Agent Societies and Service Choreographies: a Declarative Approach to Specification and Verification

Federico Chesani[1], Paola Mello[1], Marco Montali[1], and Sergio Storari[2]

[1] DEIS, University of Bologna - Viale Risorgimento 2 - 40136 Bologna, Italy
{fchesani|pmello|mmontali}@deis.unibo.it
[2] ENDIF, University of Ferrara - Via Saragat 1 - 44100 Ferrara, Italy
strsrg@unife.it

Abstract. The need for specifying choreographies when developing service oriented systems recently arose as an important issue. Although declarativeness has been identified as a key feature, several proposed approaches model choreographies by focusing on procedural aspects, e.g. by specifying control and message flows of the interacting services. A similar issue has been addressed in Multi-Agent Systems (MAS), where declarative approaches based on social semantics have been used to capture the nature of agents interaction without over-constraining their behavior.
In this paper we show how DecSerFlow can be mapped to $\mathcal{S}$CIFF in an automatic and complete way. DecSerFlow is a graphical language capable to model in an intuitive and declarative fashion service flows, whereas $\mathcal{S}$CIFF is a framework based on abductive logic programming originally developed for dealing with social interactions in MAS. By means of a running example, we show how the conjunct use of both approaches could be fruitfully exploited to declaratively specify and verify service choreographies.

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 [1]. Indeed, the need for specifying choreographies when developing service oriented systems recently arose as an important issue.

As pointed out in [1, 2], the current major proposals for modeling service interaction, such as WS-BPEL [3] and WS-CDL [4], miss to tackle some key concepts. As a consequence of the adoption of a "global view" (which inherently crosses organizational boundaries and should be consequently independent from the perspective of single participants), declarativeness becomes a fundamental requirement. Each organization perceives a choreography as a public contract which provides the rules of engagement for making all the interacting parties correctly collaborate, without stating how such a collaboration is concretely carried out; in our view, this latter information should be kept private in the entities' definition/implementation, and not directly addressed at the choreography level.

The main problem is that, although declarativeness has been identified as a key feature, several proposed approaches model choreographies by focusing on procedural aspects, e.g. by specifying the control and message flow of the interacting services. This often causes the modeler to miss the real focus of the choreography, leading to over-constrain the choreography under study and to consequently loose some acceptable interactions.

To overcome these limits, van der Aalst and Pesic have proposed DecSerFlow [5], a truly declarative graphical language for the specification of service flows. DecSerFlow adopts a more general and high-level view of services specification, by defining them through a set of policies or business rules. It does not give a complete and procedural specification of services, but concentrates on what is the (minimal) set of constraints to be fulfilled in order to successfully accomplish the interaction. Beyond its appealing graphical representation, DecSerFlow concepts have an underlying semantics in terms of Linear Temporal Logic (LTL).

The issue about what information should be captured or left out by the global view of interaction has been (and is still) matter of discussion also in the MAS research community, and in both settings we find similar efforts and proposed solutions. Therefore, it is not surprising that multi-agent and service-oriented systems share many similarities [6] (see Table 1).

	MAS	SOA
interacting agents	autonomous heterogeneous agents	autonomous heterougeneous services
communication	communicative acts	messages
local view of interaction	(external) agents policies	behavioral interfaces
global view of interaction	global interaction protocols	choreographies

Table 1. Some similarities between multi-agent and service-oriented systems

When dealing with the problem of modeling global interaction protocols within a MAS, we mainly find two complementary approaches, as in the case of choreographies: approaches with aim to exactly specify how the interaction protocol should be executed by the interacting agents (such as for example AUML [7]), and approaches which consider MAS as open societies and model interaction protocols as a way to declaratively constrain the possible interactions. Social approaches abstract away from the nature of interacting entities, supporting heterogeneity, and adopt an open perspective, i.e. let participants autonomously behave as they want, where not explicitly forbidden. Furthermore, their aim is not only to support the specification task, but also to define a precise semantics of interaction, enabling the possibility to perform verification tasks. Many prominent works center around the concept of commitment in social agencies, to represent the state of affairs during the social interaction. For example, in

[8] the semantics of communicative acts is defined by means of transitions on a finite state automaton which describes the concept of commitment; in [9], the authors adopts a variant of Event Calculus to commitment-based protocols, where commitments evolve in relation to events and fluents and the semantics of messages is given in terms of predicates on such events and fluents (to describe how messages affect commitments). In the last years, Singh et al. have applied the concept of commitment-based protocols in the context of the Service Oriented Architecture and Business Process Management, by addressing the problem of business process adaptability [10] and of protocols composition [11]. The idea of taking social semantics from the MAS world and applying it to the specification of service choreographies has been adopted also in [12], although the focus is more on the procedural aspects, rather than on the declarative ones.

