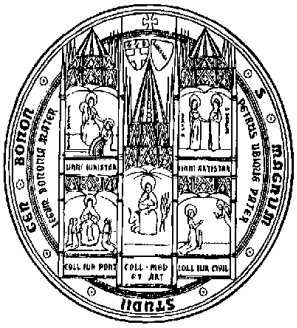


Università degli Studi di Bologna
DEIS

Towards a DecSerFlow Declarative Semantics based on Computational Logic

Federico Chesani Paola Mello Marco Montali
Sergio Storari

March 14, 2007

DEIS Technical Report no. DEIS- LIA-07-001 *LIA Series no. 79*

Towards a DecSerFlow Declarative Semantics based on Computational Logic

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

[1] *DEIS - Dip. di Elettronica, Informatica e Sistemistica*
University of Bologna
Viale Risorgimento, 2 - 40136, Bologna
Italy
[fchesani | pmello | mmontali]@deis.unibo.it
[2] *ENDIF - Engeneering Dept. in Ferrara*
University of Ferrara
Via Saragat, 1 - 44100, Ferrara
Italy
strsrg@unife.it

March 14, 2007

Abstract. In this paper we exploit a computational logic-based framework, called $\mathcal{S}$CIFF, for the formalization of DecSerFlow. DecSerFlow is a graphical, extendible high-level language for the declarative specification of service flows, and is grounded on LTL. $\mathcal{S}$CIFF was originally developed in the context of the SOCS european project , where we addressed the issue of providing a formal language to define and verify interaction protocols in open environments.

More specifically, in this work we show that $\mathcal{S}$CIFF is concretely able to formalize the DecSerFlow core template formulae, tackling two complementary issues: on one hand, it is possible to specify $\mathcal{S}$CIFF rules by using an intuitive and user-friendly graphical language; on the other hand, a DecSerFlow model may be grounded not only on LTL but also on an abductive framework, acquiring some new advantages and features.

Finally, we propose to extend DecSerFlow by exploiting some useful features of the $\mathcal{S}$CIFF framework, like for example the explicit notion of time, which could be used to specify temporal constraints and deadlines.

Contents

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 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: WS-BPEL [6] and WS-CDL [35].

Both these initiatives however have missed to tackle some important issues. We agree with the view [8, 32] that both WS-BPEL and WS-CDL languages lack of declarativeness, and more dangerous, they both lack of an underlying formal model and semantics. Hence, issues like *conformance testing*, *interoperability checking* [7] and *verification of properties* are not fully addressed by the current proposals.

To overcome these limits, van der Aalst et al. have proposed DecSerFlow [33], a 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. Hence, it does not give a complete and procedural specification of services, but concentrates on what are the (minimal) constraints to be fulfilled in order to successfully complete the interaction. Beyond its appealing graphical representation, DecSerFlow has a mapping to Linear Temporal Logic (LTL) [16, 23], and thus it may be used to verify or enforce conformance of service flows, and also to directly enact their execution.

Taking inspiration by the many analogies between the Web Services research field and the Multi Agent System (MAS) field [7], in this paper we exploit a computational logic-based framework, called $\mathcal{S}$CIFF, for the formalization of DecSerFlow. $\mathcal{S}$CIFF was originally developed in the context of the SOCS european project [31], where we addressed the issue of providing a formal language to define and verify interaction protocols in an open multi-agent setting. Hence, within the $\mathcal{S}$CIFF framework, a language suitable for specifying global interaction protocols is provided; a formal semantics is provided too, based on abductive logic programming [18]. $\mathcal{S}$CIFF considers a set of interacting peer as an open society, formalizing interaction protocols as a set of global rules which constrain the external and observable behaviour of

participants (for this reason, global rules are called Social Integrity Constraints).

The operational counterpart of $\mathcal{S}$CIFF is an abductive proof procedure capable to verify at run-time (or a-posteriori using an event log) if the peers behave in a conformant manner w.r.t. the modeled interaction protocol. Moreover, on top of the $\mathcal{S}$CIFF proof procedure a tool (namely *SOCS-SI* [3]) has been developed for automatically analyze and verify peers interactions w.r.t. a protocol expressed in the language above.

Some initial applications of $\mathcal{S}$CIFF in the field of web service choreographies can be found in [2], where the problem of verifying if a set of interacting services conform to a choreography specification is faced, and in [5], where a new proof procedure is derived from $\mathcal{S}$CIFF to perform the interoperability check between a peer behavioural interface and a choreography specification.

More specifically, in this work we show that the Social Integrity Constraints introduced in the SOCS social model are concretely able to formalize the DecSerFlow template formulae; in this way, we tackle two complementary issues: on one hand, it is possible to specify Social Integrity Constraints by using an intuitive, extendible and user-friendly graphical language; on the other hand, a DecSerFlow model may be grounded not only on LTL but also on our abductive framework, acquiring some new advantages and features.

Through the mapping to our formalism, it is possible:

- to automatically translate a DecSerFlow model into the $\mathcal{S}$CIFF framework
- to use *SOCS-SI* to check whether a set of existing services behave in a conformant manner w.r.t. the defined model
- to propose DecSerFlow extensions by exploiting some useful features of the $\mathcal{S}$CIFF framework, like for example the explicit notion of time, which could be used to specify temporal constraints and deadlines.

The paper is organized as follows: in Section 2 we introduce the $\mathcal{S}$CIFF framework, provide its declarative semantics and briefly show how the *conformance verification* issue can be addressed by using it. Then, in Section 3.1 a brief description of DecSerFlow is given. In Section 4 we specify DecSerFlow as an abductive logic program, giving an intuition about how the DecSerFlow template formulae could be translated into the $\mathcal{S}$CIFF language. The concrete mapping is then described in Sections 5, 6 and 7, where respectively existence, relation and negation formulae are tackled. Finally, Section 8 presents some preliminary proposal about extending the basic DecSerFlow template formulae to deal also with conjunctions/disjunctions of activities and explicit temporal constraints and deadlines. Conclusions and future works follow in Section 10.

2 The $\mathcal{S}$CIFF framework

In this section, we present the $\mathcal{S}$CIFF framework, describing how the conversational part of an interaction protocol as well as its static knowledge can be suitably expressed within the framework. Moreover, we provide the declarative semantics of $\mathcal{S}$CIFFand give a formal definition of conformance.

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 completely abstracts 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* [1] , 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. 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 [9], 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; it could also be the case that the same event could happen many times [2] .

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 at which the event happened. The set of all the events that have happened during a protocol execution constitutes its log (or execution trace).

[1] à la logic programming.

[2] 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.

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"*.

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 negative expectations (i.e. of what is expected not to happen).

Positive expectations about events come with form

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

where $Event$ and $Time$ can be variables, 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).

Conversely, negative expectations about events come with form

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

The complete syntax of events, happened events and expectations is given in table 2.1.

Table 2.1 Syntax of events and expectations

Event	::=	$\mathbf{H}(GroundTerm[, Integer])$
Expectation	::=	*PosExp* | *NegExp*
PosExp	::=	$\mathbf{E}(Term[, Variable \mid Integer])$
NegExp	::=	$\mathbf{EN}(Term[, Variable \mid Integer])$
Literal	::=	[**not**]*Atom*
AbducibleLiteral	::=	[**not**]*AbducibleAtom*

Given the notions of *happened event* and of (positive or negative) *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 an interaction model, and it will be addressed in the next section. The second issue, instead, is inherently related to the problem of establishing if a participant is indeed behaving in a conformant manner w.r.t. a given interaction model: the solution proposed by the $\mathcal{S}$CIFF framework is presented in Section 2.5.

2.2 Social Integrity Constraints

Social Integrity Constraints $\mathcal{IC}_S$ are forward rules, of the form

$$Body \rightarrow Head$$

whose *Body* can contain literals and (conjunctions of happened and expected) events, and whose *Head* can contain (disjunctions of) conjunctions of expectations. In table 2.2 we report the formal definition of the grammar, where *Atom* and $Term$ have the usual meaning in Logic Programming [27] and *Constraint* is interpreted as in Constraint Logic Programming [17].

Table 2.2 Syntax of Integrity Constraints (ICs)

$\mathcal{IC}_S$	::=	$[IC]^\star$
IC	::=	$Body \rightarrow Head$
$Body$	::=	$(Event \mid Expectation \mid AbducibleLiteral)\ [\ \wedge BodyLiteral\ ]^\star$
$BodyLiteral$	::=	$Event \mid ExtLiteral$
$Head$	::=	$HeadDisjunct\ [\ \vee HeadDisjunct\ ]^\star \mid false$
$HeadDisjunct$	::=	$ExtLiteral\ [\ \wedge ExtLiteral]^\star$
$ExtLiteral$	::=	$Literal \mid AbducibleLiteral \mid Expectation \mid Constraint$

CLP constraints [17] and Prolog predicates 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 (or not to happen). For example, time conditions might define orderings between messages, or enforce deadlines.

$\mathcal{IC}_S$ 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 rule:

$$\begin{aligned} &\mathbf{H}(request(User, Bank, withdraw(X)), T_r) \\ \rightarrow &\mathbf{E}(give(Bank, User, money(X)), T_a) \wedge T_a < T_r + 24 \\ \vee &\mathbf{E}(tell(Bank, User, not\_possible, reason(\ldots)), T_p) \wedge T_p > T_r. \end{aligned} \tag{1}$$

2.3 The Social Organizational Knowledge Base

Integrity Constraints are a suitable tool for effectively define 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 requested money, 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 a knowledge in the *Social Organizational Knowledge Base* KB. The KB specifies declaratively pieces of knowledge of the interaction model, such as roles descriptions, list of participants, etc. KB 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 table 2.3.

Table 2.3 Syntax of the Knowledge Base

$$
\begin{array}{rcl}
KB_{DSF} & ::= & [Clause]^{\star} \\
Clause & ::= & Head \leftarrow Body \\
Head & ::= & Atom \\
Body & ::= & ExtLiteral\ [\ \wedge\ ExtLiteral\ ]^{\star} \mid true \\
ExtLiteral & ::= & Literal \mid AbducibleLiteral \mid ExpLiteral \mid Constraint
\end{array}
$$

In the following, we will also use the classical Prolog notation for representing a clause (i.e. `Head :- Body`).

Furthermore, in our vision an interaction can be *goal directed*, i.e. a specific goal $\mathcal{G}$ 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 (i.e. by the same set of $\mathcal{IC}_S$), can be used seamlessly for achieving different goals. Such goals can be defined like the clauses of the KB. Typically, a goal is defined as expectations about the outcomes of the interaction, e.g., in a web service environment, in terms of messages (and their contents) that should be exchanged (or, more generally, in terms of activities that should be executed). If no particular goal is required to be achieved, $\mathcal{G}$ is bound to *true*.

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

In the $\mathcal{S}$CIFF framework, an interaction model is interpreted in terms of an Abductive Logic Program (ALP). In general, an ALP [18] 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 [19], Fung and Kowalski [14], Denecker and De Schreye [11], etc.).

The idea we exploited in the $\mathcal{S}$CIFF framework is to adopt abduction [4] to dynamically *generate* the expectations and to perform the *conformance checking* between expectations and happened event (to ensure that they are following the interaction model). 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.

An interaction protocol specification $\mathcal{S}$ is defined by the triple:

$$\mathcal{S} \equiv \langle KB, \mathcal{E}, \mathcal{IC}_S \rangle$$

where:

- KB is the *Social Organizational Knowledge Base*,
- $\mathcal{E}$ is the set of *abducible predicates*, namely
  - positive expectations (functor **E**)
  - negative expectations (functor **EN**)

  and
- $\mathcal{IC}_S$ is the set of *Social Integrity Constraints*.

The declarative semantics of an interaction specification is given for each specific execution trace. We call a specification grounded on an execution trace an *instance* of the specification.

Definition 2.1 (Instance) *Given an abductive specification $\mathcal{S}$ and an execution trace* **HAP**, *$\mathcal{S}_{\mathbf{HAP}}$ represents the pair $\langle \mathcal{S}, \mathbf{HAP} \rangle$ and is called the instance of the specification.*

We adopt an abductive semantics for an execution instance. The abductive computation produces a set Δ of hypotheses, which is partitioned into a set Δ' of general hypotheses and a set **EXP** of *expectations*. The set of abduced literals should entail the goal and satisfy the integrity constraints.

Definition 2.2 (Abductive Explanation) *Given an abductive specification $\mathcal{S} \equiv \langle KB, \mathcal{E}, \mathcal{IC}_S \rangle$, an instance $\mathcal{S}_{\mathbf{HAP}}$ of $\mathcal{S}$, and a goal $\mathcal{G}$, Δ is an abductive explanation of $\mathcal{S}_{\mathbf{HAP}}$ if:*

$$Comp(KB_S \cup \mathbf{HAP} \cup \Delta) \cup CET \cup T_{\mathcal{X}} \models \mathcal{IC}_S \tag{2}$$

$$Comp(KB_S \cup \mathbf{HAP} \cup \Delta) \cup CET \cup T_{\mathcal{X}} \models \mathcal{G} \tag{3}$$

where Comp represents the completion *of a theory, CET is Clark's Equational Theory [10], $T_{\mathcal{X}}$ is the theory of constraints [17] and the notion of entailment is grounded on Kunen's three valued semantics [25].*

We also require consistency between positive and negative expectations (an event cannot be expected to happen and not to happen at the same time).

