

Declarative Specification and Verification of Service Choreographies

MARCO MONTALI, MAJA PESIC, WIL M. P. VAN DER AALST, FEDERICO CHESANI,
PAOLA MELLO and SERGIO STORARI

Service oriented computing, an emerging paradigm for architecting and implementing business collaborations within and across organizational boundaries, is currently of interest to both software vendors and scientists. While the technologies for implementing and interconnecting basic services are reaching a good level of maturity, modeling service interaction from a global viewpoint, i.e., representing service choreographies, is still an open challenge. The main problem is that, although declarativeness has been identified as a key feature, several proposed approaches specify choreographies by focusing on procedural aspects, leading to over-constrained and over-specified models.

To overcome these limits, we propose to adopt DecSerFlow, a truly declarative language, to model choreographies. Thanks to its declarative nature, DecSerFlow semantics can be given in terms of logic-based languages. In particular, we present how DecSerFlow can be mapped onto *Linear Temporal Logic* and onto *Abductive Logic Programming*. We show how the mappings onto both formalisms can be concretely exploited to address the enactment of DecSerFlow models, to enrich its expressiveness and to perform a variety of different verification tasks. We illustrate the advantages of using a declarative language in conjunction with logic-based semantics by applying our approach to a running example.

Categories and Subject Descriptors: D.1.7 [**Programming Techniques**]: Visual Programming; H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services*; I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods - Temporal Logic, Rule-based Representations; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*Logic programming, Inference Engines*

General Terms: Languages, Management, Verification

Additional Key Words and Phrases: Service Choreographies, Declarative approaches, Linear Temporal Logic, Abductive Logic Programming, Conformance Checking, Interoperability, Reasoning

Authors' present address:

Wil M. P. van der Aalst - Department of Mathematics and Computer Science, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands, e-mail: w.m.p.v.d.aalst@tue.nl.

Federico Chesani, Marco Montali and Paola Mello - DEIS, University of Bologna, V.le Risorgimento 2, 40136 Bologna (BO), Italy, e-mail: {marco.montali, federico.chesani, paola.mello}@unibo.it.

Maja Pesic - Department of Technology Management, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands, e-mail: m.pesic@tm.tue.nl.

Sergio Storari - Dept. of Engineering, University of Ferrara, Via Saragat 1, 44100 Ferrara (FE), Italy, email: strsrsg@unife.it.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 2009 ACM 0000-0000/2009/0000-0001 \$5.00

1. INTRODUCTION

Service oriented computing, an emerging paradigm for architecting and implementing business collaborations within and across organizational boundaries, is currently of interest to both software vendors and scientists [van der Aalst et al. 2003]. In its web implementation, the functionality provided by business applications is encapsulated within web services: software components described at a semantic level, which can be invoked by application programs or by other services through a stack of Internet standards including HTTP, XML, SOAP [Box et al. 2000], WSDL [Christensen et al. 2001] and UDDI [Belwood et al. 2000]. Once deployed, web services provided by various organizations can be inter-connected in order to implement business collaborations, leading to *composite web services* where participating services interact in a choreography.

Let us for example consider a B2B setting, in which different organizations share their own services to mutually benefit from each other, trying to reach complex strategic goals, impossible to be pursued autonomously. In this context, it is often impossible to make the assumption that one of the involved organizations will take the lead during the interaction, acting as an orchestrator. As clearly pointed out in the WS-CDL 1.0 specification [Kavantzas et al. 2004] *“in real-world scenarios, corporate entities are often unwilling to delegate control of their business processes to their integration partners. Choreography offers a means by which the rules of participation within a collaboration can be clearly defined and agreed to, jointly. Each entity may then implement its portion of the Choreography as determined by the common or global view.”*

In a B2B setting, the birth of a service choreography is often determined by putting together external norms/regulations and internal policies, requirements, best practises, business goals of each participating organization. All these different contributions have the effect of constraining the possible allowed interactions, and they will therefore be referred to as *constraints* throughout the paper. The obtained global model should suitably mediate between *compliance* and *flexibility*: on the one hand, all interacting services must respect the agreed constraints; on the other hand, each party should be able to execute the business processes which cover its part of the choreography as free as possible, preserving interoperability and replaceability of services. In other words, we claim that a service choreography should play the role of a public global contract which focuses on the rules of engagement required to make all the interacting parties collaborate correctly, without stating how such a collaboration is concretely carried out. This kind of knowledge is inherently *declarative*.

As pointed out in [Barros et al. 2005; van der Aalst et al. 2005], while the technologies for implementing and interconnecting basic services are reaching a good level of maturity, modeling service interaction from a global viewpoint, i.e., representing service choreographies, is still an open challenge: the leading current proposals for modeling service interaction, such as WS-BPEL [Andrews et al. 2003] and WS-CDL [Kavantzas et al. 2004], fail to tackle a suitable balance between compliance and flexibility. The main problem is that, although declarativeness has been identified as a key feature, current mainstream approaches propose languages and methodologies which model choreographies by focusing on procedural aspects,

e.g. by specifying control and message flow of the interacting services. This leads to loose the declarative nature of the knowledge involved in the choreography definition, forcing the modeler to capture it at a procedural level.

To overcome these limits, we propose a framework for dealing with service choreographies at the declarative level. In particular, we adopt DecSerFlow [van der Aalst and Pesic 2006] as a truly declarative language for the graphical specification of service flows, and present a mapping from the DecSerFlow graphical constructs to two underlying logic-based languages, enabling the possibility of reasoning upon the developed models. DecSerFlow adopts a more general and high-level view of services specification, by directly defining them through a set of policies or business rules referred to as *constraints*. Hence, it does not give a complete and procedural specification of what is allowed in services, but concentrates on what is the (minimal) set of constraints to be fulfilled in order to successfully accomplish the interaction (i.e., what is forbidden and mandatory in services).

It is the declarative nature of DecSerFlow which opens the possibility of providing suitable underlying semantics in terms of logic-based languages. In particular, we present how DecSerFlow can be mapped onto Linear Temporal Logic (LTL) [Clarke et al. 1999] and onto the SCIFF framework [Alberti et al. 2008]. The LTL mapping of DecSerFlow currently focuses only on the process perspective of services (i.e., on activities executed in services), while SCIFF is able to consider activities, data elements and time. We discuss how the mappings onto both formalisms can be concretely exploited to address the enactment of DecSerFlow models, to enrich its expressiveness and to perform a variety of different verification tasks, as shown in Table I.

	LTL	SCIFF
enactment	X	
conformance checking	X	X
interoperability	X	X
conflicts and dead activities detection	X	X
mining		X
support of activities-data and quantitative time constraints		X

Table I. DecSerFlow verifications and extensions-support through LTL and SCIFF.

LTL is a special type of logic that, in addition to classical logical operators, uses several temporal operators. Mapping to LTL enables DecSerFlow to exploit automata generated from LTL expressions [Gerth et al. 1996; Giannakopoulou and Havelund 2001] for execution of individual services and verification of participating services and whole compositions. The LTL representation of DecSerFlow models also enables a posteriori verification of properties and checking of service interaction (i.e. conformance checking) in the LTL Checker [van der Aalst et al. 2005] plug-in of the process mining ProM framework [van der Aalst et al. 2007].

SCIFF is a framework based on Abductive Logic Programming (ALP) [Kakas et al. 1993], originally developed within the SOCS EU Project¹ for the specification

¹Societies 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:

and verification of global interaction protocols in open Multi-Agent Systems (MAS), which share many aspects with the Service-oriented Computing setting [Baldoni et al. 2005a]. Similarly to the case of service choreographies and DecSerFlow, the need for modeling global interaction protocols by respecting the autonomy and heterogeneity of interacting agents has motivated the shift from mentalistic approaches to declarative and social-based ones [Singh 2000]. The *SCIFF* framework belongs to the latter family: it envisages a powerful logic-based language for specifying social interaction, and is equipped with a proof procedure capable to check at run-time or a posteriori whether a set of interacting entities is behaving in a conformant manner w.r.t. a given specification. Thanks to the mapping from DecSerFlow to *SCIFF* proposed in this work, we achieve two complementary advantages. On the one hand, the mapping extends the applicability of *SCIFF* outside of the MAS setting, opening the possibility of exploiting its verification capabilities in the SOC context and by non-expert users: they have not to deal directly with the complexity of the *SCIFF* syntax, but can instead work at the intuitive graphical level of DecSerFlow, automatically obtaining the corresponding *SCIFF* specification. On the other hand, DecSerFlow can benefit of the expressiveness and verification capabilities of *SCIFF*, addressing conformance checking and static verification of DecSerFlow choreographies, enabling mining of DecSerFlow models from service execution traces, and enriching the language with data-related aspects and quantitative time constraints. Even if the main focus of a choreography is on the involved activities and their flow dependencies, adding data and quantitative time-related aspects enables to model a wider range of situations, such as desired deadlines, constraints which span over multiple choreography instances, content-based decisions points, interactions in which multiple concrete services play the same role (e.g., bidders in an auction). The possibility of addressing such kind of specifications does not depend on DecSerFlow (which can be extended for the purpose), but is instead affected by the underlying chosen formalization.

We illustrate the advantages of using a declarative language in conjunction with logic-based semantics by applying our approach to a motivating choreography example.

The remainder of this paper is organized as follows. Section 2 motivates why the challenging issue of modeling service choreographies should be faced by adopting a declarative approach. Section 3 describes the DecSerFlow language together with its mapping onto LTL. The *SCIFF* language is presented in Section 4, and in section 5 the mapping of DecSerFlow concepts onto *SCIFF* is shown. Section 6 describes then how LTL and *SCIFF* can be used for enactment and various verification tasks of DecSerFlow models. A discussion, focused on the usability of the whole framework for what concerns features of the language as well as performances and scalability of the verification techniques and current available tools, follows in Section 7. Related work is presented in Section 8, while Section 9 concludes the paper sketching ongoing and future works. To make this article as concise as possible, the complete description of all the core DecSerFlow constraints and of the corresponding mapping onto LTL and *SCIFF* is described in Appendix A.

2. MOTIVATION

To illustrate the difficulty of handling even simple choreography constraints with classical procedural approaches, let us consider a fragment of a purchase choreography, regulating the seller’s decision about the confirmation or rejection of an order. The seller could freely decide whether to confirm or refuse customer’s order, but must obey to the following constraints:

- if the warehouse cannot ship the order, then the seller must refuse it;
- the seller can accept the order only if the warehouse has previously accepted its shipment;
- both the seller and the warehouse cannot accept and reject the same order, i.e., answers are mutually exclusive.

By considering these global rules, many different compliant interactions can be established by a concrete seller and a concrete warehouse. For example, when and how the warehouse is contacted is not specified, and there could be different choreography executions in which the warehouse is not contacted at all: an execution in which the seller autonomously decides to reject the order, without asking warehouse’s opinion, is foreseen by the choreography. This execution trace clearly attests that many different compliant ways to interact are not explicitly mentioned in the choreography, but are instead implicitly supported. We argue that this is due to the fact that choreography rules constitute a form of declarative knowledge, which states what is forbidden and mandatory in services without giving details about how to carry out the interaction.

When the user tries to model this kind of knowledge using a classical procedural specification language such as WS-BPEL or WSCDL, she is forced to explicitly enumerate all the implicitly supported executions, and to introduce further unnecessary details. Consider for example the BPMN [White 2006] collaborative diagrams as a modeling language to capture the above described choreography fragment.

Figure 1 compares the adoption of BPMN collaborative diagrams versus the use of a declarative constraint-based language such as DecSerFlow when modeling the choreography fragment described above. While DecSerFlow (Figure 1(a)) is able to capture the choreography in a compact and easily understandable way, BPMN (Figures 1(b) and 1(c)) experiences difficulties when trying to suitably mediate between compliance and flexibility: unnecessary activities are introduced (such as the “contact warehouse” activity) and some acceptable execution traces are not supported. For example, both the BPMN diagrams shown in Figures 1(b) and 1(c) do not support the possibility that the warehouse refuses the shipment after the refusal of the seller; even if the refusal of the warehouse seems to be, in this case, insignificant, it could be involved in other constraints of the choreography, and should therefore be supported. Adding this behaviour would require to complicate the model, replicating execution paths and activities, and introducing ambiguous decision points. And obviously, this issue would be even more hard to handle when the modeled fragment has to be composed with other constraints to capture the whole choreography.

These difficulties arise when the modeler tries to interconnect the choreography activities by means of control and message flows. In particular, a non-exhaustive

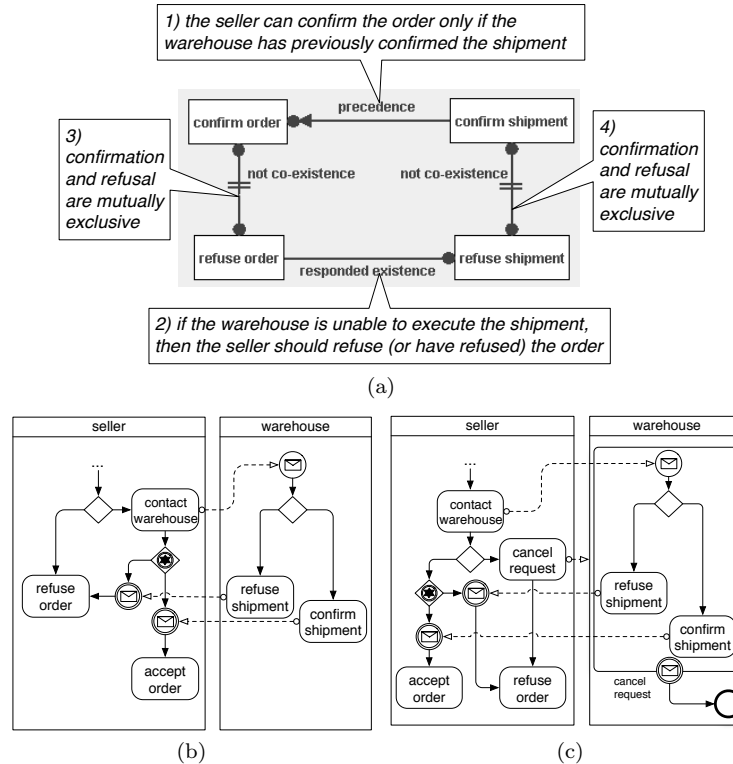


Fig. 1. Declarative vs. procedural style of modeling a simple choreography.

list of issues for which procedural languages do not provide a suitable support are: How to deal with negative information such as “the seller cannot accept and reject the same order”? How to deal with non-ordered constraints, such as the one stating that “if the warehouse refuses the order, then the seller must also refuse (or have refused) it”? Who is in charge to contact the warehouse? And when?

The difficulty of providing an answer to these question by adopting a procedural style of modeling is threefold:

- lack of proper abstractions.* Activities can be inter-connected only by means of positive temporally-ordered relationships (sequence patterns, mixed with constructs aimed at splitting/merging the control or the message flow). Modeling other kind of constraints forces the user to complicate the model. For example, capturing temporally-unordered relationships leads either to choose one ordering and impose it in the model, compromising flexibility, or to explicitly capture all the possible orderings, introducing ambiguous decision points to combine them.
- closed nature.* Procedural models makes the implicit assumption that “all that is not explicitly modeled is forbidden”, and must therefore enumerate all the allowed executions. Therefore, when a negative requirement (such as forbidding a certain activity or stating that two activities must never co-exist in the same execution) must be considered, it is not possible to make it explicit in the model; instead,

it is responsibility of the user to check whether the produced model implicitly entails the negative requirement or not. This is a difficult task, especially when the complexity of the model increases.

- premature commitment*. Since procedural approaches have a close nature and do not provide proper abstractions, they force the modeler to prematurely take decisions and make assumptions about the interaction. For example, even if the considered choreography fragment does not specify how and when the warehouse must be contacted, this choice must indeed be taken during the modeling phase.

The combination of these drawbacks has the effect that choreographies become *over-specified* and *over-constrained*: unnecessary activities and constraints are introduced, and acceptable interactions are dropped out. As a consequence, while compliance is respected, flexibility becomes sacrificed: potential partners are discarded, fruitful interactions are rejected and, at last, the choreography becomes unusable. When the modeler tries to get back flexibility by relaxing the imposed constraints and reducing premature commitments, the lack of proper abstraction and the closed nature of procedural approaches lead to further stress over-specification: the resulting choreography tends to become a tangled, unintelligible spaghetti-like model, and, at the same time, the risk of supporting undesirable behaviors increases.

3. DECSEFLOW: A TRULY DECLARATIVE SERVICE FLOW LANGUAGE

Web service composition implies collaboration of independent interacting parties, i.e., services. On the one hand, composition choreography reflects a common agreement of various parties and must be applicable to various demands of interacting parties. On the other hand, interacting parties are required to follow core rules that maintain the integrity of the choreography. Figure 2(a) shows that a choreography prevents some unwanted (i.e., forbidden) scenarios, and parties can collaborate only in scenarios allowed in the choreography. Traditional modelling languages (e.g., Petri nets [Reisig and Rozenberg 1998] and WS-BPEL [Andrews et al. 2003]) are of imperative nature because they specify a scheduling procedure of activities in the flow. All possible interactions are specified in detail in such a model (as shown in Figure 2(b)) and unpredicted or exceptional interactions are not possible between the parties. Therefore, specifying service flows with an imperative language limits the number of parties that are able to fulfill the model requirements. Instead of specifying a detailed flow procedure, DecSerFlow specifies a minimal set of rules that should be followed by the interacting parties. Figure 2(c) shows that, by explicitly specifying the rules, a declarative DecSerFlow model implicitly defines the flow as all scenarios that do not violate the rules. Clearly, the more rules a DecSerFlow model has, the less possibilities there are in the flow. Because rules constrain the model, we refer to rules as to *constraints*.

DecSerFlow process models can play two roles in the context of web services:

- DecSerFlow can be used as a *global choreography model* [Zaha et al. 2006], i.e. interactions are described from the viewpoint of an external observer who oversees all interactions between all services. It is not necessary that a global model is executable, but it can be used for describing the rules of engagement for making all the interacting parties collaborate correctly, and for verification purposes (such

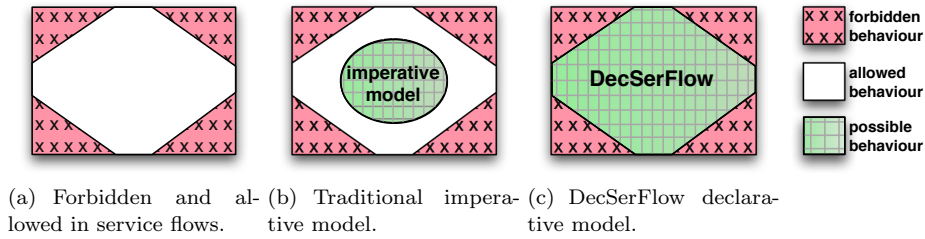


Fig. 2. DecSerFlow as a declarative language.

as conformance checking and interoperability). Here DecSerFlow is competing with languages such as the *Web Services Choreography Description Language* (WS-CDL) [Kavantzas et al. 2004].

—DecSerFlow can be used as a *local model* [Zaha et al. 2006], to specify, implement, or configure a particular service. Here DecSerFlow is competing with languages such as WS-BPEL [Andrews et al. 2003].

The remainder of this section is organized as follows. In Section 3.1 we present a running example of a Photo Service. Section 3.2 describes the building blocks of the DecSerFlow language. Section 3.3 describes the global choreography (Section 3.3.1) and a local service model (Section 3.3.2) in terms of DecSerFlow.

3.1 Running Example: Photo Shop

In this paper we use an illustrative example to describe how DecSerFlow and its underlying mappings can be suitably used for specifying and verifying choreographies. The example is concerned with a Photo Shop. Due to the high competition and booming of Internet technologies, it is common that shops for development and printing of photographs (and accompanying services) employ web services for remote placing orders. Customers (individual or other shops) can use a simple service to place orders without having to personally come to the shop. Table II shows the description of the interaction between two services: (1) Customer and (2) Photo Shop. The Customer service employs an on-line photo ordering service, while the Photo Shop service prints and delivers ordered products.

Table II. Photo Shop Example

Both the customer and the shop are responsible for executing an order and they have the following options:

Customer. The customer can enter order data, such as name, address, credit card number and preferred way of delivery, via activity “register”. Activities “photo” and “poster” can be used to order photographs and posters (respectively) by uploading files and selecting wanted formats. Customer can also order photo albums by executing activity “album”. Activities “receive” and “pay” are used when receiving and paying ordered products, respectively.

Photo Shop. The shop records order data via activity “open order”. Activity “print” is used to print ordered photos and posters. The shop delivers products and charges the customer for the service via activities “deliver” and “charge”.

Instead of following an explicitly specified order of service activities (from Table II), the two parties obey to several constraints that define the global level of service interaction (choreography), as presented in Table III:

Table III. Global Choreography Constraints

<p><i>G1.</i> The shop will not “open order” before the customer executes activity “register”. When the customer executes activity “register”, the shop will update its data via activity “open order”. This rule ensures that the shop has the right order data.</p> <p><i>G2.</i> After the customer orders photos and posters (via activities “photo” and “poster”), the shop prints ordered products via activity “print”.</p> <p><i>G3.</i> Each ordered product (“photo”, “poster” or “album”) has to be delivered via activity “deliver”. The shop will not “deliver” before at least one product is ordered.</p> <p><i>G4.</i> Customer can receive products only after the shop executes “deliver”.</p> <p><i>G5.</i> Customer can “pay” before (e.g., credit card) or after (e.g., when picking up) the shop executes its activity “charge”.</p>
--

Each of the parties can employ a local service model by their own preference, as long as these models comply with the agreed choreography, i.e., with the agreed global constraints **G1**, **G2**, **G3**, **G4** and **G5** presented in Table III. For example, the Photo Shop can implement its local process by the constraints presented in Table IV or it can even employ a procedural process model.

Table IV. Local Photo Shop Constraints

<p><i>L1.</i> Shop local service will start with activity “open order” .</p> <p><i>L2.</i> Each printed (activity “print”) item will be delivered (activity “deliver”).</p> <p><i>L3.</i> The shop will “charge” administrative fixed costs even for empty orders.</p>
--

3.2 DecSerFlow Constraint Templates

A DecSerFlow model consists of activities and constraints that represent rules to be followed while activities are executed. A constraint represents a relation between activities. For example, constraint **G3** represents a relation between activities “photo”, “poster” and “album” on one side and activity “deliver” on the other side. This type of rule is called “succession” and it specifies that some activity “A” has to be followed by some activity “B” and activity “B” cannot be executed before activity “A”. Constraint **G2** between activities “photo” and “poster” on one side and activity “print” on the other side is also a “succession”. One can imagine that one type of constraint can occur in various models between various activities. To support reusability of types of constraints, DecSerFlow language consists of a set of *constraint templates*. A constraint template represents a type of relation between activities that can be reused in various models to create constraints between activities. Each template has a unique name and consists of: (1) a Linear Temporal Logic (LTL) formula that specifies the semantics and (2) a graphical representation. LTL is a special type of logic that, in addition to classical logical operators,

uses several temporal operators: always (\square), eventually (\diamond), until (\sqcup) and next time (\circ) [Clarke et al. 1999]. When adding a constraint to a model, one works with graphical representation of the template and the underlying LTL formula remains hidden. Because of this, LTL expertise is not required for the development of DecSerFlow models. Currently, there are more than twenty DecSerFlow templates [van der Aalst and Pesic 2006], and templates can easily be added, removed or changed in DecSerFlow. Some of these templates are shown in Table V.

name	LTL expression	graphical
<i>existence</i> (A)	$\diamond(A)$	$\overset{1..*}{\boxed{A}}$
<i>existence_2</i> (A)	$\diamond(A \wedge \circ(\textit{existence}(A)))$	$\overset{2..*}{\boxed{A}}$
<i>existence_3</i> (A)	$\diamond(A \wedge \circ(\textit{existence}_2(A)))$	$\overset{3..*}{\boxed{A}}$
<i>absence_2</i> (A)	$\neg \textit{existence}_2(A)$	$\overset{0.1}{\boxed{A}}$
<i>absence_3</i> (A)	$\neg \textit{existence}_3(A)$	$\overset{0.2}{\boxed{A}}$
<i>exactly_1</i> (A)	$\textit{existence}(A) \wedge \textit{absence}_2(A)$	$\overset{1}{\boxed{A}}$
<i>exactly_2</i> (A)	$\textit{existence}_2(A) \wedge \textit{absence}_3(A)$	$\overset{2}{\boxed{A}}$
<i>response</i> (A, B)	$\square(A \Rightarrow \diamond(B))$	$\boxed{A} \xrightarrow{\bullet} \boxed{B}$
<i>precedence</i> (A, B)	$\diamond(B) \Rightarrow ((\neg B) \sqcup A)$	$\boxed{A} \xrightarrow{\bullet} \boxed{B}$
<i>succession</i> (A, B)	$\textit{response}(A, B) \wedge \textit{precedence}(A, B)$	$\boxed{A} \xrightarrow{\bullet} \bullet \boxed{B}$
<i>neg_response</i> (A, B)	$\square(A \Rightarrow \neg(\diamond(B)))$	$\boxed{A} \xrightarrow{\bullet \parallel} \boxed{B}$
<i>responded_existence</i> (A, B)	$(\diamond A) \Rightarrow (\diamond B)$	$\boxed{A} \xrightarrow{\bullet} \boxed{B}$
<i>alternate_response</i> (A, B)	$\textit{response}(A, B) \wedge \square(A \Rightarrow \circ(\textit{precedence}(B, A)))$	$\boxed{A} \xrightarrow{\bullet \rightleftarrows} \boxed{B}$
<i>chain_response</i> (A, B)	$\square(A \Rightarrow \circ(B))$	$\boxed{A} \xrightarrow{\bullet \rightleftarrows} \boxed{B}$

Table V. Some DecSerFlow templates.

