Continuous Orchestration of Web Services Via Planning

Piergiorgio Bertoli$^1$  Raman Kazhamiakin$^1$  Massimo Paolucci$^2$
Marco Pistore$^1$  Heorhi Raik$^1$  Matthias Wagner$^2$

1: FBK-Irst, via Sommarive 18, 38050, Trento, Italy
[bertoli, raman, pistore, raik]@fbk.eu

2: DoCoMo Euro-Labs, Landsberger Strasse 312, 80687 Munich, Germany
{paolucci, wagner}@docomolab-euro.com

Abstract

The service-oriented paradigm is rapidly emerging as the key approach to develop distributed business applications. Its enactment requires the ability to automatically coordinate existing services to realize novel and powerful desired functionalities, and planning-based solutions have proved to be strong candidates for this hard task. However, no current approach can satisfactorily coordinate stateful services that evolve continuously and indefinitely in an asynchronous way, such as e.g. notification services made increasingly available by business entities. This severely limits the practical applicability of current solutions. In this paper, we provide for the first time a full-fledged planning-based solution to the problem of continuously orchestrating stateful asynchronous services. To do so, we adopt a simple yet expressive requirement language, and we devise a novel planning algorithm that solves preference-ordered maintainability goals in the presence of exogenous events. Our approach is correct and complete, and our tests on a symbolic BDD-based implementation witness its ability and effectiveness in dealing with significant and realistic scenarios which no other current approach can tackle.

1. Introduction

By envisaging standards to publish and access services over the Web, the Service-Oriented Computing (SOC) paradigm promises a novel degree of interoperability between distributed applications that realize business processes. One cornerstone of SOC stands in the provision of novel and powerful desired functionalities, and planning-based solutions have proved to be strong candidates for this hard task. However, no current approach can satisfactorily coordinate stateful services that evolve continuously and indefinitely in an asynchronous way, such as e.g. notification services made increasingly available by business entities. This severely limits the practical applicability of current solutions. In this paper, we provide for the first time a full-fledged planning-based solution to the problem of continuously orchestrating stateful asynchronous services. To do so, we adopt a simple yet expressive requirement language, and we devise a novel planning algorithm that solves preference-ordered maintainability goals in the presence of exogenous events. Our approach is correct and complete, and our tests on a symbolic BDD-based implementation witness its ability and effectiveness in dealing with significant and realistic scenarios which no other current approach can tackle.

stateful protocols, such as those that can be specified using the BPEL (Andrews et al. 2003) standard language. This task is extremely complex, as it involves the ability to express preference-based requirements to account for the various possible reactions of orchestrated services, and to obtain compositions that embed arbitrary branching and cycling structures. In fact, so far only very few approaches, recasting the composition problem in terms of planning, are capable to build executable coordinations for stateful asynchronous services. Unfortunately, even those approaches take a fundamental simplifying assumption, namely that all the services being orchestrated admit some final ‘stable’ configurations which are used as goal targets for the orchestration. In general such assumption does not hold and is strongly limiting. One key usage of services is to faithfully represent the dynamic and uncontrollable evolution of the world (e.g. standard notification services report of travel delays and cancellations, overbookings, and so on); as such, a continuous and strenuous realignment of the orchestration is in order. That is, since in general ‘satisfactory’ situations are not final and stable, allowing the synthesis of continuous coordinations is essential to enact service composition in real settings.

In this paper we realize the synthesis of continuous coordinations based on the conceptual framework of (Pistore, Traverso, and Bertoli 2005), which recasts the composition problem in terms of planning; namely, we act at its core by adopting a very simple, yet expressive requirements language, and devising a novel planning algorithm. In particular, the requirement language expresses coordination constraints that are transformed into preference-ordered maintainability goals, and the algorithm deals with such goals in the presence of exogenous events (which encode independent asynchronous evolutions of services).

The paper is organized as follows. First, we introduce a motivating reference example. Then, we provide a formal background to the approach, we describe our requirements language and we discuss the encoding of service composition into planning. We then detail the planning algorithms, and present an empirical evaluation of a prototype implementation. Finally, we discuss related and future work.

2. Reference example

Our reference example is a variant of a well-known travel domain scenario, where the goal is to provide a composed
service that can deliver and manage travel packages consisting of flight tickets and hotel reservations.