Definition 2.3 (E-consistency) *A set of expectations* **EXP** *is e-consistent if and only if for each (ground) term p:*

$$\{\mathbf{E}(p), \mathbf{EN}(p)\} \not\subseteq \mathbf{EXP} \tag{4}$$

At this point it is possible to define the concepts of *fulfillment* and *violation* of a set **EXP** of expectations. Fulfillment requires:

- all the **E** expectations to have a matching happened event
- all the **EN** expectations to have no matching happened event

Thus, the following definition establishes a link between happened events and expectations, by requiring positive expectations to be matched by events, and negative expectations not to be matched by events.

Definition 2.4 (Fulfillment) *Given an execution instance* **HAP**, *a set of expectations* **EXP** *is* fulfilled *by* **HAP** *if and only if for all (ground) terms p:*

$$\forall \mathbf{E}(p) \in \mathbf{EXP} \Rightarrow \mathbf{H}(p) \in \mathbf{HAP} \qquad \forall \mathbf{EN}(p) \in \mathbf{EXP} \Rightarrow \mathbf{H}(p) \notin \mathbf{HAP} \tag{5}$$

Otherwise, **EXP** is *violated* by **HAP**.

We can now give a formal definition of *conformance*.

Definition 2.5 (Conformance) *Given an execution trace* **HAP**, *a specification* $\mathcal{S}$ *and a goal* $\mathcal{G}$, **HAP** *is conformant to* $\mathcal{S}$ *w.r.t* $\mathcal{G}$ *if and only if there exists an abductive explanation* Δ *of* $\mathcal{S}_{\mathbf{HAP}}$ *w.r.t.* $\mathcal{G}$ *such that definitions 2.3 and 2.4 hold. In this case, we write* $\mathcal{S}_{\mathbf{HAP}} \models_{\Delta} \mathcal{G}$.

In the following, we will simply state that a log is conformant to an interaction specification if it is conformant to the specification w.r.t. the goal *true*.

The last two definitions are useful to identify if two different interaction specifications exhibit the same behaviour w.r.t. the conformance evaluation of an arbitrary log.

Definition 2.6 (Implication w.r.t. conformance) *An interaction specification* $\mathcal{S}^1$ *implies w.r.t. conformance another specification* $\mathcal{S}^2$ *(*$\mathcal{S}^1 \xrightarrow{\mathcal{C}} \mathcal{S}^2$*) if and only if each execution trace evaluated as conformant to* $\mathcal{S}^1$ *w.r.t. to a goal* $\mathcal{G}$ *is evaluated as conformant to* $\mathcal{S}^2$ *w.r.t.* $\mathcal{G}$, *too. More formally,* $\mathcal{S}^1 \xrightarrow{\mathcal{C}} \mathcal{S}^2$ *if and only if*

$$\forall \mathbf{HAP}\ \forall \mathcal{G}\ \exists \Delta_1\ \mathcal{S}^1_{\mathbf{HAP}} \models_{\Delta_1} \mathcal{G} \rightarrow \exists \Delta_2\ \mathcal{S}^2_{\mathbf{HAP}} \models_{\Delta_2} \mathcal{G} \tag{6}$$

Definition 2.7 (Equivalence w.r.t. conformance) *Two specifications* $\mathcal{S}^1$ *and* $\mathcal{S}^2$ *are equivalent w.r.t. conformance (*$\mathcal{S}^1 \overset{\mathcal{C}}{\equiv} \mathcal{S}^2$*) iff* $\mathcal{S}^1 \xrightarrow{\mathcal{C}} \mathcal{S}^2$ *and* $\mathcal{S}^2 \xrightarrow{\mathcal{C}} \mathcal{S}^1$. *Two equivalent w.r.t. conformance specifications accept and reject the same execution traces.*

The operational counterpart of this declarative semantics is the $\mathcal{S}$CIFF proof procedure, briefly described in Section 2.5. $\mathcal{S}$CIFF has been proven sound and complete in relevant cases [15].

2.5 Detecting Fulfillment 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 interaction model. 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.

The $\mathcal{S}$CIFF proof procedure considers the **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

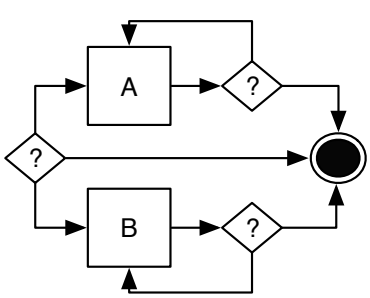
- R is the resolvent (initially set to the goal $\mathcal{G}$);
- CS is the constraint store (à la CLP [17]);
- $PSIC$ is a set of implications, derived from the $\mathcal{IC}_S$;
- **PEND** is the set of (pending) expectations, i.e. expectations have not been fulfilled (yet), nor they have been violated;
- **HAP** is the log of happened events;
- **FULF** and **VIOL** are the sets of fulfilled and violated expectations, respectively.

We do not report here all the transitions. As an example, the *fulfillment* 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 Y are unified, and the expectation is fulfilled (i.e., it is moved to the set **FULF**); in the second the new constraint $X \neq Y$ is added to the constraint store CS (it could be the case that another event, different than X, will fulfill the expectation). At the end of the computation, a *closure* transition is applied, and

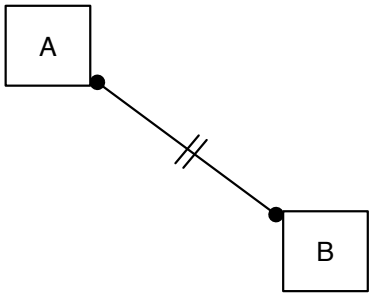
- all the **E** expectations remaining in the set **PEND** are considered as violated;
- all the **EN** expectations remaining in the set **PEND** are considered as fulfilled.

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 happens 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. *SOCS-SI* can be downloaded at `http://www.lia.deis.unibo.it/research/socs_si/socs_si.shtml`.



(a) Procedural style of modeling for the considered example.



(b) Declarative style of modeling for the considered example (DecSerFlow notation).

Figure 1. *Two different approaches for modeling the not coexistence bewteen two activities.*

3 A brief description of DecSerFlow

In this section we will briefly introduce the DecSerFlow language, just for the sake of clarity. For a detailed description of the language and its mapping to Linear Temporal Logic, see [33].

3.1 A Declarative Service Flow language

As suggested by its name, the main innovation of DecSerFlow is that it adopts a declarative style of modeling: the authors point out that following a declarative approach could give some relevant improvements to service specifications.

To illustrate the main difference between a declarative approach vs. a procedural one, they illustrate the simple example shown in figure 1.

In order to specify that two different activities should not be executed together (i.e. it is possible to execute the first or the latter activity multiple times, but the two activities exclude each other), a procedural language must explicitly represent all the possible executions, leading to some relevant problems:

- the process becomes over-specified;
- the modeler must introduce decision points to handle the possible executions, but it is not clear how and when these decisions should be evaluated.

Instead, by using a declarative language s.t. LTL, forbidding the coexistence of two activities A and B may be expressed in a simple and very compact way: $\neg(\diamond A \wedge \diamond B)$.

However, the main problem of languages like LTL is that they are hard to use and read by non-expert, hence the idea to develop a declarative graphical language (with a one-to-one mapping to LTL) as a middle tier between the user and the formal

language. Beyond this important feature, the translation of a graphical model to LTL enable the possibility to verify the conformance of an event log w.r.t. the designed model and also to automatically enact its execution.

3.2 Main concepts of DecSerFlow

The basic intuitive concepts of DecSerFlow are:

- activities, to model atomic logical unit of works;
- constraints, to represent relationships (in the sense of policies or business rules) between activities.

Constraints are given as templates, i.e. as relationships between two (or more) whatsoever activities. Each constraint template is then expressed as an LTL formula, hence the name "formulae" to indicate DecSerFlow relationships.

To extend DecSerFlow, it is sufficient to define a new graphical notation for the new template formula and specify the corresponding formalization. DecSerFlow core relationships are grouped into three families:

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

To reflect the autonomous nature of services, DecSerFlow follows an open approach to service interaction, i.e. the model should explicit express not only what has to be done (through existence and relation formulae), but also what is forbidden (through negation formula).

For a description of the basic existence, relation and negation formulae, see Sections 5, 6 and 7 respectively.

4 Formalizing DecSerFlow: our approach

We give a so called *implicit* formalization of DecSerFlow, in the sense that we would formalize the different DecSerFlow template formulae by using general Integrity Constraints (and with a general Knowledge Base), valid for all DecSerFlow models.

In order to formalize a particular DecSerFlow diagram, we have then just to compile another Knowledge Base which maps the specific diagram structure, and use it together with the general specification.

4.1 DecSerFlow as an abductive logic program

The DecSerFlow formalization $\mathcal{S}_{DSF}$ is defined by means of an abductive logic program[18] as the triple

$$\mathcal{S}_{DSF} \equiv \langle KB_{DSF}, \mathcal{E}_{DSF}, \mathcal{IC}_{DSF} \rangle$$

where:

- KB_{DSF} is the *DecSerFlow Knowledge Base* (which is used to specify knowledge common to all DecSerFlow models);
- $\mathcal{E}_{DSF}$ is the set of *abducible predicates* (i.e. positive and negative expectations);
- $\mathcal{IC}_{DSF}$ is the set of *DecSerFlow Integrity Constraints* (i.e. the set of rules which formalize DecSerFlow template formulae).

4.2 Formalization of activities

As pointed out in Section 2.1, $\mathcal{S}$CIFF completely abstracts from what has to be considered as an observable and relevant event inside the application domain.

To formalize DecSerFlow, we adopt an atomic model for activities, mapping a whatsoever activity execution to an *Event* of the form $performed(A)$, where A is the performed activity. Thus, notation $\mathbf{H}(performed(buy\_item), 18)$ means that the *buy_item* activity has been executed at time 18.

In this preliminary work we do not focus on content data associated to activities. Therefore, for each activity we will consider only:

- its description (with abuse of notation, we will use the terms activity and activity description as synonyms)
- its execution time

abstracting away from all the content data taken in input and provided as output by the activity execution. The introduction of content data associated to activities will be matter of future research, but anyway it is seamlessly addressed by our framework; as we have already pointed out, in fact, by CLP constraints and Prolog predicates $\mathcal{S}$CIFF is able to specify constraints about data and to formalize decisions or, more generally, pieces of knowledge of the interaction model.

Finally, note that, in principle, the $\mathcal{S}$CIFF language is capable to support a non atomic model for activities, mapping their start and completion to events [3] .

[3] Through the integration of abductive event calculus within the framework, is then possible to identify e.g. if an activity is in execution at a given time (see the appendix).

4.3 Fomalization of a DecSerFlow model

The formalization of a specific DecSerFlow model $\mathcal{P}_{spec}$ is defined by extending the Knowledge Base of the DecSerFlow abductive program $\mathcal{S}_{DSF}$:

$$\mathcal{P}_{spec} \equiv \langle KB_{DSF} \cup KB_{spec}, \mathcal{E}_{DSF}, \mathcal{IC}_{DSF} \rangle$$

KB_{spec} contains the specification of the model under study, i.e.:

- its activities
- its existence, relation and negation formulae
- content data associated to activities and conditions related to them (such as temporal deadlines, which we will briefly tackle in Section 8)

The following fragment of a KB_{spec} explain how a small part of the ACME example introduced in [33] could be formalized within our framework:

```
1  existence_formula(credit_card, absence_N(1)).
2  existence_formula(notify_booked, absence_N(1)).
3  relation_formula(credit_card, notify_booked, succession).
4  negation_formula(credit_card, hotel, negation_response).
5  negation_formula(credit_card, airline, negation_response).
```

4.4 An intuition about how DecSerFlow templates are formalized

The specification of the DecSerFlow template formulae by using $\mathcal{S}$CIFF conforms to the following structure:

$$\begin{aligned} & formula(Activity, type) \\ & \wedge body \\ \rightarrow & head. \end{aligned} \tag{7}$$

for the existence formulae and

$$\begin{aligned} & formula(Source, Target, type) \\ & \wedge body \\ \rightarrow & head. \end{aligned} \tag{8}$$

for relation and negation formulae.

It is worth noting that in the first line of all Integrity Constraints of this kind, activities are universally quantified variables. This ensures that each rule will be

replicated for each variable (or couple of variables) subject to the formula addressed by the rule.

