Computational Logic for Run-Time Verification of Web Services Choreographies: Exploiting the *SOCS-SI* Tool

Marco Alberti[2], Federico Chesani[1], Marco Gavanelli[2], Evelina Lamma[2], Paola Mello[1], Marco Montali[1], Sergio Storari[2], and Paolo Torroni[1]

[1] DEIS - Dipartimento di Elettronica, Informatica e Sistemistica
Facoltà di Ingegneria, Università di Bologna
viale Risorgimento, 2
40136 – Bologna, Italy
{fchesani, pmello, mmontali, ptorroni}@deis.unibo.it
[2] DI - Dipartimento di Ingegneria
Facoltà di Ingegneria, Università di Ferrara
Via Saragat, 1
44100 – Ferrara, Italy
{marco.gavanelli, marco.alberti, lme, strsrg}@unife.it

Abstract. In this work, we investigate the feasibility of using a framework based on computational logic, and mainly defined in the context of Multi-Agent Systems for Global Computing (SOCS UE Project), for modeling choreographies of Web Services with respect to the conversational aspect.

One of the fundamental motivations of using computational logic, beside its declarative and highly expressive nature, is given by its operational counterpart, that can provide a proof-theoretic framework able to verify the consistency of services designed in a cooperative and incremental manner.

In particular, in this paper we show that suitable "Social Integrity Constraints", introduced in the SOCS social model, can be used for specifying global protocols at the choreography level. In this way, we can use a suitable tool, derived from the proof-procedure defined in the context of the SOCS project, to check at run-time whether a set of existing services behave in a conformant manner w.r.t. the defined choreography.

1 Introduction

Service Oriented Architectures (SOA) have recently emerged as a new paradigm for structuring inter-/intra- business information processes. While SOA is indeed a set of principles, methodologies and architectural patterns, a more practical instance of SOA can be identified in the Web Services technology, where the business functionalities are encapsulated in software components, and can be invoked through a stack of Internet Standards.

The standardization process of the Web Service technology is at a good maturation point: in particular, the W3C Consortium has proposed standards for developing basic services and for interconnecting them on a point-to-point basis. These standards have been widely accepted; vendors like Microsoft and IBM are supporting the technology within their development tools; private firms are already developing solutions

M. Bravetti, M. Nuñes, and G. Zavattaro (Eds.): WS-FM 2006, LNCS 4184, pp. 58–72, 2006.

for their business customer, based on the web services paradigm. However, the needs for more sophisticated standards for service composition have not yet fully satisfied. Several attempts have been made (WSFL, XLang, BPML, WSCL, WSCI), leading to two dominant initiatives: BPEL [1] and WS-CDL [2].

Both these initiatives however have missed to tackle some important issues. We agree with the view [3,4] that both BPEL and WS-CDL languages lack of declarativeness, and more dangerous, they both lack an underlying formal model and semantics. Hence, issues like *run-time conformance testing*, *composition verification*, *verification of properties* are not fully addressed by the current proposals. Also semantics issues, needed in order to verify more complex properties (besides properties like livelock, deadlock, leak freedom, etc.), have been left behind.

Some of these issues have been already subject of research: generally, a mapping between choreographed/orchestrated models to specific formalisms is proposed, and then single issues are solved in the transformed model. E.g., the *composition verification* is addressed in [5,6]; *process mining* and *a-posteriori conformance testing* are addressed in [7]; livelock, deadlock, etc. properties are tackled in [8,9].

In this paper, we focus on a particular issue: the *conformance testing* (also called *run-time behaviour conformance* in [3]). Once a global protocol (or choreography) has been defined, a question arises: how is it possible to check if the actors play in a conformant manner w.r.t the defined choreography? Any solution should take into account answering the question by analyzing only the external, observable behaviour of the peers, without assuming any hypothesis or knowledge on their internals (in order to not undermine the heterogeneity).

Taking inspiration by the many analogies between the Web Services research field and the Multi Agent System (MAS) field [5], we exploit a framework, namely $\mathcal{S}$CIFF, for verifying at run-time (or a-posteriori using an *event log*) if the peers behave in a conformant manner w.r.t. a given choreography. Within the $\mathcal{S}$CIFF framework, a language suitable for specifying global choreographies is provided: a formal semantics is provided too, based on abductive logic programming [10]. We defined the $\mathcal{S}$CIFF framework in the SOCS european project [11], where we addressed the issue of providing a formal language to define multi agent protocols. Its operational counterpart is an abductive proof procedure, called $\mathcal{S}$CIFF, exploited to check the compliance of agents to protocols. Moreover, a tool (namely *SOCS-SI* [12]) has been developed for automatically analyzing and verifying peers interactions, w.r.t. a protocol expressed in the language above.

In this paper we show that suitable "Social Integrity Constraints", introduced in the SOCS social model, can be used for specifying global protocols at the choreography level. In this way, we can use a suitable tool, derived from the proof-procedure defined in the context of the SOCS project, to check at run-time whether a set of existing services behave in a conformant manner w.r.t. the defined choreography.

The paper is organized as follows: in Section 2 we introduce the $\mathcal{S}$CIFF framework and provide its declarative semantics. Then, in Section 3 we sketch how a simple choreography can be modeled within the framework. In Section 4 we show how the *run-time conformance testing* issue can be addressed in our framework, grounding our proposal to a practical example. Discussion and conclusions follow in Section 5.

2 The $\mathcal{S}$CIFF Framework

In this section, we present the $\mathcal{S}$CIFF framework, describing how the conversational part of a choreography as well as its static knowledge can be suitably expressed within the framework. Moreover, we provide a formal definition of *fulfillment* (i.e., a run-time behaviour of some peers respects a given choreography) and *violation* (i.e., when the peers does not behave in a conformant manner).

