends*, 2005.

[8] B. Bauer, J.P. Müller and J. Odell, Agent uml: a formalism for specifying multiagent software systems. In *Proceedings of the First international workshop, AOSE 2000 on Agent-oriented software engineering*, Springer Verlag, 2001, pages 91–103.

[9] P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos and A. Perini, TROPOS: An Agent-Oriented Software Development Methodology. *Journal of Autonomous Agents and Multi-Agent Systems*, **8**(3) (2004), 203–236.

[10] V. Bryl, P. Mello, M. Montali, P. Torroni and N. Zannone, B-Tropos: Agent-oriented requirements engineering meets computational logic for declarative business process modeling and verification. In *Proceedings of CLIMA-VIII, selected and revised papers*, Lecture Notes on Artificial Intelligence. Springer Verlag, 2008. To appear.

[11] F. Chesani, P. Mello, M. Montali and S. Storari, Towards a decserflow declarative semantics based on computational logic. Technical Report DEIS-LIA-07-002, DEIS, Bologna, Italy, 2007.

[12] L. Chittaro and A. Montanari, Efficient temporal reasoning in the cached event calculus, *Computational Intelligence* **12** (1996), 359–382.

[13] Amit K. Chopra and Munindar P. Singh. Producing compliant interactions: Conformance, coverage, and interoperability, in: *DALT*, M. Baldoni and U. Endriss, eds, volume 4327 of *Lecture Notes in Computer Science*, Springer Verlag, 2006, pp. 1–15.

[14] N.K. Cicekli and I. Cicekli, Formalizing the specification and execution of workflows using the event calculus, *Information Sciences* **176**(15) (2006), 2227–2267.

[15] G. Decker and M. Von Riegen, Scenarios and techniques for choreography design, in: *Business Information Systems, 10th International Conference, BIS 2007, Poznan, Poland, 25–27 April 2007, Proceedings*, W. Abramowicz, ed., volume 4439 of *Lecture Notes in Computer Science*, Springer Verlag, 2007, 121–132.

[16] W.M.P. Van der Aalst and M. Pesic, DecSerFlow: Towards a truly declarative service flow language. In *WS-FM'06*, volume 4184 of *Lecture Notes in Computer Science*, Springer Verlag, 2006.

[17] N. Desai, A.K. Chopra and M.P. Singh, Business process adaptations via protocols. In *2006 IEEE International Conference on Services Computing* (*SCC 2006*)*, 18–22 September 2006, Chicago, Illinois, USA*, IEEE Computer Society, 2006, pages 103–110.

[18] N. Fornara and M. Colombetti, Operational specification of a commitment-based agent communication language, in: *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems* (*AAMAS-2002*)*, Part II*, C. Castelfranchi and W. Lewis Johnson, eds, Bologna, Italy, ACM Press, 15–19 July 2002, pp. 535–542.

[19] M.N. Huhns and M.P. Singh, Service-oriented computing: Key concepts and principles, *IEEE Internet Computing* **9**(1) (2005), 75–81.

[20] A.C. Kakas, R.A. Kowalski and F. Toni, Abductive Logic Programming, *Journal of Logic and Computation* **2**(6) (1993), 719–770.

[21] R.A. Kowalski and M. Sergot, A logic-based calculus of events, *New Generation Computing* **4**(1) (1986), 67–95.

[22] E. Lamma, P. Mello, M. Montali, F. Riguzzi and S. Storari, Inducing declarative logic-based models from labeled traces, in: *Proceedings of the 5th International Conference on Business Process Management* (*BPM 2007*), M. Rosemann and M. Dumas, eds, volume 4714 of *Lecture Notes in Computer Science*, Brisbane, Australia, Springer Verlag, 24–27 September 2007, pages 344–359.

[23] A.U. Mallya, N. Desai, A.K. Chopra and M.P. Singh, Owl-p: Owl for protocol and processes, in: *4rd International Joint Conference on Autonomous Agents and Multiagent Systems* (*AAMAS 2005*)*, 25–29 July 2005, Utrecht, The Netherlands*, F. Dignum, V. Dignum, S. Koenig, S. Kraus, M.P. Singh and M. Wooldridge, eds, ACM, 2005, pp. 139–140.

