Planning and Monitoring Web Service Composition

M. Pistore
University of Trento - ITALY
pistore@dit.unitn.it

F. Barbon, P. Bertoli, D. Shaparau, P. Traverso
ITC-irst - Trento - ITALY
[barbon,bertoli,traverso,shaparau]@irst.itc.it

Abstract

The ability to automatically compose web services, and to monitor their execution, is an essential step to substantially decrease time and costs in the development, integration, and maintenance of complex services. In this paper, we exploit techniques based on the "Planning as Model Checking" approach to automatically compose web services and synthesize monitoring components. By relying on such a flexible technology, we are able to deal with the difficulties stemming from the unpredictability of external partner services, the opaqueness of their internal status, and the presence of complex behavioral requirements. We test our approach on a simple, yet realistic example; the results, though preliminary, provide a witness to the potentiality of this approach.

Introduction

The emerging paradigm of web services provides the basis for the interaction and coordination of business processes that are distributed among different organizations, which can exchange services and can cooperate to provide better services, e.g., to third parties organizations or to individuals. One of the big challenges for the taking up of web services is the provision of computer automated support to the composition of service oriented distributed processes, in order to decrease efforts, time, and costs in their development, integration, and maintenance. The ability to automatically plan the composition of web services, and to monitor their execution is therefore an essential step toward the real usage of web services.

BPEL4WS (Business Process Execution Language for Web Services) (Andrews *et al.* 2003) is an emerging standard for the specification and execution of service oriented business processes. BPEL4WS has been designed with two functions in mind. *Executable* BPEL4WS programs allow the specification and execution of the processes internal to an organization (*internal processes* in the following). On the other hand, *abstract* BPEL4WS specifications can be used to specify and publish the invocations of and the interactions with external web services (*external protocols* in the following). Therefore, BPEL4WS offers the natural starting point for web service composition.

In this paper, we devise a planning technique for the automated composition and automated monitoring of web services, e.g., specified with BPEL4WS processes. Automated composition allows providing services that combine other, possibly distributed, services, in order to achieve a given business goal. Starting from the description of the external protocols (e.g., expressed as an abstract BPEL4WS specification) and given a "business requirement" for the process (i.e. the goal it should satisfy, expressed in a proper goal language), the planner synthesizes automatically the code that implements the internal process and exploits the services of the partners to achieve the business goal. This code can be expressed in some process execution language like the executable part of BPEL4WS.

Our planning techniques are also exploited to automatically generate a monitor of the process, i.e., a piece of code that is able to detect and signal whether the external partners behave consistently with the specified protocols. This is vital for the practical application of web services. Run-time misbehaviors may take place even for automatically composed (and possibly validated) services, e.g. due to failures of the underlying message-passing infrastructure, or due to errors or changes in the specification of external web services.

In order to achieve these results, our planner must address the following difficulties, which are typical of several problems of planning under uncertainty:

- Nondeterminism: The planner cannot foresee the actual interaction that will take place with external processes, e.g., it cannot predict a priori whether the answer to a request for availability will be positive or negative, whether a user will confirm or not acceptance of a service, etc.
- Partial Observability: The planner can only observe the communications with external processes; that is, it has no access to their internal status and variables. For instance, the planner cannot know a priori the list of items available for selling from a service.
- Extended Goals: Business requirements often involve complex conditions on the behavior of the process, and not only on its final state. For instance, we might require that the process never gets to the state where it buys an item costing more than the available budget. Moreover, requirements need to express conditional preferences on different goals to achieve. For instance, a process should try first to reserve and confirm both a flight and an hotel from two different service providers, and only if one of the two services is not available, it should fall back and cancel both reservations.

We address these problems by developing planning techniques based on the "Planning as model checking" approach, which has been devised to deal with nondeterministic domains, partial observability, and extended goals. A protocol specification for the available external services is seen as a nondeterministic and partially observable domain, which is represented by means of a finite state machine. Business requirements are expressed in the EaGLe goal language (Dal Lago, Pistore, & Traverso 2002), and are used to drive the search in the domain, in order to synthesize a plan corresponding to the internal process defining the web-service composition. Plan generation takes advantage of symbolic model checking techniques, that compactly represent the search space deriving from nondeterministic and partially observable domains. The planner also generates monitors, i.e., automata that trace the evolutions of external processes according to the run-time observations on the interactions with the external services. At run-time, monitoring is performed by checking the actual interactions w.r.t. those admitted by the monitor; in this way incorrect interactions are detected. Again, symbolic techniques are used to represent compactly belief states and to perform monitoring effectively.

In this paper, we define the formal framework for the planning of composition and monitoring of distributed web services, describe it through an explanatory example, and implement the planning algorithm in the MBP planner. We provide a preliminary experimental evaluation. Though the results are still preliminary, and deserve further investigation and evaluation, the paper provides a witness of the potentialities of our approach.

A web-service composition domain

Our reference example consists in providing a furniture purchase & delivery service. We do so by combining two separate, independent existing services: a furniture producer, and a delivery service.

We now describe the protocols that define the interactions with the existing services. These protocols can be seen as very high level descriptions of the BPEL4WS external protocols that would define the services in a real application.

