MONITORING ASSERTION-BASED BUSINESS PROCESSES

MARCO AIELLO[◇,†]

ALEXANDER LAZOVIK[†]

[†] *DIT, University of Trento, via Sommarive, 14*
38050 Povo (TN), Italy
{*aiellom,lazovik*}*@dit.unitn.it*

[◇] *VITALab, TUWien, Argentinierstrasse, 8*
1040, Wien, Austria

Business processes that span organizational borders describe the interaction between multiple parties working towards a common objective. They also express business rules that govern the behavior of the process and account for expressing changes reflecting new business objectives and new market situations.

We developed a service request language and support framework that allow users to formulate their requests against standard business processes.[16] In this paper, we extend the approach by presenting a framework capable of automatically associating business rules with relevant processes involved in a user request. This framework plans and monitors the execution of the request and assertions against services underlying these processes. Definitions and classifications of business rules (named assertions in the paper) are given together with an assertion language for expressing them. The framework is able to handle the nondeterminism typical for service-oriented computing environments and it is based on the interleaving of planning and execution. Interestingly, the language is able to express both functional and non-functional aspects of the assertions.

1. Introduction

Service-oriented computing (SOC) is the computing paradigm that utilizes services as fundamental elements for developing applications. Services are autonomous platform-independent computational elements that can be described, published, discovered, orchestrated and programmed for the purpose of developing distributed interoperable applications. Basic services, their descriptions, and basic operations (publication, discovery, selection, and binding) that produce or utilize such descriptions constitute the foundation of the service-oriented architecture (SOA).[24]

Currently, the basic SOA does not address advanced concerns such as service management, service choreography and orchestration, service transaction management and coordination, security, and other concerns that apply to all elements in a services-based architecture. Such concerns are addressed by the extended SOA (ESOA).[24] The architectural layers in the ESOA describe a logical separation of functionality in such a way that each layer defines a set of roles and responsibilities and leans on constructs of its predecessor layer to accomplish its mission. The logical separation of functionality is based on the need to separate basic service capabilities provided by the conventional SOA (for example building simple applications) from more advanced service functionality needed for composing services and the need to distinguish between the functionality for composing services from the management of services.

The ESOA introduces among other things a service composition layer which describes the execution logic of web services based applications by defining their control flows and prescribing the rules for consistently managing their unobservable business data. In this way enterprises can describe complex processes that include multiple organizations—such as order processing, lead management, and claims handling—and execute the same business processes in systems from other vendors. This layer is representative of a family of XML-based process definition languages intended for expressing abstract and executable processes that address all aspects of enterprise business processes, including in particular those areas important for web-based services. Most notable among these languages is the Business Process Execution Language for Web Services (BPEL4WS)[5] which defines different types of composition elements like sequencing of service invocations, parallel executions, message sending, and Web Services Choreography Description Language (WS-CDL),[11] that allows to coordinate several business processes from a global point of view.

Web services technologies offer higher-level specifications that provide functionality that supports and leverages web services and enables specifications for integrating automated business processes. Currently, there are two largely complementary initiatives for developing business process definition specifications which aim to define and manage business process activities and business interaction protocols comprising collaborating web services. The terms *orchestration* and *choreography* have been widely used to describe business interaction protocols comprising collaborating web services. Orchestration (as championed by BPEL) describes how web services can interact with each other at the message level, including the business logic and execution order of the interactions from the perspective and under control of a single endpoint. Orchestration refers to an executable business process that may result in a long-lived, transactional, multi-step process model. Choreography (as championed by the Web Services Choreography Description Language) is typically associated with the public (globally visible) message exchanges, rules of interaction and agreements that occur between multiple business process endpoints, rather than a specific business process that is executed by a single party. Choreography is more collaborative in nature than orchestration. It is described from the perspectives of all parties (common view), and defines the complementary observable behavior between participants in business process collaboration. Currently, both orchestration and choreography initiatives are in their infancy and based on WSDL which is strongly emphasizes XML syntax and human-targeted descriptions.

In the web services literature there are several approaches dealing with the monitoring of the assertions over service-enabled business processes. The WS-Policy framework[28] provides a general purpose model for describing a broad range of service requirements, preferences, and capabilities. Typically, it is used when the provider describes the set of conditions the requester should satisfy before invoking the service. RuleML[8] is a powerful technique for expressing business rules over semantically annotated service. On the negative side is the lack of any support for runtime monitoring of the business rules. Srivastava et al.[27] present a review of web service composition techniques and it is argued that planning techniques can be of help in tackling the problem of web service composition. Temporally extended goals, i.e., goals expressing not only desired states to achieve but also conditions on how these are to be reached, are on expressive way of defining complex business goals.[22,16,25] Various authors have emphasized the importance of planning for web services.[12,18,19] In particular, Knoblock et al.[12] use a form of template planning based on hierarchical task networks and constraint satisfaction, McDermott[18] uses regression planning, while McIlraith and Son[19] use the Golog planner to automatically compose semantically described services. Various authors use planners over service description in DAML-S.[26,27] Feasibility of HTN planning algorithms was shown by Wu et al.[31] A finite-state machine framework for automatic composition was introduced by Berardi et al.[3] Fikouras and Freiter[6] present service orchestration based on object-oriented data models. Orriens et al.[21] use service composition rules for governing the business process construction.

We propose the use of an approach based on interleaving planning and execution in the context of nondeterministic domains to deal with assertions and user expressed requests against standard business processes that result in initiating and executing business processes from diverse organizations. The execution of these business processes in the proposed framework is governed by assertions, which are business rules applied to processes. The framework we propose deals with nondeterministic domains, where it tries to satisfy a user request by taking into account how assertions that appear at different levels, e.g., business process, role, and provider level, are applied during business process execution. The framework focuses in particular on the application of business rules that are associated with choreographies. The application of *choreography assertions* usually results in activating only selected business process segments in different organizations. These are the business process segments that satisfy the choreography constraints and consequently can be involved in the result of a user request. In addition, the execution path of business processes is monitored to make certain that environmental conditions, i.e., web service supplied information, conform to the choreography assertions and user request requirements. The proposed framework deals with three kinds of assertions depending on their operational context and complexity: simple assertions, where simple reachability conditions are checked; preservation assertions, where maintenance of some condition needs to be satisfied throughout a path comprising a set of states traversed by the process during execution time; and business entity assertions, where the evolution sequence of a particular variable is monitored for correctness. In this paper we are not concerned with the effect that choreography assertions have on orchestration assertions (assertions that apply in the local context of an organization). We henceforth use the term assertion to mean choreography assertions. As final contribution, we illustrate how the language we propose for expressing assertions can talk about functional as well as non-functional properties of services and their compositions.

The remainder of the paper is organized as follows. In Section 2, we recall the notion of service-oriented business process and introduce various kinds of assertions. Section 3 presents an interleaving planning and execution framework for the monitoring of the execution of user requests and assertions against standardized business processes. Section 3.1 presents a formal domain definition. Section 3.2 introduces the service assertion language XSAL. A formulation of an example in terms of the formally defined domain is the offered in Section 3.3; while Section 3.4 provides algorithms for the working of the framework. Section 4 illustrates how the proposed framework processes the assertions on a travel marketplace example. In Section 5 we show how to express service-level agreements and quality of service aspects in terms of the proposed service assertion language XSAL. Conclusion and future work are discussed in Section 6.