Within the SOCS EU Project [3] we have developed a language, called $\mathcal{S}$CIFF, for specifying global interactions protocols in open agent societies, giving its declarative semantics in terms of Abductive Logic Programming (ALP) [13]. Furthermore, we have equipped the $\mathcal{S}$CIFF language with a corresponding proof procedure, capable to verify at run-time (or a posteriori, by analizing a log of the interaction) whether interacting agents behave in a conformant manner w.r.t. the modeled interaction protocol. Protocols are specified only by considering the external observable behavior of interacting entities (i.e. the different observable events which occurred during the interaction), and by the concept of expectation about desired events and interactions; occurred events and positive/negative expectations are linked by means of forward rules called Social Integrity Constraints.

We believe that the conjunct use of declarative approaches coming from the Service Oriented Computing (SOC) and Multi Agent Systems (MAS) research areas could be fruitfully exploited to specify and verify service choreographies. To this aim, in this paper we show how DecSerFlow can be mapped to $\mathcal{S}$CIFF in an automatic and complete way, making the two proposals benefit from each other. We motivate the importance of adopting a declarative approach for modeling choreographies and show the feasibility of our approach by considering a simple but interesting running example.

The paper is organized as follows: sections 2 and 3 respectively introduce the running example and describe some issues which arise when modeling a choreography. Section 4 briefly introduce the DecSerFlow language, showing how the running example could be successfully modeled by using it; then, section 5 presents the $\mathcal{S}$CIFF framework and how DecSerFlow can be expressed in terms of $\mathcal{S}$CIFF Integrity Constraints. Discussion and Conclusions follow.

2 A running example

Let us consider a choreography that envisages three different roles: a *customer* which interacts with a *seller* to place an order of a set of items, and a *warehouse*

[3] SOcieties of heterogeneous ComputeeS, IST-2001-32530 (home page http://lia.deis.unibo.it/research/SOCS/).

which could participate to the interaction by communicating to the seller if it is able (or not) to ship the ordered items.

Each execution of the choreography (a *choreography instance*) is identified by the concept of *order*. The customer makes up an order by choosing one or more items from the seller list. During the order building phase (i.e. before committing an order), it is always possible to cancel the order; in this case, the user cannot choose other items within the same instance anymore, and the choreography terminates (a canceled order cannot be committed). After having committed an order, the customer expects a positive or negative answer from the seller. In case of a positive answer, a payment phase will be performed: the customer will pay for the order and, finally, the seller will deliver a single corresponding receipt.

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:

– 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.

3 What is the focus of a choreography?

By looking at the choreography description of the previous section, we notice that it 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. This is the reason why we find, inside the description, different kinds of constraints, as for example:

– time-ordered relationships among activities ("*after* having committed an order, the customer expects a positive or negative answer");
– cardinality constraints ("the seller will 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 non-oriented relationships among 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 [5], 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 [14] for a discussion).