Table V shows some of the DecSerFlow templates involving one or two activi-
ACM Transactions on the Web, Vol. V, No. N, May 2009.

ties. Note that it is also possible to make DecSerFlow templates for three or more activities. Some templates specify the minimal number of execution of an activity. For example, templates “existence” and “existence_3” specify that activity “A” has to be executed at least once and three times, and are graphically represented with “1..*” and “3..*” above the activity, respectively. There are also templates that specify the maximal number of executions of an activity. Templates “absence_2” and “absence_3” specify that activity “A” can be executed at most once or two times and are graphically represented with “0..1” and “0..2” above the activity, respectively. It is also possible to specify the exact number of executions of an activity, e.g., exactly once or two times with templates “exactly_1” or “exactly_2”. The “response” template specifies that if “A” is executed then “B” has to be executed after “A”, and is denoted with a special line between “A” and “B”. According to the “precedence” template, “B” can be executed only after “A”. The “succession” template is a conjunction of templates “response” and “precedence”. It is also possible to specify that “B” cannot be executed after “A” with template “neg_response”. The template “responded_existence” specifies that if “A” is executed then also “B” has to be executed before or after “A”, thus without specifying in which order. The “alternate_response” formula takes the order of activities into account: in addition to the semantics of the “response” template, it imposes interposition, i.e., at least one target activity has to be executed between each two executions of the source activity. Finally, “chain_response” specifies the most strict ordering relations by requiring that the target activity must be executed immediately next to the source one. For a complete description of all DecSerFlow constraints, see Appendix A.

It is worth noting that constraints are interpreted within a given case (or choreography instance). As a consequence, negative relationships, such as the absence or the negation response constraints, forbid the presence of a certain activity within the same case in which the constraint has been triggered: other cases are not affected.

Finally, note that LTL is not the only language that can be used for the specification the semantics of DecSerFlow templates. Other declarative languages can be also used. Indeed, in this paper we show that declarative SCIFF is also suitable for specifying the semantics of DecSerFlow templates. Moreover, other types of logic can also be used. For example, Computation Tree Logic (CTL) is another logic that can be used in DecSerFlow. Although LTL and CTL are similar languages, each of them has some advantages over the other. For example, there are some relationships that can be specified only in LTL or in CTL, but not in both languages [Holzmann 2003]. However, so far, the debate about which of these two languages is more expressive remains unsolved [Holzmann 2003]. Finally, we chose LTL for the specification of DecSerFlow models because we were inspired by the so called LTL Checker plug-in in the process mining tool ProM, which can be used for verification of past executions against properties specified in LTL (the LTL Checker is described in more detail in Section 6.1.3 of this paper).

3.3 DecSerFlow Models

DecSerFlow models consists of activities and constraints. Constraints represent relationships between activities and are created from DecSerFlow templates. DecSerFlow models can be used both for global models of choreographies and local

models of services. In this section we present two DecSerFlow models: one for the global choreography model and one for the local model of the Photo Shop service.

3.3.1 Global Choreography Model. Figure 3 shows a global DecSerFlow model with agreed upon choreography constraints **G1**, **G2**, **G3**, **G4** and **G5** (cf. Table III) between a customer and the shop². Each constraint in this model originates from a template: the constraint inherits its name, semantics and graphical representation from its template. However, a constraint assigns “real” activities from a model to template’s parameters. For example, the “precedence” constraint replaces parameter “A” from the “precedence” template with activity “deliver” and parameter “B” with activity “receive”.

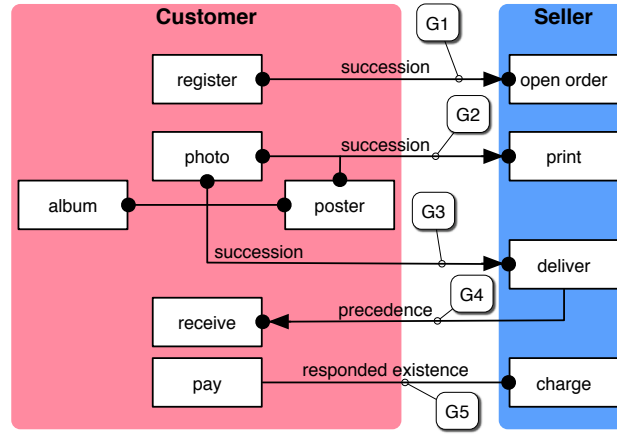


Fig. 3. Global Choreography DecSerFlow Model

Constraint “succession” between activities “register” and “open order” (**G1**) specifies that each alternation of customer data will be registered in the shop and the shop cannot open orders before the customer executes activity “register”. When this constraint is created, the activities “register” and “open order” replace parameters “A” and “B” in the template formula presented in Table V:

$$\begin{aligned}
 & \textit{succession}(\textit{register}, \textit{open order}) \\
 = & \textit{response}(\textit{register}, \textit{open order}) \wedge \textit{precedence}(\textit{register}, \textit{open order}) \\
 = & (\Box(\textit{register} \Rightarrow \Diamond(\textit{open order}))) \wedge (\Diamond(\textit{open order}) \Rightarrow ((\neg \textit{open order}) \sqcup \textit{register}))
 \end{aligned}$$

Note that this constraint (and its template) allows multiple executions of activities “register” and “open order” and also (multiple) executions of other activities between them. For example, this model allows a scenario where the user first “registers” data and then soon executes this activity again to alter data, while the shop

²Note that, for the purpose of the example, orders of different items (photos, albums and posters) have been represented as different activities.

executes activity “open order” only once after the second execution of the activity “register”.

Template “succession” is also used for the to constraints representing rules **G2** and **G3**. Unlike the previous constraint (**G1**) that utilizes only one activity (i.e., “register”) as the first parameter (i.e., “A”) in the template, **G2** and **G3** use two and three activities, respectively. This means that **G2** and **G3** *branch* parameter “A” of the **succession** on more activities. When a parameter is branched on several activities, then it is replaced by disjunction of these activities in the constraint formula. Therefore, formula for the “succession” constraint between activities “photo”, “poster” and “print” (**G2**) is:

$$\begin{aligned} & \textit{succession}(\textit{photo} \vee \textit{poster}, \textit{print}) \\ &= \textit{response}(\textit{photo} \vee \textit{poster}, \textit{print}) \wedge \textit{precedence}(\textit{photo} \vee \textit{poster}, \textit{print}) \\ &= ((\Diamond \textit{print}) \Rightarrow ((\neg \textit{print}) \sqcup (\textit{photo} \vee \textit{poster}))) \wedge \Box((\textit{photo} \vee \textit{poster}) \Rightarrow (\Diamond \textit{print})) \end{aligned}$$

This constraint specifies that shop cannot execute activity “print” before the customer executes activity “photo” or activity “poster” and that after every time activities “photo” or “poster” are executed, shop eventually executes activity “print”. This constraint allows, for example, situations where photos are ordered and printed and then posters are ordered and printed. It also allows situations where both photos and posters are first ordered and then they are printed at the same time.

Similarly, the “succession” constraint between activities “photo”, “poster”, “album” and “deliver” (**G3**) specifies that each execution of activities “photo”, “poster” or “album” is eventually followed by at least one execution of activity “deliver”. Also, activity “deliver” can be executed only after at least one of the activities “photo”, “poster” or “album” was executed. With this constraint, the shop can first collect orders for photos, posters and albums and then deliver them at once. However, it is also possible to immediately deliver each of the orders as soon as it is placed.

The “precedence” constraint between activities “receive” and “deliver” (**G4**) prevents the execution of activity “receive” before the execution of activity “deliver”. In other words, customer can execute activity “receive” only after the first execution of activity “deliver”. An example of a scenario allowed by this constraint is when two packages were sent in one delivery and the customer receives them separately (e.g., one get lost in the postal system and arrives three days later).

The last constraint is the “responded existence” constraint between activities “charge” and “pay”. It makes sure that customer executes activity “pay” when the shop executes activity “charge” (**G5**). It is possible that the payment is done before activity “charge” takes place, e.g., if the customer payed with a credit card immediately after ordering, and the shop executed activity “charge” only after all products were delivered.

Note that, although this is not the case with model in Figure 3, DecSerFlow constraints can be conditional. Conditions on constraints are logical expressions involving instance data elements. For example, a constrain can be valid (interacting parties should fulfill the constraint) only if “*price* > 1000”. Because data are not directly involved in constraints (like, e.g., activities are) but only in conditions, LTL representation of DecSerFlow covers only partially the perspective of instance

data.

The DecSerFlow model in Figure 3 shows global constraints as rules that all parties have to follow in the choreography regardless their local service models. Because of the declarative nature of DecSerFlow constraints, it is possible to employ various local models for both parties as long as they comply to such a global choreography model.

3.3.2 Local Shop Model. Besides for specification of global choreography models, DecSerFlow can be used to specify local models of services. Because of the declarative nature of DecSerFlow, service models become flexible and able to engage in a variety of choreographies. Figure 4 depicts two examples of possible local models for the Photo Shop service.

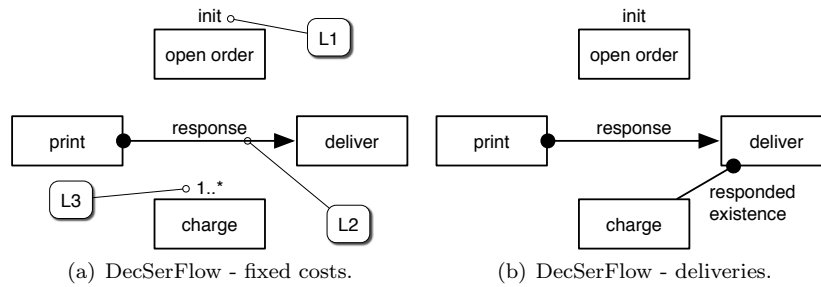


Fig. 4. Two examples of local shop models

Figure 4(a) shows a DecSerFlow model for the Photo Shop service based on the local constraints as presented in Table IV. Constraint “init” implements rule L1 and specifies that each service execution has to start with activity “register”, but this activity can be executed multiple times at later stages of the execution. Rule L2 is represented using a “response” constraint that makes sure that all printed products are eventually delivered. It might be that all products are printed first in several stages (several executions of activity “print”) and then delivered at once. It is also possible that products are delivered immediately after they are printed, without waiting for other products to be printed (e.g., some products are delivered to home address via mail and a large poster has to be picked up personally in the shop). Finally, constraint “1..*” makes sure that the shop will execute activity “charge” at least once, even if no delivery has been made and only fixed administrative costs are charged (L3). Activity “charge” can be executed multiple times in the shop process if necessary to charge part by part of the order (e.g., one part from the credit card for photos and other in the shop when the large poster is picked up).

The local shop model presented in Figure 4(a) is compliant with the global choreography model in Figure 3. Moreover, because of the declarative nature of the global model, this model is *flexible* enough to allow for many other local models of involved parties. For example, Figure 4(b) shows a DecSerFlow model of another shop which does not charge fixed costs for empty services (but only deliveries). This local model can also join the choreography in Figure 3.

4. THE SCIFF FRAMEWORK

SCIFF [Alberti et al. 2008] is a logic-based framework aimed at specifying global interaction protocols (e.g. a service choreography) in a declarative fashion as well as providing support for performing different verification tasks.

SCIFF tackles interaction by adopting a social approach, as it is called in the MAS community [Singh 2000]. Interaction is specified in a declarative manner by only constraining the external observable behaviour of interacting entities, without stating any assumption on their internal architecture and, thus, supporting heterogeneity. Moreover, as in DecSerFlow the adopted perspective is open, i.e. interacting entities can freely behave where not explicitly constrained.

The fundamental concepts used by SCIFF to specify such interaction are (i) observable and relevant events which occur at execution time, (ii) expectations about further events and courses of interaction and (iii) (Social) Integrity Constraints which allow the user to constrain the global interaction.

4.1 Events, Happened Events and Expected Events

Deciding what has to be considered an event strictly depends on the application domain. Furthermore, even if the application domain is fixed, there could be several different notions of events, because of the assumed perspective, the granularity, and so on.

The SCIFF 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. For example, in a business context, an event could be the fact that some atomic activity has been performed

$$\textit{performed}(\textit{Activity}, \textit{Originator}, \textit{InputData}, \textit{OutputData})$$

Happened events are represented as an atom

$$\mathbf{H}(\textit{Event}, \textit{Time})$$

where *Event* is a *Term* and *Time* is an integer, representing the discrete time point at which the event happened. **HAP** is the set of all the events that happened during the execution. Together, these events form a log (or execution trace).

Beside the explicit representation of “what” happened and “when”, it is possible to explicitly represent also “what” is expected, and “when” it is expected to happen. The notion of *expectation* is used to represent the (un)desired course of interaction, and plays a key role when defining interaction protocols, choreographies, and more in general any dynamically evolving process: it is quite natural, in fact, to think of such processes in terms of rules of the form “*if A happened, then B is expected (not) to happen*”.

In agreement with DecSerFlow, SCIFF pays particular attention to the *openness* of interaction; this means that the prohibition of a certain event should be explicitly expressed in the model and this is the reason why SCIFF supports also the concept of negative expectation (i.e. of what is expected not to happen).

Positive expectations about events come with form

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

where *Event* and *Time* can be variables, or they could be grounded to a particular term/value. Constraints can be specified over each variable; for example $Time > 10$ states that 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

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

Generally speaking, quantification of the variables inside happened events and positive/negative expectations follows their intuitive meaning: an happened event represents a “class” of possible occurring events, and therefore variables used in a happened event are universally quantified. For example,

$$\mathbf{H}(performed(deliver, Originator), T_d) \wedge T_d > 10$$

matches with *any* execution of the activity “deliver” at a time greater than 10 time units, performed by a *whatsoever* *Originator*. Positive expectations are existentially quantified: an expectation is fulfilled when one single matching event indeed happens; hence, specifying

$$\mathbf{E}(performed(deliver, Originator), T_d) \wedge T_d > 10$$

means that there should exist an *Originator* which performs activity “deliver” at a time greater than 10. Finally, negative expectations are universally quantified, since they specify what is forbidden and when;

$$\mathbf{EN}(performed(deliver, Originator), T_d) \wedge T_d > 10$$

means that *nobody* can perform the activity “deliver” at *any* time greater than 10.

For a complete description of variables quantification, the interested reader may refer to [Alberti et al. 2008].

4.2 Social Integrity Constraints

Social Integrity Constraints (\mathbf{IC}_S) are rules used to relate happened events and expectations. They allow the user to constrain global interaction, given some previous situation that can be represented in terms of happened events.

They are represented as forward rules of the form $Body \rightarrow Head$, where *Body* can contain literals and (conjunctions of happened and expected) events, and *Head* can contain (disjunctions of) conjunctions of expectations.

In Table VI we show the definition of a subset of the grammar (for a complete description, see [Alberti et al. 2008]), where *Atom* and *Term* have the usual meaning in Logic Programming [Lloyd 1987] and *Constraint* is interpreted as in Constraint Logic Programming (CLP) [Jaffar and Maher 1994].

CLP constraints and Prolog predicates can be used to impose conditions or restrictions on each variable that occurs in happened events and expectations. For example, time conditions might define orderings between messages, or enforce deadlines.

Definition of such predicates and of all “static” background knowledge about interaction (i.e., information independent from specific executions) is formalized inside a knowledge base *KB*, which completes the definition of Integrity Constraints. Here we could specify roles descriptions, list of participants, conditions on data, etc.

$ExtLiteral$	$::=$	$Literal \mid Exp \mid Constraint$
Exp	$::=$	$\mathbf{E}(Event, Time) \mid \mathbf{EN}(Event, Time)$
$HEvent$	$::=$	$\mathbf{H}(Event, Time)$
$Event$	$::=$	$Term$
\mathcal{IC}_S	$::=$	$[IC]^*$
IC	$::=$	$Body \rightarrow Head$
$Body$	$::=$	$(HEvent \mid Exp \mid true) [\wedge BodyLiteral]^*$
$BodyLiteral$	$::=$	$HEvent \mid ExtLiteral$
$Head$	$::=$	$HeadDisjunct [\vee HeadDisjunct]^* \mid false$
$HeadDisjunct$	$::=$	$ExtLiteral [\wedge ExtLiteral]^*$
KB	$::=$	$[Clause]^*$
$Clause$	$::=$	$CHead \leftarrow CBody$
$CHead$	$::=$	$Atom$
$CBody$	$::=$	$ExtLiteral [\wedge ExtLiteral]^* \mid true$

Table VI. Syntax of Integrity Constraints (\mathcal{IC}_S) and the Knowledge Base (KB)

KB is expressed in the form of clauses (a logic program); clauses may contain in their body expectations about the behaviour of participants, defined literals, and constraints (see Table VI).

EXAMPLE 4.1. *A rule like*

“If a premium customer pays for an item by credit card, then the seller should answer within 10 minutes by delivering a corresponding receipt, or by communicating a payment failure.”

can be translated in a straightforward manner, e.g. in the corresponding rule (supposing that times are expressed in minutes):

$$\begin{aligned}
& \mathbf{H}(\text{pay}(\text{Customer}, \text{Seller}, \text{Item}, \text{credit_card}), T_p) \\
& \wedge \text{premium_customer}(\text{Customer}, \text{Seller}) \\
& \rightarrow \mathbf{E}(\text{deliver}(\text{Seller}, \text{Customer}, \text{receipt}(\text{Item}, \text{Info}), T_d) \wedge T_d > T_p \wedge T_d < T_p + 10 \quad (1) \\
& \vee \mathbf{E}(\text{tell}(\text{Seller}, \text{Customer}, \text{failure}, \text{Reason}), T_f) \wedge T_f > T_p \wedge T_f < T_p + 10.
\end{aligned}$$

where $\text{premium_customer}(\text{Customer}, \text{Seller})$ is used to represent whether Customer is actually a premium one.

To express mutual exclusion between delivery and failure communication, we could also add a rule like

$$\begin{aligned}
& \mathbf{H}(\text{deliver}(\text{Seller}, \text{Customer}, \text{receipt}(\text{Item}, \text{Info}), T_d) \\
& \rightarrow \mathbf{EN}(\text{tell}(\text{Seller}, \text{Customer}, \text{failure}, \text{Reason}), T_f). \quad (2)
\end{aligned}$$

and viceversa³.

In the following, we show that DecSerFlow can be suitably mapped onto a SCIFF specification (see section 5), and we exploit its operational counterpart (in terms of the SCIFF proof procedure) to perform different verification tasks (see section 6.2).

³Note that such rules model the “not coexistence” formula in DecSerFlow.

5. MAPPING DECSEFLOW ONTO THE SCIFF FRAMEWORK

All the different DecSerFlow formulas can be intuitively mapped onto SCIFF Integrity Constraints. More specifically, we now introduce the mapping of basic DecSerFlow formulas, and then discuss how the expressive power of the SCIFF language could be used to extend DecSerFlow with explicit temporal constraints, such as delays and deadlines, by maintaining a complete valid underlying semantics in SCIFF. Finally, we summarize how a specific DecSerFlow diagram could be mapped onto SCIFF, by directly combining the formalization of template formulas with a specific knowledge representing the diagram.

5.1 Formalization of activities

As pointed out in Section 4.1, SCIFF 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(Activity)*. Thus, notation $\mathbf{H}(\text{performed}(\text{buy_item}), 18)$ means that the *buy_item* activity has been executed at time 18. If also the originator and input/output data have to be considered, we could simply extend the representation as follows:

$$\text{performed}(\text{Activity}, \text{Originator}, \text{InputData}, \text{OutputData})$$

It is worth noting that, in principle, a non atomic model of activities could be seamlessly supported, by mapping the start and completion of each activity to events.

5.2 Mapping of DecSerFlow constraints

Table VII introduces the mapping of some DecSerFlow formulas onto SCIFF.

Let us consider first unary formulas. The “absence_N” formula leads to the generation of a negative expectation about the execution of the involved activity, after *N* different executions of the same activity have already occurred. Instead, the “existence_N” one states that *N* different execution of the activity are expected to happen. Since SCIFF adopts an explicit notion of time, the difference between expectations about the same activity is modeled as a difference between the corresponding execution times.

The SCIFF representations of the “absence” and the “existence_N” formulas do not have any triggering condition (i.e. their body do not contain happened events): the involved expectations are always hypothesized and should always be fulfilled, independently from the course of interaction.

The same holds for the DecSerFlow “substitution” formula, which specifies that at least one of the involved activities should be executed. The substitution between “A” and “B” is mapped onto SCIFF as follows:

$$\text{true} \rightarrow \mathbf{E}(\text{performed}(A), T_A) \vee \mathbf{E}(\text{performed}(B), T_B).$$

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” formulas about “A”. The former is indeed satisfied when *N* executions of “A” happened; but these *N* happened events trigger the latter, which forbids further executions of “A”.

name	Integrity Constraint	graphical
$existence(A)$	$true \rightarrow \mathbf{E}(performed(A), T)$	
$existence_N(A)$	$true \rightarrow \bigwedge_{i=1}^N (\mathbf{E}(performed(A), T_i) \wedge T_i > T_{i-1})$	
$absence(A)$	$true \rightarrow \mathbf{EN}(performed(A), T)$	
$absence_N+1(A)$	$\bigwedge_{i=1}^N (\mathbf{H}(performed(A), T_i) \wedge T_i > T_{i-1}) \rightarrow \mathbf{EN}(performed(A), T) \wedge T > T_N$	
$exactly_N(A)$	$existenceN(A) \wedge absenceN+1(A)$	
$response(A, B)$	$\mathbf{H}(performed(A), T_A) \rightarrow \mathbf{E}(performed(B), T_B) \wedge T_B > T_A$	
$precedence(A, B)$	$\mathbf{H}(performed(B), T_B) \rightarrow \mathbf{E}(performed(A), T_A) \wedge T_A < T_B$	
$succession(A, B)$	$response(A, B) \wedge precedence(A, B)$	
$neg_response(A, B)$	$\mathbf{H}(performed(A), T_A) \rightarrow \mathbf{EN}(performed(B), T_B) \wedge T_B > T_A$	
$responded_existence(A, B)$	$\mathbf{H}(performed(A), T_A) \rightarrow \mathbf{E}(performed(B), T_B)$	
$alternate_response(A, B)$	$response(A, B) \wedge \mathbf{H}(performed(A), T_A) \wedge \mathbf{H}(performed(A), T_{A2}) \wedge T_{A2} > T_A \rightarrow \mathbf{E}(performed(B), T_B) \wedge T_B > T_A \wedge T_B < T_{A2}$	
$chain_response(A, B)$	$\mathbf{H}(performed(A), T_A) \rightarrow \mathbf{E}(performed(B), T_B) \wedge T_B > T_A \wedge \mathbf{EN}(performed(X), T_X) \wedge T_X > T_A \wedge T_X < T_B$	

Table VII. Mapping of some DecSerFlow formulas onto SCIFF.

The mapping of relation formulas has a more fixed structure. The body of each Integrity Constraint is constituted in this case by the happened event which corresponds to the formula's source; in fact, each DecSerFlow relation is triggered when its source activity is performed.