2. Business processes and assertions

A process is a possibly infinite ordering of activities with a beginning and an end; it has inputs (in terms of resources, materials and information) and a specified output. We may thus define a process as any sequence of steps that is initiated by an event, transforms information, materials, or business commitments, and produces an output.[9] In this paper, we consider business processes as a means to represent the control flow of business logic and applications. This is achieved by introducing the notion of a state and an action. A *state* represents the state of the process execution. An *action* represents a business activity, which is modeled as a transition between given states. Each action is executed on behalf of a role. A *role* represents a set of business operations that relate to the same party, e.g., a travel agency. Each role has a number of providers associated with it. The providers can be found by interacting with service registries, e.g., UDDI. A *provider* is the actual party that implements a role, e.g., a specific travel agency. It is convenient to also define the notion of a *process variable*, which is a variable associated with a process, e.g., travel packages, hotel reservations, as the process progresses through its execution path and its states change. The use of process variables guarantees that the execution of a business process can be monitored during execution as the process traverses a set of states where constraints may need to be applied to these variables. Constraints on the variables may represent user request or business rules.

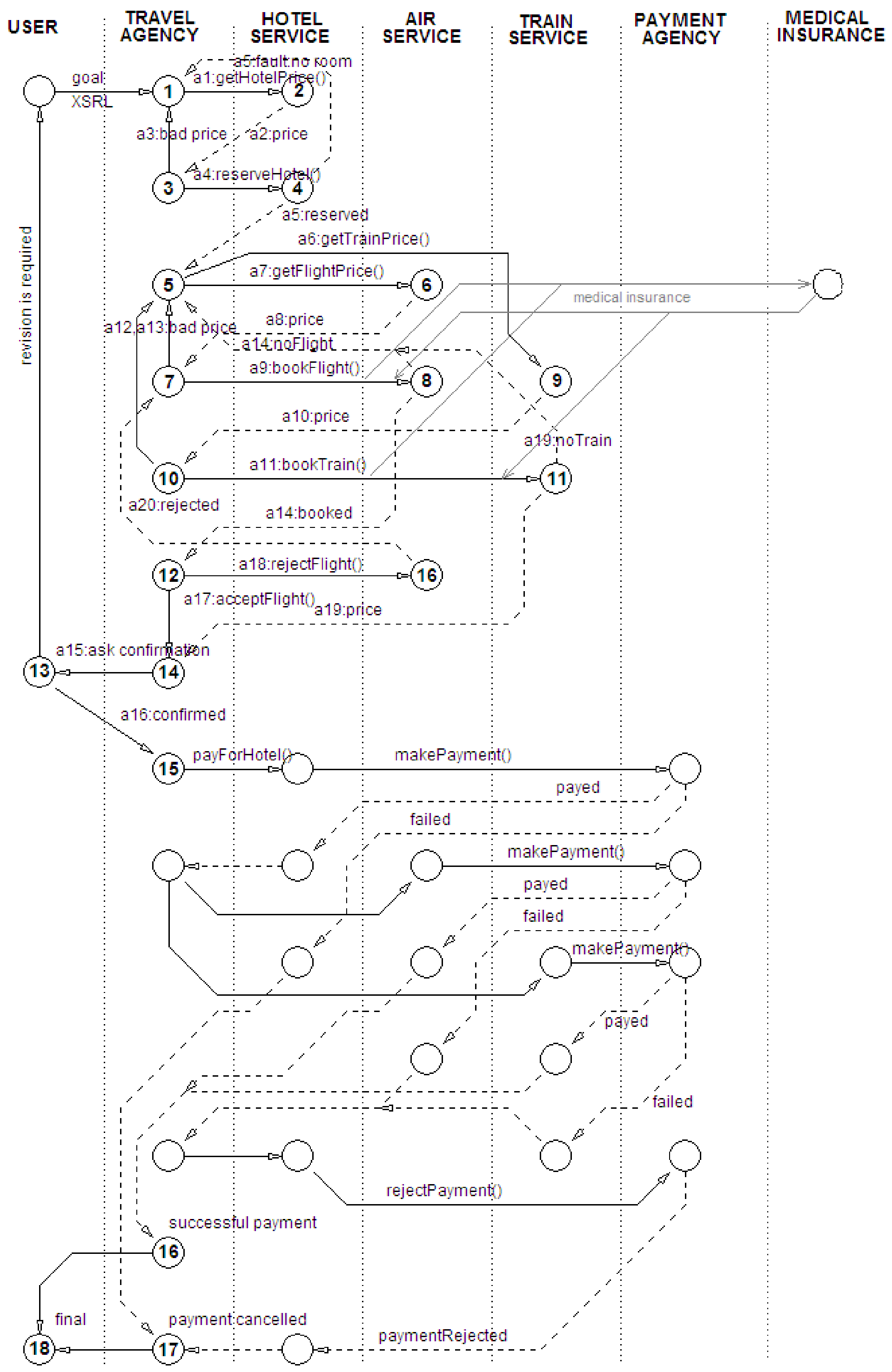

Fig. 1. A travel business process.

2.1. *An example in the travel marketplace*

Consider a user requesting a trip to Nowhereland and having a number of additional requirements regarding such a trip, e.g., that the total price of the trip be lower than 300 euro, the prices of the hotel below 200 euro, avoid using the train, and so on. To be satisfied such a request involves the interaction with various autonomous service providers, including a travel agency, a hotel company and a flight carrier. The services reside in the same travel marketplace and must follow a standard business process for that domain such as the one exemplified in Figure 1. This process 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 then 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 nondeterministic. This is illustrated, for instance, in state (4). In this state the user has accepted the hotel room price however is faced with two possible outcomes, one that a room is not available (where the system transits back to state (1)) and the other one where a room reservation can be made (state (5)). The actual path will be determined at runtime when appropriate services will provide information regarding the availability for the hotel providers.

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.

Services intervening in the process above may have additional requirements and business rules that need to be followed. A particular travel carrier may require advanced payment, a travel agency may want to always have explicit user's approval before committing to a package. At a higher level, different marketplaces may implement the same process but with different rules. For instance, one may additionally require that all air carriers use a specific communication protocol. There also could be a set of Quality of Service requirements expressed in the same manner. For example, one of the involved parties may require additional level of security, the overall process could assert the participating services to support the transactional behavior, etc. This sort of additional business rules are called assertions and are defined next.

2.2. *Assertions*

Actions within a business process are usually distributed between different parties (organization which may play different roles) that can make their changes in different portions of the process. A choreography language can guarantee the consistency of service interfaces, message ordering and message invocations, but it can not be used to check process runtime properties. Safe execution of the business process can only be ensured by a monitoring mechanism that checks the runtime properties of business process and possibly recovers from assertion violations. The monitoring of the business process based on the assertions violations is performed in the following way. First, assertions are published by the party who wants his/her assertions to be applied to business processes and monitored during execution. When executing the business process, the framework allows only those executions to proceed where published assertions are satisfied. If an assertion is violated then the system tries to find an alternative execution path in the business process that does not violate the assertion, if any. Assertions are published on different levels: business process, role or provider. During execution, assertions defined on the business process level are always taken into account; assertions defined by roles are checked only if operations for that role are invoked; provider level assertions are considered if an action of the particular provider is necessary.

More precisely, *monitoring* is a mechanism that ensures the execution of a process is consistent with respect to choreography business rules and user specified requests. As a business process spans several organizations, all of them expect that their business rules are taken into account when executing the process. Business rules are supplied by service providers and are enforced on business processes that are associated with such rules during their execution.

Business rules are expressed in the context of a process by assertions. Next we provide a definition of assertions.

Definition 2.1. An *assertion* is a condition that applies to the execution of a business process.

We use the term assertion and business rule interchangeably. An assertion may be satisfied or not during the execution of a business process, more formally:

Definition 2.2. Given a business process and one of its states, we say that an assertion is *satisfied* if the assertion is true in the specified state and in all future states visited during process execution.

We classify assertions according to two different dimensions: (i) *operational assertions:* on the basis of the operational context and complexity of the assertion; (ii) *actor assertions:* on the basis of the ownership of the assertion. Operational assertions can be further classified into three categories (Table 1):

Simple assertion. A simple assertion is a condition to be satisfied in a *given state* or a *specific set of states* in order to reach a state where the condition is sat-

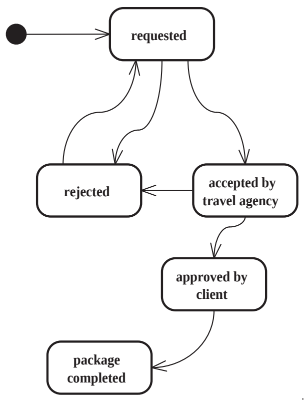

Fig. 2. A travel package business entity assertion.