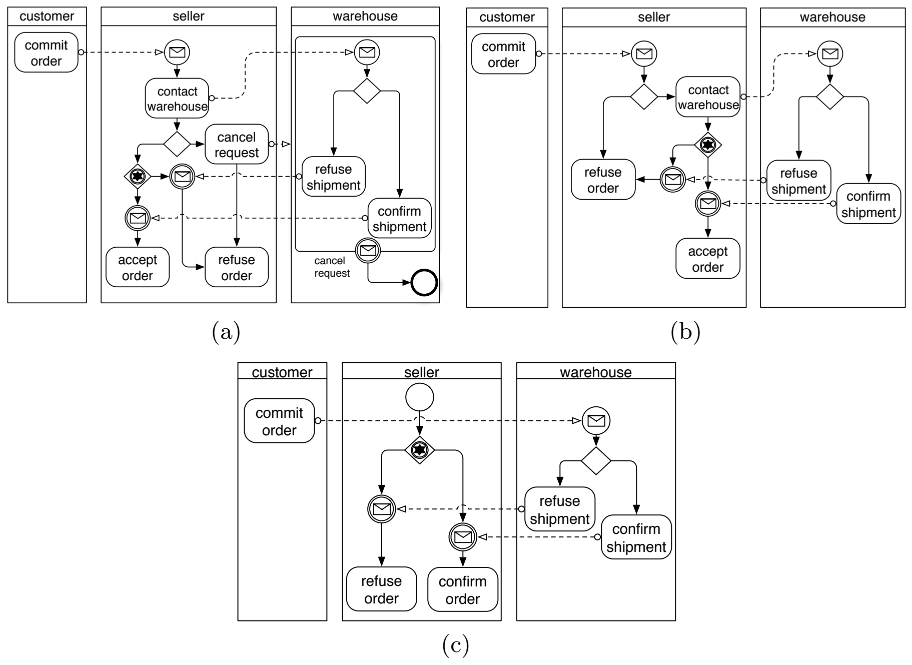


Fig. 1. Three different possible realizations of the acceptance phase in BPMN.

3.1 Avoiding over-specifications

Avoiding over-specifications is a key issue when modeling choreographies. Instead of strictly specify 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 regulate 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 Section 2. 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 commited order (because e.g. 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

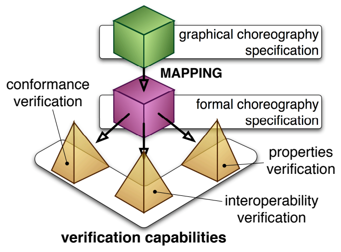

graphical choreography specification
MAPPING
conformance verification
formal choreography specification
properties verification
interoperability verification
verification capabilities

Fig. 2. A general framework for the specification and verification of choreographies

shows three different over-specified possible realizations of the acceptance phase by adopting BPMN [15] collaborative models.

Diagram 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 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 diagram 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, diagram 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 shows that approaching the choreography modeling task by adopting a typical control+message flow perspective leads to pointlessly complicate the model, loosing some acceptable interactions. 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.

3.2 Towards a framework encompassing semantics and verification capabilities

Besides being able to really capture the different concepts involved in a choreography, possibly in a user-friendly way, a modeling language should be supported by an underlying formal (possibly declarative) semantics, hence making possible different kind of verifications. Figure 2 shows the schema of a general choreography specification and verification framework.

The framework is mainly composed by three different parts: *(i)* a (graphical) high-level modeling language, capable to specify choreographies; *(ii)* an underlying formal language, equipped with different verification capabilities; and *(iii)* a mapping between the two specification languages, in order to automatically obtain the formal description from the graphical one.

W.r.t. the verification issue, we cite three fundamental ones:

- properties verification, to ensure that a choreography meets some general (such as livelock and deadlock freedom) or specific (i.e. domain dependent) properties;
- conformance verification, to verify (at run-time or a posteriori, by analyzing a message log) whether a set of services executing the choreography behaves as prescribed by the model;
- interoperability verification [16], to check if a concrete service behavioral interface is capable to play a given role within the choreography.

It is worth noting that such three verification issues are the same as the ones introduced by Guerin and Pitt in the context of open MAS [17]: *(i)* verify protocol properties, *(ii)* verify compliance by observation, and *(iii)* verify that an agent will always comply.

We propose to ground the general framework shown in Figure 2 by adopting DecSerFlow as a graphical specification language, and to exploit $\mathcal{S}$CIFF as its underlying formalism. To demonstrate the feasibility of our approach, we show how our running example could be successfully expressed in DecSerFlow, and then provide the mapping of the different DecSerFlow concepts to $\mathcal{S}$CIFF Integrity Constraints. In [18] we already introduced the use of $\mathcal{S}$CIFF for specifying choreographies and performing the conformance verification task, leaving out the high-level specification language and the corresponding mapping; this work could be considered as a first step to fill this gap.

4 Choreography modeling in DecSerFlow

In [5], 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).

As described in [5], 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, a lá policies/business rules. Constraints are given as templates, i.e. as relationships between two (or more) whatsoever activities: typically, the terms *source* and *target* activities indicate activities linked by a relationship, where the execution of the source activity "activates" the relation and impose some constraint on the target activity. The meaning of each constraint template is expressed as an LTL formula, 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;

source		**template name**	**target**	**description** (from the example)
cancel order	C_1	negation response	choose item	in case of cancelation, the user cannot choose other items [. . .] anymore
	C_2	responded absence	commit order	a canceled order cannot be commited
commit order	C_3	response	refuse or confirm order	after having committed an order, the customer expects a positive or negative answer from the seller
	C_4	precedence	confirm shipment	the seller could confirm the order only if the warehouse has previously confirmed the shipment
confirm order	C_5	response	payment	in the former situation [positive answer], a payment phase will be performed
refuse shipment	C_6	responded existence	refuse order	if the warehouse [. . .] is unable to execute the shipment, then the seller should refuse (or have refused) the order
payment	C_7	response	receipt delivery	the customer will pay for the order and, then, the seller should deliver a single corresponding receipt
receipt delivery	C_8	cardinality 0..1		the seller will deliver a single corresponding receipt

Table 2. Mapping the statements of the running example to DecSerFlow constraints

– *negation formulas*, the negated version of relation formulas.