Our version of the scenario involves five services, represented in Fig. 1, whose actual implementation can be carried out in a standard service description language such as e.g. BPEL. In particular, Fig. 1 (a) shows the expected customer interface (CI), which allows the user to book a compound travel package (if available), and then to receive information on its modifications (e.g., in case of the flight delays or cancellation). Figure 1 (b,c) represent the services for booking a flight ticket (FB) and for cancelling it (FC) respectively. The service depicted in Fig. 1 (d) shows a flight status notification service (FSN), which sends notifications about the flight delays or cancellations. Finally, Fig. 1 (e) represents a unique protocol (HR) managing in full a hotel reservation, i.e., able to create, modify, and delete it. Notice that some relevant service configurations are annotated by labels that identify specific situations, e.g. that a reservation has been created (CR) or deleted (DEL), or lack of availability (NAV).

The problem of providing a composed service in this scenario is complicated by two key factors. The first stands in the fact that the involved services have a sophisticated behavior: they are stateful and non-deterministic, their internal behavior is not directly observable, and message interactions are asynchronous. The second, crucial point that makes this case study really challenging is that, to keep the user in the control of the travel package at all times, we need to cover the whole process of reserving and managing the travel package. This means not only to book the hotel and flight reservation, but also to align further flight modifications with the modification of the hotel reservation and of the travel package. More precisely, we aim to build a composed process which should satisfy the following goals:

1. It should provide a way to transactionally book a hotel and reserve a flight, i.e. the hotel shall not be booked if the flight is not available and vice versa.
2. If the flight is notified to be delayed, the composed service has to provide a way to modify the hotel reservation (and the resulting offer) as well. If this is not possible, the reservations should be cancelled.
3. If the flight is notified to be cancelled, the other reservations should be cancelled as well.

Crucially, such requirements cannot be expressed in terms of finally achieving some state or outcome. Requirement (1) asks reaching an intermediate state where further events and actions may still take place, while (2,3) define reaction rules that the process should perform in order to handle specific uncontrollable events. Moreover, requirements (1,2) express potential alternatives that should be considered due to non-determinism of the services, and a preference order among them (booking is preferred to non-booking, and modification to cancellation). No current approach to service composition can tackle, for this kind of services, a combination of requirements like the ones above. In order to describe our contribution in this direction, we devote the next section to recap some fundamental background.

Figure 1: Component services in the travel domain scenario

Figure 2: The approach

3. Background

Our conceptual approach to service composition, recasting the problem as one of planning, follows (Pistore, Traverso, and Bertoli 2005) and is represented in Fig. 2. The starting point to composition is the presence of some component services $W_1, \ldots, W_n$, which are stateful asynchronous entities expressed in languages such as e.g. BPEL, and of a composition goal $p$ that specifies declarative and behavioral constraints for a novel desired service. Such new composite

---

1In the graphical representation, input and output operations are prepended by "?" and "!" respectively. For space reasons, we abstract away from details related to the data manipulated by the services.
service $W$ must be realized as an orchestrator for the components, and it is the result of the composition proper. One key idea in this approach is that both the component services and the orchestration service are expressed as finite state transition systems (STSs) that evolve by reacting to external inputs, and producing outputs. In particular, the components’ STSs can be encoded into a planning domain $D$ whose dynamics represent the possible behaviors of services, while the requirement $\rho$ is converted into a corresponding planning goal $g$. That is, the (STSs of the) components services, considered concurrently, form the overall planning domain under exam, whereas the plan to be obtained, whose execution as an “orchestrator” of the components must realize the composition goal, corresponds to the STS of the orchestration service.

Referring to STSs, the formal statement of the composition problem in this setting stands on three key notions: a notion of parallel product, stating that can be in one of its possible states, and an notion $\Sigma$ final computations and decisions. That is, $\Sigma$ is the set of states and $\Sigma^0$, is defined as follows:

$\Sigma_i||\Sigma_j = \langle S_1 \times S_2, S_1^0 \times S_2^0, I_1 \cup I_2, O_1 \cup O_2, R_1||R_2, F_1||F_2 \rangle$

where:

- $\langle (s_1, s_2), a, (s_1', s_2) \rangle \in (R_1||R_2)$ if $\langle s_1, a, s_1' \rangle \in R_1$;
- $\langle (s_1, s_2), a, (s_1', s_2) \rangle \in (R_1||R_2)$ if $\langle s_2, a, s_2' \rangle \in R_2$;

and $\langle F_1||F_2 \rangle(s_1, s_2) = F_1(s_1) \cup F_2(s_2)$.

Finally, the interactions of the STS $\Sigma_c$ of the composed service with the domain $\Sigma$, that it controls by exchanging inputs and outputs, are modelled by the notion of a controlled system.

