International Journal on Digital Libraries (2006) 6(3): 235–246
DOI 10.1007/s00799-006-0002-5

REGULAR PAPER

Alexander Lazovik · Marco Aiello · Mike Papazoglou

Planning and monitoring the execution of web service requests

Published online: 24 March 2006

Abstract Interaction with web services enabled marketplaces would be greatly facilitated if users were given a high level service request language to express their goals in complex business domains. This can be achieved by using a planning framework which monitors the execution of planned goals against predefined standard business processes and interacts with the user to achieve goal satisfaction.

We present a planning architecture that accepts high level requests, expressed in a service request language known as XSRL. The planning framework is based on the principle of interleaving planning and execution. This is accomplished on the basis of refinement and revision as new service-related information is gathered from service repositories such as UDDI and web services instances, and as execution circumstances necessitate change. The planning system interacts with the user whenever confirmation or verification is needed.

1 Introduction

Service oriented computing (SOC) is rapidly becoming the prominent paradigm for distributed computing and electronic business applications. SOC allows service providers and service application developers to construct value-added services by combining existing services that are resident on the Web. To achieve this, firstly, web services must be described in terms of the standard web service definition language WSDL (http://www.w3.org/TR/wsdl) and subsequently must be inter-linked to express how collections of web services work jointly to realize more complex functionalities typified by business processes. A new web service can be defined in terms of compositions of existing (constituent) services on the basis of the standard Business Process Execution Language for Web Services (BPEL4WS or BPEL for short, http://www-106.ibm.com/developerworks/library/ws-bpel/). BPEL models the actual behavior of a participant in a business interaction as well as the visible message exchange behavior of each of the parties involved in the business protocol. A BPEL process is defined "in the abstract" by referencing and inter-linking portTypes specified in the WSDL definitions of the web services involved in a process. A BPEL process is a reusable definition that can be deployed in different ways and in different scenarios, while maintaining a uniform application-level behavior across all of them. Service compositions in BPEL are described in such a way (e.g., WSDL over UDDI) that allows automated discovery and offers request matching on service descriptions.

In many situations it is desirable to empower a user to gain explicit control over the execution of BPEL expressions and dynamically change the nature of the web service interactions conducted with a particular business partner depending on the state of the process. Consider for example the case of a traveler deciding to change his/her hotel reservation to take advantage of an unexpectedly lowly priced weekend offer. Users may need to change message property values in the midst of a computation, e.g., update their holiday budget based on ticket, hotel prices and availability, evaluate different behavioral alternatives or scenarios during a computation and change their course of action dynamically, or revisit different execution paths based on non-deterministic message property values that result from the invocation of services involved in a process. This implies that BPEL execution must be made adaptable at run-time to meet the changing needs of users and businesses. Obviously, BPEL specifications do not allow for the flexibility required to react swiftly to unforeseen circumstances or opportunities as choices are predefined and statically bound in BPEL programs. To meet such requirements serious re-coding efforts are

A. Lazovik (✉) · M. Aiello
Department of Information and Telecommunication Technologies, University of Trento, Via Sommarive, 14, 38050 Trento, Italy
E-mail: {lazovik, aiellom}@dit.unitn.it

A. Lazovik
ITC-IRST, Via Sommarive, 18, 38050 Trento, Italy

M. Papazoglou
Infolab, Tilburg University, P.O. Box 90153, NL-5000 LE, The Netherlands
E-mail: mikep@uvt.nl

needed every time that there is need for even a slight deviation.

Such advanced functionality can be better supported by a service request language and its appropriate run-time support environment to allow users to express their needs on the basis of the characteristics and functionality of standard business processes whose services are found in UDDI registries. A service request language provides for a formal means of describing desired service attributes and functionality, including temporal and non-temporal constraints between services, service scheduling preferences, alternative options and so on.

Our research work concentrates on developing a service request language for XML-based web services that contains a set of appropriate constructs for expressing requests and constraints over requests as well as scheduling operators. We have named this language XSRL for XML Service Request Language [2, 19]. XSRL expresses a request against standard processes defined in a vertical domain, e.g., e-travel, and returns a set of documents as the result of executing the request, e.g., by sending end-to-end holiday packages (documents). The user requests generate a plan based on a standard business process that invokes a series of web services and interacts with the user to satisfy her/his request.

The remainder of the paper is organized as follows. In Sect. 2 an overview of related work is given. Then, in Sect. 3 an example in the travel domain which runs throughout the paper is presented. The architecture of the proposed framework is illustrated in Sect. 4, in particular, we define the planning domain in Sect. 4.1, we present an example of domain in Sect. (4.2), we introduce an enhanced syntax and semantics for XSRL in Sect. (4.3) and provide algorithms for satisfying XSRL requests in Sect. (4.4). In Sect. 5 we exemplify the functionality of the architecture on the running example. Sect. 6 presents conclusions, while proofs of algorithm correctness are sketched in the Appendix.

2 Related work

In service-oriented computing, several initiatives have been proposed to enable integration between heterogeneous systems. In particular, the web service protocol stack [10] includes the Web Service Description Language [27], the Simple Object Access Protocol [1], Universal Description, Discovery and Integration [25] that allows platform- and language-independent service publishing, discovery and invocation. Business Process Execution Language for Web Services [6] is focused on representation of web service executions, where composition is known in advance. Choreography Description Language defines, from a global viewpoint, observable inter-enterprise behavior, where ordered message exchanges result in accomplishing a common business goal [13].

Despite all the efforts, service composition is still an extremely complicated task. Complexity comes from different places. First, the number of services and partners available on the Web is high and steadily increasing, making it difficult to choose the right service to find and invoke. Second, in a true service-oriented architecture, there is no single owner of the business process, that is, every change to a process has to be approved by all involved parties. Therefore, having consistent and stable business processes that satisfy business goals of all participants and ensure correctness at runtime is hard to achieve. Third, the execution of a business process depends on the behavior of involved partners that is not known when the process is designed, thus, designer of the process has to take into account all possible service behaviors.

That is why having a mechanism for automatic or semi-automatic service composition is crucial for successful enterprise application integration. Several approaches have been proposed to achieve these issues. Service composition is somewhat similar to composition of workflows [26] and techniques developed for workflows can be reused for composition of services. For example, in [7] it is proposed as a configurable approach to service composition. However, workflow composition frameworks do not take into account issues specific to service-oriented computing: dynamic binding, highly heterogeneous environments, absence of single ownership and control over process execution. In the context of Semantic Web Services there have been proposed several approaches for service compositions, e.g., knowledge-based semantic web service composition [9], service discovery and composition based on semantic matching [18], semi-automatic composition of web services based on semantic descriptions [23]. All these approaches work under the assumption of having available rich semantic service description and run-time information. In contrast to this, in a pure service-oriented environment on the one hand, there is little semantic description and, on the other hand, one deals with incomplete knowledge about service behavior and required information is gathered and analyzed during execution.