In order to present the DecSerFlow notation and how it could be effectively used to model service choreographies, we show how our running example could be expressed as a DecSerFlow diagram. In our example, we will use only a limited number of DecSerFlow relations, such as *responded existence* (if A is performed, then also B must be performed, either before or after A) and *response* (if A is performed, then B must be performed after). For a complete description of the DecSerFlow language and its underlying LTL formalization, the interested reader is referred to [5].

4.1 Modeling the running example

Table 2 shows how the different statements of our running example could be translated to DecSerFlow activities and constraints in an intuitive and straightforward way.

For example, to specify that only a single receipt should be delivered by the seller, we may 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. A *responded existence* relation is used to model the relationship between the refusal 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. DecSerFlow's *response* relation imposes a forward temporal order on the responded existence formula; for example, constraint C_3 states that after having executed the order commitment, 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). Obviously, a *precedence* formula is provided too, (e.g. C_4).

DecSerFlow defines also more complex relationships, which are not part of our running example. An example is the *chain response* formula, which allows the user to model the typical strict sequence relationships of business processes: it states that whenever the source happens, then the target should be performed immediately after it.

For each positive relationships, DecSerFlow defines a corresponding negative version. Basically, negative relations forbids 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. 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). It is worth noting that, as pointed out in [5], some negative relations are equivalent; e.g., stating that B is responded absence of A is equivalent to specify that A and B should not coexist in the same execution instance.

4.2 Completing the DecSerFlow model

By deeply analyzing the running example, we could complete the DecSerFlow diagram shown in Table 2 with other useful inferred constraints, in order to really model all the intented concepts of the description; the result is shown in Table 3, while in Figure 3 the whole set of constraints is shown using the DecSerFlow graphical notation (see also Tables 4 and 5 for the correspondence between the DecSerFlow graphical symbols and their meaning).

C_{15} and C_{16} deal with the core concept of the choreography, which is actually the commitment of one order. Since such an order could be canceled, we attach an *absence*(1) constraint to the order commitment activity (to express that at most one order can be committed), and bind the cancelation and the commitment with a *mutual substitution* DecSerFlow relation, which states that at least one of the two bounded activities has to be executed (i.e. an order should be committed or canceled).

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

The $\mathcal{S}$CIFF [13] language was originally introduced for the specification of global interaction protocols in open agent societies. As we have already pointed out, it does not make any assumption about participants internals, but instead focuses

source		**type**	**target**	**description** (from the example)
refuse order	C_9	precedence	commit order	An answer from the seller is valid only if it is performed after order commitment
confirm order	C_{10}	precedence	commit order	
payment	C_5	precedence	confirm order	A valid payment should be preceded by the confirmation of the order
deliver receipt	C_7	precedence	payment	The receipt should be delivered only if the order has been paid
target		**type**	**target**	**description** (from the example)
confirm order	C_{11}	not co-existence	refuse order	Possible answers are mutually exclusive
confirm shipment	C_{12}	not co-existence	refuse shipment	
commit order	C_{13}	precedence	choose item	an order is made up by at least one chosen item
cancel order	C_{14}	precedence	choose item	
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}	mutual substitution	cancel order	

Table 3. Inferred DecSerFlow constraints to complete the running example

on the observable and relevant events which occur within the society at run-time. To let the user decides 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".

$\mathcal{S}$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". In agreement with DecSerFlow, $\mathcal{S}$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 neg-