For example, let us consider the response relation, which is formalized by using the following Integrity Constraint (see Section 6.2.2).

$$\begin{aligned} & relation\_formula(A, B, response) \\ & \wedge \mathbf{H}(performed(A), T_A) \\ \rightarrow & \mathbf{E}(performed(B), T_B) \\ & \wedge T_B > T_A. \end{aligned} \tag{9}$$

This rule may be read as follows: "*for each A, for each B and for each T_A, if A and B are subject to a response formula and A is executed at time T_A, then there exists a T_B after T_A at which B is expected to be performed*".

Let us now consider the following KB_{spec} fragment:

```
1  relation_formula(ask_for_payment, pay, response).
2  relation_formula(receive_spam, delete_spam, response).
```

During the execution, the $\mathcal{S}$CIFF proof procedure rewrites the response Integrity Constraint by considering the two response relations, obtaining:

$$\begin{aligned} & \mathbf{H}(performed(ask\_for\_payment), T_A) \\ \rightarrow & \mathbf{E}(performed(pay), T_B) \wedge T_B > T_A. \\ & \\ & \mathbf{H}(performed(receive\_spam), T_A) \\ \rightarrow & \mathbf{E}(performed(delete\_spam), T_B) \wedge T_B > T_A. \end{aligned} \tag{10}$$

4.5 Equivalence between Integrity Constrains

When formalizing the different DecSerFlow template formulae, it becomes sometimes important to know if two different specifications (in our case, two different Integrity Constraints sets) are in some sense inter-changeable, i.e., roughly speaking, they formalize the same concept. An emblematic example is the one in [33], where the authors point out the equivalence of some negation formulae.

By using our framework, we can formally prove this concept of equivalence. In particular, we will demonstrate that our proposed DecSerFlow templates formalizations verify the equivalences pointed out in [33].

It is important to note that the equivalence of two Integrity Constraints sets is a particular case of the equivalence w.r.t. conformance of two interaction specifications shown in Definition 2.7.

We briefly motivate this sentence by starting from the following so called "separation" hypotheses, which will be always exhibited by our rules: for each Integrity Constraint of the proposed $\mathcal{S}_{DSF}$, the Integrity Constraint's body cannot contain any abducible (i.e. positive or negative expectation).

From this hypotheses follows that the goal does not affect the Integrity Constraints, and that the Integrity Constraints do not affect each other. The only way to making them "interact" is, in fact, the goal or a rule's head leads to the hypotheses of an abducible and a matching abducible is contained in the body of another rule [4] .

Let us suppose that $\mathcal{S}^1$ and $\mathcal{S}^2$ represent two (different) proposal for the formalization of some DecSerFlow template formulae, different by means of their Integrity Constraints. The two specifications share both the set of abducible predicate $\mathcal{E}_{DSF} = \{\mathbf{E}, \mathbf{EN}\}$ and the knowledge base. If the specification satisfy the "separation" hypotheses, i.e. there is no rule's body containing expectations (and hence each rule triggers independently), then the proof of $\langle KB, \mathcal{E}_{DSF}, \mathcal{IC}_{S1}\rangle \equiv \mathcal{S}^1 \overset{\mathcal{C}}{\equiv} \mathcal{S}^2 \equiv \langle KB, \mathcal{E}_{DSF}, \mathcal{IC}_{S2}\rangle$ reduces to prove the equivalence of the two Integrity Constraints sets. Therefore, in the following we will say, with abuse of notation, that two Integrity Constraints sets are equivalent w.r.t. conformance ($\mathcal{IC}_{S1} \overset{\mathcal{C}}{\equiv} \mathcal{IC}_{S2}$) to denote equivalence of these kinds of specification.

Furthermore, under the "separation" hypotheses the Integrity Constraints enjoy compositionality [5] w.r.t. $\overset{\mathcal{C}}{\equiv}$, in the sense that

$$\mathcal{IC}_{S1} \overset{\mathcal{C}}{\equiv} \mathcal{IC}_{S2} \rightarrow \mathcal{IC}_{S1} \cup \mathcal{IC}_{S3} \overset{\mathcal{C}}{\equiv} \mathcal{IC}_{S2} \cup \mathcal{IC}_{S3}$$

As a consequence, if we want to prove whether two different rules could be interchangeably used within the DecSerFlow formalization $\mathcal{S}_{DSF}$, it is sufficient just to consider them without taking into account the other Integrity Constraints.

5 Existence Formulae

In this section we will briefly describe DecSerFlow existence template formulae and show how they may be translated into the $\mathcal{S}$CIFF language.

5.1 Summary of the DecSerFlow Existence Formulae

DecSerFlow existence formulae are unary constraints over activities cardinality; a brief description of them is given in table 1.

[4] For example, the rule $\mathbf{H}(a_1, T_1) \rightarrow \mathbf{E}(a_2, T_2) \wedge T_2 > T_1$ affects the rule $\mathbf{E}(a_2, T_2) \rightarrow \mathbf{EN}(a_3, T_3) \wedge T_3 > T_2$.

[5] Roughly speaking, compositionality states that the semantics of a whole is given by the union of the semantics of its parts.

Table 1. *Existence Formulae in DecSerFlow*

	name	description
	absence	A should *not* be performed.
	existence_N	A should be performed *at least* N times.
	absence_N	A should be performed *at most* N times.
	exactly_N	A should be performed *exactly* N times.

5.2 Formalization of the Existence Formulae

Table 2 shows how the different DecSerFlow existence formulae could be formalized by using the $\mathcal{S}$CIFF language.

Table 2. *Schema of the existence templates formalization*

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

We assume $T_0 = 0$

As we have already pointed out, each policy of the model is mapped ino the KB_{spec} (by using a particular fact). The KB_{spec} defines the structure of the model under study, while KB_{DSF} and $\mathcal{IC}_{DSF}$ formalize the different template formulae.

Table 3 shows how the existence formulae of a DecSerFlow model could be represented in its corresponding KB_{spec}.

5.2.1 Absence

The absence of an activity is simply modeled by imposing that its execution is expected not to happen at all:

$$\begin{aligned} &existence\_formula(A, absence) \\ \rightarrow &\mathbf{EN}(performed(A), T_A). \end{aligned} \tag{11}$$

Thank to the universal quantification of the variable A, the Integrity Constraint

Table 3. *Predicates to be used in the KB_{spec} for specifying the existence formulae of a DecSerFlow model*

	$existence\_formula(A, absence)$ or $existence\_formula(A, exactly\_N(0))$
	$existence\_formula(A, existence\_N(N))$
	$existence\_formula(A, absence\_N(N))$
	$existence\_formula(A, exactly(N))$ (with $N > 0$)

expresses that *for each* A, if A is subject to an absence formula, then A is expected not to happen.

5.2.2 Existence_N

The presence of the execution of A at least N times is imposed by generating N different expectations about it. The difference between expectations is expressed as a difference between their execution times.

In order to generate N different expectations about a certain event, with N not known a priori, KB_{DSF} provides the predicate $expect_N(X, N, T_{first}, T_{last})$, where X is the event to be expected N times and T_{first} (T_{last} respectively) is an optional parameter representing the minimum (the maximum resp.) time at which one of the expectations is fulfilled.

The Integrity Constraint is then simply:

$$\begin{aligned} &existence\_formula(A, existence\_N(N)) \\ \rightarrow &expect\_N(performed(A), N). \end{aligned} \tag{12}$$

5.2.3 Absence_N

The presence of at most N executions of A is modeled as an Integrity Constraint whose body contains the happening of N different $performed(A)$ and whose head imposes that the N+1-th execution of A is forbidden (i.e. expected not to happen).

In a similar way than the $expect\_N$ predicate, to express the happening of N different executions of A, KB_{DSF} provides the predicate $happened_N(X, N, T_{first}, T_{last})$, where X is the event to be considered and T_{first} (T_{last} respectively) the first happening time (the last resp.) of the series.

To forbid the N+1-th execution of A we have to consider the time at which the last of the N considered events happens. This time is used to impose that further

$performed(A)$ cannot happen after it:

$$\begin{aligned} & existence\_formula(A, absence\_N(N)) \\ & \wedge happened\_N(performed(A), N, T_{first}, T_{last}) \\ \rightarrow & \mathbf{EN}(performed(A), T) \\ & \wedge T > T_{last}. \end{aligned} \tag{13}$$

5.2.4 Exactly_N

In order to express that A is expected to be executed exactly N times, it is possible to combine together the absence_N and the existence_N formulae about A. The existence_N formula is satisfied when N executions of A happened. But these N happened events trigger the absence_N Integrity Constraint, generating the negative expectation about further executions of A.

This mechanism is realized within the KB_{DSF} by classifying the exactly_N formula both as an absence_N and an existence_N ones:

```
1  existence_formula(Activity, existence_N(N)) :-
2      existence_formula(Activity, exactly_N(N)),
3      N>0.
4
5  existence_formula(Activity, absence_N(N)) :-
6      existence_formula(Activity, exactly_N(N)),
7      N>0.
```

6 Relation Formulae

In this section we will briefly describe the DecSerFlow relation template formulae and show how they may be translated into the $\mathcal{S}$CIFF language.

6.1 Summary of the DecSerFlow Relation Formulae

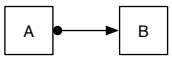
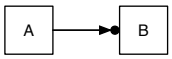
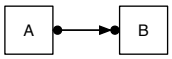
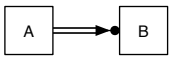
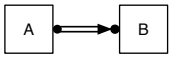
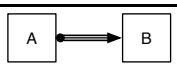
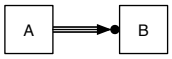
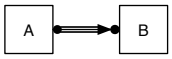
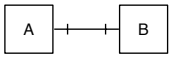
For the sake of simplicity, for the moment we focus only on binary relation formulae, which are positive relationships between two activities (see table 4 for their brief description). In section 8 we will discuss how the formalization of these formulae could be easily extended in order to deal with n-to-n relation formulae and to consider deadlines.

6.2 Formalization of the Relation Formulae

Each relation formula of a DecSerFlow model is formalized inside the KB_{spec} as a fact of the type

$$relation\_formula(A, B, Type)$$

Table 4. *Relation Formulae in DecSerFlow*

	name	description
	responded existence	if A is performed, then B has to be performed either before or after A
	coexistence	if one of A or B are performed, the other one has to be executed, too
	response	every time A is performed, B should be performed *after* it
	precedence	if B is performed, A should have been performed *before* it
	succession	every execution of A should be *followed* by the execution of B and each B should be *preceded* by A
	alternate response	A should be *followed* by B and, *between* the execution of two activities A, there should be at least one B
	alternate precedence	B should be *preceded* by A and, *between* the execution of two activities B, there should be at least one A
	alternate succession	It enforces both the alternate response and the alternate precedence formulae, imposing that the *succession* formula should hold between A and B and that the execution of two activities A (B respectively) has to be interleaved by the execution of at least one B (A resp.)
	chain response	The *next* activity *after* the execution of A should be B
	chain precedence	The *first preceding* activity *before* the execution of B should be A
	chain succession	Activities A and B should be always performed *next to each other*
	mutual substitution	*At least one* of activities A and B should be performed.

For example, $relation\_formula(buy\_item, send\_receipt, chain\_succession)$ expresses that a chain succession relation holds between the *buy item* and the *send receipt* activities.

A first thing to note is that, as in the case of the exactly_N existence formula, some relation formulae are expressed by the combination of two other ones. In particular, the coexistence formula is equivalent to two responded existence formulae, inverse to each other, and each of the three "succession" formulae is expressed in terms of its corresponding "response" and (inverse) "precedence" ones (e.g. the chain succession formula between A and B is the combination of the chain response

between A and B and the chain precedence between B and A).

These equivalences are defined within the KB_{DSF}:

```
1  relation_formula(A, B, responded_existence) :-
2      relation_formula(A, B, coexistence).
3
4  relation_formula(B, A, responded_existence) :-
5      relation_formula(A, B, coexistence).
6
7  relation_formula(A, B, response) :-
8      relation_formula(A, B, succession).
9
10 relation_formula(B, A, precedence) :-
11     relation_formula(A, B, succession).
12
13 relation_formula(A, B, alternate_response) :-
14     relation_formula(A, B, alternate_succession).
15
16 relation_formula(B, A, alternate_precedence) :-
17     relation_formula(A, B, alternate_succession).
18
19 relation_formula(A, B, chain_response) :-
20     relation_formula(A, B, chain_succession).
21
22 relation_formula(B, A, chain_precedence) :-
23     relation_formula(A, B, chain_succession).
```

Furthermore, we notice that each "response" formula has the same formalization as the corresponding "precedence" one, except for the fact that the former imposes forward temporal constraints (specifying what is expected *after* an event occurrence), whereas the latter imposes backward temporal constraints (specifying what is expected to have happened *before* an event occurrence).

