DecSerFlow: Towards a Truly Declarative Service Flow Language

W.M.P. van der Aalst and M. Pesic

Department of Information Systems, Eindhoven University of Technology, P.O.Box 513, NL-5600 MB, Eindhoven, The Netherlands.
w.m.p.v.d.aalst@tm.tue.nl,m.pesic@tm.tue.nl

Abstract. The need for process support in the context of web services has triggered the development of many languages, systems, and standards. Industry has been developing software solutions and proposing standards such as BPEL, while researchers have been advocating the use of formal methods such as Petri nets and π-calculus. The languages developed for *service flows*, i.e., process specification languages for web services, have adopted many concepts from classical workflow management systems. As a result, these languages are rather procedural and this does not fit well with the autonomous nature of services. Therefore, we propose *DecSerFlow* as a *Declarative Service Flow Language*. DecSerFlow can be used to specify, enact, and monitor service flows. The language is extendible (i.e., constructs can be added without changing the engine or semantical basis) and can be used to enforce or to check the conformance of service flows. Although the language has an appealing graphical representation, it is grounded in temporal logic.

Key words: Service flows, web services, workflow management, flexibility, temporal logic.

1 Introduction

The *Business Process Execution Language for Web Services* (BPEL4WS, or BPEL for short) has become the de-facto standard for implementing processes based on web services [7]. Systems such as Oracle BPEL Process Manager, IBM WebSphere Application Server Enterprise, IBM WebSphere Studio Application Developer Integration Edition, and Microsoft BizTalk Server 2004 support BPEL, thus illustrating the practical relevance of this language. Although intended as a language for connecting web services, its application is not limited to cross-organizational processes. It is expected that in the near future a wide variety of process-aware information systems [8] will be realized using BPEL. Whilst being a powerful language, BPEL is of a *procedural nature* and not very different from classical workflow languages e.g., the languages used by systems such as Staffware, COSA, SAP Workflow, and IBM WebSphere MQ Workflow (formerly know as FlowMark). Also other languages proposed in the context of

web services are of a procedural nature, e.g., the *Web Services Choreography Description Language* (WS-CDL) [16]. In this paper, we will not discuss these languages in detail. The interested reader is referred to [2, 3, 20] for a critical review of languages like BPEL. Instead, we will demonstrate that it is possible to use a more declarative style of specification by introducing *DecSerFlow*: a *Declarative Service Flow Language*.

To explain the difference between a procedural style and a declarative style of modeling, we use a simple example. Suppose that there are two activities A and B. Both *can* be executed multiple times but they *exclude* each other, i.e., after the first occurrence of A it is not allowed to do B anymore and after the first occurrence of B it is not allowed to do A. The following execution sequences are possible based on this verbal description: *[ ]* (the empty execution sequence), *[A]*, *[B]*, *[A,A]*, *[B,B]*, etc. In a *procedural* language it is difficult to specify the above process without implicitly introducing additional assumptions and constraints. In a procedural language one typically needs to make a choice with respect to whether no activities are to be executed, only A activities are to be executed, or only B activities are to be executed. Moreover, the number of times A or B needs to be executed also has to be decided. This means that one or more decision activities need to be executed before the execution of "real" activities can start. (Note that this is related to the Deferred Choice pattern described in [4].) The introduction of these decision activities typically leads to an over-specification of the process. Designers may be tempted to make this decision before the actual execution of the first A or B. This triggers the following two questions: (1) "How is this decision made?" and (2) "When is this decision made?". The designer may even remove the choice altogether and simply state that one can only do A activities. Using a more *declarative* style can avoid this over-specification. For example, in *Linear Temporal Logic* (LTL) [11–13] one can write $\neg(\Diamond A \wedge \Diamond B)$. This means that it cannot be the case that eventually A is executed and that eventually B is executed. This shows that a very compact LTL expression $(\neg(\Diamond A \wedge \Diamond B))$ can describe exactly what is needed without forcing the designer to specify more than strictly needed. Unfortunately, languages like LTL are difficult to use for non-experts. Therefore, we have developed a graphical language (DecSerFlow) that allows for the easy specification of processes in a declarative manner. DecSerFlow is mapped onto LTL. The innovative aspects of our approach based on DecSerFlow are:

- DecSerFlow allows for a *declarative style* of modeling which is highly relevant in the context of service flows (unlike languages like BPEL).
- Through the *graphical representation* of DecSerFlow this language is easy to use and we avoid the problems of textual languages like LTL.
- We use LTL not only for the verification of model properties: we also use the LTL formulas generated by DecSerFlow to *dynamically monitor services* and to realize an *enactment engine*.
- DecSerFlow is an extendible language (i.e., we supply an editor to extend the language with user-defined graphical constructs without the need to modify any part of the system).

– DecSerFlow can be used to specify two types of constraints: *hard* constraints and *soft* constraints. Hard constraints are enforced by the engine while soft constraints are only used to warn before the violation takes place and to monitor observed violations.

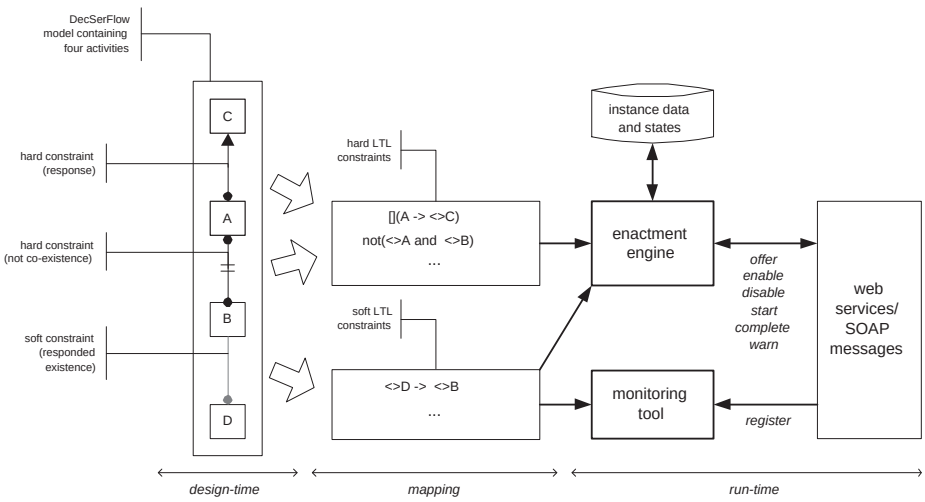


Fig. 1. Overview of the role played by DecSerFlow in supporting services flows.

Figure 1 provides an overview of the way we envision DecSerFlow to be used. At design-time, a graphical model is made using the DecSerFlow notation. (Note that at design-time users can also add new modeling elements - types of constraints.) The left-hand side of Figure 1 shows a process composed of four activities, A, B, C, and D. Moreover, three constraints are shown. The connection between A and C means that any occurrence of A should eventually be followed by at least one occurrence of C (i.e., $\Box(A \rightarrow \Diamond C)$ in LTL terms). The connection between A and B means that it cannot be the case that eventually A is executed and that eventually B is executed. This is the constraint described before, i.e., $\neg(\Diamond A \wedge \Diamond B)$ in LTL terms. The last constraint connecting D and B is a soft constraint. This constraint states that any occurrence of D implies also the occurrence of B (before or after the occurrence of D), e.g., *[B,D,D,D,D]*, *[D,D,D,B]*, and *[B,B,B]* are valid executions. The LTL formulation of this constraint is $\Diamond D \rightarrow \Diamond B$.