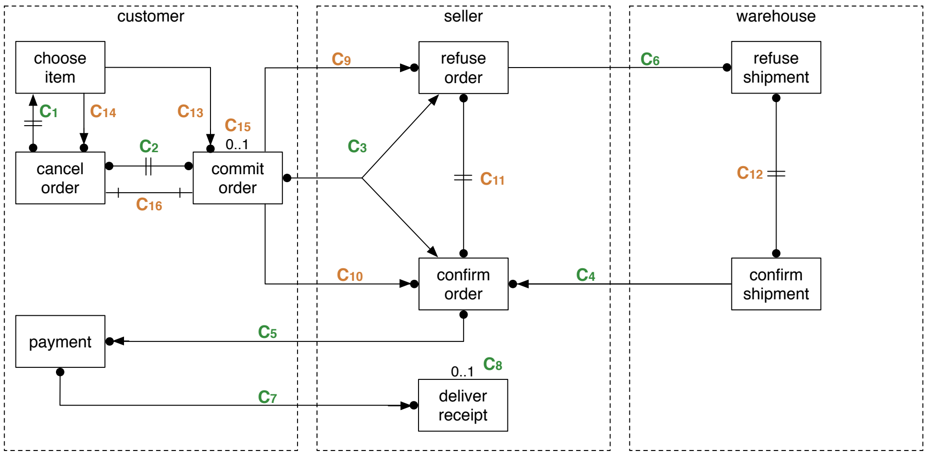


Fig. 3. DecSerFlow model of the running example

ative 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 (a là Constraint Logic Programming), like $T > 10$, as well as Prolog predicates can be specified over the variables; attaching the example constraint on the above expectation means that the expectation is about an event to happen at a time greater than 10. Conversely, 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.

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. model the interaction protocol. They come as rules of the form *body* $\rightarrow$ *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:

$$\begin{aligned}&\mathbf{H}(pay(Customer, Seller, Item), T_p)\\ \rightarrow&\mathbf{E}(deliver(Seller, Customer, receipt(Order, Id)), T_d) \wedge T_d > T_p \wedge T_d < T_p + 24.\end{aligned}$$

$\mathcal{S}$CIFF accepts also a (Prolog) knowledge base, where the user can define all the pieces of knowledge which are independent from the interaction. Defined predicates could be used inside Integrity Constraints, reconciling forward, abductive reasoning with backward, goal-oriented reasoning. Finally, note that interaction is considered to be goal oriented: the same interaction protocol could be seamlessly used for achieving different goals, which can be expressed by means of Prolog predicates and expectations.

The $\mathcal{S}$CIFF semantics is based on Abductive Logic Programming: an interaction specification (i.e. the set of rules regulating the allowed possible interactions) is mapped to 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 conformance of a set of interacting entities w.r.t. the considered protocol by hypothesizing positive (resp. negative) expectations and 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 [13].

5.1 Expressing DecSerFlow concepts as Integrity Constraints

Let us now consider again our running example, in order to explain how the different DecSerFlow concepts could be mapped to $\mathcal{S}$CIFF Integrity Constraints.

Roughly speaking, each DecSerFlow constraint is mapped to a set of $\mathcal{S}$CIFF Integrity Constraints. The body of the Integrity Constraint which maps a relation or negation formula is constituted by the happened event which corresponds to the formula's source (each DecSerFlow relation is triggered when its source activity is performed). Depending on the nature of the relation, the head is instead is determined by (a disjunction of) positive or negative expectations.

For example, to specify that a generic activity A is subject to an *absence*(N) cardinality constraint, $\mathcal{S}$CIFF uses an Integrity Constraint which states that if N different executions of A are performed, then the $N+1$-th is forbidden. Since $\mathcal{S}$CIFF adopts an explicit notion of time, differences between executions are modeled as differences between the involved execution times; hence, the *absence*(N) on activity A can be specified as follows[4]:

$$\bigwedge_{i=1}^{N}\Big(\mathbf{H}(A,T_i)\wedge T_i > T_{i-1}\Big)\rightarrow \mathbf{EN}(A,T)\wedge T > T_N.$$

Furthermore, thanks to the explicit notion of time, another interesting feature of the mapping is that the "response" and "precedence" version of each formula are formalized in the same way, but by imposing opposite constraints on the involved times. Table 4 explicitly points out such similarities by showing how the responded existence, response and precedence constraints, as well as their negated version, can be mapped to $\mathcal{S}$CIFF.

Some DecSerFlow formulas 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 (see Table 5). Therefore, they define, in some sense, the initial goal of the choreography, since the corresponding expectations are generated independently from the interaction.

Table 6 represents the complete mapping of the DecSerFlow model shown in Figure 3. For the sake of simplicity, we have left out the information about

[4] We suppose that $T_0 = 0$ and that at a given time only one activity can happen.