Definition 3 (Controlled System)

Let $\Sigma = \langle S, S^0, I, O, R, S^F, F \rangle$ and $\Sigma_c = \langle S_c, S^0_c, I_c, O_c, R_c, S^F_c, F_c \rangle$ be two state transition systems. $\Sigma_c \triangleright \Sigma$, describing the behaviors of system $\Sigma$ when controlled by $\Sigma_c$, is defined as follows:

$\Sigma_c \triangleright \Sigma = \langle S_c \times S, S^0_c \times S^0, I_c \cup I, O_c \cup O, S^F_c \times S^F, F_c \cup F \rangle$

where:

$\langle (s_c, s), (b_c \land b), a, (s'_c', s') \rangle \in (R_c \land R)$, if

$\langle s_c, b_c, a, s'_c \rangle \in R_c$ and $\langle s, b, a, s' \rangle \in R$.

In this setting, service composition is stated as: “given the services $W_1, \ldots, W_n$, represented by the STSs $\Sigma W_1, \ldots, \Sigma W_n$, and a composition goal $\rho$, identify a composed service $\Sigma_c$ such that $\Sigma_c \triangleright (\Sigma W_1 \ldots \Sigma W_n) = \rho$.” As shown in (Pistore, Traverso, and Bertoli 2005), this can be solved by planning for a domain $D$ which suitably represents $\Sigma = \Sigma W_1 \ldots \Sigma W_n$ and for a suitable interpretation of $\rho$ as a planning goal. Specifically, to make use of effective planning techniques for fully observable domains, the partial observability of services, modeled by the $\tau$ action, is compiled away from $\Sigma$, building a so-called belief-level system $\Sigma_B$, an STS whose states correspond to beliefs of $\Sigma$ - that is, to sets of states of $\Sigma$ which are equally plausible for an external observer sensing the STS’s inputs and outputs. $\Sigma_B$ can then be encoded into a planning domain $D$, and, under mild assumptions, one can prove that the composition problem is solved by identifying (the STS corresponding to) a plan $\pi$ that satisfies the composition goal $\rho$ for the planning domain $D$.

Of course, this approach can be instantiated in several ways, using different languages and semantics for the composition requirements. In particular, since services expose conditional and non-controllable behaviors, it is often necessary to distinguish alternate admissible situations which are ranked differently under the perspective of the composite service. For this reason, preferences play an important role and have been accounted for in approaches such as (Pistore, Traverso, and Bertoli 2005; Marconi, Pistore, and Traverso 2006). Furthermore, data exchanges play a crucial role in service coordination, and cannot be captured by speaking exclusively of state configurations. For this reason, (Marconi, Pistore, and Traverso 2006) also embeds a modular language to express data-flow orchestration constraints.

In spite of the technical complexity of the solutions mentioned above, none of them captures the crucial fact that services may require a continuous coordination, since they may
represent the everlasting and uncontrollable evolution of actual real-world agents. In fact, none of the approaches can deal with our reference scenario of Sec. 2. This calls for devising a different requirement language and planning algorithm, as shown in the next sections.

4. The requirement language

We now present a simple requirement language that allows fulfilling our desiderata to model in an easy-to-specify, compositional way coordination requirements which express (i) in which situations we intend to get, and continuously maintain, the orchestrated components, possibly ordered according certain preferences; (ii) linking the dynamics of different services; and (iii) reaction rules, which define how the composed service shall react, in different situations, to actions asynchronously performed by some component service. We remark that not only (ii) and (iii) are not considered in solutions such as (Pistore, Traverso, and Bertoli 2005; Marconi, Pistore, and Traverso 2006), but also for (i) we give a different interpretation of “stable” configurations: here, the idea is that a configuration can be always affected even once reached, and it is the task of the orchestrator to take recovery actions to bring the domain back to admissible configurations, while obeying at all times the constraints expressed in (ii) and (iii).

Definition 4 (Composition Requirement) A composition requirement is defined with the following generic constraint template

\[
\text{clause} \Rightarrow (\text{clause}_1 \triangleright \ldots \triangleright \text{clause}_n),
\]

where \(\text{clause} \equiv \top \mid \text{S}.s \mid \text{S} \uparrow a \mid \text{cl}_1 \lor \text{cl}_2 \mid \text{cl}_1 \land \text{cl}_2.\)

Here \(\text{cl}_1\) and \(\text{cl}_2\) are clauses, \(\text{S}.s\) is used to define the fact that the service \(\text{S}\) is in the state \(s\), and \(\text{S} \uparrow a\) defines that the service \(\text{S}\) has performed the action \(a\).

The left side of the constraint defines the “premise” of the requirement. When different from \(\top\), it defines a “reaction rule”: whenever the corresponding situation or actions take place, the composite service should try to “recover” from it by achieving what is defined by the right side. Otherwise, the requirement expresses the need to unconditionally reach what is defined by the right side. In both cases, the right side of the constraint defines the expected results. Each of them logically groups simpler results, which may either require to reach a configuration, or require that an action takes place. These results are ranked according to the order of preference denoted by the \(\triangleright\) symbol, from the most preferred to the least preferred. The following example clarifies the usage of both unconditional and reaction-rule requirements.