[24] A.U. Mallya and M.P. Singh, Modeling exceptions via commitment protocols, in: *4rd International Joint Conference on Autonomous Agents and Multiagent Systems* (*AAMAS 2005*)*, 25–29 July 2005, Utrecht, The Netherlands*, F. Dignum, V. Dignum, S. Koenig, S. Kraus, M.P. Singh and M. Wooldridge, eds, ACM, 2005, pp. 122–129.

[25] A.U. Mallya and M.P. Singh, Incorporating commitment protocols into tropos, in: *Agent-Oriented Software Engineering VI, 6th International Workshop, AOSE 2005, Utrecht, The Netherlands, 25 July 2005. Revised and Invited Papers*, J.P. Müller and F. Zambonelli, eds, volume 3950 of *Lecture Notes in Computer Science*, Springer Verlag, 2006, pp. 69–80.

[26] A.U. Mallya, P. Yolum and M.P. Singh, Resolving commitments among autonomous agents, in: *Advances in Agent Communication, International Workshop on Agent Communication Languages, ACL 2003, Melbourne, Australia, 14 July 2003*, F. Dignum, ed., volume 2922 of *Lecture Notes in Computer Science*, Springer Verlag, 2003, pp. 166–182.

[27] M. Montali, M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello and P. Torroni, Verification from declarative specifications. In *Proceedings of the 24th International Conference on Logic Programming* (*ICLP 2008*), to appear.

[28] M. Montali, W.M.P. van der Aalst, M. Pesic, F. Chesani, P. Mello and S. Storari, Declarative specification and verification of service choreographies. *ACM Transactions on the Web*, Submitted.

[29] M. Pesic, H. Schonenberg and W.M.P. van der Aalst, Declare: Full support for loosely-structured processes. In *11th IEEE International Enterprise Distributed Object Computing Conference* (*EDOC 2007*)*, 15–19 October 2007, Annapolis, Maryland, USA*, IEEE Computer Society, 2007, pages 287–300.

[30] M. Pradella, C.A. Furia, A. Morzenti and P. San Pietro, Zot, a flexible bounded model and satisfiability checker. In *15th International Symposium on Formal Methods* (*FM 2008*), 2008. Posters and Research Tools.

[31] D. Robertson, A. Barker, P. Besana, A. Bundy, Y.-H. Chen-Burger, D. Dupplaw, F. Giunchiglia, F. van Harmelen, F. Hassan, S. Kotoulas, D. Lambert, G. Li, J. McGinnis, F. McNeill and N. Osman, Adrian Perreau de Pinninck Bas, Ronny Siebes, Carles Sierra, and Chris Walton. Models of interaction as a grounding for peer-to-peer knowledge sharing, *LNCS Advances in Web Semantics*, 2007.

[32] M. Rouached, O. Perrin and C. Godart, Towards formal verification of web service composition. In *4th International Conference on Business Process Management*, volume 4102 of *Lecture Notes in Computer Science*, Springer Verlag, 2006, pages 257–273.

[33] M. Shanahan, An abductive event calculus planner, *Journal of Logic Programming* **44**(1–3) (2000), 207–240.

[34] M.P. Singh, An ontology for commitments in multiagent systems: Toward a unification of normative concepts, *Artificial Intelligence and Law* (7) (1999), 97–113.

[35] 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: *International Workshop on Web Services and Formal Methods, WS-FM 2005, Versailles, France, 1–3 September 2005, Proceedings*, M. Bravetti, L. Kloul and G. Zavattaro, eds, volume 3670 of *Lecture Notes in Computer Science*, Springer Verlag, 2005, pp. 35–50.

[36] W3C, Web services choreography description language version 1.0.

[37] S.A. White, Business process modeling notation specification, Technical report, OMG, 2006.