As Figure 1 shows, it is possible to automatically map the graphical model onto LTL formulas. These formulas can be used by the enactment engine to control the service flow, e.g., on the basis of hard constraints the engine can allow or prohibit certain activities and on the basis of soft constraints warnings can be issued. The soft constraints can also be used by the monitoring tool to detect and analyze violations.

Currently, we have implemented a graphical editor and the mapping of the editor to LTL. This editor supports user-defined notations as described before. We are currently investigating different ways to enact LTL formulas and in this paper we described our current efforts. Although we do not elaborate this this in this paper, our implementation will also incorporate data as is show in Figure 1. Data is used for routing purposes by making constraints data dependent, i.e., a constraint only applies if its guard evaluates to true. Moreover, in the context of the ProM (Process Mining) framework [6, 18] we have developed an LTL checker [1] to compare actual behavior with specified behavior. The actual behavior can be recorded by a dedicated process engine. However, it can also be obtained by monitoring SOAP messages as described in [3].

The approach described in Figure 1 is not limited to service flows. It can be applied in any context where *autonomous* entities are executing activities. These autonomous entities can be other organizations but also people or groups of people. This is the reason that DecSerFlow has a "sister language" named *ConDec* which aims at supporting teamwork and workflow flexibility [17]. Both languages/applications share the same concepts and tools.

The remainder of this paper is organized as follows. Section 2 introduces the DecSerFlow language. Then, a non-trivial example is given in Section 3. Section 4 discusses different ways to construct an enactment (and monitoring) engine based on DecSerFlow. Finally, Section 5 concludes the paper by discussing different research directions.

2 DecSerFlow: A Declarative Service Flow Language

Languages such as *Linear Temporal Logic* (LTL) [11–13] allow for the a more declarative style of modeling. These languages include temporal operators such as next-time ($\bigcirc F$), eventually ($\Diamond F$), always ($\Box F$), and until ($F \sqcup G$). However, such languages are difficult to read. Therefore, we define an extendible graphical syntax for some typical constraints encountered in service flows. The combination of this graphical language and the mapping of this graphical language to LTL forms the *Declarative Service Flow (DecSerFlow) Language* . We propose DecSerFlow for the *specification of a single service, simple service compositions, and more complex choreographies.*

Developing a model in DecSerFlow starts with creating activities. The notion of an activity is like in any other workflow-like language, i.e., an activity is atomic and corresponds to a logical unit of work. However, the nature of the *relations between activities* in DecSerFlow can be quite different than in traditional procedural workflow languages (like Petri nets and BPEL). For example, places between activities in a Petri net describe causal dependencies and can be used to specify sequential, parallel, alternative, and iterative routing. Using such mechanisms it is both possible and necessary to strictly define *how* the flow will be executed. We refer to relations between activities in DecSerFlow as *constraints*. Each of the constraints represents a policy (or a business rule). At any point in time during the execution of a service, each constraint evaluates to

true or *false*. This value can change during the execution. If a constraint has the value *true*, the referring policy is fulfilled. If a constraint has the value *false*, the policy is violated. The execution of a service is *correct* (according to the DecSerFlow model) at some point in time if all constraints (from the DecSerFlow model) evaluate to *true*. Similarly, a service has *completed correctly* if at the end of the execution all constraints evaluate to *true*. The goal of the execution of any DecSerFlow model is not to keep the values of all constraints *true* at all times during the execution. A constraint which has the value *false* during the execution is not considered an error. Consider for example the LTL expression $\Box(A \longrightarrow \Diamond B)$ where A and B are activities, i.e., each execution of A is eventually followed by B. Initially (before any activity is executed), this LTL expression evaluates to *true*. After executing A the LTL expression evaluates to *false* and this value remains *false* until B is executed. This illustrates that a constraint may be temporarily violated. However, the goal is to end the service execution in a state where all constraints evaluate to *true*.

To create constraints in DecSerFlow we use *constraint templates*. Each constraint template consists of a formula written in LTL and a graphical representation of the formula. An example is the "response constraint", which is denoted by a special arc connecting two activities A and B. The semantics of such an arc connecting A and B are given by the LTL expression $\Box(A \longrightarrow \Diamond B)$, i.e., any execution of A is eventually followed by (at least one) execution of B. We have developed a starting set of constraint templates and we will use these templates to create a DecSerFlow model. This set of templates is inspired by a collection of specification patterns for model checking and other finite-state verification tools [9]. Constraint templates define various types of dependencies between activities at an abstract level. Once defined, a template can be reused to specify constraints between activities in various DecSerFlow models. It is fairly easy to change, remove and add templates, which makes DecSerFlow an "open language" that can evolve and be extended according to the demands from different domains.[1] In the initial set of constraint templates we distinguish three groups: (1) "existence", (2) "relation", and (3) "negation" templates. Because a template assigns a graphical representation to an LTL formula, we will refer to such a template as a formula.

Before giving an overview of the initial set of formulas and their notation, we give a small example explaining the basic idea. Figure 2 shows a DecSerFlow model consisting of four activities: A, B, C, and D. Each activity is tagged with a constraint describing the number of times the activity should be executed, these are the so-called "existence formulas". The arc between A and B is an example of a "relation formula" and corresponds to the LTL expression discussed before: $\Box(\ A \longrightarrow \Diamond\ B\ )$. The connection between C and D denotes another "relation formula": $\Diamond\ D \longrightarrow \Diamond\ C$, i.e., if D is executed at least once, C is also executed at least once. The connection between B and C denotes a "negation formula"

[1] Note that we have developed a graphical editor for DecSerFlow that supports the creation of user defined templates, i.e., the user can define the graphical representation of a generic constraint and give its corresponding semantics in terms of LTL.

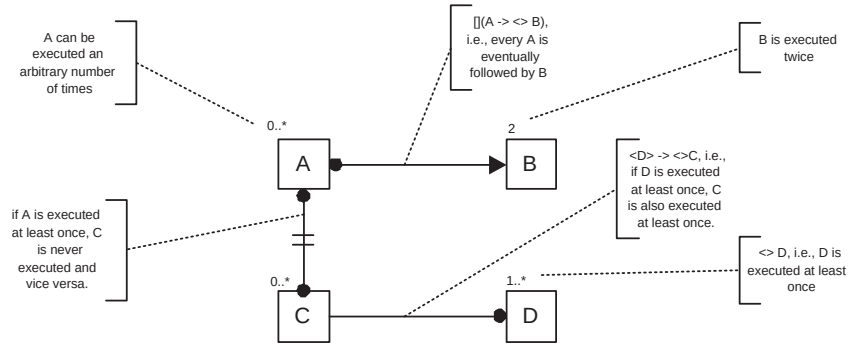


Fig. 2. A DecSerFlow model showing some example notations.

(the LTL expression is not show here). Note that it is not easy to provide a classical procedural model (e.g., a Petri net) that allows for all behaviour modeled Figure 2.

Existence formulas. Figure 3 shows the so-called "existence formulas". These formulas define the possible number of executions (cardinality) of an activity. For example, the first formula is called *existence*. The name and the formula heading are shown in the first column. From this, we can see that it takes one parameter (A), which is the name of an activity. The body of the formula is written in LTL and can be seen in the second column. In this case the LTL expression $\Diamond(activity == A)$ ensures that the activity given as the parameter A will execute at least once. Note that we write $\Diamond(activity == A)$ rather than $\Diamond(A)$. The reason is that in a state we also want to access other properties, i.e., not just the activity name but also information on data, time, and resources. Therefore, we need to use a slightly more verbose notation ($activity == A$). The diagram in the third column is the graphical representation of the formula, which is assigned to the template. Parameter A is an activity and it is represented as a square with the name of the activity. The constraint is represented by a cardinality annotation above the square. In this case the cardinality is at least one, which is represented by *1..**. The first group of existence formulas are of the cardinality "N or more", denoted by *N..**. Next, the formula *absence* ensures that the activity should never execute in the service. The group of formulas with names *absence_N* uses negations of *existence_N* to specify that an activity can be executed at most *N-1* times. The last group of existence formulas defines an exact number of executions of an activity. For example, if a constraint is defined based on the formula *exactly_2*, the referring activity has to be executed exactly two times in the service.