6.2.1 Responded Existence

The responded existence formula states that when the source activity happens, then the destination activity is expected, either before or after the source one (i.e. no temporal constraint between the two activities execution times is imposed):

$$\begin{aligned} & relation\_formula(A, B, responded\_presence) \\ & \wedge \mathbf{H}(performed(A), T_A) \\ \rightarrow & \mathbf{E}(performed(B), T_B). \end{aligned} \tag{14}$$

6.2.2 Response

The response relation extends the responded existence one by imposing that when the source activity happens, the destination has to happen after it. Therefore, it is formalized by using a temporal constraint which states that the expected execution time of the destination activity should be greater than the time at which the source happens:

$$\begin{aligned} & relation\_formula(A, B, response) \\ & \wedge \mathbf{H}(performed(A), T_A) \\ \rightarrow & \mathbf{E}(performed(B), T_B) \\ & \wedge T_B > T_A. \end{aligned} \tag{15}$$

6.2.3 Precedence

As we have already pointed out, each of the precedence formula imposes the same behaviour of the corresponding response one, but the temporal constraints are inverted. Thus, the precedence relation has the same formalization as the response one, but the destination time must be lower than the source time:

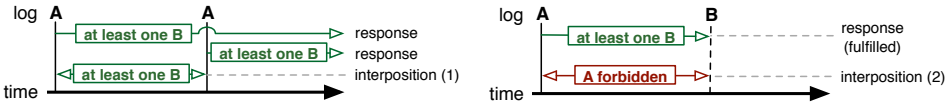
$$\begin{aligned} & relation\_formula(A, B, precedence) \\ & \wedge \mathbf{H}(performed(A), T_A) \\ \rightarrow & \mathbf{E}(performed(B), T_B) \\ & \wedge T_B < T_A. \end{aligned} \tag{16}$$

6.2.4 Alternate Response

Following the description shown in table 4, we could formalize the alternate response between two activities by imposing both:

- a response relation between them
- a second Integrity Constraint which preserves the interposition:

$$\begin{aligned} & relation\_formula(A, B, alternate\_response) \\ & \wedge \mathbf{H}(performed(A), T_A) \\ & \wedge \mathbf{H}(performed(A), T_{A2}) \\ & \wedge T_{A2} > T_A \\ \rightarrow & \mathbf{E}(performed(B), T_B) \\ & \wedge T_B > T_A \wedge T_B < T_{A2}. \end{aligned} \tag{17}$$



(a) Interposition by expecting b between two a

(b) Interposition by forbidding another a between an a and the (first) b after it

Figure 2. *Two different perspectives about the representation of the alternate response relation between two activities a and b.*

Although correct, this formulation has a main problem: even if the relation is a forward one (i.e. it constrains what has to be done after the happening of the relation source), in (17) the expectation about the execution of B between two A is backwards triggered only when the second A happens.

To face this issue, we give a different formalization of the alternate response formula, using only one integrity contraint and, at the same time, overcoming the above limit. The intuition is that the interposition of one activity (say, B) between two other activities (say, A) could be expressed in two different ways: by directly saying that between two A at least one B should be executed (Integrity Constraint (17)), or by saying that the execution of a second A is forbidden between the first A and the (first) execution of B after it (see also figure 2).

By adopting the second perspective, the formalization of the alternate response relation is:

$$\begin{aligned} & relation\_formula(A, B, alternate\_response) \\ & \wedge \mathbf{H}(performed(A), T_A) \\ \rightarrow & \mathbf{E}(performed(B), T_B) \\ & \wedge T_B > T_A \\ & \wedge \mathbf{EN}(performed(A), T_{A2}) \\ & \wedge T_{A2} > T_A \wedge T_{A2} < T_B. \end{aligned} \tag{18}$$

Remark 6.1 *The two formulations of the alternate response relation are equivalent w.r.t. conformance.*

Proof. Let us consider two activities a and b, with

$$relation\_formula(a, b, alternate\_response)$$

We have to prove that, for a and b,

$$(15) \wedge (17) \overset{\mathcal{C}}{\equiv} (18)$$

$(15) \wedge (17) \stackrel{\mathcal{C}}{\leftarrow} (18)$
In order to disconfirm the implication, we have to find a counter-example, i.e. a log which satisfies (18) but violates (15) or (17). The first thing to note is that, trivially, $(18) \stackrel{\mathcal{C}}{\rightarrow} (15)$. In fact, if (18) holds, we have two main possibilities:

- The log does not contain any execution of activity a; in this case, both the Integrity Constraints are trivially satisfied (their bodies are false).
- The log contains one or more execution of activity a. In order to satisfy the positive expectation about the execution of b after a in rule (18), for each a the log must contain also a corresponding b after it. This event fullfills the positive expectation of (18), but fullfills also (15).

As a consequence, the proof reduces to find a log which satisfies (18) and violates (17).

To violate (17), the log must contain at least two executions of activity a (say, $\mathbf{H}(performed(a), T_1)$ and $\mathbf{H}(performed(a), T_2)$, having $T_2 > T_1$), with no execution of b between them. For both executions of a, the log must also satisfy the "response" part of (18), i.e. it must contain at least one execution of b after T_1 and one after T_2. Therefore, two different cases may arise:

- the log contains at least one $performed(b)$ within the interval (T_1, T_2) and at least one b after T_2; however, this is impossible, since the execution of b between the two a is forbidden by hypotheses.
- the log does not contain any $performed(b)$ within the interval (T_1, T_2) and contains at least one $performed(b)$ after T_2; in this case, the log satisfies the following property: between all the eventual executions of b and the happening of a (at time T_1) there exists a second execution of a (the one at time T_2). However, this violates the negative expectation of (18), leading to contradiction.

$(15) \wedge (17) \stackrel{\mathcal{C}}{\rightarrow} (18)$
Let us suppose, by absurdum, that there exists a log which satisfies (15) and (17) but violates (18). To violate (18), the log must satisfy its body (i.e. must contain at least one execution of activity a, say, at time T_1) and violate its head. The head of (18) is violated when the expectation about the execution of b after a or the negative expectation about the execution of another a between a and b are violated.

The first possibility is clearly contradictory, because we are trying to violate the same expectation to be fulfilled in (15), which constrains the log to contain also at least one execution of activity b (say, at T_2) after T_1.

To verify the second possibility, the execution trace has to contain also a second execution of a, say, at a time T_3 between T_1 and T_2. However, the presence of this a

contradicts the hypotheses by violating (17), beacuse between T_1 and T_3 there does not exist any execution of b. $\square$

6.2.5 Alternate Precedence

The alternate precedence formula is treated in a similar way as the alternate response. It could be formalized by imposing the precedence relation and the "interposition" rule (17), or by adopting the second approach, i.e. by using a variant of (18):

$$\begin{aligned} & relation\_formula(A, B, alternate\_precedence) \\ & \wedge \mathbf{H}(performed(A), T_A) \\ \rightarrow & \mathbf{E}(performed(B), T_B) \\ & \wedge T_B < T_A \\ & \wedge \mathbf{EN}(performed(A), T_{A2}) \\ & \wedge T_{A2} \in (T_B, T_A). \end{aligned} \tag{19}$$

The equivalence between these two formulations can be proven in a very similar way than the case of the alternate response.

6.2.6 Chain Response

An activity B is chain response of an activity A if the former belongs to the next state of the latter.

In the $\mathcal{S}$CIFF language, the notion of next state is not a first-class object (like e.g. in linear temporal logic, by using the operator $\bigcirc$). However, $\mathcal{S}$CIFF has an explicit notion of time (or time stamp), which may be used in some sense to identify execution states.

Let us consider two whatsoever activities, say, A and B. Basically, we consider an happening of B at a time T_2 to belong to the next state w.r.t. an happening of A at a time T_1 iff between these two times nothing happens (i.e. the interaction state determined by the execution of A at T_1 is not changed until T_2).

Therefore, to express the chain response relation between A and B we say that B is expected to happen after A and that, between the two activities, all events are

forbidden [6] .

$$\begin{aligned} & relation\_formula(A, B, chain\_response) \\ & \wedge \mathbf{H}(performed(A), T_A) \\ \rightarrow & \mathbf{E}(performed(B), T_B) \\ & \wedge T_B > T_A \\ & \wedge \mathbf{EN}(performed(X), T) \\ & \wedge T > T_A \wedge T < T_B. \end{aligned} \tag{20}$$

6.2.7 Chain Precedence

An activity B is chain precedence of an activity A if it belongs to the previous state of the latter (i.e. B is expected to happen before A and nothing can happen between them):

$$\begin{aligned} & relation\_formula(A, B, chain\_precedence) \\ & \wedge \mathbf{H}(performed(A), T_A) \\ \rightarrow & \mathbf{E}(performed(B), T_B) \\ & \wedge T_B < T_A \\ & \wedge \mathbf{EN}(performed(X), T) \\ & \wedge T > T_B \wedge T < T_A. \end{aligned} \tag{21}$$

6.2.8 Mutual Substitution

Considering two activities, the mutual substitution formula specifies that at least one activity among them has to be performed. To formalize this condition, we make use of an Integrity Constraint with a disjunction of expectation in the head:

$$\begin{aligned} & relation\_formula(A, B, mutual\_substitution) \\ \rightarrow & \mathbf{E}(performed(A), T_A) \\ \vee & \mathbf{E}(performed(B), T_B). \end{aligned} \tag{22}$$

7 Negation formulae

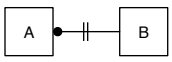
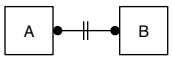
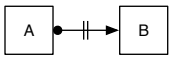
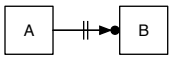
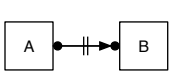
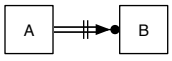
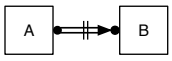
In this section we will briefly describe the DecSerFlow negation template formulae and show how they may be translated into the $\mathcal{S}$CIFF language.

[6] Note that variable X in (20) is universally quantified.

7.1 Summary of the DecSerFlow Negation Formulae

Negation formulae express the complementary behaviour w.r.t. relation ones. As pointed out in [33], however, an important difference is that some negation formulae are different ways to express the same policy, i.e. some negation formula are equivalent w.r.t. conformance. Table 5 introduces the different types of negation formulae, while figure 7.1 explicitly expresses the equivalences.

Table 5. *Negation Formulae in DecSerFlow*

	name	**description**
	responded absence	if A is performed, then B cannot be performed neither before or after A
	not coexistence	if one of A or B are performed, the other one cannot be executed
	negation response	when A is performed, B cannot be performed *after* it
	negation precedence	if B is performed, A should not have been performed *before* it
	negation succession	the execution of A should not be *followed* by the execution of B and each B should not be *preceded* by A
	negation alternate response	B cannot be performed *between* the execution of two activities A
	negation alternate precedence	A cannot be performed *between* the execution of two activities B
	negation alternate succession	It enforces both the negation alternate response and the negation alternate precedence formulae, imposing that two A cannot be interleaved by the execution of B and viceversa.
	negation chain response	B cannot be performed directly after A (i.e. A's next activity should be different than B)
	negation chain precedence	A cannot be the *first preceding* activity *before* B
	negation chain succession	Activities A and B cannot never be performed *next to each other*

In the following, we will give the formalization of negation formulae and prove the equivalence shown in figure 7.1. Note that, even if equivalent w.r.t. conformance, some formulae are preferrable to others, because they explicitly abduces more informations about the log structure, i.e. they give more feedbeck about the interaction to the user.

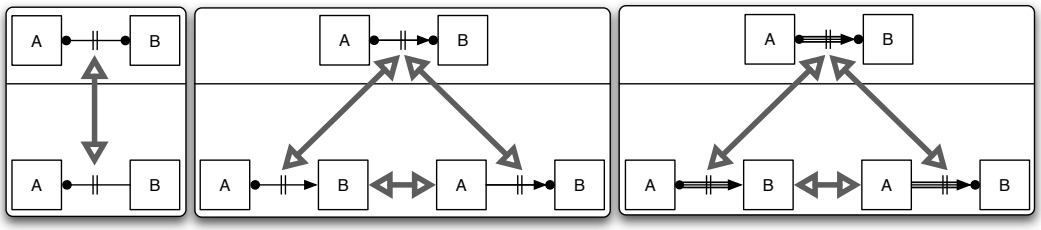


Figure 3. *Equivalences w.r.t. conformance between negation formulae.*

7.2 Formalization of the Negation Formulae

Similarly to the case of relation formulae, each negation formula of a DecSerFlow model is formalized inside the KB_{spec} as a fact of the type

$$negation\_formula(A, B, Type)$$

Furthermore, also some negation formulae are a composition of other ones:

```
1   negation_formula(A, B, responded_absence):-
2       negation_formula(A, B, not_coexistence).
3
4   negation_formula(B, A, responded_absence):-
5       negation_formula(A, B, not_coexistence).
6
7   negation_formula(A, B, negation_response):-
8       negation_formula(A, B, negation_succession).
9
10  negation_formula(B, A, negation_precedence):-
11      negation_formula(A, B, negation_succession).
12
13  negation_formula(A, B, negation_alternate_response):-
14      negation_formula(A, B, negation_alternate_precedence).
15
16  negation_formula(A, B, negation_alternate_response):-
17      negation_formula(A, B, negation_alternate_succession).
18
19  negation_formula(B, A, negation_alternate_precedence):-
20      negation_formula(A, B, negation_alternate_succession).
21
22  negation_formula(A, B, negation_chain_response):-
23      negation_formula(A, B, negation_chain_succession).
```

```
24
25   negation_formula(B, A, negation_chain_precedence):-
26       negation_formula(A, B, negation_chain_succession).
```

The main difference w.r.t. relation formulae is that the negation alternate response and the negation alternate precedence express exactly the same behaviour (namely the absence of an activity between two executions of another one).

