Verification of Choreographies During Execution Using the Reactive Event Calculus

Federico Chesani, Paola Mello, Marco Montali, and Paolo Torroni

DEIS - University of Bologna, V.le Risorgimento 2, 40136 Bologna - Italy
{federico.chesani,paola.mello,marco.montali,paolo.torroni}@unibo.it

Abstract. This article presents a run-time verification method of web service behaviour with respect to choreographies. We start from DecSerFlow as a graphical choreography description language. We select a core set of DecSerFlow elements and formalize them using a reactive version of the Event Calculus, based on the computational logic SCIFF framework. Our choice enables us to enrich DecSerFlow and the Event Calculus with quantitative time constraints and to model compensation actions.

1 Introduction

Recent years have seen a wide adoption of the Service-Oriented Architecture (SOA) paradigm, both in the research field as well as in industrial settings, to enable distributed applications within intra- and inter-organizational scenarios. Such applications typically consist of a composition of heterogenous interacting services, each one providing a specific functionality. Complex business processes are realized by properly guiding and constraining service interactions. When collaboration is performed across different organizations, *service choreographies* come into play. A choreography models the interaction from a global viewpoint. As stated by the authors of WS-CDL [1], "*[a choreography] offers a means by which the rules of participation within a collaboration can be clearly defined and agreed to, jointly.*"

Recent research has demonstrated a possible use of choreographies before service execution, either to establish an agreement among services [2,3], or to derive skeletons of local models [4,5] to be used for implementing the services. A different issue is to verify that a running service follows a given choreography. This is a task that has to be carried out during execution, when potential mismatches between a service's behavioural interface and its real implementation may lead to unexpected/undesired interactions. Therefore, monitoring and verifying the behaviour of services at execution time is a fundamental requirement.

Choreographies often involve the specification of complex constraints, such as conditions on the reached state or the possibility of violating certain prescriptions, at the expense of some compensating activity. Suitable, expressive formalisms are needed to model such constraints in an accurate way. Candidates could be temporal logic languages, such as linear temporal logic (LTL), branching time temporal logic (CTL) or CTL* [6], which can encode formulae such

R. Bruni and K. Wolf (Eds.): WS-FM 2008, LNCS 5387, pp. 55–72, 2009.

as that a condition will eventually be true, or a condition must be true until another one becomes true, etc. However, these logics do not accommodate quantitative time, i.e, they enable reasoning about what happens "next" or "at some point in the future," but not "before 60 time ticks." Extensions to temporal logic languages, such as metric temporal logic [7], have been proposed to accommodate explicit time, but they can be hardly used for runtime verification because of their high computational complexity [8]. The well known "state-explosion" problem for temporal logics is even more critical when considering declarative specification languages such as DecSerFlow [9], where the system itself is specified as a conjunction of LTL formulae.

An alternative to temporal logics is the Event Calculus [10] ($\mathcal{EC}$ for short). Many authors believe the $\mathcal{EC}$ to be well suited for expressing the complex constraints of choreographies, especially because it enables the modeler to specify temporal requirements, in a declarative and easily understandable way. In fact, the $\mathcal{EC}$ has been (and is being) extensively applied in the SOA setting. However, little emphasis has been given so far to the possible adoption of the $\mathcal{EC}$ for performing compliance verification of service interaction during execution. We believe that this is mainly due to the lack of suitable underlying reasoning tools.

In this paper, we propose to adopt a reactive version of Event Calculus ($\mathcal{REC}$[11]) to perform run-time verification of the observed behaviour. $\mathcal{REC}$ is formalized as an axiom theory on top of the $\mathcal{S}$CIFF framework [12], a logic based formalism with a sound and complete proof procedure and an efficient implementation [13]. The literature is rich in languages proposed to specify service choreographies. WS-CDL [1] is one of the most prominent procedural ones. We have chosen to represent choreographies in DecSerFlow [9], a graphical representation language introduced by van der Aalst and Pesic to specify and constrain service flows in a declarative manner. This choice is motivated by the capability of DecSerFlow to capture in a flexible and concise way the "contractual nature" of choreographies. However, our approach based on $\mathcal{REC}$ is general and does not depend on a specific choreography specification language.

Besides providing a mapping from DecSerFlow to $\mathcal{REC}$, in this article we show how the approach can be easily extended (by adding new axioms) to support deadlines modeling and verification, and to reify the violations generated by the proof procedure during verification. This latter feature gives us two main advantages: *(i)* when a violation is detected, the proof does not terminate reporting the error, but continues the verification task; *(ii)* violations can be notied to the user, and even considered as rst-class objects during the modeling phase: hence compensation mechanisms related to the violation can be easily specified.

We show the benefits of our approach by way of a motivating example.

2 Background

In this section we briefly introduce the two components of our run-time verification framework, namely DecSerFlow as a specification language, and $\mathcal{REC}$ as its underlying reasoning mechanism.

2.1 DecSerFlow

DecSerFlow is a graphical language which specifies service flows by adopting a declarative style of modeling. Instead of defining rigid service flows, which may lead–especially with procedural languages like WS-CDL and BPEL–to over-specified and over-constrained models, DecSerFlow focuses on the minimal set of constraints which must be satisfied in order to correctly carry out the interaction. This makes DecSerFlow especially suited for representing the "contractual nature" of service choreographies. A DecSerFlow model is composed by a set of activities, which represent atomic units of work (such as message exchanges), and relations among activities, used to specify constraints on their execution. DecSerFlow provides constructs to define positive and negative constraints, that specify the desired and undesired courses of interaction while leaving undefined other possibilities of interaction that are neither desired nor undesired. Positive and negative constraints make the DecSerFlow approach *open*: services can interact freely unless when in the presence of constraints.