Relation formulas. Figure 4 shows the so-called "relations formulas". While an "existence formula" describes the cardinality of one activity, a "relation formula" defines relation(s) (dependencies) between two activities. All relation formulas

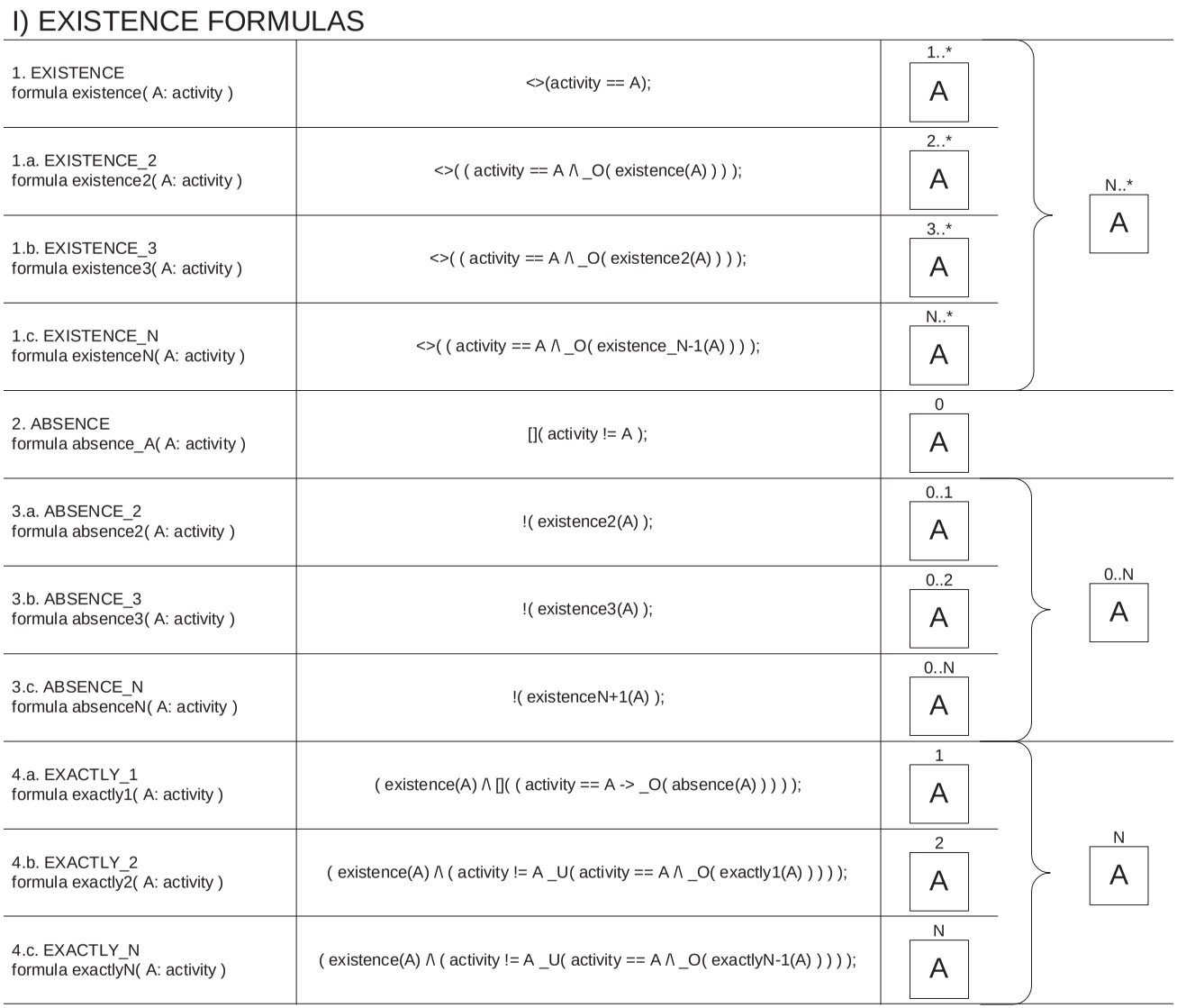


Fig. 3. Notations for the “existence formulas”.

II) RELATION BETWEEN EVENTS FORMULAS

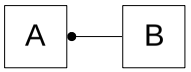
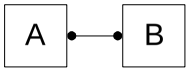
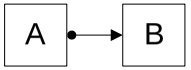
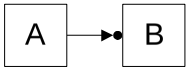
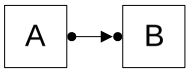
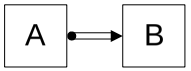
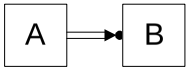
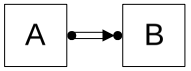
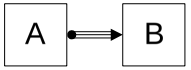
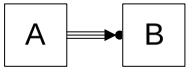
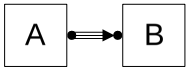
1. RESPONDED EXISTENCE formula existence_A_response_B(A: activity, B: activity)	(existence_A(A) -> existenceA(B));	
2. CO-EXISTENCE formula co_existence_A_and_B(A: activity, B: activity)	(existence(A) <-> existence(B));	
3. RESPONSE formula A_response_B(A: activity, B: activity)	[]((activity == A -> existence(B)));	
4. PRECEDENCE formula A_precedence_B(A: activity, B: activity)	(existence_A(B) -> (!(activity == B) _U activity == A));	
5. SUCCESSION formula A_succession_B(A: activity, B: activity)	(A_response_B(A,B) /\ A_precedence_B(A,B));	
6. ALTERNATE RESPONSE formula A_alternate_response_B(A: activity, B: activity)	(A_response_B(A,B) /\ B_always_between_A(A,B)*);	
7. ALTERNATE PRECEDENCE formula A_alternate_precedence_B(A: activity, B: activity)	(A_precedence_B(A,B) /\ B_always_between_A(B,A)*);	
8. ALTERNATE SUCCESSION formula A_alternate_succession_B(A: activity, B: activity)	(A_alternate_precedence_B(A,B) /\ A_alternate_response_B(A,B));	
9. CHAIN RESPONSE formula chain_A_response_B(A: activity, B: activity)	A_response_B(A,B) /\ []((activity == A -> _O(activity == B)));	
10. CHAIN PRECEDENCE formula chain_A_precedence_B(A: activity, B: activity)	(A_precedence_B(A,B) /\ []((_O(activity == B) -> activity == A)));	
11. CHAIN SUCCESSION formula chain_A_succession_B(A: activity, B: activity)	(chain_A_response_B(A,B) /\ chain_A_precedence_B(A,B));	
* subformula B_always_between_A(A: activity, B: activity)	[]((activity == A -> _O(A_precedence_B(B,A))));	

Fig. 4. Notations for the “relation formulas”.

have two activities as parameters and two activities in the graphical representation. The line between the two activities in the graphical representation should be unique for the formula, and reflect the semantics of the relation. The *responded existence* formula specifies that if activity A is executed, activity B also has to be executed either before or after the activity A. According to the *co-existence* formula, if one of the activities A or B is executed, the other one has to be executed also.