The protocol provided by the furniture producer is depicted in Fig. 1.3. The protocol becomes active upon a request for a given article; the article may be available or not - in the latter case, this is signalled to the request applicant, and the protocol terminates with failure. In case the article is available, the applicant is notified with informations about the size of the product, and the protocol stops waiting for either a positive or negative acknowledgement, upon which it either continues, or stops failing. Should the applicant decide that the size of the furniture is acceptable, the service provides him with the cost and production time; once more, the protocol waits for a positive or negative acknowledgement, this time terminating in any case (with success or failure).

The protocol provided by the delivery service is depicted in Fig. 1.2. The protocol starts upon a request for transporting an object of a given size to a given location. This might not be possible, in which case the applicant is notified, and the protocol terminates failing. Otherwise, a cost and delivery time are computed and signalled to the applicant; the protocol suspends for either a positive or negative acknowledgement, terminating (with success or failure resp.) upon its reception.

The idea is that of combining these services so that the user may directly ask the combined service to purchase and deliver a given article at a given place. To do so, we exploit a description of the expected protocol the user will execute when interacting with the service. This is depicted in Fig. 1.1. Namely, the user sends a request to get a given article at a given location, and expects either a signal that this is not possible (in which case the protocol terminates, failing), or an offer indicating the price and cost of the service. At this time, the user may either accept or refuse the offer, terminating its interaction in both cases.

Thus a typical (nominal) interaction between the user, the combined purchase & delivery service P&S, the producer, and the shipper would go as follows:

1. the user asks P&S for a certain article A, that he wants to be transported at location L;
2. P&S asks the producer some data about the article, namely its size, the cost, and how much time does it take to produce it;
3. P&S asks the delivery service the price and cost of transporting an object of such a size to L;
4. P&S provides the user a proposal which takes into account the overall cost (plus an added cost for P&S) and time to achieve its goal;
5. the user confirms its order, which is dispatched by P&S to the delivery and producer.

Of course this is only the nominal case, and other interactions should be considered, e.g., for the cases the producer and/or delivery services are not able to satisfy the request, or the user refuses the final offer.

At a high level, the block schema in Fig. 1.4 describes the data flow amongst our integrated web service, the two services composing it, and the user. This can be perceived as (an abstraction of) the WSDL description of the dataflow.

Planning as Model Checking

The "Planning as Model Checking" framework (Bertoli *et al.* 2003) is a general framework for planning under uncertainty. It allows for *non-determinism* in the initial state and in the outcome of action execution. It permits to model planning domains with different *degrees of observability*, ranging from full observability (the whole state of the world is observable at run-time), to null observability (no information is available at run-time), to the general case of partial observability (only some domain information is available at run time). Finally, it allows for expressing *temporally extended planning goals*, i.e., goals that define conditions on sequences of states resulting from the execution of a plan, rather than just on final states.

Planning domains and plans

In our framework, a domain is a model of a generic system with its own dynamics. The plan is also modeled as a system with an internal dynamics, which monitors the evolution of the domain by means of *observations* describing the visible part of the domain state, and which controls the evolutions of the domain by triggering *actions* (see Fig. 2).

A planning domain is defined in terms of its *states*, of the *actions* it accepts, and of the possible *observations* that the domain can exhibit. Some of the states are marked as valid *initial states* for the domain. A *transition function* describes how (the execution of) an action leads from one state to possibly many different states. Finally, an *observation function* defines the observation associated to each state of the domain.

