ASTRO: Supporting Composition and Execution of Web Services*

M.Trainotti M.Pistore

University of Trento
Via Sommarive 14
38050 Povo (Trento) - Italy

G.Calabrese G.Zacco G.Lucchese
F.Barbon P.Bertoli P.Traverso

ITC-Irst
Via Sommarive 18
38050 Povo (Trento) - Italy

Abstract

Web services are rapidly emerging as the reference paradigm for the interaction and coordination of distributed business processes. In several research papers we have shown how advanced automated planning techniques can be exploited to automatically compose web services, and to synthesize monitoring components that control their execution. In this demo we show how these techniques have been implemented in the ASTRO toolset (http://www.astroproject.org), a set of tools that extend existing platforms for web service design and execution with automated composition and execution monitoring functionalities.

Introduction

Web services are rapidly emerging as the reference paradigm for the interaction and coordination of distributed business processes. 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.

In previous works (1; 2; 3), we have shown how automated planning techniques based on the "Planning via Model Checking" paradigm can effectively support these functionalities. More precisely, the algorithms proposed in (1; 2; 3) are based on web service specifications described in BPEL4WS, a standard language that can be used both for describing existing web services in terms of their interfaces (i.e., of the operations that are needed to interact with them) and for defining the executable code that implements composite services.

Automated web service composition starts from the description of a number of protocols defining available external services (expressed as BPEL4WS specifications), and a "business requirement" for a new composed process (i.e., the goal that should be satisfied by the new service, expressed in a proper goal language). Given this, the planner must synthesize automatically the code that implements the internal process that, exploiting the services of the external partners, achieves the business requirement. This code is then emitted as executable BPEL4WS code.

The automated synthesis techniques provided by the "Planning via Model Checking" framework can be also exploited to generate process monitors, i.e., pieces of code that detect and signal whether the external partners behave consistently with the specified protocols. This is vital to detect unpredictable run-time misbehaviors (such as those that may originate by dynamic modifications of the partners' protocols), or other events in the executions of the web services that need to be reported and analyzed.

Notice that these problems require to deal with nondeterminism (since the behavior of external services cannot be foreseen a priori), partial observability (since their status is opaque to the composed service), and extended goals (since realistic business requirements specify complex expected behaviors rather than just final states). By tackling the problem of composing and monitoring web services, we have shown the capabilities of the "Planning via Model Checking" approach in realizing such a complex planning task.

In this demo we show how these techniques can extend existing commercial platforms for web service design and execution. More precisely, we describe the ASTRO toolset (http://www.astroproject.org), which implements automated composition and monitor generation functionalities as extensions of the Active WebFlow platform. Active WebFlow (http://www.activebpel.org/) is a commercial tool for designing and developing BPEL4WS processes which is based on the Eclipse platform. It also provides an open-source BPEL4WS execution engine, called Active BPEL. By implementing automated composition and monitoring within Active WebFlow, these advanced functionalities can be combined with the other "standard" functionalities provided by the platform (such as inspecting BPEL4WS code, writing or modifying business processes, deploying these processes and executing them) and become integral part of the life cycle of business process design and execution.

The rest of the paper is structured as follows. We start with the description of a service composition scenario which is used to illustrate the proposed approach.

*This work is partially funded by the MIUR-FIRB project RBNE0195K5, "Knowledge Level Automated Software Engineering", and by the MIUR-PRIN 2004 project "Advanced Artificial Intelligence Systems for Web Services".

Then we describe the architecture and the functionalities of the ASTRO toolset. Finally, we present a demonstration of the application of this toolset to the reference composition scenario.

A service composition scenario

The demo is based on a classical web service composition problem, namely that of the Virtual Travel Agency (VTA). It consists in providing a combined flight and hotel booking service by composing two separate, independent existing services: a Flight booking service, and a Hotel booking service.

The Hotel booking service becomes active upon a request for a room in a given location (e.g., Paris) for a given period of time. In the case the booking is not possible (i.e., there are no available rooms), this is signaled to the request applicant, and the protocol terminates with failure. Otherwise, the applicant is notified with information about the hotel (e.g., Hilton), cost of the room, etc. and the protocol stops waiting for either a positive or negative acknowledgment. In the first case, an agreement has been reached and the room is booked. In the latter case, the interaction terminates with failure.