While the LTL formalization implicitly models concepts like *before* and *after* by exploiting temporal modalities, SCIFF specifies them by explicitly constraining time variables, i.e. by adopting a point algebra [Vilain et al. 1990] and exploiting the underlying CLP solver; hence, to formalize the “response” formula SCIFF states that if the source activity “A” happens at time “ T_A ”, then the target activity B is expected to happen at a time “ $T_B > T_A$ ”. The “precedence” version of each

DecSerFlow relation is therefore formalized in the same way as the “response” one, except from the fact that temporal constraints are inverted.

Formalization of “alternate” formulas imposes, in addition to normal “response” / “precedence” behaviour, the interposition between activities; closely following the natural language description, interposition is expressed by stating that between two executions of the source activity the target activity must be performed at least once.

The “chain_response” formula is instead formalized by applying the “response” rule and by forbidding *all* events between the execution of source and target activities. In this way, we map the concept of *next state* in LTL with the one of *first next time at which* some new activity is performed (and, in the “chain_response” case, such a new activity must be the target one). This is a proper formalization when execution times may be explicitly constrained, and this is one of the added feature that SCIFF provides to DecSerFlow (see section 5.4).

Finally, it is worth noting that mapping of negation formulas resembles very closely the one of relation formulas. The main obvious difference is that while relation formulas specifies what should be done, negation formulas specifies what is forbidden, hence their formalization substitutes the concept of positive expectation with the the one of negative expectation.

5.3 Branching formulas

For the sake of simplicity, in the previous section we have limited our mapping to binary relation and negation formulas. However, SCIFF is able to capture also branching formulas, i.e. relation and negation formulas which envisage more than two activities. As sketched in section 3.2, the presence of n source or target activities is interpreted by DecSerFlow in a disjunctive manner. More specifically, when n source activities “ A_1 ”, ..., “ A_n ” are used, then the formula should be satisfied whenever “ A_1 ” or “ A_2 ” or ... or “ A_n ” is executed; hence, modeling a formula with disjunctive sources is a short-cut for applying the formula on each source activity. The intended meaning can then be easily captured by replicating the SCIFF formalization for each single source activity. The presence of n target activities means instead that the formula is satisfiable in different ways, i.e. it is true whenever it is satisfied at least by one of the target activities. Hence, the formalization of a formula with disjunctive targets can be expressed by considering disjunction of target expectations (together with the corresponding temporal constraints) as rule’s head.

Table 5.3 shows how such a formalization is applied to the case of a branching “responded existence” formula. It ⁴ tackles also the situation, not envisaged by core DecSerFlow formulas, of a branching “responded existence” where target/source multiplicity is interpreted in a conjunctive manner. Such an interpretation behaves in the opposite way w.r.t. the disjunctive one. A formula with conjunct targets is fulfilled when it is true for all target activities, and hence it can be formalized by replicating the corresponding Integrity Constraint for each activity. A more complex case is the one in which the formula has conjunct source activities: it should trigger only when all such activities are executed. SCIFF is directly able to represent this feature: the corresponding rule will have as body the conjunction of

⁴But the same holds also for the other relation and negation formulas.

name	representation	equivalent representation	formalization
<i>responded_existence</i> ($A_1 \vee A_2, B$)			$\mathbf{H}(\text{performed}(A_1), T_A)$ $\rightarrow \mathbf{E}(\text{performed}(B), T_B).$ $\mathbf{H}(\text{performed}(A_2), T_A)$ $\rightarrow \mathbf{E}(\text{performed}(B), T_B).$
<i>responded_existence</i> ($A, B_1 \vee B_2$)			$\mathbf{H}(\text{performed}(A_1), T_A)$ $\rightarrow \mathbf{E}(\text{performed}(B_1), T_{B_1})$ $\vee \mathbf{E}(\text{performed}(B_2), T_{B_2}).$
<i>responded_existence</i> ($A, B_1 \wedge B_2$)			$\mathbf{H}(\text{performed}(A), T_A)$ $\rightarrow \mathbf{E}(\text{performed}(B_1), T_{B_1}).$ $\mathbf{H}(\text{performed}(A), T_A)$ $\rightarrow \mathbf{E}(\text{performed}(B_2), T_{B_2}).$
<i>responded_existence</i> ($A_1 \wedge A_2, B$)			$\mathbf{H}(\text{performed}(A_1), T_{A_2})$ $\wedge \mathbf{H}(\text{performed}(A_2), T_{A_2})$ $\rightarrow \mathbf{E}(\text{performed}(B), T_B).$

 Table VIII. Formalization of a branching “responded existence” formula in *SCIFF*.

the involved happened events.

5.4 Extending DecSerFlow with quantitative temporal constraints

Another interesting feature, due to *SCIFF* reasoning capabilities on content data (and therefore also on execution times), is the possibility to extend the basic DecSerFlow relation formulas (and the simple negation formulas, i.e. “negation response” and “negation precedence”) with quantitative information over times, e.g. to express delays and deadlines. Such an information is used to reduce the validity of formula’s target time (or, in the negative case, to delimit the forbidding of the target), by defining either a lower or an upper bound on it.

Let us modify, for example, the photo choreography shown in Figure 3 by specifying also that “at most 24 hours can elapse between the order of a product and the corresponding delivery”. By assuming that times are expressed in hours, such a statement could be represented by augmenting the “succession” formula between the three kinds of order and the “deliver” activity with the knowledge about the deadline: the delivery time should be after the order one, but also less than the order one plus 24 hours (and vice versa). This could be seamlessly modeled in *SCIFF* by extending the formalization of the “succession” formula as follows (for simplicity, in the formalization we consider only activities “photo” and “deliver”):

$$\begin{aligned} \mathbf{H}(\text{performed}(\text{photo}), T_p) &\rightarrow \mathbf{E}(\text{performed}(\text{deliver}), T_d) \\ &\wedge T_d > T_p \wedge T_d < T_p + 24. \end{aligned} \quad (3)$$

$$\begin{aligned} \mathbf{H}(\text{performed}(\text{deliver}), T_d) &\rightarrow \mathbf{E}(\text{performed}(\text{photo}), T_p) \\ &\wedge T_p < T_d \wedge T_p > T_d - 24. \end{aligned} \quad (4)$$

To graphically show these temporal extensions, a possible choice is to annotate the different DecSerFlow constraints with a time interval marked off by two non negative instants (T_{min} and T_{max}) which could be considered both in an exclusive

or inclusive manner. As usually, parentheses ((\dots)) are used to indicate exclusion and square brackets ($[\dots]$) to indicate inclusion. The interval is treated as relative w.r.t. the time at which the source happens, and is translated backward or forward w.r.t. it depending on the nature of the formula (i.e. whether it is a “response” or “precedence” one).

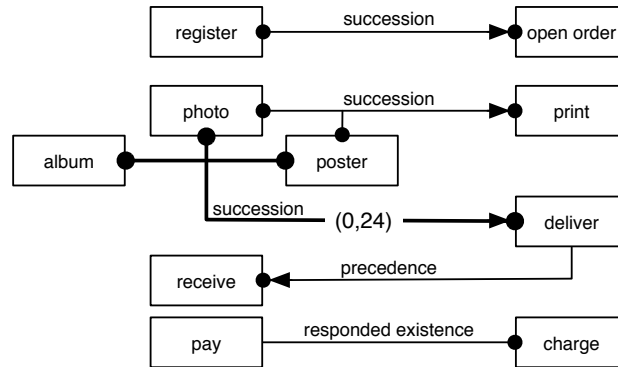


Fig. 5. Modification of the DecSerFlow running example by adding a temporal deadline.

For example, let us consider again the modified running example, denoting with T_o and T_d the execution times of one of the order activities and “deliver” respectively. As shown in Figure 5, the annotation of the succession formula to reflect the declared deadline should be $(0, 24)$, since T_d has to belong to the time interval $(T_p, T_p + 24)$ and, conversely, T_o to the interval $(T_d - 24, T_d)$.

This intended meaning is clarified in Figure 6, which summarizes how temporal annotations could be used to model different kind of quantitative temporal relationships in case of simple relation formulas, namely “responded existence”, “response” and “precedence”.

Quantitative temporal constraints should not be interpreted as a way to *force* the emission of a message from a certain service; indeed, when the focus is on the choreography the concrete interacting services execute in an autonomous way, and cannot be controlled. Instead, temporal constraints should be considered as a mean to specify further requirements on the interacting service, contributing to the definition of the QoS that must be guaranteed during the interaction: compliant executions must not only respect the modeled constraints, but also satisfy all the temporal requirements. For example, the deadline of 24 hours introduced in Figure 5 could represent a QoS requirement of the customer, who considers as good candidates to interact with only photo services able to deliver within a maximum timespan. This information can be used either to statically select “good” photo services, i.e., photo services whose behavioural interface respects (promises to respect) the desired temporal constraint, and to check at run-time if the real behaviour effectively satisfies it. Identifying a violation may be useful in this setting to alert the customer that the photo service is breaking the choreographic contract.

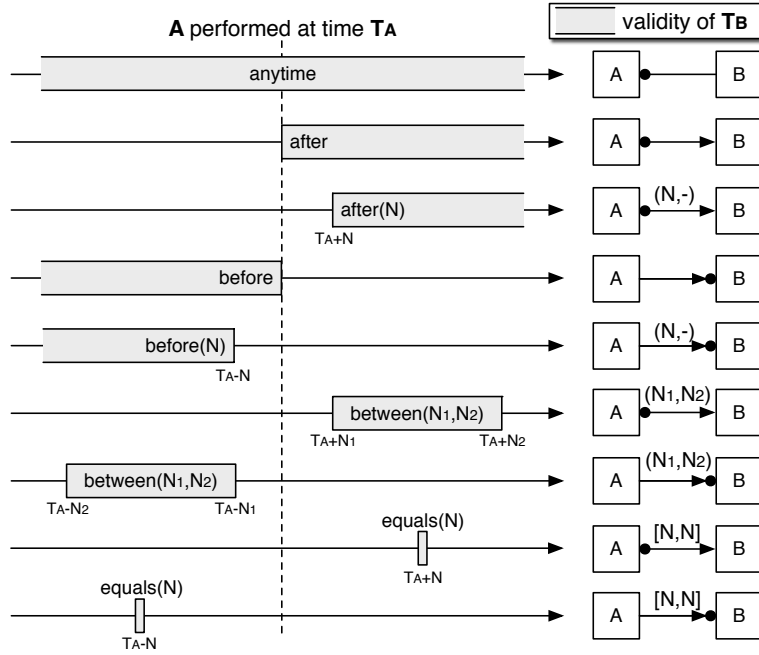


Fig. 6. Temporal constraints templates and their corresponding representation on simple formulas.

5.5 Cross-flow Constraints

Similarly to the approach presented in the previous Section aimed at extending DecSerFlow specifications w.r.t. the temporal dimension, also other kinds of data may be modeled and constrained by SCIFF. For example, rule 1 (page 17) shows how sender, receiver and content data of a message can be seamlessly introduced and used. However, introducing data and their corresponding constraints at the graphical level of DecSerFlow is a complex task, mainly because the right balance between expressiveness and usability must be found. Even if the introduction of data and data-related conditions into the DecSerFlow notation has not been yet investigated, in this Section we briefly sketch how the possibility of dealing with data, provided by SCIFF, could be exploited to model a wider range of constraints.

In particular, let us review the concept of negative relationship in DecSerFlow. Negative relationships deal with the forbidding of an activity under certain circumstances. For example, the “negation response” between two activities “a” and “b” states that if “a” is executed, then “b” cannot be executed afterwards. The forbidding of “b” is limited to the case inside which “a” has been executed: each choreography instance follows its own evolution, independently from the other cases. However, there could be situations in which it would be desirable that constraints span across multiple cases. An example of a *cross-flow* constraint would be that if the seller detect that a certain customer C is behaving in a fraudulent way, then it will never deliver anything to C in the future, even in new instances of execution.

SCIFF is able to easily capture this requirement. A possible solution would be

associate each activity to a case identifier, modeled as a special datum. By default, constraints operate within the same instance, and therefore a “normal” negation response would be modeled as follows:

$$\mathbf{H}(\text{performed}(a, \text{Case}), T_a) \rightarrow \mathbf{EN}(\text{performed}(b, \text{Case}), T_b) \wedge T_b > T_a.$$

Here, the same *Case* variable is shared by the events associated to “a” and “b”, and therefore if “a” is executed inside case *c1*, then the forbidding of “b” will be imposed only on the same *c1*.

Cross-flow constraints could be therefore modeled by simply introducing two different case variables. For example, to model that “when a seller *S* detect that a customer *C* is behaving in a fraudulent way, it will never deliver goods to *C*” the following (extended) negation response could be adopted:

$$\begin{aligned} &\mathbf{H}(\text{performed}(\text{fraud_detected}(C), S, \text{Case}), T_f) \\ &\rightarrow \mathbf{EN}(\text{performed}(\text{deliver}(S, C, G), \text{Case}_2), T_d) \wedge T_d > T_a. \end{aligned}$$

Since two different variables *Case* and *Case*₂ are used, when a fraud is detected the forbidding is imposed on any case (*Case* included): negative expectation are universally quantified. Contrarywise, the same customer *C* is involved in the body and in the head of the rule, and therefore the forbidding is imposed only on that *C*, without affecting the interaction between *S* and other customers.

5.6 Explicit and implicit formalization of DecSerFlow templates

We have shown the mapping of core DecSerFlow templates to SCIFF Integrity Constraints. Obviously, for a given model these different rules will be grounded on each specific instance, substituting involved activities with the concrete names. In this respect, for each relationships of the model the formalization will explicitly contain a corresponding set of rules. However, it is possible to generalize the formalization of DecSerFlow formulas by directly representing templates. In this way, specific concrete rules are implicitly model: the translation of a specific DecSerFlow diagram simply reduces to compile a knowledge base with a list of facts representing the different modeled formulas.

The general formalization is realized by adding as a first conjunct in the bodies of rules a predicate which represents the corresponding relationship. This predicate will match, for a given knowledge base, with all the facts representing instances of such relationship.

To define relation and negation templates, we therefore adopt the following pattern:

$$\begin{aligned} &\text{formula_Type}(\text{Source}, \text{Target}) \\ &\quad \wedge \text{body} \rightarrow \text{head}. \end{aligned} \tag{5}$$

It is worth noting that in the first line of all Integrity Constraints of this kind, variables (i.e. activities) are universally quantified. This ensures that, when considering a specific diagram, each rule will be replicated for all concrete (ground) activities subject to the formula addressed by the rule.

An example which clarifies this approach in case of the “response” template follows.

EXAMPLE 5.1. *Let us consider the general specification of the “response” relation, which is formalized as follows:*

$$\begin{aligned} & \text{response}(A, B) \\ & \wedge \mathbf{H}(\text{performed}(A), T_A) \rightarrow \mathbf{E}(\text{performed}(B), T_B) \wedge T_B > T_A. \end{aligned} \quad (6)$$

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 should exist a T_B after T_A at which B is expected to be performed”.

Let us now consider the simple following knowledge base:

`response(ask_for_payment, pay).`
`response(receive_spam, delete_spam).`

During execution, SCIFF will find two different matches for the “response” formula, automatically grounding the above Integrity Constraint on each concrete relationship:

$$\begin{aligned} & \mathbf{H}(\text{performed}(\text{ask_for_payment}), T_A) \rightarrow \mathbf{E}(\text{performed}(\text{pay}), T_B) \wedge T_B > T_A. \\ & \mathbf{H}(\text{performed}(\text{receive_spam}), T_A) \rightarrow \mathbf{E}(\text{performed}(\text{delete_spam}), T_B) \wedge T_B > T_A. \end{aligned}$$

5.7 Implicit and explicit mapping of the running example onto SCIFF

By using this kind of “implicit” formalization we have now the possibility to completely separate the formalization of the general DecSerFlow templates and the formalization of a specific model. In particular, DecSerFlow can be mapped onto an abductive logic program whose Integrity Constraints are the ones which implicitly formalize template formulas, and whose knowledge base is used to capture the general background knowledge of DecSerFlow concepts. For example, in such a knowledge base we will find that the “succession” formula is defined in terms of the “response” and the “precedence” ones:

$$\begin{aligned} & \text{response}(A, B) \leftarrow \text{succession}(A, B). \\ & \text{precedence}(A, B) \leftarrow \text{succession}(A, B). \end{aligned}$$

Then, as already pointed out in example 5.1, formalizing a particular DecSerFlow diagram just implies to (i) compile another knowledge base which maps the specific diagram structure enumerating all its constraints as facts, and (ii) use it together with the general specification.

Tables IX and X show how the running example depicted in Figure 3 can be mapped onto SCIFF by respectively adopting explicit rules grounded on the example or by exploiting the possibility to use the general mapping and simply formalize the specific diagram as a list of facts. It is worth noting that, exactly as shown in example 5.1, matching the general implicit DecSerFlow Integrity Constraints with the knowledge base of Table X will have the effect of obtaining the rules shown in Table IX.

G_1	a	$\mathbf{H}(\text{performed}(\text{register}), T_r)$	\rightarrow	$\mathbf{E}(\text{performed}(\text{open_order}), T_o) \wedge T_o > T_r.$
	b	$\mathbf{H}(\text{performed}(\text{open_order}), T_o)$	\rightarrow	$\mathbf{E}(\text{performed}(\text{register}), T_r) \wedge T_r < T_o.$
G_2	a	$\mathbf{H}(\text{performed}(\text{photo}), T_{ph})$	\rightarrow	$\mathbf{E}(\text{performed}(\text{print}), T_{pr}) \wedge T_{pr} > T_{ph}.$
	b	$\mathbf{H}(\text{performed}(\text{poster}), T_{po})$	\rightarrow	$\mathbf{E}(\text{performed}(\text{print}), T_{pr}) \wedge T_{pr} > T_{po}.$
	c	$\mathbf{H}(\text{performed}(\text{print}), T_{pr})$	\rightarrow	$\mathbf{E}(\text{performed}(\text{photo}), T_{ph}) \vee T_{ph} < T_{pr}$ $\vee \mathbf{E}(\text{performed}(\text{poster}), T_{po}) \vee T_{po} < T_{pr}.$
G_3	a	$\mathbf{H}(\text{performed}(\text{photo}), T_p)$	\rightarrow	$\mathbf{E}(\text{performed}(\text{deliver}), T_d) \wedge T_d > T_p.$
	b	$\mathbf{H}(\text{performed}(\text{poster}), T_p)$	\rightarrow	$\mathbf{E}(\text{performed}(\text{deliver}), T_d) \wedge T_d > T_p.$
	c	$\mathbf{H}(\text{performed}(\text{album}), T_a)$	\rightarrow	$\mathbf{E}(\text{performed}(\text{deliver}), T_d) \wedge T_d > T_a.$
	d	$\mathbf{H}(\text{performed}(\text{deliver}), T_d)$	\rightarrow	$\mathbf{E}(\text{performed}(\text{photo}), T_{ph}) \wedge T_{ph} < T_d$ $\vee \mathbf{E}(\text{performed}(\text{poster}), T_{po}) \wedge T_{po} < T_d$ $\vee \mathbf{E}(\text{performed}(\text{album}), T_a) \wedge T_a < T_d.$
G_4		$\mathbf{H}(\text{performed}(\text{receive}), T_r)$	\rightarrow	$\mathbf{E}(\text{performed}(\text{deliver}), T_d) \wedge T_d < T_r.$
G_5		$\mathbf{H}(\text{performed}(\text{charge}), T_c)$	\rightarrow	$\mathbf{E}(\text{performed}(\text{pay}), T_p).$

Table IX. Explicit SCIFF mapping of the DecSerFlow running example.

G_1	<i>succession(register, open_order).</i>
G_2	<i>succession([photo, poster], print).</i>
G_3	<i>succession([photo, poster, album], deliver).</i>
G_4	<i>precedence(deliver, receive).</i>
G_5	<i>responded_existence(charge, pay).</i>

Table X. Mapping the DecSerFlow running example to a simple knowledge base.

6. ENACTMENT, VERIFICATION AND DYNAMIC CHANGE OF DECSEFLOW MODELS

The LTL and SCIFF notations of DecSerFlow enables various verification techniques and even enactment of DecSerFlow models. Through the combination of DecSerFlow as a modeling language and the two underlying semantics with their corresponding verification techniques, we aim to realize a comprehensive framework for the specification, enactment, and verification of service choreographies (see Figure 7). DECLARE [Pesic et al. 2007]⁵ is a tool that can be used to develop and execute models specified in DecSerFlow or any other LTL based language. Although it is not implemented as a web service application, DECLARE can be used to experiment with templates and models in order to better understand templates and the execution semantics of DecSerFlow models. DECLARE uses the approach described in this section for the execution, dynamic change and verification of DecSerFlow models.

⁵DECLARE can be downloaded from <http://declare.sf.net>.

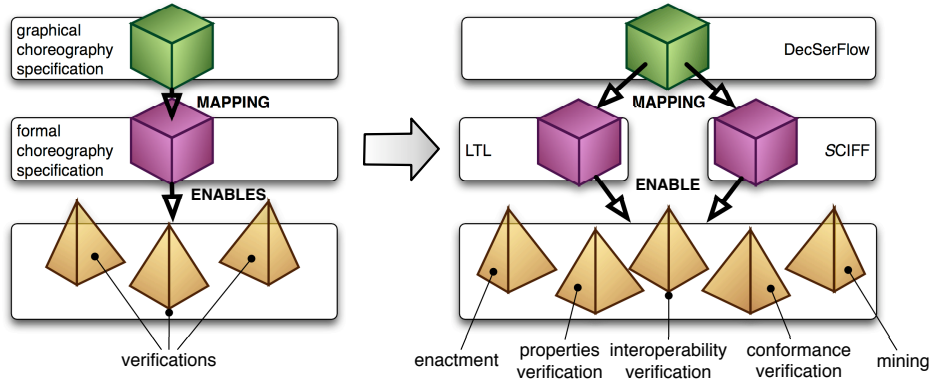


Fig. 7. General schema of a framework for the specification and verification of service choreographies, and its possible realization by the conjunct use of DecSerFlow, LTL and SCIFF.

6.1 LTL Enactment and Verification

The LTL mapping of DecSerFlow enables the computer-supported execution of local models of participating services, verification of models, monitoring (conformance-checking) of service execution and verification of composition interoperability (i.e., can different models be combined into one composition).

6.1.1 Supporting Execution of DecSerFlow Models. While interacting within a global choreography, each service should align its execution with its local model and the global choreography model. With respect to DecSerFlow, this means that a completed interaction must satisfy the local model of interacting services and the global choreography model. A DecSerFlow model is satisfied if it is executed in such a way that all its constraints are satisfied at the end of the execution. Some application can regulate a correct execution of DecSerFlow models thanks to (1) the constraint semantics expressed with LTL formulas and (2) the possibility to generate an automaton that represents all executions that satisfy an LTL formula. The desire to generate automata for LTL formulas and to define algorithms for this purpose originates in the field of model checking [Clarke et al. 1999; Gerth et al. 1996; Giannakopoulou and Havelund 2001]. In this field, systems can be checked against certain properties specified in LTL using the generated automata. the computer-supported regulation of a correct execution of DecSerFlow relies on the automaton generated from LTL specifications of constraints in the model (i.e., one automaton is generated for a formula representing a *conjunction of all constraints*) [van der Aalst and Pesic 2006]. Because it is generated for the conjunction of all constraints form a DecSerFlow model, such an automaton represents exactly all correct executions of the model (i.e., all executions that satisfy all constraints) [Clarke et al. 1999; Gerth et al. 1996; Giannakopoulou and Havelund 2001]. In other words, using this automaton, it is possible to, during execution, (1) monitor the current state of the execution by monitoring the current state of the automaton, and (2) precisely identify which activities can be executed next given the current state of the automaton. Consider, for example, the “precedence” DecSerFlow template. If a DecSerFlow model contains a “precedence” constraint between two activities “de-

liver” and “receive” (e.g., Figure 3), then the automaton created for this model will allow execution of “receive” only after “deliver” is executed, as shown in Figure 8. The theory on automata and their generation from LTL formulas is out of scope of this paper and we refer the interested reader to [Clarke et al. 1999; Gerth et al. 1996; Giannakopoulou and Havelund 2001].

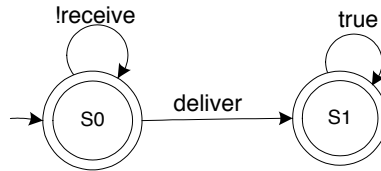


Fig. 8. Automaton for the LTL notation of constraint “precedence(deliver, receive)”.