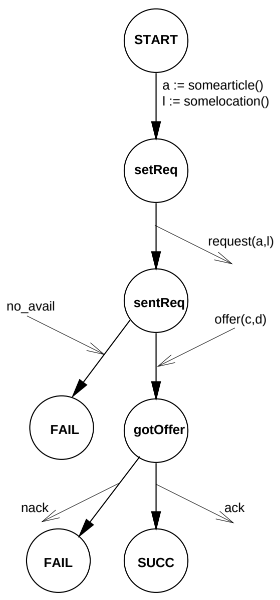

Fig.1.1: The user protocol

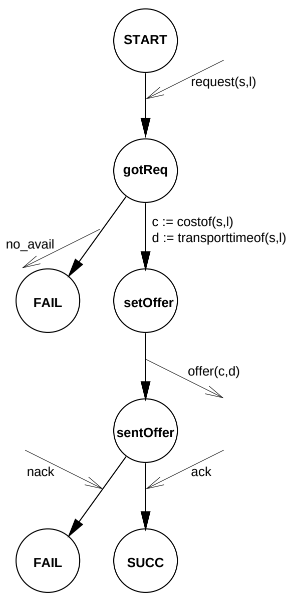

Fig.1.2: The shipper protocol

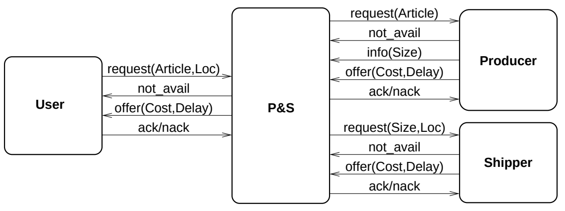

Fig.1.4: The composed web service

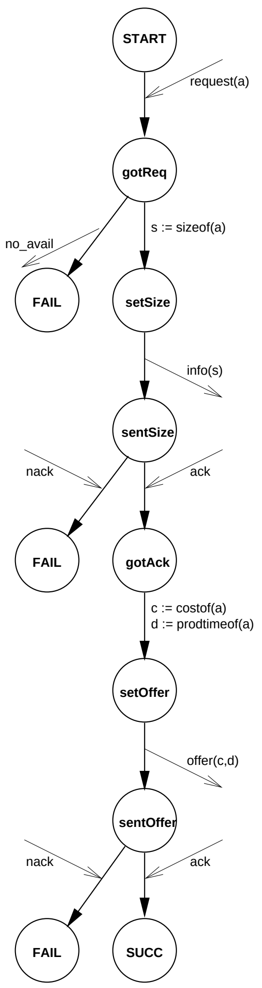

Fig.1.3: The producer protocol

Figure 1: The protocols and their combination.

Definition 1 (planning domain) *A* nondeterministic planning domain with partial observability *is a tuple* $\mathcal{D} = \langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{I}, \mathcal{T}, \mathcal{X} \rangle$*, where:*

- $\mathcal{S}$ *is the set of* states.
- $\mathcal{A}$ *is the set of* actions.
- $\mathcal{O}$ *is the set of* observations.
- $\mathcal{I} \subseteq \mathcal{S}$ *is the set of* initial states*; we require* $\mathcal{I} \neq \emptyset$*.*
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow 2^{\mathcal{S}}$ *is the* transition function*; it associates to each current state* $s \in \mathcal{S}$ *and to each action* $a \in \mathcal{A}$ *the set* $\mathcal{T}(s, a) \subseteq \mathcal{S}$ *of next states.*
- $\mathcal{X} : \mathcal{S} \rightarrow \mathcal{O}$ *is the* observation function.

We are interested in complex plans, that may encode sequential, conditional and iterative behaviors, and are expressive enough for dealing with partial observability and with extended goals. We model them as finite state machines, as follows.

Definition 2 (plan) *A* plan *for planning domain* $\mathcal{D} = \langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{I}, \mathcal{T}, \mathcal{X} \rangle$ *is a tuple* $\Pi = \langle \mathcal{C}, c_0, \alpha, \epsilon \rangle$*, where:*

- $\mathcal{C}$ *is the set of* plan contexts.
- $c_0 \in \mathcal{C}$ *is the* initial context.
- $\alpha : \mathcal{C} \times \mathcal{O} \rightharpoonup \mathcal{A}$ *is the* action function*; it associates to a plan context* c *and an observation* o *an action* $a = \alpha(c, o)$ *to be executed.*
- $\epsilon : \mathcal{C} \times \mathcal{O} \rightharpoonup \mathcal{C}$ *is the* context evolutions function*; it associates to a plan context* c *and an observation* o *a new plan context* $c' = \epsilon(c, o)$*.*

The contexts are the internal states of the plan; they permit to take account, e.g., the knowledge gathered during the previous execution steps. Actions to be executed, defined by function α, depend on the observation and on the context. Once an action is executed, function ϵ updates the plan context. Functions α and ϵ are deterministic (we do not consider nondeterministic plans), and can be partial, since a plan may be undefined on the context-observation pairs that are never reached during plan execution.

The execution of a plan over a domain can be described in terms of transitions between configurations that describe the state of the domain and of the plan.

Definition 3 (configuration) *A* configuration *for domain* $\mathcal{D} = \langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{I}, \mathcal{T}, \mathcal{X} \rangle$ *and plan* $\Pi = \langle \mathcal{C}, c_0, \alpha, \epsilon \rangle$ *is a tuple* (s, o, c) *such that* $s \in \mathcal{S}$*,* $o \in \mathcal{X}(s)$*, and* $c \in \mathcal{C}$*. Configuration* (s, o, c) *may evolve into configuration* (s', o', c')*, written* $(s, o, c) \rightarrow (s', o', c')$*, if* $s' \in \mathcal{T}(s, \alpha(c, o))$*,* $o' \in \mathcal{X}(s')$*, and* $c' = \epsilon(c, o)$*. Configuration* (s, o, c) *is* initial *if* $s \in \mathcal{I}$ *and* $c = c_0$*.*

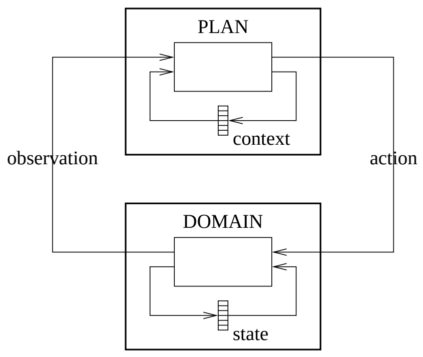

Figure 2: The model of domain and plan.

Intuitively, a configuration is a snapshot of the domain controlled by the plan. Observations are included in configurations in order to take into account that more than one observation may correspond to the same state.