2.1 Events, Happened Events and Expected Events

The definition of *Event* greatly varies, depending on the application domain. For example, in the Web Service domain, an event could be the fact that a certain web method has been invoked; in a Semantic Web scenario instead, an event could be the fact that some information available on a site has been updated. Moreover, within the same application domain there could be several different notions of events, depending on the assumed perspective, the granularity, etc.

The $\mathcal{S}$CIFF language abstracts completely from the problem of deciding "what is an event", and rather lets the developers decide which are the important events for modeling the domain, at the desired level. Each event that can be described by a *Term*, can be used in $\mathcal{S}$CIFF. For example, in a peer-to-peer communication system, an event could be the fact that someone communicates something to someone else (i.e., a *communicative* action has been performed):

$$tell(alice, bob, msgContent)$$

Another event could be the fact that a web service has updated some information stored into an external database, or that a bank clerk, upon the request of a customer, has provided him/her some money (like in Eq. 2). Of course, in order to perform some reasoning about such events, accessibility to such information is a mandatory requirement.

In the $\mathcal{S}$CIFF framework, similarly to what has been done in [13], we distinguish between the description of the *event*, and the fact that the event has happened. Typically, an event happens at a certain time instant; moreover the same event could happen many times [1] . Happened events are represented as an atom $\mathbf{H}(Event, Time)$, where $Event$ is a *Term*, and $Time$ is an integer, representing the discrete time point in which the event happened.

One innovative contribution of the $\mathcal{S}$CIFF framework is the introduction of *expectations* about events. Indeed, beside the explicit representation of "what" happened and "when", it is possible to explicitly represent also "what" is expected, and "when" it is expected. The notion of *expectation* plays a key role when defining global 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"*. Expectations about events come with form

$$\mathbf{E}(Event, Time)$$

[1] In our approach the happening of identical events at the same time instant are considered as if only one event happens; if the same event happens more than once, but at different time instants, then they are indeed considered as different happenings.

where $Event$ and $Time$ can be a variable, or they could be grounded to a particular term/value. Constraints, like $Time > 10$, can be specified over the variables: in the given example, the expectation is about an event to happen at a time greater than 10 (hence the event is expected to happen *after* the time instant 10).

Given the notions of *happened event* and of *expected event*, two fundamental issues arise: first, how it is possible to specify the link between these two notions. Second, how it is possible to verify if all the expectations have been effectively satisfied. The first issue is fundamental in order to easy the definition of a choreography, and it will be addressed in the rest of this section. The second issue, instead, is inherently related to the problem of establishing if a web service is indeed behaving in a compliant manner w.r.t. a given choreography: the solution proposed by the $\mathcal{S}$CIFF framework is presented in Section 4.1.

2.2 Choreography Integrity Constraints

Choreography Integrity Constraints $\mathcal{IC}_{chor}$ are forward rules, of the form

$$Body \rightarrow Head$$

whose *Body* can contain literals and (happened and expected) events, and whose *Head* can contain (disjunctions of) conjunctions of expectations. In Eq. (1) we report the formal definition of the grammar, where $Atom$ and $Term$ have the usual meaning in Logic Programming [14] and $Constraint$ is interpreted as in Constraint Logic Programming [15].

$$\begin{aligned}
\mathcal{IC}_{chor} &::= [IC]^\star \\
IC &::= Body \rightarrow Head \\
Body &::= (HapEvent|Expect)\ [\wedge BodyLit]^\star \\
BodyLit &::= HapEvent|Expect|Literal|Constraint \\
Head &::= Disjunct\ [\ \vee Disjunct\ ]^\star|false \\
Disjunct &::= Expect\ [\ \wedge (Expect|Literal|Constraint)]^\star \\
Expect &::= \mathbf{E}(Term\ [, T]) \\
HapEvent &::= \mathbf{H}(Term\ [, T]) \\
Literal &::= Atom \mid \neg Atom
\end{aligned} \tag{1}$$

The syntax of $\mathcal{IC}_{chor}$ is a simplified version of that one defined for the SOCS Integrity Constraints [16]. In particular, in the context of choreographies, we do not consider negative expectations (informally, expectations about prohibited events) and explicit negation. In fact, we assume that choreographies completely specify all the events that must happen (by means of expectations), and that not expected events are indeed forbidden. This assumption is formally specified by the definition of violation of a choreography, that we provide later in the paper (see Def. 2).

CLP constraints [15] can be used to impose relations or restrictions on any of the variables that occur in an expectation, like imposing conditions on the role of the participants, or on the time instants the events are expected to happen. For example, time conditions might define orderings between the messages, or enforce deadlines.

$\mathcal{IC}_{chor}$ allows the user to define how an interaction should evolve, given some previous situation, that can be represented in terms of happened events. Rules like:

> *"if a customer requests the withdrawal of X euros from the bank account, the bank should give the requested money within 24 hours from the request, or should explicitly notify the user of the impossibility"*

can be translated straightforward, e.g. in the corresponding $\mathcal{IC}_{chor}$:

$$\begin{aligned} &\mathbf{H}(request(User, Bank, withdraw(X)), T_r) \\ \rightarrow &\mathbf{E}(give(Bank, User, money(X)), T_a) \wedge T_a < T_r + 24 \\ \vee &\mathbf{E}(tell(Bank, User, not\_possible, reason(\ldots)), T_p) \end{aligned} \tag{2}$$

2.3 The Choreography Knowledge Base

The *Integrity Constraints* are a suitable tool for effectively defining the desired behaviour of the participants to an interaction, as well as the evolution of the interaction itself. However, they mostly capture the "dynamic" aspects of the interactions, while more static information is not so easily tackled by these rules. For example, a common situation is the one where, before giving the money requested, the bank could check if the customer's deposit contains enough money to cover the withdrawal. Or, if the customer indeed has a bank account with that bank, and hence if he/she is entailed to ask for a withdrawal.

