UNIVERSITY
OF TRENTO

DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

38050 Povo – Trento (Italy), Via Sommarive 14
http://www.dit.unitn.it

PLANNING AND MONITORING THE EXECUTION OF WEB SERVICE REQUESTS

Alexander Lazovik, Marco Aiello
and Mike Papazoglou

September 2003

Technical Report # DIT-03-049

Planning and monitoring the execution of web service requests

Alexander Lazovik[1,2], Marco Aiello[1], and Mike Papazoglou[1,3]

[1] Department of Information and Telecommunication Technologies
University of Trento
Via Sommarive, 14, 38050 Trento, Italy
{lazovik,aiellom}@dit.unitn.it

[2] ITC-IRST
Via Sommarive, 18, 38050 Trento, Italy

[3] Infolab
Tilburg University
PO Box 90153, NL-5000 LE, The Netherlands
mikep@uvt.nl

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 could 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 XSRL (Xml Service Request Language). 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 UDDI and web services instances, and as execution circumstances necessitate change. The 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 for 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 [11] 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) [4]. 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 [4]. 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 their 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 needed every time that there is need for even a slight deviation.

Such advanced functionality can only be 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 [1, 9]. 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 a 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 Section 2 an example in the traveling domain which runs throughout the paper is presented. The architecture of the proposed framework is illustrated in Section 3, in particular, we define the planning domain (3.1), we present an example of domain (3.2),

we introduce an enhanced syntax and semantics for XSRL (3.3) and provide algorithms for satisfying XSRL requests (3.4). In Section 4 we exemplify the functionality of the architecture on the running example. The paper is concluded by a summary and brief overview of related work.

2 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, may be willing to spend no more than 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 night stay in Paris. This will not be considered in this example as it provides no additional explanation of the ideas behind the presented system. The wishes of the user have no 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 travel agents, hotel services and so on. It is common practice these days that standard business descriptions and terminology descriptions be given in XML schema, e.g., for the automotive industry, travel industry, chemical industry and so on (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 standard business process for reserving a trip in the e-travel domain is given in Figure 1. This process is called a business domain 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 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 we reach state (14). In this state a confirmation of an 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)).