Planning under partial observability

When planning under partial observability, the plan executor cannot in general get to know exactly what is the current state of the domain: the limited available sensing inhibits removing the uncertainty present at the initial execution step, or introduced by executing nondeterministic actions. Thus, at each plan execution step, the plan executor has to consider a set of domain states, each equally plausible given the initial knowledge and the observed behavior of the domain so far. Such a set of states is called a *belief state* ((Bonet & Geffner 2000; Bertoli *et al.* 2001)). Executing an action α evolves an belief B into another belief B' which contains all of the possible states that can be reached through α from some state of B. Notice that hazardous actions may exist that are applicable only for some of the states in the current belief. These actions should not be considered in the plan, since their execution may lead to undesired execution failures.

The available sensing is exploited initially, and after each action execution: if observation o holds after executing action α, the resulting belief shall rule out states not compatible with o. Thus in general, given a belief B, performing an action α (executable on B) and taking into account the obtained observation o gets to a new belief $Evolve(B, \alpha, o)$:

$$Evolve(B, \alpha, o) = \big(\bigcup_{s \in B} \mathcal{T}(s, \alpha) \big) \cap \mathcal{X}^{-1}(o).$$

When planning under partial observability, we are generally interested to knowledge goals, i.e., goals defined in terms of knowledge-level conditions of the form "$\mathbf{K}\, p$" that hold iff every state in the current belief satisfies p. Planning in this framework thus consists in searching through the possible evolutions of initial beliefs, to retrieve a conditional course of actions that leads to beliefs that satisfy the goal.

The search space for a partially observable domain is what is called a *belief space*; its nodes are beliefs, connected by edges that describe the above *Evolve* function. Notice that the space has an inherently nondeterministic nature, stemming from the fact that several different observations may be obtained after executing a given action onto a given belief. As such, the search in a partially observable domain can be described as search inside a fully observable "belief-level" domain $\mathcal{D}_K$ whose states are the beliefs of $\mathcal{D}$, and whose nondeterministic transition function mimics *Evolve*.

Definition 4 (knowledge level domain) *The* knowledge level domain *for domain $\mathcal{D}$ is a tuple $\mathcal{D}_K = \langle \mathcal{B}, \mathcal{A}, \mathcal{O}, \mathcal{I}_B, \mathcal{T}_B, \mathcal{X}_B \rangle$, where:*

- $\mathcal{B} = \{B \in 2^{\mathcal{S}} : \forall s, s' \in B. \mathcal{X}(s) = \mathcal{X}(s')\}$.
- *$\mathcal{A}$ and $\mathcal{O}$ are defined as in the domain $\mathcal{D}$.*
- *$\mathcal{I}_B = \bigcup_{o \in \mathcal{X}(\mathcal{I})} (\mathcal{I} \cap \mathcal{X}^{-1}(o))$ is the set of initial beliefs, i.e. all the beliefs compatible with $\mathcal{I}$, and with some initial observation value.*
- *$\mathcal{T}_B : \mathcal{B} \times \mathcal{A} \to 2^{\mathcal{B}}$ is the* transition function*; it associates to each current belief $B \in \mathcal{B}$ and to each action $a \in \mathcal{A}$ such that $\mathcal{T}(s, a) \neq \emptyset$ for all $s \in B$ the set of next beliefs $\mathcal{T}_B(B, a) = \{Evolve(s, a, o) : o \in \mathcal{X}(\mathcal{T}(B, a))\}$.*
- *$\mathcal{X}_B : \mathcal{B} \to \mathcal{O}$ associates to each belief B the observation $\mathcal{X}_B(B) = \mathcal{X}(s)$ for all $s \in B$.*

Thus, algorithms for planning under partial observability can be obtained by suitably recasting the algorithms for full observability on the associated knowledge-level domain. Actually, the following result holds:

Fact 5 *Let $\mathcal{D}$ be a ground-level domain and g be a knowledge-level goal for $\mathcal{D}$. Let also $\mathcal{D}_K$ be the knowledge level domain for $\mathcal{D}$ and g_K be the goal interpreting g on $\mathcal{D}_K$ (i.e., interpreting $\mathbf{K}\, p$ formulas of g as formulas that hold in the beliefs containing only states that satisfy p). Then Π is a valid plan for g on $\mathcal{D}$ if, and only if, Π is a valid plan for g_K on $\mathcal{D}_K$.*

A potential problem of this approach is that, in most of the cases, this domain is exponentially larger that the ground domain and the approach is not viable in practice. In (Bertoli *et al.* 2001; Bertoli, Cimatti, & Roveri 2001), efficient heuristic techniques are defined to avoid generating the whole (knowledge-level) planning domain. The key idea is to generate only the relevant subset, using heuristics for deciding worth directions to expand knowledges. Although these works are currently limited to reachability knowledge goals, the same idea can be applied also to more general goals, like the ones described in the next subsection.

Notice that the knowledge level domain describes all the valid *observable* behaviors of the domain. That is, it can be fruitfully exploited to monitor whether the observed domain behavior is consistent with the domain model.

Planning with extended goals

Originally, the planning as model checking framework has exploited CTL as temporally extended goal language. CTL (Emerson 1990) is a well-known and widely used language for the formal verification of properties by model checking. It provides the ability to express temporal behaviors that take into account the non-determinism of the domain. Planning with CTL goals has been studied in (Pistore & Traverso 2001; Pistore, Bettin, & Traverso 2001) under the hypothesis of full observability and in (Bertoli *et al.* 2003;

Bertoli & Pistore 2004) in the more general case of partial observability.