[38] P. Yolum and M.P. Singh, Flexible protocol specification and execution: applying event calculus planning using commitments. In *Proceedings of the First International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2002, 15–19 July 2002, Bologna, Italy*, ACM, 2002, pages 527–534.

[39] J.M. Zaha, A. Barros, M. Dumas and A.H.M. ter Hofstede, Let's dance: A language for service behavior modeling. QUT ePrints 4468, Faculty of Information Technology, Queensland University of Technology, 2006.

[40] J.M. Zaha, M. Dumas, A.H.M. ter Hofstede, A. Barros and G. Dekker, Service Interaction Modeling: Bridging Global and Local Views, QUT ePrints 4032, Faculty of Information Technology, Queensland University of Technology, 2006.

Authors' Bios

Dr. Federico Chesani is a post-doc at the DEIS department. He got his PhD in 2007, with a thesis entitled "Specification, Execution and Verification of Interaction Protocols". His research topics concern computational logic and abduction, argumentation, and applications of logic-based tools to Semantic Web Services, Service Discovery, Composition and Contracting, commitment-based multi-agent systems, integration between rule-based languages and ontologies. He has been author and co-author of more than 50 refereed papers, and has been also involved as a reviewer for international conferences and workshops such as CLIMA and IJCAI. He is also member of the Logic Programming National Interest Group (GULP).

Prof. Dr. Paola Mello is Full Professor in AI since 1994. She has published over 180 papers on implementation, application and theoretical aspects of programming languages, especially logic languages and their extensions towards modular and object-oriented programming, artificial intelligence, knowledge representation, expert systems and multi-agent systems, and the application of computational logic to workflow patterns and Web Service composition. She is founding member of the Italian Association for Artificial Intelligence and was executive member of the Logic Programming National Interest Group (GULP) and of several national and international (UE) research projects, including COMPUNET, Esprit ALPES, CRAFT, IST-FET SOCS.

Dr. Marco Montali is currently a post-doc researcher in the Artificial Intelligence group at DEIS. He received his PhD in 2009 from the University of Bologna, with a thesis entitled "Specification and Verification of Open Declarative Interaction Models – A Logic-Based Approac". During his PhD, he investigated an abductive logic programming framework for the specification and verification of declarative interaction models, studying its application in the context of Business Processes, Clinical Guidelines, Service Choreographies and Multiagent Systems. His dissertation received the 'Marco Cadoli' prize, awarded by the Italian Association on Logic Programming (GULP) for the best two theses focused on Computational Logic and discussed between 2007 and 2009. He is co-author of more than 40

papers focusing on computational logics and extensions, formal verification and monitoring, (declarative) business process modeling and business rules, service choreographies, clinical guidelines and care-flow protocols, process mining, commitment-based multiagent systems.

Dr. Sergio Storari He worked as a Technician at the Department of Engineering of the University of Ferrara. He graduated in Electrical Engineering at the University of Ferrara in 1998 and attained his Ph.D. from the University of Bologna in 2004. His research activity centers on artificial intelligence, knowledge-based systems, data mining and multi-agent systems. He is member of the Italian Association for Artificial Intelligence (AI*IA), associated with ECCAI.

Assist. Prof. Dr. Paolo Torroni is an assistant professor in Computing since 2004. He has published more than 50 refereed articles on theoretical aspects and implementation of declarative languages, computational logic-based languages for the specification and verification of multi-agent interaction, argumentation, web services, negotiation, dialogue, resource exchange and allocation, and planning. He was actively involved in several national and international (UE) research projects mainly about computational logic and multi-agent systems. In recent years, he has been involved in the organization of international events aimed at promoting declarative technologies for agents, and he has given conference tutorials on these topics. He is secretary of the Logic Programming National Interest Group (GULP), founded in 1987, counting 100 members to date. He is Steering Committee member of several international events like CLIMA, DALT and MALLOW, and since 2003 he has co-organized 7 international workshops and co-edited 5 books for Springer on topics related to the project.