Note that it is possible that a constraint (and a model) is temporarily violated at some points of execution and satisfied at the end of execution. For example, the “responded existence” template and a (local or global) model containing a “responded existence” constraint between activities “charge” and “pay” (e.g., Figure 3). This constraint specifies that, if “charge” is executed, then “pay” must also be executed before or after “charge”. As long as none of these two activities is executed, the constraint and the model are satisfied. At the moment when “charge” is executed for the first time, the constraint and the model become temporarily violated. This is only a temporary violation because it is still possible to satisfy this constraint and the model in the future, by executing activity “pay”. Indeed, this execution can be considered to be a correct execution only after activity “pay” is executed, because only then the “responded existence” and the model become satisfied. The automata generated from LTL specifications of each constraint and the conjunction of all constraints in the model can be used to monitor states of constraints and the model. For example, Figure 9 shows the automaton created for constraint “responded existence(charge, pay)”. This automaton can indeed be used to monitor the state of this constraint: executing activity “charge” brings the automaton to a non-accepting (denoted by a single border) state “S1”, and only executing activity “pay” again brings the automaton to the accepting (denoted by a double border) state “S2”.

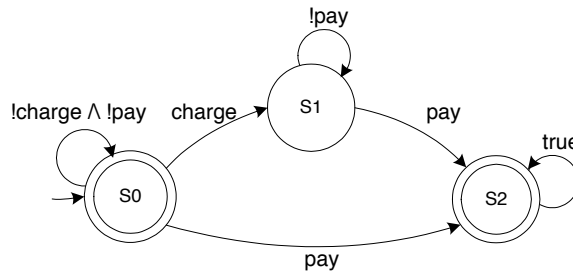


Fig. 9. Automaton for the LTL notation of constraint “responded existence(charge, pay)”.

The automata generated for each constraint and the conjunction of all constraints in a model can be used to: (1) make sure that interacting activities execute only activities that eventually lead to the satisfaction of their local models and the global choreography model, and (2) provide feedback about the current state of the interaction with respect to the satisfaction of local models and the global choreography model and the satisfaction of each constraint from these models (state of one constraint can be monitored using the automaton generated for the LTL notation of that particular constraint).

Note that the word “execution” should not be understood literally when it comes to web services because it is not possible to enforce a certain execution of a service. Instead, the method described in this section could be used as a guideline towards a deadlock-free execution. The automata generated from a conjunction of LTL specifications of constraints ensures that deadlocks do not occur. On the one hand, if activities of services are executed in a way allowed by the automata, a deadlock will not occur. On the other hand, if the automaton is not in an accepting state, interaction constraints are not satisfied and the interaction cannot yet be successfully completed. Note that, enforcing an execution becomes even more unrealistic when it comes to timed constraints, i.e., constraints with deadlines. For example, it is not possible to enforce that a service executes a task within five days.

6.1.2 Verifying Local and Global DecSerFlow Models. The risk of introducing errors in DecSerFlow models is high because it is hard to maintain an overall understanding of many different constraints. Two types of errors can occur in DecSerFlow models due to an unwanted combination of constraints: *dead activities* and *conflicts*.

Figure 10(a) shows a composition of one global and two local models (customer and shop) for the photo shop example. In this case, the customer does not change the global choreography constraints. Due to two “succession” constraints in the global model, each ordered photo and poster will be printed in the shop and delivered. However, the shop developed uses a local model that contains a “not co-existence” constraint specifying that activities “print” and “deliver” exclude each other. This means that the shop either prints or delivers within one order, but never both. Therefore, if activities “photo” or “poster” are executed the choreography cannot be successfully executed because it will not be possible to satisfy the three constraints (i.e., two “succession” constraints and the “not co-existence” constraint). Therefore, it will never be possible to execute activities “photo” and “poster” in the customer service, i.e., these are *dead activities*. Moreover, activity “print” is also a dead activity since it should be executed after activities “photo” or “poster”, which are dead activities.

While it is still possible to execute choreography presented in Figure 10(a) (with ordering albums or empty orders as only possibilities), the example presented in Figure 10(b) is not executable at all, i.e., it contains a *conflict*; this means that the DecSerFlow model is *inconsistent*. As described before, the local shop model handles only album orders (i.e., preventing executions of dead activities “photo” and “poster”) in the global choreography model. The local customer model in Figure 10(b) imposes execution of activity “photo” via constraint “1..*” which makes it impossible for this customer and shop to interact in this global choreography

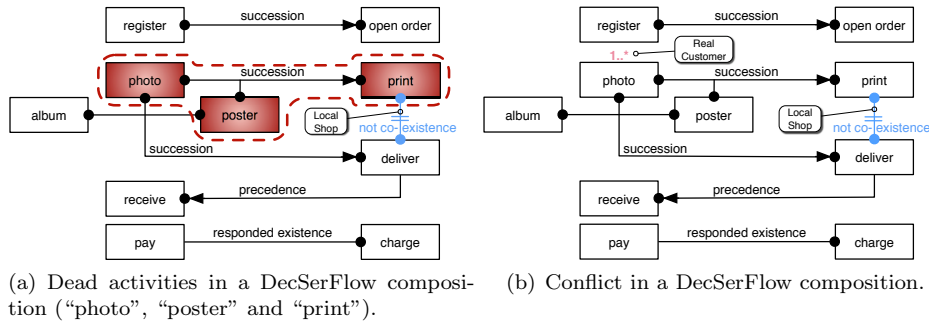


Fig. 10. Verification of DecSerFlow models.

model. The Conflict in the model in Figure 10(b) is caused by the combination of four constraints: the “1..*” constraint, the two “succession” constraints and the “not co-existence” constraint.

6.1.2.1 Detecting Dead Activities and Conflicts. Errors such as the ones just described can easily be detected in DecSerFlow models using the automata generated [Clarke et al. 1999; Gerth et al. 1996; Giannakopoulou and Havelund 2001] from constraints (cf. Section 6.1.1). To verify a combination of several models (i.e., global model and two local models), an automaton is created for a conjunction of all constraints in all models. The generated automata allow for all possible execution traces of the DecSerFlow model at hand. Models are executed by triggering activities via automaton transition, where each transition triggers none, one or more activities. A dead activity is an activity that never appears in any of the execution traces, i.e., there is no transition in the automaton that allows this activity. A conflict is detected when the automata does not parse any trace, i.e., the automata does not contain any state or transition.

DecSerFlow models can be verified against dead activities and conflicts in the DECLARE tool. For a full verification support, DECLARE not only detects errors, but also reports the smallest subset of constraints that causes the error. To achieve this, the tool verifies the conjunctions of smaller groups of constraints by searching through the power set of constraints. Figure 11 shows the verification report generated by DECLARE for model in Figure 10(a): activities “photo”, “poster” and “print” are dead due to the combination of three constraints.

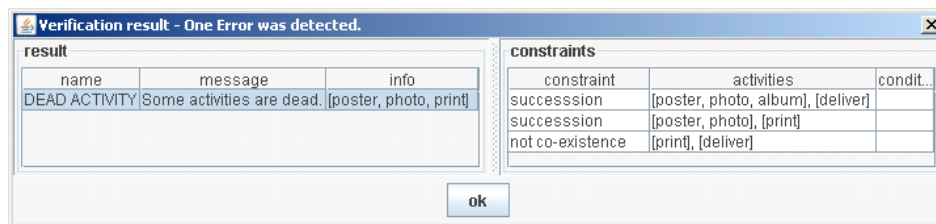


Fig. 11. DECLARE reports the three dead activities of model depicted in Figure 10(a).

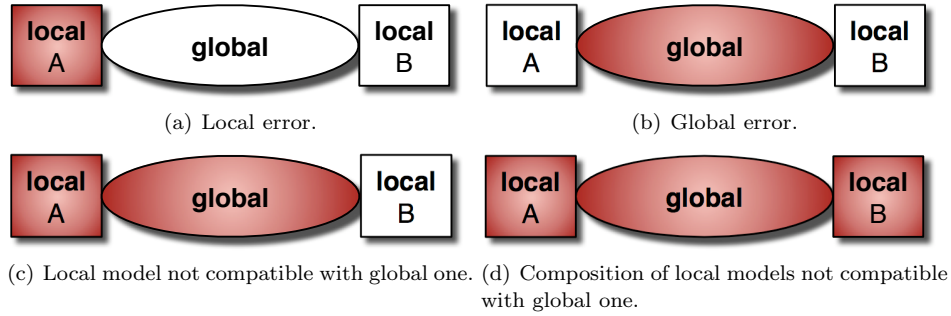


Fig. 12. Implications of errors in DecSerFlow models.

6.1.2.2 *Interoperability Verification.* Errors discovered in verification can imply four types of problems in interoperability of service compositions, as illustrated by Figure 12. First, local service models can be verified (cf. Figure 12(a)). For example, if a conflict is found, the local model has to be fixed before being employed in any choreography, i.e., the local model cannot be employed in any choreography. Second, a global choreography model may contain errors. Global models with conflicts cannot be enacted by a set of parties (regardless of the local models), as shown in Figure 12(b). Third, an error can be discovered in the composition of a local model of one party and the global choreography model (cf. Figure 12(c)). In this case, the party is not compatible with the global choreography rules. Finally, Figure 12(d) shows that an error can be discovered in the service composition, i.e., local models are not compatible with each other with respect to the global model. For example, if this is a conflict error, then the two parties cannot join the choreography. Note that the model presented in Figure 12(b) is an example of a composition where the book shop and the print shop are not compatible with the global choreography model (cf. Figure 12(d)).

6.1.3 *Monitoring DecSerFlow Services.* Besides for execution and verification of models and choreographies, DecSerFlow can be used for service monitoring (i.e., for conformance checking of completed service executions) using the ProM (Process Mining) framework [van der Aalst et al. 2007; van der Aalst et al. 2005]. The ProM framework is an open-source infrastructure for process mining techniques. One of the more than 150 plug-ins offered by ProM is the so-called LTL Checker [van der Aalst et al. 2005]. For each process instance, LTL Checker determines whether an LTL expression holds or not, i.e., given an LTL expression all process instances are partitioned on two classes: compliant and non-compliant. Because each DecSerFlow constraint is represented by an LTL expression, it is possible to use the ProM LTL Checker to assess conformance of a DecSerFlow model in the context of a real log.

6.1.4 *Dynamic Change of DecSerFlow Models.* DecSerFlow models can be changed dynamically, i.e., while they are being executed, by adding and removing activities and constraints. Note that it is not allowed to remove an activity that is involved in a constraint. This problem can be solved in two ways: (1) removing such an activity is rejected, or (2) the activity and all related constraints are removed. In

this way, we prevent situations where a model contains a constraints involving an activity that is not in the model. When it comes to continuing the execution after a dynamic change, DecSerFlow uses the following procedure. First, the automaton is created from a conjunction of all constraints in the new model and attempt is made to ‘replay’ the current execution trace (i.e., a list of all executed activities) on this automaton. If this attempt succeeds, the execution continues using the new automata. If the attempt fails, this means that the current execution trace contradicts to the new model. In this case the dynamic change is rejected and the execution resumes with the old model. Note that the automata generated for DecSerFlow constraints allow for execution of activities that are not involved in constraints. Consider, for example, the automaton for constraint “responded existence(charge,pay)” shown in Figure 9. Although this constraint involves only activities “charge” and “pay”, it allows execution of other activities. For example, transitions “/pay” and “/charge ∧ /pay” allow execution of other activities. This is a useful feature when it comes to dynamic change for two reasons. First, it is possible to remove an activity even after it has been executed, because it will be possible to ‘replay’ the removed activity on the automaton. Second, adding a new activity means that, from that moment, it becomes possible to execute it in the automaton.

6.2 SCIFF Verification

Beside the possibility of extending the DecSerFlow notation with data-related and quantitative temporal constraints, mapping DecSerFlow to SCIFF enables conformance checking, both at run-time or a posteriori, of service execution w.r.t. a DecSerFlow diagram (maintaining a complete support even when considering its extensions), and mining of DecSerFlow models starting from a set of execution traces, previously labeled as compliant or not. Furthermore, SCIFF has been extended to deal also with static verifications (interoperability, discovery of conflicts and dead activities).

6.2.1 Abductive declarative and operational semantics of the SCIFF framework.

Within the logic programming setting, a typical approach is to define both a declarative and operational semantics for logic programs (in our specific case, for SCIFF interaction specifications). Roughly speaking, declarative semantics aims at defining the “meaning” of what is specified, whereas operational semantics describes a general-purpose algorithm capable of concretely exploit the specification. The main advantages of such an approach are that specifications are interpreted in a clear and intuitive way, and that it is possible to prove soundness and completeness of the operational semantics w.r.t. the declarative one, ensuring that its behaviour really respect the intended meaning.

In the SCIFF framework, declarative semantics of interaction specifications is given in terms of an Abductive Logic Program (ALP), whereas the corresponding operational semantics is given in terms of an abductive proof procedure [Alberti et al. 2008], thought for performing different verification tasks.

In general, an ALP [Kakas et al. 1993] 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 of IC is to control the ways elements of A are hypothesised, or “abduced”.

In *SCIFF*, similarly to the general ALP setting, an interaction specification \mathcal{S} is defined by the triple:

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

where:

- KB is the *Social Knowledge Base*, suitable for specifying the static knowledge on interacting entities;
- \mathcal{E} is the set of *abducible predicates*, namely positive expectations (functor \mathbf{E}) and negative expectations (functor \mathbf{EN});
- \mathcal{IC}_S is the set of *Social Integrity Constraints*, used to specify the rules of interaction.

Reasoning in abductive logic programming is usually goal-directed (being G a goal), and it accounts to find an *abductive explanation* 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 ([Kakas and Mancarella 1990; Fung and Kowalski 1997; Denecker and Schreye 1998], etc.). In *SCIFF*, the major difference is that not only the logic program has to be taken into account, but also the (dynamic) set of occurring happened events (which incrementally compose the execution trace). Furthermore, when modeling DecSerFlow constraints in *SCIFF* the goal is not directly exploited; actually, we could consider as goal the conjunction of “existence $_N$ ”, “exactly $_N$ ”, “absence” and “mutual substitution” formulas inside the model: they directly impose expectations about (not) performing certain activities independently from executions (indeed, their formalization consists in rules with a *true* body).

The idea we exploited in the *SCIFF* framework is to adopt abduction to dynamically *generate* the expectations. Expectations are defined as abducibles, and they are hypothesised by the *SCIFF* abductive proof procedure [Alberti et al. 2008], i.e. the proof procedure makes hypotheses about the expected peers behaviour. The set of abduced expectations must satisfy the Integrity Constraints which formalize the choreography.

6.2.2 Conformance checking with *SCIFF*. As sketched in the previous section, the main and original aim of the *SCIFF* framework was not only to provide a suitable and rich language for describing global interactions, but also to equip such a language with different verification capabilities.

The major innovation of *SCIFF*’s declarative semantics w.r.t. classical abductive frameworks is the concept of *fulfillment*, which defines in an intuitive way the relationship between happened events and expectations and makes *SCIFF* suitable for verification. In particular, *SCIFF* is thought to realize the conformance checking task, namely to verify whether a set of interacting entities behave accordingly to the specification.

The basic intuitive idea of conformance in *SCIFF*, which is indeed supported both by a declarative and operational semantics, is to take into account the hypothesized

expectations and link them with the actual peers behaviour, to check whether happened events really adhere to expectations. In particular, a positive expectation requires a corresponding matching happened event, whereas a negative expectation forbid the presence of a matching occurred event. When this is the case, we say that the expectation is *fulfilled*.

EXAMPLE 6.1. *Let us consider the “time-extended” succession formula between “photo” and “deliver” activities shown in Figure 5, whose SCIFF (explicit) formalization is described by Integrity Constraints (3) and (4).*

Let us also consider the following execution trace:

$$\begin{aligned} & \mathbf{H}(\text{performed}(\text{photo}), 16). \\ & \mathbf{H}(\text{performed}(\text{photo}), 19). \\ & \mathbf{H}(\text{performed}(\text{deliver}), 30). \end{aligned}$$

Each of the first two happened events matches with the body of (3), leading to generate two expectations about the execution of the “deliver” activity, whereas the third happened event triggers the “precedence” part of the “succession” formula, generating a backward expectation about a previous execution of activity “photo”:

$$\begin{aligned} \mathbf{EXP} = \{ & \mathbf{E}(\text{performed}(\text{deliver}), T_d) \wedge T_d > 16 \wedge T_d < 40, \\ & \mathbf{E}(\text{performed}(\text{deliver}), T_{d'}) \wedge T_{d'} > 19 \wedge T_{d'} < 43, \\ & \mathbf{E}(\text{performed}(\text{photo}), T_p) \wedge T_p < 30 \wedge T_p > 6\} \end{aligned}$$

All the three expectations actually have a matching happened event in the execution trace, which is therefore evaluated as conformant. In particular, the SCIFF proof procedure will find two different solutions for fulfilling the expectations: one with $T_d/30$, $T_{d'}/30$, $T_p/16$ and one with the same unification for T_d and $T_{d'}$ but having $T_p/19$. Indeed, there are two different executions of activity “photo” capable to satisfy the third expectation.

Let us now consider the same execution trace but containing the execution of “deliver” at time 42. In this case, the history is evaluated as non conformant: deadline about the delivery is not respected for the first execution of activity “photo”, and the first expectation does not have any corresponding happened event.

The formal definition of fulfillment follows the above described intuition.

Definition 6.2 Fulfillment. Given an execution trace \mathbf{HAP} , a set of expectations \mathbf{EXP} is *fulfilled* by \mathbf{HAP} if and only if for all (ground) terms p :

$$\forall \mathbf{E}(p) \in \mathbf{EXP} \Rightarrow \mathbf{H}(p) \in \mathbf{HAP} \quad \forall \mathbf{EN}(p) \in \mathbf{EXP} \Rightarrow \mathbf{H}(p) \notin \mathbf{HAP} \quad (7)$$

Otherwise, \mathbf{EXP} is *violated* by \mathbf{HAP} .

Starting from the concept of fulfillment, it is now possible to give a formal definition of *conformance*⁶.

Definition 6.3 Conformance. Given an execution trace \mathbf{HAP} and an interaction specification \mathcal{S} , \mathbf{HAP} is conformant to \mathcal{S} if and only if there exists an E-consistent

⁶For the sake of simplicity, we have omitted the goal, which is considered to be *true*.

⁷ abductive explanation **EXP** such that definition 6.2 holds.

Intuitively, conformance ensures that all positive expectations have indeed a corresponding happened event, and that no forbidden event occurs.

The conformance checking task is concretely performed by the *SCIFF* proof procedure, which has been proven sound and complete w.r.t. its declarative semantics [Alberti et al. 2008]. The proof procedure is a transition system which extends the well known IFF proof-procedure [Fung and Kowalski 1997], by dealing with confirmation of hypothesized expectations and with dynamic occurring happened events. The latter feature makes *SCIFF* able to monitor the behaviour of interacting entities both a posteriori, by analyzing a complete execution trace of the interaction, or at run-time, by considering occurred events as soon as they happen and waiting if expectations can be still confirmed.

The proof procedure has been wrapped into the SOCS-SI [Alberti et al. 2006] tool, which is capable to accept different event sources and to visualize step-by-step the status of the proof, showing pending, fulfilled and violated expectations. SOCS-SI typically works in a run-time setting: it is able to dynamically acquire happened events and, by exploiting the proof procedure, to raise violations as soon as they happen. Figure 13 shows a screenshot of the tool, dealing with the violation of example 6.1.

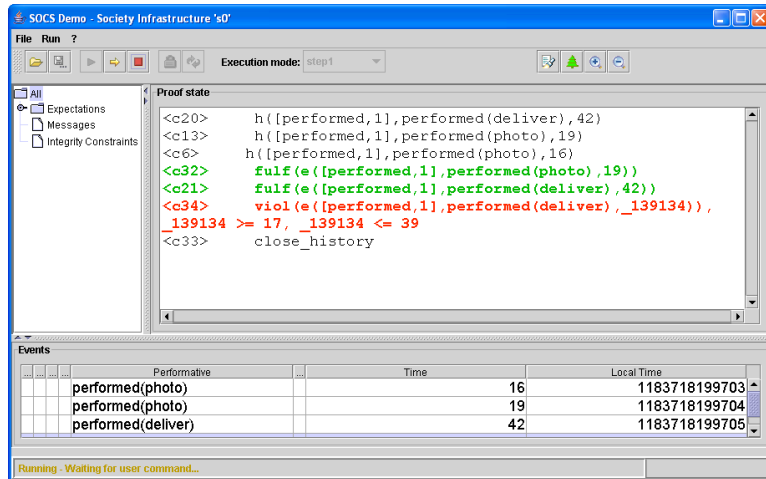


Fig. 13. Screenshot of the SOCS-SI tool

Furthermore, *SCIFF* is being embedded into a ProM plug-in called *SCIFF-Checker*, to the aim of classifying process instance w.r.t. declarative *SCIFF* rules, in the style of LTL Checker.

⁷A set of expectations **EXP** is E-consistent if and only if no event is expected to happen and not to happen at the same time, i.e. if, for each (ground) term p :

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

6.2.3 Interoperability and Detection of Conflicts and Dead Activities in SCIFF.

The SCIFF proof procedure has been extended to deal also with verification of properties (g-SCIFF [Alberti et al. 2005]); the term “property” means in this context a specific domain dependent property. The basic underlying idea of g-SCIFF is to consider the desired property as the initial goal, and to apply fulfillment of a positive expectation (see definition 6.2) by checking whether it already has a matching happened event and, if it is not the case, by “hypothesizing” it. Hence, it operates by simulating a sequence of intensional happened events which fulfill the positive expectations and by checking that negative expectations are not violated. Simulated events are intensional in the sense that they are partially specified (i.e., they may contain variables). If the given property can be actually satisfied, g-SCIFF also returns as proof a partially specified execution trace capable to fulfill both the Integrity Constraints and the property.

The problem of detecting a conflict in a DecSerFlow model can be then treated as the problem of finding a successful g-SCIFF derivation for the goal *true*, by considering the Integrity Constraints which formalize the model. If it is not the case, then there does not exist any possible execution trace for such a model, and therefore it is a conflicting one.

EXAMPLE 6.4. *Let us consider the DecSerFlow model of Figure 10(b). Its corresponding (explicit) SCIFF formalization is given by the Integrity Constraints shown in Table IX, together with the following rules (for the sake of simplicity, we omit the “init” constraint):*

$$true \rightarrow \mathbf{E}(\text{performed}(\text{photo}), T_{ph}). \quad (8)$$

$$\mathbf{H}(\text{performed}(\text{print}), T_p) \rightarrow \mathbf{EN}(\text{performed}(\text{deliver}), T_d). \quad (9)$$

$$\mathbf{H}(\text{performed}(\text{deliver}), T_d) \rightarrow \mathbf{EN}(\text{performed}(\text{print}), T_p). \quad (10)$$

Rule 8 is used to model the “existence” of at least one “photo” activity, whereas rules 9 and 10 impose the “not coexistence” between “print” and “deliver”.

To verify if the model contains conflicts, we simply invoke g-SCIFF and check if such a proof fails. The proof procedure starts by applying rule 8 (since its body is *true*), generating an expectation about the execution of activity “photo” (at any time). Such an expectation becomes an happened event, which triggers Integrity Constraints G_{2a} and G_{3a} , generating two forward expectations about activities “print” and “deliver”. g-SCIFF selects now the expectation about printing the photo, transforming it to an happened event. Such an event matches with the antecedent of rules G_{2c} and (9). Let us consider now the latter rule, which leads to generate a negative expectation about the delivery; by explicitly showing only the pending expectations (i.e., expectations which still do not have a matching happened event) the status of the proof is the following:

$$\mathbf{HAP} = \{\mathbf{H}(\text{performed}(\text{photo}), T_{ph}), \mathbf{H}(\text{performed}(\text{print}), T_p) \wedge T_p > T_{ph}\}$$

$$\mathbf{EXP} = \{\mathbf{E}(\text{performed}(\text{deliver}), T_d) \wedge T_d > T_{ph}, \mathbf{EN}(\text{performed}(\text{deliver}), T_d)\}$$

The set **EXP** is not E-consistent: after time T_{ph} , the “deliver” activity is expected to happen and not to happen at the same time. As a consequence, it is impossible to fulfill the delivery expectation, and no successful proof can be found by g-SCIFF. This attests that actually a conflict is present in the model.