Example 1 The composition requirements identified in Sec. 2 may be represented as follows:

1. \(\top \Rightarrow (\text{cl}_1.\text{CR} \land \text{FB}.\text{CR} \land \text{HR}.\text{CR} \land \text{FC}.\text{ST} \land \text{FSN}.\text{ST}) \triangleright ((\text{cl}_1.\text{NAV} \land \text{FB}.\text{NAV} \land \text{HR}.\text{ST} \land \text{FC}.\text{ST} \land \text{FSN}.\text{ST}) \lor (\text{cl}_1.\text{NAV} \land \text{FB}.\text{ST} \land \text{HR}.\text{NAV} \land \text{FC}.\text{ST} \land \text{FSN}.\text{ST}) \lor (\text{cl}_1.\text{REF} \land \text{FB}.\text{REF} \land \text{HR}.\text{REF} \land \text{FC}.\text{ST} \land \text{FSN}.\text{ST}) \lor (\text{cl}_1.\text{DEL} \land \text{FB}.\text{CR} \land \text{HR}.\text{DEL} \land \text{FC}.\text{DEL} \land \text{FSN}.\text{DEL}))\)

That is, the composition should try to get and keep the customer, flight and hotel services in their “created” state \(\text{CR}(\text{and the deletion services not started}); should this not be possible, then either they all should be deleted (if they were started), or they should be in a consistent situation where unavailabilities have been properly notified to the user.

2. \(\text{FSN} \uparrow \text{f} \text{Delayed} \Rightarrow (\text{cl}_1 \land \text{changed} \land \text{HR} \uparrow \text{change})\)

Here as a reaction to the notification of flight delay, we require the corresponding request of modification of the hotel and of the user offer.

3. \(\text{FSN} \uparrow \text{f} \text{Cancelled} \Rightarrow (\text{HR} \downarrow \text{delete} \land \text{cl}_1 \uparrow \text{deleted}).\)

In other words, if the flight is notified to be cancelled, the composed service should cancel also the hotel reservation and the offer.

Notice that an orchestration satisfying these requirements must react as soon as the conditions in the right-hand side of (1) are not satisfied, as well as to the notifications in the left-hand sides of (2) and (3).

5. Encoding as planning problem

Given a set of requirements specified as above, and a set of component services, we aim to build a composed service that aims to satisfy all of them simultaneously, according to their above-mentioned “unconditional” and “reactive” semantics, and following the preference orders specified for right side clauses.

Our first step for this is to define, for each requirement \(\text{cl}_i\), a corresponding \(\text{STS} \Sigma_{\text{C}_i}\), and an associated propositional formula \(\rho_{\text{C}_i}\) that holds when the requirement is satisfied. Then, we embed such \(\text{STSs}\) as part of the planning domain, and we consider the formulae \(\rho_{\text{C}_i}\) to build the overall planning goal. Specifically, given a clause \(\text{cl}\), we define a corresponding \(\text{STS}\) that contains a single output action \(a_{\text{cl}}\) representing the completion of the clause. The diagrams corresponding to the different clauses, to their combinations, and to the representing diagram itself are represented in Fig. 3. Intuitively, they have the following meaning:

- The \(\text{STS}\) for the \(\top\) clause (Fig. 3(a)) is completed immediately.
- The \(\text{STS}\) for \(\text{S}.s\) (Fig. 3(b)) is blocked until the service is not in the required state: the transition is guarded with the corresponding proposition.
- The \(\text{STS}\) for \(\text{S} \uparrow a\) (Fig. 3(c)) waits for the corresponding service action. When it happens, a completion is reported.
- The \(\text{STS}\) for \(\text{cl}_1 \lor \text{cl}_2\) (Fig. 3(d)) waits for any of the sub-clauses to complete, while the \(\text{STSs}\) of the \(\text{cl}_1 \land \text{cl}_2\) (Fig. 3(e)) waits for both of them to be completed.

The \(\text{STSs}\) that represents the evolution of a composition requirement is represented in Fig. 3(f). The \(\text{STS}\) is initially in an accepting state \((s_0)\). If the premise takes place \((a_{\text{cl}}\) is reported), then it moves to a non-accepting state, from which it may be satisfied by completing one of the clauses \(a_{cl_1}, \ldots, a_{cl_n}\) (moving to states \(s_1, \ldots, s_n\) respectively). The corresponding goal with preferences will have the following form:

\[
\rho_c = (s_0, s_1, \ldots, s_n).
\]

That is, we require that whenever the premise takes place,
the composition tries to move the STS to one of the accepting states, respecting the ordering of preferences.