Table 1. Assertion levels.

Assertion	Where satisfied
simple	in a state, where assertion condition is satisfied
preservation	for all states along the process execution
entity lifecycle	specified entity must preserve evolution specified in assertion

isfied. Simple assertions are also named *reachability* assertions. An example of such an assertion in the context of a travel domain is the requirement of having a medical insurance if the period of being abroad is more than two weeks. To comply with this assertion we must ensure that if the client requests a travel package with duration beyond two weeks then a medical insurance must be subscribed before the business process progresses successfully.

Preservation assertion. A preservation assertion is a condition to be maintained throughout *all states touched* during the execution of a business process. Preservation assertions are also named *maintainability* assertions. In the same travel example as above, consider a situation in which special offers exist for clients who hold a frequent flyer loyalty card, e.g., OneWorld. An assertion for the use of such card would require that all invoked services accept the card to provide discount or points. To comply with this assertion the execution of the business process will attempt to maintain the execution on those paths where services adhering to the loyalty program are available.

Business entity assertion. A business entity assertion is a property that applies to the evolution sequence of a process variable during process execution. For instance, a business entity assertion can be associated with the status of a travel package, as shown in Figure 2. Initially, the "status" variable assumes the value 'requested' when the travel package operation is started. From this state, the request can be 'rejected', if the travel agency fails to satisfy it and, eventually, return in a 'requested' status. Alternatively, the status variable can be 'accepted by travel agency' and subsequently be 'approved by client' and finally become a 'package completed'. To comply with this assertion the execution of the business process must ensure that the states of the travel package variable are reached in the prescribed sequence and only change value according to the valid states of the business entity assertion described above.

Table 2. Assertion levels.

Assertion level	**Where stored**	**Usage**
business process	domain description	concatenated with user request
role	service description	applied if action of the role is invoked
provider	service registry	applied if provider action is invoked

Assertions are not only classified on the basis of their operational dimension but also on the basis of ownership. Based on the ownership criterion, we may distinguish between three types of assertions (Table 2):

Business process-level. The business process level assertions are applied to the whole business process. The business process execution environment verifies these assertions during all executions and for all used services. Assertions of this type are maintained by the party who defines the choreography message sequences. These assertions are stored together with the business process itself. The business entity assertion defined in Figure 2 is an example of of business process level assertion. It defines the possible evolutions of the status of travel package for all executions in the business process. Another example is the following. Usually business processes have an assertion of always reaching the final state despite of the nondeterminism inherent in dealing with web service implementations, e.g., purchase a travel package. This assertion ensures process consistency with organization rules and policies.

Role-level. Role-level assertions are employed for all the providers implementing a specific role. Typically these assertions represent the constraints defined by the standardizing organizations, government, etc. For example, due to governmental laws all travel agencies may require that together with a flight ticket also a medical insurance is purchased, whenever the final destination is in a particular set of locations where health risk exist. These assertions

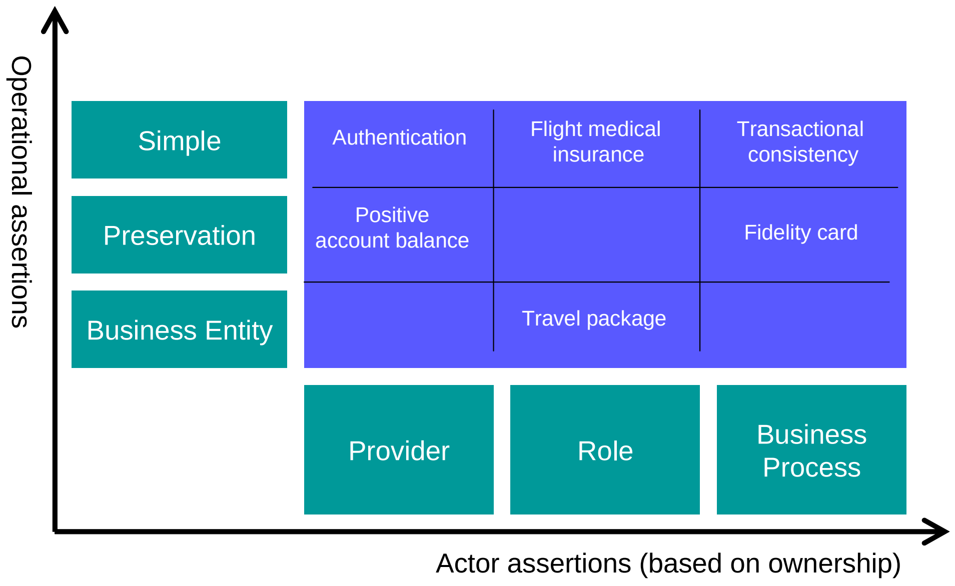

Fig. 3. Two-dimensional classification of assertions, with examples.

are defined together with the service interfaces and stored together with the service descriptions.

Provider-level. At the lowest granularity level assertions are published by a particular service provider. These assertions are stored in service registries together with service implementations. Provider-level assertions are used when a particular provider wants to enforce consistency of the business process and its business rules at runtime. For instance, provider role assertions may involve payment service providers having additional constraints, such as, protocol communication preferences, organization licensing, authentication, etc.

Assertions are classified on both the operational and the ownership dimension. The examples provided during the current presentation are summarized in the matrix in Figure 3. An example of a simple provider assertion could be a requirement of user authentication before using the asserted service. A specific bank provider could require a preservation of a positive amount on the account. This is an example of the preservation/provider assertion. For role-based assertions, examples are the requirement for all travelers to have valid medical insurance or, for travel agencies, that the travel package must be processed with respect to the package entity assertion. Global business process assertion may include, for example, transactional consistency requirement. The users of the business process might have a special fidelity card, that gives them some advantages along the whole execution of the business process.

3. Monitoring framework

One of the biggest challenges that web service enabled e-marketplaces face is the lack of support for appropriate service request languages that retrieve and aggregate services that contribute to the solution of a business problem. Users typically require services from an service-based marketplace on the basis of service characteristics and functionality as supplied by service providers. A service request language provides for a formal means of describing desired service attributes and functionality, including temporal and non-temporal constraints between services, and service scheduling preferences.

Our previous work focused on developing a service request language for web services in service-marketplaces that contains a set of appropriate constructs for expressing requests and constraints over requests as well as scheduling operators.[1,22,16] This language, named XSRL for XML Service Request Language,[1,22] enables a user to formulate complex requests against standard business processes. These standard processes are provided by a *market maker* (a consortium of organizations) that brings the suppliers and vendors together. The market maker assumes the responsibility of creating a service-marketplace administration and performs maintenance tasks to ensure the administration is open for business and, in general, provides facilities for the design and delivery of business processes that meet specific business needs and conforms to industry standards.[24] Standard business processes are described in a choreography language such as Web Services Choreography Description Language (WS-CDL).[11] WS-CDL specifies the common observable behavior of all participants engaged in business collaboration. Each participant could be implemented by completely different languages such as web services applications, whose implementation is based on executable business process languages like BPEL, XPDL and BPML.

XSRL expresses a request and executes it according to the user preferences. The framework that takes XSRL request as input returns a product as the result of the request, e.g., constructs an end-to-end holiday packages (documents) comprising a number of flight and accommodation choices. XSRL is equipped with constructs for expressing quantitative requests, such as, booking of a room for two nights, spending between 100 and 200 euro, etc., but also qualitative operations for sequencing goals, such as, booking a room only after having booked a plane, for stating preferences, e.g., flying rather than taking a train to a destination, for stating the maintaining of a condition during execution, such as, keeping the budget below 500 euro. Loosely speaking, the response documents can be perceived as a series of plans that potentially satisfy a request. In expressing an XSRL request it is important that a user is enabled to specify the way that the request needs to be planned and executed.