Detecting dead activities is a very similar task. To verify whether an activity is dead or not, it is sufficient to run g-SCIFF by giving as goal the execution of the activity, at any time. Failure of the proof means that it is impossible to perform the activity under study, namely that it is actually a dead activity. From the DecSerFlow point of view, giving as goal the execution of an activity at any time can be modeled by attaching to the activity a “1..*” cardinality constraint. Hence, proving that an activity is dead is the same as proving that the DecSerFlow model, augmented with such an existence formula, contains conflicts.

As an example, let us consider the DecSerFlow diagram of Figure 10(a) and the task of verifying that activity “photo” is dead. By adding the “1..*” cardinality constraint on the “photo” activity, we obtain (except for the “init” constraint on activity “open order”) the model of Figure 10(b), which has been proven to contain conflicts in example 6.4; therefore, “photo” is a dead activity.

Algorithm 1 summarizes how the discovery of dead activities can be addressed by g-SCIFF.

Input: \mathcal{S}_M , SCIFF formalization of the DecSerFlow model \mathcal{M}
Output: \mathcal{D} , the set of dead activities
 $\mathcal{D} \leftarrow \emptyset$;
foreach *Activity* $A \in \mathcal{M}$ **do**
 $\mathcal{S}'_M \leftarrow \mathcal{S}_M \cup \text{existence}(A)$;
 if $\text{call}(g\text{-SCIFF}(\mathcal{S}'_M))$ *fails* **then**
 $\mathcal{D} \leftarrow \mathcal{D} \cup A$;
 end
end

Algorithm 1: Detection of dead activities with g-SCIFF.

Let us finally deal with the interoperability problem. There are many different definitions of interoperability [Chopra and Singh 2006; Baldoni et al. 2006], which mainly differ by the “degree of similarity” they require between the local and the global models. DecSerFlow leads to the definition of a very weak interoperability: as described in section 6.1.2 a local model is considered interoperable w.r.t. a global one if the composition of the two models admits at least one execution trace, i.e. if the composition is conflicts free. It is clear that such a verification does not ensure that the two models completely overlap, nor that if a local model is interoperable w.r.t. a global model it will correctly comply with any other local model which has been proven interoperable (see Figure 12). Adopting a proof-theoretic approach like the one of SCIFF leads to face this kind of weak interoperability by simply composing the formalizations of models under study (i.e., adopting the “implicit” approach, by joining the knowledge bases of each specific model) and using g-SCIFF for testing conflict freeness on the composite model.

6.2.4 Mining of DecSerFlow specifications by using SCIFF as an intermediate language. An important advantage of adopting a logic programming representation (like SCIFF), relies in the possibility to apply on it all the algorithms and techniques developed within the logic programming setting. More specifically, it makes possible to apply Inductive Logic Programming (ILP) [Muggleton and De Raedt

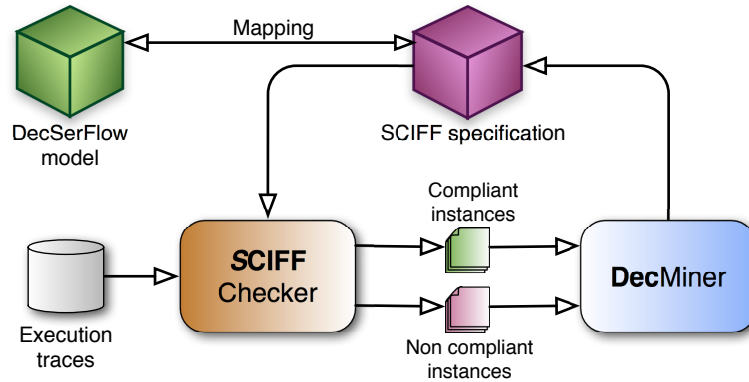


Fig. 14. The “mining-checking” cycle, in which *SCIFF* is used as the source/target language and DecSerFlow as the representation notation.

1994] techniques for learning declarative models from examples and background knowledge.

Such a possibility has been concretely exploited by adapting the system ICL [De Raedt and Van Laer 1995] to the problem of learning *SCIFF* constraints. In [Lamma et al. 2007], the authors cast the problem of mining declarative specifications of processes as a learning from interpretation problem. In particular, they consider the discriminant problem that is solved by ICL, which starts by considering a set of positive and negative interpretations and aims to learn a clausal theory that discriminates the two. In their case they assume to have a set of compliant and non-compliant process execution traces and find a *SCIFF* theory that accurately classifies a new trace of the process as compliant or not.

This learning process has been extended in [Lamma et al. 2007] to learn DecSerFlow models. Here, the mapping of DecSerFlow onto *SCIFF* presented in this work is exploited in the opposite way: some of the learned Integrity Constraints can be in fact considered as the *SCIFF* representation of DecSerFlow formulas, especially if the language bias of the learning algorithm is opportunely tuned. In this context, *SCIFF* is therefore used as an intermediate language for learning DecSerFlow models starting from a set of execution traces, previously labeled as compliant or not. A tool called DecMiner is actually being implemented inside the ProM framework to cover all the phases of such a mining process.

6.2.5 The “mining-checking” cycle. Having shown the feasibility of using the *SCIFF* language as the core element of both a framework for conformance checking and an algorithm to mine declarative process models, we may put together the two settings to realize a “mining-checking” cycle, shown in Figure 14.

From one side we may start from a set of positive and negative execution traces and apply DecMiner to mine a *SCIFF* theory; such a theory can then be partially rendered in a graphical way by applying the inverse mapping onto DecSerFlow, and used in conjunct with the *SCIFF* proof procedure to classify new execution traces.

From the other side, the modeler may design a DecSerFlow diagram for classifying a set of process execution traces; the actual classification can be performed by

automatically mapping the diagram onto *SCIFF* and checking conformance of the different logs. The classified logs may be finally used as input of *DecMiner*, to induce a new *SCIFF* theory (and thus a new *DecSerFlow* model); if the language bias is opportunely tuned, such a model could be noticeably different from the initial one, hence expressing the same classification criterion by shifting the point of view.

7. THE FRAMEWORK IN USE

Having introduced the main components of the service choreographies framework sketched in Figure 7, an important issue is to evaluate its usability, for what concerns both the *DecSerFlow* language itself and performances of the verification tasks. We will try to briefly assess the usability of the language by considering the Cognitive Dimensions framework [Green 1989], and then evaluate verification techniques by means of some benchmarks and summarizing current available tools.

7.1 Usability of the language

Cognitive Dimensions [Green 1989] are a useful tool to subjectively assess the usability of languages and notations in an easy-to-comprehend way. They have been applied to a broad range of programming languages and environments/editors, also visual [Green and Petre 1996]. Although a deep and extensive analysis of *DecSerFlow* from the end-users viewpoint has not yet been carried out, we will try to briefly review its usability in terms of some Cognitive Dimensions (whose definition is briefly listed in table XI⁸).

Closeness of mapping	Closeness of representation to domain
Abstraction	Types and availability of abstraction mechanisms
Consistency	Similar semantics are expressed in similar syntactic forms
Hidden dependencies	Important links between entities are not visible
Premature commitment	Constraints on the order of doing things
Progressive evaluation	Work-to-date can be checked at any time

Table XI. Some Cognitive Dimensions.

The main strength of *DecSerFlow* relies on the *closeness of mapping* between the notation and the problem of capture choreography constraints: it provides various expressive *abstractions* to constrain activities execution in many different ways, overcoming both over-constraining and over-specification issues. *DecSerFlow* diagrams can range from classical procedural models (by only using the *chain succession* formula, which is the *DecSerFlow* counterpart of the sequence relationship in procedural languages) to purely declarative/loosely-coupled ones (e.g. by imposing constraints such as the responded presence, or by modeling the forbidding of activities with negation formulas). Such a flexibility can be summarized by stating that *DecSerFlow* follows an *open* approach: interacting services can freely behave

⁸See <http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf>.

where not explicitly constrained⁹. In our view, such an approach is of key importance when modeling service choreographies, which are open in nature: they should impose only the strictly necessary rules of collaboration, allowing as much as possible different concrete services to interoperate. This is a fundamental requirement for satisfying the re-usability principle of Service Oriented Architecture.

DecSerFlow not only provides a plethora of different abstractions: it is also *abstraction tolerant*, supporting the modeler in the definition of new constraint templates.

Another valuable feature of DecSerFlow is the *consistency* of its core formulas: they have a representation which coherently combine only the few basic intuitive concepts shown in Table XII. For example, the representation of “succession” relationships can be easily inferred: both semantics and representation of this kind of formula is determined by combining/overlapping the semantics/representation of the corresponding “response” and “precedence” ones.

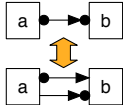
concept		notation
unary formula		cardinality constraints a lá UML
relationship source		•
negation		
temporal ordering		→
relationship’s strength	normal	—
	alternate	≡
	chain	≡≡
succession representation		response + precedence formulas, e.g.: 

Table XII. Basic DecSerFlow graphical elements and their corresponding meaning.

Even though DecSerFlow combines simple concepts, rendered in a consistent way, when the complexity of models increases their readability would quickly be compromised. The semantics of a DecSerFlow model is determined as the conjunction of its formulas: the user is driven to adopt a non-structured approach to modeling, avoiding *premature commitments*; but unfortunately, from the other side the overall meaning tends to become unclear: because of the interplay between the different formulas, many *hidden dependencies* among activities are implicitly introduced.

To better clarify the problem, let us consider the simple example of Figure 15.

Suppose that activity “a” is executed; this leads to forbid further executions of “a” (due to the absence formula attached to it), but implicitly also to forbid the execution of both “b” and “c”. Actually, either by executing “b” or “c” activity “a” should be eventually executed afterwards, but this would be impossible, because it cannot be performed anymore. This situation arises from the interplay between the “absence₂” and “response” formulas, which introduces an hidden “negation response” in the diagram (hidden relationships are shown in Figure 15 as dashed

⁹Note that, indeed, the “chain succession” expresses a “closed” relationship, because it completely fixes the sequencing of involved activities.

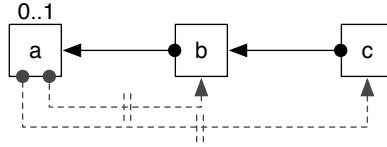
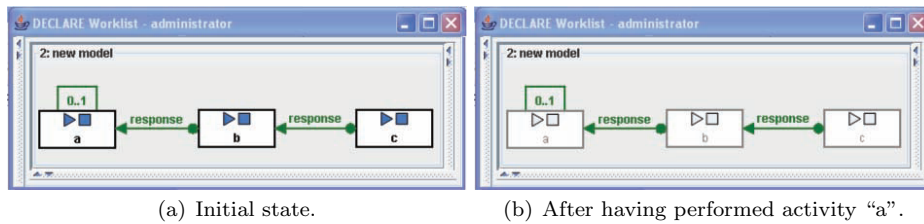


Fig. 15. Hidden dependencies in a simple DecSerFlow model.

connections). In complex cases, such an interplay could lead to produce diagrams containing dead activities or even inconsistent models.

Anyway, the possibility of mapping DecSerFlow onto different logic-based languages shown in this article comes in support: DecSerFlow models can be verified at any time to ensure consistency and discover dead activities, satisfying the important Cognitive Dimension of *progressive evaluation*. Not only, the modeler can also exploit the enactment tool in order to simulate interactions and see how the state of the different formulas evolves when executing activities.

As shown in Figure 16, the enactment of the simple DecSerFlow diagram shown in Figure 15 leads to blocking the execution of all activities after having performed “a” once, giving an explicit feedback about the hidden dependencies of the model.



(a) Initial state.

(b) After having performed activity “a”.

Fig. 16. Enactment of the simple example shown in Figure 15.

7.2 Quantitative evaluation of the proposed techniques

In order to assess the usability of the framework, a key point is to evaluate performances and scalability of the presented techniques. The performance issues related to the LTL-based notation are presented in Section 7.2.1, while Section 7.2.2 discusses the performance of the SCIFF notation. We will mainly focus on static verification, i.e. conflict detection and discovery of dead activities (which are the most expensive one for both approaches)¹⁰.

7.2.1 Performance of the LTL-based notation of DecSerFlow. When it comes to the LTL-based notation of DecSerFlow, performance is an issue related to the complexity of models with a large number of constraints. Due to the use of LTL for constraint specification, performance dramatically decreases when the number and complexity of constraints in DecSerFlow models rises.

¹⁰Note that, for the sake of efficiency and in order to avoid some trivial loops, some of the SCIFF rules are, in this case, transformed onto an equivalent representation, which simply leads to an a-priori pruning of some useless choice-points

As described in Section 6.1, an automaton is generated for a *conjunction of formulas of all constraints* (i.e., the so-called “conjunction formula”) in an DecSerFlow model. Because this automaton represents exactly all correct executions of the model, the automaton is used for the computer-supported execution and fully automated verification of DecSerFlow models based on LTL (cf. sections 6.1.1 and 6.1.2). Since the automata generated for an LTL formula is exponential in the size of the formula [Clarke et al. 1999; Gerth et al. 1996; Giannakopoulou and Havelund 2001; Latvala 2003; Demri and Schnoebelen 1998; Demri et al. 2006; Flum and Grohe 2006], the time needed for generating these automata becomes very long for big LT formulas. This can cause various problems in the context of DecSerFlow. For example, generating such automaton for a DecSerFlow model with many complex constraints may be extremely slow.

There are two possible causes of this problem. First, the more constraints there are in a model, the larger the “conjunction” formula for the model will be. Second, as shown in Appendix A.2, various DecSerFlow templates have different LTL formulas. For example, the LTL formula for the “succession” template is significantly more demanding from a computational point of view than the formula for the “existence” template (both formalizations are presented in Table V).

Consider, for example, the global choreography model presented in Figure 3 on page 12. Loading a new instance of this model in DECLARE takes approximately *17 minutes* on a computer with a Pentium 4 processor of 3GHz and 1.49GB of RAM using Microsoft Windows XP Professional version 2002. If the “succession” constraint between activities “album”, “photo”, “poster” and “deliver” is removed from the original DecSerFlow model, then generating the automaton for the “conjunction” formula in DECLARE on the same computer takes approximately *30 seconds*. Obviously, the size of the LTL formula for this triple-branched “succession” constraint dramatically increases the time to construct the automaton for the “conjunction formula”. Naturally, DecSerFlow models with only few simple constraints perform much better. For example, creating an automaton for the conjunction of all constraints for the two local models shown in figures 4(a) and 4(b) on page 14 takes approximately *200 milliseconds* and *100 milliseconds*, respectively.

The efficiency problem described above can occur at several points. First, when initiating a computer-supported execution of a DecSerFlow model (cf. Section 6.1.1), an automaton is generated for the “conjunction formula”, which can cause the initiation to take a long time. Second, the same automaton is created during verification of single DecSerFlow models and during interoperability verification in order to identify possible errors (cf. Section 6.1.2), which may cause the error detection to last too long. Moreover, during verification an automaton is generated for combinations of constraints in order to identify the cause of error, which can cause the verification to be even more time-consuming.

Existence of *errors* (i.e., dead activities or conflicts) in a DecSerFlow model can significantly *decrease* the time needed for the generation of the automaton for the “conjunction” of all constraints because the automaton then “represents” a model with less possibilities (i.e., less possible executions). Consider, for example, the DecSerFlow model for the global choreography shown in Figure 3, for which it takes approximately *17 minutes* to create the automaton. When errors are introduced

in this model (e.g., dead activities shown in Figures 10(a) and a conflict shown in Figure 10(b) on page 30), the performance increases. Indeed, the automaton for the “conjunction” formula is created within approximately *14 seconds* and *100 milliseconds* for the two DecSerFlow models shown in figures 10(a) and 10(b) on page 30, respectively.

Note that the LTL Checker [van der Aalst et al. 2005] (cf. Section 6.1.3) does not use the automata for conformance checking of a DecSerFlow model in the context of a real log, and, therefore, does not encounter the above described performance problem.

7.2.2 Performance of the SCIFF notation of DecSerFlow. SCIFF and model checking face the static verification problem in a complementary way: with LTL the verification consists in first building a-priori the automaton of the entire DecSerFlow model, and then checking the reachability of a termination state on such an automaton; SCIFF instead adopts a generative approach, i.e. it directly employs model’s constraints trying to dynamically build a proof in a depth-first way. Such a proof consists in an execution trace which satisfies all constraints (this ensure that at least one possibility to execute the model actually exists). As a consequence, also advantages and lacks are complementary:

- SCIFF scales very well in the number of constraints in the model, whereas LTL suffers of the state-explosion problem. Furthermore, SCIFF uses for verification only the strictly necessary rules. Actually, consistency verification typically depends on the presence of existence (“existence_N” and “exactly_N” in particular) and “mutual substitution” formulas, because they directly impose the necessary execution of some activities, triggering in turn consistency on relationships attached to such activities, and so on. While LTL builds the whole automaton without taking into account such a peculiar feature, SCIFF starts by considering just these kind of formulas (Figure 17 sketches how SCIFF deals with inconsistency of the model shown in Figure 10(b)). The extreme case is the one in which a model does not contain any “existence” nor “mutual substitution” formula, like in the running example shown in Figure 3: SCIFF immediately evaluates it as conflicts free, independently of its size, because the void execution trace is accepted.
- From the other side, a distinguishing feature of LTL is its capability of handling “infinite” systems, namely models which contain loops¹¹; being SCIFF a generative approach, it is instead not able to treat looping models, because it loops as well. A naive solution to this problem is to change SCIFF’s search strategy in the space of execution traces, by e.g. adopting a *bounded iterative deepening* approach. Obviously, a bounded search strategy would undermine completeness; as a consequence, we will study the insertion of loops-detection algorithms in the proof procedure (note that this problem has been deeply investigated in the field of logic programming [Shen et al. 2003]). It is worth noting that such a problem only affects SCIFF when used in a generative manner: when performing conformance verification of execution traces, reasoning is actually driven by events

¹¹Such kind of DecSerFlow models are evaluated as inconsistent, because they do not eventually terminate (and this would be inconvenient for a choreography or a process model).

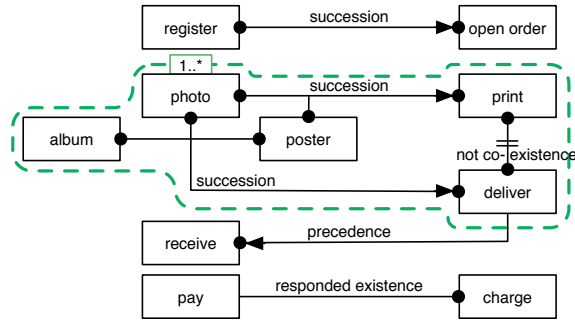


Fig. 17. Consistency verification of the diagram shown in Figure 10(b) with SCIFF: only the subset of formulas “triggered” by the existence of activity “photo” is used; presence of conflicts is detected in 350 milliseconds.

	Formulas/activities	Time (sec.)
1	3	0.00
2	7	0.01
3	15	0.01
4	31	0.02
5	63	0.04
6	127	0.10
7	255	0.34
8	511	1.04
9	1023	3.64
10	2047	14.01
11	4095	56.85
12	8191	249.77
13	16383	1100.26
14	32767	6194.99

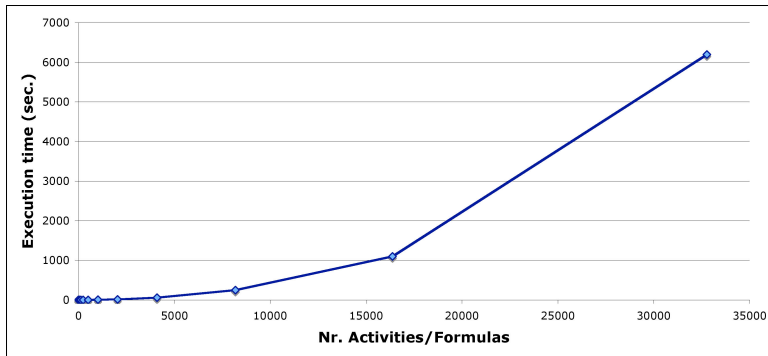
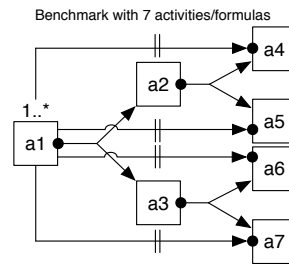
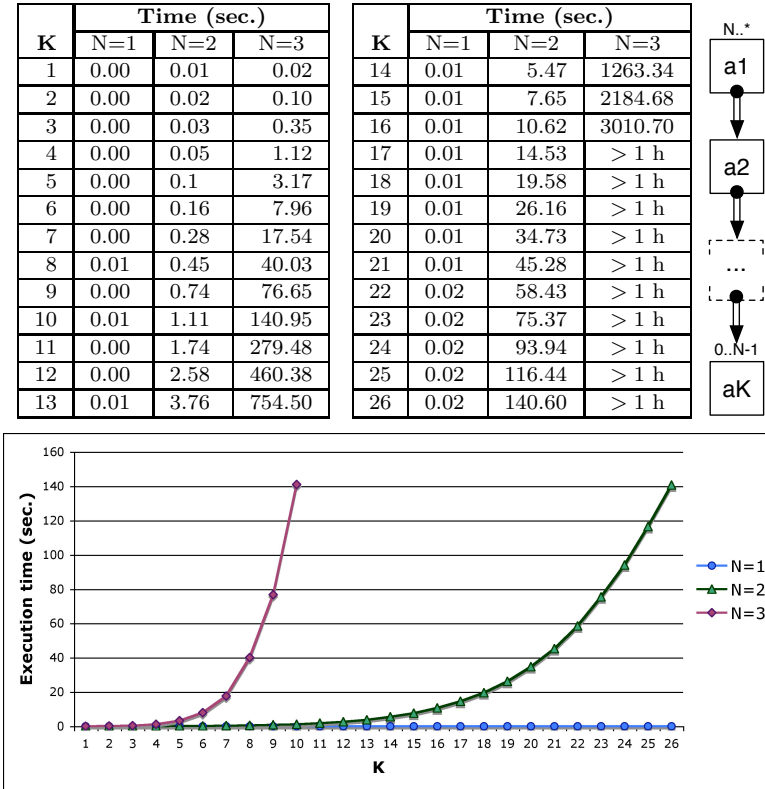


Fig. 18. Results of the branching response inconsistency benchmark.

contained in the log, and therefore SCIFF will not loop.

To study the scalability of SCIFF when discovering conflicts/dead activities, we have tested it on some simple yet relevant benchmarks, which involve different formulas. All benchmarks deal with inconsistent models, to the aim of testing

Fig. 19. Results of the *alternate response inconsistency* benchmark.

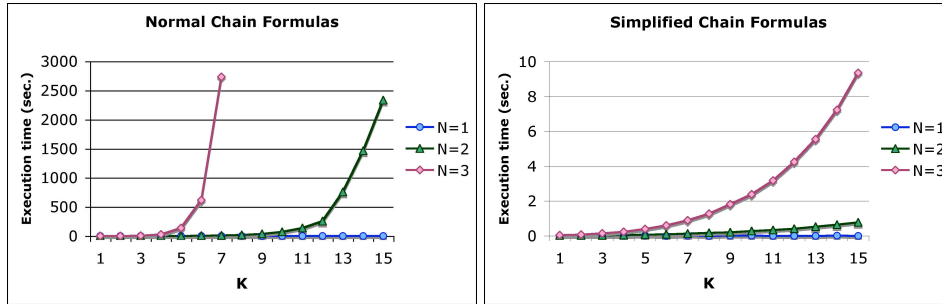
SCIFF in the worst case: to prove that a model contains a conflict, SCIFF has to explore the entire search space. Experiments have been performed on a MacBook Intel CoreDuo 2 GHz machine.

The first benchmark aims to evaluate the scalability of the approach when “branching responses” are used, i.e. different alternatives are present in the model. The structure of the model is the following (Figure 18 graphically shows the benchmark in case of 7 activities). One activity, namely a_1 , is expected to be executed at least once. After a_1 , either one between two activities should be executed, and so on. “Branching responses” follow one another until a “frontier” is reached; all the activities belonging to this “frontier” have an outgoing “negation precedence” formula w.r.t. a_1 , and therefore no path can be executed without leading to a conflict.

Figure 18 summarizes SCIFF’s behaviour when the number of activities (which is the same as the number of formulas, except from the “existence_1” on the first activity) increases. Such experiments attests that SCIFF scales very well: it is able to detect inconsistency of a model containing 4095 formulas in less than one minute.