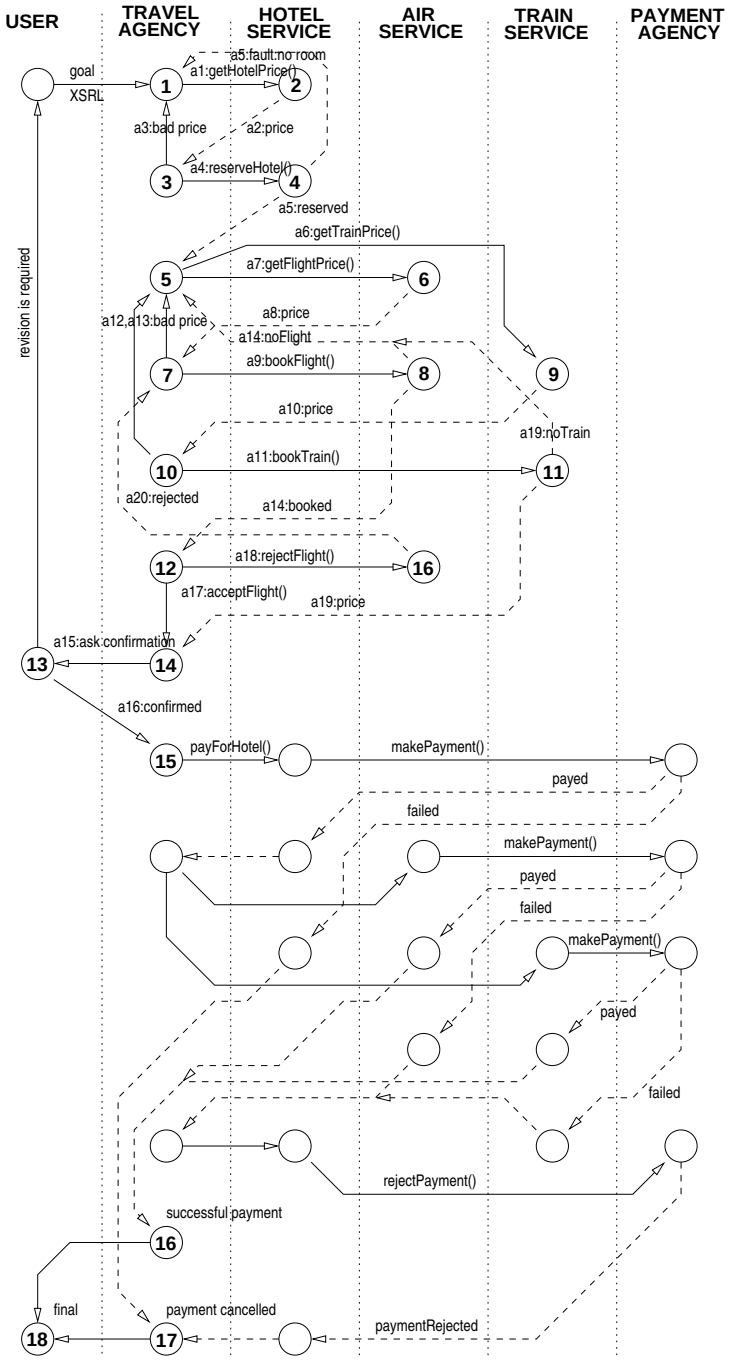

Fig. 1. Business domain.

The lower part of the business process models the payment of the travel package just booked as an atomic action. This means the entire trip is payment atomic.

3 The XSRL framework

The planning architecture proposed is based on the notion of interleaving planning and execution. 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 the state transition diagram given in Figure 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 not available. During execution the system interacts with UDDI to find suitable service providers, in a web service enabled marketplace, and with the user to ask confirmation or request for additional information, if necessary.

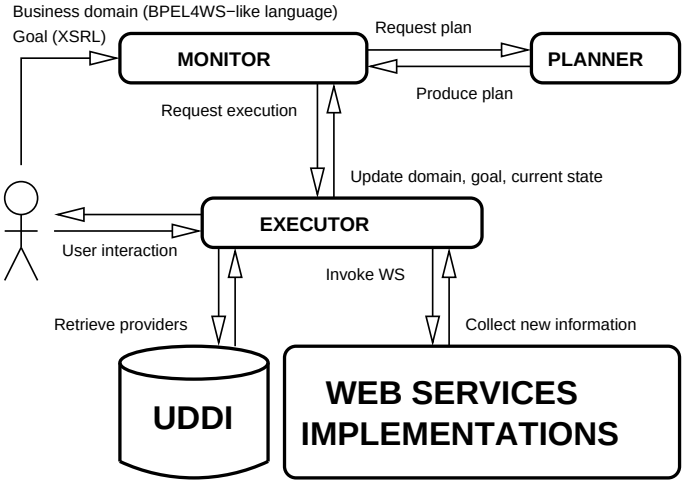

Fig. 2. High-level architecture.

The planning framework, shown in Figure 2 comprises four interacting components: monitor, planner, executor, and the run-time support environment.

Figure 2 illustrates that the user issues 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 correctly specified). The *executor* processes the plan provided

by the planner by invoking the corresponding web services. It is also responsible for finding a set of *providers* (web service implementors) for a particular service in the *UDDI* registry. The executor may contact the user for confirmation if user interaction is demanded by 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 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.

3.1 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 the other things, it lacks formal semantics. Thus, we use BPEL and extrapolate from it state-transition systems enriched with web service domain operators and constructs.

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 non-determinisms 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* $\mathbf{D} =< \mathcal{S}, Var, Act, R, P, Out, Tr, Role_{Act}, Role_P >$, *where:*

- $\mathcal{S}$ *is the set of* states *which represent the states of the business process state-transition system.*
- Var *is the* variable space *generated by the the Cartesian product of a 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* $Tr(s_i, a, o_a) = s_j$ *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* $\exists s' \, \forall o \in Out \;\; Tr(s, a, o) = s'$.
- $Role_{Act} : Act \rightarrow R$ *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 i-th element and the first k elements of $v \in Var$ are knowledge variables.

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.

3.2 A domain instance

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

There are fourteen states $\mathcal{S} = \{1, 2, \ldots, 14\}$ in the upper half of the figure. The set of variables Var is $\{hotelReserved, hotelPrice, location, trainBooked, trainPrice, flightBooked, flightPrice, confirmed, money\}$, 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 nineteen actions that can be performed in the domain $Act = \{a_1, \ldots, a_{19}\}$.

Four roles are involved in the process $R = \{hotel, air, travel\text{-}agency, train\}$ and the $Role_{Act}$ relation associates to each of them the following actions: *hotel* has $\{a_1, a_2, a_4, a_5\}$ associated, *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, $Tr(4, a_5, normal) = 5$, $Tr(4, a_5, NoRoomFault) = 1$ 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 follow:

- $a_2, normal$: $hotelPrice = result$
- $a_3, normal$: $hotelPrice = 0$

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

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 reseting 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 space reasons.