DecSerFlow constraints are grouped into three families (see Table 1, 2 and 3 for a complete description of all the basic constraints):

– *existence constraints*: unary cardinality constraints expressing how many times an activity can/should be executed;
– *relation constraints*: binary constraints which impose the presence of a certain activity when some other activity is performed;
– *negation constraints*: the negative version of relation constraints, used to explicitly forbid the execution of a certain activity when some other activity is performed.

Intuititely, a service composition is compliant with a DecSerFlow choreography if all positive constraints are eventually satisfied, and no activity forbidden by any negation constraint is performed. The DecSerFlow semantics is defined for finite execution traces.

Table 1. DecSerFlow existence constraints. In [9], `choice` used to be called `mutual substitution` and had a slightly different notation.

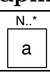
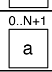
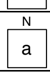
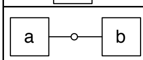
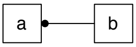
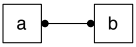
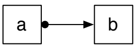
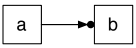
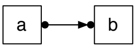
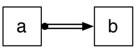
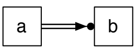
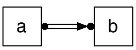
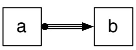
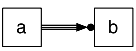
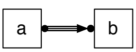
graphical	**description**	**equivalent to**
	`existence(N,a)`. `a` must be executed at least N times	basic
	`absence(N+1,a)`. `a` cannot be executed more than N times	basic
	`exactly(N,a)`. `a` must be executed exactly N times	`existence(N,a)`$\wedge$ `absence(N+1,a)`
	`choice(a,b)`. At least one activity among `a` and `b` must be executed	`existence(1,a`$\vee$`b)`

Table 2. An overview of DecSerFlow relation constraints

graphical	description	equivalent to
	`responded existence(a,b)`. If `a` is executed, then `b` must be executed (before or after `a`)	basic
	`coexistence(a,b)`. Either both `a` and `b` are executed, or none of them is executed	`resp. existence(a,b)`$\wedge$ `resp. existence(b,a)`
	`response(a,b)`. If `a` is executed, then `b` must be executed afterwards	basic
	`precedence(a,b)`. `b` can be executed only after `a` is executed	basic
	`succession(a,b)`.	`response(a,b)`$\wedge$ `precedence(a,b)`
	`alternate response(a,b)`. `b` is response of `a` and there has to be a `b` between two `a`	`response(a,b)`$\wedge$ `interposition(a,b,a)`
	`alternate precedence(a,b)`. `b` is preceded by `a` and there has to be an `a` between two `b`	`precedence(a,b)`$\wedge$ `interposition(b,a,b)`
	`alternate succession(a,b)`.	`alt. response(a,b)`$\wedge$ `alt. precedence(a,b)`
	`chain response(a,b)`. If `a` is executed then `b` is executed next (immediately after `a`)	`response(a,b)`$\wedge$ `interposition(a,b,X)` $\wedge$`X` $\neq$ `b`
	`chain precedence(a,b)`. `b` can be executed only if `a` was the last executed activity	`precedence(a,b)`$\wedge$ `interposition(X,a,b)` $\wedge$`X` $\neq$ `a`
	`chain succession(a,b)`. `a` and `b` are always executed next to each other	`chain response(a,b)`$\wedge$ `chain precedence(a,b)`

2.2 $\mathcal{REC}$: A Reactive Event Calculus in $\mathcal{S}$CIFF

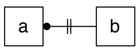
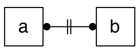
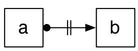
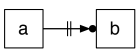
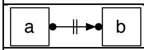
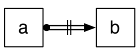
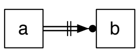
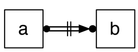
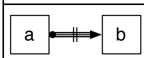
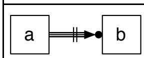
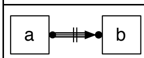
The Event Calculus ($\mathcal{EC}$) is a framework, based on first-order logic, which enables reasoning about the effects of events [10,14]. The basic elements of the calculus are *events* which happen during the execution[1], and properties (called *fluents*) which describe a partial state of the world. To model a given event-based system, the user must simply provide a declarative description of how possible occurring events affect the corresponding fluents.

In the classical $\mathcal{EC}$ setting, given a description of the system and a set of desired temporal requirements, two main reasoning tasks can be carried out: *narrative verification*, exploiting $\mathcal{EC}$ in a deductive manner, to check whether a given execution trace of the system satisfies the requirements, and *planning*, using abduction to simulate narratives of the systems, trying to produce a possible execution which satisfies the requirements.

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

[1] We will consider only atomic events, i.e., events occur at a certain point in time.

Table 3. An overview of DecSerFlow negation constraints

graphical	description	equivalent to
	`responded absence(a,b)`. If `a` is executed, then `b` cannot be ever executed	`not coexistence(a,b)`
	`not coexistence(a,b)`. `a` and `b` cannot be both executed	`neg. response(a,b)`$\wedge$ `neg. response(b,a)`
	`negation response(a,b)`. If `a` is executed, then `b` cannot be executed afterwards	basic
	`negation precedence(a,b)`. `b` cannot be executed if `a` was executed before	`neg. response(a,b)`
	`negation succession(a,b)`.	`neg. response(a,b)`
	`negation alternate response(a,b)`. `b` cannot be executed between two `a`	`neg. interpos.(a,b,a)`
	`negation alternate precedence(a,b)`.`a` cannot be executed between two `b`	`neg. interpos.(b,a,b)`
	`negation alternate succession(a,b)`.	`neg. alt. resp.(a,b)`$\wedge$ `neg. alt. prec.(a,b)`
	`negation chain response(a,b)`. `b` cannot be executed next to (i.e., immediately after) `a`	`interposition(a,X,b)` $\wedge$`X` $\neq$ `b`
	`negation chain precedence(a,b)`. `a` cannot be last executed activity before `b`	`n. chain response(a,b)`
	`negation chain succession(a,b)`. `a` and `b` cannot be executed next to each other	`n. chain response(a,b)`