To model composition in the presence of multiple constraints, we will combine requirements of this form by a straightforward “flattening” operator $\oplus$ such that $\rho_1 \oplus \rho_2$ is a list of preference-ordered propositional formulae, each corresponding to a set of configurations to be reached and then continuously maintained. A solution to such a problem must consist of a plan whose possible executions on the domain can be either finite and terminating in $\rho$, or infinite and traversing $\rho$ infinitely often. Furthermore, it must enforce optimality, identifying at all times the action that directs towards the best currently achievable goal. To find such a solution, our planning algorithm performs two key steps:

1. first, we restrict the domain $D$ to those configurations that are ‘recoverable’, that is for which it is possible, by executing a suitable course of action, to come back to $\rho$ in the face of the domain’s autonomous evolution caused by uncontrollable events. We call such restricted domain $D_R$. 2. then, for each state in $D_R$, we identify which is the best action that must be performed to achieve the goals $\rho_1, \ldots, \rho_n$ according to their preference order.

Step (1) is realized by the **computeRecoverable** algorithm. We now describe a planning algorithm that, given a fully observable planning domain including exogenous events, finds a plan for a composition requirement $\rho = (\rho_1, \ldots, \rho_m)$ consisting of a list of preference-ordered propositional formulae, each corresponding to a set of configurations to be reached and then continuously maintained. A solution to such a problem must consist of a plan whose possible executions on the domain can be either finite and terminating in $\rho$, or infinite and traversing $\rho$ infinitely often. Furthermore, it must enforce optimality, identifying at all times the action that directs towards the best currently achievable goal. To find such a solution, our planning algorithm performs two key steps:

1. first, we restrict the domain $D$ to those configurations that are ‘recoverable’, that is for which it is possible, by executing a suitable course of action, to come back to $\rho$ in the face of the domain’s autonomous evolution caused by uncontrollable events. We call such restricted domain $D_R$. 2. then, for each state in $D_R$, we identify which is the best action that must be performed to achieve the goals $\rho_1, \ldots, \rho_n$ according to their preference order.

Step (1) is realized by the **computeRecoverable** algorithm. We now describe a planning algorithm that, given a fully observable planning domain including exogenous events, finds a plan for a composition requirement $\rho = (\rho_1, \ldots, \rho_m)$ consisting of a list of preference-ordered propositional formulae, each corresponding to a set of configurations to be reached and then continuously maintained. A solution to such a problem must consist of a plan whose possible executions on the domain can be either finite and terminating in $\rho$, or infinite and traversing $\rho$ infinitely often. Furthermore, it must enforce optimality, identifying at all times the action that directs towards the best currently achievable goal. To find such a solution, our planning algorithm performs two key steps:

1. first, we restrict the domain $D$ to those configurations that are ‘recoverable’, that is for which it is possible, by executing a suitable course of action, to come back to $\rho$ in the face of the domain’s autonomous evolution caused by uncontrollable events. We call such restricted domain $D_R$. 2. then, for each state in $D_R$, we identify which is the best action that must be performed to achieve the goals $\rho_1, \ldots, \rho_n$ according to their preference order.

Step (1) is realized by the **computeRecoverable** algorithm. We now describe a planning algorithm that, given a fully observable planning domain including exogenous events, finds a plan for a composition requirement $\rho = (\rho_1, \ldots, \rho_m)$ consisting of a list of preference-ordered propositional formulae, each corresponding to a set of configurations to be reached and then continuously maintained. A solution to such a problem must consist of a plan whose possible executions on the domain can be either finite and terminating in $\rho$, or infinite and traversing $\rho$ infinitely often. Furthermore, it must enforce optimality, identifying at all times the action that directs towards the best currently achievable goal. To find such a solution, our planning algorithm performs two key steps:

1. first, we restrict the domain $D$ to those configurations that are ‘recoverable’, that is for which it is possible, by executing a suitable course of action, to come back to $\rho$ in the face of the domain’s autonomous evolution caused by uncontrollable events. We call such restricted domain $D_R$. 2. then, for each state in $D_R$, we identify which is the best action that must be performed to achieve the goals $\rho_1, \ldots, \rho_n$ according to their preference order.