The protocol provided by the Flight booking service is similar. It starts upon a request for flights that guarantee to stay in a given location (e.g., Paris) for a given period of time. This might not be possible, in which case the applicant is notified, and the protocol terminates failing. Otherwise, information on the flights (carrier, cost, schedule...) are computed and returned to the applicant. The protocol suspends for either a positive or negative acknowledgment, terminating (with success or failure resp.) upon its reception.

The expected protocol that the user will execute when interacting with the VTA goes as follows. The user sends a request to stay in a given location during a given period of time, and expects either a negative answer if this is not possible (in which case the protocol terminates, failing), or an offer indicating hotel, flights and cost of the trip. At this time, the user may either accept or refuse the offer, terminating its interaction in both cases.

Of course several different interaction sequences are possible with these services; e.g., in a *nominal* scenario, none of the services answers negatively to a request; in non-nominal scenarios, unavailability of suitable flights or rooms, as well as user refusals, may make it impossible to reach an agreement for the trip. Taking this into account, the business requirement for the composed service is composed of two subgoals. The "nominal" subgoal consists in reaching the agreement on flights and room. This includes enforcing that the data communicated to the various processes are mutually consistent; e.g., the number of nights booked in the hotel depends on the schedule of the selected flights. The "recovery" subgoal consists in ensuring that every partner has rolled back from previous pending requests, and must be only pursued when the nominal subgoal cannot be achieved anymore.

By automated composition of the VTA process, we mean the automated generation of the code that has to be executed on the VTA server, so that requests from the user are answered combining the Flight and Hotel services in a suitable way. This composition has to implement the two sub-goals described above.

After the VTA process has been generated, its executions must be monitored, in order to detect problems in the interactions with the other partners participating to the scenario. Properties to be monitored include "correctness" checks (e.g., the partners obey the declared protocols; the flight schedules are compatible with the requests...). It is also possible to monitor "business" properties, such as the fact that, when an offer for a trip is sent to the user, this offer gets accepted or not.

The ASTRO toolset

This section presents a general overview of the ASTRO toolset. At the current stage, it consists of the following tools: WS-gen, WS-mon, WS-console and WS-animator.

WS-gen is responsible for generating the automated composition. It consists in a back-end layer and a front-end layer. The back-end layer takes as input the BPEL4WS specifications of the interaction protocols that the composite service has to implement, a "choreographic" file describing the connections between the composition's partners, and a goal file defining the composition requirement. It consists of two applications (see Fig.1): BPELTranslator converts the BPEL4WS specification files and the choreography file in an intermediate (.smv) file which is adequate for representing "Planning via Model Checking" problems; WSYNTH takes as input the problem domain, computes the plan which fulfills the requirements, and emits the plan in BPEL4WS format. The front-end (see Fig.2) is responsible for controlling the composition process and for managing the generated BPEL4WS specification; it has been implemented as an Eclipse plugin, and is hence integrated in the Active WebFlow environment.

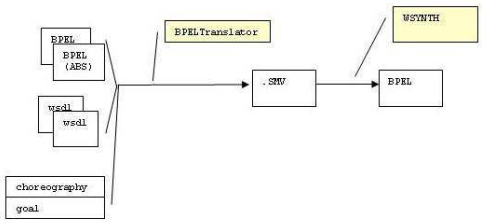


Fig. 1: WS-gen architecture

WS-mon is responsible for generats the Java code implements the monitors for the composed process and deploying them to the monitor framework. Similar to WS-gen, it consists in a back-end layer and a front-end