We present the XSRL syntax in the following and refer readers to our previous work on the web service requests[16] for explanations of the language semantics and detailed examples of its use.

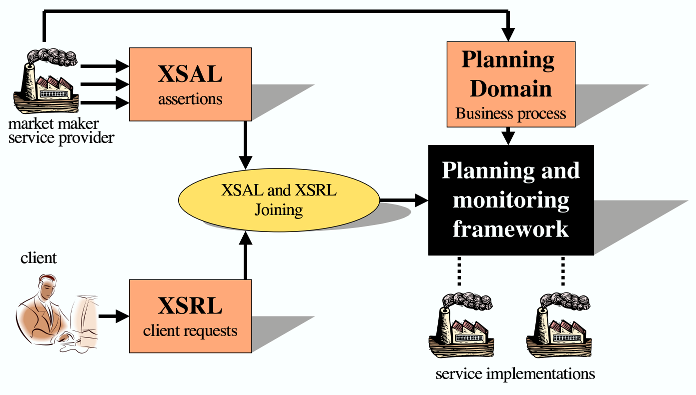

Fig. 4. Handling of XSAL and XSRL requests.

```
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.
```

XSRL and its supporting framework are a powerful tool for enabling a user to formulate requests against business processes but it currently lacks support for

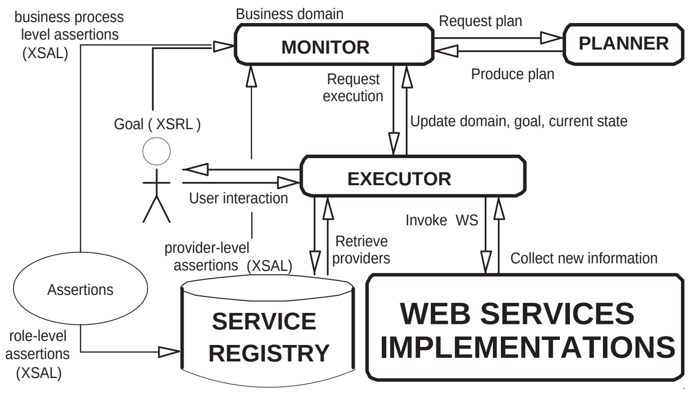

Fig. 5. Planning and monitoring framework.

choreography assertions supplied by service providers and/or market makers that can be associated with the execution of a choreographed process. Assertions are essential means for the actors delivering the services and market makers to apply enterprise/marketplace policies and conditions. This limitation of XSRL is addressed by explaining how it is extended by means of an assertion language, which we name XSAL (XML Service Assertion Language). This language is defined in Section 3.2.

XSRL and XSAL work in tandem during the planning and monitoring of business processes in order to satisfy the user requests in conjunction with applying service provider and marketplace maker supplied assertions. Figure 4 illustrates marketplace makers and actual service providers involved in the marketplace. These are seen to provide a set of assertions in XSAL which govern the behavior and execution of standard business processes. Assertions are associated with the standard business processes against which requests are specified. In Figure 4, a user or client states his/her requests in XSRL. These are combined with the appropriate XSAL assertions and then forwarded to the planning and monitoring framework presented in Figure 5. The planning and monitoring framework interacts with the actual implementations of the services in the service marketplace.

To deal with assertions and user requests we propose a system based on the interleaving of planning and execution. The proposed framework, shown in Figure 5, consists of four components: monitor, planner, executor and runtime support environment and can be seen as an extension of the monitoring framework introduced by Lazovik et al. to deal with user requests expressed in XSRL.[16]

The *monitor* manages the overall process of interleaving planning and execution. It takes user requests, the business process, the business process level assertions and starts interacting with the planner. The *planner* synthesizes a plan and returns it to the monitor. The plan is a sequence of actions to be executed. The planner returns a failure if there was no possible execution satisfying the user request in the given domain without violating the assertions. In case of failure, the monitor eliminates eventual optional goals and assertions or it tries to change service providers. For example, if the business process fails to satisfy the assertion published by one hotel service provider, the framework can try to switch to another hotel service provider whose assertions are less strict. If the planner fails for all possible combinations then the overall execution of the business process fails. Assume that a correct plan exists and therefore it is synthesized. Then the monitor passes it to the executor. The *executor* is responsible for executing the plan. While executing each action of the plan, the executor may gather new information from the service registry or from the service implementations. Whenever new information is obtained, replanning is potentially needed and the domain updated with the just gathered information is returned back to the monitor. The framework works iteratively until the request is satisfied under the given assertions or there is no satisfying execution.

3.1. *Planning domain description*

Using a framework based on interleaved planning and execution demands a formal specification of the business process in terms of planning domains. None of the existing business process definition languages can be straightforwardly used as a domain description for our framework. For example, WS-CDL lacks monitoring mechanisms, BPEL lacks choreography protocol support. However, one can devise extensions and modification to these protocols in order to use them as domain descriptions. The latter is beyond the scope of the present treatment. The domain representation that we adopt is a state-transition system. It is able to represent nondeterministic actions and potentially incomplete knowledge about the environment. Information that is unknown in advance is gathered at runtime by invocations of web services and by contacting the service registry (UDDI) to obtain web service generated information, e.g., current balances, debt histories, etc. Formally, a *domain* D is a tuple $< \mathcal{S}, \mathcal{A}, \mathcal{V}, \mathcal{R}, \mathcal{P} >$, where:

- $\mathcal{S}$ is a set of states a business process can be in.
- $\mathcal{A}$ is a set of actions. An *action* represents an atomic activity of the business process. Each action is associated with a role. If an action has only one outcome it is called *deterministic*, it is called *nondeterministic* otherwise. An action is said to be *retractable* in a particular state if there exists a sequence of actions that deterministically brings back to the state where the action was initially applied, preserving the variables values.
- $\mathcal{V}$ is a set of process variables. A variable set includes all the message definitions that are part of the business process. During the execution of the process some of the variables can be undefined.
- $\mathcal{R}$ is a set of roles. *Roles* represent service interfaces that are used in the business process.
- $\mathcal{P}$ is a set of providers. A *provider* gives a service specification and possibly a service implementation. A provider is associated with one of the process roles.

In addition to a domain definition the following relations between the domain items are specified:

- $\rightarrow$: $\mathcal{S} \times \mathcal{A} \rightarrow 2^{\mathcal{S}}$ is a transition relation between states and actions. A *transition* represents the application of an action to a state and returns a set of states resulting from that action. Transitions are used to represent the skeleton of the business process control flow. An action can be associated with several transitions.
- $role : \mathcal{A} \rightarrow \mathcal{R}$ is a relation between actions and roles associating a role to each action in the domain. If the service interface is defined in terms of WSDL, the relation is extracted from the port types definition.
- $providers : \mathcal{R} \rightarrow 2^{\mathcal{P}}$ is a relation between roles and providers associating a provider to each role. This mapping is defined in the service registry and typically available to the system only at runtime.
- $f_{a,o} : \mathcal{V} \rightarrow \mathcal{V}$ is a semantic function associated with an action a with an outcome o. An outcome can be either 'normal' or 'failure'. For each action, there can be several exception outcomes (i.e., failures) but there can be only a single normal outcome. All action outcomes are defined with the service interface definition.