Such type of knowledge is independent of the single instance of interaction, but is often referred during the interaction. The $\mathcal{S}$CIFF framework allows to define such knowledge in the *Choreography Knowledge Base* KB_{chor}. The KB_{chor} specifies declaratively pieces of knowledge of the choreography, such as roles descriptions, list of participants, etc. KB_{chor} is expressed in the form of clauses (a logic program); the clauses may contain in their body expectations about the behaviour of participants, defined literals, and constraints, while their heads are atoms. The syntax is reported in Equation (3).

$$\begin{aligned} KB_{chor} &::= [Clause]^{\star} \\ Clause &::= Atom \leftarrow Cond \\ Cond &::= ExtLiteral\ [\ \wedge ExtLiteral\ ]^{\star} \\ ExtLiteral &::= Literal | Expectation | Constraint \\ Expectation &::= \mathbf{E}(Term\ [, T]) \\ Literal &::= Atom \mid \neg Atom \mid true \end{aligned} \tag{3}$$

Moreover, in our vision, a choreography can be *goal directed*, i.e. a specific goal $\mathcal{G}_{chor}$ can be specified. E.g., a choreography used in an electronic auction system could have the goal of selling all the goods in the store. Another goal could be instead to sell at least n items at a price higher than a given threshold. Hence, the same auction mechanism described by the same rules (integrity constraints), can be used seamlessly for achieving different goals. Such goals can be defined like the clauses of the KB_{chor}, as specified in Eq. 3. Typically, a goal is defined as expectations about the outcomes of the choreography, i.e. in terms of messages (and their contents) that should be exchanged. If no particular goal is required to be achieved, $\mathcal{G}_{chor}$ is bound to *true*.

2.4 Declarative Semantics of the $\mathcal{S}$CIFF Framework

In the $\mathcal{S}$CIFF framework, a choreography is interpreted in terms of an Abductive Logic Program (ALP). In general, an ALP [10] is a triple $\langle P, A, IC\rangle$, where P is a logic program, A is a set of predicates named *abducibles*, and IC is a set of integrity constraints. Roughly speaking, the role of P is to define predicates, the role of A is to fill-in the parts of P which are unknown, and the role if IC is to control the ways elements of A are hypothesised, or "abduced". Reasoning in abductive logic programming is usually goal-directed (being G a goal), and it accounts to finding a set of abduced hypotheses Δ built from predicates in A such that $P \cup \Delta \models G$ and $P \cup \Delta \models IC$. In the past, a number of proof-procedures have been proposed to compute Δ (see Kakas and Mancarella [17], Fung and Kowalski [18], Denecker and De Schreye [19], etc.).

The idea we exploited in the $\mathcal{S}$CIFF framework is to adopt abduction to dynamically *generate* the expectations and to perform the *conformance check*. Expectations are defined as abducibles, and are hypothesised by the abductive proof procedure, i.e. the proof procedure makes hypotheses about the behaviour of the peers. A confirmation step, where these hypotheses must be confirmed by happened events, is then performed: if no set of hypotheses can be fulfilled, a violation is detected. In this paper, we also require that all the happened events are indeed expected.

A choreography specification $\mathcal{C}$ is defined by the triple:

$$\mathcal{C} \equiv \langle KB_{chor}, \mathcal{E}_{chor}, \mathcal{IC}_{chor}\rangle$$

where:

- KB_{chor} is the *Knowledge Base*,
- $\mathcal{E}_{chor}$ is the set of *abducible predicates* (i.e. expectations), and
- $\mathcal{IC}_{chor}$ is the set of *Choreography Integrity Constraints*.

A *choreography instance* $\mathcal{C}_{HAP}$ is a choreography specification grounded on a set $\mathbf{HAP}$ of happened events. We give semantics to a choreography instance by defining those sets $\mathbf{PEND}$ (Δ in the abductive framework) of expectations which, together with the choreography's knowledge base and the happened events $\mathbf{HAP}$, imply an instance of the goal (Eq. 4) - if any - and *satisfy* the integrity constraints (Eq. 5).

$$KB_{chor} \cup \mathbf{HAP} \cup \mathbf{PEND} \models \mathcal{G}_{chor} \tag{4}$$

$$KB_{chor} \cup \mathbf{HAP} \cup \mathbf{PEND} \models \mathcal{IC}_{chor} \tag{5}$$

At this point it is possible to define the concepts of *fulfillment* and *violation* of a set $\mathbf{PEND}$ of expectations. Fulfillment requires all the $\mathbf{E}$ expectations to have a matching happened event, and that all the happened event were indeed expected:

Definition 1. (Fulfillment) *Given a choreography instance $\mathcal{C}_{\mathbf{HAP}}$, a set of expectations* $\mathbf{PEND}$ *is* fulfilled *if and only if for all (ground) terms p:*

$$\mathbf{HAP} \cup \mathbf{PEND} \cup \{\mathbf{E}(p) \leftrightarrow \mathbf{H}(p)\} \not\models \mathit{false} \tag{6}$$

Symmetrically, we define violation as follows:

Definition 2. (Violation) *Given a choreography instance* $\mathcal{C}_{\mathbf{HAP}}$*, a set of expectations* $\mathbf{PEND}$ *is* violated *if and only if there exists a (ground) term* p *such that:*

$$\mathbf{HAP} \cup \mathbf{PEND} \cup \{\mathbf{E}(p) \leftrightarrow \mathbf{H}(p)\} \models \mathit{false} \tag{7}$$

Notice that, w.r.t. the original $\mathcal{S}$CIFF framework defined for the MAS scenario, the definitions of *fulfillment* and *violation* are slightly different. In fact in the Web Services scenario we consider as a violation all the events that happen without being expected. Notice that two different kinds of violation are detected by $\mathcal{S}$CIFF: $i)$ an expected event does not have a corresponding happened event, and therefore the expectation is not fulfilled; $ii)$ an event happens without an explicit corresponding expectation.