underlying reasoning tools. In a companion paper [11], we show how the computational logic-based $\mathcal{S}$CIFF framework [12] can be adopted to provide a reactive axiomatization of $\mathcal{EC}$ (called $\mathcal{REC}$), enabling reasoning about events and fluents at run-time. $\mathcal{S}$CIFF is a framework originally designed for the specification and run-time verification of global interaction protocols in open Multi-Agent Systems. Its usage for run-time verification of service choreographies has been presented at previous editions of this workshop series [15]. $\mathcal{S}$CIFF consists of a rule-based language with a declarative semantics for specifying what are the (un)desired courses of interaction as events occur. A corresponding execution model (the $\mathcal{S}$CIFF proof-procedure [12], implemented in the SOCS-SI tool [13]) enables run-time monitoring and compliance checking of the interacting entities' behavior. The $\mathcal{S}$CIFF proof-procedure is sound and complete w.r.t. its declarative semantics, and it natively provides the capability of reasoning upon dynamically occurring events, using constraint propagation to update the status of fluents. To represent time, $\mathcal{S}$CIFF uses variables that can range over finite domains or over real numbers, and that are associated to events. Therefore, while the procedure does not model itself the flow of time, the current time can be inferred, with some approximation, from the time of occurring events. For example, the expiration of a deadline can be made known to the reasoning engine by way of "tick" event, real or fictitious such as a "tick", which occurs at or after that time.

Table 4. The $\mathcal{REC}$ ontology

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

The $\mathcal{REC}$ ontology is shown in Table 4; the main difference w.r.t. the classical $\mathcal{EC}$ ontology is that while $\mathcal{EC}$ focuses on time intervals inside which a fluent has been terminated or initiated, $\mathcal{REC}$ focuses on the maximum time intervals inside which the fluent uninterruptedly holds (represented by the `holds/3` predicate).

$\mathcal{REC}$ integrates the advantages of $\mathcal{S}$CIFF and $\mathcal{EC}$, by embedding the latter inside a framework that supports run-time reasoning, while extending $\mathcal{S}$CIFF with fluents-based reasoning.

3 Mapping DecSerFlow to Event Calculus

We now present the mapping of DecSerFlow onto $\mathcal{EC}$. To this end, we follow a two-fold approach. We first show that all DecSerFlow constraints can be represented in terms of a small core set[2]. Then, we provide a fluent-based formalizations of such a set[3].

3.1 Expressing DecSerFlow with a Core Set of Constraints

Table 1, 2 and 3 respectively recall the basic existence, relation and negation DecSerFlow constraints, by also showing how constraints can be expressed by using a small set of core constraints. To this purpose, two further ternary constraints are used; they represent the concept of positive and negative interposition. In particular, `interposition(a,b,c)` states that between any execution of activity `a` and a future execution of activity `c`, `b` must be performed at least once. `negation interposition(a,b,c)` expresses the opposite constraint, specifying that the execution of `a` and a following `c` cannot be interleaved by `b`. X is sometimes used to represent an arbitrary activity (i.e., it is a variable matching with any activity).

[2] Some equivalences are already stated in [9].

[3] We will use the Prolog notation: variables starting by upper case, constants by lower case. To differentiate between formalisms, we use `teletype` for DecSerFlow formula names, and *italics* for Prolog terms and rules in the knowledge base.

All the 26 basic DecSerFlow constraints can be expressed in terms of eight core constraints:

- the two basic cardinality constraints (`existence` and `absence`);
- the three fundamental positive temporal orderings (`responded existence` for *any* ordering, `response` for the *after* ordering, `precedence` for the *before* ordering);
- the `negation response` constraint;
- the positive/negative `interposition` patterns.

For example, the `chain response` between `a` and `b` (see Table 2) can be expressed using a `response` formula and by stating that between each occurrence of activity `a` and another arbitrary activity different than `b`, there must exist at least an intermediate execution of `b` (hence `b` is necessarily next to `a`). The `not coexistence` constraint (Table 3) can instead be reduced to two opposite `negation responses`. In fact, expressing that two activities cannot coexist in a single execution is the same as stating that the *first* happening activity forbids future executions of the other one.

3.2 A Fluent-Based Formalization of DecSerFlow

The formalization of DecSerFlow in $\mathcal{REC}$ is composed by two parts (see Figure 1 for an overview):

- a general part, which describes how the different DecSerFlow constraints can be formalized as fluents in the $\mathcal{EC}$ setting;
- a specific part, whose purpose is to describe a specific DecSerFlow diagram.

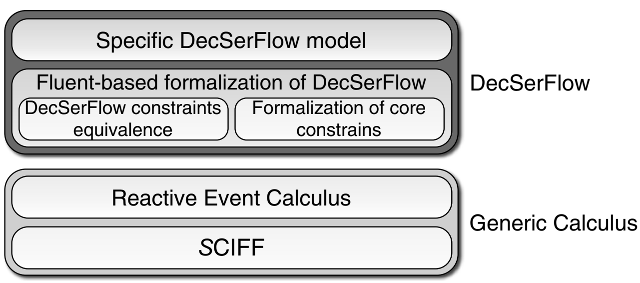


Fig. 1. Building parts of the DecSerFlow formalization in $\mathcal{REC}$

The specific part is a set of `constraint/2` facts. Each one of them corresponds to a DecSerFlow constraint in the diagram. For example, *constraint(*c_1*, response(order_item, ack))* states that the DecSerFlow choreography contains a constraint named c_1 which models a `response` between the `order_item` and `ack` activities.

The generic DecSerFlow formalization in $\mathcal{EC}$ splits itself in two sub-parts.