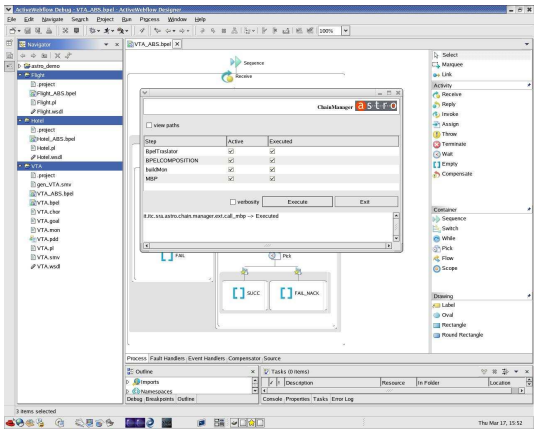


Fig. 2: WS-gen front end

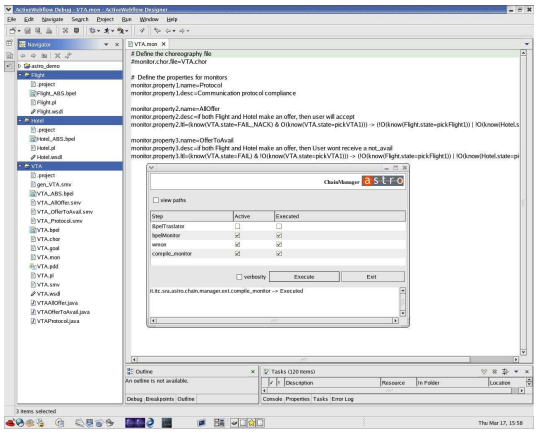


Fig. 4: WS-mon front end

layer. The back-end takes as input BPEL4WS specifications and a "choreographic" file, while the goal file is replaced by a file specifying the properties to by monitored. The back-end layer consists in three applications (see Fig.3): BPELTranslator, which is in common with WS-gen, converts the BPEL4WS specification files and the choreography file in a .smv file which describes the problem domain; WMON takes as input the problem domain, computes the plan which fulfills the monitoring requirements, and emits this plan in Java format; and the DEPLOYER compiles the Java class and deploy them to the monitor framework. The front-end (see

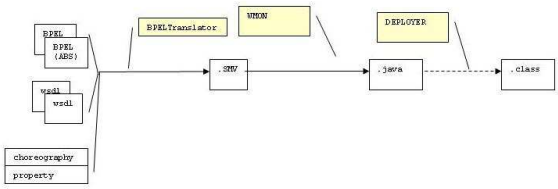


Fig. 3: WS-mon architecture

Fig.4), which is responsible for controlling the monitor generation process, has been implemented as an Eclipse plugin, and is hence integrated in the Active WebFlow environment.

The run-time monitor framework is responsible for executing the monitors associated to a given process every time an instance of that process is executed. It is also responsible for reporting the status of these monitors to the user in a convenient way. It consists of a back-end layer and a front-end layer (see Fig.5). The back-end layer has been implemented as an extension of the Active BPEL execution engine; the main goal is to sniff the input/output messages directed to the process that has to be monitored and to forward them to the Java monitors instances. The front-end implementation, WS-console, extends the Active BPEL administration console in order to present the status of the

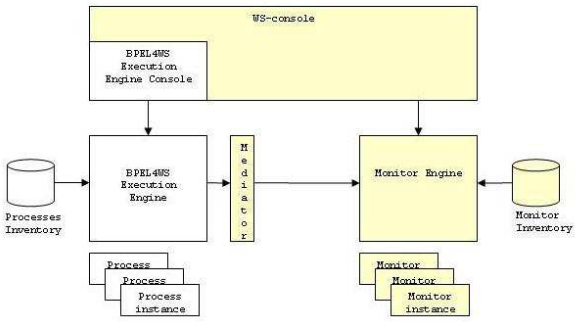


Fig. 5: Monitor framework architecture

monitors associated with each process instance. In this way, violations of the monitored properties are easy to be checked by the user (see Fig.6).

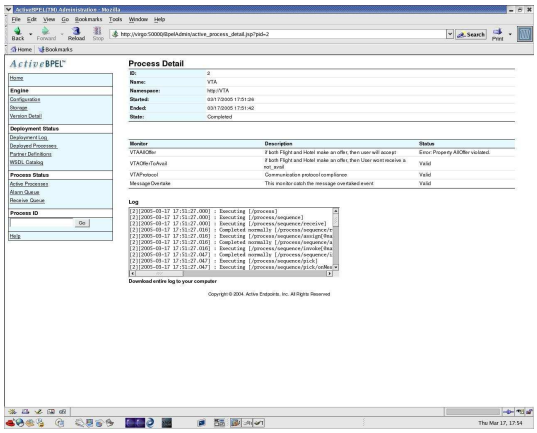


Fig. 6: WS-console

Finally, WS-animator (see Fig.7) is another Eclipse plugin, which gives the user the possibility to "execute" the composite process (in our case, the VTA). More precisely, it allows the user to play the roles of the actors

interacting with the composite process, while the Active WebFlow engine executes it.