Step (1) is realized by the **computeRecoverable** algorithm. We now describe a planning algorithm that, given a fully observable planning domain including exogenous events, finds a plan for a composition requirement $\rho = (\rho_1, \ldots, \rho_m)$ consisting of a list of preference-ordered propositional formulae, each corresponding to a set of configurations to be reached and then continuously maintained. A solution to such a problem must consist of a plan whose possible executions on the domain can be either finite and terminating in $\rho$, or infinite and traversing $\rho$ infinitely often. Furthermore, it must enforce optimality, identifying at all times the action that directs towards the best currently achievable goal. To find such a solution, our planning algorithm performs two key steps:

1. first, we restrict the domain $D$ to those configurations that are ‘recoverable’, that is for which it is possible, by executing a suitable course of action, to come back to $\rho$ in the face of the domain’s autonomous evolution caused by uncontrollable events. We call such restricted domain $D_R$. 2. then, for each state in $D_R$, we identify which is the best action that must be performed to achieve the goals $\rho_1, \ldots, \rho_n$ according to their preference order.

Step (1) is realized by the **computeRecoverable** algorithm. We now describe a planning algorithm that, given a fully observable planning domain including exogenous events, finds a plan for a composition requirement $\rho = (\rho_1, \ldots, \rho_m)$ consisting of a list of preference-ordered propositional formulae, each corresponding to a set of configurations to be reached and then continuously maintained. A solution to such a problem must consist of a plan whose possible executions on the domain can be either finite and terminating in $\rho$, or infinite and traversing $\rho$ infinitely often. Furthermore, it must enforce optimality, identifying at all times the action that directs towards the best currently achievable goal. To find such a solution, our planning algorithm performs two key steps:

1. first, we restrict the domain $D$ to those configurations that are ‘recoverable’, that is for which it is possible, by executing a suitable course of action, to come back to $\rho$ in the face of the domain’s autonomous evolution caused by uncontrollable events. We call such restricted domain $D_R$. 2. then, for each state in $D_R$, we identify which is the best action that must be performed to achieve the goals $\rho_1, \ldots, \rho_n$ according to their preference order.

Step (1) is realized by the **computeRecoverable** algorithm. We now describe a planning algorithm that, given a fully observable planning domain including exogenous events, finds a plan for a composition requirement $\rho = (\rho_1, \ldots, \rho_m)$ consisting of a list of preference-ordered propositional formulae, each corresponding to a set of configurations to be reached and then continuously maintained. A solution to such a problem must consist of a plan whose possible executions on the domain can be either finite and terminating in $\rho$, or infinite and traversing $\rho$ infinitely often. Furthermore, it must enforce optimality, identifying at all times the action that directs towards the best currently achievable goal. To find such a solution, our planning algorithm performs two key steps:

1. first, we restrict the domain $D$ to those configurations that are ‘recoverable’, that is for which it is possible, by executing a suitable course of action, to come back to $\rho$ in the face of the domain’s autonomous evolution caused by uncontrollable events. We call such restricted domain $D_R$. 2. then, for each state in $D_R$, we identify which is the best action that must be performed to achieve the goals $\rho_1, \ldots, \rho_n$ according to their preference order.

Step (1) is realized by the **computeRecoverable** algorithm. We now describe a planning algorithm that, given a fully observable planning domain including exogenous events, finds a plan for a composition requirement $\rho = (\rho_1, \ldots, \rho_m)$ consisting of a list of preference-ordered propositional formulae, each corresponding to a set of configurations to be reached and then continuously maintained. A solution to such a problem must consist of a plan whose possible executions on the domain can be either finite and terminating in $\rho$, or infinite and traversing $\rho$ infinitely often. Furthermore, it must enforce optimality, identifying at all times the action that directs towards the best currently achievable goal. To find such a solution, our planning algorithm performs two key steps:

1. first, we restrict the domain $D$ to those configurations that are ‘recoverable’, that is for which it is possible, by executing a suitable course of action, to come back to $\rho$ in the face of the domain’s autonomous evolution caused by uncontrollable events. We call such restricted domain $D_R$. 2. then, for each state in $D_R$, we identify which is the best action that must be performed to achieve the goals $\rho_1, \ldots, \rho_n$ according to their preference order.
Figure 4: The domain restriction algorithm.

to finally merge state-action tables into an overall plan, by
layering them according to the respective preferences.

In particular, as also discussed in (Shaparau, Pistore, and
Traverso 2006), to correctly consider preferences so that the
resulting overall plan is preference-optimal, one has to build
state-action tables starting from the less preferred goals and
going to the most preferred one: only in this way, the state-
action table built for \( \rho_i \) can be used as a “recovery” basis for
the one referring to the more preferable \( \rho_{i-1} \).