The two next benchmarks aim at evaluate SCIFF’s behaviour respectively in case of “alternate” and “chain response” formulas, especially when they are combined with existence formulas which impose more executions of the same activity. The

NORMAL				SIMPLIFIED			
K	Time (sec.)			K	Time (sec.)		
	N=1	N=2	N=3		N=1	N=2	N=3
1	0.00	0.01	0.03	1	0.00	0.01	0.04
2	0.00	0.02	0.35	2	0.00	0.02	0.07
3	0.00	0.09	4.04	3	0.00	0.03	0.15
4	0.00	0.34	27.12	4	0.00	0.05	0.24
5	0.01	1.13	134.77	5	0.10	0.07	0.39
6	0.00	3.02	616.94	6	0.00	0.09	0.60
7	0.00	7.63	2733.64	7	0.01	0.13	0.89
8	0.00	17.10	> 1 h	8	0.00	0.17	1.27
9	0.01	37.09	> 1 h	9	0.01	0.21	1.81
10	0.01	71.73	> 1 h	10	0.01	0.28	2.38
11	0.01	137.03	> 1 h	11	0.00	0.33	3.18
12	0.00	256.48	> 1 h	12	0.00	0.41	4.25
13	0.01	756.34	> 1 h	13	0.00	0.53	5.55
14	0.01	1460.47	> 1 h	14	0.01	0.65	7.24
15	0.01	2332.65	> 1 h	15	0.00	0.78	9.36

Fig. 20. Results of the *chain response inconsistency* benchmark.

second (third resp.) benchmark impose at least N executions of an activity, which is source of a sequence of K activities connected by “alternate” (“chain” resp.) “response” formulas; the last activity of the sequence is subject to an absence formula which imposes at most $N - 1$ executions (the model is therefore inconsistent, because also such last activity should be performed at least N times).

Actually, discovering conflicts in alternate/chain response formulas when only one execution of the source activity is imposed reduces to the case of simple responses. This is attested also by SCIFF: both for “alternate” (Fig. 19) and “chain response” (fig. 20) formulas, when $N = 1$ it answers almost immediately. When N increases, verification becomes more difficult; for “alternate responses” (but similarly also for “chain” formulas), this is due to the fact that more executions of the source activity trigger the *interposition* part of the formula, imposing and propagating a huge amount of temporal constrains.

Finally, note that performances in case of “chain” formulas are slower because its formalization in SCIFF, which contains a time-constrained forbidding of *all* events, is rather difficult to be handled (see section 5). If we restrict ourselves to the basic DecSerFlow core formulas, for performing consistency verification we can adopt, without losing generality, a simplified version of “chain response”,

which states that the target should happen at the immediate next integer w.r.t. the source execution time ¹². By adopting such a simplification, a dramatic speed-up of verification times is experienced (see Figure 20 to have an overview about a comparison between the two formalizations).

Summarizing, *SCIFF* is able to verify in acceptable times even complex DecSerFlow models. Its performances become slower when the model contains both existence formulas with an high repetition value and many “strict” relationships (such as “chain responses”). However, it is rather uncommon to find choreography models in which a certain activity is a-priori constrained by an existence formula to be executed many times; furthermore, when the modeler adopts many different strict relationships in the same diagram, she is breaking DecSerFlow philosophy, which is to develop loosely-coupled choreographies: the right choice would probably be to adopt a more classical procedural language (like e.g. BPMN).

The interested reader is referred to [Montali et al. 2008] for further experimentations/benchmarks, together with a comparison with explicit and symbolic model checking. The results obtained in [Montali et al. 2008] confirm that *SCIFF* is clearly superior to explicit model checking when statically verifying DecSerFlow models, and that it outperforms symbolic model checking in many cases. Also in these benchmarks, the global trend is that *SCIFF* scales very well in the number of constraints, while experiences more difficulties when existence formulas with high repetition values are introduced in the model. However, establishing a precise relationship between the size of a DecSerFlow model and the performance of *SCIFF* is not a trivial task: as we have seen in the presented benchmarks, the performance is affected not only by the number of constraints, but also by the interplay between such constraints. For example, *SCIFF* answers immediately when testing conflict-freedom on models containing no existence constraint, independently from the size of the model.

SCIFF verification times are even faster when performing conformance checking of execution traces; just to give an intuition about performances, we have exploited it to analyze real execution traces of a clinical screening process [Chesani et al. 2007]: a *SCIFF* specification containing 12 rules (with branches and constrains on both execution times and content data) has been tested on 1950 execution traces, ranging from 1 to 18 events, in approximately 12 minutes.

Finally, the interested reader is referred to [Lamma et al. 2007; Chesani et al. 2009] for a preliminary quantitative evaluation related to mining DecSerFlow specification from labeled execution traces, using *SCIFF* as an intermediate format.

7.3 Tool support

As far as now, different tools can be exploited to verify DecSerFlow models; some of them are well-established and some others are still under testing. One of our main ongoing objectives is to integrate such tools in order to cover all the different parts of the framework for the specification and verification of service choreographies depicted in Figure 7.

Some of the tools have been developed as part of ProM. ProM is an open source

¹²When times are not quantitatively constrained, we can map the concept of “next state” to the one of “next integer”.

framework (under the Common Public License, CPL) for process mining, available at <http://www.processmining.org>; it is plug-able, i.e., people can plug-in new pieces of functionality. Beside a plethora of mining techniques, ProM offers a wide range of plug-ins related to model transformations and model analysis (e.g., verification of soundness, analysis of deadlocks, invariants, reductions, etc.).

Figure 21 sketches current available tools together with their relationships. A brief description of them follows.

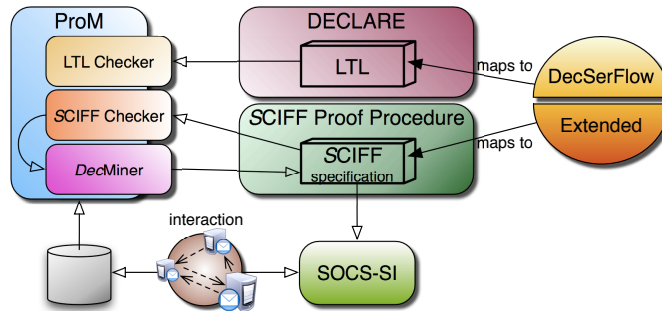


Fig. 21. Tools for the specification and verification of DecSerFlow models.

7.3.1 *DECLARE*. DECLARE [Pesic et al. 2007] is the main tool for editing and enacting DecSerFlow models. It is mainly composed by two parts: an editor, supporting the user both in the development of new graphical models as well as in the definition of new declarative constraints (by specifying their graphical appealing and the underlying LTL formalization); an enactment module, capable to concretely execute DecSerFlow models giving a step-by-step feedback about constraints state. The editor also provides support for checking the correctness of designed models, by identifying conflicts and discovering dead activities. DECLARE can be downloaded from <http://is.tm.tue.nl/staff/mpesic/declare.htm>.

7.3.2 *LTL Checker*. The LTL Checker [van der Aalst et al. 2005] is a ProM plug-in for performing process analysis by exploiting LTL. It offers an environment to provide parameters for predefined parameterized LTL expressions and check these expressions (as properties related to activities, data, human resources and time) with respect to some event log in MXML [van Dongen and van der Aalst 2005] format. Currently, we do not yet provide a direct connection between the DecSerFlow editor (tool DECLARE) and the ProM LTL checker. Although it is possible to export DecSerFlow templates, constraints and models to LTL checker and check conformance in ProM, it is not yet possible to visualize violations on the DecSerFlow editor in DECLARE. Such a connection is matter of ongoing work. The LTL Checker can be downloaded together with Prom from <http://www.processmining.org>.

7.3.3 *SCIFF Proof Procedure*. The SCIFF Proof Procedure, downloadable from <http://lia.deis.unibo.it/research/sciff/>, is a CHR-based implementation

[Alberti et al. 2005] of *SCIFF*'s operational semantics¹³. Its main application is to verify conformance of a set of happened events w.r.t. a *SCIFF* interaction specification, by checking whether positive (negative resp.) expectations indeed have (not have resp.) a corresponding matching happened event. By setting some options, *SCIFF* can also be configured to work in a generative manner, which is the basis for performing conflicts detection and discovery of dead activities in DecSerFlow models. An additional module containing useful predicates for verifying DecSerFlow models, as well as the complete set of *SCIFF* projects which have been used to quantitatively evaluate the framework, can be downloaded from <http://lia.deis.unibo.it/research/climb/>. We are currently developing a graphical editor which supports extended DecSerFlow models (i.e., models containing complex branching and time-annotated relationships) and automatically translates them onto *SCIFF* specifications.

7.3.4 SOCS-SI. SOCS-SI [Alberti et al. 2006] wraps the *SCIFF* proof procedure into a java-based tool, to the aim of exploiting conformance verification at run-time. In particular, it can be interfaced with different event sources and supports a step-by-step visualization of the proof status, showing pending, fulfilled and violated expectations (see Figure 6.1). Violations are raised as soon as they happen, because temporal aspects (deadlines in particular) are taken into account. To download the software, visit http://www.lia.deis.unibo.it/research/socs_si/.

7.3.5 SCIFF-Checker. Conformance verification of past execution traces w.r.t. a *SCIFF* specification has been integrated inside ProM as an analysis plug-in, which resembles the LTL Checker. In this application, called *SCIFF-Checker* [Chesani et al. 2008], all the peculiar features of the *SCIFF* language are extensively applied to classify execution logs by considering not only causal relationships among activities, but also their execution times, originators and involved content data. The tool provides different template rules (included the DecSerFlow ones) whose activity types, originators and execution times can be constrained and specialized by the user through a GUI. Results obtained by applying classification can be then directly exploited by the DecMiner plug-in, supporting the “mining-checking” cycle sketched in Section 6.2.5. *SCIFF-Checker* can be downloaded as part of the latest version of ProM.

7.3.6 DecMiner. DecMiner is a ProM plug-in which implements the mining algorithm described in [Lamma et al. 2007]. Its purpose is to mine a declarative constraint-based specification starting from a set of MXML execution traces previously labeled as conformant or not. Such a specification is composed by *SCIFF* rules. Because the structure of rules which can be mined by the algorithm can be configured by the user, DecMiner restricts to *SCIFF* rules that map DecSerFlow formulas. As a consequence, it is also able to apply the mapping presented in this work in the opposite way, automatically obtaining a graphical DecSerFlow model as result of the mining process. The plug-in can be downloaded as part of the latest version of ProM. The interested reader is referred to [Chesani et al. 2009] for

¹³The proof procedure has been implemented in SICStus Prolog, available from <http://www.sics.se/sicstus/>.

a description of the plug-in and its experimentation.

8. RELATED WORK

In this work we propose a declarative DecSerFlow language for the specification of service flows. Compared to the related research in the area, the most important added value of DecSerFlow is in the *comprehensiveness*: to our knowledge DecSerFlow is the only language which, besides for (1) the declarative modelling of service interaction protocols, can be used for (2) design-time verification (i.e., detection of contradictory constraints), (2) checking interoperability of services, (3) preventing deadlocks, (4) monitoring service executions and (5) learning service models from past executions. Many other approaches deal with each of these areas, but, to our knowledge, the work presented in this paper is the only one that tackles all of them. In the remainder of this section we will describe the most interesting related works: each paragraph describes the related work in one area and explains the added value of DecSerFlow.

Process modelling, enactment and verification is the focus in the field of workflow technology [Georgakopoulos et al. 1995]. Most process modelling languages (e.g., Petri nets, BPMN [White 2006], WS-BPEL [Andrews et al. 2003]) are highly procedural. Petri nets have been used for the modelling of workflows [van der Aalst and van Hee 2002; Chrzastowski-Wachtel 2003; Dumas et al. 2005] but also for the orchestration of web services [Mecella et al. 2002]. Another example of procedural languages for modelling and verification of web services are Message Sequence Charts (MSCs), which explicitly specify the ordering of message exchange between services [Foster et al. 2003]. The procedural nature of such modelling languages is an obstacle in developing choreographies of autonomous web services because possible orderings of message exchange between services must be explicitly included in the model [Zaha et al. 2006]. The declarative flavor of DecSerFlow is more suitable for modelling interactions of autonomous services because the possible execution orderings of activities are implicitly derived from constraints, as all orderings that satisfy these constraints. Furthermore, it is worth noting that even if process modeling languages such as WS-BPEL and BPMN provide support for data, such a perspective is often lost when they are translated to an underlying formal language. The mapping of DecSerFlow to SCIFF presented in this work enables the possibility of maintaining data-related and quantitative time aspects also at the formal level. Even if the central focus is usually on the activities and their flow dependencies, such additional perspectives are very important in settings like service discovery and contracting. For example, quantitative time constraints could be exploited by the user to express that she is looking for an e-shop able to deliver the ordered items within a maximum time span; since SCIFF provides support for reasoning about quantitative time constraints, such a requirement can be used to select only the services whose behavioral interface respect it. The interested reader may refer to [Alberti et al. 2007; Alberti et al. 2009] for the application of SCIFF in the context of service discovery and contracting.

Beside DecSerFlow, many other declarative languages, where possible orderings are implicitly derived from a set of constraints (i.e., rules), have been proposed in order to solve this problem. In [Zaha et al. 2006], Zaha et al. propose a declar-

ative language called *Let's Dance* for modeling interactions of web services. This language uses Computation Tree Logic (CTL) for flexible modeling of message exchange between services. A straight-forward graphical notation is used to represent patterns in message exchange, while π -calculus [Milner et al. 1992] captures the execution semantics [Decker et al. 2006]. A restricted version of LTL is used in [Hallé and Villemaire 2009] and translated into XQuery for monitoring of web services. LTL is also used in [Deutsch et al. 2006] for verification of correctness properties of service compositions. The SPIN model checker [Holzmann 2003] is used in [Fu et al. 2005] to verify LTL properties of service conversations. DecSerFlow is also a declarative language, which uses LTL and SCIFF mapping for formal specification of service interactions. Moreover, to the best of our knowledge, DecSerFlow is the only declarative language for modelling and monitoring of web services that also enables verification, interoperability check, model learning and deadlock-free execution.

The importance of monitoring web services has been raised by many researchers. Moreover, monitoring is addressed with several approaches: business rules [Lazovik et al. 2004], WS-BPEL [Baresi et al. 2004], event calculus [Mahbub and Spanoudakis 2004], WS-Agreement [Ludwig et al. 2004], etc. Advanced conformance checking techniques described in [Rozinat and van der Aalst 2006] are used in [van der Aalst et al. 2005] and implemented in the ProM framework [van der Aalst et al. 2007]; this approach has been applied to SOAP messages generated from Oracle BPEL. The work presented in this paper differs from these approaches because it presents *one declarative language* that can be used for monitoring, modelling, design-time verification, deadlock-free execution, interoperability check and model learning. In [Rouached et al. 2006] the authors use an extension of the Event Calculus (EC) of Kowalski and Sergot ([Kowalski and Sergot 1986]) to declaratively model event based requirements specifications. The choice of EC is motivated by both practical and formal needs, that are shared by the SCIFF approach. In particular, in contrast to pure state-transition representations, both the EC and SCIFF representations include an explicit time structure and are very close to most event-based specifications. However, SCIFF deals with explicit time by using suitable CLP constraints on finite domains, while they use a temporal formalism based on Event Calculus. We plan to deeply investigate the relations between SCIFF and EC, and possibly to integrate the approaches in future work.

Besides for monitoring of web services (“run-time conformance checking”), we also propose DecSerFlow for *design-time conformance*, i.e., detecting errors in models before enactment. Both mappings of DecSerFlow enable a simple mechanism that checks at design-time the correctness of a model and the compatibility of different services. Inheritance notions [van der Aalst and Basten 2002] are explored in the context of workflow management and implemented in Woflan [Verbeek et al. 2001]. Petri nets are used for design-time conformance and compatibility in [Martens 2005a; 2005b; Massuthe et al. 2005; Schlingloff et al. 2005]. For example, [Martens 2005b] focuses on the problem of consistency between executable and abstract processes while [Massuthe et al. 2005] presents an approach where for a given composite service the required other services are generated. Also related is [Foster et al. 2003], where Message Sequence Charts (MSCs) are compiled into the “Finite

State Process” notation to describe and reason about web service compositions. To the best of our knowledge, the work presented in this article is the first attempt to automatically verify declarative service models. Automatic service composition has been addressed in OWL-S [OWL Services Coalition 2003] which looks how atomic services interact with the real world, the Roman model approach [Berardi et al. 2005] that uses finite state machines, and Mealy machine [Bultan et al. 2003] that focuses on message exchange between services. Compatibility of synchronous communication via message exchange in web services is investigated in [Bordeaux et al. 2004; Beyer et al. 2005; Benatallah et al. 2006; Ponnekanti and Fox 2004], while DecSerFlow allows asynchronous communication and focuses on the process perspective, rather than message exchange. DecSerFlow contributes to this area are with the verification techniques described in section 6, which make it possible to easily discover errors and incompatibility in *declarative models*. However, while the cited approaches focus on automatic composition of services (i.e., automatic choreography generation from participating services), DecSerFlow assumes that all relevant process models of the composition are available and then verifies their interoperability.

In the research literature it is possible to find several definitions of *interoperability*, and there is not a complete agreement about its exactly meaning. For example, in [Baldoni et al. 2005b] the authors state that interoperability aims to check if a service, described by its behavioural interface, can play a given role within a choreography. In their approach however, both choreography and behavioural interface are described from a procedural viewpoint, and a complete specification of all the allowed interactions is given. SCIFF has been used to address this type of interoperability in [Alberti et al. 2006]. A different notion of interoperability is given in [Chopra and Singh 2006], where the authors represent global choreographies and local services in terms of state transition systems (and their conjunction as the product of the two transition systems). They define a notion of interoperability as a set of feature that the resulting transition system should guarantee. Although their idea of interoperability is in some sense “broader” that the one given in [Baldoni et al. 2005b; Alberti et al. 2006], it is still related to the procedural aspects of the interaction between the services. The interoperability notion discussed in this work instead is more related to assuring that declarative constraints specified in terms of DecSerFlow are indeed satisfied, given the DecSerFlow representation of both a global choreography and of a service. DecSerFlow in fact focus on the declarative aspects and features of a global choreographies, leaving the interaction unconstrained as much as possible.

Another issue tackled in this work is the problem of mining DecSerFlow models starting from a set of service execution traces. Process mining [van der Aalst and Weijters 2004; van der Aalst et al. 2003] extracts knowledge from event logs (e.g., process models [van der Aalst et al. 2004; Agrawal et al. 1998; Cook and Wolf 1998; Gaaloul et al. 2004; Gaaloul and Godart 2005; Herbst 2000] or social networks [van der Aalst and Song 2004]). In particular, Agrawal et al. [1998] introduced the idea of applying process mining to workflow management. The authors propose an approach for inducing a process representation in the form of a directed graph encoding the precedence relationships. van der Aalst et al. [2004] presents the α -

algorithm for mining Petri nets from data and identifies for which class of models the approach is guaranteed to work. The α -algorithm is based on the discovery of binary relations in the log, such as the “follows” relation. In [van Dongen and van der Aalst 2004] the authors describe an algorithm which derives causal dependencies between activities and use them for constructing instance graphs, presented in terms of Event-driven Process Chains (EPCs). A recent work of Greco et al. [2006] describe a mining technique where a process model is induced in the form of a disjunction of special graphs called workflow schemas. The *SCIFF*-based approach sketched in Section 6.2.4 differs from all of these works. First, *SCIFF* uses a declarative representation, which can be rendered as a DecSerFlow diagram by applying an inverse mapping. Moreover, *SCIFF* learns from both compliant and non compliant traces (rather than from compliant traces only), and is able to model and reason upon data, by exploiting either the underlying Constraints Solver or the Prolog inference engine. Various levels of web services mining (web service operations, interactions, and workflows) are proposed in [Gombotz and Dustdar 2005; Dustdar et al. 2004]. Our approach fits in their framework and shows that web-services mining is indeed possible.

As pointed out in [Baldoni et al. 2005a], Service oriented architectures and Multi Agent Systems share many issues and features, and the problem of representing global interactions and of verifying them has been tackled also in the MAS field. In particular, it is possible to find in the literature two complementary approaches, as in the case of choreographies: approaches with aim to exactly specify how the interaction protocol should be executed by the interacting agents (such as for example AUML [Bauer et al. 2001]), and approaches which consider MAS as open societies and model interaction protocols by declaratively constraining the possible interactions. For example, in [Fornara and Colombetti 2002] the semantics of communicative acts is defined by means of transitions on a finite state automaton which describes the concept of commitment; in [Yolum and Singh 2002], the authors adopts a variant of Event Calculus to commitment-based protocols, where commitments evolve in relation to events and fluents and the semantics of messages is given in terms of predicates on such events and fluents (to describe how messages affect commitments). Recently, Singh et al. have applied the concept of commitment-based protocols in the context of the Service Oriented Architecture and Business Process Management, by addressing the problem of business process adaptability [Desai et al. 2006] and of protocols composition [Mallya et al. 2005]. *SCIFF* was originally thought for dealing with social interaction in open MAS, and the mapping proposed in this work further attests that the MAS and SOC settings are closely related and can benefit from each other.

9. CONCLUSIONS AND FUTURE WORKS

In this work, we have made a first step towards a framework capable to tackle both specification and verification of service choreographies. By claiming that a choreography is inherently declarative, we have presented the DecSerFlow graphical language for modeling service choreographies. DecSerFlow adopts an open and declarative approach, specifying choreographies by means of the minimal set of constraints which should be satisfied by the interacting entities to correctly collaborate.

Thus, the approach respects the autonomous nature of participating services and does not lead to over-specifying nor over-constraining them.

Furthermore, we have concretely shown how the DecSerFlow concepts can be mapped onto different underlying logic-based formalisms, namely LTL and SCIFF (a framework based on abductive logic programming). After having introduced the complete mapping onto both settings, we have described how the related model-checking and proof-theoretic techniques can be fruitfully applied in order to enact DecSerFlow models and to perform a variety of different verification tasks, such as conformance checking, static verification of conflicts and dead activities, interoperability between global and local models, mining of DecSerFlow models from a set of compliant and non compliant histories.

We have also motivated the feasibility of the approach by briefly reviewing the DecSerFlow language in terms of some Cognitive Dimensions, and by quantitatively evaluating performances and scalability of the verification techniques (especially for what concerns static verification, which is the most difficult one for SCIFF). Obviously, the empirical evaluation by using the Cognitive Dimensions is only a first step towards the assessment of DecSerFlow's usability; we will therefore extend such an evaluation by conducting a comprehensive user study covering both the use of DecSerFlow to specify choreographies and the exploitation of the toolset to validate them.

The possibility of carrying out a suitable user study is conditioned by the presence of a stable prototypical implementation integrating the various related tool (and relying on ProM and DECLARE as glue environments). For the time being, the two underlying DecSerFlow formalisms are used independently; we are currently investigating their relationships, to the aim of really exploiting their advantages and of realizing a unified framework for choreographies specification and verification. Such an investigation will also be helpful to understand some theoretical relationships between LTL and SCIFF.

Even if DecSerFlow is proposed as complementary w.r.t. classical procedural approaches, an interesting open issue concerns how these different approaches could benefit from each other. Relevant research issues arise when trying to shift from one proposal to another. From one side (from procedural languages to DecSerFlow), such a shift would enable the possibility to abstract procedural models by focusing on their core constraints and, even more important, to make the different verification techniques described in this paper applicable also to procedural models¹⁴. From the other side (from DecSerFlow to procedural languages), DecSerFlow models could be used as core of a top-down methodology aimed at deriving executable procedural specifications from declarative constraints; this methodology could be applied, for instance, to derive skeletons of WS-BPEL behavioural interfaces starting from a declarative choreography specified in DecSerFlow. Another choice would be to opt for an integration between declarative and procedural approaches, to the aim of obtaining *semi-open* specification which suitably mediate between the two. A first investigation in this direction has been made in [Pesic 2008], where a lay-

¹⁴A possible solution to this problem could be to simulate and collect in a MXML log different positive and negative executions of the procedural model, and then try to mine a DecSerFlow model from the generated traces.

ered approach is proposed, in which non-atomic activities belonging to a declarative model can be specified in terms of a YAWL¹⁵ process and vice-versa. An hybrid approach, in which declarative and procedural specifications co-exist at the same level, will be matter of future research. It is worth noting that, in this case, the integration poses foundational issues, because a clear semantics must be defined to specify how a closed approach and an open approach (equipped with negative constraints) affect each other, how possible conflicts should be resolved, and so on.