Artificial Intelligence (AI) techniques can provide a solution to the problem of service composition. In particular, there have been several proposals using AI planning. In [24], a review of web service composition techniques is presented and it is argued that planning techniques can be of help in tackling the problem of automatic web service composition. Various authors have emphasized the importance of planning for web services [12, 16, 17, 24]. In particular, Knoblock et al. [12] use a form of template planning based on hierarchical task networks and constraint satisfaction. The authors focus on information gathering and integration rather than on service composition. In [16], regression planning for composition is used taking into account incomplete knowledge about planning domain. In [17], the Golog planner is used to automatically compose semantically described services. Knowledge-sensing actions are used to gather information at runtime. The two latter approaches describe the

goal as a set of desired states ignoring conditions on how these states are reached. A finite-state machine framework for automatic composition was introduced in [3]. Our work is primarily based on planning as model checking under non-determinism for extended goals [11, 20]. Extensions toward interleaving planning and execution in the above context are reported in [5]. The latter work emphasizes on state explosion problems rather than information gathering, furthermore, it does not handle numeric values. Temporally extended goals, i.e., goals expressing not only desired states to be achieved but also conditions on how these are to be reached, are an expressive way of defining complex business goals [14, 19, 21]. Our approach differs from these recently proposed planning approaches for web services in that it is based on non-deterministic planning whereas most of the previously cited approaches focus on gathering information, on applying deterministic planning techniques, on using precompiled plans or on assuming rich semantic annotations of services.

3 Running example: organizing a trip

Suppose a user is planning a one night trip to Paris and is interested in a number of possibilities in connection with this trip. These include making a hotel reservation in Paris, avoiding to travel by train, if possible, and spending an overall amount not greater than 300 euros for the whole package. Further, the user prefers to spend less than 100 euros for a hotel room but, if this is not possible, he may be willing to spend up to 200 euros for that room. The user wants to pay under the condition that he receives a confirmation for the entire package. Of course, the user would also need to specify dates for his trip and accommodation in Paris. This will not be considered in this example as it provides no additional details about the concepts behind the presented system. The wishes of the user have not much meaning unless they are matched against a standard business process in the e-travel domain. What the user requires is a business process description that prescribes how to interact with an e-travel marketplace infrastructure such as travel agents, hotel services and so on.

Nowadays, standard business descriptions and terminology descriptions are given in XML schemas, e.g., for the automotive industry, travel industry, chemical industry and so on, see http://xml.coverpages.org/xmlApplications.html. We expect that in the near future abstract definitions of such business process will be given in BPEL or similar service orchestration languages.

A snippet of a simple hypothetical business process for reserving a trip in the e-travel domain is given in Fig. 1. From a planning perspective process described in Fig. 1 comprises planning domains and is modeled as a state transition diagram, that is, every node represents a state in which the process can be, while labeled arcs indicate how the process changes state. Actors involved in the process are shown at the top of the diagram. The actors include the user, a travel

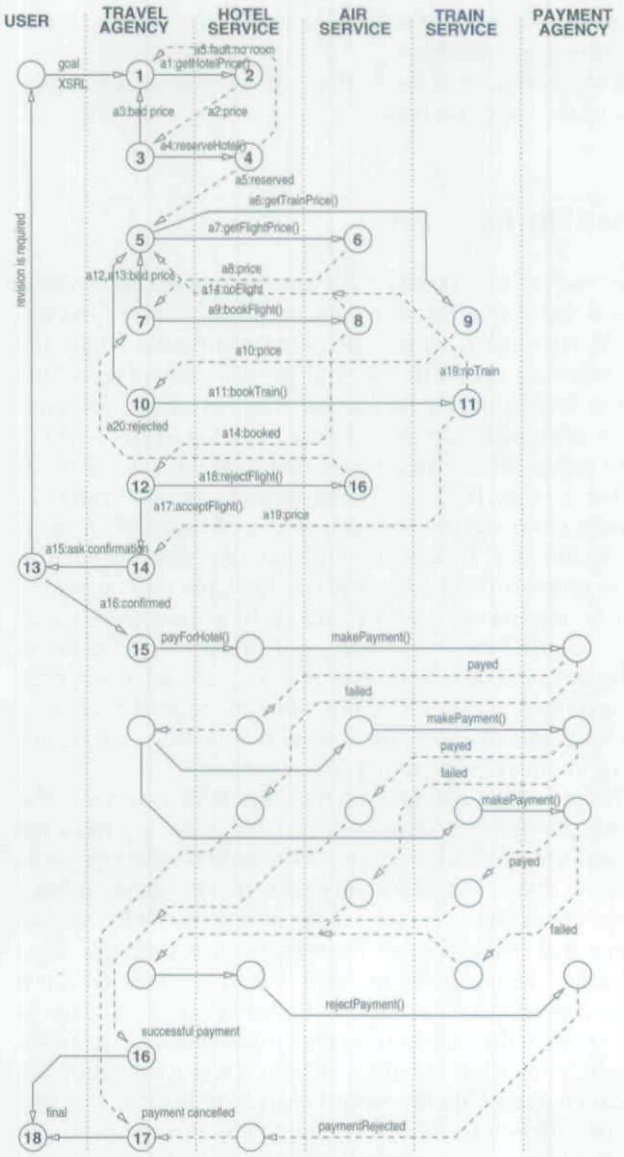

Fig. 1 An e-travel business domain

agency, a hotel service, an air service, a train service and a payment service.

The process is initiated by the user contacting a travel agency, hence, (1) is the initial state. The state is changed to (2) by requesting a quote from an hotel (action a_1). The dashed arcs represent web service responses, in particular arc a_2 brings the system in the state (3). The execution continues along these lines by traversing the paths in the state transition diagram until state (14) is reached. In this state a confirmation of a hotel and of a flight or train is given by the travel agency and the user is prompted for acceptance of the travel package (13).

The state transition diagram is non-deterministic. This is illustrated, for instance, in state (4). In this state the user has accepted the hotel room price but is faced with two possible outcomes, one that a room is not available (where the

system transits back to state (1)) and the other where a room reservation is made (state (5)).

The lower part of the business process models the payment of the travel package.

4 The XSRL framework

Two types of uncertainty for transitions between business process states may arise: nondeterministic failures and unknown outcomes from actions. Nondeterministic failure occurs when an action has several possible outcomes which are not known before invocation. The list of possible outcomes is known a priori and thus modeled in the domain. There exists several techniques that deal with this kind of nondeterminism [8, 20, 22]. The second type of uncertainty requires additional processing before application of the planning techniques. Unknown outcomes of action invocations can be properly handled only at run-time, therefore planning must be interleaved with execution. In a framework based on the interleaving of planning and execution, information on the outcome of action invocation is gathered at run-time and used to replan consistently with the original goal. This idea leads to a planning framework that is based on the notion of interleaving planning and execution.

We propose a planning architecture which works in the following way. The framework receives a request from the user and tries to fulfill it against a standard business process, assuming that it is syntactically correct. The standard business process can be specified in the abstract in BPEL and we assume that is represented graphically by a state transition diagram as the one given in Fig. 1. The framework returns a failure if the request cannot be satisfied in the given business process under the current run-time circumstances, e.g., ticket dates or hotel prices are not available. During execution the system interacts with the service registry to find suitable service providers, in a web service enabled marketplace, and with the user to ask confirmation or request additional information, if necessary.