The core of this algorithm is the computeSATables routine
in Fig. 5. For each goal \( \rho_i \), starting from the least pre-
ferred, the algorithm first computes all the state-action pairs
for which a “strong” solution exists to get to \( \rho_i \), and then
it iteratively enriches such table by considering the already
found state-action tables for \( \rho_j \) with \( j = i+1, \ldots, n \). In particular, the table is enriched with state-action pairs that
guarantee both (in a strong way) to satisfy \( \rho_j \), and, in a weak
way, \( \rho_i \). In case new state-action pairs are added, a new analysis of “strong solutions” is in order; otherwise, the “enriching”
steps to consider lower-preference goals by incrementing \( j \).

We remark that, while the structure of the algorithm is inspired by the work in (Shaparau, Pistore, and Traverso
2006), it bears significant technical differences. First, since
we consider maintainability goals, we need to build looping
plans. This leads us to adopt a StgLFP routine with a struc-
ture similar to that of PruneUnconnected, that enforces a
full fixpoint search and returns a state-action table where
goals may appear as intermediate states. This also explains
why, here, we do not add goals during weakening. Second,
since our planning domain includes exogenous events, the

Figure 5: The algorithms for identifying and merging state-
action tables.

pre-image primitives (including WkPreImg, which is analo-
gous to StgLFP, but acts existentially on outcomes and
events) are significantly different from the ones used in (Sha-
parau, Pistore, and Traverso 2006).

Once the state-action tables for the goals are ready, they
are easily merged, following preferences, by the mergeTa-
bles routine, also reported in Fig. 5.

Finally, the key steps (1) and (2) are glued together by
the main planning routine in Fig. 6, which essentially in-
vokes them in turn. Notice that, as demonstrated by the
correctness and completeness statements below, for the sec-
ond step it is enough to simply restrict the goal states to the

Figure 6: The main routine.
recoverable ones, since then the search never reaches un-
recoverable states.

**Theorem 1** Function Planning(I,gList) always terminates
on a planning domain D with exogenous events. If Planning(I,gList) \( \pi \neq \bot \), then \( \pi \) is a solution to the problem of achieving and maintaining in a strong way the goal \( \bigcup \text{gList}[i] \). If Planning(I,gList) = \( \bot \), then no solution to that problem exists.

Proof sketch: Concerning termination, it is easy to see that computeRecoverable implements a terminating greatest fix-point computation, since the function pruneUnconnected incrementally computes a set of states but never exceeds the set of states it takes as input. To see that computeSATables terminates, it is immediate to see that its outer loop performs a finite number of iterations. But also its inner while loop iterates a finite number of times, since the size of SA increases monotonically and is bounded.

To see that the plan found by the algorithm is a solution, notice first that it contains all the initial states. According to the definition of computeSATables, the table will contain states from which all executions are guaranteed to lead to some goal states. According to the function Planning, the goal states are restricted to recoverable goal states. According to the function computeRecoverable, actions from the recoverable states also always lead to goal states, and thus are also contained in the resulting table. Therefore, the proposed plan is a solution.

To see that the algorithm always returns a solution if any exists, we observe the following properties. First, the algorithm returns a plan containing all the states for which there exists a strong plan to reach any goal state. Second, the plan includes all the recoverable goal states. According to the definition, the solution to the problem always leads to one of the goal states and therefore will be discovered by the algorithm.

**Theorem 2** Let \( \prec_s \) be a partial ordering relation between plans as defined in (Shaparau, Pistore, and Traverso 2006). If Planning(I,gList) = \( \pi \neq \bot \), then \( \pi \) is an optimal solution for the problem of achieving and maintaining in a strong way the goals \( \text{gList}[1], \ldots, \text{gList}[n] \) according to the preferences.

Proof sketch: Say \( 0 \leq i \leq j \). If, according to the definition of \( \prec_s \), an optimal plan provides a weak solution for gList[i] and a strong solution for gList[j] from state \( s \), then according to the computeSATables function, \( s \in \text{StatesOf}(\text{gList}[i]) \) and \( s \notin \text{StatesOf}(\text{gList}[k]), 0 \leq k < i \). Moreover, all the states reachable from \( s \) are included into some \( \text{StatesOf}(\text{gList}[l]), i < l \leq j \). Then, from the mergeTables procedure it is clear that the extracted plan \( \pi \) is a strong solution for \( \text{glist}[i] \) and a weak solution for \( \text{glist}[j] \) from \( s \), i.e., it is optimal.

### 7. Experiments

To empirically evaluate our approach, we implemented the algorithms of Sec. 6 based on symbolic BDD libraries to effectively represent and manipulate planning domains. We also realized the conversion of requirements as STSs explained in Sec. 5, and integrated these contributions within the architecture of Fig. 2. We tested our algorithm and approach first on the reference scenario, and then on a set of scenarios designed to evaluate the scalability of the approach over different dimensions. All tests have been run on a 2.6GHz, 4Gb Dual Core machine running Linux.