Formalization of constraints equivalence. The first part is a set of predicate definitions for `core_constraint/2`. They implement the reduction of the 26 basic DecSerFlow constraints to the set of eight core constraints listed above. In this way, we provide a full implementation of DecSerFlow (not only the core constraints). Examples of such definitions are those below, relating `alternate response` with `response` and `interposition`[4]:

$$core\_constraint(C, response(A, B)) \leftarrow constraint(C, alt\_response(A, B)).$$
$$core\_constraint(C, interposition(A, B, A)) \leftarrow constraint(C, alt\_response(A, B)).$$

Fluent-based formalization of the core constraints. The second part is a set of predicate definitions for `initially/1`, `initiates/3` and `terminates/3`. In other words, it is a knowledge base which formalizes constraints in terms of fluents, linking their initiation and termination to activities.

The fluents chosen to model DecSerFlow reflect the double nature of its constraints: some relations explicitly forbid the execution of a certain activity, whereas other ones express the necessity of performing some activity, becoming temporarily unsatisfied until such an activity indeed happens. More specifically, we exploit a $forbidden(C, A)$ fluent to model that an activity A is forbidden by a constraint C, and a $satisfied(C)$ fluent to model that a constraint C is satisfied.

Table 5 briefly indicates our usage of fluents in the formalization of the DecSerFlow core constraints. Some parts of the formalization are left implicit for ease of presentation. In particular, Table 5 omits the binding between each formalization and its corresponding `core_constraint`. For example, the complete formalization of `response` would be:

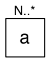
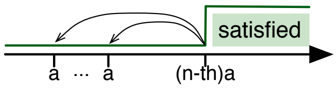
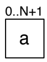
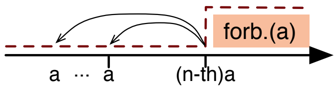
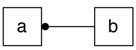
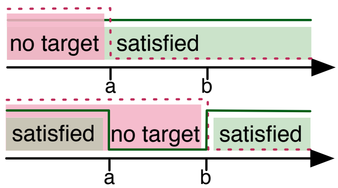
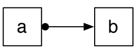
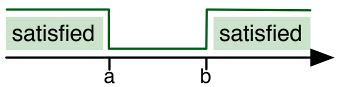
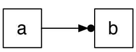
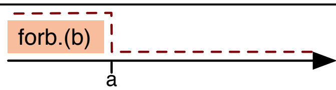
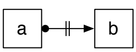
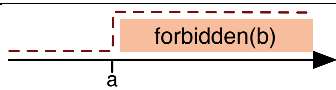
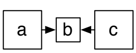
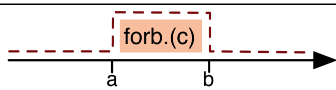
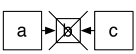
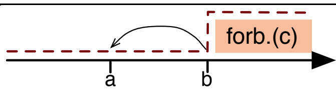
$$initially(satisfied(C)) \leftarrow core\_constraint(C, response(A, B)).$$
$$terminates(A, satisfied(C), \_) \leftarrow core\_constraint(C, response(A, B)).$$
$$initiates(B, satisfied(C), \_) \leftarrow core\_constraint(C, response(A, B)).$$

The formalization of `existence` (Tab. 5(1)) and `absence` (Tab. 5(2)) constraints is straightforward: the first constraint is satisfied when the n-th occurrence of `a` is executed, whereas the second one forbids further executions of `a` when its n-th occurrence happens. To obtain the time at which the n-th occurrence of activity `a` happens, we use a conjunction of n happened events involving `a`; then, we order such happened events by means of temporal constraints. The last happened event provides the desired time.

`responded existence` (Tab. 5(3)) is more complex to deal with, mainly due to the fact that it does not impose any ordering, whereas $\mathcal{EC}$, which considers the effects of events, reasons "forwards." To capture its semantics, we differentiate between two cases: the one in which `b` happens before any occurrence of `a`, and

[4] Note that the parameters of `core_constraint/2` have the same meaning of the parameters of `constraint/2`.

Table 5. A fluent-based formalization of DecSerFlow core constraints (f is used as constraint identifier); the last two constraints express the concepts of positive and negative interposition

constraint	intuition	formalization
		(1) $initiates(a, satisfied(f), T_n) \leftarrow \bigwedge_{i=1,\, T_0=0}^{n} (happens(a, T_i) \wedge T_i > T_{i-1}).$
		(2) $initiates(a, forbidden(a, f), T_n) \leftarrow \bigwedge_{i=1,\, T_0=0}^{n} (happens(a, T_i) \wedge T_i > T_{i-1}).$
		(3) $initially(no\_target(f)).$ $terminates(b, no\_target(f), \_).$ $initially(satisfied(f)).$ $terminates(a, satisfied(f), T) \leftarrow holdsat(no\_target(f), T).$ $initiates(b, satisfied(f), \_).$
		(4) $initially(satisfied(f)).$ $terminates(a, satisfied(f), \_).$ $initiates(b, satisfied(f), \_).$
		(5) $initially(forbidden(b, f)).$ $terminates(a, forbidden(b, f), \_).$
		(6) $initiates(a, forbidden(b, f), \_).$
		(7) $initiates(a, forbidden(c, f), \_).$ $terminates(b, forbidden(c, f), \_).$
		(8) $initiates(b, forbidden(c, f), T) \leftarrow happens(a, T_a) \wedge T_a < T.$

the reverse. In the first case, the constraint is always satisfied: when `a` happens, `b` is already present in the execution trace, thus no further expectation is triggered. In the second case, instead, the occurrence of `a` switches the constraint to an unsatisfied state, waiting for activity `b` to be executed (as in the case of `response`, Tab. 5(4)). Since the happening of `a` concretely affects the status of the *satisfied* fluent only if no `b` was previously performed, we have to explicitly track the happening of `b` with another fluent (called *no_target* in Table 5).