The planning framework, shown in Fig. 2, comprises four interacting components: monitor, planner, executor, and run-time support environment. Figure 2 illustrates the user issuing a request to the system expressed against a business process (domain). The *monitor* manages the overall process of the interleaved planning and execution. First, it requests the *planner* to construct a plan. Subsequently, the planner either produces a plan or returns a failure (if the request is not satisfiable in the given domain). The *executor* processes the plan provided by the planner by invoking the corresponding web services. It is also responsible for finding a set of web service *providers* for a particular service in the *UDDI* registry. The executor may contact the user for confirmation if user interaction is specified in the business process. The executor does not always execute an entire plan. It rather executes it in steps. It may gather new information, e.g., hotel rates, from the environment (UDDI) and inform the monitor, which in turn may request a new plan to be generated

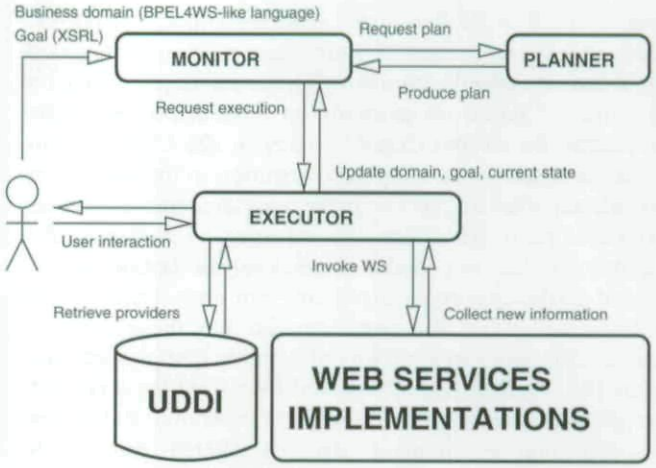

Fig. 2 High-level XSRL architecture

in the light of the information obtained. The executor updates the monitor regarding the status of the execution when re-planning is potentially needed or when it terminates the execution of a plan.

4.1 Planning domain

To perform automatic planning and execution, it is necessary to formally define the domain under which the system acts. Although such a formalization can potentially be extracted from a BPEL definition, BPEL cannot be used directly as, among other things, it lacks formal semantics. Thus, we use a formal extension of BPEL based on a state-transition system enriched with web service domain operators and constructs. One may think of extrapolating a state representation from a BPEL specification.

State-transition systems are the basis of most AI planning systems and form the core of our formalization. In particular, we use a representation able to represent nondeterminism and the potential absence of information of the environment (incomplete information).

Definition 1 (*Planning domain*) A *non-deterministic web services planning domain* is a tuple $\mathbf{D} = \langle \mathcal{S}, Var, Act, R, P, Out, Tr, Role_{Act}, Role_P \rangle$, where:

- $\mathcal{S}$ is the set of *states* in which the business process can be.
- Var is the space of *variables*. It is the Cartesian product of any number of arbitrary domains such as the integers, the real numbers and boolean values. Further, we define the first k elements of the variable space as *knowledge variables*.
- Act is the set of *actions* that can be performed in the transition system.
- R is a set of *service roles* associated with actions.
- P is a set of *service providers* identified by their URI.
- Out is a set of *output types* representing the possible response message types from services.

- $Tr : \mathcal{S} \times Act \times Out \rightarrow \mathcal{S}$ is the *transition function*. The generic element of this relation $\mathrm{Tr}(s_i, a, o_a) = s_j$ represents the transition from state s_i to state s_j by means of action a with output type o_a. An action a is called *deterministic* in a state s if $\exists s' \forall o \in Out \ \mathrm{Tr}(s, a, o) = s'$. It is *non-deterministic* otherwise.
- $Role_{Act} : Act \rightarrow R$ is the *role association function* which relates actions to service roles.
- $Role_P : R \rightarrow 2^P$ is the *role assignment function* that associates every provider to a role in the process.

To assign meaning to the elements of the transition relation we use semantic rules. A semantic rule is an arbitrary function $f : Act \times Var \times Out \rightarrow Var$. Finally, we say that an action $a \in Act$ is *knowledge gathering* (or a *sensing*) action if it affects at least one knowledge variable. Formally, knowledge variables are associated with actions and output types as follows $\forall o \in Out \ (\exists i \leq k : \ f(a, v, o)_i \neq v_i)$ where $f()_i$ represents the restriction of the function i to the ith element and the first k elements of $v \in Var$ are knowledge variables.

We have no restrictions on what this function can be and what is the semantics of the returned values, and it is up to the business process and domain designers to define these rules.

The concept behind the presented formalization of the planning domain is that a given business process is, at any instant, in a state from which a number of actions can be performed to move to a new state. Roles, which represent service interfaces, are associated to actions and implemented by service providers.

4.2 A domain instance

To provide more intuition for the planning domain just presented, we formalize the upper half of the travel business process in Fig. 1 in accordance with Definition 1. In fact, Definition 1 has a number of additional features with respect to the figure. In particular, in the figure the set of variables, the set of service providers, the role assignment function and the semantic rules are not represented.

There are 14 states $\mathcal{S} = \{1, 2, \ldots, 14\}$ in the upper half of the figure. The set of variables is $Var = \{$*hotelReserved*, *hotelPrice*, *location*, *trainBooked*, *trainPrice*, *flightBooked*, *flightPrice*, *confirmed*, *money*$\}$, among which one distinguishes the boolean variables (*hotelReserved*, *trainBooked*, *flightBooked*, *confirmed*) from the real variables (*hotelPrice*, *trainPrice*, *flightPrice*, *money*) and a variable representing location names (*location*). In the set of variables a subset is defined to be of knowledge variables. In the example, we define *hotelPrice*, *trainPrice*, *flightPrice* to be knowledge variables. There are also 19 actions that can be performed in the domain $Act = \{a_1, \ldots, a_{19}\}$.

Four roles are involved in the process $R = \{$*hotel*, *air*, *travel-agency*, *train*$\}$ and the $Role_{Act}$ relation associates to each of them the following actions: *hotel* has $\{a_1, a_2, a_4, a_5\}$, *travel-agency* has $\{a_3, a_{12}, a_{13}, a_{15}, a_{16}, a_{17}\}$, *air* has $\{a_7, a_8, a_9, a_{14}, a_{18}, a_{20}\}$, and *train* has the set of actions $\{a_6, a_{10}, a_{11}, a_{19}\}$ associated. The set of actual service providers for this services obtained by contacting the UDDI could be *Hilton* and *BestWestern* for the *hotel* role, *BritishArways*, *Virgin* for *air* role, *ClubMed* for the travel agency and *TrenItalia* for the train role. The set of output messages is $Out = \{$*normal*, *NoRoomFault*, *NoSeatOnFlight*, *NoSeatOnTrain*$\}$.

Finally, the transition function is given by the set of labeled arcs in the figure, for example, $\mathrm{Tr}(4, a_5, normal) = 5$, $\mathrm{Tr}(4, a_5, NoRoomFault) = 1$ represent that the action a_5 with a *normal* output brings the system into state 5, while the state 1 is reached with the $NoRoomFault$ message. Semantic rules are associated with all actions. The rules for actions Act:

- a_2, *normal*: *hotelPrice* = *result*
- a_3, *normal*: *hotelPrice* = 0
- a_5, *normal*: *money*+ = *hotelPrice*; *hotelReserved* = *true*
- a_5, *NoRoomFault*: *hotelPrice* = 0
- a_8, *normal*: *flightPrice* = *result*
- a_{10}, *normal*: *trainPrice* = *result*
- a_{12}, *normal*: *trainPrice* = 0
- a_{13}, *normal*: *flightPrice* = 0
- a_{14}, *normal*: *money*+ = *flightPrice*; *flightBooked* = *true*
- a_{16}, *normal*: *confirmed* = *true*
- a_{19}, *normal*: *money*+ = *trainPrice*; *trainBooked* = *true*
- a_{20}, *normal*: *money*− = *flightPrice*; *flightBooked* = *false*

For instance, the semantic rule for action a_5 with a *normal* output message increments the value of the *money* variable with the price of the reserved hotel and sets the $hotelReserved$ variable to true. While the same action with an $NoRoomFault$ output message yields the resetting of hotel price to zero.

The domain could easily be enriched with further details. For example, one might consider reservation dates, flight numbers and so on. To take this into account one only needs to define additional variables that store this information and enrich the semantic rules attached to the actions in order to update these variables during execution. This is not illustrated in this paper for paucity of space.