The operational counterpart of this declarative semantics is the $\mathcal{S}$CIFF proof procedure described in Section 4. $\mathcal{S}$CIFF has been proven sound and complete in relevant cases [20].

3 Specifying a Choreography in the $\mathcal{S}$CIFF Framework

In this section we develop a simple example in the $\mathcal{S}$CIFF framework. To our purposes, let us consider a revised version of the choreography proposed in [3]. The choreography (shown in Figure 1) models a 3-party interaction, in which a supplier coordinates with its warehouse in order to sell and ship electronic devices. Due to some laws, the supplier should trade only with customers who do not belong to a publicly known list of banned countries.

The choreography starts when a $Customer$ communicates a purchase order to the $Supplier$. $Supplier$ reacts to this request asking the $Warehouse$ about the availability of the ordered item. Once $Supplier$ has received the response, it decides to cancel or confirm the order, basing this choice upon $Item$'s availability and $Customer$'s country. In the former case, the choreography terminates, whereas in the latter one a concurrent phase is performed: $Customer$ sends an order payment, while $Warehouse$ handles the item's shipment. When both the payment and the shipment confirmation are received by $Supplier$, it delivers a final receipt to the $Customer$. The specification of this choreography is given in Spec. 3.1 [2] . The events are represented in the form $msgType(sender, receiver, content_1, \ldots, content_n)$, where the $msgType$, $sender$, $receiver$ and $content_i$ retain their intuitive meaning.

(IC_1) specifies that, when $Customer$ sends to $Supplier$ the purchase order, including the requested $Item$ and his/her $Country$, $Supplier$ should request $Item$'s availability to $Warehouse$. $Warehouse$ should respond within 10 minutes to $Supplier$'s request giving the corresponding quantity Qty (IC_2). The deadline is imposed as a CLP constraint over the variable T_{qty}, that represents the time in which the response is sent.

[2] For the sake of clarity, we omit roles specification, which may be simply expressed in the KB_{chor}. Moreover, although it is possible to introduce expectations also in the body of the $\mathcal{IC}_{chor}$, here we show an example where the bodies of the rules contain only happened events.

Specification 3.1 $\mathcal{IC}_{chor}$ specification of the example in figure 1

$$\begin{array}{l}\mathbf{H}(purchase\_order(Customer, Supplier, Item, Country), T_{po}) \\ \rightarrow \mathbf{E}(check\_availability(Supplier, Warehouse, Item), T_{ca}) \wedge T_{ca} > T_{po}\end{array} \quad (IC_1)$$

$$\begin{array}{l}\mathbf{H}(check\_availability(Supplier, Warehouse, Item), T_{ca}) \\ \rightarrow \mathbf{E}(inform(Warehouse, Supplier, Item, Qty), T_{qty}) \\ \quad \wedge T_{qty} > T_{ca} \wedge T_{qty} < T_{ca} + 10\end{array} \quad (IC_2)$$

$$\begin{array}{l}\mathbf{H}(purchase\_order(Customer, Supplier, Item, Country), T_{po}) \\ \quad \wedge \mathbf{H}(inform(Warehouse, Supplier, Item, Qty), T_{qty}) \\ \rightarrow \mathbf{E}(accept\_order(Supplier, Customer, Item), T_{ao}) \\ \quad \wedge ok(Qty, Country) \wedge T_{ao} > T_{po} \wedge T_{ao} > T_{qty} \\ \vee \mathbf{E}(reject\_order(Supplier, Customer, Item), T_{ro}) \\ \quad \wedge \neg ok(Qty, Country) \wedge T_{ro} > T_{po} \wedge T_{ro} > T_{qty}\end{array} \quad (IC_3)$$

$$\begin{array}{l}\mathbf{H}(accept\_order(Supplier, Customer, Item), T_{ao}) \\ \rightarrow \mathbf{E}(shipment\_order(Supplier, Warehouse, Item, Customer), T_{so}) \\ \quad \wedge \mathbf{E}(payment(Customer, Supplier, Item), T_p) \wedge T_{so} > T_{ao} \wedge T_p > T_{ao}\end{array} \quad (IC_4)$$

$$\begin{array}{l}\mathbf{H}(shipment\_order(Supplier, Warehouse, Item, Customer), T_{so}) \\ \rightarrow \mathbf{E}(request\_details(Warehouse, Customer), T_{rd}) \wedge T_{rd} > T_{so}\end{array} \quad (IC_5)$$

$$\begin{array}{l}\mathbf{H}(request\_details(Warehouse, Customer), T_{rd}) \\ \rightarrow \mathbf{E}(inform(Customer, Warehouse, Details), T_{det}) \wedge T_{det} > T_{rd}\end{array} \quad (IC_6)$$

$$\begin{array}{l}\mathbf{H}(shipment\_order(Supplier, Warehouse, Item, Customer), T_{so}) \\ \quad \wedge \mathbf{H}(inform(Customer, Warehouse, Details), T_{det}) \\ \rightarrow \mathbf{E}(confirm\_shipment(Warehouse, Supplier, Item), T_{cs}) \wedge T_{cs} > T_{so} \wedge T_{cs} > T_{det}\end{array} \quad (IC_7)$$

$$\begin{array}{l}\mathbf{H}(payment(Customer, Supplier, Item), T_p) \\ \quad \wedge \mathbf{H}(confirm\_shipment(Warehouse, Supplier, Item), T_{cs}) \\ \rightarrow \mathbf{E}(delivery(Supplier, Customer, Item, Receipt), T_{del}) \wedge T_{del} > T_{cs} \wedge T_{del} > T_p\end{array} \quad (IC_8)$$