However, in many practical cases, CTL is inadequate for planning, since it cannot express different kinds of goals that are relevant for nondeterministic domains. An example is given by goals of the form "Try to achieve condition c and, in case of failure do achieve condition d". In nondeterministic domains it may be impossible to guarantee that condition c is satisfied along all possible execution paths. It is hence necessary to define recovery conditions, or second-preference goals (condition d in our case) that should only be taken into account if the fulfillment of the main goal becomes impossible. Unfortunately, in CTL it is impossible to express such goals. First, CTL is not able to capture the requirement that the plan should "do its best" to achieve condition c whenever possible. Second, CTL is not able to express preferences among goals. EaGLe (Dal Lago, Pistore, & Traverso 2002) has been designed with the purpose to overcome these limitations of CTL and to provide a language for temporally extended planning goals in non-deterministic domains.

Let propositional formulas $p \in \mathcal{P}rop$ define conditions on the states of a planning domain. The *extended goals* $g \in \mathcal{G}$ over $\mathcal{P}rop$ are defined as follows:

$$
\begin{aligned}
g := p \mid g \ \textbf{And} \ g \mid g \ \textbf{Then} \ g \mid g \ \textbf{Fail} \ g \mid \textbf{Repeat} \ g \mid \\
\textbf{DoReach} \ p \mid \textbf{TryReach} \ p \mid \\
\textbf{DoMaint} \ p \mid \textbf{TryMaint} \ p
\end{aligned}
$$

Goal **DoReach** p specifies that condition p has to be eventually reached in a strong way, for all nondeterministic outcomes of plan execution. Similarly, goal **DoMaint** q specifies that property q should be maintained true despite non-determinism. Goals **TryReach** p and **TryMaint** q are weaker versions of these goals, where the plan is required to do "everything that is possible" to achieve condition p or maintain condition q, but failure is accepted if unavoidable. We remark that these operators are very different from operators EF and EG in CTL, which require plans that have just a possibility to achieve the goal.

Construct g_1 **Fail** g_2 is used to model preferences among goals and recovery from failure. More precisely, goal g_1 is considered first. Only if the achievement or maintenance of this goal fails, then goal g_2 is used as a recovery or second-choice goal. Consider for instance goal **TryReach** c **Fail** **DoReach** d. The sub-goal **TryReach** c requires to find a plan that tries to reach condition c. During the execution of the plan, a state may be reached from which it is not possible to reach c. When such a state is reached, goal **TryReach** c fails and the recovery goal **DoReach** d is considered.

Goal g_1 **Then** g_2 requires to achieve goal g_1 first, and then to move to goal g_2. Goal **Repeat** g specifies that sub-goal g should be achieved cyclically, until it fails. Finally, goal g_1 **And** g_2 requires the achievement of both subgoals g_1 and g_2.

A formal semantics and a planning algorithm for EaGLe goals in fully observable domains can be found in (Dal Lago, Pistore, & Traverso 2002).

Planning for web-service composition

In this section we describe how the techniques provided by the planning as model checking framework can be applied to solve the problem of planning and monitoring of web-service composition.

Modeling the domain

The first step to model the domain, and to put ourselves in a position to synthesize a combined web service, is to model each of the protocols of the external partners as a planning domain, according to Definition 1. The states of the domain are used to codify the states of the protocol, the current values of the variables, and the contents of the input and output channels. The modeling of the state of the protocol and of its variables is straightforward. For instance, in the case of the shipper protocol, the possible states are: START, which holds initially; FAIL and SUCC, which indicates the protocol termination with failure and success respectively, and GOTREQ, SETOFFER and SENTOFFER, associated to the intermediate phases of the protocol. The internal variables involved in the shipper protocol describe the size of the required item, the required delivery location, the cost and time for the service. Each of these is presented as a finitely ranged variable; an "undefined" value indicates that the variable has not been initialized yet.

The modeling of channels is more complex; each channel features a channel state (empty or full), and in case it is full, a content. This is modeled as follows:

- a status variable tells us whether the channel is EMPTY or FULL;
- if the channel carries some values, auxiliary channel variables are used to store these values if the channel is FULL. Otherwise, they are undefined.

For instance, in the case of the shipper, the input channel REQUEST conveys the item size and the location as data, and is hence modeled with variables REQUEST.STATUS, REQUEST.SIZE, and REQUEST.LOC.

The only actions in the model correspond to channel read/write operations, since these are the only operations available to an external agent for controlling the evolution of the protocol. A receive action can only be executed on an output channel of the protocol which is FULL; its execution empties the channel and updates the protocol state. A send action, on the other hand, is possible only on an EMPTY input channel of the protocol; if the input channel carries values, the action also specifies the actual values transmitted on the channel; when this action is executed, the channel is marked as FULL, and the transmitted values are stored in the appropriate channel variables.

The transitions in the model capture different aspects of the evolution of the system. From one side, they describe the update of the channel variables due to send/receive actions executed by the agent interacting with the protocol. From the other side, they model the internal evolution of the protocol. This internal evolution may correspond to the receive/send operations complementary to the send/receive actions executed externally. Or it may correspond to updates in the internal variables, when assignment steps of the protocols are executed. Finally, to model the possibility of a protocol being idle, a no-op operation can be performed by the protocol along a transition. In this case, the only updates that occur in the transition are those related to the send/receive actions of the external agent.

