11

Assumption-Based Composition and Monitoring of Web Services

Marco Pistore and Paolo Traverso

ITC-IRST
Via Sommarive 18, Povo, 38050 Trento, Italy
[pistore,traverso]@itc.it

Abstract. We propose an approach to the automated synthesis and the run-time monitoring of web service compositions. Automated synthesis, given a set of existing component services that are modeled in the BPEL language, and given a composition requirement, generates a new BPEL process that, once deployed, interacts with the components to satisfy the requirement. The composition requirement expresses assumptions under which component services are supposed to participate in the composition, as well as conditions that the composition is expected to guarantee. Run-time monitoring matches the actual behaviors of the service compositions against the assumptions expressed in the composition requirement, and reports violations. We describe the implementation of the proposed approach, which exploits efficient synthesis techniques, and discuss its scalability and practical applicability.

11.1 Introduction

Service composition is one of the key ideas underlying web services and service-oriented applications: composed services perform new functionalities by interacting with component services that are available on the web. The automated synthesis of composed services is one of the key tasks that can significantly support the design and development of service-oriented applications. Given a set of component services and a description of the interaction protocols that one has to follow in order to exploit them (e.g., in BPEL [3]), and given a composition requirement, the problem is to automatically synthesize a composed service that, once deployed, interacts with the components to satisfy the requirements.

Beyond composition requirements and component descriptions, in real life scenarios, key elements of the problem are the so-called *choreographic assumptions*, i.e., assumptions under which the component services are supposed to participate in the composition. Such assumptions may not be necessarily encoded in the descriptions of each component service. For instance, an assumption between an on-line shop and an electronic payment service can represent the agreement that the bank will always accept to process a request for a

money transfer, even if the interaction protocol may allow for a refusal of the request from the bank. Similarly, a reasonable agreement may be that it is always possible to cancel an order before paying for it, even if the interface description of the service may allow for a refusal of cancellations even before paying. Together with the description of the interactions with the available components, the choreographic assumptions constitute, therefore, the environment in which the composed service has to operate. As a consequence, the automated synthesis task should take them into account.

Beyond influencing the synthesis at design time, assumptions should also be monitored at run-time. Run-time monitoring should match the actual behaviors of service compositions against the assumptions, and report violations. This is a compelling requirement in service-oriented applications, which are most often developed by composing services that are made available by third parties, and which are autonomously developed and managed. Moreover, monitors are also needed to detect those problems that can emerge only at run-time. This is the case, for instance, of situations that, even if admissible in general at design time, must be promptly revealed when they happen, e.g., the fact that a bank refuses to transfer money to a partner on-line shop, even if this was part of an agreement.

While there have been several works on the automated synthesis of web services, see, e.g., [33, 27, 12, 41, 37], and several works on monitoring web services, see, e.g., [5, 30], much less emphasis has been devoted to the problem of the "assumption-based synthesis and monitoring of web services," i,e., to the problem of automatically generating composed services by possibly taking into account assumptions at design time, which are then monitored at run-time.

In this chapter, we address this problem: given a formal composition requirement, given a set of component service descriptions in BPEL, and given a set of choreographic assumptions expressed in temporal logic, we synthesize automatically an executable BPEL process that, once deployed, satisfies the composition requirement, as well as a set of Java monitors that report at run-time possible assumption violations. The automated generation of the composed BPEL process takes into account the choreographic assumptions during the synthesis, by discarding behaviors that violate them during the search for a solution. We synthesize a composed service that is not supposed to work and to satisfy the requirements in the case some assumptions are violated. A first advantage of this assumption-based synthesis is that the search for a solution may be simpler and scale up to more complex problems than the previous approach, since assumptions can be used to prune the search space (see, e.g., [2]). But, more important, this approach is mandatory in the case the composition only exists under the choreographic assumptions. This means that the assumptions are so crucial that, if they are violated, the composition does not make sense. In these cases, assumption-based synthesis and monitoring is the only viable solution.

The chapter is structured as follows. We start with a motivating example (Sect. 11.2) and with a conceptual architecture for automated composition and monitoring (Sect. 11.3). The next two sections describe our approach to assumption-based synthesis (Sect. 11.4) and monitoring (Sect. 11.5). We conclude with an evaluation of the approach (Sect. 11.6) and an analysis of related works (Sect. 11.7).

11.2 Motivating Example

As motivating example, we consider a virtual travel agency (VTA) that offers a transportation and accommodation service to clients, by interacting with three external services: one for booking flights, another for booking hotel rooms, and a third one for managing the payment (Fig. 11.1). When the agency receives a request from a client for a travel to a given location and period of time, it contacts the flight service. If available flights are found, the flight service returns an offer including the cost. Similarly, the agency contacts the hotel service and asks for a hotel room for the period of permanence in the desired location. If flight and hotel are both available, then the agency prepares and sends to the client an aggregated offer, which includes travel and accommodation, and sends it to the client. If the client decides to accept the offer, then she/he sends payment information (e.g., the credit card number) to the agency, which starts the payment procedure interacting with the payment service. Eventually, the flight and hotel services will receive a confirmation of the payment, and will emit electronic tickets that the agency will forward to the client.