3.2. *Service assertion language*

In Section 2.2, we showed that the business rules can be expressed using assertions. The assertions are defined as statements that either true or false in any of the given state. They are classified into simple, preservation, and business entity assertions. The assertions need to be stated in a uniform and unambiguous way by the parties involved in the business process. XSAL (Xml Service Assertion Language) serves this purpose. The syntax of XSAL is defined using BNF notation as follows:

```
xsal      <- '<XSAL>' assertion '</XSAL>'
assertion <- statement | achieve-all | then | prefer

achieve-all  <- '<ACHIEVE-ALL>' +assertion '</ACHIEVE-ALL>'
then         <- '<BEFORE>' assertion '</BEFORE>'
                '<THEN>'   assertion '</THEN>'
prefer       <- '<PREFER>' assertion '</PREFER>'
                '<TO>'     assertion '</TO>'

statement    <- entity | vital | optional |
                atomic | vital-maint | optional-maint

entity   <- '< ENTITY VARIABLE = ' var '>'
                       start-from
                      follows*
              '</ENTITY>'
start-from <- '<START-FROM>' proposition '</START-FROM>'
follows    <- '<FOLLOWS>'    proposition '</FOLLOWS>'
              '<BY>'         proposition '</BY>'
```

```
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.
```

One may observe the similarity between XSAL and XSRL. In fact, these two languages share the same expressive power and interpretation, though their intended use is quite different as XSAL is used for expressing assertions while XSRL is used for expressing user requests. Before assessing the formal connection among these two languages we shall first provide the intuitive meaning behind XSAL expressions.

The atomic objects of XSAL are propositions, that is, boolean combination of linear inequalities and boolean propositions. These can be either true or false in any given state. Propositions are further combined by sequencing operators to form assertions. The sequencing operators are: `achieve-all`, `then`, `prefer`. `achieve-all` succeeds when all nested assertions defined inside the tag `<ACHIEVE-ALL>` are satisfied, it fails otherwise. The construct `then` is satisfied when the first statement is satisfied and, from the state where the first statement is satisfied, the second is also satisfied. It fails otherwise. The construct `prefer` succeeds if the first statement is satisfiable, if not then it succeeds if the second statement is satisfiable, it fails if both statements are unsatisfiable.

The operational assertions can be expressed using the XSAL language. All of the following operators take propositions as arguments. The *simple*, or *reachability*, assertions are expressed by XSAL reachability constructs. Formally, reachability constraints require satisfaction of some proposition before execution of the service that has reachability assertion. But strictness of the satisfaction depends on the particular operator. There are three corresponding XSAL operators: `atomic`, `vital`, and `optional`. The `atomic` operator is used when an assertion is strictly important for the party that specifies it and it must be satisfied regardless of any form of nondeterminism. More formally, before executing a service that has this type of assertion, constrained propositions must be true. If there is no such execution then the execution fails immediately. The `vital` operator is used when less strict assertions need to be applied. It tries to find a successful execution to satisfy the constrained proposition. It executes until it has a chance to reach the successful state and fails otherwise. The last operator (`optional`) is the least strict constraint and demands the satisfaction of the assertion if possible, if not the assertion is ignored.

The *preservation*, or *maintainability*, assertions are expressed by XSAL maintainability constructs. This constructs are used when preservation of some value is needed not only in a single state but during a whole execution sequence. When executing a service with such type of assertions only execution that preserves the constrained value can be followed. Retractable actions must be handled with care. In fact, if such an action is invoked and later retracted all associated assertions are ignored. As in the case with simple assertions, maintainability assertions can be of different types from the point of their strictness. We define two types: `vital-maint` and `optional-maint`. The first one is used when the proposition value must be preserved along the whole execution regardless of the nondeterminism. The second (`optional-maint`) is used when the maintenance assertion is optional. The system in this case should always intend to preserve the asserted proposition but if it fails then other executions are still acceptable.

The `entity` expression is used to form *business entity* assertions. This expression begins by relating to a particular variable. It specifies its starting value in the `start-from` statement and it is continued by any number of `follows` statements which specify the possible evolutions of the variable. Assertions of this type are always strict.

The semantics of XSAL can be defined following two trajectories: (i) considering formal semantic definition based on execution structures over planning domains;[16] (ii) providing translation rules for transforming XSAL expressions into XSRL and combining them with XSRL expressions. Recalling that the semantics of XSRL has been defined [16] on the basis of the work on EaGLe,[13] we purse the second trajectory as it is more intuitive and better shows the relation occurring between XSAL and XSRL. As a point of notation, we add a `.t` postfix to denote the XSAL expression translated into XSRL, `(...)` to denote the passing of a parameter to a rule, e.g., `start-from (var)` and `follows (var)` takes `var` as a parameter. Expressions where the translation is omitted are propagated unchanged. The symbol '`*`' in the reduction rule denotes the usual Kleene star.

```
xsal   <- '<XSAL>' statement   '</XSAL>'
xsrl.t =  '<XSRL>' statement.t '</XSRL>'

entity <- '<ENTITY VARIABLE = ' var '>'
                    start-from (var)
                    follows (var)*
            '</ENTITY >'
```

```
entity.t = start-from.t +
             '<THEN>'
                 '<ACHIEVE-ALL>'
                     follows.t*
                 '</ACHIEVE-ALL>'
             '</THEN>'

start-from (var) <- '<START-FROM>' proposition '</START-FROM>'
start-from.t = '<BEFORE>' var proposition '</BEFORE>'


follows (var)   <- '<FOLLOWS>' proposition1   '</FOLLOWS>'
             '<BY>'      proposition2   '</BY>'
follows.t =  '<BEFORE>'
                 '<EQUAL>' var proposition1 '</EQUAL>'
             '</BEFORE>'
             '<THEN>'
                 '<EQUAL>' var proposition2 '</EQUAL>'
             '<THEN>'
```

¿From the translation one notes that the constructs on propositions, on sequencing and preference statements are the same in both languages XSAL and XSRL. The XSAL business entity assertion construct is not present in XSRL and is translated into the sequencing operators `before-then` binding the business entity variable to propositions.

3.3. *A domain instance*

Let us revisit the example in the travel domain introduced in Figure 1 to explain the XSAL use and constructs. Next we present it according to the formal definition of a domain D presented in Section 3.1.

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 $Act = \{a_1, \ldots, a_{19}\}$.

Several roles are involved in the travel business process, that is, $\mathcal{R} = \{$`user`, `hotel`, `air`, `train`, `payment`, `insurance`$\}$. The `user` role represents the requesting party. Typically it is a human user, but it could also be any application software utilizing the business process. The set of actual providers for the roles $\mathcal{R}$ are stored in the service registry.

Arrows in Figure 1 are the process actions. For example, states (3) and (4) are connected by the action `reserveHotel` of the `hotel` role. It has two outcomes: normal, where the variable `hotelReserved` is set to true and exception, where the hotel remains unreserved. This action is an example of a nondeterministic action. The two arrows from the state (4) represent different outcomes for this action. Other examples of actions are `bookFlight` for the `air` role and `getTrainPrice` for `train` role.

Assertions work in conjunction with the travel business process and are defined in XSAL. The business process level assertion that ensures that the process always reaches the `final` state is expressed in the following way: **atomic** $final$. Here and in the following we omit XML tags for brevity. An examples of a role-level assertion is the requirement for insurance in case of prolonged stay abroad: **vital** ($healthRisk \rightarrow insuranceTaken$), where $\rightarrow$ represents logical implication and is expressed using the `<NOT>` and `<OR>` XSAL expressions, as usual.

At the provider level, the hotel provider may prefer, for example, a specific credit card type for payment: **optional** $cardType = \text{VISA}$.

The maintenance assertion for customers of loyalty services that was introduced in Section 3.1 is encoded as follows: **optional**($loyaltyCard \rightarrow (roleType = acceptsLoyaltyCard)$).