The observations of the protocol are limited to:

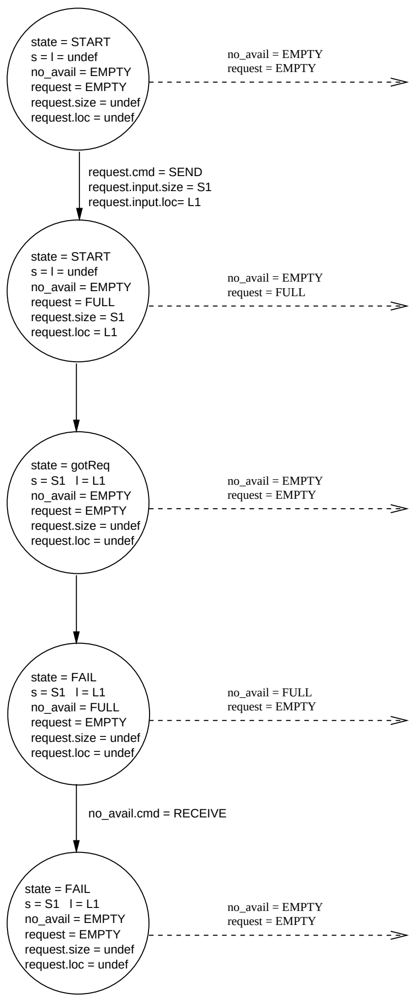

Figure 3: The shipper protocol modeling.

- the status (EMPTY or FULL) of the input and output channels;
- whenever a channel is read, the values contained in that channel.

Fig. 3 describes a portion of the domain modeling the shipper protocol. It corresponds to the reception of a request and to the corresponding negative (*nack*) answer. The actions executed by the external agent are represented on the vertical arrows. The observations corresponding to the states of the protocol are represented on the horizontal arrows.

Monitoring web service interaction

The construction defined above defines the ground domain for each of the external partners. Given one of these ground domains, we can perform the power-set construction described in Definition 4 to obtain the associated knowledge level domain. Since this describes all the admissible observable behaviors of the domain, and in particular all plausible message sequences amongst this domain (service) and its partners, we can use it to monitor whether the behaviors of the external partners conform to the protocol.

Such a monitor is able to detect protocol violations at run-time, and trigger an appropriate handling of such a failure. Of course, it cannot detect the undesired condition where a portion of the protocol is stuck, and does not proceed sending a message when it should. However, this can be easily captured by enriching the monitor with a time-out mechanism, such that unexpected message sending time-outs are properly signalled.

Planning for web service composition

The goal of the planning task is to synthesize the process that interacts with the three external processes in order to provide the purchase & delivery service to the users.

The planning domain consists of the combination of the three ground-level domains for the external partners. Formally, this combination is a synchronous product, which allows the three protocols to evolve independently and in parallel.

The business goal of P&S can be described as follows:

1. The service should try to reach the ideal situation where the user has confirmed his order, and the service has confirmed the associated (sub-)orders to the producer and shipper services. In this situation, the data associated to the orders have to be mutually consistent, e.g. the time for building and delivering a furniture shall be the sum of the time for building it, and that for delivering it.
2. Upon failure of the above goal, e.g., because the user refuses the offer, the service should absolutely reach a fall-back situation where every (sub-)order has been canceled. That is, there should be no chance that the service has committed to some (sub-)order before the user cancels his order.

This goal can be expressed by the following EaGLe formula, where the **TryReach** and **DoReach** clauses represents the above portions 1 and 2 of the requirement.

TryReach

$$\begin{aligned}&\mathbf{K}\,(user.succ \wedge producer.succ \wedge shipper.succ \wedge\\&\quad user.d = add\_delay(producer.d, shipper.d) \wedge\\&\quad user.c = add\_cost(producer.c, shipper.c))\end{aligned}$$

Fail DoReach

$$\mathbf{K}\,(user.fail \wedge producer.fail \wedge shipper.fail)$$

We remark that the capabilities of expressing failure of EaGLe are essential here. Also notice that the above planning goal has to be interpreted at the knowledge level: the executor wants to reach a state where it *knows* that a successful situation has been reached; or, subordinately, to reach a state where it *knows* that all orders have been called off.

The planning domain is only partially observable by the executor of the process that we want to synthesize. Thus, solving the problem implies using dedicated algorithms for planning under partial observability with EaGLe goals, or, alternatively, planning for the fully observable associated knowledge level domain, and interpreting the goal as a ground goal (rather than as a knowledge-level goal). We pursue this latter approach, so that we can reuse existing Eagle planning algorithms under full observability.