Our test on the reference scenario uses the requirements given in Example 1, and also includes data flow requirements to appropriately route data, defined with the data-net approach of (Marconi, Pistore, and Traverso 2006). Our prototype generated a plan corresponding to an orchestration service that fully realizes the above requirements. Specifically, the composed process tries to perform the flight booking and hotel reservation, correctly taking into account possible non-deterministic outcomes of the component services, and creates a travel offer upon successful reservations. Then the process continuously handles the delays or cancellations from the flight status notification service. In the first case, it tries to modify the hotel reservation: if the hotel agrees and the user accepts the new offer, the process is ready to handle new modifications. Otherwise, the process deletes the reservations and terminates. In the second case, the process cancels the hotel reservation and informs the user. This complex plan includes several branching points and two different loops, and realizing such an orchestration would be far from trivial even for a skilled analyst. The composed service was generated in about 35 seconds; given the complexity of the task, this experiment is a first important witness of the practical applicability of our approach.

We then proceeded to analyze three sets of scalable sce-
narios. In the first, we evaluate the scalability of coordinating an increasing number of services. The scenario involves an ‘inviter’ service $I$ and $n$ ‘guests’ services $G_i$. The inviter sends a invitation, and then keeps listening for responses; vice versa, a guest is activated by an invitation, and then can continuously send updates on his decision. Our goal is to propagate $I$’s invitation to all guests, and then to keep $I$ continuously updated on the responses of each guest. Our results are shown in Fig. 7, left. The performance scales up with a polynomial behavior, and is capable to tackle a fairly large number of services in a reasonable time. In the second scenario, we evaluate the scalability w.r.t. the number of coordination constraints. For this purpose, we take a master and a slave service, the master continuously producing a command out of a set of $n$ possible ones, and the slave awaiting for commands to be executed. To keep the two services continuously aligned, we use a set of $n$ coordination constraints. Our results are shown in Fig. 7, center. Also in this case, performance is very good, and scales up polynomially in the number of constraints. Essentially, these two scalable tests consistently show that a linear growth in the size of the planning domain maps into a polynomial behavior.

Finally, we evaluate the scalability w.r.t. the number of preferences in the goal. We do so by running the example on a set of services simulating a robot scenario where each ‘robot’ service can be commanded to guard a door, but may then break down or autonomously decide it needs recharging, becoming (temporarily or finally) unavailable. Considering 20 robots and 20 doors, we consider goals that use $n = 1, \ldots, 20$ preferences to express we intend to keep $n$ doors guarded, but whenever this cannot be granted, as many as possible. As we see from Fig. 7, right, the performance is essentially linear in the number of preferences. This says that, thanks to the implementation being based on BDD representations of state sets, the search for a preference-specific subgoal $\rho_i$ does not strongly depend on the size of $\rho_i$; then, the behavior maps the linear number of preference-specific sub-searches in the algorithm.

8. Related Work and Conclusions

Our work is mainly related to two areas: SOC and planning. In SOC, service composition has received considerable attention, and attempts at composing stateful services have been proposed in (Berardi et al. 2005; Pistore, Traverso, and Bertoli 2005; Marconi, Pistore, and Traverso 2006; Hull 2005). None of these, however, are capable to capture the coordination constraints we describe here, nor to generate executable orchestrations that satisfy such requirements. We regard ours as an important step toward the practical applicability of planning-based approaches to service composition. Also clearly related is the companion paper (Bertoli et al. 2009), whose focus is however mainly methodological. This work clearly differs in focus and technical content, detailing the core algorithms, showing them correct, and evaluating the scalability of the approach.

In planning, several works have considered expressive goals that can capture maintainability, e.g. using temporal logics ((Bacchus and Kabanza 1998; Kvarnstrm and Doherty 2001; Perreira and Barros 2008)), while other works have considered user preferences, see e.g. (Brafman and Chernyavsky 2005; Shaparau, Pistore, and Traverso 2008). As well, there is increasing interest in planning formalisms that consider the presence of exogenous events (Andre A. Cire and Adi Botea 2008), often recurring to modelings that embed them as nondeterministic outcomes. However, to the best of our knowledge, no approach has been presented so far that integrates these three aspects together. Our approach is the first achieving this, and empirical tests witness its effectiveness and scalability.

In the future, we plan to thoroughly validate our approach on complex scenarios, and to consider the adoption and integration of HTN techniques within our requirement language, as this can significantly strengthen the practical applicability of our approach by allowing end-users to easily specify complex composition strategies.

References