Other ongoing works concern the extension of the DecSerFlow expressiveness. Currently, DecSerFlow templates can relate only to service activities while SCIFF and the ProM LTL Checker deal also with conformance checking against properties related to activities, time and data. This limitation of DecSerFlow can be eliminated by extending the language with such concepts.

Extension with time perspective would enable DecSerFlow to offer templates that involve deadlines. For example, as introduced in Section 5.4, the “response” template can be extended to specify the rule that activity “B” has to be executed no later than 5 days after activity “A”. To be able to support the semantics of deadlines, LTL can be replaced by the real-time temporal logic - a logic that can be translated into timed automata [Bouajjani et al. 1996]. Further on, timed automata can be used for execution and verification of models containing time perspective. Thanks to the possibility of exploiting the underlying CLP solver to adopt a temporal point algebra, SCIFF is instead directly able to capture such an extension. An ongoing issue concerns the use of SCIFF as an enactment module for extended models; the basic idea is to exploit the detection of dead activities but by considering also a partial execution trace (which represents the already executed activities inside the process instance): in this way, SCIFF can be used to discover, step-by-step, which activities cannot be executed without undermining model’s consistency.

Data elements would enrich DecSerFlow and allow for specifying more complex templates. Consider, for example, the photo service described in Section 3. Although it is generally possible to deliver ordered products to a home address or to the shop, one can imagine that large format posters can only be picked up personally. In this case, a special constraint would specify that if the size of a poster is “large” then type of delivery cannot be “home”. However, incorporating data perspective in DecSerFlow is a complex task. Data elements introduce many issues that need to be solved: are templates divided into ones that involve activities and ones that involve data, or can templates be mixed (one template involving activities and data)? How do we deal with the data scope (e.g., data has certain value before, after or between events, etc.)? Another complex issue is to find the right graphical representation of such templates dealing with data.

Investigating the possibility to extend DecSerFlow with time and data will add much to the semantics of DecSerFlow and make better use of SCIFF and its capability to express both perspectives. By adopting the implicit DecSerFlow formalization shown in Section 5.6, information about content data and the involved knowledge could be seamlessly expressed inside the specific knowledge base of the formalized model.

Finally, an extended formalization capable to deal also with non-atomic activities

¹⁵<http://www.yawl-system.com/newYAWL>

and more complex relationships, taking into account also exceptions and compensation issues, will be matter of future works.

ACKNOWLEDGMENT

This work has been partially supported by the PRIN 2005 project “Specification and Verification of Agent Interaction Protocols” and by the FIRB project “TO-CAI.IT”. We would like to thank Marco Gavanelli, Evelina Lamma, Marco Alberti, Paolo Torroni, Fabrizio Riguzzi and all colleagues that took part to the SOCS project. We also thank the anonymous reviewers for their valuable suggestions and helpful comments which helped to improve this work.

REFERENCES

- AGRAWAL, R., GUNOPULOS, D., AND LEYMAN, F. 1998. Mining Process Models from Workflow Logs. In *Sixth International Conference on Extending Database Technology*. 469–483.
- ALBERTI, M., CATTAFI, M., CHESANI, F., GAVANELLI, M., LAMMA, E., MELLO, P., MONTALI, M., AND TORRONI, P. 2009. Integrating abductive logic programming and description logics in a dynamic contracting architecture. In *Proceedings of the IEEE 7th International Conference on Web Services (ICWS 2009)*.
- ALBERTI, M., CHESANI, F., GAVANELLI, M., AND LAMMA, E. 2005. The chr-based implementation of a system for generation and confirmation of hypotheses. In *19th Workshop on (Constraint) Logic Programming, Ulm, Germany, February 21-23, 2005*, A. Wolf, T. W. Frühwirth, and M. Meister, Eds. Ulmer Informatik-Berichte, vol. 2005-01. Universität Ulm, Germany, 111–122.
- ALBERTI, M., CHESANI, F., GAVANELLI, M., LAMMA, E., MELLO, P., MONTALI, M., AND TORRONI, P. 2007. Web Service contracting: Specification and Reasoning with SCIFF. In *Proceedings of the 4th European Semantic Web Conference (ESWC’07)*, E. Franconi, M. Kifer, and W. May, Eds. Lecture Notes in Artificial Intelligence, vol. 4519. Springer Verlag, 68–83.
- ALBERTI, M., CHESANI, F., GAVANELLI, M., LAMMA, E., MELLO, P., AND TORRONI, P. 2005. Security Protocols Verification in Abductive Logic Programming: A Case Study. In *Proc. of ESAW’05*. LNCS, vol. 3963. Springer, 106–124.
- ALBERTI, M., CHESANI, F., GAVANELLI, M., LAMMA, E., MELLO, P., AND TORRONI, P. 2006. Compliance verification of agent interaction: a logic-based software tool. *Applied Artificial Intelligence* 20, 2-4, 133–157.
- ALBERTI, M., CHESANI, F., GAVANELLI, M., LAMMA, E., MELLO, P., AND TORRONI, P. 2008. Verifiable agent interaction in abductive logic programming: the SCIFF framework. *ACM Transactions on Computational Logic* 9, 4, 1–43.
- ALBERTI, M., GAVANELLI, M., LAMMA, E., CHESANI, F., MELLO, P., AND MONTALI, M. 2006. An abductive framework for a-priori verification of web services. In *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 10-12, 2006, Venice, Italy*, A. Bossi and M. J. Maher, Eds. ACM, 39–50.
- ANDREWS, T., CURBERA, F., DHOLAKIA, H., GOLAND, Y., KLEIN, J., LEYMAN, F., LIU, K., ROLLER, D., SMITH, D., THATTE, S., TRICKOVIC, I., AND WEERAWARANA, S. 2003. Business Process Execution Language for Web Services, Version 1.1. Standards proposal by BEA Systems, International Business Machines Corporation, and Microsoft Corporation.
- BALDONI, M., BAROGLIO, C., MARTELLI, A., AND PATTI, V. 2006. A priori conformance verification for guaranteeing interoperability in open environments. In *Service-Oriented Computing - ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, 2006, Proceedings*, A. Dan and W. Lamersdorf, Eds. Lecture Notes in Computer Science, vol. 4294. Springer, 339–351.
- BALDONI, M., BAROGLIO, C., MARTELLI, A., PATTI, V., AND SCHIFANELLA, C. 2005a. Verifying the conformance of web services to global interaction protocols: A first step. In *International Workshop on Web Services and Formal Methods, WS-FM 2005, Versailles, France, September*. ACM Transactions on the Web, Vol. V, No. N, May 2009.

- ber 1-3, 2005, *Proceedings*, M. Bravetti, L. Kloul, and G. Zavattaro, Eds. Lecture Notes in Computer Science, vol. 3670. Springer, 257–271.
- BALDONI, M., BAROGLIO, C., MARTELLI, A., PATTI, V., AND SCHIFANELLA, C. 2005b. Verifying the conformance of web services to global interaction protocols: A first step. In *EPEW/WS-FM*, M. Bravetti, L. Kloul, and G. Zavattaro, Eds. *Formal Techniques for Computer Systems and Business Processes, European Performance Engineering Workshop, EPEW 2005 and International Workshop on Web Services and Formal Methods, WS-FM 2005, Versailles, France, September 1-3, 2005, Proceedings 3670*.
- BARESI, L., GHEZZI, C., AND GUINEA, S. 2004. Smart Monitors for Composed Services. In *ICSOC '04: Proceedings of the 2nd International Conference on Service Oriented Computing*. ACM Press, New York, NY, USA, 193–202.
- BARROS, A., DUMAS, M., AND OAKS, P. 2005. A critical overview of the web services choreography description language (WS-CDL). *BPTrends*.
- BAUER, B., MÜLLER, J. P., AND ODELL, J. 2001. Agent uml: a formalism for specifying multiagent software systems. In *First international workshop, AOSE 2000 on Agent-oriented software engineering*. Springer-Verlag, 91–103.
- BELWOOD, T., CLÉMENT, L., EHNEBUSKE, D., HATELY, A., HONDO, M., HUSBAND, Y. L., JANUSZEWSKI, K., LEE, S., MCKEE, B., MUNTER, J., AND VON RIEGEN, C. 2000. UDDI Version 3.0. http://uddi.org/pubs/uddi_v3.htm.
- BENATALLAH, B., CASATI, F., AND TOUMANI, F. 2006. Representing, analysing and managing web service protocols. *Data Knowl. Eng.* 58, 3, 327–357.
- BERARDI, D., CALVANESE, D., GIACOMO, G. D., LENZERINI, M., AND MECELLA, M. 2005. Automatic service composition based on behavioral descriptions. *International Journal of Cooperative Information Systems* 14, 4, 333–376.
- BEYER, D., CHAKRABARTI, A., AND HENZINGER, T. 2005. Web service interfaces. In *Proceedings of the 14th international conference on World Wide Web*. 148–159.
- BORDEAUX, L., SALAÜN, G., BERARDI, D., AND MECELLA, M. 2004. When are two web services compatible? In *Proceedings of the 5th international workshop on Technologies for E-Services (TES 2004)*, M. Shan, U. Dayal, and M. Hsu, Eds. 15–28.
- BOUAJJANI, A., LAKHNECH, Y., AND YOVINE, S. 1996. Model-checking for extended timed temporal logics. In *FTRTFT '96: Proceedings of the 4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer-Verlag, London, UK, 306–326.
- BOX, D., EHNEBUSKE, D., KAKIVAYA, G., LAYMAN, A., MENDELSON, N., NIELSEN, H., THATTE, S., AND WINER, D. 2000. Simple Object Access Protocol (SOAP) 1.1. <http://www.w3.org/TR/soap>.
- BULTAN, T., FU, X., HULL, R., AND SU, J. 2003. Conversation specification: a new approach to design and analysis of e-service composition. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*. ACM Press, New York, NY, USA, 403–410.
- CHESANI, F., LAMMA, E., MELLO, P., MONTALI, M., RIGUZZI, F., AND STORARI, S. 2009. Exploiting inductive logic programming techniques for declarative process mining. *LNCS Transactions on Petri Nets and Other Models of Concurrency (ToPNoC), Special Issue on Concurrency in Process-Aware Information Systems*.
- CHESANI, F., MELLO, P., MONTALI, M., RIGUZZI, F., SEBASTIANIS, M., AND STORARI, S. 2008. Checking compliance of execution traces to business rules: an approach based on logic programming. In *4th Workshop on Business Process Intelligence (BPI 2008)*, M. Castellanos, A. K. A. de Medeiros, J. Mendling, and B. Weber, Eds. LNBIP. Springer Verlag. To appear.
- CHESANI, F., MONTALI, M., MELLO, P., AND STORARI, S. 2007. Testing careflow process execution conformance by translating a graphical language to computational logic. In *Proceedings of the 11th Conference on Artificial Intelligence in Medicine (AIME 07)*, A. Abu-Hanna, R. Bellazzi, and J. Hunter, Eds. LNAI, vol. To appear. Springer-Verlag.
- CHOPRA, A. K. AND SINGH, M. P. 2006. Producing compliant interactions: Conformance, coverage, and interoperability. In *Declarative Agent Languages and Technologies IV, 4th International Workshop, DALT 2006, Hakodate, Japan, May 8, 2006, Selected, Revised and Invited Papers*. Lecture Notes in Computer Science, vol. 4327. Springer, 1–15.

- CHRISTENSEN, E., CURBERA, F., MEREDITH, G., AND WEERAWARANA, S. 2001. Web Services Description Language (WSDL) 1.1. <http://www.w3.org/TR/wsdl>.
- CHRZASTOWSKI-WACHTTEL, P. 2003. A Top-down Petri Net Based Approach for Dynamic Workflow Modeling. In *International Conference on Business Process Management (BPM 2003)*, W. van der Aalst, A. ter Hofstede, and M. Weske, Eds. Vol. 2678. 336–353.
- CLARKE, E., GRUMBERG, O., AND PELED, D. 1999. *Model Checking*. The MIT Press, Cambridge, Massachusetts and London, UK.
- COOK, J. AND WOLF, A. 1998. Discovering Models of Software Processes from Event-Based Data. *ACM Transactions on Software Engineering and Methodology* 7, 3, 215–249.
- DE RAEDT, L. AND VAN LAER, W. 1995. Inductive constraint logic. In *Proceedings of the 6th Conference on Algorithmic Learning Theory*. LNAI, vol. 997. Springer Verlag.
- DECKER, G., ZAHA, J., AND DUMAS, M. 2006. Execution Semantics for Service Choreographies. In *Proceedings of the 3rd Workshop on Web Services and Formal Method (WS-FM 2006)*, M. Bravetti, M. Núñez, and G. Zavattaro, Eds. Lecture Notes in Computer Science, vol. 4184. Springer-Verlag, 163–177.
- DEMRI, S., LAROUSSINIE, F., AND SCHNOEBELEN, P. 2006. A Parametric Analysis of the State-Explosion Problem in Model Checking. *Journal of Computer and System Sciences* 72, 4, 547–575.
- DEMRI, S. AND SCHNOEBELEN, P. 1998. The Complexity of Propositional Linear Temporal Logics in Simple Cases. In *Proceedings of 15th Annual Symposium on Theoretical Aspects of Computer Science (STACS 98)*, G. Goos, J. Hartmanis, and J. Leeuwen, Eds. Lecture Notes in Computer Science, vol. 1373/1998. Springer-Verlag, Paris, France, 61–72.
- DENECKER, M. AND SCHREYE, D. D. 1998. SLDNFA: an abductive procedure for abductive logic programs. *Journal of Logic Programming* 34, 2, 111–167.
- DESAI, N., CHOPRA, A. K., AND SINGH, M. P. 2006. Business process adaptations via protocols. In *2006 IEEE International Conference on Services Computing (SCC 2006), 18-22 September 2006, Chicago, Illinois, USA*. IEEE Computer Society, 103–110.
- DEUTSCH, A., SUI, L., VIANU, V., AND ZHOU, D. 2006. Verification of Communicating Data-Driven Web Services. In *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, New York, NY, USA, 90–99.
- DUMAS, M., VAN DER AALST, W., AND TER HOFSTED, A. 2005. *Process-Aware Information Systems: Bridging People and Software through Process Technology*.
- DUSTDAR, S., GOMBOTZ, R., AND BAINA, K. 2004. Web Services Interaction Mining. Technical Report TUV-1841-2004-16, Information Systems Institute, Vienna University of Technology, Wien, Austria.
- FLUM, J. AND GROHE, M. 2006. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag.
- FORNARA, N. AND COLOMBETTI, M. 2002. Operational specification of a commitment-based agent communication language. C. Castelfranchi and W. Lewis Johnson, Eds. Bologna, Italy, 535–542.
- FOSTER, H., UCHITEL, S., MAGEE, J., AND KRAMER, J. 2003. Model-based Verification of Web Service Composition. In *Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE)*. Montreal, Canada, 152–161.
- FU, X., BULTAN, T., AND SU, J. 2005. Synchronizability of conversations among web services. *IEEE Transactions on Software Engineering* 31, 12, 1042–1055. Member-Tevfik Bultan and Senior Member-Jianwen Su.
- FUNG, T. H. AND KOWALSKI, R. A. 1997. The IFF proof procedure for abductive logic programming. *Journal of Logic Programming* 33, 2 (Nov.), 151–165.
- GAALLOUL, W., BHIRI, S., AND GODART, C. 2004. Discovering Workflow Transactional Behavior from Event-Based Log. In *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2004*, R. Meersman, Z. Tari, W. Aalst, C. Bussler, and A. G. et al., Eds. Vol. 3290. 3–18.
- GAALLOUL, W. AND GODART, C. 2005. Mining Workflow Recovery from Event Based Logs. In *Business Process Management (BPM 2005)*, W. Aalst, B. Benatallah, F. Casati, and F. Curbera, Eds. Vol. 3649. 169–185.

- GEORGAKOPOULOS, D., HORNICK, M., AND SHETH, A. 1995. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases* 3, 119–153.
- GERTH, R., PELED, D., VARDI, M., AND WOLPER, P. 1996. Simple On-The-Fly Automatic Verification of Linear Temporal Logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*. Chapman & Hall, Ltd., London, UK, 3–18.
- GIANNAKOPOULOU, D. AND HAVELUND, K. 2001. Automata-based verification of temporal properties on running programs. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*. IEEE Computer Society, Washington, DC, USA, 412.
- GOMBOTZ, R. AND DUSTDAR, S. 2005. On Web Services Mining. In *First International Workshop on Business Process Intelligence (BPI'05)*, M. Castellanos and T. Weijters, Eds. Nancy, France, 58–70.
- GRECO, G., GUZZO, A., PONTIERI, L., AND SACCÀ, D. 2006. Discovering expressive process models by clustering log traces. *IEEE Transactions on Knowledge and Data Engineering* 18, 8, 1010–1027.
- GREEN, T. R. G. 1989. Cognitive dimensions of notations. *People and Computers V*, 443–460.
- GREEN, T. R. G. AND PETRE, M. 1996. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing* 7, 131–174.
- HALLÉ, S. AND VILLEMAIRE, R. 2009. Runtime Monitoring of Web Service Choreographies Using Streaming XML. In *(to appear in) Proceedings of the 24th Annual ACM Symposium on Applied Computing (ACM SAC 2009)*.
- HERBST, J. 2000. A Machine Learning Approach to Workflow Management. In *Proceedings 11th European Conference on Machine Learning*. Vol. 1810. 183–194.
- HOLZMANN, G. 2003. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, Massachusetts, USA.
- JAFFAR, J. AND MAHER, M. 1994. Constraint logic programming: a survey. *Journal of Logic Programming* 19-20, 503–582.
- KAKAS, A. C., KOWALSKI, R. A., AND TONI, F. 1993. Abductive Logic Programming. *Journal of Logic and Computation* 2, 6, 719–770.
- KAKAS, A. C. AND MANCARELLA, P. 1990. On the relation between Truth Maintenance and Abduction. In *Proceedings of the 1st Pacific Rim International Conference on Artificial Intelligence, PRICAI-90, Nagoya, Japan*, T. Fukumura, Ed. 438–443.
- KAVANTZAS, N., BURDETT, D., RITZINGER, G., FLETCHER, T., AND LAFON, Y. 2004. Web Services Choreography Description Language, Version 1.0. W3C Working Draft 17-12-04.
- KOWALSKI, R. A. AND SERGOT, M. 1986. A logic-based calculus of events. *New Gen. Comput.* 4, 1, 67–95.
- LAMMA, E., MELLO, P., MONTALI, M., RIGUZZI, F., AND STORARI, S. 2007. Inducing declarative logic-based models from labeled traces. In *Proceedings of the 5th International Conference on Business Process Management (BPM 2007)*, G. Alonso, P. Dadam, and M. Rosemann, Eds. LNCS, vol. 4714. Springer, 344–359.
- LAMMA, E., MELLO, P., RIGUZZI, F., AND STORARI, S. 2007. Applying inductive logic programming to process mining. In *Proceedings of the 17th International Conference on Inductive Logic Programming*. Springer.
- LATVALA, T. 2003. Efficient Model Checking of Safety Properties. In *Proceedings of the 10th SPIN Workshop on Model Checking of Software*. Lecture Notes in Computer Science, vol. 2648. Springer Verlag, Berlin, 74–88.
- LAZOVIK, A., AIELLO, M., AND PAPAZOGLU, M. 2004. Associating Assertions with Business Processes and Monitoring their Execution. In *ICSOC '04: Proceedings of the 2nd International Conference on Service Oriented Computing*. ACM Press, New York, NY, USA, 94–104.
- LLOYD, J. W. 1987. *Foundations of Logic Programming*, 2nd extended ed.
- LUDWIG, H., DAN, A., AND KEARNEY, R. 2004. Crona: An Architecture and Library for Creation and Monitoring of WS-agreements. In *ICSOC '04: Proceedings of the 2nd International Conference on Service Oriented Computing*. ACM Press, New York, NY, USA, 65–74.

- MAHBUB, K. AND SPANOUDAKIS, G. 2004. A Framework for Requirements Monitoring of Service Based Systems. In *ICSOC '04: Proceedings of the 2nd International Conference on Service Oriented Computing*. ACM Press, New York, NY, USA, 84–93.
- MALLYA, A. U., DESAI, N., CHOPRA, A. K., AND SINGH, M. P. 2005. Owl-p: Owl for protocol and processes. In *4rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005), July 25-29, 2005, Utrecht, The Netherlands*, F. Dignum, V. Dignum, S. Koenig, S. Kraus, M. P. Singh, and M. Wooldridge, Eds. ACM, 139–140.
- MARTENS, A. 2005a. Analyzing Web Service Based Business Processes. In *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE 2005)*, M. Cerioli, Ed. Vol. 3442. 19–33.
- MARTENS, A. 2005b. Consistency between executable and abstract processes. In *Proceedings of International IEEE Conference on e-Technology, e-Commerce, and e-Services (EEE'05)*. IEEE Computer Society Press, 60–67.
- MASSUTHE, P., REISIG, W., AND SCHMIDT, K. 2005. An Operating Guideline Approach to the SOA. In *Proceedings of the 2nd South-East European Workshop on Formal Methods 2005 (SEEFM05)*. Ohrid, Republic of Macedonia.
- MECELLA, M., PARISI-PRESICCE, F., AND PERNICI, B. 2002. Modeling E-service Orchestration through Petri Nets. In *Proceedings of the Third International Workshop on Technologies for E-Services*. Vol. 2644. 38–47.
- MILNER, R., PARROW, J., AND WALKER, D. 1992. A Calculus of Mobile Processes. *Information and Computation* 100, 1, 1–40.
- MONTALI, M., ALBERTI, M., CHESANI, F., GAVANELLI, M., LAMMA, E., MELLO, P., AND TORRONI, P. 2008. Verification from declarative specifications using Logic Programming. In *24th International Conference on Logic Programming (ICLP)*, M. G. D. L. Banda and E. Pontelli, Eds. Number 5366 in Lecture Notes in Computer Science. Springer Verlag, Udine, Italy, 440–454.
- MUGGLETON, S. AND DE RAEDT, L. 1994. Inductive logic programming: Theory and methods. *J. Logic Program.* 19/20, 629–679.
- OWL SERVICES COALITION. 2003. *OWL-S: Semantic markup for web services*.
- PESIC, M. 2008. Constraint-based workflow management systems: Shifting controls to users. Ph.D. thesis, Beta Research School for Operations Management and Logistics, Eindhoven.
- PESIC, M., SCHONENBERG, H., AND VAN DER AALST, W. 2007. Declare: Full support for loosely-structured processes. In *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), 15-19 October 2007, Annapolis, Maryland, USA*. IEEE Computer Society, 287–300.
- PONNEKANTI, S. AND FOX, A. 2004. Interoperability among independently evolving web services. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*. Springer-Verlag New York, Inc., New York, NY, USA, 331–351.
- REISIG, W. AND ROZENBERG, G., Eds. 1998. *Lectures on Petri Nets I: Basic Models*. Vol. 1491.
- ROUACHED, M., PERRIN, O., AND GODART, C. 2006. Towards formal verification of web service composition. In *4th International Conference on Business Process Management*. LNCS, vol. 4102. Springer, 257–273.
- ROZINAT, A. AND VAN DER AALST, W. 2006. Conformance Testing: Measuring the Fit and Appropriateness of Event Logs and Process Models. In *BPM 2005 Workshops (Workshop on Business Process Intelligence)*, C. Bussler et al., Ed. Vol. 3812. 163–176.
- SCHLINGLOFF, B., MARTENS, A., AND SCHMIDT, K. 2005. Modeling and model checking web services. *Electronic Notes in Theoretical Computer Science: Issue on Logic and Communication in Multi-Agent Systems* 126, 3–26.
- SHEN, Y.-D., YOU, J.-H., YUAN, L.-Y., SHEN, S. S. P., AND YANG, Q. 2003. A dynamic approach to characterizing termination of general logic programs. *ACM Transactions on Computational Logic* 4, 4, 417–430.
- SINGH, M. P. 2000. A social semantics for agent communication languages. In *Issues in Agent Communication*, F. Dignum and M. Greaves, Eds. Lecture Notes in Computer Science, vol. 1916. Springer, 31–45.
- ACM Transactions on the Web, Vol. V, No. N, May 2009.