Specification 3.2 KB_{chor} with some banned countries

```
ok( Qty, Country):-
     Qty>0,
     not banned_country( Country).

banned_country( shackLand).
banned_country( badLand).
```

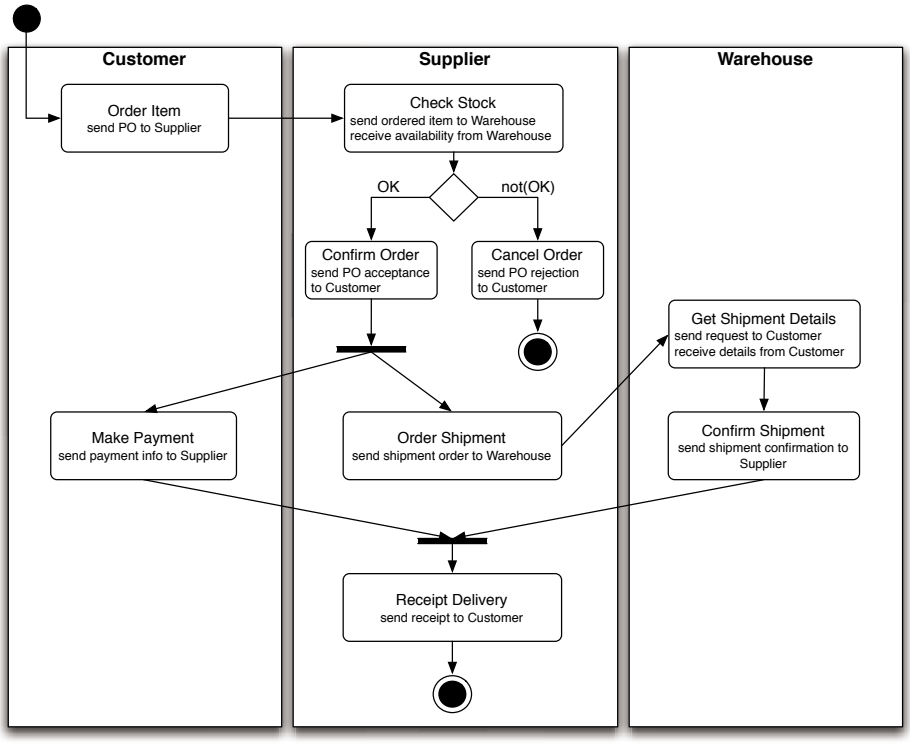

Fig. 1. A simple choreography example

After having received the requested quantity, $Supplier$ decides whether to accept or reject $Customer$'s order (IC_3). As we have pointed out, the decision depends upon the quantity and the $Country$ the $Customer$ belongs to; $Supplier$ may accept the order only when Qty is positive and customer's $Country$ is not in the list of banned countries. This last condition has been expressed using a predicate defined in the KB_{chor}, showed in Spec. 3.2. If $Supplier$ has accepted the purchase order, then $Customer$ is expected to pay for the requested $Item$ and, at the same time, $Supplier$ will send a shipment order to $Warehouse$, communicating the involved $Item$ and $Customer$'s identity (IC_4). $Warehouse$ will use $Customer$'s identity in order to communicate with him/her and asking for shipment details (IC_5). [3]

When $Customer$ receives the request for details, then he/she is expected to respond giving his/her own $Details$ (IC_6). After having received them, $Warehouse$ should sends to $Supplier$ a shipment confirmation (IC_7). Finally, (IC_8) states that when both the payment and the shipment confirmation actually happen $Supplier$ is expected to deliver a $Receipt$ to $Customer$.

[3] This could be viewed, at a higher level, as a channel passing mechanism, since $Customer$ is used as a content part of the first message, and as receiver of the second one.

4 Run-Time Conformance Verification of Web Services Interactions

In Section 2 we have introduced some key concepts of our approach, in particular *happened events* and *expectations*, and a declarative semantics, together with the notion of *fulfillment* and *violation* of a choreography specification. In this section we show how, by exploiting these concepts, it is possible to perform the run-time conformance check, by the operational counterpart of the declarative semantics, represented by the $\mathcal{S}$CIFF proof procedure. We also show how $\mathcal{S}$CIFF operates on a concrete interaction example.

4.1 Detecting Fulfilment and Violation: The $\mathcal{S}$CIFF Proof Procedure and the *SOCS-SI* Tool

We developed the $\mathcal{S}$CIFF proof procedure for the automatic verification of compliance of interactions w.r.t. a given choreography. Then, we developed a Java-based application, *SOCS-SI*, that receives as input the specification of a choreography and the happening events, and provides as output the answer about the conformance issue. *SOCS-SI* uses the $\mathcal{S}$CIFF proof procedure as inference engine, and provides a Graphical User Interface for accessing the results of the conformance task.