`precedence` (Tab. 5(5)) is captured by observing that the backward constraint "`b` must be preceded by `a`" can be rephrased in a forward manner as "`a` enables the possibility of executing b". We formalize this by imposing that the constraint causes b to be initially forbidden, until the first execution of activity a happens.

The formalization of `negation response` (Tab. 5(6)) is straightforward: the happening of the source activity a causes b to be forbidden.

The `interposition` constraint (Tab. 5(7)) is captured by rephrasing "if c is performed after a, then at least one instance of activity b must be executed in between" as "when a is executed, c is forbidden until b is executed". Similarly, `negative interposition` (Tab. 5(8)) can be formalized by stating that when activity b is performed after a, then c becomes forbidden: its execution would lead to violate the constraint.

The proposed formalization can be easily adapted to deal also with branching constraints, which are interpreted in DecSerFlow in a disjunctive manner. For example, let us consider a `response` constraint, having both branching sources a and b and branching targets c and d. It is interpreted as follows: "when either a or b are executed, then c or d must be executed afterwards". To model such a behavior, we extend the way constraints are represented by considering lists of activities instead of individual activities (e.g., the above described branching `response` can be modeled as $formula(c_1, response([a,b],[c,d])))$. We then adapt the formalization shown in Table 5, using the built-in Prolog predicate `member/2`[5] to specify that each source (target resp.) activity is able to terminate (initiate resp.) the corresponding *satisfied* fluent:

$$\begin{aligned}
initially(satisfied(C)) &\leftarrow core_constraint(C, response(As, Bs)).\\
terminates(A, satisfied(C), _) &\leftarrow core_constraint(C, response(As, Bs))\\
&\quad \wedge member(A, As).\\
initiates(B, satisfied(C), _) &\leftarrow core_constraint(C, response(As, Bs))\\
&\quad \wedge member(B, Bs).
\end{aligned}$$

3.3 Characterizing Compliant Executions

To effectively perform compliance verification of a service composition w.r.t. a DecSerFlow model, we finally have to define a suitable semantics for the *satisfied* and *forbidden* fluents, reflecting their intuitive meaning. More specifically, a correct execution must fulfill the following requirements:

- all constraints which involve a "positive" relation must eventually converge to a fulfilled state. This means that the satisfied fluent corresponding to the positive relation holds from a given point on and it is never declipped thereafter. We denote the set of "positive" constraints by $\mathcal{C}_{SAT}$. Since the "positive" behavior is formalized by means of a *satisfied* fluent, such a requirement can be expressed as a goal imposing that, for all contraints in $\mathcal{C}_{SAT}$, the corresponding *satisfied* fluent must hold when the interaction is completed. We model the completion of interaction as a special, last `complete` event,

[5] `member(El, L)` is true if `El` belongs to the list `L`.

happening at a time T_∞ (s.t. no further event will happen after T_∞). Thus, we have a goal:

$$\bigwedge_{\{c|c \in \mathcal{C}_{SAT}\}} holdsat(satisfied(c), T_\infty). \tag{1}$$

– the semantics of *forbidden* fluents is given as a *denial*, stating that if a certain activity A happens when it is forbidden by some negative constraint, then the execution is unsuccessful:

$$happens(A,T) \wedge holdsat(forbidden(\_, A), T) \rightarrow \bot.dov \tag{2}$$

In order to be compliant, services must eventually satisfy all the positive relations without undermining the negative ones.

4 Verification of Quantitative Time Constraints

We now discuss how it can be extended to model and verify quantitative temporal constraints, which are an important aspect when monitoring service interaction. In the context of DecSerFlow, temporal constraints can be used to extend positive relations with the concepts of delays and deadlines, i.e. minimum/maximum time intervals that should be respected between the execution of two activities[6].

To specify that "when an order is paid, a receipt must be delivered within 24 time units" the modeler may use a `response` constraint c_1, adding the information that c_1 cannot persist in a non-satisfied state for more than 24 time units. We suppose that, to describe this condition, the user simply uses a $deadline(satisfied(c_1), 24)$ declaration. In general, $deadline(F, D)$ states that fluent F can persist in a "not-holding" state at most D time units.

To capture and verify deadlines, we then add four new axioms. Let us suppose that fluent F is associated to a $deadline(F, D)$ condition. When F is terminated, a new fluent $d\_check(F, T_e)$ is initiated. This fluent represents that F is currently monitored, to check if the associated deadline will be met by the execution; T_e denotes the time at which the deadline will expire. Such a situation can be formalized by means of the following axiom:

$$\begin{aligned} initiates(A, d\_check(F, T_e), T) \leftarrow\ & deadline(F, D), terminates(A, F, T), \\ & T_e == T + D. \end{aligned} \tag{3}$$

The fluent $d\_check(F, T_e)$ can be terminated in two cases. In the first case, an event capable to terminate F happens within the deadline (i.e., within T_e):

$$terminates(A, d\_check(F, T_e), T) \leftarrow deadline(F, \_), initiates(A, F, T), T < T_e. \tag{4}$$

[6] In the following, we will focus only on deadlines; delays can be handled in a similar way.