In the following we use XSAL to codify the business entity assertion that was presented in Figure 2. The XSAL syntax for this assertion is:

entity *travelPackage*
start-from *requested*
 follows *requested* **by** *rejected* $\vee$ *accepted by travel agency*
 follows *rejected* **by** *requested*
 follows *accepted by travel agency* **by**
  rejected $\vee$ *approved by client*
 follows *approved by client* **by** *package completed*

Additional details like precise hotel information, seats type, payment numbers, etc. can be easily integrated in the above example. To do so, one should add corresponding variables and modify the semantic functions of the actions to take into account the introduced variables. Here we omit such additional details to improve readability.

3.4. *Planning and monitoring algorithms*

Having introduced a planning and monitoring framework and the assertion language XSAL, we present the algorithms which handle XSAL assertions together with XSRL requests. Referring to Figure 5, we recall that the framework consists of three main components, that is, a monitoring, a executor and a planner. We present algorithms for these components separately.

The monitor takes a domain d, that is built on the basis of the business process, an initial state s and a goal g. The initial request of the user to the system is combined together with business process assertions, thus, the monitoring algorithm

Algorithm 3.1 monitor(domain d, state s, goal g)

```
$\pi$ = assert-plan($d, s, g$)
if $\pi = \emptyset$ then
  return success
else
  if $\pi$ = failure then
    if chooseNewProvider($provider$) then
      $d'$ = updateDomain($d$)
      $assert_{provider}$ = extractAssertions($provider$)
      $g'$ = updateGoal($g, assert_{provider}$)
      return monitor ($d', s, g'$)
    else
      $g'$ = generate-rollback-goal()
      monitor($d$, $s$, $g'$)
      return failure
    end if
  end if
  ($d', s', g'$) = execute($\pi, d, s, g$)
  return monitor ($d', s', g'$)
end if
```

is invoked initially with the following goal: **achieve-all**$(request, assert_{bp})$, where $request$ is the user request and $assert_{bp}$ is the set of business process level assertions.

The *monitor* (Algorithm 3.1) is the core of the interleaved planning and execution process. It invokes the planner and the executor in order to satisfy the user requests and the assertions, and it recovers from failures. The algorithm is an extension of the monitoring algorithm presented by Lazovik et al,[16] where the most notable difference is the updating of the goal to take into account the provider level assertions. When a new provider is chosen then the goal is modified in the following way. First, assertions that are associated with the previously assigned provider being de-assigned are eliminated from the goal. Second, assertions of the new provider are added to a goal by using the **achieve-all** operator. The modification of the goal to take assertions into account is performed by the `extractAssertions` and `updateGoal` functions.

The *executor* (Algorithm 3.2) takes a plan and executes it in the marketplace. It contacts the service registry when a service implementation for a given role is necessary, it executes actions of the plan and it checks whether replanning is required. When a new provider is requested from the service registry, its assertions are added to the goal g in the following way **achieve-all**$(g, assert_{provider})$. This is achieved via the `extractAssertions` and `updateGoal` functions.

The planning algorithm is presented in two parts: one dealing with role level assertion and one actually synthesizing a plan. The *assert-planner* (Algorithm 3.3)

Algorithm 3.2 execute(plan π, domain d, state s, goal g)

```
repeat
  $a$ = firstAction($\pi$)
  $\pi = \pi - a$
  if webServiceAction(a) then
    if noProviderForRole($role_a$) then
      $providersList$ = contactServiceRegistry($role_a$)
      $provider$ = chooseProvider($providersList$)
      $assert_{provider}$ = extractAssertions($provider$)
      $g'$ = updateGoal($g, assert_{provider}$)
      return $(d', s', g')$
    else
      $provider$ = previouslyChosenProvider($role_a$)
    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 $\pi = \emptyset$
return $(d', s', g')$
```

checks validity of role-level assertions. The assert-planner works in the following way. First, it produces an initial plan by invoking the plan function (Algorithm 3.4). If the planner succeeds by producing a plan then the assert-planner checks if the plan contains actions with new assertions. If it does, then all assertions are added to the goal and replanning is requested. If the planner fails to synthesize a plan then the assert-planner marks all actions that possibly violate the plan as optional goals and request replanning. Optional goals are added to the current goal g in the following way: $g' = \textbf{achieve-all}(g, \textbf{optional}\neg a_1, \ldots, \textbf{optional}\neg a_n)$, where $\textbf{optional}\neg a_i$ indicates that the action a_i should be avoided, if possible. The assert-planner returns a plan if the user request and all assertions are satisfied and failure otherwise.

The *planner* (Algorithm 3.4) is based on the existing planner based on model checking[4] and is an extension of the one we have proposed for dealing with user requests.[16] The planner is responsible for synthesizing a plan based on a given domain d, an initial state s and a goal g. The planner returns a plan if it exists and failure otherwise. The planner checks all possible combinations of optional goals before returning a failure.

Algorithm 3.3 assert-plan(domain d, state s, goal g)

```
$\pi$ = plan($d$, $s$, $g$)
**if** $\pi \neq$ failure **then**
  $\{assert_{a_1}, \ldots, assert_{a_n}\}$ = extractAssertions($\pi$)
  $g'$ = updateGoal($g$, $\{assert_{a_1}, \ldots, assert_{a_n}\}$)
  **if** $g' = g$ **then**
    **return** $\pi$
  **else**
    **return** assert-plan($d$, $s$, $g'$)
  **end if**
**else**
  $g'$ = checkViolatedActions($g$, $d$)
  **if** $g' = g$ **then**
    **return** failure
  **else**
    **return** assert-plan($d$, $s$, $g'$)
  **end if**
**end if**
```

Algorithm 3.4 plan(domain d, state s, goal g)

```
domain$_{bool}$ = booleanize($d$)
**repeat**
  goal$_{bool}$= booleanize($g$)
  $\pi$ = MBPplan(domain$_{bool}$,$s$,goal$_{bool}$)
  **if** $\pi \neq$ failure **then**
    **return** $\pi$
  **else**
    **if** there are untraversed combinations of optional goals **then**
      modify $g$ accordingly
    **else**
      **return** failure
    **end if**
  **end if**
**until** true
**return** failure
```

4. Monitoring a sample business process

To illustrate the application of the algorithms just presented in the context of the planning and monitoring framework, we use the example presented in Section 2.1 and formalized in Section 3.3. Suppose a user is planning a trip to Nowhereland and is interested in a number of possibilities in connection with this trip. These include making a hotel reservation, avoiding to travel by train, if possible, and spending an overall amount not greater than 300 euro for the whole package. Further, the user prefers to spend less than 100 euro for a hotel room but, if this is not possible, may be willing to spend no more than 200 euro for that room. This would be expressed by the following XSRL request:

```
achieve-all
    achieve-all
      prefer vital-maint $hotelPrice < 100$
          to vital-maint $hotelPrice < 200$
      optional-maint $\neg trainBooked$
      vital $confirmed \wedge$
           $location = ``Nowhereland'' \wedge$
           $hotelReserved$
  vital-maint $price < 300$
```

In addition, suppose that two XSAL business process level assertion such as **atomic** $final$ and the business entity assertion of Figure 2 were published. The system starts by combining the user request with the business process assertions in an **achieve-all** construct. The monitor invokes the assert-planner which in turn invokes the planner. The first actions of the initial plan provided by the planner, given the above goal, the business process assertion and the domain as shown in Figure 1, is the following sequence of actions: `getHotelPrice`, `reserveHotel`.

The monitor then sends the plan to the executor to start interacting with web service implementations. By these invocation a travel agency and a hotel provider are selected and a room is reserved. Suppose that the government considers Nowhereland to be a health risky location. Then the role level assertion **vital** ($healthRisk \rightarrow insuranceTaken$) coming from the service registry together with the travel agency role is considered. At this point, the executor returns control to the monitor which in turn requests a new plan from the assert-planner taking into account the given role-level assertion. The new plan generated will now comprise an action bringing the process in the `obtained a medical insurance` state.