While the previous formulas do not consider the order of activities, formulas *response*, *precedence* and *succession* do consider the ordering of activities. Formula *response* requires that every time activity A executes, activity B has to be executed after it. Note that this is a very relaxed relation of response, because B does not have to execute immediately after A, and another A can be executed between the first A and the subsequent B. For example, the execution sequence *[B,A,A,A,C,B]* satisfies the formula *response*. The formula *precedence* requires that activity B is preceded by activity A. i.e., it specifies that if activity B was executed, it could not have been executed until the activity A was executed. According to this formula, the execution sequence *[A,C,B,B,A]* is correct. The combination of the *response* and *precedence* formulas defines a bi-directional execution order of two activities and is called *succession*. In this formula, both *response* and *precedence* relations have to hold between the activities A and B. Thus, this formula specifies that every activity A has to be followed by an activity B and there has to be an activity A before every activity B. For example, the execution sequence *[A,C,A,B,B]* satisfies the *succession* formula.

Formulas *alternate response*, *alternate precedence* and *alternate succession* strengthen the *response*, *precedence* and *succession* formulas, respectively. If activity B is *alternate response* of the activity A, then after the execution of an activity A activity B has to be executed and between the execution of each two activities A at least one activity B has to be executed. In other words, after activity A there must be an activity B, and before that activity B there can not be another activity A. The execution sequence *[B,A,C,B,A,B]* satisfies the *alternate response*. Similarly, in the *alternate precedence* every instance of activity B has to be preceded by an instance of activity A and the next instance of activity B can not be executed before the next instance of activity A is executed. According to the *alternate precedence*, the execution sequence *[A,C,B,A,B,A]* is correct. The *alternate succession* is a combination of the *alternate response* and *alternate precedence* and the sequence *[A,C,B,A,B,A,B]* would satisfy this formula.

Even more strict ordering relations formulas are *chain response*, *chain precedence* and *chain succession*, which require that the executions of the two activities (A and B) are next to each other. In the *chain response* the next activity after the activity A has to be activity B and the execution *[B,A,B,C,A,B]* would be correct. The *chain precedence* formula requires that the activity A is the first preceding activity before B and, hence, the sequence *[A,B,C,A,B,A]* is correct. Since the *chain succession* formula is the combination of the *chain response* and *chain precedence* formulas, it requires that activities A and B are always exe-

cuted next to each other. The execution sequence *[A,B,C,A,B,A,B]* is correct with respect to this formula.

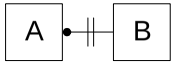
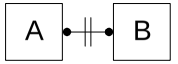
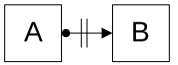
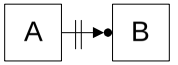
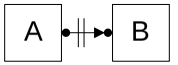
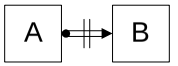
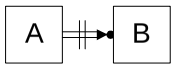
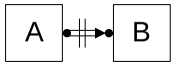
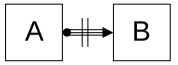
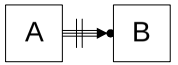
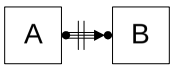
Negation formulas. Figure 5 shows the "negation formulas", which are the negated versions of the "relation formulas". The first two formulas negate the *responded existence* and *co-existence* formulas. The *responded absence* formula specifies that if activity A is executed activity B must never be executed (not before nor after the activity A). The *not co-existence* formula applies *responded absence* from A to B and from B to A. However, if we look at the *responded absence* formula we can see that if existence of A implies the absence of B and we first execute activity B, it will not be possible to execute activity A anymore because the formula will become permanently incorrect. This means that the formula *responded absence* is symmetric with respect to the input, i.e., we can swap the roles of A and B without changing the outcome. Therefore formula *responded absence* will be skipped and we will use only the *not co-existence* formula. The graphical representation is a modified representation of the *co-existence* formula with the negation symbol in the middle of the line. An example of a correct execution sequence for the formula *not co-existence* is *[A,C,A,A]*, while the sequence *[A,C,A,A,B]* would not be correct.

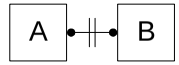
The *negation response* formula specifies that after the execution of activity A, activity B can not be executed. According to the formula *negation precedence* activity B can not be preceded by activity A. These two formulas have the same effect because if it is not possible to have activity B executed after activity A, then it is not possible to have activity A executed before activity B. Since the formula *negation succession* combines these two formulas, it also has the same effect and we will use only the *negation succession* formula. The graphical representation of this formula is a modified representation of the *succession* formula with a negation symbol in the middle of the line. The execution sequence *[B,B,C,A,C,A,A]* is an example of a correct sequence, while *[A,C,B]* would be an incorrect execution.

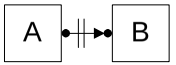
Formulas *negation alternate response*, *negation alternate precedence* and *negation alternate succession* are easy to understand. The formula *negation alternate response* specifies that the activity B can not be executed between the two subsequent executions of the activity A. According to this formula the execution sequence *[B,A,C,A,B]* is correct. In the case of the *negation alternate precedence* activity A can not be executed between two subsequent executions of the activity B. The execution sequence *[A,B,C,B,A]* is correct for *negation alternate precedence*. The formula *negation alternate succession* requires both *negation alternate response* and *negation alternate precedence* to be satisfied. An example of a correct execution sequence for the *negation alternate succession* formula is *[B,C,B,A,C,A]*. Graphical representations of these three formulas are similar to the representations of *alternate response*, *alternate precedence* and *alternate succession* with the negation symbol in the middle of the line.

The last three formulas are negations of formulas *chain response*, *chain precedence* and *chain succession*. According to the formula *negation chain response*, activity B can not be executed directly after the activity A. Formula *negation*

III) NEGATION RELATION BETWEEN EVENTS FORMULAS

12.a. RESPONDED ABSENCE formula existence_A_response_notB(A: activity, B: activity)	(existence_A(A) -> absence(B));	
12.b. NOT CO_EXISTENCE formula existence_A_response_notB(A: activity, B: activity)	(existence_A_response_notB(A,B) /\ existence_A_response_notB(B,A));	
13.a. NEGATION RESPONSE formula A_response_notB(A: activity, B: activity)	[]((activity == A -> absence(B)));	
13.b. NEGATION PRECEDENCE formula notA_precedence_B(A: activity, B: activity)	[]((existence(B) -> activity != A));	
13.c. NEGATION SUCCESSION formula notA_succession_notB(A: activity, B: activity)	(A_response_notB(A,B) /\ notA_precedence_B(A,B));	
14. NEGATION ALTERNATE REPONSE formula A_not_alternate_response_B(A: activity, B: activity)	B_never_between_A(A,B)**;	
15. NEGATION ALTERNATE PRECEDENCE formula A_not_alternate_precedence_B(A: activity, B: activity)	B_never_between_A(B,A)**;	
16. NEGATION ALTERNATE SUCCESSION A_not_alternate_succession_B(A: activity, B: activity)	(A_not_alternate_precedence_B(A,B) /\ A_not_alternate_response_B(A,B));	
17.a.. NEGATION CHAIN RESPONSE formula chain_A_response_notB(A: activity, B: activity)	[]((activity == A -> _O(activity != B)));	
17.b. NEGATION CHAIN PRECEDENCE formula chain_notA_precedence_B(A: activity, B: activity)	[]((_O(activity == B) -> activity != A));	
17.c. NEGATION CHAIN SUCCESSION formula chain_A_notsuccession_B(A: activity, B: activity)	(chain_A_response_notB(A,B) /\ chain_notA_precedence_B(A,B));	
** subformula B_never_between_A(A: activity, B: activity)	[]((activity == A -> _O((<>(activity == A) -> (activity != B _U activity == A)))));	





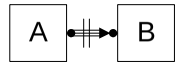


Fig. 5. Notations for the "negations formulas".

chain precedence specifies that activity B can never be directly preceded by activity A. These two formulas have the same effect because they forbid the activities A and B to be executed directly next to each other. Since the formula *negation chain succession* requires both *negation chain response* and *negation chain precedence* to be executed, these three formulas all have the same effect and we will use only *negation chain succession*. The graphical representation of this formula is a modified version of the representation of the *chain succession* formula with the negation symbol in the middle of the line. The execution sequence *[B,A,C,B,A]* is correct according to the *negation chain succession* formula, while the sequence *[B,A,B,A]* would not be correct.