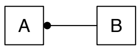
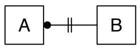
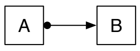
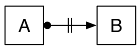
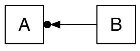
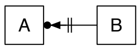
DecSerFlow formula	Meaning	$\mathcal{S}$CIFF Integrity Constraint
	if A is executed, then B should be executed too	$\mathbf{H}(A,T_A) \rightarrow \mathbf{E}(B,T_B)$
	if A is executed, then B cannot be executed	$\mathbf{H}(A,T_A) \rightarrow \mathbf{EN}(B,T_B)$
	if A is executed, then B should be executed after it	$\mathbf{H}(A,T_A) \rightarrow \mathbf{E}(B,T_B) \quad \wedge T_B > T_A$
	if A is executed, then B cannot be executed after it	$\mathbf{H}(A,T_A) \rightarrow \mathbf{EN}(B,T_B) \wedge T_B > T_A$
	if A is executed, then B should be executed before it	$\mathbf{H}(A,T_A) \rightarrow \mathbf{E}(B,T_B) \quad \wedge T_B < T_A$
	if A is executed, then B cannot be executed before it	$\mathbf{H}(A,T_A) \rightarrow \mathbf{EN}(B,T_B) \wedge T_B < T_A$

Table 4. Mapping of the simple DecSerFlow relation and negation formulas in $\mathcal{S}$CIFF

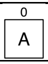
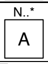
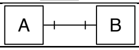
DecSerFlow formula	Meaning	$\mathcal{S}$CIFF Integrity Constraint
	A is forbidden	$\rightarrow \mathbf{EN}(A,T_A)$
	A has to be executed at least N times	$\rightarrow \bigwedge_{i=1}^{N} \Big(\mathbf{E}(A,T_i) \wedge T_i > T_{i-1}$
	A or B should be executed	$\rightarrow \mathbf{E}(A,T_A) \vee \mathbf{E}(B,T_A)$

Table 5. Mapping of "goal-oriented" DecSerFlow formulas

activities originators (i.e. about the role responsible for an activity); such an information could be seamlessly added to the $\mathcal{S}$CIFF formalization, but it is not envisaged in the current version of DecSerFlow.

As already pointed out, DecSerFlow defines other constraints, missing in our running example. Anyway, they are mapped to $\mathcal{S}$CIFF Integrity Constraints too (see [19] for a complete description of such a mapping). For example, the following rule maps the *chain response* between A and B:

$$\mathbf{H}(A,T_A) \rightarrow \mathbf{E}(B,T_B) \wedge T_B > T_A \wedge \mathbf{EN}(X,T_X) \wedge T_X > T_A \wedge T_X < T_B.$$

The translation tries to intuitively capture the notion of *next state*, which is directly expressed in LTL as a temporal modality (by using the operator $\circ$). It relies on the fact that if B should belong to the next state of A, then between the two execution times no other activity should be performed. For a description of the complete translation of core DecSerFlow concepts to $\mathcal{S}$CIFF, see [19].

6 Discussion and Conclusions

In this work we have proposed a conjunct use of declarative approaches coming from the SOC and MAS research areas, to the aim of specifying and verifying service choreographies.