- VAN DER AALST, W. AND BASTEN, T. 2002. Inheritance of Workflows: An Approach to Tackling Problems Related to Change. *Theoretical Computer Science* 270, 1-2, 125–203.
- VAN DER AALST, W., DE BEER, H., AND VAN DONGEN, B. 2005. Process Mining and Verification of Properties: An Approach based on Temporal Logic. In *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2005*, R. Meersman and Z. T. et al., Eds. Vol. 3760. 130–147.
- VAN DER AALST, W., DUMAS, M., OUYANG, C., ROZINAT, A., AND VERBEEK, H. 2005. Choreography Conformance Checking: An Approach based on BPEL and Petri Nets (extended version). BPM Center Report BPM-05-25, BPMcenter.org, To appear in *ACM Transactions on Internet Technology*, Special Issue on Middleware for Service-Oriented Architectures.
- VAN DER AALST, W., DUMAS, M., AND TER HOFSTEDÉ, A. 2003. Web Service Composition Languages: Old Wine in New Bottles? In *Proceeding of the 29th EUROMICRO Conference: New Waves in System Architecture*, G. Chroust and C. Hofer, Eds. IEEE Computer Society, Los Alamitos, CA, 298–305.
- VAN DER AALST, W., DUMAS, M., TER HOFSTEDÉ, A., RUSSELL, N., VERBEEK, H. M. W., AND WOHEDE, P. 2005. Life after BPEL? In *International Workshop on Web Services and Formal Methods, WS-FM 2005, Versailles, France, September 1-3, 2005, Proceedings*, M. Bravetti, L. Kloul, and G. Zavattaro, Eds. Lecture Notes in Computer Science, vol. 3670. Springer, 35–50.
- VAN DER AALST, W. AND SONG, M. 2004. Mining Social Networks: Uncovering Interaction Patterns in Business Processes. In *International Conference on Business Process Management (BPM 2004)*, J. Desel, B. Pernici, and M. Weske, Eds. Vol. 3080. 244–260.
- VAN DER AALST, W., VAN DONGEN, B., GÜNTHER, C., MANS, R., DE MEDEIROS, A. A., ROZINAT, A., RUBIN, V., SONG, M., VERBEEK, H., AND WEIJTERS, A. 2007. ProM 4.0: Comprehensive Support for Real Process Analysis. In *Application and Theory of Petri Nets and Other Models of Concurrency (ICATPN 2007)*, J. Kleijn and A. Yakovlev, Eds. Vol. 4546. 484–494.
- VAN DER AALST, W., VAN DONGEN, B., HERBST, J., MARUSTER, L., SCHIMM, G., AND WEIJTERS, A. 2003. Workflow Mining: A Survey of Issues and Approaches. *Data and Knowledge Engineering* 47, 2, 237–267.
- VAN DER AALST, W. AND VAN HEE, K. 2002. *Workflow Management: Models, Methods, and Systems*. MIT press, Cambridge, MA.
- VAN DER AALST, W. AND WEIJTERS, A., Eds. 2004. *Process Mining*. Special Issue of *Computers in Industry*, Volume 53, Number 3. Elsevier Science Publishers, Amsterdam.
- VAN DER AALST, W., WEIJTERS, A., AND MARUSTER, L. 2004. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering* 16, 9, 1128–1142.
- VAN DER AALST, W. M. P. AND PESIC, M. 2006. Decserflow: Towards a truly declarative service flow language. In *Web Services and Formal Methods, Third International Workshop, WS-FM 2006 Vienna, Austria, September 8-9, 2006, Proceedings*, M. Bravetti, M. Núñez, and G. Zavattaro, Eds. Lecture Notes in Computer Science, vol. 4184. Springer, 1–23.
- VAN DONGEN, B. AND VAN DER AALST, W. 2005. A Meta Model for Process Mining Data. In *Proceedings of the CAiSE'05 Workshops (EMOI-INTEROP Workshop)*, J. Casto and E. Teniente, Eds. Vol. 2. FEUP, Porto, Portugal, 309–320.
- VAN DONGEN, B. F. AND VAN DER AALST, W. M. P. 2004. Multi-phase process mining: Building instance graphs. In *Conceptual Modeling - ER 2004, 23rd International Conference on Conceptual Modeling*. LNCS, vol. 3288. Springer, 362–376.
- VERBEEK, H., BASTEN, T., AND VAN DER AALST, W. 2001. Diagnosing Workflow Processes using Woflan. *The Computer Journal* 44, 4, 246–279.
- VILAIN, M., KAUTZ, H., AND VAN BEEK, P. 1990. Constraint propagation algorithms for temporal reasoning: a revised report. 373–381.
- WHITE, S. A. 2006. Business process modeling notation specification 1.0. Tech. rep., OMG.
- YOLUM, P. AND SINGH, M. 2002. Flexible protocol specification and execution: applying event calculus planning using commitments. In *The First International Joint Conference on Au-*
ACM Transactions on the Web, Vol. V, No. N, May 2009.

onomous Agents & Multiagent Systems, AAMAS 2002, July 15-19, 2002, Bologna, Italy, Proceedings. 527–534.

ZAHA, J., BARROS, A., DUMAS, M., AND HOFSTEDE, A. 2006. Let's Dance: A Language for Service Behavior Modeling. In *Proceedings of the 14th International Conference on Cooperative Information Systems (CoopIS 2006)*, R. Meersman and Z. Tari, Eds. Lecture Notes in Computer Science, vol. 4275. Springer-Verlag, 145–162.

ZAHA, J., BARROS, A., DUMAS, M., AND TER HOFSTEDE, A. 2006. Let's dance: A language for service behavior modeling. QUT ePrints 4468, Faculty of Information Technology, Queensland University of Technology.

ZAHA, J., DUMAS, M., HOFSTEDE, A., BARROS, A., AND DEKKER, G. 2006. Service Interaction Modeling: Bridging Global and Local Views. QUT ePrints 4032, Faculty of Information Technology, Queensland University of Technology.

A. DECSEFLOW TEMPLATES AND MAPPINGS ONTO LTL AND SCIFF

DecSerFlow is proposed as a language containing more than twenty initial templates. Note that templates can be easily changed, added or removed in/from DecSerFlow. The remainder of this section is organized as follows. In Section A.1 we present DecSerFlow templates. Sections A.2 and A.3 DecSerFlow show the mappings onto LTL and onto SCIFF, respectively.

A.1 DecSerFlow Templates

DecSerFlow templates currently involve only activities, i.e., templates specify relations between service activities. Templates can involve an arbitrary number of activities, but for the sake of simplicity we present only unary and binary templates. Unary templates are presented in Table XIII. These templates specify the possible number of executions of an activity and are graphically represented as a cardinality constraint above the activity. The first group of templates (i.e., the “existence_N” templates) specify the minimal number of executions of an activity. The second group of templates (i.e., the “absence_N” templates) specifies the maximal number of executions of an activity. Finally, the third group of templates (i.e., the “exactly_N” templates) specifies the exact number of executions of an activity.

Table XIV presents binary templates for specifying relations between two activities (“A” and “B”). These templates are graphically represented as special lines (type of the line, symbols and line edges, etc...) between the two activities. The first two templates (i.e., “responded_existence” and “coexistence”) do not take into account the order in which “A” and “B” are executed. The second group of templates (i.e., “response”, “precedence” and “succession”) takes into account the order in which the two activities are executed in the most general way. The third group of templates (i.e., “alternate_response”, “alternate_precedence” and “alternate_succession”) takes the order of activities into account and impose interposition, i.e., one activity has to be executed between each two executions of the other activity. The fourth group of templates (i.e., “chain_response”, “chain_precedence” and “chain_succession”) specifies the most strict ordering relations by requiring that the two activities are executed immediately next to each other.

Binary templates that specify “negative” relations are presented in Table XV. Each of these templates presents a negation of a unary template presented in Table XIV. The graphical representation is similar to the one for the corresponding template from Table XIV – it has an extra negation symbol in the middle of the line. Note that there exists equivalence between some of the “negation” templates, i.e., some templates have *identical semantics*. Template “responded_absence” can be omitted because it is equivalent to the “not_coexistence” template. Templates “neg_response” and “neg_precedence” can be omitted because they are equivalent with the “neg_succession” template. Similarly, templates “neg_chain_response” and “neg_chain_precedence” are equivalent to the template “neg_chain_succession” and they can, therefore, be omitted too.

A.2 DecSerFlow Templates: LTL Mapping

The semantics of DecSerFlow templates can easily be specified in LTL. This section presents LTL mappings (i.e., LTL formulas) for each of the DecSerFlow templates

Table XIII. DecSerFlow existence templates.

name	representation	description
$existence_1(A)$	$\boxed{A}^{1..*}$	A is executed at least once.
$existence_2(A)$	$\boxed{A}^{2..*}$	A is executed at least two times.
$existence_3(A)$	$\boxed{A}^{3..*}$	A is executed at least three times.
...
$existence_N(N, A)$	$\boxed{A}^{N..*}$	A is executed at least N times.
$absence(A)$	\boxed{A}^0	A is never executed.
$absence_2(A)$	$\boxed{A}^{0..1}$	A is executed at most once.
$absence_3(N, A)$	$\boxed{A}^{0..2}$	A is executed at most two times.
...
$absence_N+1(A)$	$\boxed{A}^{0..N}$	A is executed at most N times.
$exactly_1(A)$	\boxed{A}^1	A is executed exactly once.
$exactly_2(A)$	\boxed{A}^2	A is executed exactly two times.
...
$exactly_N(A)$	\boxed{A}^N	A is executed exactly N times.


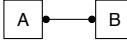


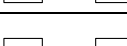
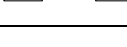
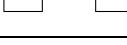


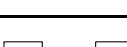
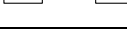
presented in Section A.1. Table XVI presents LTL specifications for (unary) existence templates from Table XIII. Tables XVII and XVIII present LTL specifications for binary templates from Tables XIV and XV, respectively. Note that Tables XVII and XVIII contain two additional formulas that do not appear in the Tables XIV and XV. The “interposition” formula in Table XVII is used in templates “alternate_response” and “alternate_precedence” and it specifies that there is at least one “B” between every two executions of “A”. Opposite to this, formula “negative_interposition” from Table XVII specifies that there cannot be any B between every two executions of A. The “negative_interposition” formula is used in templates “neg_alternate_response” and “neg_alternate_precedence”.

Table XVI show that new LTL formula has to be specified for each of the unary templates, i.e., it is not possible to define general LTL formulas with a parameterized number of executions of an activity.

A.3 DecSerFlow Templates: SCIFF Mapping

The semantics of DecSerFlow templates can also be specified in terms of SCIFF. This section presents the SCIFF mapping for each of the DecSerFlow templates presented in Section A.1. Table XIX presents SCIFF specifications for (unary)

Table XIV. DecSerFlow relation templates.

name	representation	description
$responded_existence(A, B)$		If A is executed, then B has to be executed before or after A.
$coexistence(A, B)$		If A is executed, then B has to be executed before or after A and vice versa.
$response(A, B)$		Every A is eventually followed by at least one B.
$precedence(A, B)$		B can be executed only after A is executed.
$succession(A, B)$		B is response of A and A is precedence of B.
$alternate_response(A, B)$		B is response of A and there has to be at least one B between every two As.
$alternate_precedence(A, B)$		A is precedence of B and there has to be at least one A between every two Bs.
$alternate_succession(A, B)$		B is alternate response of A and A is alternate precedence of B.
$chain_response(A, B)$		If A is executed then B is executed next (immediately after A).
$chain_precedence(A, B)$		B can be executed only if A was previously executed (immediately before B).
$chain_succession(A, B)$		A and B are always executed next to each other, i.e., first A and then immediately B.

existence templates shown in Table XIII. Tables XX and XXI present the \mathcal{SCIFF} specifications for binary templates from Tables XIV and XV, respectively.

Unlike LTL, that requires a separate specification of each of the unary templates (cf. Table XVI), the \mathcal{SCIFF} formalism can specify these templates in a general way (see section 5.6). Table XIX presents the \mathcal{SCIFF} template specifications for unary existence templates (cf. Table XIII) with two parameters: (1) parameter “A” represents an activity and (2) parameter “N” the number of executions.

\mathcal{SCIFF} specification of binary templates in Tables XIX contain parameters “A” that represents an activity and an additional parameter to represent moments in time “T” when “A” is executed. Parameters “ T_i ” are “free”, i.e., they do not have to be concretely specified but are required by the \mathcal{SCIFF} formal specification language to denote (and, eventually, explicitly constrain) the different execution times. Note that, e.g., in the “existence_N” template “ T_i ” is not an information related to the template formula, but is used to specify that “A” is executed multiple times (i.e., $T_i > T_{i-1}$ allows to differentiate the execution of two activities “A”).

\mathcal{SCIFF} specification of binary templates in Tables XX and XXI contain parameters “A” and “B” that represent activities and additional parameters to represent moments in time: (1) “ T_A ” when “A” is executed and (2) “ T_B ” when “B” is exe-

Table XV. DecSerFlow negation templates.

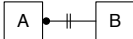
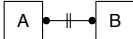
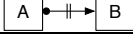
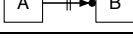


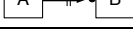
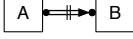



name	representation	description
<i>responded_absence</i> (A, B)		If A is executed, then B can never be executed.
<i>not_coexistence</i> (A, B)		Either A or B are executed, but not both.
<i>neg_response</i> (A, B)		B can never be executed after A.
<i>neg_precedence</i> (A, B)		B cannot be executed if A is executed before.
<i>neg_succession</i> (A, B)		B is not_response of A and A is not_precedence of B.
<i>neg_alt_response</i> (A, B)		There cannot be any B between every two As.
<i>neg_alt_precedence</i> (A, B)		There cannot be any A between every two Bs.
<i>neg_alt_succession</i> (A, B)		There cannot be any B between every two As and there cannot be any A between every two Bs.
<i>neg_chain_response</i> (A, B)		B cannot be the next (immediately) after A.
<i>neg_chain_precedence</i> (A, B)		A cannot be previous (immediately) before B.
<i>neg_chain_succession</i> (A, B)		A and B cannot be executed in a sequence.

Table XVI. DecSerFlow existence templates in LTL.

name	LTL formula
<i>existence_1</i> (A)	$\diamond(A)$
<i>existence_2</i> (A)	$\diamond(A \wedge \circ(\textit{existence}_1(A)))$
<i>existence_3</i> (A)	$\diamond(A \wedge \circ(\textit{existence}_2(A)))$
...	...
<i>existence_N</i> (N, A)	$\diamond(A \wedge \circ(\textit{existence}_{N-1}(A)))$
<i>absence_1</i> (A)	$\neg(\textit{existence}_1(A))$
<i>absence_2</i> (A)	$\neg(\textit{existence}_2(A))$
<i>absence_3</i> (A)	$\neg(\textit{existence}_3(A))$
...	...
<i>absence_{N+1}</i> (A)	$\neg(\textit{existence}_{N+1}(A))$
<i>exactly_1</i> (A)	$\textit{existence}_1(A) \wedge \textit{absence}_2(A)$
<i>exactly_2</i> (A)	$\textit{existence}_2(A) \wedge \textit{absence}_3(A)$
...	...
<i>exactly_N</i> (A)	$\textit{existence}_N(A) \wedge \textit{absence}_{N+1}(A)$

cuted. Just like in Table XIX, parameters “ T_A ” and “ T_B ” are “free”, i.e., they do not have to be concretely specified (they could be substituted by the classical Prolog anonymous variable “_”). Note that, e.g., in the “responded_existence” template “ T_A ” and “ T_B ” are not used for the template semantics, while in the template

Table XVII. DecSerFlow relation templates in LTL.

name	LTL formula
<i>responded_existence</i> (A, B)	$\diamond(A) \Rightarrow \diamond(B)$
<i>coexistence</i> (A, B)	$\diamond(A) \Leftrightarrow \diamond(B)$
<i>response</i> (A, B)	$\square(A \Rightarrow \diamond(B))$
<i>precedence</i> (A, B)	$\diamond(B) \Rightarrow ((\neg B) \sqcup A)$
<i>succession</i> (A, B)	$response(A, B) \wedge precedence(A, B)$
<i>alternate_response</i> (A, B)	$response(A, B) \wedge \square(A \Rightarrow \circ(precedence(B, A)))$
<i>alternate_precedence</i> (A, B)	$precedence(A, B) \wedge \square(B \Rightarrow \circ(precedence(A, B)))$
<i>alternate_succession</i> (A, B)	$alternate_response(A, B) \wedge$ $alternate_precedence(A, B)$
<i>chain_response</i> (A, B)	$\square(A \Rightarrow \circ(B))$
<i>chain_precedence</i> (A, B)	$precedence(A, B) \wedge \square(\circ(B) \Rightarrow A)$
<i>chain_succession</i> (A, B)	$chain_response(A, B) \wedge chain_precedence(A, B)$
<i>interposition</i> (A, B)	$\square(A \rightarrow \circ(precedence(B, A)))$

Table XVIII. DecSerFlow negation templates in LTL.

name	LTL formula
<i>responded_absence</i> (A, B)	$\diamond(A) \Rightarrow \neg(\diamond(B))$
<i>not_coexistence</i> (A, B)	$neg_existence_response(A, B) \wedge$ $neg_existence_response(B, A)$
<i>neg_response</i> (A, B)	$\square(A \Rightarrow \neg(\diamond(B)))$
<i>neg_precedence</i> (A, B)	$\square(\diamond(B) \Rightarrow (\neg A))$
<i>neg_succession</i> (A, B)	$neg_response(A, B) \wedge neg_precedence(A, B)$
<i>neg_alt_response</i> (A, B)	$negative_interposition(A, B)$
<i>neg_alt_precedence</i> (A, B)	$negative_interposition(B, A)$
<i>neg_alt_succession</i> (A, B)	$neg_alt_response(A, B) \wedge neg_alt_precedence(A, B)$
<i>neg_chain_response</i> (A, B)	$B \square(A \Rightarrow \circ(\neg(B)))$
<i>neg_chain_precedence</i> (A, B)	$\square(\circ(B) \Rightarrow \neg(A))$
<i>neg_chain_succession</i> (A, B)	$neg_chain_response(A, B) \wedge$ $neg_chain_precedence(A, B)$
<i>negative_interposition</i> (A, B)	$\square(A \rightarrow \circ((\neg B) \sqcup A))$

Table XIX. DecSerFlow existence templates in SCIFF.

name	SCIFF formula
<i>absence</i> (A)	$true \rightarrow \mathbf{EN}(performed(A), T)$
<i>existence_N</i> (N, A)	$true \rightarrow \bigwedge_{i=1}^N (\mathbf{E}(performed(A), T_i) \wedge T_i > T_{i-1})$
<i>absence_{N+1}</i> (N, A)	$\bigwedge_{i=1}^N (\mathbf{H}(performed(A), T_i) \wedge T_i > T_{i-1})$ $\rightarrow \mathbf{EN}(performed(A), T) \wedge T > T_N$
<i>exactly_N</i> (N, A)	$existence_N(N, A) \wedge absence_N + 1(N, A)$

We assume $T_0 = 0$

“response” they are used to specify that “B” is executed after “A” (i.e., $T_B > T_A$).

Formulas “chain_response” and “chain_precedence” use the “next” predicate to specify that two activities are executed immediately next to each other. This is formalized by stating that no activity should be performed between the two activities, i.e. by formalizing the *next* concept as follows:

$$\begin{aligned} \text{next}(T_2, T_1) \leftarrow \\ \mathbf{EN}(\text{performed}(X), T_X) \wedge T_X > T_1 \wedge T_X < T_2. \end{aligned}$$

Such a concept is part of the general knowledge base.

Table XX. DecSerFlow relation templates in SCIFF.

name	SCIFF formula
<i>responded_existence</i> (A, B)	$\mathbf{H}(\text{performed}(A), T_A)$ $\rightarrow \mathbf{E}(\text{performed}(B), T_B).$
<i>coexistence</i> (A, B)	$\text{responded_existence}(A, B)$ $\wedge \text{responded_existence}(B, A).$
<i>response</i> (A, B)	$\mathbf{H}(\text{performed}(A), T_A)$ $\rightarrow \mathbf{E}(\text{performed}(B), T_B) \wedge T_B > T_A.$
<i>precedence</i> (A, B)	$\mathbf{H}(\text{performed}(B), T_B)$ $\rightarrow \mathbf{E}(\text{performed}(A), T_A) \wedge T_A < T_B.$
<i>succession</i> (A, B)	$\text{response}(A, B)$ $\wedge \text{precedence}(A, B).$
<i>alternate_response</i> (A, B)	$\text{response}(A, B)$ $\wedge \text{interposition}(A, B).$
<i>alternate_precedence</i> (A, B)	$\text{precedence}(A, B)$ $\wedge \text{interposition}(B, A).$
<i>alternate_succession</i> (A, B)	$\text{alternate_response}(A, B)$ $\wedge \text{alternate_precedence}(A, B).$
<i>chain_response</i> (A, B)	$\mathbf{H}(\text{performed}(A), T_A)$ $\rightarrow \mathbf{E}(\text{performed}(B), T_B) \wedge T_B > T_A$ $\wedge \text{next}(T_B, T_A).$
<i>chain_precedence</i> (A, B)	$\mathbf{H}(\text{performed}(B), T_B)$ $\rightarrow \mathbf{E}(\text{performed}(A), T_A) \wedge T_A < T_B$ $\wedge \text{next}(T_B, T_A).$
<i>chain_succession</i> (A, B)	$\text{chain_response}(A, B)$ $\wedge \text{chain_precedence}(A, B).$
<i>interposition</i> (A, B)	$\mathbf{H}(\text{performed}(A), T_A)$ $\wedge \mathbf{H}(\text{performed}(A), T_{A2}) \wedge T_{A2} > T_A$ $\rightarrow \mathbf{E}(\text{performed}(B), T_B)$ $\wedge T_B > T_A \wedge T_B < T_{A2}.$

Table XXI shows the SCIFF formalization of “negation” templates from Table XV. For the sake of simplicity and reusability, in formulas “neg_chain_response”

and “neg_chain_precedence”, we introduce a predicate to specify that if it is the case that “A” is executed and “B” is executed after “A”, then the corresponding execution times should not be next to each other. This is expressed by stating that at least one activity should be performed between the two activities execution times, i.e. by formalizing the *not_next* concept as follows:

$$\begin{aligned} \text{not_next}(T_2, T_1) \leftarrow \\ \mathbf{E}(\text{performed}(X), T_X) \wedge T_X > T_1 \wedge T_X < T_2. \end{aligned}$$

Table XXI. DecSerFlow negation templates in \mathcal{SCIFF} .

name	\mathcal{SCIFF} formula
<i>responded_absence</i> (A, B)	$\mathbf{H}(\text{performed}(A), T_A)$ $\rightarrow \mathbf{EN}(\text{performed}(B), T_B)$.
<i>not_coexistence</i> (A, B)	$\text{responded_absence}(A, B)$ $\wedge \text{responded_absence}(B, A)$.
<i>neg_response</i> (A, B)	$\mathbf{H}(\text{performed}(A), T_A)$ $\rightarrow \mathbf{EN}(\text{performed}(B), T_B) \wedge T_B > T_A$.
<i>neg_precedence</i> (A, B)	$\mathbf{H}(\text{performed}(B), T_B)$ $\rightarrow \mathbf{EN}(\text{performed}(A), T_A) \wedge T_A < T_B$.
<i>neg_succession</i> (A, B)	$\text{negation_response}(A, B)$ $\wedge \text{negation_precedence}(A, B)$.
<i>neg_alt_response</i> (A, B)	$\text{negative_interposition}(A, B)$.
<i>neg_alt_precedence</i> (A, B)	$\text{negative_interposition}(B, A)$.
<i>neg_alt_succession</i> (A, B)	$\text{neg_alt_response}(A, B)$ $\wedge \text{neg_alt_precedence}(A, B)$.
<i>neg_chain_response</i> (A, B)	$\mathbf{H}(\text{performed}(A), T_A)$ $\wedge \mathbf{H}(\text{performed}(B), T_B) \wedge T_B > T_A$ $\rightarrow \text{not_next}(T_B, T_A)$.
<i>neg_chain_precedence</i> (A, B)	$\mathbf{H}(\text{performed}(B), T_B)$ $\wedge \mathbf{H}(\text{performed}(A), T_A) \wedge T_A < T_B$ $\rightarrow \text{not_next}(T_B, T_A)$.
<i>neg_chain_succession</i> (A, B)	$\text{neg_chain_response}(A, B)$ $\wedge \text{neg_chain_precedence}(A, B)$.
<i>negative_interposition</i> (A, B)	$\mathbf{H}(\text{performed}(A), T_A)$ $\wedge \mathbf{H}(\text{performed}(A), T_{A2}) \wedge T_{A2} > T_A$ $\rightarrow \mathbf{EN}(\text{performed}(B), T_B)$ $\wedge T_B > T_A \wedge T_B < T_{A2}$.