Figures 4 and 5 only show binary relationships. However, these can easily be extended to deal with more activities. Consider for example the *response* relationship, i.e., $\Box(A \longrightarrow \Diamond B)$. We will allow multiple arcs to start from the same dot, e.g., an arc to B, C, and D. The meaning is $\Box(A \longrightarrow \Diamond(B \vee C \vee D))$, i.e., every occurrence of A is eventually followed by an occurrence of B, C, or D. Moreover, as indicated before, the set of formulas is not fixed and we also aim at supporting data. In fact, we have defined more formulas than the ones shown in figures 3, 4, and 5. For example, the *mutual substitution* relation formula specifies that at least one of two activities should occur (i.e., $\Diamond(A \vee B)$).

After this introduction to DecSerFlow we specify a concrete example. The interested reader is referred to a technical report with more information about DecSerFlow [5]. Moreover, for more information on ConDec, the sister language of DecSerFlow aiming a teamwork and workflow flexibility, we refer to [17].

3 Modelling Services With DecSerFlow: The Acme Travel Example

In this section we use the "Acme Travel Company case" to illustrate DecSerFlow. The description of the business process of the Acme Travel service is adopted from [19] is as follows:

1. Acme Travel receives an itinerary from Karla, the customer.
2. After checking the itinerary for errors, the process determines which reservations to make, sending simultaneous requests to the appropriate airline and hotel agencies to make the appropriate reservations [2].
3. If any of the reservation activities fails, the itinerary is cancelled by performing the "compensate" activity and Karla is notified of the problem.
4. Acme Travel waits for confirmation of the two reservation requests.
5. Upon receipt of confirmation, Acme Travel notifies Karla of the successful completion of the process and sends her the reservation confirmation numbers and the final itinerary details.

[2] The original Acme Travel service business process consists of three possible bookings: airline, hotel and vehicle. However, for the simplicity, we consider only the possibilities to book airline and hotel.

6. Once Karla is notified of either the success or failure of her requested itinerary, she may submit another travel request.

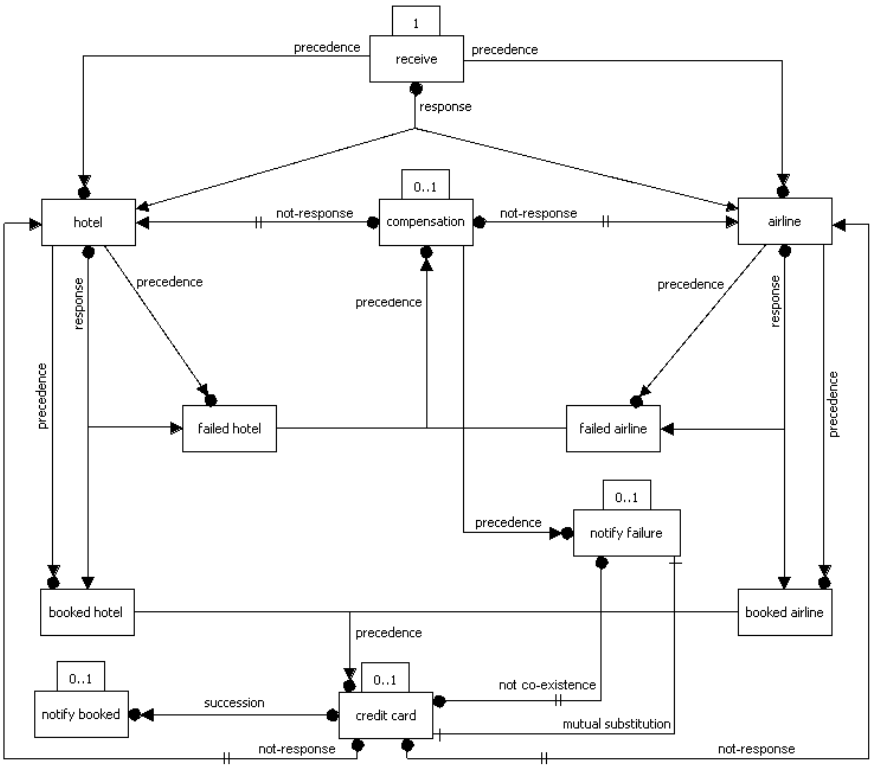


Fig. 6. DecSerFlow - Acme Travel Company

Figure 6 shows the DecSerFlow model of the Acme business process. We first define the possible activities within the service to model the business process of Acme. In this case, we define eleven activities:

receive - A request for booking is received from the customer;
airline - A request for booking is sent to an airline reservation service;
hotel - A request for booking is sent to a hotel reservation service;
booked hotel - A hotel reservation service sends a positive response for a requested booking, i.e., the hotel can be booked;
failed hotel - A hotel reservation service sends a negative response for a requested booking, i.e., the hotel cannot be booked;
booked airline - An airline reservation service sends a positive response for a requested booking, i.e., the airline can be booked;

failed airline - An airline reservation service sends a negative response for a requested booking, i.e., the airline cannot be booked;
compensation - The whole booking has failed;
notify failure - Notify the customer that the booking has failed;
credit card - Register and charge a successful booking; and
notify booked - Notify the customer that the booking was successful.

In principle, a DecSerFlow model consisting only of a set of activities is a correct model. If a DecSerFlow model consisting only of eleven activities would be implemented in the Acme service, it would be possible that the service executes any of the eleven activities, an arbitrary number of times, in an arbitrary order. It would also be possible not to execute any activity. To prevent such an "chaotic" behavior of the service, we can add *constraints* to the service process model. A constraint in service represents a rule that the service execution has to fulfill. The Acme DecSerFlow model shown in Figure 6 uses two of the three types of constraint formulas mentioned before: "existence" and "relation" constraints.

In Section 2, we presented several standard "existence" constraints. These constraints define the possible number of executions of an activity in a service. We refer to the possible number of executions of an activity in a service as the cardinality of that activity. Without any constraints in the service model, an activity can be executed an arbitrary number of times - the activity has the cardinality of (0..*). The "existence" constraints are graphically represented as cardinalities above activities (cf. Figure 6). Activity *receive* has the constraint *exactly_1* (cf. Section 2), and specifies that this activity will be executed exactly once in one instance (per one customer request) of the Acme service. Because the booking request can succeed or fail, but not both, activities *compensation*, *notify failure*, *credit card*, and *notify booking* have the constraint "absence_2", which specifies that each of these activities will be executed at most once. We do not define any "existence" constraints on activities *hotel* and *airline* and thus allow these two activities to execute an arbitrary number of times in the Acme service. If the customer does not wish to book a hotel or an airline, the Acme service will not execute the corresponding activity. In case that a booking of a hotel or an airline is requested, the booking request might be sent multiple times until the booking succeeds or fails. A booking of a hotel or an airline will be followed with a positive activity (i.e., *booked hotel* or *booked airline*) or a negative activity (i.e., *failed hotel* or *failed airline*). Therefore, activities *booked hotel*, *booked airline*, *failed hotel* and *failed airline* also can be executed an arbitrary number of times in the service.