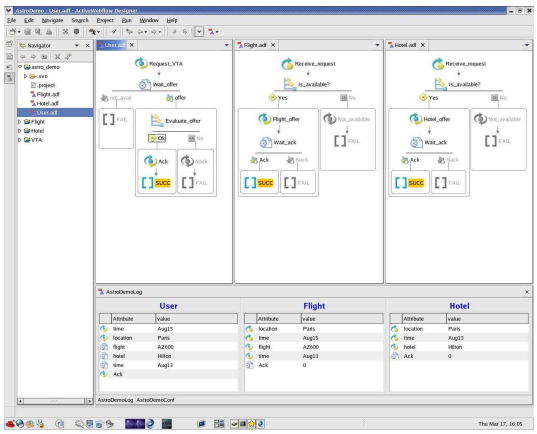


Fig. 7: WS-animator

The DEMO

In this section we describe a demonstration of the capabilities of the ASTRO toolset. The demo consists of a set of steps corresponding to the execution of service composition and a monitor synthesis task (see Fig.8).

Step 1. Within Active WebFlow, the user selects the projects (Flight, Hotel, VTA) which are part of the composition scenario. These projects contain the WSDL and the abstract BPEL4WS files describing the interfaces of the existing web services (and the protocol that the VTA has to expose to the end user). Moreover, the VTA project contains the .goal, .mon and .chor configuration files; those files define the requirements and choreography for the process and monitor composition.

Step 2. WS-gen is invoked. After the generation is terminated, the left panel gives a glimpse of the generated files; in particular, the composed process VTA.bpel is ready to be deployed to the BPEL4WS execution engine.

Step 3. The composed process is deployed into the Active BPEL execution engine via the Active WebFlow console; now the composed process is ready to receive the client requests.

Step 4. After the composition and deployment of the BPEL4WS process, WS-mon is used to generate the associated monitors. The left panel gives a glimpse of the generated files, and in particular the Java files implementing the monitor processes.

Step 5. To test the generated service, User, Flight and Hotel processes are executed in WS-animator, while the composed process is executed in Active BPEL execution engine. This configuration gives the possibility to test the composed process controlling the execution of the partner processes.

Step 6. After the execution of a nominal scenario within WS-animator, all the services end in a SUCCESS state. In this scenario, the user has request an offer to the composite service for a flight and a hotel specifying a date and a location. The Flight has received the flight request from the composite service, checked for its availability and sent back the flight number and date. The Hotel has received the request for an hotel reservation for a date (the one sent by the Flight) from the composite service, checked for its availability and sent back the hotel. The user has received the offer from the composite service and has accepted it.

Step 7. WS-console presents the states of the monitors for the instance of the VTA service corresponding to the nominal scenario presented above. All the monitor instances are valid.

Step 8. After the execution of a scenario where the User refuses a travel offer, WS-animator shows all the services terminated in a FAIL state. In this scenario, the user has requested an offer to the composite service for a flight and a hotel specifying a date and a location. The Flight has received the flight request from the composite service, checked for its availability and sent back the flight number and date. The Hotel has received the request for an hotel reservation for a date (the one sent by the Flight) from the composite service, checked for its availability and sent back the hotel. The user has received the offer from the composite service and has denied it. The denial has been forwarded to Flight and Hotel.

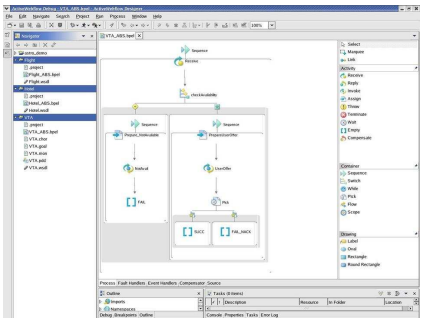
Step 9. WS-console presents the states of the monitors for the instance of the VTA service of the scenario where the user refuses the travel offer. This scenario violates one of the monitored properties, namely "if both Flight and Hotel make an offer, the user will accept it". This violation is reported in WS-console.