Suppose further that the selected hotel is "MyHotel" which comes with the provider level assertion **optional** $cardType = \text{VISA}$. Then, when the executor runs the request payment from the user the *cardType* is asked to be VISA. If the user refuses such option, execution nevertheless proceeds. Note that if the assertion was **vital** $cardType = \text{VISA}$ then the user's refusal would result in a assertion violation and thus a plan failure.

As for a maintainability assertion, suppose that the travel agency is asked by the client to provide services complying with a given loyalty card. Therefore, the travel agency publishes the following assertion: **optional**($loyaltyCard \rightarrow (roleType = acceptsLoyaltyCard)$). This is taken into account by the assert-planner as soon as the user has specified the card in his request.

As for the business entity assertion requiring a travel package to be assembled following specific rules (Figure 2), this assertion is always taken into account by the assert-planner when providing new plans to the monitor. Finally, the execution proceeds until the travel package is completed and the user approval is requested. At this point the business level assertion **atomic** $final$ is the last to be satisfied. This is achieved by a plan going to the `final` state of the business process.

5. Discussion: expressing QoS properties

In Section 3.2 we introduced the Xml Service Assertion Language and showed how it can be used to express objectives and preferences of the parties involved in the execution of a business process. These objectives may be exposed in a service-level agreement (SLA). But SLAs are not limited to functional requirements, often service providers want to expose to the users of their services various quality of service features. These agreements, often in the form of legal contracts, define what service the provider offers and define the quality of service or QoS that they offer. Because of the formal nature of the SLAs, the quality of service needs to be specified in measurable terms, such as the guaranteed uptime of the service, the guaranteed maximum and average response times of the service, etc.[20] Various non-functional properties of services are the object of SLAs, most notably: availability, accessibility, performance, reliability, security, transactionality, and regulatory. There are several specification proposals to address QoS and SLAs, for example, WS-Policy[28] or Web Service Level Agreement.[30]

Interestingly, XSAL is able to capture most of these qualitative and quantitative QoS properties in its assertions. Next we show examples of how XSAL expression are used to express quality of service properties and be therefore used as fundamental blocks of SLAs. The advantage of using XSAL for this purpose is twofold. On the one hand, it has the appropriate expressive power to express non-functional properties during the agreement negotiation, on the other hand, it comes with a monitoring framework which serves the purpose of checking at runtime that the SLA terms are not violated.

At runtime, instead of rejecting the violated service, the system tries to satisfy the failed assertion or, if that fails too, checks if there are any other business process executions that satisfy the original goal and preferences. For example, let us imagine that the business process failed to present valid credentials to the bank service. The framework first tries to check if there are any activities in the business process that can possibly provide the necessary credentials. If that fails, then the system tries either to ignore the service or select a different bank provider, if there are more available.

Let us now consider what types of service-level agreements can be captured by XSAL expressions. The most relevant categories for QoS requirements in the context of web services are: availability, accessibility, performance, reliability, regulatory, security and transactional behavior.[17] Let us consider them individually.

5.1. *Availability, accessibility, performance, and reliability*

Availability is the quality of whether a web service is available and ready to be invoked. It is defined as a probability of service availability. Sometimes, the time to recover is also added to availability terms, defining the time it takes to repair a temporally non-available service. *Accessibility* is expressed as a probability measure to define whether the service is able to perform a given operation. The service could be available but not accessible if, for instance, the hosting server is overloaded. *Performance* shows how fast the server processes the requests and how many requests are served in the time unit. *Reliability* represents the service degree of being capable of maintaining service quality.

All these properties define the ability of a service to process requests efficiently. These kind of quality aspects are useful for mission-critical business processes requiring high levels of availability and excellent performance.

An example of using XSAL to express availability assertion in the banking domain follows. Suppose one service desires to get the latest financial information in real time. In this case, the business process contains at least two services: a requester and a service providing the necessary data. High performance requirements are expressed in the requester assertion: **vital** $dataFetched \wedge latency \leq 20ms$. Having this assertion the framework checks all the services that provide the financial data (that is those services satisfying the variable $dataFetched$) and selects only the services with the response time lower than $20ms$. The information about the latency time is taken from the service-level agreement of the provider service. The same schema can be applied to check accessibility and reliability quality aspects.

5.2. *Security*

Security is a paramount aspect of service-oriented architectures in its various facets,[10,2,7] such as message encryption, authentication, and access control. Message encryption is usually handled at the platform level, therefore, XSAL's use is limited to simple encryption requirement expressions, e.g., **vital** $encryption \geq 128bit$.

More interesting is the case of authentication. Service-level agreements contain an XSAL assertion that defines the required security information. For example, the bank provider asks for a particular credential to be provided, e.g., **atomic** $login = \texttt{true} \wedge provider = \text{``}Visa''$. When the framework processes this assertion, it tries to satisfy it by looking if there is an execution in the business process that invokes the service operation that satisfies the variable *login* and sets the security provider *provider* to "Visa". If that fails, then the framework tries to satisfy the initial request in a different way, not using the bank or, if that is not possible, tries different bank providers. The key point of using XSAL assertions is that the framework is delegated to adjusting the execution of the business process according to the provider assertions.

Access control service-level agreements can also be expressed in XSAL. Imagine a situation in which a bank exposes several service implementations. Then, the particular implementation is unknown until instantiation of the providers: every service implementation may contain different requirements, assertions, and preferences based on user access rights, therefore, the future executions of the business process strongly depend on the exposed constraints. In other words, the behavior of the business process depends on the access control rights. For example, say a travel agency considers two different types of users: *normal* and *loyal*. The implementation for the second might contain the following assertion: **optional** $loyalpartner = \texttt{true}$. This assertion requires the system to prefer providers that are partners of the travel agency, that might offer special discounts, finally allowing to provide better services to the agency's client.

5.3. *Transactionality*

The loosely coupled and stateless nature of initial web service proposals has posed new challenges for the execution of sequences of service operations which needed to be treated, for instance, atomically. Transactionality of service invocation demands different solutions from traditional database style transactions (see for instance the survey by Papazoglou and van den Heuvel[23]).

Often sequences of service invocations have to support atomic behavior, when, if some service fails, all intermediate changes have to be rolled back. The question such as whether transactions are applicable in the web service environment or what kind of transactions need to be supported (e.g., atomic or long-lived with compensation) are out of the scope of the present paper. However, XSAL with its **atomic** assertion guarantees some form of transactionality.

Consider the traditional transactional model, using two-phase commit with satisfaction of all ACID properties. WS-AtomicTransaction[29] is a standard that deals with this kind of transactions. In this model, transactions are always consistent and atomic. However, this is only achieved if all of the participants support the corresponding transactional agreement. Sometimes this is weakened and services control their desirable transactional behavior by publishing corresponding attributes. For instance, a Java EJB specification may contain the following attributes: `notSupported`, `supports`, `required`, `requiresNew`, `mandatory`, `never`, and a bank provider exposing some of its data might ask for all the invoked services to support transactionality. An XSAL assertion to achieve such a guarantee is the following: **atomic-maint** $attr \neq \texttt{notSupported}$. The framework takes the assertion and allows the publishing service to participate in the transactions with all participants who support the assertion. In the same way, transaction isolation levels could be set according with specific service quality requirements.

For the long-lived transactions the situation is different. Special attention has to be payed to consistency and atomicity, as transactions based on compensation do not guarantee them. XSAL can express such requirements. First, one could check the consistency of data lifecycle by using the **entity** assertion (e.g., Figure 2). In general, to ensure consistency and atomicity the following two operators are used:

atomic and **atomic-maint**. They ensure that execution satisfies the assertion despite any possible nondeterministic failures. For example, *consistent* is a variable that is true in all so-called consistent states and false in all other. The assertion **atomic** *consistent* guarantees that the execution terminates in one of the "consistent" states. That is, before executing a transaction, the framework checks whether all possible executions end up in states that do not violate the assertion. This is a different notion from that of atomicity in ACID transactions as no roll-back is involved, nevertheless is a form of guarantee which starts a sequence of service invocations only if it is possible to arrive to a final state despite any form of nondeterminism.