The Acme DecSerFlow model as defined so far - only consisting of a set of activities and "existence" constraints - is a correct model. If this model would be implemented in the Acme service, the service could execute its activities in an arbitrary order, complying with the execution cardinality of each activity, as defined with "existence" constraints.

To define relations between activities in the service (and implicitly their possible order) we use the so-called "relation" constraints as defined in Section 2. Unlike "existence" constraints that were defined for single activities (unary),

"relation" constraints define relations between two or more activities (e.g., a binary relationship).

After receiving the booking request from the customer, the request is checked. The customer can request to book a hotel and an airline for a destination, or only one of these. The constraint *response* from the activity *receive* is a so-called branched constraint. It has two branches: one to the activity *hotel* and the other to the activity *airline*. This branched response specifies that after the activity *receive* is executed, eventually there will be at least one execution of one of the activities *hotel* and *airline*. It is still possible that both of the activities *hotel* and *airline* execute an arbitrary number of times, as long as at least one of them executes after the activity *receive*. However, since it would not be desirable to execute any of the activities *hotel* and *airline* before the activity *receive*, we add two *precedence* constraints: (1) the *precedence* constraint between activities *receive* and *hotel* specifies that the activity *hotel* cannot execute before the activity *receive* executes, and (2) the *precedence* constraint between activities *receive* and *airline* specifies that the activity *airline* cannot execute before the activity *receive* executes. The branched constraint *response* and the two *precedence* constraints between activities *receive*, *hotel* and *airline* specify that activities *hotel* and *airline* will not execute until the activity *receive* executes, and that after the activity *receive* executes, at least one of the activities *hotel* an *airline* will execute. Activities *hotel* and *airline* can an arbitrary number of times and in an arbitrary order.

Activities *booked hotel* and *failed hotel* handle the possible responses of a hotel reservation service on the request of Acme service to book a hotel (which is sent when the activity *hotel* is executed). With the branched *response* constraint from the activity *hotel* we specify that after every execution of this activity, at least one of the activities *booked hotel* and *failed hotel* will execute. Note that, due to errors, this constraint allows for some requests for the hotel reservation to remain without response. Logically, with the two *precedence* constraints between the activity *hotel* and activities *booked hotel* and *failed hotel* we prevent that either of the activities *booked hotel* and *failed hotel* execute before the activity *hotel* executes. This is necessary, since the response from the hotel reservation service can not arrive before a reservation request is sent. The same constraints are added between activities *airline*, *booked airline* and *failed airline*, because the communication of the Acme service with the airline service is the same like the communication with the hotel reservation service.

Only after receiving at least one of the two negative responses (activities *failed hotel* and *failed airline*), the Acme service can cancel the whole booking by executing the activity *compensation*. This is specified by the branched *precedence* constraint between the activity *compensation* and activities *failed hotel* and *failed booking*. After the *compensation* activity is executed, activities *hotel* and *airline* can not execute again in the service, because the whole booking is cancelled. The two *not-response* constraints between the activity *compensation* and activities *hotel* and *airline*, make sure that after the activity *compensation* executes, none of the activities *hotel* and *airline* can execute. The *precedence*

constraint between activities *notify failure* and *compensation* specifies that the activity *notify failure* cannot execute before the activity *compensate*. Note that after the activity *compensation* executes, there might still be some responses arriving from the reservation services. If an satisfactory booking response arrives after the activity *compensate* is executed, the Acme service can still decide to accept the booking. This is why the activity *notify failure* does not always necessarily execute after the activity *compensate*.

After at least one positive reservation response arrives, the Acme service can decide to accept and finalize the whole booking. This is specified with the branched *precedence* constraint between the activity *credit card* and activities *booked hotel* and *booked airline*. After the booking is charged, the new requests will not be sent to the hotel and airline reservation services, i.e., the activities *hotel* and *airline* cannot execute after the activity *credit card*. This is achieved with the two *not-response* constraints between the activity *credit card* and activities *hotel* and *airline*. Only and always after the booking is charged, the customer will be notified about the successful booking. The *succession* constraint between activities *credit card* and *notify booked* specifies that the activity *notify booked* cannot execute before the activity *credit card* and that it will have to execute after the activity *credit card*.

To conclude the booking of a customer, the Acme service will either accept or decline the requested booking. This means that in the service either one of the taks *notify failure* and *credit card* will execute. Note that even after the activity *compensation*, Acme can still receive an positive reservation response and accept the booking. The *not co-existence* constraint between activities *credit card* and *notify failure* specifies that only one of these two activities can execute in the service because it is not possible to both charge the booking and notify the customer about failure. However, eventually one of the activities *credit card* or *notify failure* will execute, as specified with the *mutual substitution* constraint between these two activities.

Note that the Acme service model in Figure 6 allows for many alternative executions of the service. For example, it is possible to handle the both late and lost reposes of reservation services. It is also possible to send requests to different reservation services regardless the order of the reception of responses. Even after the cancellation has started by executing the activity consumption, it is still possible to receive a positive response and successfully finalize the requested booking.

It is important to note that Figure 6 uses a declarative style of modelling. The DecSerFlow model allows for much more variability than a typical procedural process model (e.g., a BPEL specification). However, because the language is extendible it is possible to add constructs one can find in traditional languages, e.g., it is relatively easy to add the "place concept" from Petri nets or the "sequence concept" from BPEL. As a result, DecSerFlow can be applied using different styles ranging from highly procedural to highly declarative.

4 Enacting DecSerFlow Models of Web Services

Every DecSerFlow model consists of a set of activities and constraints. Constraints define rules that the service has to fulfill. At the end of the service execution all constraints should be fulfilled. The semantics of a constraint is defined with the LTL formula that is assigned to it. We use these LTL formulas to execute a DecSerFlow model. Every LTL formula can be translated into an Buchi automaton [10]. There are several algorithms for translating LTL expressions into Buchi automata. Different algorithms have been studied in the field of *model checking* [15]. The SPIN tool [14] is one of the most widely used tools for model checking. Using SPIN, one can develop a model of a system in Promela (PROcess MEta LAnguage) [14]. To check the model with respect to some requirements, we can write these requirements as LTL expressions. SPIN can automatically verify the correctness of the specified LTL requirements in the developed Promela model. For verification purposes, SPIN uses an algorithm for translating LTL expressions to Buchi automata [10].

A DecSerFlow model typically has multiple constraints. All of the constraints need to be taken into account at any moment of the service execution. For this purpose we can take one of the two strategies: (1) we can construct an automaton for each of the LTL expressions and then execute these automatons in parallel, or (2) construct and execute a single automaton for the whole model (i.e., construct an automaton for the conjunction of all LTL expressions).

When executing a service by executing referring Buchi automaton(s), we have to deal with two problems. First, the standard algorithms (e.g., the one presented in [10]) construct a *non-deterministic* finite automaton. A nondeterministic finite automaton is a finite state machine where for each pair (state, input symbol) there may be several possible next states. This means that for each pair (state of a DecSerFlow model, executed activity) there may be several possible next states of the DecSerFlow model. This is a problem because, given a execution history, it is not always possible to pinpoint the current state in the automaton. Second, algorithms such as the one presented in [10] construct a finite automaton for *infinite words*. Because we assume that a service will eventually finish with the execution, we have to use an automaton that can read finite words.