The second case deals with the expiration of the deadline. $\mathcal{S}$CIFF has no notion of the flow of time: it becomes aware of the current time only when a new event occurs. Therefore, we can keep $\mathcal{S}$CIFF up-to-date by generating special `tick` events. The deadline expiration is then detected and handled as soon as the first `tick` event after the deadline occurs:

$$terminates(tick, deadline\_check(F, T_e), T) \leftarrow \; deadline(F, \_), T \geq T_e. \tag{5}$$

A further axiom recognizes this abnormal situation, by evaluating whether the *deadline_check* has been terminated after the expiration time (and generating a violation if it is the case):

$$happens(tick, T) \wedge holdsat(deadline\_check(F, T_e), T) \wedge T \geq T_e \rightarrow \bot. \tag{6}$$

5 Extending the Calculus

In this section we show how violations can be captured and reified within the calculus itself. On the one hand, capturing violations prevents the termination of the proof procedure when an error is detected. On the other hand, reifying violations enable the possibility to consider them as first-class object during the modeling phase, supporting the possibility of specifying and verifying complex requirements such as compensating activities.

5.1 Reification of Violations

As described in Sections 3.3 and 4, two different kinds of non-compliance can be identified at run-time: violation of a negative constraint, by executing a forbidden activity, or violation of a positive constraint, if it is not satisfied when the execution terminates or, if a deadline is present, within the required expiration time.

In its basic form, $\mathcal{S}$CIFF reacts to violations by terminating with answer "no": the observed happened events are evaluated as non compliant with the choreography. This is undesirable in a monitoring setting: we would like to continue the verification task even if some constraint has been violated.

To prevent termination of the proof, the underlying idea is to *reify* violations as occurrences of special events. In other words, we explicitly capture the possible run-time violations of a fluent F by generating a corresponding $violation(F)$ event upon violation of F. If we want to capture and handle violations, then we must remove axioms (1), (2) and (4), and substitute them with a corresponding "soft" version. In particular, a soft version of axiom (1) states that, for each constraint $C \in \mathcal{C}_{SAT}$, if the corresponding $satisfied$ fluent does not hold at T_∞, then a corresponding $violation(satisfied(C))$ event must be generated:

$$\begin{aligned} &happens(complete, T_\infty) \wedge \\ &not\_holdsat(satisfied(C), T_\infty) \rightarrow happens(violation(satisfied(C)), T_\infty). \end{aligned} \tag{7}$$

The same applies for axiom (4) (dealing with the deadline expiration), which becomes

$$\begin{aligned} &happens(tick, T) \wedge \\ &holdsat(deadline\_check(F, T_e), T) \wedge T \geq T_e \rightarrow happens(violation(F), T). \end{aligned} \tag{8}$$

A soft version of axiom (2) is the following axiom:

$$\begin{aligned} &happens(A, T) \wedge \\ &holdsat(forbidden(C, A), T) \rightarrow happens(violation(forbidden(C)), T). \end{aligned} \tag{9}$$

Reifying violations opens many possibilities. For example, we could associate an "importance degree" to each constraints, identifying and handling different levels of violation. In the next section we will briefly focus on another possibility, namely the specification of how to compensate for a violation.

5.2 Dealing with Compensations

Among the many possibilities offered by the reification of violations, an interesting option is to attach DecSerFlow constraints to such a generated event. This could be a way to specify how the interacting services must *compensate* for a violation, or to define a context for violations, i.e. to model constraints which become *soft* only in certain situations in the choreography.

Compensation can be modeled by e.g. inserting a `response` constraint having a violation event as source, and the compensation activity as target; `chain response` could be then used to handle critical violations: it states that when the violation is detected, the next immediate activity to be executed is the compensating one.

Contextualization of violations can be modeled using backward DecSerFlow constraints (e.g., `precedence`). For example, modeling a `precedence` constraint involving an activity A and the event $violation(C)$ states that as soon as the event violation(C) is raised, the REC verify if previously an execution of the activity A has been performed (the activity A representing some how the idea of context). In such a case, the violation can be managed, otherwise a definitive, non compliant response is provided as a result.

6 Monitoring Example

We now briefly discuss a simple yet significative example of a choreography fragment, showing how the proposed approach can be fruitfully applied for run-time monitoring. Figure 2 shows the graphical DecSerFlow representation of the example, while Table 6 sketches its corresponding formalization.

The choreography involves a customer, who creates an order by choosing one or more items, and a seller, who collects the ordered items and finally gives a

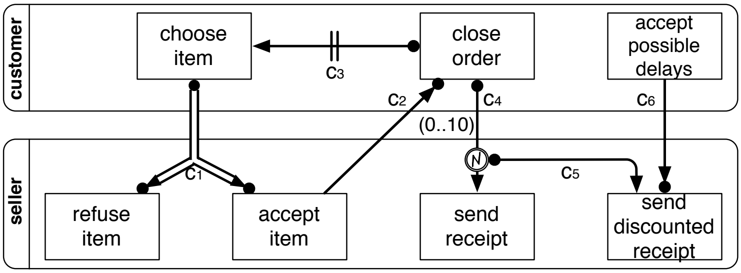


Fig. 2. A DecSerFlow choreography fragment, extended with a deadline and a compensation

Table 6. Formalization of the choreography fragment shown in Figure 2

ID	$\mathcal{REC}$ **Specification**
c_1	$formula(c_1, alternate\_succession([choose\_item], [refuse\_item, accept\_item]))$.
c_2	$formula(c_2, precedence([accept\_item], [close\_order]))$.
c_3	$formula(c_3, negation\_response([close\_order], [choose\_item]))$.
c_4	$formula(c_4, response([close\_order], [send\_receipt]))$.
	$deadline(satisfied(c_4), 10)$.
c_5	$formula(c_5, response([violation(c_4)], [send\_discounted\_receipt]))$.
c_6	$formula(c_6, precedence([accept\_possible\_delays], [send\_discounted\_receipt]))$.

receipt. The seller is committed to issue the final receipt within a pre-established deadline. Moreover, the seller offers the customer a fixed discount if he/she accepts some delays; in case of a delay, the seller also promises a further discount directly on the receipt.

In particular, the following rules of engagement must be fulfilled by the interacting services. It is worth noting that each constraint can be easily mapped by means of an (extended) DecSerFlow relation.