We proceed as follows. We generate the knowledge level domain by combining the three monitors defined previously. Similarly to what happens for the ground level domains, this computation consists of a synchronous product. Inside the knowledge level domain, we mark as *success* the belief states which contain only states satisfying the ideal condition that the services *tries* to reach (i.e. $user.succ \wedge producer.succ \wedge shipper.succ \wedge user.d = add\_delay(producer.d, shipper.d) \wedge user.c = add\_cost(producer.c, shipper.c) \wedge empty\_channels$), and as $failure$ the belief states which contain only states satisfying the condition that the service *has* to reach in case the preferred objective fails (i.e. $user.fail \wedge producer.fail \wedge shipper.fail \wedge empty\_channels$). Finally, we plan on this domain with respect to EaGLe goal:

$$\textbf{TryReach}\ success\ \textbf{Fail}\ \textbf{DoReach}\ failure.$$

Fact 5 guarantees that the approach outlined above for planning under partial observability with EaGLe goals is correct and complete.

Experimental results

In order to test the effectiveness and the performance of the approach described above, we have conducted some experiments using the MBP planner. Some extensions to the planner have been necessary to the purpose. First, we have implemented the procedure for translating protocols similar to the ones described in Fig. 1 into the ground-level planning domains represented in the input language of MBP. Second we have implemented a routine that performs the power-set construction of Definition 4 and that builds the monitors corresponding to the three external protocols. MBP already provides algorithms for planning with EaGLe goals, which we have exploited in the last step of the planning algorithm.

We have run MBP on four variants of the case study considered in this paper, of different degrees of complexity. In the easiest case, CASE 1, we considered a reduced domain with only the user and the shipper, and with only one possible value for each type of objects in the domain (article, location, delay, cost, size). In CASE 2 we have considered all three protocols, but again only one possible value for each type of object. In CASE 3 we have considered the three protocols, with two objects for each type, but removing the parts of the shipper and producer protocols concerning the size of the product. In CASE 4, finally, is the complete protocol. We remark that CASE 1 and CASE 2 are used to test our algorithms, even if they are admittedly unrealistic, since the process knows, already before the interaction starts, the article that the user will ask and the cost will be charged to the user. In CASE 3 and CASE 4, a real composition of services is necessary to satisfy the goal.

In all four cases we have experimented also with a variant of the shipper protocol, which does not allow for action *nack*.

The experiments have been executed on an Intel Pentium 4, 1.8 GHz, 512 MB memory, running Linux 2.4.18. The results, in Fig. 4, report the following information:

- Generate: the time necessary to generate the MBP description of knowledge domains from their description of the three external protocols.
- BuildMonitor: the time necessary to parse and build the three internal MBP models corresponding to the monitors of the three external protocols; after the models have been built, it is possible to monitor in real-time the evolution of the protocols.
- BuildDomain: the time necessary to parse and build the internal MBP model of the combination of the three knowledge level domains.
- Planning: the time required to find a plan (or to check that no plan exists) starting from the planning domain built in the previous step.
- Result: whether a plan is found or not.

The last two results are reported both in the original domains and in the domains without "*nack*" being handled by the shipper.

The experiments show that the planning algorithm correctly detects that it is not possible to satisfy the goal if we remove the *nack* action handling in the shipper, since we cannot unroll the contract with the shipper and to satisfy the recovery goal **DoReach** $failure$ in case of failure of the main goal.

MBP has been able to complete all the planning and monitor construction tasks with a decreasing performance when the complexity of the protocols grow. Currently, the most serious bottleneck is in the time required to build the internal MBP representation of the knowledge level planning domain (BuildDomain). This is due to the usage of a dumb algorithm for the generation of the knowledge level domain (currently, the file specifying the domain for CASE 4 is more than 50MB long). An optimized algorithm should drastically reduce the BuildDomain time. The long time required in Planning in the case the algorithm finds a plan w.r.t. the case there is no plan is mostly due to the time required to extract the plan, once the existence of a plan has been proven. Also the plan extraction time can be optimized by exploiting the specific form of plans that we have in the domains under investigation.

Related work and conclusions

In this paper, we define, implement and experiment with a framework for planning the composition and monitoring of BPEL4WS web services. As far as we know, there is no previous attempt to automatically plan for the composition and monitoring of service oriented processes that takes into account nondeterminism, partial observability, and extended goals.

The problem of simulation, verification, and automated composition of web services has been tackled in the semantic web framework, mainly based on the DAML-S ontology for describing the capabilities of web services (Ankolekar 2002). In (Narayanan & McIlraith 2002), the authors propose an approach to the simulation, verification, and automated composition of web services based a translation of DAML-S to situation calculus and Petri Nets, so that it is possible to reason about, analyze, prove properties of, and automatically compose web services. However, as far as we understand, in this framework, the automated composition is limited to sequential composition of atomic services for reachability goals, and do not consider the general case of possible interleavings among processes and of extended business goals. Moreover, Petri Nets are a rather expressive framework, but algorithms that analyze them have less chances to scale up to complex problems compared to symbolic model checking techniques.

Different planning approaches have been proposed for the composition of web services, from HTNs (Wu *et al.* 2003) to regression planning based on extensions

	With shipper nack					Without shipper nack	
	Generate	BuilMonitor	BuildDomain	Planning	Result	Planning	Result
CASE 1	1 sec.	1 sec.	5 sec.	1 sec.	YES	0 sec.	NO
CASE 2	2 sec.	3 sec.	11 sec.	19 sec.	YES	2 sec.	NO
CASE 3	44 sec.	245 sec.	1616 sec.	392 sec.	YES	26 sec.	NO
CASE 4	61 sec.	997 sec.	13215 sec.	3321 sec.	YES	112 sec.	NO

Figure 4: Results of the experiments.

of PDDL (Dermott 1998), but how to deal with non-determinism, partial observability, and how to generate conditional and iterative behaviors (in the style of BPEL4WS) in these frameworks is still an open issue.