4.1 Executing a Non-Deterministic Automaton

In this section we describe a simple algorithm that can be used to successfully execute a non-deterministic automaton in the context of the execution of a DecSerFlow model. We use a simple example of a model with three activities, as shown in Figure 7(a). This model consists of activities *curse*, *pray*, and *bless* and a constraint *response* between activities *curse* and *pray*. All three activities can be executed an arbitrary number of times because there are no "existence" constraints to specify cardinalities of activities. Constraint *response* between activities *curse* and *pray* specifies that, every time a person curses, (s)he should eventually pray after this.

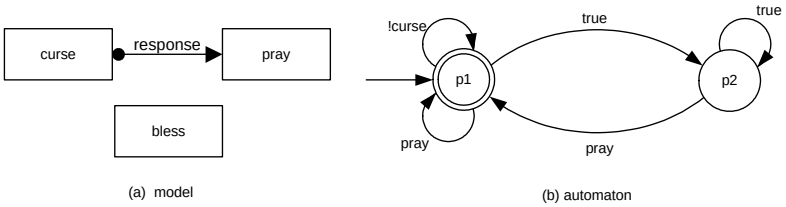


Fig. 7. A simple DecSerFlow model.

For this model we construct the automaton [10], as shown in Figure 7(b). This automaton consists of two states: *p1* (accepting and initial state) and *p2*. In the beginning we assume the automaton to be in the initial state *p1*. There are three transitions possible from this state: (1) transition with the label *pray* is applied when the activity *pray* is executed, (2) transition with the label *!curse* is applied when activities *pray* or *bless* are executed, and (3) transition *true* leads to the state *p2* and is applied when any of the activities is executed. In the state *p2* two transitions are possible: (1) transition with the label *true* is applied when any of the three activities are executed and (2) transition *pray* leads to the state *p1* and is applied when the activity *pray* executes.

In a simplified case of a deterministic automaton, we would execute the model by checking at which state the automaton currently is, i.e., we would constantly store the information about the *current state* of the automaton. If the automaton is in an accepting state, the constraint(s) are fulfilled and vice versa. When executing an activity, the automaton would simply move to the next state by a transition that can be applied for that activity. When executing an activity in the case of a non-deterministic automaton, there can be *multiple* possible next states to move to. The automaton shown in Figure 7(b) is a non-deterministic automaton. Take, for example, the situation when the automaton is in the state *p1* and the activity *pray* executes. In this case (at the state *p1*), we could apply any of the transitions *pray* (the automaton remains in the state *p1*), *!curse* (the automaton remains in the state *p1*), or *true* (the automaton changes the state to *p2*) - this is a non-deterministic situation. Because we use the current state of the automaton to determine if the constraint(s) are fulfilled or not and the next possible activities, the information about the current state of the automaton is important.

A simple solution for the execution of a non-deterministic automaton is to consider a set of possible current states[3] instead of a single current state. In the situation described above (when the activity *pray* is executed in the state *p1*) we would consider the automaton to transfer to the set of possible states {p1,p2}. We take the optimistic approach and consider an automaton to be in an accepting state if any of the states in the set of current possible states is an accepting state of the automaton. Figure 8 shows the algorithm for the execution of a non-deterministic automaton. We use two data types: (1) *state* consists

[3] This set can have at most all states of the automaton.

of an array of incoming transitions and an array of outgoing transitions and (2) *transition* has a label (e.g., *!curse*), source state and target state. Function *nextState* generates an array of states (a set of possible next states) given the array (set) of *current* possible states and the *activity* name. This function loops through the array of *current* states. For each current state it loops through all the *outgoing* transitions. For each of the outgoing transitions it checks is the label of the transition complies with the *activity* name. If (1) the activity is accepted by the transition label and (2) the target state of the transition is not in the array of the next states, the target state of the transition is added to the array of the next states.

```
State {                    Transition{
  Transition[] in;           String label;
  Transition[] out;          State source;
}                            State target;
                           }
1  State[] nextState(State[] current, String activity){
2    State[] next;
3     for i = 0 to current.length - 1 do{          // Look at all current possible states.
4          State curr = current[i];                // For every current state
5        for j = 0 to curr.out.length - 1 {       // look at all out-transitions.
6            Transition out = curr.out[j];        // For every out-transition,
7            if (out.label parses activity)       // if the out-transition suits the activity,
8             then if ( out.target not in  next )    // if the target state is not already in the set of new possible states
9                       then next = next +  out.target;  // add the target state to the set of new possible states.
10       }
11    }
12    return next;
13 }
```

Fig. 8. Execution of a non-deterministic automaton.

Table 1 shows the execution of the automaton shown in Figure 7 (b). At the beginning, the set of possible states contains all initial states, which is in this case {p1}. For example, if activity *bless* is executed in the initial state, then the automaton could apply transition *!curse* (and stay in the state *p1*) or it could apply transition *true* (and move to the state *p2*). Thus, if the activity *bless* is executed when the automaton is in a state in {p1} (i.e., the initial set of possible states), the automaton can move to any state in the the set of new possible states {p1,p2}. If the automaton is, for example, in the set of possible states {p1,p2} and activity *bless* is executed, the automaton transfers to the set of possible states {p1,p2} that is formed as intersection of sets of possible states for each of the starting states *p1* ({p1,p2}) and *p2* ({p2}). Since *p1* is the accepting state of the automaton in Figure 7 (b), we consider the execution of the DecSerFlow model from Figure 7 (a) to be correct (i.e., all constraints are fulfilled) if the set of current possible states contains the state *p1*. Thus, we consider the model to be executed correctly, if the set of current possible states of the automaton is either {*p1*} or {*p1,p2*}.

Table 1. Execution of the non-deterministic automaton in Figure 7

	automaton possible states		automaton possible states
nr.	from	activity	to
1	{p1}	bless	{p1,p2}
2	{p1}	curse	{p2}
3	{p1}	pray	{p1,p2}
4	{p2}	bless	{p2}
5	{p2}	curse	{p2}
6	{p2}	pray	{p1}
7	{p1,p2}	bless	{p1,p2}
8	{p1,p2}	curse	{p2}
9	{p1,p2}	pray	{p1,p2}

4.2 Executing Finite Traces

The algorithm presented in [10] is originally dedicated for model checking of concurrent systems. Because these systems are not designed to halt during normal execution, the resulting automaton is an automaton over *infinite words (traces, runs)*. An infinite trace is accepted by the automaton [10] *iff* it visits an accepting state infinitely often. This type of acceptance cannot be applied for the case of service execution, because we require that such an execution will eventually complete.

There are two strategies that can enable checking of the acceptance of an finite trace in an automaton generated by [10]: (1) we can introduce special invisible "end" activity and constraint in a DecSerFlow model before the automaton is created or (2) we can adopt a modified version of this algorithm, which reads *finite words (traces, runs)* [11].