4.3 XSRL

To express requests for composition of web services we propose the language XSRL (Xml Service Request Language) [2, 19]. We also provide an extension of XSRL to deal with the interleaving of planning and execution. The improved XSRL syntax is defined as follows:

```
xsrl    <- '<XSRL>' goal '</XSRL>'
goal    <- achieve-all | proposition | then |
```

```
          vital | prefer | optional | atomic |
          vital-maint | optional-maint

achieve-all     <-
    '<ACHIEVE-ALL>' +goal '</ACHIEVE-ALL>'
then            <-
    '<BEFORE>' goal '</BEFORE>'
    '<THEN>' goal '</THEN>
prefer          <-
    '<PREFER>' goal '</PREFER>'
    '<TO>' goal '</TO>'
vital           <-
    '<VITAL>' proposition '</VITAL>'
optional        <-
    '<OPTIONAL>' proposition '</OPTIONAL>'
atomic          <-
    '<ATOMIC>' proposition '</ATOMIC>'
vital-maint     <-
    '<VITAL-MAINT>' proposition
        '</VITAL-MAINT>'
optional-maint <-
    '<OPTIONAL-MAINT>'
            proposition
    '</OPTIONAL-MAINT>'
proposition     <- '<CONST ATT="true|false">'
                     | var |
                   '<AND>' +proposition
                     '</AND>' |
                   '<OR>' +proposition
                     '</OR>' |
                   '<NOT>' proposition
                     '</NOT>' |
                   '<GREATER>' var '</GREATER>'
                       '<THAN>' rval '</THAN>' |
                   '<LESS>' var '</LESS>'
                       '<THAN>' rval '</THAN>' |
                   '<EQUAL>' var rval '</EQUAL>'
var             <- a..zA..Z[rval]
rval            <- +a..zA..Z0..9.
```

The atomic objects of the language are propositions, that is, boolean combination of linear inequalities and boolean propositions. These can be either true or not in any given state. Propositions are further combined by sequencing operators to form goals. The sequencing operators are: achieve-all, then, prefer. `<ACHIEVE-ALL> +goal </ACHIEVE-ALL>` succeeds when all subgoals defined inside the tag `<ACHIEVE-ALL>` are satisfied, it fails otherwise. `<BEFORE> goal1 </BEFORE><THEN> goal2 </THEN>` is satisfied, if `goal1` is satisfied and, starting from the state where `goal1` is satisfied, `goal2` is also satisfied, it fails otherwise. `<PREFER> goal1 </PREFER><TO> goal2 </TO>` succeeds if `goal1` is satisfiable, if not then it succeeds if `goal2` is satisfiable, it fails if both `goal1` and `goal2` are unsatisfiable. `<ACHIEVE-ALL>` provides a way of collecting goals that have all to be satisfied, the operator `<THEN>` is a way of sequencing goals, while `<PREFER>` enables the user to express user preferences over goals. Note that by nesting preference statements, one may give a total order over any number of sub-goals.

A number of operators take propositions as arguments. These are used to express 'how' to satisfy the propositions. `<VITAL> proposition </VITAL>` is satisfied if there exists a state satisfying `proposition` which is reachable from any future state, it fails otherwise. `<OPTIONAL>proposition </OPTIONAL>` is always satisfied as a goal. Its meaning is that, if there exists a reachable state satisfying `proposition`, then this state must be reached, otherwise the goal is ignored. `<ATOMIC> proposition </ATOMIC>` means that `proposition` have to be reached from the current state despite non-determinism of the domain. If there is no such path to a satisfaction state, it fails. Note the requirements of this operator are stronger than the `<VITAL>` operator. The `<VITAL>` operator does not guarantee satisfaction of the goal if the execution of the plan is always non-deterministically taking the 'wrong' path, this means that non-deterministic action executions always bring the system in a state different from the one in which the final goal is achieved. `<VITAL-MAINT>proposition </VITAL-MAINT>` is satisfied if for all states in the execution path `proposition` is true. If there is a state in which `proposition` is not true, then it fails. `<OPTIONAL-MAINT>` is analogous to the previous one, but as a goal it does not fail if such a path does not exist.

In Sect. 3 we have presented an e-Travel domain and the desire of a user wanting to go to Paris for a one night trip. Let us show how this request is expressed in XSRL. Omitting XML tags, the request in XSRL is:

achieve-all
  before
    achieve-all
      prefer **vital-maint** $hotelPrice < 100$ **to**
                 vital-maint $hotelPrice < 200$
      optional-maint $\neg$ *trainBooked*
      vital *confirmed* $\wedge$
               $location =$ "*Paris*" $\wedge$
               hotelReserved
  then
    atomic *final*
  vital-maint $price < 300$

High-level **achieve-all** expression defines that both its sub-goals (**before** – **then** and **vital-maint** $price < 300$) have to be achieved. **before** – **then** declares that the user first want to: (i) reserve a hotel in Paris; (ii) have a hotel price of 100 preferred to 200; (iii) avoid train if possible. Then, **atomic** *final* requires that the final state of the process has to be reached in any case. In all process states price has to be less than 300 that is defined by **vital-maint** $price < 300$.

4.3.1 Formal semantics of XSRL

To provide the formal semantics of XSRL, we adapt the definitions of plan and of execution structure from [11]. We additionally define the notion of booleanization. A plan is defined as a sequence of actions executed in given context.

Definition 2 (*Plan*) A *plan* for a domain D is a tuple $\pi = \langle C, c_0, action, ctxt \rangle$ where

- C is a set of contexts,
- $c_0 \in C$ is the initial context,
- $action : \mathcal{S} \times C \rightarrow Act$ is the action function,
- $ctxt : \mathcal{S} \times C \times \mathcal{S} \rightarrow C$ is the context function

XSRL in addition to dealing with boolean variables used in typical goal languages, such as the one proposed in [20], deals with variables that range over domains such as reals, integers, and so on. To allow for this we introduce the notion of 'booleanization'. The idea behind booleanization is that constraints expressed in the goal over domains ranging over variables are treated as boolean propositions. For example, consider the expression $money < 100$ with an integer variable $money$. After booleanization this becomes a boolean proposition that can be either true or false.

Definition 3 (*Booleanization*) The booleanization of a domain **D** with respect to a goal g is a tuple $BD = \langle \mathcal{S}', Prop, Act, R, P, Out, Tr', Role_{Act}, Role_P \rangle$ derived from the original domain **D** in the following way. The set of variables Var is replaced by the set of boolean proposition $Prop$ according to the following rules:

- all boolean variables in Var are also in P,
- all linear constraints appearing in g are added as boolean propositions in P,
- all variables in Var that do not appear in g are omitted in P.

The set of states and transition function are changed to fit the earlier introduction of boolean propositions.

An execution structure of a plan over a booleanized domain for a given goal, represents the possible ways a plan can be executed and it is essential to determine the reachability of a given goal from a particular state.