7.2.1 Responded Absence

The responded absence formula forbids the execution of a given activity when another one is performed:

$$\begin{aligned} & negation\_formula(A, B, responded\_absence) \\ & \wedge \mathbf{H}(performed(A), T_A) \\ \rightarrow & \mathbf{EN}(performed(B), T_B). \end{aligned} \tag{23}$$

7.2.2 Equivalence w.r.t conformance between the Responded Absence and the Not Coexistence formulae

Given two activities a and b, the proof of equivalence w.r.t. conformance between the responded absence and the not coexistence formulae reduces to prove that

$$\mathbf{H}(performed(a), T_a) \rightarrow \mathbf{EN}(performed(b), T_b) \tag{24}$$

$$\overset{\mathcal{C}}{\equiv}$$

$$\mathbf{H}(performed(b), T_b) \rightarrow \mathbf{EN}(performed(a), T_a) \tag{25}$$

Proof. Since the two formulations are completely symmetrical, we skecth only the proof of one side of the equivalence.
$(24) \xrightarrow{\mathcal{C}} (25)$
Let us consider, by absurdum, that there exists a log which satisfies (24) and violates (25). In order to violate (25), the log must contain the execution of both activities b and a. The presence of a $performed(a)$ in the log triggers (24), generating a negative expectation about the execution of activity b, which is however part of the execution trace, thus leading to contradiction. □

7.2.3 Negation Response

The negation response extends the responded absence one by saying that the destination is forbidden after the execution of the source:

$$
\begin{aligned}
&negation\_formula(A, B, negation\_response) \\
&\quad \wedge \mathbf{H}(performed(A), T_A) \\
\rightarrow &\mathbf{EN}(performed(B), T_B) \\
&\quad \wedge T_B > T_A.
\end{aligned}
\tag{26}
$$

7.2.4 Negation Precedence

The negation precedence inverts the temporal constraint of the negation response by specifying that the destination is forbidden before the execution of the source:

$$
\begin{aligned}
&negation\_formula(A, B, negation\_response) \\
&\quad \wedge \mathbf{H}(performed(A), T_A) \\
\rightarrow &\mathbf{EN}(performed(B), T_B) \\
&\quad \wedge T_B < T_A.
\end{aligned}
\tag{27}
$$

7.2.5 Equivalence w.r.t. conformance between the Negation Response and the inverse Negation Precedence formulae

Given two activities a and b, we have to prove that specifying that b is a negation response of a is equivalent w.r.t. conformance to specifying that a is a negation precedence of b, i.e.:

$$
\mathbf{H}(performed(a), T_a) \rightarrow \mathbf{EN}(performed(b), T_b) \wedge T_b > T_a \tag{28}
$$

$$
\stackrel{\mathcal{C}}{\equiv}
$$

$$
\mathbf{H}(performed(b), T_b) \rightarrow \mathbf{EN}(performed(a), T_a) \wedge T_a < T_b \tag{29}
$$

Proof.
(28)$\stackrel{\mathcal{C}}{\rightarrow}$(29)
To violate (29), the log must contain the happening of $performed(b)$, say, at a time T_1, and the execution of activity a at a previous time w.r.t. T_1 (i.e. at a time $T_2 < T_1$). The happening of a $performed(a)$ at time T_2 triggers the negation response, and therefore a negative expectation about a consequent execution of b is abduced. This expectation is violated by the happening of $performed(b)$ at time T_1, and thus it is impossible to verify (28) by violating (29) at the same time.

(28)$\stackrel{\mathcal{C}}{\leftarrow}$(29)
Trivially provable by applying the same procedure adopted above. □

7.2.6 Negation Alternate Response

Following the natural language description, an activity B is negation alternate response of another activity A if between two different execution of A activity B is not performed.

Therefore, a possible formalization of the negation alternate response could be, in a straightforward way:

$$\begin{aligned} & negation\_formula(A, B, negation\_alternate\_response) \\ & \wedge \mathbf{H}(performed(A), T_A) \\ & \wedge \mathbf{H}(performed(A), T_{A2}) \\ & \wedge T_{A2} > T_A \\ \rightarrow & \mathbf{EN}(performed(B), T_B) \\ & \wedge T_B \in (T_A, T_{A2}). \end{aligned} \tag{30}$$

This rule formalizes the negative version of the interposition Integrity Constraint (rule (17)). As in the case of alternate response, however, this formalization is not fully "forward", because it triggers only when a second execution of activity A happens, generating a negative expectation backwards in time (namely before T_{A2}).

Hence, we would express the negative interposition formula by adopting a different approach. Let us suppose that, in the log, an activity A has been performed. If no B is executed after it, the negative interposition is preserved. If instead the log contains also the execution of B after A, the interposition is fulfilled only if no more A are performed after B.

Thus, a completely forward formulation of the negation alternate response is the one which takes into account the case of an activity B performed after the execution of A, forbidding the execution of further A:

$$\begin{aligned} & negation\_formula(A, B, negation\_alternate\_response) \\ & \wedge \mathbf{H}(performed(A), T_A) \\ & \wedge \mathbf{H}(performed(B), T_B) \\ & T_B > T_A \\ \rightarrow & \mathbf{EN}(performed(A), T_{A2}) \\ & \wedge T_{A2} > T_B. \end{aligned} \tag{31}$$

Remark 7.1 *The two formulations of the negation alternate response are equivalent w.r.t. conformance.*

Proof. We have to prove that, considering an activity b negation alternate response of another activity a, (30)$\overset{\mathcal{C}}{\equiv}$(31) for them.

(30) $\stackrel{\mathcal{C}}{\rightarrow}$(31)
As usual, we try to find a log which violates (31) and satisfies (30). (31) is violated when the log contains the sequence $performed(a)$, $performed(b)$, $performed(a)$ (at, say, times T_1, T_2 and T_3, with $T_3 > T_2 > T_1$). The execution of two instances of activity a triggers rule (30) therefore imposing that no b is executable inside the time interval (T_1, T_3). However, the execution of b at time T_2 contradicts this expectation.

(30) $\stackrel{\mathcal{C}}{\leftarrow}$(31)
Trivially provable by following the same procedure: to violate (30) the log has to contain the happening of the sequence $performed(a)$, $performed(b)$, $performed(a)$, which is evaluated as non conformant by (31). □

7.2.7 Negation Chain Response

To formalize the negation chain response formula between two activities A and B we have to specify that the first following activity after A cannot be B. Basically, we make use of the chain response formalization, by imposing that the next activity w.r.t. A must be different than B. Moreover, we must taking into account also the possibility of A being the last activity of the execution trace. The two alternative cases are modeled by using a disjunction in the rule's head:

$$\begin{aligned}
&negation_formula(A, B, negation_chain_response)\\
&\quad \wedge \mathbf{H}(performed(A), T_A)\\
\rightarrow &\mathbf{E}(performed(X), T_X)\\
&\quad \wedge T_X > T_A\\
&\quad \wedge \mathbf{EN}(performed(Y), T_Y)\\
&\quad \wedge T_Y \in (T_A, T_X)\\
&\quad \wedge X \neq B\\
&\vee \mathbf{EN}(performed(X), T_X)\\
&\quad \wedge T_X > T_A.
\end{aligned} \tag{32}$$

7.2.8 Negation Chain Precedence

Similarly to the negation chain response, the negation chain precedence between two activities A and B is formalized by specifying either that A has a first preceding activity which cannot be B or that A is the first activity of the log (i.e. it does not have a preceding activity at all).

$$
\begin{aligned}
& negation\_formula(A, B, negation\_chain\_precedence) \\
& \wedge \mathbf{H}(performed(A), T_A) \\
\rightarrow & \mathbf{E}(performed(X), T_X) \\
& \wedge T_X < T_A \\
& \wedge \mathbf{EN}(performed(Y), T_Y) \\
& \wedge T_Y \in (T_X, T_A) \\
& \wedge X \neq B \\
\vee & \mathbf{EN}(performed(X), T_X) \\
& \wedge T_X < T_A.
\end{aligned} \tag{33}
$$

7.2.9 Equivalence w.r.t. conformance between the Negation Chain Response and the inverse Negation Chain Precedence formulae

Given two activities a and b, we have to prove that specifying that b is a negation chain response of a is equivalent w.r.t. conformance to specifying that a is a negation chain precedence of b, namely

$$
\begin{aligned}
\mathbf{H}(performed(a), T_a) \quad \rightarrow \quad & \mathbf{E}(performed(X), T_X) \wedge T_X > T_a \qquad (34) \\
& \wedge \mathbf{EN}(performed(Y), T_Y) \wedge T_Y \in (T_a, T_Y) \wedge X \neq b \\
\vee \quad & \mathbf{EN}(X, T_X) \wedge T_X > T_a \\
\overset{\mathcal{C}}{\equiv} & \\
\mathbf{H}(performed(b), T_b) \quad \rightarrow \quad & \mathbf{E}(performed(X), T_X) \wedge T_X < T_b \qquad (35) \\
& \wedge \mathbf{EN}(performed(Y), T_Y) \wedge T_Y \in (T_X, T_b) \wedge X \neq a \\
\vee \quad & \mathbf{EN}(X, T_X) \wedge T_X < T_b
\end{aligned}
$$

Proof. As in the previous cases, we prove the two sides of the equivalence by contradiction.

(35) $\overset{\mathcal{C}}{\rightarrow}$(36)
Rule (36) is violated by an execution traces if:

- It contains the execution of activity b, but not as the first one.
- Being not the first activity, b has, in fact, a first preceding activity, and this activity is just a.

Hence, the considered log contains the execution of activity a followed by b as its next activity, and this means that the negation chain response rule (35) will evaluate the log as non conformant.

(35) $\overset{\mathcal{C}}{\leftarrow}$(36)
Similarly to the just considered case, (35) is violated by an execution traces if it contains the execution of an activity a which cannot be the last one, but is immediately followed by the execution of b. However, this behaviour violates rule (36), which specifies that a should not be the immediately preceding activity of b. □

8 Proposed extensions of DecSerFlow: a first step

In this section we propose some extensions of the basic DecSerFlow formulae (i.e. existence formulae and binary relation and negation formulae), namely:

- relation and negation formulae considering compositions (disjunctions and conjunctions) of activities, following what has been proposed in [33];
- temporal constraints as a further formula parameter.

8.1 Temporal constrained formulae

Since the $\mathcal{S}$CIFF framework treats explicitly time variables, supporting the specification of temporal constraints as CLP constraints, we could extend DecSerFlow policies with this feature.

We have identified some temporal constraints templates which could be useful for a DecSerFlow user (see figure 4):

- $after(N)$ template (with $N > 0$), to specify that an activity has to be executed after at least N time units w.r.t. another one;
- $before(N)$ template (with $N < 0$), to specify that an activity has to be executed at least before N time units w.r.t. another one;
- $between(N_1, N_2)$ template, to specify that an activity has to be executed between N_1 and N_2 time units after (if $N_1 \geq 0$ and $N_2 > 0$) or before (if $N_1 < 0$ and $N_2 \leq 0$) another one;
- $equals(N)$ template, to impose that an activity is executed exactly N time units after (if $N \geq 0$) or before (if $N < 0$) another one.