References

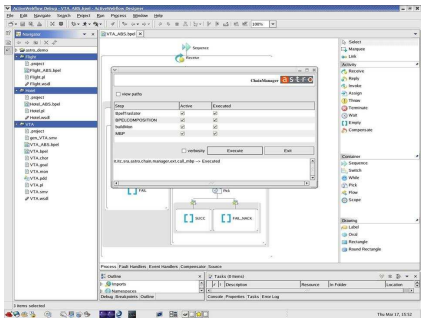
[1] Pistore, M.; Barbon, F.; Bertoli, P.; Shaparau. D.; and Traverso, P. 2004. Planning and Monitoring Web Service Composition. In *Proc. AIMSA'04*.

[2] Pistore, M.; Traverso, P.; and Bertoli, P. 2005. Automated Composition of Web Services by Planning in Asyncronous Domains. In *Proc. ICAPS'05*.

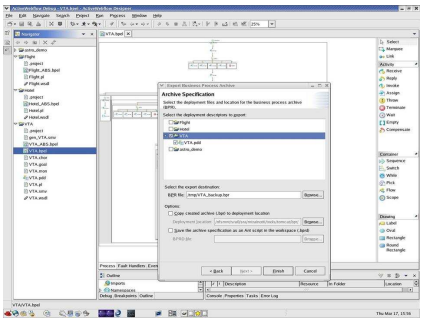
[3] Pistore, M.; Marconi, A.; Bertoli, P.; and Traverso, P. 2005. Automated Composition of Web Services by Planning at the Knowledge Level. In *Proc. IJCAI'05*.



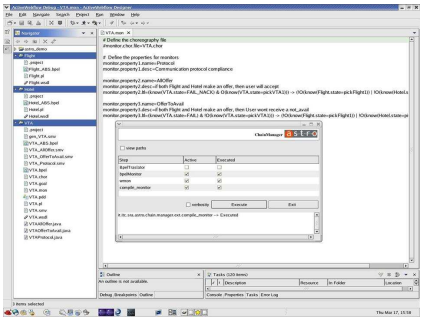
Step 1



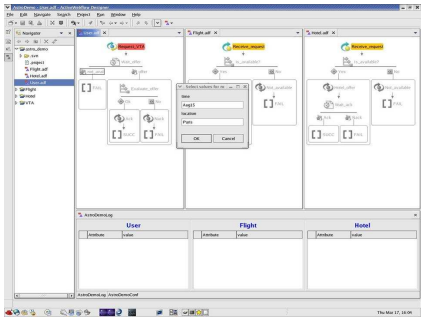
Step 2



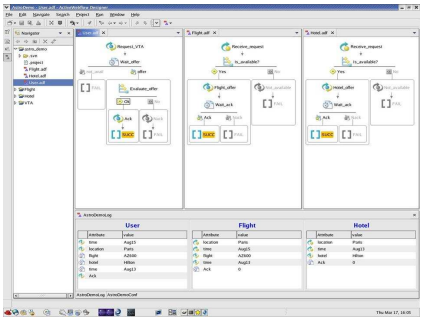
Step 3



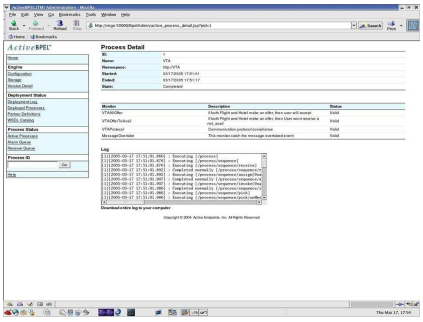
Step 4



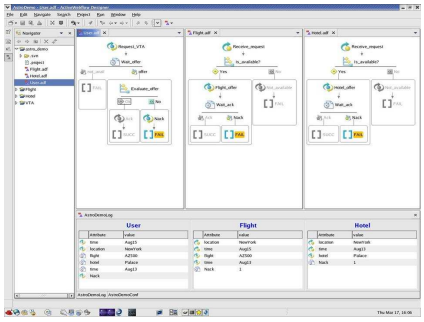
Step 5



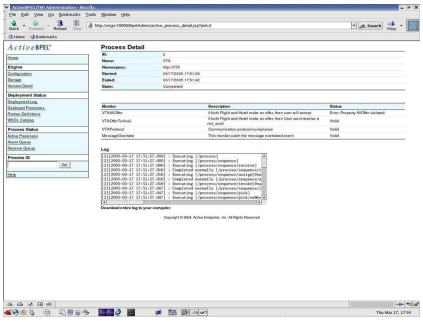
Step 6



Step 7



Step 8



Step 9

Fig. 8: The demo