Definition 4 (*Execution Structure*) The execution structure of plan π in the booleanization of domain D with respect to goal g from state s_0 is the structure $K = \langle S, R, L \rangle$, where

- $S = \{(s, c) : action(s, c) \text{ is defined}\}$ is the set of states of the execution structure,
- $R = \{((s, c), (s', c')) : \text{if } \exists (s, c) \rightarrow (s', c') \text{ and } ctxt(s, c, s') = c'\}$ is the relation
- $L(s, c) = \{b \in P\}$,

The execution structure of a plan in a domain represents how the domain is traversed by the plan. Before defining the notion of goal satisfaction, we need to introduce a few elements of notation. We use the symbol σ to denote *finite paths*. S denotes the set of all states in the execution structure K. Given a set Σ of finite paths, the set of minimal paths in Σ is defined as $min\{\Sigma\} = \{\sigma \in \Sigma : \forall \sigma' < \sigma =: \sigma' \notin \Sigma\}$. Given a goal g, $S_g(s)$ represents the the set of finite paths that lead to the satisfaction of goal g from state s, while $F_g(s)$ represents the set of finite paths that lead to a failure. A state s' is said to be *reachable* from the state s if there exists a path starting from s and leading to s'. A plan is denoted by π.

The notion of goal satisfaction $K, s \models g$ is defined in terms of the set of failure states for the goal g on the execution structure K derived from a booleanized domain with starting state s as follows

$K, s \models g$ iff $F_g(s) = \emptyset$

The set of failure states $F_g(s)$ for a goal g from a state s is defined inductively in the following way:

p
$S(s) = \{(s)\}$, $F(s) = \emptyset$, that is, $p \in L(s)$ for all proposition letters p of the booleanized domain, otherwise $S(s) = \emptyset$, $F(s) = \{(s)\}$

$\neg p$, $p_1 \wedge p_2$, $p_1 \vee p_1$
not p, p_1 and p_1, p_1 or p_1

achieve-all $g_1 \ldots g_n$
$S(s) = min\{\sigma : \exists \sigma_1 \leq \sigma\ \sigma_1 \in S_{g_1}(s) \wedge \ldots \wedge \exists \sigma_n \leq \sigma\ \sigma_n \in S_{g_n}(s)\}$
$F(s) = min\{F_{g_1}(s) \cup \ldots \cup F_{g_n}(s)\}$

before g_1 then g_2
$S(s) = \{\sigma_1; \sigma_2 : \sigma_1 \in S_{g_1}(s) \wedge \sigma_2 \in S_{g_2}(last(\sigma_1))\}$
$F(s) = \{\sigma_1 : \sigma_1 \in F_{g_1}(s)\} \cup \{\sigma_1; \sigma_2 : \sigma_1 \in S_{g_1}(s) \wedge \sigma_2 \in F_{g_2}(last(\sigma_1))\}$

prefer g_1 to g_2
$S(s) = \{\sigma_1 : \sigma_1 \in S_{g_1}(s)\} \cup \{\sigma_1; \sigma_2 : \sigma_1 \in F_{g_1}(s) \wedge \sigma_2 \in S_{g_2}(last(\sigma_1))\}$
$F(s) = \{\sigma_1; \sigma_2 : \sigma_1 \in F_{g_1}(s) \wedge \sigma_2 \in F_{g_2}(last(\sigma_1))\}$

atomic p
if there is some infinite path ρ such that $\forall s' \in \rho\ s' \not\models p$ then
$S(s) = \emptyset$, $F(s) = \{s\}$, otherwise:
$S(s) = min\{\sigma : first(\sigma) = s \wedge last(\sigma) \models p\}$, $F(s) = \emptyset$

vital p
$S(s) = min\{\sigma : first(\sigma) = s \wedge last(\sigma) \models p\}$
$F(s) = min\{\sigma : first(\sigma) = s \wedge \forall s' \in \sigma\ s' \not\models p \wedge \forall \sigma' \geq \sigma\ last(\sigma') \not\models p\}$

optional p
- if $\exists \pi : \pi, s \models vital\ p$, otherwise
- if $\forall \pi' \neq \pi : \pi', s \not\models vital\ p$

optional-maint p
- if $\exists \pi : \pi, s \models vital\ maint\ p$, otherwise
- if $\forall \pi' \neq \pi : \pi', s \not\models vital\ maint\ p$

vital-maint p
if $K, s' \models p$ holds for all states s' reachable from s then
$S(s) = \emptyset$, $F(s) = \emptyset$, otherwise $S(s) = \emptyset$, $F(s) = \{s\}$

The satisfaction of a goal is thus defined in terms of whether a goal may fail or not during execution.

A *solution* to an XSRL request is defined in terms of the plan and one of the possible plan executions. This execution is required to satisfy all XSRL goal propositions. Formally,

Definition 5 (*Solution*) A *solution* for a domain D with respect to a goal g from state s_0 is the tuple $\langle \pi, \sigma \rangle$, where:

π is a valid plan for domain D and goal g: $K_{D,\pi}, s_0 \models g$

σ is one of the possible executions of the plan π, that satisfies the goal g

A *problem* of interleaving planning and execution is the finding of a solution for given domain, goal and initial state.

4.4 Interleaving planning and execution

The architecture presented in Fig. 2 divides the framework into three main functional units: a monitor, a planner and an executor. In this section we provide three algorithms for each of these units.

Algorithm 1 monitor(domain d, state s, goal g)

```
π = plan(d, s, g)
if π = ∅ then
  return success
else
  if π = failure then
    if chooseNewProvider(provider) then
      d' = updateDomain(d)
      return monitor (d', s, g')
    else
      g' = generate-rollback-goal()
      monitor(d, s, g')
      return failure
    end if
  end if
  (d', s', g') = execute(π, d, s, g)
  return monitor (d', s', g')
end if
```

The *monitor* (Algorithm 1) is responsible of invoking the planner, recovering from failure and invoking the execution of plans. Starting with a domain, an initial state and an XSRL goal, it invokes the planner requesting the synthesis of a plan. Then monitor analyzes the plan. An empty plan means that the goal has been reached and the request has been successfully met. If the planner returns failure, i.e., the goal cannot be satisfied under the current execution context, then it attempts to change a provider. `chooseNewProvider` contacts the executor module which has a list of possible providers for services and keeps track of which providers have been considering during the execution of the plan. If a new provider can be assigned, the execution proceeds, otherwise the monitor tries to rollback all changes to a domain and returns failure. Finally, if a non-empty plan has been produced, the plan is passed on to the executor by invoking the `execute` function. This function returns an updated domain, current state and the new XSRL goal for which one needs to continue the monitoring.

Note that after the execution phase the original goal can be updated. This is necessary for reachability goals only (goals that are not part of any maintainability goal). The idea behind is simple: if one reserves a hotel he/she does not need to look for plans that reserves hotels in the following iterations. We eliminate such subgoals when they are satisfied.

Algorithm 2 execute(plan π, domain d, state s, goal g)

```
repeat
  a = firstAction(π)
  π = π − a
  if webServiceAction(a) then
    role = Rol_Act(a)
    if noProviderForRole(role) then
      providersList = contactUDDI(role)
      provider = chooseProvider(providersList)
    else
      provider = previouslyChosenProvider(role)
    end if
    message = invoke(a, provider)
  end if
  (d', s', g') = update(d, s, g, a,message)
  if isKnowledgeGathering(a) ∨ goalFailed(g) then
    return (d', s', g')
  end if
until π = ∅
return (d', s', g')
```

The *executor* (Algorithm 2) starts from a plan, a domain, an initial state and an XSRL goal. It iterates by attempting the execution of all the actions of the input plan. The `firstAction` of the plan is stored in the variable a and then removed from the plan. If this action requires interaction with a web service, then one needs to seek for a provider for that action. The construct $role$ stores the role associated with the current action. If the executor has not assigned a provider for that role during the execution so far, then the UDDI is contacted to ask for providers for the given role. A provider is chosen from the list of possible providers using some heuristic function (the first provider, the one for which there are good references, etc.). If, on the other hand, a provider has already been assigned to a role, then we must continue executing the following actions assigned to the role with the same provider. Once the provider has been identified, the provider is invoked with action a and the possible return messages are stored in the `message` variable. The next step is that of updating the domain, the current state and the goal by the effects of having executed the action. This step is necessary as the execution of the action may have brought the system into a new state, it may have changed the values of some variables and it may have satisfied subgoals of the current goal. If the action has been a knowledge gathering action, we have acquired new information and return the current status to the monitor in order to perform re-planning, otherwise we reiterate the cycle by looking at the following action of the plan.