Figure 4 highlights also that the different DecSerFlow formulae are implicitly associated to some of this temporal constraints templates. Basically, we could factorize the different relation formulae (and, in a similar way, the different negation

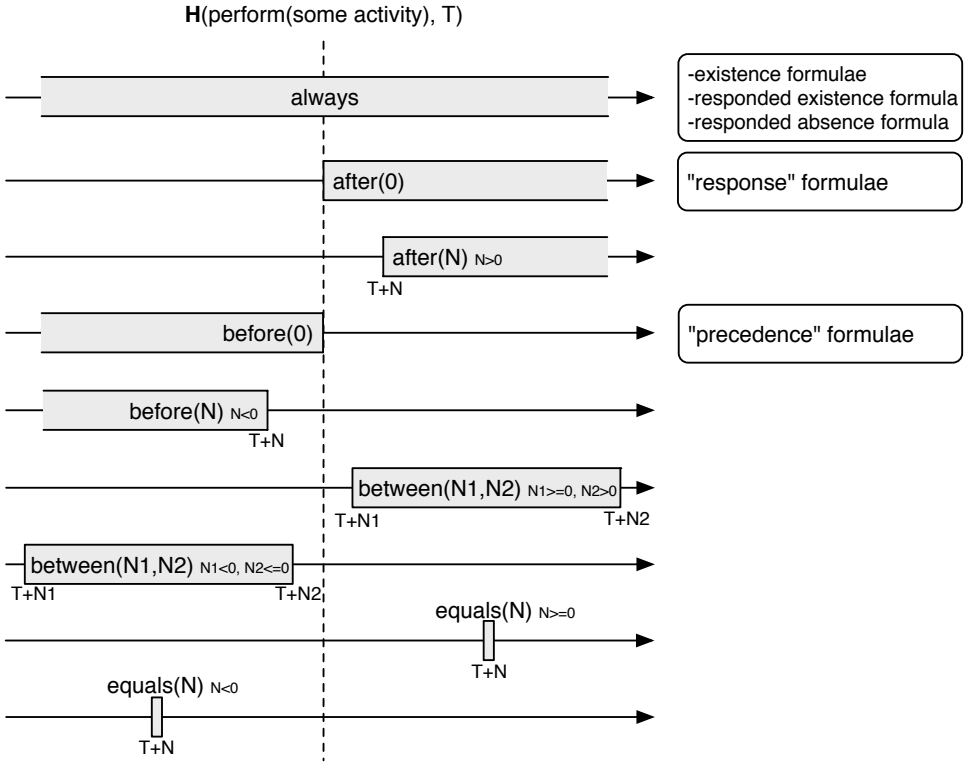


Figure 4. *Temporal constraints templates for atomic activities*

formulae) in three families, and then express all the relations within a family by using the same rule template but changing the temporal constraint used in it. See table 6 for a sketch of this factorization.

Table 6. *Factorization of the different relation formulae*

	family	**temporal constraint**
responded existence	simple	$always$
response	simple	$after(0)$
precedence	simple	$before(0)$
alternate response	alternate	$after(0)$
alternate precedence	alternate	$before(0)$
alternate chain response	chain	$after(0)$
alternate chain precedence	chain	$before(0)$

By using this extended formalization, we could for example specify that after a user price request, then the seller should answer within 5 time units by inserting in

the KB_{spec} the following fact:

$$relation\_formula(price\_request, price\_answer, response, before(5))$$

will be rewritten by $\mathcal{S}$CIFF as

$$relation(price\_request, price\_answer, simple, between(0, 5))$$

When the price request happens, say, at time T_1, then $\mathcal{S}$CIFF abduces an expectation about the happening of the price answer at a time T_2 such that $T_2 > T_1 + 0$ and $T_2 < T_1 + 5$.

Temporal constraints could be used also in conjunction with existence formulae, for example to specify that the *check stock* activity should be performed at least three times within 500 time units w.r.t. the beginning of the execution (by using a fact of the type $existence\_formula(check\_stock, existence\_N(3), before(500))$, where 500 is treated, in this case, as an absolute time).

8.2 Formulae with composition of activities

As described in [33], DecSerFlow relationships could be easily extended to deal with more activities. The authors propose, for example, to support relationships with multiple outgoing arcs. The intended meaning is, in this case, disjunction of activities as target of the relationship.

Furthermore, it could be interesting to specify also conjunction of activities; this feature could make possible to represent that the seller must send a receipt to the client after having received both the payment and the warehouse acceptance.

Table 7 summarizes how disjunction and conjunction of activities could be used to specify complex relationships and what interaction patterns [7] they realize. These patterns could be implemented by applying minor changes to the Integrity Constraints proposed for binary formulae [8] .

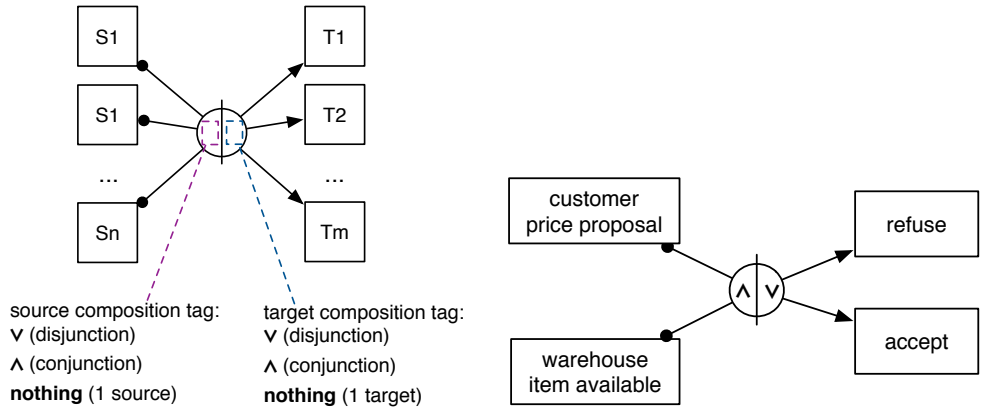
Table 7. *Meaning of relationships with composite activities*

	Symbol	**Relationship Source**	**Relationship Target**
conjunction	$\wedge$	Synchronization	Parallel Split
disjunction	$\vee$	Simple Merge	Deferred Choice

Note that DecSerFlow indirectly already supports disjunction in the sources and conjunction in the targets through relationships replication. For example, designing a DecSerFlow model which contains a response relation between activities a and b

[7] We use the term "interaction pattern" to distinguish it from the workflow patterns [34]. In fact, in this context we consider pattern between activities and not between flows.

[8] We do not report here how these extensions are actually being implemented within $\mathcal{S}_{DSF}$.



(a) Schema of an extended relationship (b) An example of extended response

Figure 5. *A preliminary proposal for extended relationships with composite activities*

and a response relation between a and c is the same as specify a unique formula whose source is a and whose target is the conjunction between b and c. However, having these features as first-class objects could be useful to the aim of specifying DecSerFlow models in a simple and concise manner and to factorize similar policies.

In order to support the two different types of source and target compositions of activities, however, we need to extend the DecSerFlow notation of relationships to explicitly take into account them. Figure 5(a) shows a preliminary composition representation proposal. When a relationships has two or more sources and/or targets, then a circle in the middle is added. The circle contains a mark for the relationship sources (targets respectively) to represent how sources (resp. targets) are composed:

- $\wedge$ for conjunction;
- $\vee$ for disjunction;
- nothing if the relationship has only one source (resp. target).

An example of extended response is shown in figure 5(b); it states that, after having received both a price proposal from the customer about an item and an acknowledgement from the warehouse about the availability of an item, then the seller should choose to accept or refuse customer's offer.

By allowing a composite conjunction of activities as a source, we have to extend also the handling of temporal constraints. Let us consider for example the extended response of figure 5(b): the execution of the refuse or acceptance activities should happen after both the source activities have been performed, i.e. after the execution time of *the last activity of the source*. An extended precedence relation with conjunct

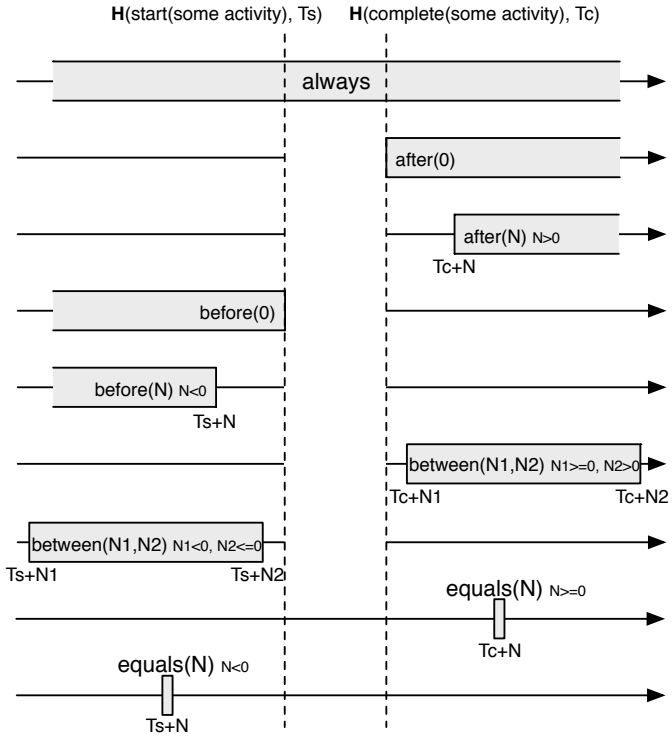


Figure 6. *Temporal constraints templates for non-atomic activities*

source activities would exhibit the opposite behaviour (i.e. the target would be expected to be performed before the time of the first source activity).

Note that this behaviour resembles the case of a non-atomic activity: we could consider the conjunction of a set of source activities as a single non-atomic activity, with the following features:

- its start time is the execution time of the first (atomic) activity in the series;
- its completion time is the execution time of the last (atomic) activity in the series.

Figure 6 shows how the proposed temporal constraints templates are affected by considering non-atomic activities (hence also conjunctions of atomic activities).

Finally, an example which mix composite activities and temporal constraints is shown figure 7. The simple DecSerFlow model represents the invitation fragment of a screening protocol. After having invited a patient, the screening center waits for a positive or negative response (but not both) within 7 days; if no answer is received within this deadline, on the 8th day a new invitation is sent.

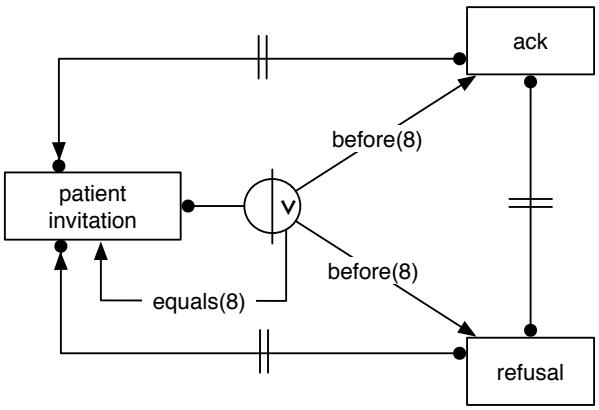


Figure 7. *A simple DecSerFlow model with composite activities and temporal constraints*

9 The Acme Travel Example

In this section we show how the example introduced in [33] could be represented in our framework and how $\mathcal{S}$CIFF is able to perform the conformance verification on a simple execution trace.

9.1 Formalization of the Acme Model

The Acme travel example is shown in figure 8 (for its complete description, the interested reader could refer to [33]).

Its mapping into the $\mathcal{S}$CIFF framework is straightforward: we have just to compile a knowledge base KB_{spec} specifying the structure of the model. The entire translation is listed in table 9.1; some relation and negation formulae use an extended notation, to directly represent disjunction or conjunction of targets. Furthermore, note that an equivalent formalization could be obtained by imposing negation successions instead of negation responses (we have indeed seen that the two formulae are equivalent w.r.t. conformance). It is worth noting also that the presence of a disjunction within a succession formula (see line 2 of the specification 9.1) is correctly interpreted by $\mathcal{S}$CIFF as a deferred choice following the response side, and as a simple merge following the precedence one.

9.2 Conformance evaluation on a simple execution trace

We now briefly describe how $\mathcal{S}$CIFF is able to prove conformance of an execution trace w.r.t. the Acme Travel example. In the following, we will use a compact notation to represent disjunction of expectations.

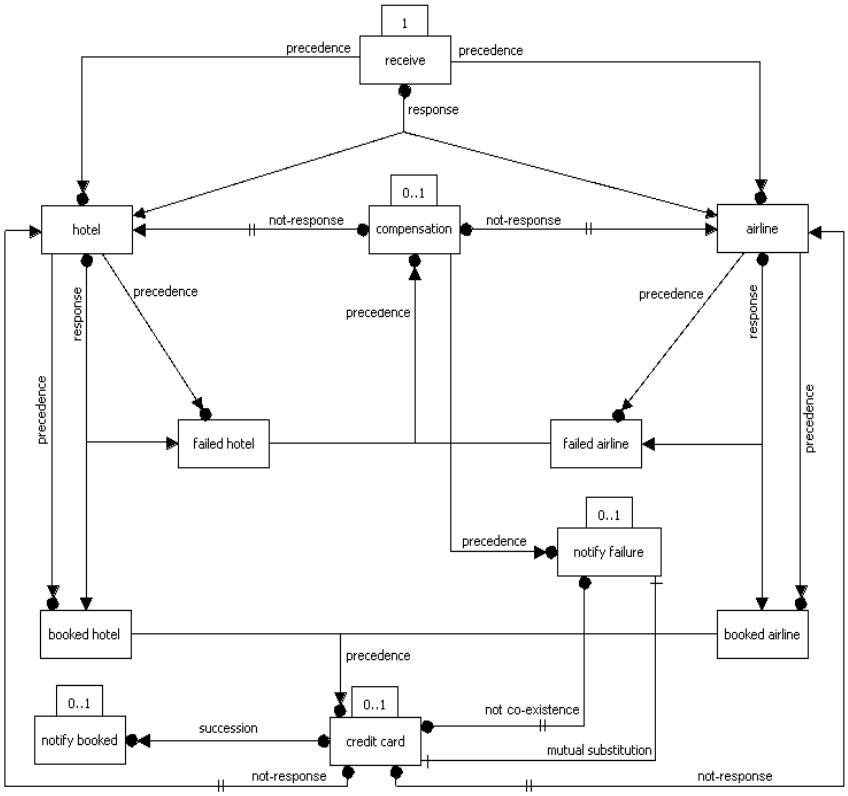


Figure 8. *Example of a DecSerFlow model [33]*

Initially, $\mathcal{S}$CIFF abduces a set of expectations about the (existence side of the) exactly_N formula [9] and the mutual substitution formula of the Acme example, stating that one *receive* message is expected and one among the *credit_card* or the *notify_failure* messages is expected too:

$$\begin{aligned}\mathbf{PEND}_0 = \{ \quad & \mathbf{E}(performed(receive), T_r), && (by\ (12)) \\ & \mathbf{E}(performed(credit\_card), T_{cc}) \\ \vee\ & \mathbf{E}(performed(notify\_failure), T_{nf})\} && (by\ (22)) \\ \mathbf{FULF}_0 = \quad & \emptyset\end{aligned}$$

Let us now consider a simple log, to the aim of analyzing the evolution of the $\mathcal{S}$CIFF proof procedure when proving if it is conformant or not:

1. $\mathbf{H}(performed(receive), 1)$. The exchange of the *receive* message at time 1 satisfies the corresponding expectation, triggering at the same time the (absence side of the) corresponding exactly_N formula (by forbidding further exchanges of this message). It also leads, by applying the response side of its succession

[9] Remember indeed that some template formulae are defined in terms of two other ones.