3.3 XSRL

XSRL (Xml Service Request Language) was first introduced in [1, 9] as a request language for compositions of web services in the context of no interleaving of planning and execution (off-line planning). This paper provides an extension of XSRL dealing with the interleaving of planning and execution. The improved XSRL syntax is defined as follows:

```
xsrl      <- '<XSRL>' goal '</XSRL>'
goal      <- proposition | and | then | vital |
             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.
```

Before dealing with the details of the semantics of XSRL constructs, we provide their intuitive meaning. 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, 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 a 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` should 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.

To provide the formal semantics of XSRL, we adapt the definitions of plan and of execution structure from [6]. We additionally define the notion of booleanization. A plan is defined as a sequence of actions executed in given execution 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 *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 [10], deals with variables that range over domains like 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 rule:*

– *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 above 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 execute and 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*

– $S = \{(s, c) : action(s, c)$ *is defined* $\}$ *is the set of states of the execution structure,*
– $R = \{((s, c), (s', c')) : if\ \exists (s, c) \rightarrow (s', c')$ *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 $min\{\Sigma\} = \{\sigma \in \Sigma : \forall \sigma' < \sigma \implies \sigma' \notin \Sigma\}$. 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

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

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

$S(s) = \emptyset$, $F(s) = \{(s)\}$

$\neg p$

not p

$p_1 \wedge p_2$

p_1 and p_1

achieve-all $g_1..g_n$

$S(s) = min\{\sigma : \exists \sigma_1 \leq \sigma\ \sigma_1 \in S_{g_1}(s) \wedge \ldots \wedge \exists \sigma_n \leq \sigma\ \sigma_n \in S_{g_n}(s)\}$

$F(s) = min\{F_{g_1}(s) \cup \ldots \cup F_{g_n}(s)\}$

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 has thus been defined in terms of whether a goal can fail or not during execution.

3.4 Interleaving planning and execution

The architecture presented in Figure 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 models. The *monitor* (Algorithm 1) is responsi-

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 then
      d' = updateDomain(d)
      return monitor (d', s, g)
    else
      return failure
    end if
  end if
  (d', s', g') = execute(π, d, s, g)
  return monitor (d', s', g')
end if
```

ble of invoking the planner, recovering from failure and invoking the execution of plans. It works in the following way. Starting from a domain, an initial state and an XSRL goal, it invokes the planner to synthesize a plan for the input values. It then 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 inability to satisfy a goal 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 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 XSRL goal for which one needs to continue the monitoring. 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

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) then
    return (d′, s′, g′)
  end if
until π = ∅
return (d′, s′, g′)
```

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 in a new state, may have changed the values of some variables and may have satisfied subgoals of the current goal. If the action has been a knowledge gathering action, we have acquired new information and we want to 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)

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

```
domain_bool = booleanize(d)
goal_bool= booleanize(g)
return MBPplan(domain_bool,s,goal_bool)
```

is very short as it relies on an existing planner (MBP, [2, 6]). MBP is a model based planner which, given a domain description and a goal, synthesizes a plan for the given goal of returns failure if a plan does not exist. 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

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

1. 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 below) and new states maybe introduced.
2. Every non boolean linear constraint in the goal is transformed into a boolean proposition. Note that two distinct propositions such as $price < 10$ and $price > 5$ are introduced to take into account two constraints on the same variable.
3. The truth of the propositions is established on the domain 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 independently.

4 Executing a sample XSRL request

To exemplify the concepts behind the algorithms just presented, we provide a sample XSRL request run against the domain introduced in Section 2. In this section, we show how the sample request of Section 2 is handled by the proposed architecture. First, the goal g of going to Paris is expressed in XSRL as shown in Figure 3.

To illustrate the execution of g on the domain d of Figure 1 using the algorithms of Section 3.4, we rewrite the XSRL request omitting XML tags as follows:

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 will execute as follows: Algorithm 1 will be invoked on the domain d in Section 3.2 with initial state $s = 1$ and the goal g in Figure 3. The first step will be to invoke the planner of Algorithm 3 with (d, s, g). As there exists a plan for the booleanized version of (d, s, g) the planner will return a plan π with initial actions a_1, a_2, a_4. Subsequently, the execute function (Algorithm 2) will be invoked on (π, d, s, g). The first action would be a_1=`getHotelPrice`. The role associated with the action a_1 would be 'hotel service'. Since this is the first

```
<XSRL>
 <ACHIEVE-ALL>
  <BEFORE>
   <ACHIEVE-ALL>
    <PREFER>
     <VITAL-MAINT>
      <LESS>
       hotelPrice
      </LESS>
      <THAN>100</THAN>
     </VITAL-MAINT>
    </PREFER>
    <TO>
     <VITAL-MAINT>
      <LESS>
       hotelPrice
      </LESS>
      <THAN>200</THAN>
     </VITAL-MAINT>
    </TO>
   <OPTIONAL-MAINT>
    <NOT>
     trainBooked
    </NOT>
   </OPTIONAL-MAINT>
  </BEFORE>
  <VITAL>
   <AND>
    confirmed
    <EQUAL>
     location
     ''Paris''
    </EQUAL>
    hotelReserved
   </AND>
  </VITAL>
 </ACHIEVE-ALL>
</BEFORE>
  <THEN>
   <ATOMIC>
    final
   </ATOMIC>
  </THEN>
  <VITAL-MAINT>
   <LESS>
    price
   </LESS>
   <THAN>300</THAN>
  </VITAL-MAINT>
 </ACHIEVE-ALL>
</XSRL>
```