C_1	$\mathbf{H}(cancel\_order, T_c)$	$\rightarrow \mathbf{EN}(choose\_item, T_i) \wedge T_i > T_c.$
C_2	$\mathbf{H}(cancel\_order, T_c)$	$\rightarrow \mathbf{EN}(commit\_order, T_o).$
	$\mathbf{H}(commit\_order, T_o)$	$\rightarrow \mathbf{EN}(cancel\_order, T_c).$
C_3	$\mathbf{H}(commit\_order, T_o)$	$\rightarrow \mathbf{E}(confirm\_order, T_c) \wedge T_c > T_o$
		$\vee\ \mathbf{E}(refuse\_order, T_r) \wedge T_r > T_o.$
C_4	$\mathbf{H}(confirm\_order, T_o)$	$\rightarrow \mathbf{E}(confirm\_shipment, T_s) \wedge T_s < T_o.$
C_5	$\mathbf{H}(confirm\_order, T_c)$	$\rightarrow \mathbf{E}(payment, T_p) \wedge T_p > T_c.$
	$\mathbf{H}(payment, T_p)$	$\rightarrow \mathbf{E}(confirm\_order, T_c) \wedge T_c < T_p.$
C_6	$\mathbf{H}(refuse\_shipment, T_s)$	$\rightarrow \mathbf{E}(refuse\_order, T_o).$
C_7	$\mathbf{H}(payment, T_p)$	$\rightarrow \mathbf{E}(deliver\_receipt, T_d) \wedge T_d > T_p.$
	$\mathbf{H}(deliver\_receipt, T_d)$	$\rightarrow \mathbf{E}(payment, T_p) \wedge T_p < T_d.$
C_8	$\mathbf{H}(deliver\_receipt, T_{d1})$	$\rightarrow \mathbf{EN}(deliver\_receipt, T_{d2}) \wedge T_{d2} > T_{d1}.$
C_9	$\mathbf{H}(refuse\_order, T_r)$	$\rightarrow \mathbf{E}(commit\_order, T_o) \wedge T_o < T_r.$
C_{10}	$\mathbf{H}(confirm\_order, T_c)$	$\rightarrow \mathbf{E}(commit\_order, T_o) \wedge T_o < T_c.$
C_{11}	$\mathbf{H}(refuse\_order, T_r)$	$\rightarrow \mathbf{EN}(confirm\_order, T_c).$
	$\mathbf{H}(confirm\_order, T_c)$	$\rightarrow \mathbf{EN}(refuse\_order, T_r).$
C_{12}	$\mathbf{H}(refuse\_shipment, T_r)$	$\rightarrow \mathbf{EN}(confirm\_shipment, T_c).$
	$\mathbf{H}(confirm\_shipment, T_c)$	$\rightarrow \mathbf{EN}(refuse\_shipment, T_r).$
C_{13}	$\mathbf{H}(commit\_order, T_c)$	$\rightarrow \mathbf{E}(choose\_item, T_i) \wedge T_i < T_c.$
C_{14}	$\mathbf{H}(cancel\_order, T_c)$	$\rightarrow \mathbf{E}(choose\_item, T_i) \wedge T_i < T_c.$
C_{15}	$\mathbf{H}(commit\_order, T_{c1})$	$\rightarrow \mathbf{EN}(commit\_order, T_{c2}) \wedge T_{c2} > T_{c1}.$
C_{16}		$\rightarrow \mathbf{E}(commit\_order, T_o)$
		$\vee\ \mathbf{E}(cancel\_order, T_c).$

Table 6. Mapping of the DecSerFlow running example to $\mathcal{S}$CIFF

In particular, we have chosen DecSerFlow as the modeling language and $\mathcal{S}$CIFF as its underlying formalization. To make DecSerFlow benefit of $\mathcal{S}$CIFF in an automatic way, we have shown how the different DecSerFlow concepts can be mapped to $\mathcal{S}$CIFF Integrity Constraints and applied our methodology on a running example. The advantage of such a translation is twofold: on one hand, it is possible to specify $\mathcal{S}$CIFF rules by using an intuitive, extensible and user-friendly graphical language; on the other hand, a DecSerFlow model may be grounded not only on LTL but also on the $\mathcal{S}$CIFF abductive framework, acquiring some new advantages and features, such as:

– Expressivity of the language. The $\mathcal{S}$CIFF language is capable to model rich constraints and conditions on data and execution times involved in the interaction; we are currently studying how DecSerFlow could be extended to graphically represent such constraints.
– Verification capabilities of the $\mathcal{S}$CIFF framework. As described in [13, 18], by translating DecSerFlow to a $\mathcal{S}$CIFF specification we could automatically use it to perform the conformance verification task. Furthermore, $\mathcal{S}$CIFF has been extended to deal also with the verification of properties [20] and interoperability [21]; we intend to study how such extended proofs could be applied to DecSerFlow models, aiming at covering all the building parts of the general framework schema shown in figure 2.
– Possibility to mine DecSerFlow models from execution traces. Since $\mathcal{S}$CIFF belongs to the logic programming setting, it is possible to apply all the reasoning techniques developed inside such a setting on it. In particular, in [22] we have shown how an Inductive Logic Programming algorithm can be adapted to mine $\mathcal{S}$CIFF rules from event logs; thanks to the one-to-one mapping of DecSerFlow concepts to $\mathcal{S}$CIFF, it is then possible to automatically obtain a corresponding DecSerFlow description of the mined model.

Finally, as future work we envisage a deep comparison between $\mathcal{S}$CIFF and LTL, to better understand their strength, weaknesses and relationships and to exploit the possibility to have two different mappings of DecSerFlow.