Table 9.1 $\mathcal{S}$CIFF representation of the Acme Travel example

```
1  existence_formula(receive, exactly_N(1)).
2  relation_formula(receive, or([hotel,airline]), succession).
3
4  relation_formula(hotel, or([failed_hotel,booked_hotel]), succession).
5
6  relation_formula(airline, or([failed_airline,booked_airline]), succession).
7
8  existence_formula(compensation, absence_N(1)).
9  negation_formula(compensation, and([airline,hotel]), negation_response).
10 relation_formula(compensation, or([failed_hotel,failed_airline]), precedence).
11
12 existence_formula(credit_card, absence_N(1)).
13 relation_formula(credit_card, or([booked_hotel,booked_airline]), precedence).
14 negation_formula(credit_card, notify_failure, not_coexistence).
15 negation_formula(credit_card, and([airline,hotel]), negation_response).
16 relation_formula(credit_card, notify_booked, succession).
17 relation_formula(credit_card, notify_failure, mutual_substitution).
18
19 existence_formula(notify_booked,absence_N(1)).
20
21 existence_formula(notify_failure, absence_N(1)).
22 relation_formula(notify_failure, compensation, precedence).
```

formula (line 2 in 9.1), to expecting the disjunction of *hotel* or *airline* booking at a time greater than 1:

$$\begin{aligned}
\mathbf{PEND}_1 = \{ \quad & \mathbf{EN}(performed(receive), T_r) \wedge T_r > 1, \ (by\ (13)) \\
& \mathbf{E}(performed(hotel), T_h) \wedge T_h > 1 \\
\vee\ & \mathbf{E}(performed(airline), T_a) \wedge T_a > 1, \quad (by\ (15)) \\
& \mathbf{E}(performed(credit_card), T_{cc}) \\
\vee\ & \mathbf{E}(performed(notify_failure), T_{nf})\} \\
\mathbf{FULF}_1 = \{ \quad & \mathbf{E}(performed(receive), 1)\}
\end{aligned}$$

2. $\mathbf{H}(performed(airline), 4)$. This event fulfills the expectation about the booking request, generating a backward (successfull) expectation to ensure that a *receive* message has been exchanged before it (by applying the precedence side of the booking succession formula, see line 2 in 9.1); moreover, it triggers the succession formula (line 6) which states that the indication of a correct or failed booking is expected:

$$
\begin{aligned}
\mathbf{PEND}_2 = \{ \quad & \mathbf{EN}(performed(receive), T_r) \wedge T_r > 1, \\
& \mathbf{E}(performed(booked\_airline), T_b) \wedge T_b > 4 \\
\vee \ & \mathbf{E}(performed(failed\_airline), T_f) \wedge T_f > 4, \ (by\ (15)) \\
& \mathbf{E}(performed(credit\_card), T_{cc}) \\
\vee \ & \mathbf{E}(performed(notify\_failure), T_{nf})\} \\
\mathbf{FULF}_2 = \{ \quad & \mathbf{E}(performed(receive), 1), \\
& \mathbf{E}(performed(airline), 4) \wedge 4 > 1, \\
& \mathbf{E}(performed(receive), 1) \wedge 1 < 4\} \qquad (by\ (16))
\end{aligned}
$$

3. $\mathbf{H}(performed(booked\_airline), 10)$. The happening of a positive feedback about the booking satisfies the expectations generated at the previous step. Furthermore, the precedence side of the booking indication formula (line 6) is triggered and satisfied (a preceding corresponding request is actually contained in the execution trace).

$$
\begin{aligned}
\mathbf{PEND}_3 = \{ \quad & \mathbf{EN}(performed(receive), T_r) \wedge T_r > 1, \\
& \mathbf{E}(performed(credit\_card), T_{cc}) \\
\vee \ & \mathbf{E}(performed(notify\_failure), T_{nf})\} \\
\mathbf{FULF}_3 = \{ \quad & \mathbf{E}(performed(receive), 1), \\
& \mathbf{E}(performed(airline), 4), \\
& \mathbf{E}(performed(booked\_airline), 10) \wedge 10 > 4, \\
& \mathbf{E}(performed(airline), 4) \wedge 4 < 10\} \qquad (by\ (16))
\end{aligned}
$$

4. $\mathbf{H}(performed(credit\_card), 12)$. The charging of the successful booking is source of many formulae (lines 12-17). More specifically, its happening satisfies the mutual substitution formula (line 17) and triggers:

   - the absence_N formula (line 12), forbidding further payments;
   - a backward expectation about the presence of a positive answer w.r.t. the booking of an airline or hotel before time 12 (line 13), actually fulfilled by the execution trace;
   - a negative expectation to impose the non existence of a failure notification in the log (line 14);
   - two forward negative expectations to impose that no more hotel and airline bookings can be requested (line 15);
   - an expectation about the last message of the execution, i.e. about the presence of a booking notification back to the client after time 12 (response side of the succession formula at line 16).

$$
\begin{aligned}
\mathbf{PEND}_4 = \{\ & \mathbf{EN}(performed(receive), T_r) \wedge T_r > 1, \\
& \mathbf{EN}(performed(credit\_card), T_{cc}) \wedge T_{cc} > 12, && (by\ (13)) \\
& \mathbf{EN}(performed(notify\_failure), T_{nf}), && (by\ (23)) \\
& \mathbf{EN}(performed(hotel), T_h) \wedge T_h > 12, && (by\ (26)) \\
& \mathbf{EN}(performed(airline), T_a) \wedge T_a > 12, && (by\ (26)) \\
& \mathbf{E}(performed(notify\_booked), T_{nb}) \wedge T_{nb} > 12\} && (by\ (15)) \\
\mathbf{FULF}_4 = \{\ & \mathbf{E}(performed(receive), 1), \\
& \mathbf{E}(performed(airline), 4), \\
& \mathbf{E}(performed(booked\_airline), 10), \\
& \mathbf{E}(performed(credit\_card), 12), \\
& \mathbf{E}(performed(booked\_airline), 10) \wedge 10 < 12\} && (by\ (16))
\end{aligned}
$$

5. $\mathbf{H}(performed(notify\_booked), 18)$. The booking notification satisfies the expectation generated at the previous step, generating a backward successful expectation about the presence of a credit card payment before it.

$$
\begin{aligned}
\mathbf{PEND}_5 = \{\ & \mathbf{EN}(performed(receive), T_r) \wedge T_r > 1, \\
& \mathbf{EN}(performed(credit\_card), T_{cc}) \wedge T_{cc} > 12, && (by\ (13)) \\
& \mathbf{EN}(performed(notify\_failure), T_{nf}), \\
& \mathbf{EN}(performed(hotel), T_h) \wedge T_h > 12, && (by\ (26)) \\
& \mathbf{EN}(performed(airline), T_a) \wedge T_a > 12, && (by\ (26)) \\
\mathbf{FULF}_5 = \{\ & \mathbf{E}(performed(receive), 1), \\
& \mathbf{E}(performed(airline), 4), \\
& \mathbf{E}(performed(booked\_airline), 10), \\
& \mathbf{E}(performed(credit\_card), 12)\}
\end{aligned}
$$

Note that, at this point, there does not exists a pending positive expectation any more. This means that, if the execution trace is completed, $\mathcal{S}$CIFF evaluates it as conformant:

- all the positive expectations have been fulfilled by a matching happened event;
- no more events will happen and, therefore, all the negative expectations are fulfilled, too.

Let us now instead considering the case in which the log still contains an happened event, namely

$$\mathbf{H}(performed(hotel), 20)$$

This event clearly violates the Acme DecSerFlow model, since no more booking request could be made after having executed the credit card payment.

$\mathcal{S}$CIFF correctly detects the non conformance; since the happening of an hotel booking request at time 20 matches a negative expectation contained in the set $\mathbf{PEND}_5$, i.e.

$$\mathbf{EN}(performed(hotel), T_h) \wedge T_h > 12$$

the execution trace is evaluated as non conformant (see Definition 2.5).

10 Conclusions and Future Works

In this preliminary work we have exploited a computational logic-based framework, called $\mathcal{S}$CIFF, to the aim of giving an abductive formalization to DecSerFlow. We have shown how the different DecSerFlow template formulae could be translated into general $\mathcal{S}$CIFF rules (valid for all models), making possible to formalize a DecSerFlow model by simply compiling a corresponding knowledge base.

Furthermore, we have sketched how the different DecSerFlow formulae could be extended in order to deal with composite activities (following what has been proposed in [33]) and to explicitly express temporal constraints.

A first implementation of "basic" formulae (i.e. existence formulae and binary relationships) has been developed; the implementation of the extended formulae has partially been implemented and is yet under study.

As future works, after having completed our formalization and having extensively tested and verified it, we foresee two main research directions:

- To exploit how the $\mathcal{S}$CIFF formalization could be used not only for verification purposes, but also to properly enact the execution of one or more services following a DecSerFlow model. A preliminary study of using $\mathcal{S}$CIFF to animate interacting agents can be found in [1].

- To consider also content data of activities and content-based choices. Anyway, note that the formalization of these aspects is seamlessly addressed by our framework (through CLP constraints and the Prolog knowledge base).

Finally, we envisage a third research direction, whose aim is to learn a set of $\mathcal{S}$CIFF Integrity Constraints from a set of execution traces. Sometimes, the interaction protocol to be modeled is not fully formalized; this may cause the corresponding DecSerFlow model to be incomplete, thus leading to misclassifications (conformant service interaction execution traces classified as non conformant and vice versa).

In order to support the definition of the DecSerFlow model in such situations, we are working on a methodology for analyzing a set of execution logs, manually labeled as conformant or not, and learning the minimal set of $\mathcal{S}$CIFF rules capable to correctly classify them. The learned constraints are very similar to the ones proposed in this work for mapping the different DecSerFlow template formulae, so the basic idea is to translate them back to a DecSerFlow format. The interested reader is referred to [26] for a preliminary work on this topic.

11 Acknowledgements

This work has been partially supported by the MIUR PRIN 2005 project *Specification and verification of agent interaction protocols* and by the MIUR FIRB project *TOCAI.IT, Tecnologie Orientate alla Conoscenza per Aggregazione di Imprese in Internet*. We would like to thank Prof. van der Aalst for the valuable discussion at the BPM 2006 conference and for having inspired this work. Furthermore, a special thank goes to Marco Alberti, Marco Gavanelli, Evelina Lamma and Paolo Torroni, who have actively contributed to the work here presented.

A Abductive Event Calculus

AEC [13] is a classical application of abductive proof-procedures, and it can be used for planning in agent systems [22, 28]. $\mathcal{S}$CIFF easily accomodates AEC; we will give two different formulations of it. The first one, developed by Marco Gavanelli, is particularly suitable to treat planning problems, whereas the latter could be useful when proving conformance of an execution trace (e.g. because a non-atomic approach for activities is adopted).

In order to understand AEC we must give some background.