The *planner* function (Algorithm 3) is very short as it relies on an existing planner (MBP, [4, 11]). MBP is a model-based planner which, given a domain description and a goal, synthesizes a plan for the given goal or returns failure if a plan does not exist. Since MBP deals only with domains and goals in which the variables are boolean a preliminary step is necessary in order to adapt MBP to our framework. This reduction, called booleanization, takes all linear constraints over non boolean variables and turns them into boolean propositions which are true, false or undefined in the cur-

Algorithm 3 plan(domain d, state s, goal g)

```
domain_bool = booleanize(d)
repeat
  goal_bool = booleanize(g)
  plan = MBPplan(domain_bool, s, goal_bool)
  if plan != failure then
    return plan
  else
    if there are untraversed combinations of optional goals
     then modify g accordingly
    else
      return failure
    end if
  end if
until true
return failure
```

rent state of the domain. The same reduction is necessary for the goal. The planner returns a sequence of actions for 'reaching' the booleanized goal. For brevity, we do not give the full details of booleanization here, but simply explain the basic concept behind it:

(i) The booleanized domain is as the original one except that instead of the set of variables we have a set of proposition letters specified by the rules (i) and (ii).
(ii) Non boolean linear constraints in the goal are transformed into boolean propositions. Note that two distinct propositions (e.g., $price < 10$ and $price > 5$) are introduced to take into account two constraints on the same variable.
(iii) The truth of the propositions is established recursively by starting from the current state, looking at the current values of the variables and moving along the actions using semantic rules to establish the truth of propositions. In case of conflicting values for a proposition in a state (e.g., the case of two actions with different semantic rules entering in the same state), the state is divided into two states and then the propagation proceeds further from each state. If an action enters an already visited state without proposition conflicting value then the booleanization process is complete.

After the booleanization, the domain is passed to a model-based planner. The planner is invoked until the plan is found or all combinations of optional goals are attempted. The algorithm works with optional goals in the following way. First, it processes them as vital and, in case of failure, the planner function iterates through the optional goals, eliminating (or reintroducing) them from a goal until it can synthesize a plan or all combinations of optional goals have been taken into account. For instance, for an optional goal "booking a train, if possible": first the planner tries to find a plan with "booking a train" as a vital condition and then, in case of failure, it tries to synthesize a plan without any restriction on trains. There is no particular rule on which goals are eliminated first and in which order. The algorithm only ensures to the user a complete search throughout all optional goals combinations. This approach gives us correct but possibly non optimal solution, for instance, the algorithm may find a solution with a hotel price equal to 200, where there may exist hotels with prices equal to 180. This is caused by the non optimality of solutions generated a planner such as MBP. An optimal search would require a higher level of complexity.

4.4.1 Algorithm correctness

Algorithms 1–3 can be shown to be sound and complete under specific assumptions. In case the assumptions are not satisfied, completeness may be at stake. We introduce a number of definitions necessary to prove correctness of the algorithms while in the Appendix we give a proof sketch. First we qualify some actions as being knowledge-gathering and retractable:

Knowledge-gathering action: An action $a \in Act$ is said to be a *knowledge-gathering (or a sensing)* action if it affects at least one knowledge variable, where *knowledge variable* is a variable that can be assigned to a web service returned message value.
Retractable action: An action $a \in Act$ is said to be *retractable* in a state $s \in S$ if there exists a sequence of actions that deterministically, independently of the output of a, brings back to the state s preserving all non-knowledge variables values.

Next we define the notion of a successful execution of a plan.

Definition 6 (*Successful execution*) Given a domain D, goal g and an initial state s_0, an execution σ for a valid plan π is *successful* if it satisfies the goal g when executed: $K_{D,\sigma}, s_0 \models g$.

Let us consider the following assumptions for the purpose of considering algorithms' correctness.

(i) All actions are retractable.
(ii) All knowledge-gathering actions always return the same values for the same provider set and for the same knowledge variable values. That is, an action is knowledge-gathering only for the first invocation on a particular provider.
(iii) An action always has the same output type after invocation for the same provider set and for the same knowledge variable values. However, note that it is not known what is the action output type before its first invocation.
(iv) If there exists a valid plan for an original domain then it is also valid for a booleanized domain.
(v) The goal is allowed to contain only non-knowledge variables.
(vi) All knowledge variables are allowed to be modified by assignments of web service invocations, that is, knowledge variables are prohibited to be changed by other semantic functions.

By assumption (vi), knowledge-gathering actions are service operations that return values known only at execution time. Therefore, replanning is requested after invocation of any knowledge-gathering action and only in this

case. With assumption (v), variables can be divided into two classes:

- *Critical* variables, which can be a part of the goal and their integrity must be preserved. The user can constraint only critical variables.
- *Knowledge-gathering* variables represent the framework knowledge about the web services environment. These variables cannot be constrained in the goal.

We are now in the position to show that Algorithms 1–3 are *sound* and *complete*. As ususal, by soundness we mean that an algorithm returns a solution if there exists at least one solution. Completeness requires the algorithm to return a failure if no solution exists. Formally:

Theorem 1 (Algorithm soundness and completeness) *Given a domain D, a goal g and an initial state s_0, under assumptions (i)–(vi) Algorithms 1–3 are* sound *and* complete, *that is*:

1. *if there exists a non-empty set of solutions Ω, s.t. $\forall \langle \pi, \sigma \rangle \in \Omega : K_{D,\pi}, s_0 \models g$ and $K_{D,\sigma}, s_0 \models g$ then plan π of one of the solutions $\langle \pi, \sigma \rangle$ is found and its successful execution σ is executed by Algorithms 1–3.*
2. *if the set of solutions is empty $\Omega = \emptyset$ then Algorithm 1 returns failure*

The proof of Theorem 1 is shown in Appendix.

The proof of Theorem 1 builds on assumptions (i)–(iv). Let us now consider the importance of these assumptions in the proof of the theorem. If assumption (i) does not hold true then two possible problems arise. First, the algorithms may not find a solution even if it exists, because incomplete information about environment execution of some non-retractable action can lead to a state from which there is no plan which satisfies a goal. For instance, someone has a goal to go to the seaside spending less than 200 euros. If he reserves an expensive flight, say, spending 190 euros, she/he will probably not find a hotel with the rest of her/his money. If the reservation of the flight is retractable, then he can choose a cheaper flight leaving enough money for a hotel. On the other hand, if the flight reservation is non-retractable then he cannot cancel her/his booking. Therefore, the overall goal fails even if there was a solution. Second, algorithm does not ensure the satisfaction of the domain integrity property. The reason is that it depends on satisfaction of a rollback goal, that is possible only if all already invoked actions are retractable. Assumptions (ii) and (iii) are used for proving lemmas. They are necessary to ensure termination, that is, the number of the mutual calls between Algorithms 1–3 must be finite. From assumption (ii) and (iii) it also follows that if the plan is executed in the same context, the result is the same for all executions. Assumption (iv) ensures that the booleanization process preserves the validity of a plan if it is valid for an original domain. The booleanization process, defined in Sect. 4.4, booleanizes the domain and a goal. By assumption (iv) we state that synthesis of a plan is invariant over these changes. Assumptions (v) and (vi) are introduced for simplification of definitions of knowledge-gathering actions that are restricted only to service invocations and the domain integrity property.