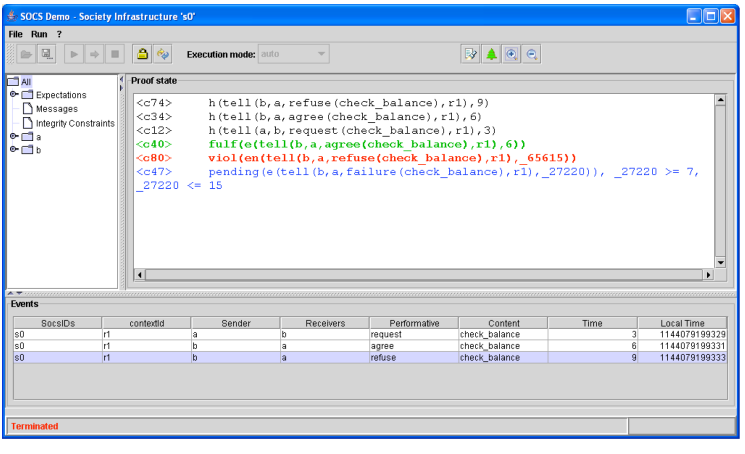

Fig. 2. The *SOCS-SI* tool

The $\mathcal{S}$CIFF proof procedure considers the $\mathbf{H}$ events as predicates defined by a set of incoming atoms, and is devoted to generate expectations corresponding to a given set of happened events and to check that expectations indeed match with those events. The proof procedure is based on a rewriting system transforming one node to another (or to others) as specified by rewriting steps called *transitions*. A node can be either the special node *false*, or defined by the following tuple

$$T \equiv \langle R, CS, PSIC, \mathbf{PEND}, \mathbf{HAP}, \mathbf{FULF}, \mathbf{VIOL} \rangle$$

where

- R is the resolvent (initially set to the goal G);
- CS is the constraint store (à la CLP [15]);
- $PSIC$ is a set of implications, derived from the $\mathcal{IC}_{chor}$;
- **PEND** is the set of (pending) expectations (i.e., expectations have not been fulfilled (yet), nor they have been violated=;
- **HAP** is the history of happened events;
- **FULF** and **VIOL** are the sets of fulfilled and violated expectations, respectively.

We cannot report here all the transitions, due to lack of space; the interested reader can refer to [21]. As an example, the *fulfilment* transition is devoted to prove that an expectation $\mathbf{E}(X, T_x)$ has been fulfilled by an event $\mathbf{H}(Y, T_y)$. Two nodes are generated: in the first, X and T_x are unified respectively with Y and T_y, and the expectation is fulfilled (i.e., it is moved to the set **FULF**); in the second a new constraint that imposes disunification between (X, T_x) and (Y, T_y) is added to the constraint store CS. At the end of the computation, a *closure* transition is applied, and all the expectations remaining in the set **PEND** are considered as violated. The $\mathcal{S}$CIFF proof procedure can be downloaded at `http://lia.deis.unibo.it/research/sciff/`.

The *SOCS-SI* software tool is a Java-based application, that provides to the user a GUI to access the outcomes of the $\mathcal{S}$CIFF proof procedure. It has been developed to accept events that happen dynamically, from various events source. It accepts, as event source, also a log file containing the log of the relevant events. In this way, it is possible to perform the conformance verification $i)$ at run-time, by checking immediately the incoming happened events (possibly raising violations as soon as possible), and $ii)$ a posteriori, analyzing log files. When performing run-time verification, if time events (i.e., events that represent the current time instant) are provided (possibly by an external source, e.g. a clock), *SOCS-SI* is able to use such information to detect deadline expirations with a discrete approximation to the nearest greater time instant. A snapshot of *SOCS-SI* GUI is shown in Figure 2. *SOCS-SI* can be downloaded at `http://www.lia.deis.unibo.it/research/socs_si/socs_si.shtml`.

4.2 Example of Run-Time Conformance Verification

In our scenario, the criminal $bankJob$ beagle wants to buy a device from the online shop $devOnline$, whose warehouse is $devWare$. $devOnline$ is quite greedy, and therefore trades with everyone, without checking if the customer comes from one of the banned countries. As a consequence, even if $bankJob$ comes from $shackLand$, one of the banned countries, $devOnline$ sells him the requested device, thus violating the choreography. Table 1 contains the log of the scenario from the viewpoint of $devOnline$; note that messages are expressed in high level way, abstracting from the SOAP exchange format.

When the first event (labeled m_1 in Table 1) happens, (IC_1) is triggered, and an expectation about $devOnline$'s behaviour is consequently generated:

$$\mathbf{PEND} = \{\ \mathbf{E}(check\_availability(devOnline, Warehouse, device), T_{ca}) \wedge T_{ca} > 2\}$$

Table 1. Log of messages exchanged by $devOnline$ in our scenario

Id	message	sender	receiver	content	time
m_1	purchase_order	bankJob	devOnline	[device,shackLand]	2
m_2	check_availability	devOnline	devWare	[device]	3
m_3	inform	devWare	devOnline	[device,3]	10
m_4	accept_order	devOnline	bankJob	[device]	12
m_5	shipment_order	devOnline	devWare	[device,bankJob]	13
m_6	confirm_shipment	devWare	devOnline	[device]	16
m_7	payment	bankJob	devOnline	[device]	19
m_8	delivery	devOnline	bankJob	[device,receipt]	21

The happening of m_2 fullfills the pending expectation and matches with the body of (IC_2), generating a new one:

$$\begin{aligned}\mathbf{FULF} &= \{\ \mathbf{E}(check\_availability(devOnline, devWare, device), 3)\}\\ \mathbf{PEND} &= \{\ \mathbf{E}(inform(devWare, devOnline, device, Qty), T_{Qty})\\ &\quad \wedge T_{qty} > 3 \wedge T_{qty} < 13\}\end{aligned}$$