References

1. Barros, A., Dumas, M., Oaks, P.: A critical overview of the web services choreography description language (WS-CDL). BPTrends (2005)
2. van der Aalst, W.M.P., Dumas, M., ter Hofstede, A.H.M., Russell, N., Verbeek, H.M.W., Wohed, P.: Life after BPEL? In Bravetti, M., Kloul, L., Zavattaro, G., eds.: International Workshop on Web Services and Formal Methods, WS-FM 2005, Versailles, France, September 1-3, 2005, Proceedings. Volume 3670 of Lecture Notes in Computer Science., Springer (2005) 35–50
3. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Business Process Execution Language for Web Services version 1.1. (2003)
4. W3C: Web services choreography description language version 1.0
5. der Aalst, W.M.P.V., Pesic, M.: DecSerFlow: Towards a truly declarative service flow language. In: WS-FM'06. Volume 4184 of LNCS., Springer (2006)
6. Baldoni, M., Baroglio, C., Martelli, A., Patti, V., Schifanella, C.: Verifying the conformance of web services to global interaction protocols: A first step. In Bravetti, M., Kloul, L., Zavattaro, G., eds.: International Workshop on Web Services and Formal Methods, WS-FM 2005, Versailles, France, September 1-3, 2005, Proceedings. Volume 3670 of Lecture Notes in Computer Science., Springer (2005) 257–271
7. Bauer, B., Ḿuller, J.P., Odell, J.: Agent uml: a formalism for specifying multiagent software systems. In: First international workshop, AOSE 2000 on Agent-oriented software engineering, Springer-Verlag (2001) 91–103
8. Fornara, N., Colombetti, M.: Operational specification of a commitment-based agent communication language, Bologna, Italy (July 15–19 2002) 535–542

9. Yolum, P., Singh, M.: Flexible protocol specification and execution: applying event calculus planning using commitments. 527–534
10. Desai, N., Chopra, A.K., Singh, M.P.: 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) 103–110
11. Mallya, A.U., Desai, N., Chopra, A.K., Singh, M.P.: Owl-p: Owl for protocol and processes. In Dignum, F., Dignum, V., Koenig, S., Kraus, S., Singh, M.P., Wooldridge, M., eds.: 4rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005), July 25-29, 2005, Utrecht, The Netherlands, ACM (2005) 139–140
12. Walton, C.: Protocols for web service invocation. Proceedings of the AAAI Fall Symposium on Agents and the Semantic Web (ASW05), Arlington, Virginia, USA. (November 2005)
13. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verifiable agent interaction in abductive logic programming: the SCIFF framework. TOCL (2007) Accepted for publication.
14. Chopra, A.K., Singh, M.P.: Producing compliant interactions: Conformance, coverage, and interoperability. In Baldoni, M., Endriss, U., eds.: DALT. Volume 4327 of Lecture Notes in Computer Science., Springer (2006) 1–15
15. White, S.A.: Business process modeling notation specification. Technical report, OMG (2006)
16. Baldoni, M., Baroglio, C., Martelli, A., Patti, V.: A priori conformance verification for guaranteeing interoperability in open environments. In: Service-Oriented Computing - ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, 2006, Proceedings. Volume 4294 of Lecture Notes in Computer Science., Springer (2006) 339–351
17. Guerin, F., Pitt, J.: Proving properties of open agent systems, Bologna, Italy (July 15–19 2002) 557–558
18. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Montali, M., Storari, S., Torroni, P.: Computational logic for run-time verification of web services choreographies: Exploiting the *socs-si* tool. In Bravetti, M., Núñez, M., Zavattaro, G., eds.: Web Services and Formal Methods, Third International Workshop, WS-FM 2006 Vienna, Austria, September 8-9, 2006, Proceedings. Volume 4184 of Lecture Notes in Computer Science., Springer (2006) 58–72
19. Chesani, F., Mello, P., Montali, M., Storari, S.: Towards a decserflow declarative semantics based on computational logic. Technical Report DEIS-LIA-07-002, DEIS, Bologna, Italy (2007)
20. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Security protocols verification in Abductive Logic Programming: a case study. In Dikenelli, O., Gleizes, M.P., Ricci, A., eds.: Proceedings of ESAW'05, Kuşadasi, Aydin, Turkey, October 26-28, 2005. Volume 3963. (2006) 106–124
21. Alberti, M., Gavanelli, M., Lamma, E., Chesani, F., Mello, P., Montali, M.: An abductive framework for a-priori verification of web services. In Bossi, A., Maher, M.J., eds.: Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 10-12, 2006, Venice, Italy, ACM (2006) 39–50
22. Lamma, E., Mello, P., Montali, M., Riguzzi, F., Storari, S.: Inducing declarative logic-based models from labeled traces. In: Proceedings of the 5th International Conference on Business Process Management (BPM 2007), LNCS (2007) To appear