In the first strategy we use the original algorithm for the generation of automata, but we slightly change the DecSerFlow model before creating the automaton. To be able to check if a finite trace is accepting, we add one "invisible" activity and one "invisible" constraint to every DecSerFlow model and then construct the automaton. With this we specify that each execution of the model will eventually end. We introduce an "invisible" activity e, which represents the *ending* activity in the model. We use this activity to specify that the service will end - the *termination* constraint. This constraint has the LTL formula $\Diamond e \wedge (\Box(e \longrightarrow \bigcirc e))$, and it specifies that: (1) the service will eventually end - the "invisible" activity e will eventually be executed, and (2) after this activity, no other activity will be executed but the activity e itself, infinitely often. Take, for example, a simple DecSerFlow model with one constraint *existence(receive)*, (i.e., $\Diamond receive$), which specifies that the activity *receive* will execute at least once. To execute this model we first add the termination constraint and consider a conjunction of these two constraints: $\Diamond receive \wedge \Diamond e \wedge (\Box(e \longrightarrow \bigcirc e))$. This conjunction ensures that the trace will have the prefix required by the original DecSerFlow model and an infinite suffix containing only the "ending"

activity *e*. The whole conjunction is then translated into an automaton using the original algorithm [10]. We check the acceptance of the finite trace (prefix) of the original DecSerFlow model by checking if the automaton is in a so-called *end* state: (1) if the automaton is in an accepting state and (2) if from this moment an accepting state can be visited infinitely often only by executing the "ending" activity *e*. To prevent deadlocks, the automaton is purged (before the execution) from the states from which none of the *end* states is reachable.

As the second strategy, we can use a modification of the original algorithm. The original algorithm for translating LTL formulas to Buchi automatons [10] is modified to be used for verification of *finite* executions of software programs [11]. The algorithm for translating LTL formulas into automatons over finite words introduces a change into the acceptance criteria of the original algorithm [11]. However, this algorithm assumes that any program would have to start executing, i.e., it does not consider empty traces. Therefore, an initial state is not accepting in some cases where it should be accenting for an empty trace. However, we assume that an "empty" execution of a DecSerFlow model (that does not violate any constraint) is in principle an accepting execution. Therefore, we introduce an "invisible" initial activity *init*. Using LTL we require this to be the first activity. Moreover, to any execution sequence we add a prefix containing one *init* activity, i.e., before the service can start, activity *init* is automatically executed. After this, it is possible to determine if the state where no activities have been executed (empty trace) is in an accepting state or not.

After completing the DecSerFlow editor, we are currently experimenting with different ways in which we can build useful automatons for enactment. Since we are using LTL not just for analysis but as the care technology for the engine, we also have to address issues such as performance and reliability.

5 Conclusion

This paper advocated a more declarative style of modeling in the context of web services. Therefore, we proposed a new, more declarative language: DecSerFlow. Although DecSerFlow is graphical, it is grounded in temporal logic. It can be used for the *enactment* of processes, but it is also suited for the *specification* of a single service or a complete choreography.

Besides being *declarative*, the language is also *extendible*, i.e., it is possible to add new constructs without changing the core of the language. We have developed a graphical editor to support DecSerFlow. This editor allows users to specify service flows. Moreover, the user can add user-defined constraint templates by simply selecting a graphical representation and providing parameterized semantics in terms of LTL. Currently, we are working on an engine that is able to support enactment and monitoring. If a constraint is used for enactment, it is impossible to permanently violate a constraint because the system will not allow activities that violate this constraint. If a constraint is used for monitoring, the system will allow the violation of this constraint. However, the engine will issue a warning and log the violation. The automatic construction of an automaton

suitable for enactment and on-the-fly monitoring is far from trivial as shown in Section 4.

There is also a very interesting link between DecSerFlow and *process mining* [6]. In [3] we showed that it is possible to translate abstract BPEL into Petri nets and SOAP messages exchanged between services into event logs represented using our MXML format (i.e., the format used by ProM `www.processmining.org`). As a result, we could compare the modeled behavior (in terms of a Petri net) and the observed behavior (in some event log) using the conformance checker [18]. A similar approach can be followed by using the LTL checker in ProM [1]. Using the LTL checker it is possible to check LTL formulas over event logs (e.g., monitored SOAP messages). In principle it is possible to use the LTL formulas generated based on the DecSerFlow specification and load them into the LTL checker in ProM. This allows the users of ProM to specify constraints graphically rather than using the textual language that is used now.

References

1. W.M.P. van der Aalst, H.T. de Beer, and B.F. van Dongen. Process Mining and Verification of Properties: An Approach based on Temporal Logic. In R. Meersman and Z. Tari et al., editors, *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2005*, volume 3760 of *Lecture Notes in Computer Science*, pages 130–147. Springer-Verlag, Berlin, 2005.
2. W.M.P. van der Aalst, M. Dumas, A.H.M. ter Hofstede, N. Russell, H.M.W. Verbeek, and P. Wohed. Life After BPEL? In M. Bravetti, L. Kloul, and G. Zavattaro, editors, *WS-FM 2005*, volume 3670 of *Lecture Notes in Computer Science*, pages 35–50. Springer-Verlag, Berlin, 2005.
3. W.M.P. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, and H.M.W. Verbeek. Choreography Conformance Checking: An Approach based on BPEL and Petri Nets (extended version). BPM Center Report BPM-05-25, BPMcenter.org, 2005.
4. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
5. W.M.P. van der Aalst and M. Pesic. Specifying, Discovering, and Monitoring Service Flows: Making Web Services Process-Aware. BPM Center Report BPM-06-09, BPM Center, BPMcenter.org, 2006. http://www.BPMcenter.org/reports/2006/BPM-06-09.pdf.
6. W.M.P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.
7. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services, Version 1.1. Standards proposal by BEA Systems, International Business Machines Corporation, and Microsoft Corporation, 2003.
8. M. Dumas, W.M.P. van der Aalst, and A.H.M. ter Hofstede. *Process-Aware Information Systems: Bridging People and Software through Process Technology*. Wiley & Sons, 2005.

9. M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 411–420, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.
10. R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. 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*, pages 3–18, London, UK, 1996. Chapman & Hall, Ltd.
11. D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE'01)*, pages 412–416. IEEE Computer Society Press, Providence, 2001.
12. K. Havelund and G. Rosu. Monitoring Programs Using Rewriting. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE'01)*, pages 135–143. IEEE Computer Society Press, Providence, 2001.
13. K. Havelund and G. Rosu. Synthesizing Monitors for Safety Properties. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer-Verlag, Berlin, 2002.
14. G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, Boston, Massachusetts, USA, 2003.
15. E.M. Clarke Jr., O. Grumberg, and D.A. Peled. *Model Checking.* The MIT Press, Cambridge, Massachusetts and London, UK, 1999.
16. N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, and Y. Lafon. Web Services Choreography Description Language, Version 1.0. W3C Working Draft 17-12-04, 2004.
17. M. Pesic and W.M.P. van der Aalst. A Declarative Approach for Flexible Business Processes Management. In *BPM 2006 Workshops: International Workshop on Dynamic Process Management (DPM 2006)*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2006.
18. A. Rozinat and W.M.P. van der Aalst. Conformance Testing: Measuring the Fit and Appropriateness of Event Logs and Process Models. In C. Bussler et al., editor, *BPM 2005 Workshops (Workshop on Business Process Intelligence)*, volume 3812 of *Lecture Notes in Computer Science*, pages 163–176. Springer-Verlag, Berlin, 2006.
19. J. Snell. Automating business processes and transactions in Web services: An introduction to BPELWS, WS-Coordination, and WS-Transaction. http://www-128.ibm.com/developerworks/webservices/library/ws-autobp/, June 2006.
20. P. Wohed, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. Analysis of Web Services Composition Languages: The Case of BPEL4WS. In I.Y. Song, S.W. Liddle, T.W. Ling, and P. Scheuermann, editors, *22nd International Conference on Conceptual Modeling (ER 2003)*, volume 2813 of *Lecture Notes in Computer Science*, pages 200–215. Springer-Verlag, Berlin, 2003.