Fig. 3. An XSRL request.

action UDDI will be contacted to get a list of providers associated with this role. We assume that we 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=`price` which is again 'hotel service'. Note that this action will modify 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 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 new current 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=`reserveHotel`.

The next action a_5 in the plan is non-deterministic, i.e., both states 1 and 5 could be reached with this action. Let us assume that we have received a confirmation message from the provider 'Hilton' and the current state is therefore 5. The following actions will ask for a flight price and reserve a seat in an analogous manner assuming that the cheapest flight provider 'Virgin' will be

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=`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 will proceed smoothly until we reach state 14. The following action is that of asking the user for confirmation before payment. If this 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 there the final state 18 is always reachable from the current state 15.

5 Summary

AI planning provides a sound framework for developing a web-services request language and for synthesizing correct 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 language XSRL and are dealt by a framework which interleaves planning and execution in order to dynamically adapt to the opportunities offered by available web services and to the desires and 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.

From the AI planning perspective, our work is primarily based on planning as model checking under non-determinism for extended goals [10, 6] using the MBP planner [2]. Extensions toward interleaving planning and execution in the above context are reported in [3]. The latter work emphasizes on state explosion problems rather than information gathering, furthermore, it does not handle numeric values. Various authors have emphasized the importance of planning for web services [5, 7, 8]. In particular, Knoblock et al. [5] use a form of template planning based on hierarchical task networks and constraint satisfaction, in [7] regression planning is used, while in [8] the Golog planner is used to automatically compose semantically described service. 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.

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.

Preliminary experiments with the MBP planner have been conducted to illustrate the feasibility of the approach. In the next phase of experiments we plan to implement the algorithms described in this paper to test the proposed framework.

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 `chooseProvider` 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.

References

1. M. Aiello, M. Papazoglou, J. Yang, M. Carman, M. Pistore, L. Serafini, and P. Traverso. A request language for web-services based on planning and constraint satisfaction. In *VLDB Workshop on Technologies for E-Services (TES02)*, 2002.
2. P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. MBP: A Model Based Planner. In *Proc. IJCAI'01 Workshop on Planning under Uncertainty and Incomplete Information*, 2001.
3. P. Bertoli, A. Cimatti, and P. Traverso. Interleaving Execution and Planning via Symbolic Model Checking. In *Proc. of ICAPS'03 Workshop on Planning under Uncertainty and Incomplete Information*, Trento, Italy, 2003.
4. BPEL. *Business Process Execution Language for Web Services*, August 2002. http://www-106.ibm.com/developerworks/library/ws-bpel/.
5. C. A. Knoblock, S. Minton, J. L. Ambite, M. Muslea, J. Oh, , and M. Frank. Mixed-initiative, multi-source information assistants. In *Proceedings of the World Wide Web Conference*, 2001.
6. U. Dal Lago, M. Pistore, and P. Traverso. Planning with a language for extended goals. In *18^{th} National Conference on Artificial Intelligence (AAAI-02)*, 2002.
7. D. McDermott. Estimated-regression planning for interactions with Web Services. In *6^{th} Int. Conf. on AI Planning and Scheduling*. AAAI Press, 2002.
8. S. McIlraith and T. C. Son. Adapting Golog for composition of semantic web-services. In D. Fensel, F. Giunchiglia, D. McGuinness, and M. Williams, editors, *Conf. on principles of Knowledge Representation (KR)*, 2002.
9. M. Papazoglou, M. Aiello, M. Pistore, and J. Yang. Planning for requests against web services. *IEEE Data Engineering Bulletin*, 25(4):41–46, 2002.
10. M. Pistore and P. Traverso. Planning as model checking for extended goals in non-deterministic domains. In *Proc. 7th International Joint Conference on Artificial Intelligence (IJCAI-01)*, 2001.
11. WSDL. *Web Services Description Language 1.1*, March 2001. http://www.w3.org/TR/wsdl.