Let us now formulate the integrity property of the provided algorithms: if Algorithm 1 fails to find a solution then critical variables must remain unchanged.

Corollary 1 (Domain integrity) *Given a domain D, a goal g and an initial state s_0, under assumptions (i)–(vi) domain integrity is preserved by Algorithms 1–3, that is, if the Algorithm 1 returns failure then critical variables are unchanged.*

For proof of Corollary 1, see Appendix.

The integrity property ensures the satisfaction of the "all-or-nothing" principle. The domain is changed only in case of successful execution and is restored to its initial state if the goal cannot be satisfied. For instance, if the user asks for a hotel then money are spent if and only if the hotel is booked, and no money is taken from the user in case the reservation process fails.

5 Executing a sample XSRL request

In Sect. 3 we have presented an e-Travel domain and the desire of a user wanting to go to Paris for a one night trip. Let us first express such request in XSRL and then show how such request is executed by our framework on the domain in Fig. 1. Omitting XML tags, the request in XSRL is:

achieve-all
  before
    achieve-all
      prefer **vital-maint** *hotelPrice* < 100 **to**
                 vital-maint *hotelPrice* < 200
      optional-maint $\neg$ *trainBooked*
      vital *confirmed* $\wedge$
               location = "*Paris*" $\wedge$
               hotelReserved
  then
    atomic *final*
  vital-maint *price* < 300

This XSRL request is executed as follows: Algorithm 1 is invoked on the domain d (Sect. 4.2) with initial state $s = 1$ and the defined goal g. The first step is to invoke Algorithm 3 with (d, s, g). As there exists a plan for the booleanized version of (d, s, g) the planner returns a plan π with initial actions a_1, a_2, a_4. Subsequently, the execute function (Algorithm 2) is invoked on (π, d, s, g). The first action is a_1=`getHotelPrice`. The role associated with the action a_1 is 'hotel service'. Since this is the first action for this role, UDDI will be contacted to get a list of providers associated with this role. Suppose, to get a list with two providers: 'Hilton' and 'BestWestern' and further that the first one is chosen. Subsequently, the service is invoked. The update of the domain moves the current state to 2. Since a_1 is not a knowledge gathering action, execution of the plan

continues. Following this, the execution proceeds by considering the role of $a_2 = \texttt{price}$ which is again 'hotel service'. Note that this action modifies the knowledge variable price as the interaction with the hotel provider will return a price value. Since we have already chosen the provider 'Hilton' for the hotel service role, we continue with it and store in `message` the price of, say, 150 euros. Next, the domain, goal and current state are updated accordingly. In particular, the new current state is 3 and the goal is unchanged. Since the action is a knowledge gathering one, the executor returns the control to the monitor specifying the updated domain, current state, and goal. The monitor function invokes the planner on the state 3. Again a plan exists because, even if the cost of the hotel is more than the 100 preferred value it is still less than 200 euros. The initial sequence of actions of the new plan is now $a_4, a_5, (a_7$ or $a_1)$. Interleaving of planning and execution proceeds analogously as in the previous points by executing the action $a_4 = \texttt{reserveHotel}$.

The next action a_5 in the plan is non-deterministic, i.e., both states 1 and 5 could be reached. Let us assume that we have received a confirmation message from the provider 'Hilton'. The current state is therefore 5. The following actions request a flight price and reserve a seat in an analogous manner assuming that the cheapest flight provider 'Virgin' is chosen with a ticket price of, say, 200 euros. The choice of 'Virgin' is achieved if the heuristic behind the `chooseProvider` function in Algorithm 2 orders the providers by offered prices. The planner will produce a new plan whose next action is $a_6 = \texttt{getTrainPrice}$ since the flight action will be retracted as the `vital-maint` goal of spending less than 300 euros is violated. Suppose that the price returned by a train provider is of 140 euros. The execution of the plan proceeds smoothly until we reach state 14. The following action is asking the user for confirmation before payment. If it is accepted, the new state is 15 and the goal is updated by considering the subgoal after the `then` statement. The last subgoal of `atomic` *final* is achieved as the final state 18 is always reachable from the current state 15.

6 Conclusions

AI planning provides a sound framework for developing a web services request language and for synthesizing plans for it. Based on this premise we have developed a framework for planning and monitoring the execution of web service requests against standardized business processes. The requests are expressed in the XSRL language and are processed by a framework which interleaves planning and execution in order to dynamically adapt to the opportunities offered by available web services and to the preferences of users. The request language results in the generation of executable plans describing both the sequence of plan actions to be undertaken in order to satisfy a request and the necessary information essential to develop each planned action.

We have defined the full semantics of XSRL in terms of execution structures and we have provided algorithms that satisfy XSRL requests based on UDDI supplied information and information gathered from web service interactions.

Services that XSRL combines in its answer may have conflicting business rules or policies attached to them. The issue of how constraints extracted from different business goals are taken into account in the proposed framework is examined in [15].

An issue for future investigation is the interaction of the system with UDDI registries. In particular, UDDI could be enhanced by providing better support for provider selection, e.g., based on service quality characteristics. This has an impact, among other things, on the `choose-Provider` function. From the point of view of planning, there are several aspects that need to be addressed. For example, the current version of the planner does not keep track of previous computations or "remember" history and patterns of interactions.

Appendix

This section contains a proof sketch for Theorem 1 in Sect. 4.4. To prove Theorem 1 we first need to prove the two following properties about plan executions.

Lemma 1 (Repeatable executions) *Given a domain D, goal g and an initial state s_0, if the assumptions (ii) and (iii) are satisfied, then the execution σ for a plan π is repeatable, that is, the execution σ of the plan π is invariant with respect of the number of times the plan π is executed.*

Proof (Repeatable executions) An execution of a plan depends on an environment. More precisely, it depends on the knowledge variables and on actions output types. From assumption (ii) it follows that knowledge-gathering actions return the same values when invoked in the same context. Thus, the environment for all plan executions is the same. By assumption (iii) for the same knowledge variables values, actions have a deterministic outcome. It follows that all executions of a plan in the same context are the same. □

Lemma 2 (Infinite executions) *Given a domain D, goal g and an initial state s_0, if the assumptions (ii) and (iii) are satisfied, then the infinite execution σ for a plan π is always successful, that is, $K_{D,\pi}, s_0 \models g$.*

Proof (Infinite executions) A plan consists of a finite number of states, contexts and transitions between them, but it can imply executions that have infinitely many action invocations. When a plan is executed, Algorithm 2 checks if the goal fails after every action. Thus, infinite execution is possible only when the goal is satisfied after each action, that is, if $K_{D,\pi}, s_0 \models g$. □

Proof of Theorem 1 (Algorithm completeness) The proof is split into two parts. First, we prove that if at least one solution $\langle \pi, \sigma \rangle$ exists then Algorithm 1 finds a plan π and successfully executes its execution σ. Secondly, the completeness property is proved: Algorithm 1 returns a failure if there is no solution for the given input.

Soundness. From [11] it follows that the planner for extended goals based on model checking always synthesizes a valid plan if at least one exists, and returns failure otherwise. A valid plan is the plan that for a given booleanized domain D_{bool} satisfies the goal g: $K_{D_{\text{bool}},\pi}, s_0 \models g$. From assumption (iv) it follows that if a valid plan exists for domain D then it also exists for a booleanized one, and, therefore, the model-based planner finds it.