The happening of m_3 fulfills the current pending expectation respecting the deadline. Moreover, it triggers (IC_3), and two different hypotheses are considered (acceptance and rejection of the order). However, since the predicate `ok(3,shackLand)` is evaluated by $\mathcal{S}$CIFF to false, only the expectation about the order rejection is considered:

$$\begin{aligned}\mathbf{FULF} &= \{\ \mathbf{E}(check\_availability(devOnline, devWare, device), 3),\\ &\quad \mathbf{E}(inform(devWare, devOnline, device, 3), 10)\}\\ \mathbf{PEND} &= \{\ \mathbf{E}(reject\_order(devOnline, bankJob, device), T_{ro})\\ &\quad \wedge T_{ro} > 3 \wedge T_{ro} > 10\}\end{aligned}$$

As a consequence, when $devOnline$ accepts the purchase order of $bankJob$ sending the message m_4, the $\mathcal{S}$CIFF proof procedure detects a violation, since m_4 is not explicitly expected.

5 Discussion and Conclusion

In this paper, we have addressed the run-time conformance verification issue w.r.t. web services interaction. We propose to use the $\mathcal{S}$CIFF framework and the *SOCS-SI* tool, and to adapt them to the Web Services peculiar features. Indeed, the presented proposal is part of a bigger and complex framework, sketched in Figure 3. We envisage two major research directions:

1. a *translation issue*, where a choreography specification is automatically translated to its corresponding $\mathcal{IC}_{chor}$ and KB_{chor}, together with its $\mathcal{G}_{chor}$;
2. a *verification issue*, that consists in three different types of verification (each one addressed by its own proof-theoretic verification tool).

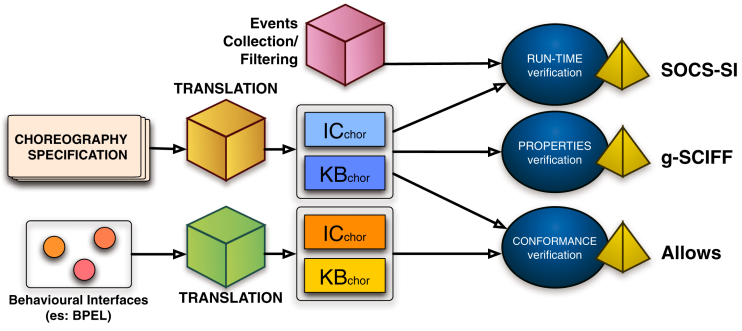

Fig. 3. Global view of our ongoing research

With respect to the translation issue, currently the link between known and widely accepted formalisms, such as BPEL and WS-CDL, and our model, is missing. We are aware that this part is of a fundamental importance, in order to effectively support our framework. Therefore, we are currently studying a translation algorithm capable to automatically convert a WS-BPEL/WS-CDL specification to our formalism. We are also working on the automatic translation of graphical specifications, like for example BPMN [22]. A first algorithm, that translates a simple graphical workflow language, has been presented in [23].

With respect to the verification issue, we envisage three possible types of verification. The first type has been addressed in this work, and is tackled by the *SOCS-SI* tool and the $\mathcal{S}$CIFF proof-procedure. Noticeably, $\mathcal{S}$CIFF operates indifferently off-line on a complete log or at run-time on events as soon as they happen. Therefore, the same tool is able to perform the conformance verification at run-time or a-posteriori. To support this type of verification, however, a low-level mechanism for capturing the interaction events is needed. We do not address this issue, but we recognize it is an important one, to the end of developing a real system.

The second type of verification is about the proof of "high level" properties: in fact, besides control-flow properties (like deadlock, liveness, etc.), it is interesting to check if a group of peers, whose interaction follows a given choreography, can benefit of particular properties. E.g., in a e-commerce scenario, a buyer is guaranteed to receive the good he paid for, and the seller is guaranteed to be paid. Assuming the peers behave correctly (w.r.t. the choreography), the fact that a property holds or not is a consequence of how the choreography has been specified. To this end, we have developed the g-$\mathcal{S}$CIFF, an extension of the $\mathcal{S}$CIFF proof procedure, and we applied it to verify some properties of a security protocol [24]. Other approaches tackle this issue by means of model checking techniques: e.g., in [25], the authors use model checking techniques to formally verify that requirements are met by web service systems, and to tackle the property verification issue. High level properties are expressed by means of *Linear Temporal Logic* formulas, and then verified using model checkers like SPIN or NuSMV.

The third type of verification aims to check if a web service, described by its behavioural interface, can play a given role within a choreography. This issue is known as "A-Priori Conformance Verification", and it has been tackled by many works in the

research literature ([5,6], to cite some). We have addressed this problem in [26], combining $\mathcal{S}$CIFF and g-$\mathcal{S}$CIFF: the interested reader can refer to such paper for a comparison of the mentioned approaches.

We would like to clearly state that this is an ongoing work, and that it is far from being concluded. Several aspects have not yet been exhaustively researched: beside the automatic translation from other formalisms to our model, we need to test our approach on significant choreography specifications (currently, we have performed some tests on global interaction protocols for multi agent systems [27]).