- Every `choose item` activity must be followed by an answer from the seller, either positive or negative; no further upload can be executed until the response is sent. Conversely, each positive/negative response must be preceded by a `choose item` activity, and no further response can be sent until a new item is chosen (constraint c_1).
- If at least one uploaded item has been accepted by the seller, then it is possible for the customer to close the order (constraint c_2).
- When an order has been closed, no further item can be choosen (constraint c_3); moreover, the seller is committed to send a corresponding receipt by at most 10 time units (constraint c_4).
- If the seller does not meet the deadline, it must deliver a discounted receipt (constraint c_5, modeled as a `response` constraint triggered by the violation of constraint c_4; the graphical representation of the violation is inspired by the BPMN *intermediate error* event).

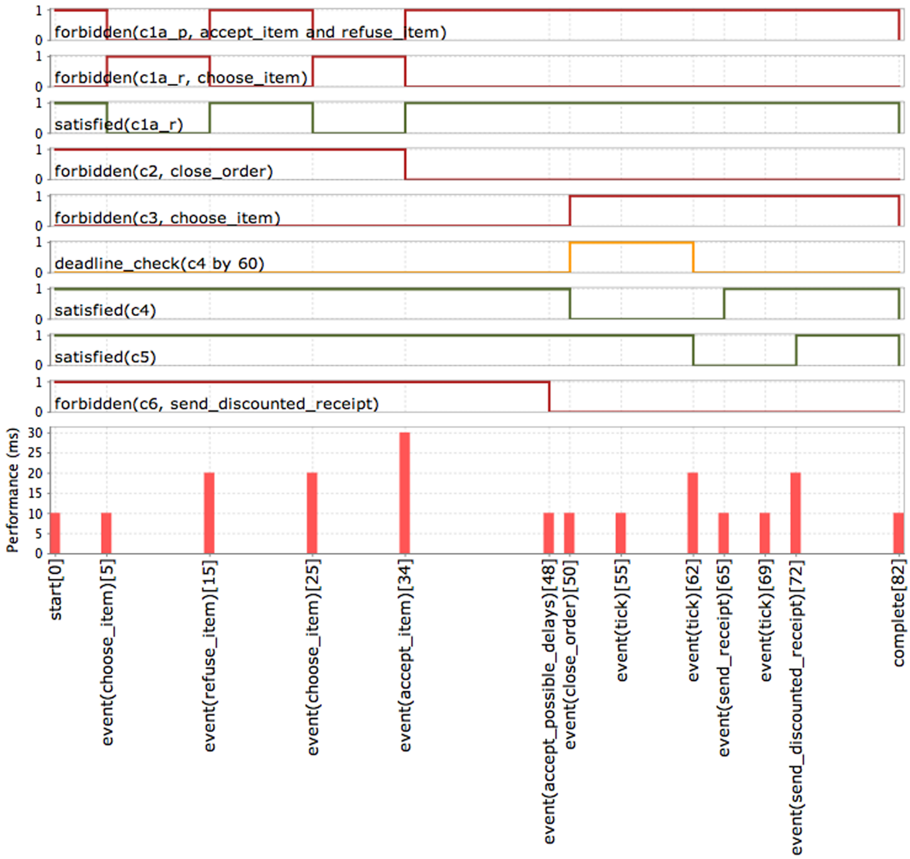


Fig. 3. Fluents trend generated by $\mathcal{REC}$ when monitoring a specific interaction, and using the diagram of Figure 2 as model. The verification time spent for reacting to each happened event is also reported.

– The possibility of sending a discounted receipt is enabled only if the customer has previously accepted the possibility of experiencing delays (constraint c_6).

Note that the obtained DecSerFlow diagram contains two constraints (c_4 and c_5) which are not envisaged by standard DecSerFlow, but are seamlessly supported by $\mathcal{REC}$ thanks to the extensions presented above.

Figure 3 illustrates how $\mathcal{REC}$ is able to reason upon a specific course of interaction w.r.t. the above described DecSerFlow model. Clipping and declipping of fluents are handled at run-time, thus giving a constantly updated snapshot of the reached interaction status. In the bottom part of the figure, verification performance is reported, showing the amount of time spent by $\mathcal{REC}$ in order to dynamically react to and reason upon occurring events.

The central part of the execution shows how $\mathcal{REC}$ deals with a deadline expiration. Indeed, as soon as the activity `close_order` is executed (at time 50), constraint c_4 becomes unsatisfied, and a corresponding deadline check is initiated, having 60 as expiration time. At time 62, a `tick` event makes the proof aware that the deadline related to the satisfaction of constraint $c4$ is expired. As a consequence, $\mathcal{S}$CIFF reacts by terminating the *deadline_check* fluent and by installing the corresponding compensation; this is attested by the fact that constraint c_5 becomes unsatisfied.

7 Related Work

Event Calculus has been extensively applied to specify and verify event-based systems in many different settings. We will restrict our attention to the applications related to the SOA research field.

Rouached et al. propose a framework for engineering and verifying WS-BPEL processes is [16]. $\mathcal{EC}$ is used to provide an underlying semantics to WS-BPEL, enabling verification before and after execution. In particular, $\mathcal{EC}$ is exploited to verify consistency and safety of a service composition (i.e. to statically check if the specification always guarantees the desired requirements), and to check whether an already completed execution has deviated from the prescribed requirements. The authors rely on an inductive theorem prover for the verification task. Although our work adopts DecSerFlow as specification language, the mapping of WS-BPEL presented in [16] can be directly implemented on top of $\mathcal{REC}$. In [17], Aydın and colleagues use the Abductive Event Calculus to synthesize a web service composition starting from a goal. The composition process is a planning problem, where the functionality provided by individual services are (atomic) actions, requiring some inputs and producing certain outputs. Being $\mathcal{REC}$ based on an abductive proof-procedure, we will investigate the possibility of adopting $\mathcal{REC}$ to deal also with this issue.