Let us assume that solution $\langle \pi, \sigma \rangle$ exists such that $K_{D,\pi}, s_0 \models g$ and $K_{D,\sigma}, s_0 \models g$. From assumption (i) it follows that all actions are retractable. Therefore we can always return to an initial state with the same critical variables values. Thus, without loss of generality, we can assume that at the beginning of every iteration the corresponding compensated actions are executed to return the domain to its initial state.

Let us define the algorithm *iteration* as a pair of planner–executor invocation in Algorithm 1. As it follows from the theorem assumptions (ii) and (iii) the number of algorithm iterations is finite. Therefore, either an executor is stuck in an infinite execution or the planner is invoked for all possible combinations of providers. From Lemma 2 it follows that if an executor processes the infinite execution then the execution satisfies the goal. On the other hand, if the planner is invoked for all possible combinations of providers, it should, finally, synthesize a plan yielding a solution. From Lemma 1 it follows that each plan π has a repeatable execution σ, and, therefore a synthesis of solution plan π implies that executor processes the execution σ from a solution pair $\langle \pi, \sigma \rangle$.

Completeness. It is obvious that if the plan π is synthesized and its execution successfully completed, they form a solution. As follows from Lemma 2 infinite executions are always successful. Therefore, by definition of a solution, a pair $\langle \pi, \sigma \rangle$ is a solution. We have already shown that the number of iterations is finite, therefore, if there is no solution for the problem then Algorithm 1 returns failure in a finite number of steps. □

Finally, we consider the domain integrity property.

Proof of Corollary 1 (Domain integrity) We have already shown that Algorithm 1 returns failure if there is no solution. Before returning a failure, the rollback plan is synthesized and executed. It is always successful according to assumption (i), and, therefore, the algorithm preserves domain integrity. □

References

1. Simple Object Access Protocol 1.1: http://www.w3.org/TR/soap (2000)
2. Aiello, M., Papazoglou, M., Yang, J., Carman, M., Pistore, M., Serafini, L., Traverso, P.: A request language for web-services based on planning and constraint satisfaction. In: Proceedings of the VLDB Workshop on Technologies for E-Services (TES02). Lecture Notes in Computer Sciences, pp. 76–85. Springer (2002)
3. Berardi, D., Calvanese, D., De Giacomo, G., Mecella, M.: Reasoning about Actions for e-Service Composition. In: Proceedings of ICAPS'03 Workshop on Planning for Web Services (2003)
4. Bertoli, P., Cimatti, A., Pistore, M., Roveri, M., Traverso, P.: MBP: a model based planner. In: Proceedings of the IJCAI'01 Workshop on Planning under Uncertainty and Incomplete Information (2001)
5. Bertoli, P., Cimatti, A., Traverso, P.: Interleaving execution and planning via symbolic model checking. In: Proceedings of the ICAPS'03 Workshop on Planning under Uncertainty and Incomplete Information (2003)
6. BPEL: Business Process Execution Language for Web Services, http://www-106.ibm.com/developerworks/library/ws-bpel/ (2002)
7. Casati, F., Sayal, M., Shan, M.-C.: Developing e-services for composing e-services. In: Dittrich, K.R., Geppert, A., Norrie, M.C. (eds.) Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE), Lecture Notes in Computer Science 2068, pp. 171–186. Springer-Verlag (2001)
8. Cassandra, A., Kaebling, L., Littman, M.: Acting optimally in partially observable stochastic domains. In: Proceedings of the AAAI-94, pp. 1023–1028. AAAI Press (1994)
9. Chen, L., Shadbolt, N.R., Goble, C., Tao, F., Cox, S.J., Puleston, C., Smart, P.: Towards a knowledge-based approach to semantic service composition. In: Goos, G., Hartmanis, J., van Leeuwen, J. (eds.) Proceedings of the 2nd International Semantic Web Conference (ISWC2003). Lecture Notes in Computer Sciences 2870, pp. 319–334. Springer-Verlag (2003)
10. Curbera, F., Khalaf, R., Mukhi, N., Tai, S., Weerawarana, S.: The next step in web services. Commun. ACM **46**(10), 29–34 (2003)
11. Dal Lago, U., Pistore, M., Traverso, P.: Planning with a language for extended goals. In: Proceedings of the 18th National Conference of Artificial Intelligence (AAAI-02), pp. 447–454. AAAI Press (2002)
12. Knoblock, C.A., Minton, S., Ambite, J.L., Muslea, M., Oh, J., Frank, M.: Mixed-initiative, multi-source information assistants. In: Proceedings of the World Wide Web Conference, pp. 697–707. ACM Press (2001)
13. Web Service Choreography Description Language: http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427 (2004)
14. Lazovik, A., Aiello, M., Papazoglou, M.: Planning and monitoring the execution of web service requests. In: Orlowska, M., Weerawarana, S., Papazoglou, M. (eds.), Proceedings of the Conference on Service-Oriented Computing (ICSOC-03). Lecture Notes in Computer Sciences 2910, pp. 335–350. Springer, Berlin Heidelberg New York (2003)
15. Lazovik, A., Aiello, M., Papazoglou, M.: Associating assertions with business processes and monitoring their execution. In: Aiello, M., Aoyama, M., Curbera, F., Papazoglou, M. (eds.): Proceedings of the Conference on Service-Oriented Computing (ICSOC-04), pp. 94–104. ACM Press (2004)
16. McDermott, D.: Estimated-regression planning for interactions with Web Services. In: Ghallab, M., Hertzberg, J., Traverso, P. (eds.) Proceedings of the 6th International Conference on AI Planning and Scheduling. AAAI Press (2002)
17. McIlraith, S., Son, T.C.: Adapting Golog for composition of semantic web-services. In: Fensel, D., Giunchiglia, F., McGuinness, D., Williams, M. (eds.) Proceedings of the Conference on Principles of Knowledge Representation (KR) (2002)
18. Paolucci, M., Kawamura, T., Payne, T., Sycara, K.: Semantic matching of web services capabilities. In: Horrocks, I., Hendler, J. (eds.) International Semantic Web Conference (ISWC2002). Lecture Notes in Computer Science 2342, pp 333–347. Springer-Verlag (2002)
19. Papazoglou, M., Aiello, M., Pistore, M., Yang, J.: Planning for requests against web services. IEEE Data Eng. Bull. **25**(4), 41–46 (2002)
20. Pistore, M., Traverso, P.: Planning as model checking for extended goals in non-deterministic domains. In: Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI-01) (2001)
21. Pistore, M., Barbon, F., Bertoli, P., Shaparau, D., Traverso, P.: Planning and Monitoring Web Service Composition. In: Proceedings of the ICAPS'04 Workshop on Planning and Scheduling for Web and Grid Services (2004)
22. Rintanen, J., Constructing conditional plans by a theorem-prover. J. Artif. Intell. Res. **10**, 323–352 (1999)
23. Sirin, E., Hendler, J., Parsia, B.: Semi-automatic composition of web services using semantic descriptions. In: Proceedings of the Web Services: Modeling, Architecture and Infrastructure Workshop in ICEIS 2003 (2003)
24. Srivastava, B., Koehler, J.: Web service composition—current solutions and open problems. In: Proceedings of the ICAPS'03 Workshop on Planning for Web Services (2003)
25. UDDI: Universal Description, Discovery, and Integration. http://www.uddi.org (2002)
26. van der Aalst, W., van Hee, K.: Workflow Management: Models, Methods, and Systems. The MIT Press (2002)
27. WSDL: Web Services Description Language 1.1. http://www.w3.org/TR/wsdl (March 2001)