However, we claim that our proposal indeed offers some noticeable advantages. First, the proposed specification language is declarative, intuitive and of highly expressive nature; $\mathcal{IC}_{chor}$ are human readable and clearly represent how the choreography should be followed by the interacting services. Moreover, a single specification language can be used to perform several different types of verification. Second, we claim the importance of modeling messages data and content as well as control flow among them. This kind of "content awareness" is required to model constraints about the content of messages and to formalize decisions or, more generally, pieces of knowledge of the choreography. Moreover, deadline specification is easily performed by means of CLP constraints, and business rules can be seamlessly expressed in the choreography knowledge base. Since the knowledge base is defined as an abductive logic program, powerful forms of reasoning, such as planning and diagnosis, can be easily integrated into the framework.

Acknowledgements. This work has been partially funded by the MIUR Projects PRIN 2005: *"Linguaggi per la specifica e la verifica di protocolli di interazione fra agenti"* and *"Vincoli e preferenze come formalismo unificante per l'analisi di sistemi informatici e la soluzione di problemi reali"*.

References

1. 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) Available at http://www-128.ibm.com/developerworks/library/specification/ws-bpel/.
2. W3C: (Web services choreography description language version 1.0) Home Page: `http://www.w3.org/TR/ws-cdl-10/`.
3. Barros, A., Dumas, M., Oaks, P.: A critical overview of the web services choreography description language (WS-CDL). BPTrends (2005)
4. van der Aalst, W., Dumas, M., ter Hofstede, A., Russell, N., Verbeek, H.M.W., Wohed, P.: Life after BPEL? In Bravetti, M., Kloul, L., Zavattaro, G., eds.: EPEW/WS-FM. Volume 3670 of LNCS., Springer (2005) 35–50
5. 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.: EPEW/WS-FM. Volume 3670 of LNCS., Springer (2005)
6. Kazhamiakin, R., Pistore, M.: A parametric communication model for the verification of bpel4ws compositions. In: EPEW/WS-FM. (2005) 318–332
7. van der Aalst, W.: Business alignment: Using process mining as a tool for delta analysis and conformance testing. Requirements Engineering Journal **to appear** (2005)

8. Ouyang, C., van der Aalst, W., Breutel, S., Dumas, M., ter Hofstede, A., , Verbeek, H.: Formal semantics and analysis of control flow in ws-bpel. Technical Report BPM-05-15, BPMcenter.org (2005)
9. Rozinat, A., van der Aalst, W.M.P.: Conformance testing: Measuring the fit and appropriateness of event logs and process models. In Bussler, C., Haller, A., eds.: Business Process Management Workshops. Volume 3812. (2005) 163–176
10. Kakas, A.C., Kowalski, R.A., Toni, F.: Abductive Logic Programming. Journal of Logic and Computation **2**(6) (1993) 719–770
11. (Societies Of ComputeeS (SOCS): a computational logic model for the description, analysis and verification of global and open societies of heterogeneous computees. IST-2001-32530) Home Page: `http://lia.deis.unibo.it/Research/SOCS/`.
12. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Compliance verification of agent interaction: a logic-based software tool. Applied Artificial Intelligence **20**(2-4) (2006) 133–157
13. Bry, F., Eckert, M., Patranjan, P.: Reactivity on the web: Paradigms and applications of the language xchange. Journal of Web Engineering **5**(1) (2006) 3–24
14. Lloyd, J.W.: Foundations of Logic Programming. 2nd edn. Springer-Verlag (1987)
15. Jaffar, J., Maher, M.: Constraint logic programming: a survey. Journal of Logic Programming **19-20** (1994) 503–582
16. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: The SOCS computational logic approach for the specification and verification of agent societies. In Priami, C., Quaglia, P., eds.: Global Computing: IST/FET Intl. Workshop, GC 2004 Rovereto, Italy, March 9-12. Volume 3267 of LNAI. Springer-Verlag (2005) 324–339
17. Kakas, A.C., Mancarella, P.: On the relation between Truth Maintenance and Abduction. In Fukumura, T., ed.: Proc. PRICAI-90, Nagoya, Japan, (Ohmsha Ltd.) 438–443
18. Fung, T.H., Kowalski, R.A.: The IFF proof procedure for abductive logic programming. Journal of Logic Programming **33**(2) (1997) 151–165
19. Denecker, M., Schreye, D.D.: SLDNFA: an abductive procedure for abductive logic programs. Journal of Logic Programming **34**(2) (1998) 111–167
20. Gavanelli, M., Lamma, E., Mello, P.: Proof of properties of the SCIFF proof-procedure. Technical Report CS-2005-01, Computer science group, Dept. of Engineering, Ferrara University (2005) http://www.ing.unife.it/informatica/tr/.
21. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: The sciff abductive proof-procedure. In: Proc. of the 9th National Congress on Artificial Intelligence, AI*IA 2005. Volume 3673 of LNAI., Springer-Verlag (2005) 135–147
22. Initiative, B.P.M.: (Business process modeling notation)
23. Chesani, F., Ciampolini, A., Mello, P., Montali, M., Storari, S.: Testing guidelines conformance by translating a graphical language to computational logic, Workshop on AI techniques in healthcare. In conjunction with ECAI (2006) To appear.
24. 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., Ricci, A., eds.: Proc. of ESAW'05, Ege University (2005) 283–295
25. Kazhamiakin, R., Pistore, M., Roveri, M.: Formal verification of requirements using spin: A case study on web services. In: Proc. of the Software Engineering and Formal Methods (SEFM'04), Washington, DC, USA, IEEE Computer Society (2004) 406–415
26. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Montali, M.: An abductive framework for a-priori verification of web services. In Maher, M., ed.: Principles and Practice of Declarative Programming (PPDP'06), ACM Press (2006) to appear.
27. (The socs protocols repository) Available at `http://edu59.deis.unibo.it:8079/SOCSProtocolsRepository/jsp/index.jsp`.