5.4. *Regulatory*

Mani and Nagarayan[17] define the *regulatory* quality of service aspect which represents the conformance of services to specified standards. This type of service-level agreement is usually processed at the level of underlying platform, since it is truly non-functional property. This kind of non-functional property sis beyond the scope of the presented XSAL framework.

6. Conclusions

We have introduced the assertion language XSAL for expressing business rules in the form of assertions over business processes. XSAL is deployable using the framework we propose which is capable of automatically associating business rules with relevant processes involved in a user request. This allows for consistency and conformance to organizational rules and policies when executing a business process. Additionally, it offers runtime control over its execution. We have classified assertions with respect to two process characteristics: operational context and ownership. With respect to the operational context, we distinguish between simple, preservation and business entity assertions. Regarding ownership, we distinguish between business process, role and provider level assertions. We have then introduced a framework for planning user requests that comply with assertions and monitoring their execution to recover from violating conditions. Specialized algorithms for planning, monitoring and executing requests and assertions have been proposed for this framework. Finally, we have shown how XSAL has the expressive power to define both functional and non-functional properties in the assertions.

The proposed framework and the XSAL language open interesting research issues. One involves the performance of the framework, in particular, the way providers are selected from the service registry is crucial for the efficiency and effectiveness of the architecture. The current proposal does not address this issue, in other words, providers are chosen randomly. A better solution is to select providers based on provider-level assertions (for instance by comparing active assertions), on reputation and history of previous interactions with the provider, or optimizing some specific QoS parameter (e.g., cost of the service or average latency of the service).

The proposed framework plans for requests and assertions, then monitoring the execution of the plans. If there is one possible execution path that can satisfy the request and comply with its associated assertions, this will be found and executed, if not, a failure will be returned. In case that a request succeeds no information is currently provided regarding the quality of the execution. That is, if more possible execution paths complying with the assertions and the user request exist, then only one is guaranteed to be taken. An open issue concerns the comparison of potential solutions (execution trajectories) against optimality metrics, e.g., the shortest plan, the cheapest, the fastest or any other optimality criteria. Solutions to the latter concerns could be addressed resorting to different planning techniques to handle assertions. Constraint programming techniques allow for the identification, for instance, of optimal plans. Initial considerations on how to encode service requests as a constraint programming problem have been proposed and we are currently working on the details of the encoding.[14]

Acknowledgments

We thank Mike Papazoglou for inspiring discussions. The present work is the continuation and extension of the joint work with Mike on XSRL: a service request language and supporting framework that allow users to formulate their requests against standard business processes.[1,22,15]

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. B. Atkinson, G. Della-Libera, S. Hada, M. Hondo, P. Hallam-Baker, J. Klein, B. LaMacchia, P. Leach, J. Manferdelli, H. Maruyama, A. Nadalin, N. Nagaratnam, H. Prafullchandra, J. Shewchuk, and D. Simon. *Web Services Security*. Microsoft, IBM, VeriSign, 1.0 edition, April 2002. available via `http://www-128.ibm.com/developerworks/webservices/library/ws-secure/` on 25/10/2005.
3. D. Berardi, D. Calvanese, G. D. Giacomo, and M. Mecella. Reasoning about Actions for e-Service Composition. In *Proceedings of ICAPS'03 Workshop on Planning for Web Services*, Trento, Italy, June 2003.
4. 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.
5. BPEL4WS. *Business Process Execution Language for Web Services*, May 2003. `http://www-106.ibm.com/developerworks/library/ws-bpel/`.
6. I. Fikouras and E. Freiter. Service discovery and orchestration for distributed service repositories. In *Conf. on Service-Oriented Computing (ICSOC-03)*, Lecture Notes in Computer Sciences, pages 59–74. Springer, 2003.
7. D. Geer. Taking steps to secure web services. *COMP*, 36(10):14–16, October 2003.

8. B. N. Grosof. Representing e-commerce rules via situated courteous logic programs in ruleml*1. *Electronic Commerce: Research and Applications*, 3(1):2–20, 2004.
9. P. Harmon. Analyzing activities. *Business Process Trends*, 1(4), 2003.
10. J. B. D. Joshi, W. G. Aref, A. Ghafoor, and E. H. Spafford. Security models for web-based applications. *Commun. ACM*, 44(2):38–44, 2001.
11. Kavantzas. *Web Services Choreography Description Language 1.0*, April 2004. `http://lists.w3.org/Archives/Public/www-archive/2004Apr/att-0004/cdl_v1-editors-apr03-2004-pdf.pdf`.
12. 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.
13. U. D. Lago, M. Pistore, and P. Traverso. Planning with a language for extended goals. In *18th National Conference on Artificial Intelligence (AAAI-02)*, 2002.
14. A. Lazovik, M. Aiello, and R. Gennari. Encoding requests to web service compositions as constraints. In *Principles and Practice of Constraint Programming (CP-05)*, 2005.
15. A. Lazovik, M. Aiello, and M. Papazoglou. Associating assertions with business processes and monitoring their execution. In M. Aiello, M. Aoyama, F. Curbera, and M. Papazoglou, editors, *Conf. on Service-Oriented Computing (ICSOC-04)*, pages 94–104. ACM Press, 2004.
16. A. Lazovik, M. Aiello, and M. Papazoglou. Planning and monitoring the execution of web service requests. *Journal on Digital Libraries*, 2005. To appear.
17. A. Mani and A. Nagarayan. Understanding quality of service for web services. `http://www-128.ibm.com/developerworks/webservices/library/ws-quality.html`, 2002.
18. D. McDermott. Estimated-regression planning for interactions with Web Services. In *6th Int. Conf. on AI Planning and Scheduling*. AAAI Press, 2002.
19. 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.
20. M. Mullender and M. Burner. Application architecture: Conceptual view. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnea/html/eaappconintro.asp`, 2002.
21. B. Orriens, J. Yang, and M. Papazoglou. Model driven service composition. In *Conf. on Service-Oriented Computing (ICSOC-03)*, Lecture Notes in Computer Sciences, pages 75–90. Springer, 2003.
22. M. Papazoglou, M. Aiello, M. Pistore, and J. Yang. Planning for requests against web services. *IEEE Data Engineering Bulletin*, 25(4):41–46, 2002.
23. M. Papazoglou and W. van den Heuvel. Service oriented architectures. *VLDB Journal*, 2005. To appear.
24. M. P. Papazoglou and D. Georgakopoulos. Service-oriented computing. *Commun. ACM*, 46(10):24–28, 2003.
25. M. Pistore, F. Barbon, P. Bertoli, D. Shaparau, and P. Traverso. Planning and Monitoring Web Service Composition . In *ICAPS'04 Workshop on Planning and Scheduling for Web and Grid Services*, June 2004.
26. M. Sheshagiri, M. desJardins, and T. Finin. A Planner for Composing Services Described in DAML-S. In *Proceedings of ICAPS'03 Workshop on Planning for Web Services*, Trento, Italy, June 2003.
27. B. Srivastava and J. Koehler. Web Service Composition - Current Solutions and Open Problems. In *Proceedings of ICAPS'03 Workshop on Planning for Web Services*, Trento, Italy, June 2003.
28. WS-Policy. *Web Services Policy Framework*, May 2003. `http://www-106.ibm.com/`

developerworks/library/ws-polfram/.

29. WS-Transaction. *Web Services Transaction*, August 2002. http://www-106.ibm.com/developerworks/webservices/library/ws-transpec/.
30. WSLA. *Web Service Level Agreements*, 2003. http://www.research.ibm.com/wsla.
31. D. Wu, E. Sirin, J. Hendler, D. Nau, and B. Parsia. Automatic Web Services Composition Using SHOP2. In *Proceedings of ICAPS'03 Workshop on Planning for Web Services*, Trento, Italy, June 2003.