A.1 Event Calculus

The Event Calculus (EC) [24, 29] is a framework to reason about properties (called *fluents*) that may hold in a system inside time intervals. The EC consists of four ingredients:

1. A set of known causal relations, stating which events initiate or terminate the validity of a fluent. For example, in the description of a robot in the block world we can imagine the fluents $ontable(X)$ (block X is on the table) and $holding(X)$ (the robot holds in its hand the block X). Rules could state that if the robot is holding the block, then the action of putting a block on the table initiates the fluent "block X is on the table":

   $$initiates(putdown(X), ontable(X), T) \leftarrow holdsat(holding(X), T).$$

   On the contrary, the fluent "block X is on the table" is terminated by the action of picking X up. Therefore the definition:

   $$terminates(pickup(X), ontable(X)).$$

2. The initial situation provided by the *initially* predicate. For example, the robot is initially holding block number 1:

   $$initially(holding(1)).$$

3. A narrative of happened events; for example

   $$happens(putdown(1), 3).$$
   $$happens(pickup(1), 5).$$

4. The general theory of EC, defined as a set of domain-independent rules which state that a fluent holds at a given time if it was either initially true, or if it

has become true after an event, and it has not ever since been *clipped*, i.e., its truth has not been terminated in the meanwhile.

$$\begin{array}{rl} holdsat(F,T) \leftarrow & initially(F), not\ clipped(0,F,T). \\ holdsat(F,T) \leftarrow & happens(E,T_1), initiates(E,F), \\ & not\ clipped(T_1,F,T). \\ clipped(T_1,F,T_2) \leftarrow & happens(E,T), T_1 < T < T_2, terminates(E,F). \end{array} \tag{36}$$

Based on this theory, by deduction one can infer for instance that the fluent $ontable(1)$ is true at time 4 and it is false at times 2 and 10.

A.2 Abductive Event Calculus in $\mathcal{S}$CIFF

Building on this result, Eshghi [13] pointed out that planning problems can be solved by interpreting the event calculus in abduction. The user states the initial situation (through the *initially* predicate) and a goal, typically requiring the validity of some fluents in the final situation. The narrative of events is no longer given, but is considered as a set of actions that should be performed in order to obtain the goal; i.e., *happens* atoms are abducible. In the example, if the goal was

$$holdsat(ontable(1), 10) \tag{37}$$

the Abductive Event Calculus (AEC) would reply that in order to obtain the goal, the *putdown* action should be performed before time 10.

Many other authors address planning through abduction. Some of them consider the precedence relationship between events to be abducible [30]. Others use a discrete representation of time and thus rely on efficient constraint solvers [21, 12].

The $\mathcal{S}$CIFF framework easily accommodates the AEC. Additionally, it keeps happened events separate from expected events, which we consider to be an improvement, in terms of representation: what is supposed to happen, not necessarily coincides with what is actually happening. An agent could plan to perform an action, but the action might fail. In the blocks world example, a block could slip, thus making a *pickup* action unsuccessful. The robot expected to pickup the block, but the actual action did not match. This unexpected event generates the need for alternative possible course of events, such as a retrial, or a totally different plan.

Therefore, in this implementation of AEC via $\mathcal{S}$CIFF, plans are defined through **E** predicates rather than **H** events. If the agent wants to get to a goal state, it should perform plan for and execute actions, which makes such actions *expected*: by no means, the actions in the plan are already *happened* at planning time.

Positive **E** expectations state actions that should be taken in order for the plan to be effective. Negative **EN** expectations (which do not exist in previous abductive event calculus proposals) inform about those actions that should not be executed in order for the plan to be successful.

Table A.1 Abductive Event Calculus theory in $\mathcal{S}$CIFF.

$\mathcal{IC}_S$:

$$\mathbf{unclipped}(T_1, F, T_2), \mathit{terminates}(A, F) \rightarrow \mathbf{EN}(A, T), T_1 < T < T_2.$$

KB :

$$\mathit{holdsat}(F, T) \leftarrow \mathit{initially}(F), \mathbf{unclipped}(0, F, T).$$
$$\mathit{holdsat}(F, T) \leftarrow \mathbf{E}(A, T_1), 0 < T_1 < T, \mathit{initiates}(A, F), \mathbf{unclipped}(T_1, F, T).$$

The $\mathcal{S}$CIFF implementation of the AEC theory is shown in Tab. A.1.

The rules in the KB are direct translation of the first two in Eq. (36). We introduce a new abducible predicate, **unclipped**, to represent the *not*(*clipped*) literals found in the classical event calculus. In order for the plan to be successful, the fluent should not be clipped in the given time interval. This is ensured by the application of the integrity constraint in Tab. A.1, which imposes that every event that would terminate the validity of the fluent is expected not to happen (in the given time interval). This mechanism lets us exploit better the underlying constraint solver, which is tailored to reason about positive and negative expectations. Moreover, the planner will explicitly provide, in the form of negative (**EN**) expectations, which actions should be avoided in order not to endanger the execution of the plan.

In the blocks world example $\mathcal{S}$CIFF provides

$$\mathbf{E}(\mathit{putdown}(1), T_1), \mathbf{EN}(\mathit{pickup}(1), T_2), T_1 < T_2 < 10$$

i.e., in order to achieve the goal $\mathit{holdsat}(\mathit{ontable}(1), 10)$, expressed in Eq. (37), the robot should drop block 1, and avoid picking it up before time 10. The times of these planned actions are given in terms of domains (intervals), thanks to the underlying constraint solver. In order to obtain punctual times, one can anytime resort to grounding. This very useful feature is present in $\mathcal{S}$CIFF as well as in other frameworks of literature, such as *ACLP* [20] and $\mathcal{A}$-System [21].

A.3 Abcutive Event Calculus in $\mathcal{S}$CIFF: another proposal

The AEC formalization shown in the previous section is suitable to deal with planning problems. However, we are also interested in finding a good formulation capable to capture run-time aspects, i.e. to automatically infer the validity of fluents by analyzing an execution trace.

To this aim, we make use of four different Integrity Constraints:

1. If a fluent initially holds, then it is **unclipped** from the initial time (i.e. 0) till an unknown time T_f;

2. When an event happens at a certain time (say, T), and this event initiates a certain fluent, then the fluent becomes **unclipped** from T till a time T_f;

3. If a fluent is **unclipped** from T_1 to T_2, then one of the events capable to terminate the fluent is expected to happen at time T_2;

4. If a fluent is **unclipped** from T_1 to T_2, then no event which terminates the fluent can happen inside the time interval (T_1, T_2).

The $\mathcal{S}$CIFF implementation that follows this point of view is described in table A.2.

Table A.2 Another formulation of Abductive Event Calculus theory in $\mathcal{S}$CIFF.

$\mathcal{IC}_S$:

$$\mathit{initially}(F) \rightarrow \mathbf{unclipped}(0, F, T_f) \wedge T_f > 0$$
$$\mathbf{H}(Ev, T) \wedge \mathit{initiates}(Ev, F) \rightarrow \mathbf{unclipped}(T, F, T_f) \wedge T_f > T$$
$$\mathbf{unclipped}(T_1, F, T_2) \rightarrow \mathit{initiates}(Ev, F) \wedge \mathbf{E}(Ev, T_2).$$
$$\mathbf{unclipped}(T_1, F, T_2) \wedge \mathit{terminates}(Ev, F) \rightarrow \mathbf{EN}(Ev, T) \wedge T > T_1 \wedge T < T_2.$$

KB :

$$\mathit{holdsat}(F, T) \leftarrow \mathit{initially}(F), \mathbf{unclipped}(0, F, T_2), T < T_2.$$
$$\mathit{holdsat}(F, T) \leftarrow \mathbf{E}(A, T_1), 0 < T_1 < T, \mathit{initiates}(A, F), \mathbf{unclipped}(T_1, F, T_2), T < T_2.$$

References

[1] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, and P. Mello. A verifiable logic-based agent architecture. In Floriana Esposito, Zbigniew W. Ras, Donato Malerba, and Giovanni Semeraro, editors, *Foundations of Intelligent Systems, 16th International Symposium, ISMIS 2006, Bari, Italy, September 27-29, 2006, Proceedings*, volume 4203 of *Lecture Notes in Computer Science*, pages 188–197. Springer, 2006.

[2] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, M. Montali, S. Storari, and P. Torroni. Computational logic for run-time verification of web services choreographies: Exploiting the *socs-si* tool. In Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro, editors, *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*, pages 58–72. Springer, 2006.

[3] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Compliance verification of agent interaction: a logic-based software tool. *Applied Artificial Intelligence*, 20(2-4):133–157, February-April 2006.

[4] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Verifiable agent interaction in abductive logic programming: the sciff proof-procedure. Technical Report DEIS-LIA-06-001, DEIS, Bologna, Italy, 2006.

[5] M. Alberti, M. Gavanelli, E. Lamma, F. Chesani, P. Mello, and M. Montali. An abductive framework for a-priori verification of web services. In Annalisa Bossi and Michael J. Maher, editors, *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 10-12, 2006, Venice, Italy*, pages 39–50. ACM, 2006.

[6] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. *Business Process Execution Language for Web Services version 1.1*, 2003. Available at http://www-128.ibm.com/developerworks/library/specification/ws-bpel/.

[7] M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. A priori conformance verification for guaranteeing interoperability in open environments. In Asit Dan and Winfried Lamersdorf, editors, *Service-Oriented Computing - ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, 2006, Proceedings*, volume 4294 of *Lecture Notes in Computer Science*, pages 339–351. Springer, 2006.

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

[9] F. Bry, M. Eckert, and P. Patranjan. Reactivity on the web: Paradigms and applications of the language xchange. *Journal of Web Engineering*, 5(1):3–24, 2006.

[10] K. L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[11] M. Denecker and D. De Schreye. SLDNFA: an abductive procedure for abductive logic programs. *Journal of Logic Programming*, 34(2):111–167, 1998.

[12] U. Endriss, P. Mancarella, F. Sadri, G. Terreni, and F. Toni. The ciff proof procedure for abductive logic programming with constraints. In José Júlio Alferes and João Alexandre Leite, editors, *Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004, Proceedings*, volume 3229 of *Lecture Notes in Computer Science*, pages 31–43. Springer, 2004.

[13] K. Eshghi. Abductive planning with the event calculus. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington*, Cambridge, MA, 1988. MIT Press.

[14] T. H. Fung and R. A. Kowalski. The IFF proof procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165, November 1997.

[15] M. Gavanelli, E. Lamma, and P. Mello. 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/.

[16] D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 412, Washington, DC, USA, 2001. IEEE Computer Society.

[17] J. Jaffar and M.J. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19-20:503–582, 1994.

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

[19] A. C. Kakas and P. Mancarella. On the relation between Truth Maintenance and Abduction. In T. Fukumura, editor, *Proceedings of the 1st Pacific Rim International Conference on Artificial Intelligence, PRICAI-90, Nagoya, Japan*, pages 438–443. Ohmsha Ltd., 1990.

[20] A. C. Kakas, A. Michael, and C. Mourlas. ACLP: Abductive Constraint Logic Programming. *Journal of Logic Programming*, 44(1-3):129–177, July 2000.

[21] A. C. Kakas, B. van Nuffelen, and M. Denecker. A-System: Problem solving through abduction. In B. Nebel, editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, Seattle, Washington, USA (IJCAI-01)*, pages 591–596, Seattle, Washington, USA, August 2001. Morgan Kaufmann Publishers.

[22] A.C. Kakas, P. Mancarella, F. Sadri, K. Stathis, and F. Toni. The KGP model of agency. In R. Lopez de Mantaras and L. Saitta, editors, *Proceedings of the Sixteenth European Conference on Artificial Intelligence, Valencia, Spain (ECAI 2004)*. IOS Press, August 2004.

[23] K.Havelund and G. Rosu. Synthesizing monitors for safety properties. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002,*

Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.

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

[25] K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4(4):289–308, 1987.

[26] E. Lamma, P. Mello, F. Riguzzi, and S. Storari. A methodology for learning social integrity constraints from labeled service interaction logs. Technical Report DEIS-LIA-07-001, DEIS, Bologna, Italy, 2007.

[27] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd extended edition, 1987.

[28] P. Mancarella, F. Sadri, G. Terreni, and F. Toni. Planning partially for situated agents. In João Alexandre Leite and Paolo Torroni, editors, *Computational Logic in Multi-Agent Systems, 5th International Workshop, CLIMA V, Lisbon, Portugal, September 29-30, 2004, Revised Selected and Invited Papers*, volume 3487 of *Lecture Notes in Computer Science*, pages 230–248. Springer, 2005.

[29] M. Shanahan. The event calculus explained. In Michael Wooldridge and Manuela M. Veloso, editors, *Artificial Intelligence Today: Recent Trends and Developments*, volume 1600 of *Lecture Notes in Computer Science*, pages 409–430. Springer Verlag, 1999.

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

[31] 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/`.

[32] 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 Mario Bravetti, Leïla Kloul, and Gianluigi Zavattaro, editors, *EPEW/WS-FM*, volume 3670 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2005.

[33] W. M. P. van der Aalst and M. Pesic. Decserflow: Towards a truly declarative service flow language. In Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro, editors, *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*, pages 1–23. Springer, 2006.

[34] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[35] W3C. Web services choreography description language version 1.0. Home Page: `http://www.w3.org/TR/ws-cdl-10/`.