Other planning techniques have been applied to related but somehow orthogonal problems in the field of web services. The interactive composition of information gathering services has been tackled in (Thakkar, Knoblock, & Ambite 2003) by using CSP techniques. In (Lazovik, Aiello, & Papazoglou 2003) an interleaved approach of planning and execution is used; planning technique are exploited to provide viable plans for the execution of the composition, given a specific query of the user; if these plans turn out to violate some user constraints at run time, then a re planning task is started. Finally, works in the field of Data and Computational Grids is more and more moving toward the problem of composing complex workflows by means of planning and scheduling techniques (Blythe, Deelman, & Gil 2003).

The results reported in this paper are still preliminary, and significant extensions are planned for the future. The plan extraction routine within the EaGLe planning algorithm, and the powerset construction of the knowledge level planning domain, are currently far from optimized, and experiments show that they constitute a bottleneck. It is our intention in the short term to redesign them, so that they exploit the structure of the problem under exam, and of the resulting plan.

Moreover, the computationally complex powerset construction of the knowledge level domain can be avoided altogether by providing algorithms for natively planning with extended goals under partial observability. This is a is a main goal in our research line; a preliminary result appears in (Bertoli & Pistore 2004), focusing on the CTL goal language. The extension of this work to the EaGLe language, and the exploitation of symbolic representation techniques for this problem, is far from trivial.

The example considered in this paper and the experimental analysis is still limited to an explanatory and simple case. We intend to test our approach over realistic case studies stemming from concrete needs from the industry or the public apparatus. This will require, among other tasks to extend MBP to accept BPEL4WS code in input for the domain specification, and to return BPEL4WS plans in output.

References

Andrews, T.; Curbera, F.; Dolakia, H.; Goland, J.; Klein, J.; Leymann, F.; Liu, K.; Roller, D.; Smith, D.; Thatte, S.; Trickovic, I.; and Weeravarana, S. 2003. Business Process Execution Language for Web Services.

Ankolekar, A. 2002. DAML-S: Web Service Description for the Semantic Web. In *Proceedings of the 1st International Semantic Web Conference (ISWC 02)*.

Bertoli, P., and Pistore, M. 2004. Planning with Extended Goals and Partial Observability. In *Proceedings of ICAPS'04*. To be published.

Bertoli, P.; Cimatti, A.; Roveri, M.; and Traverso, P. 2001. Planning in Nondeterministic Domains under Partial Observability via Symbolic Model Checking. In Nebel, B., ed., *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001*, 473–478. Morgan Kaufmann Publishers.

Bertoli, P.; Cimatti, A.; Pistore, M.; and Traverso, P. 2003. A Framework for Planning with Extended Goals under Partial Observability. In *Proc. ICAPS'03*, 215–224.

Bertoli, P.; Cimatti, A.; and Roveri, M. 2001. Conditional Planning under Partial Observability as Heuristic-Symbolic Search in Belief Space. In *Proceedings of the Sixth European Conference on Planning (ECP'01)*.

Blythe, J.; Deelman, E.; and Gil, Y. 2003. Planning for Workflow Construction and Maintenance on the Grid. In *Proceedings of ICAPS'03 Workshop on Planning for Web Services*.

Bonet, B., and Geffner, H. 2000. Planning with Incomplete Information as Heuristic Search in Belief Space. In *Proc. AIPS 2000*.

Dal Lago, U.; Pistore, M.; and Traverso, P. 2002. Planning with a Language for Extended Goals. In *Proc. AAAI'02*.

Dermott, D. M. 1998. The Planning Domain Definition Language Manual. Technical Report 1165, Yale Computer Science University. CVC Report 98-003.

Emerson, E. A. 1990. Temporal and modal logic. In van Leeuwen, J., ed., *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier.

Lazovik A.; Aiello M.; and Papazoglou M. 2003. Planning and Monitoring the Execution of Web Service Requests. In *Proceedings of the First International Conference on Service-Oriented Computing (ICSOC'03)*.

Narayanan, S., and McIlraith, S. 2002. Simulation, Verification and Automated Composition of Web Services. In *Proceedings of the Eleventh International World Wide Web Conference (WWW-11)*.

Pistore, M., and Traverso, P. 2001. Planning as Model Checking for Extended Goals in Non-deterministic Domains. In *Proc. IJCAI'01*.

Pistore, M.; Bettin, R.; and Traverso, P. 2001. Symbolic Techniques for Planning with Extended Goals in Non-Deterministic Domains. In *Proc. ECP'01*.

Thakkar, S.; Knoblock, C.; and Ambite, J. L. 2003. A View Integration Approach to Dynamic Composition of Web Services. In *Proceedings of ICAPS'03 Workshop on Planning for Web Services*.

Wu, D.; Parsia, B.; Sirin, E.; Hendler, J.; and Nau, D. 2003. Automating DAML-S Web Services Somposition using SHOP2. In *Proceedings of the Second International Semantic Web Conference (ISWC2003)*.