Few authors have considered adopting the $\mathcal{EC}$ to perform run-time reasoning. Among those who have, Mahbub and Spanoudakis present a framework [18] for monitoring the compliance of a WS-BPEL service composition w.r.t. behavioral properties automatically extracted from the composition process, or assumptions/requirements expressed by the user. $\mathcal{EC}$ is exploited to monitor the actual behavior of interacting services and report different kinds of violation. The approach is extended in [19], where an extension of WS-Agreement is used to specify requirements. The monitoring framework relies on an ad hoc event processing algorithm, which fetches occurred events updating the status of involved fluents.

8 Conclusion

In this article we have presented a method for run-time verification of choreographies specified in DecSerFlow that makes use of a SCIFF implementation of the Event Calculus. The main features of our method are the presence of an *execution model*, which enables an efficient monitoring of the evolution of fluents and their verification; the coherence of an overall *declarative* framework, in which no information is lost when passing from DecSerFlow to SCIFF; and the *flexibility* of the language, which makes it possible to capture aspects of complex requirements, such as qualitative temporal conditions and violation handling by compensation, in a simple and intuitive way. We have chosen to start from DecSerFlow partly because it is well suited for representing the contractual nature of service choreographies, and to specify the desired and undesired courses of interaction while leaving undefined other possibilities of interaction that are neither desired nor undesired. We believe that this is a promising approach and in the future we plan to focus on other declarative and contractual aspects of choreographies. In particular, we intend to study the role of social commitments [20] in the choreographies and to investigate possible integrations of commitments into our framework.

Acknowledgments. This work has been partially supported by the FIRB project *TOCAI.IT*. The authors would like to thank the anonymous reviewers for their helpful comments.

References

1. Kavantzas, N., Burdett, D., Ritzinger, G., Fletcher, T., Lafon, Y.: Web services choreography description language. W3C Working Draft 17-12-04 (2004), `http://www.w3.org/TR/ws-cdl-10/`
2. Baldoni, M., Baroglio, C., Martelli, A., Patti, V.: A priori conformance verification for guaranteeing interoperability in open environments. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 339–351. Springer, Heidelberg (2006)
3. Li, J., Zhu, H., Pu, G.: Conformance validation between choreography and orchestration. In: First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, TASE 2007, Shanghai, China, pp. 473–482. IEEE Computer Society Press, Los Alamitos (2007)
4. Zaha, J., Dumas, M., Hofstede, A., Barros, A., Dekker, G.: Service Interaction Modeling: Bridging Global and Local Views. QUT ePrints 4032, Faculty of Information Technology, Queensland University of Technology (2006)
5. Decker, G., Weske, M.: Local enforceability in interaction petri nets. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 305–319. Springer, Heidelberg (2007)
6. Allen Emerson, E., Halpern, J.: "Sometimes" and "Not Never" revisited: On branching times versus linear time. Journal of the ACM 33, 151–178 (1986)
7. Alur, R., Henzinger, T.A.: Real-time logics: complexity and expressiveness. Information and Computation 104, 35–77 (1993)
8. Wang, F.: Formal verification of timed systems: A survey and perspective. Proceedings of the IEEE 92(8), 1283–1305 (2004)
9. van der Aalst, W.M.P., Pesic, M.: DecSerFlow: Towards a truly declarative service flow language. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 1–23. Springer, Heidelberg (2006)
10. Kowalski, R.A., Sergot, M.: A logic-based calculus of events. New Generation Computing 4(1), 67–95 (1986)

11. Chesani, F., Montali, M., Mello, P., Torroni, P.: An efficient SCIFF implementation of Reactive Event Calculus. Technical Report LIA-08-003, University of Bologna, Italy. LIA Series No. 89 (May 2008)
12. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verifiable agent interaction in abductive logic programming: the SCIFF framework. ACM Transactions on Computational Logic 9(4) (to appear, 2008)
13. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Compliance verification of agent interaction: a logic-based software tool. Applied Artificial Intelligence 20(2-4), 133–157 (2006)
14. Shanahan, M.: The event calculus explained. In: Veloso, M.M., Wooldridge, M.J. (eds.) Artificial Intelligence Today. LNCS, vol. 1600, pp. 409–430. Springer, Heidelberg (1999)
15. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Montali, M., Storari, S., Torroni, P.: Computational logic for run-time verification of web services choreographies: Exploiting the *socs-si* tool. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 58–72. Springer, Heidelberg (2006)
16. Rouached, M., Fdhila, W., Godart, C.: A semantic framework to engineering ws-bpel processes. International Journal on Information Systems and e-business Management (2008)
17. Aydin, O., Cicekli, N.K., Cicekli, I.: Automated web services composition with event calculus. In: Proceedings of the 8th International Workshop in Engineering Societies in the Agents World (ESAW 2007) (2007)
18. Mahbub, K., Spanoudakis, G.: Run-time monitoring of requirements for systems composed of web-services: Initial implementation and evaluation experience. In: 2005 IEEE International Conference on Web Services (ICWS 2005), Orlando, FL, USA, pp. 257–265. IEEE Computer Society Press, Los Alamitos (2005)
19. Mahbub, K., Spanoudakis, G.: Monitoring ws-agreements: An event calculus-based approach. In: Baresi, L., Nitto, E.D. (eds.) Test and Analysis of Web Services, pp. 265–306. Springer, Heidelberg (2007)
20. Yolum, P., Singh, M.: Flexible protocol specification and execution: applying event calculus planning using commitments. In: The First International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2002, Bologna, Italy, Proceedings, pp. 527–534 (2002)