In this chapter, we assume that BPEL [17] is used to describe the behavior of the involved services. BPEL provides an operational description of the (stateful) behavior of web services on top of the service interfaces defined in their WSDL specifications. A BPEL description identifies the partners of a service, its internal variables, and the operations that are triggered upon the invocation of the service by some of the partners. Operations include assigning variables, invoking other services and receiving responses, forking

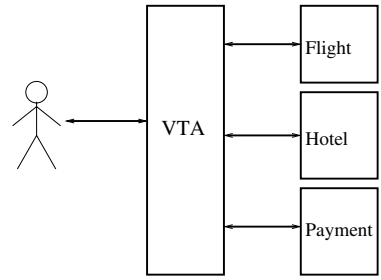

Fig. 11.1. The virtual travel agency example

parallel threads of execution, and non-deterministically picking one amongst different courses of actions. Standard imperative constructs such as if-then-else, case choices, and loops, are also supported.

BPEL can describe a web service at two different levels of abstraction. An *executable BPEL* model describes the actual behavior of a participant in a business process in terms of the internal activities and of the interactions undertaken with the partners. In an *abstract BPEL* model, instead, only the interface behavior of a service is described. That is, only the flow of messages exchanged with the other parties is defined, without revealing internal behaviors.

In this example, flight, hotel, and payment services are the external services. We assume, therefore, that their abstract BPEL specifications are available and can be downloaded from the web. These specifications describe the interaction protocols that the agency is expected to respect when interacting with the external services. In Fig. 11.2 we see the graphical descriptions of the BPEL processes for the flight, the hotel, and the payment service. We also assume that the protocol that the VTA follows in the interactions with the client is given as an abstract BPEL specification, which we also represent in Fig. 11.2. This protocol has to be considered an additional input to the composition, as it defines the interactions with the fourth partner of the VTA, namely its user. Notice that, in Fig. 11.2, transitions whose labels start with a "?" correspond to inputs received by the service, while labels starting with a "!" denote outputs of the service. Labels starting neither with "?" nor with "!" are *internal* actions of the protocol and correspond, for instance, to decisions or other private computations.

When automated web service composition techniques like those described in [41] are applied in this framework, the executable BPEL process of the VTA is generated automatically starting from the abstract BPEL specifications of the component services and from a composition requirement that specifies the goal of the composite process. In our example, the composition requirement specifies that the goal of the agency is to "sell holiday packages," i.e., to find a suitable flight and a suitable hotel for the client, and to manage the payment procedure. Achieving this goal means leading the interaction protocols with flight, hotel, payment service, and client to the successful final states (the states marked with SUCC in Fig. 11.2). However, there are cases in which the goal "sell holiday packages" cannot be achieved by the VTA: it might be that there are no flights or rooms available, the client may not accept the offer, the payment procedure may fail, etc. In all these cases, the VTA should at least guarantee that there are "no pending commitments" at the end of the execution: i.e., the VTA has to avoid the cases where a flight or a room are booked, if the client has not accepted or is not able to pay for them. If one of the interaction protocol fails, then we have to guarantee that all of them terminate in failure states (i.e., final states not marked with SUCC). The composition requirement for the VTA is hence something like "do whatever

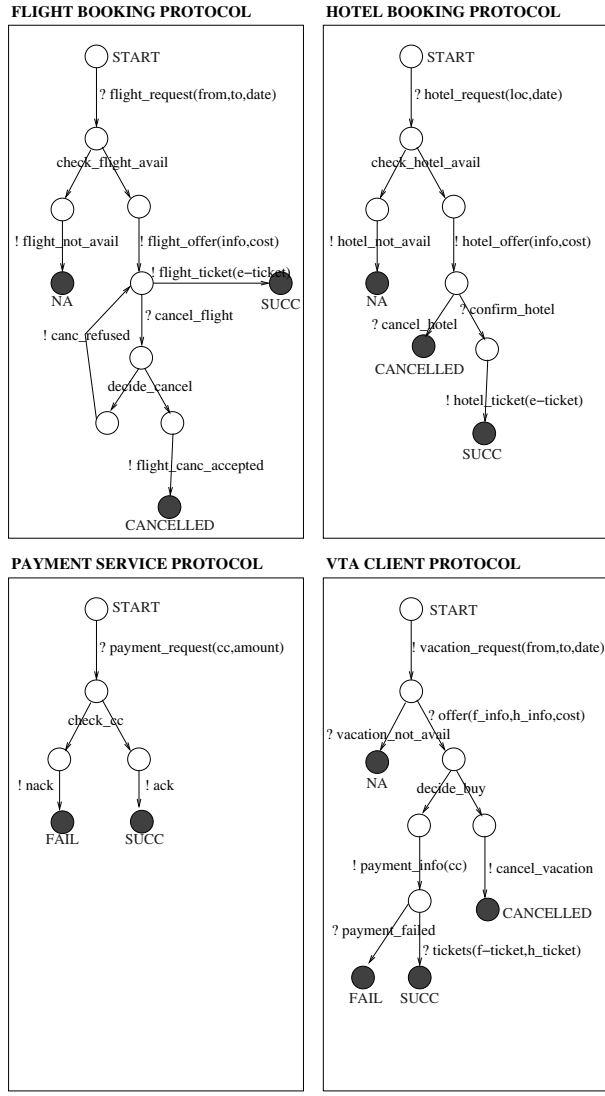

Fig. 11.2. Abstract BPEL protocols

possible to 'sell holiday packages,' but if something goes wrong guarantee that there are 'no pending commitments'."

In Fig. 11.3 we report a possible executable BPEL process that implements the VTA and satisfies the composition requirement just described. One can notice that the BPEL process behaves as an orchestrator that interacts with the flight, the hotel, the payment services, and the user according to the

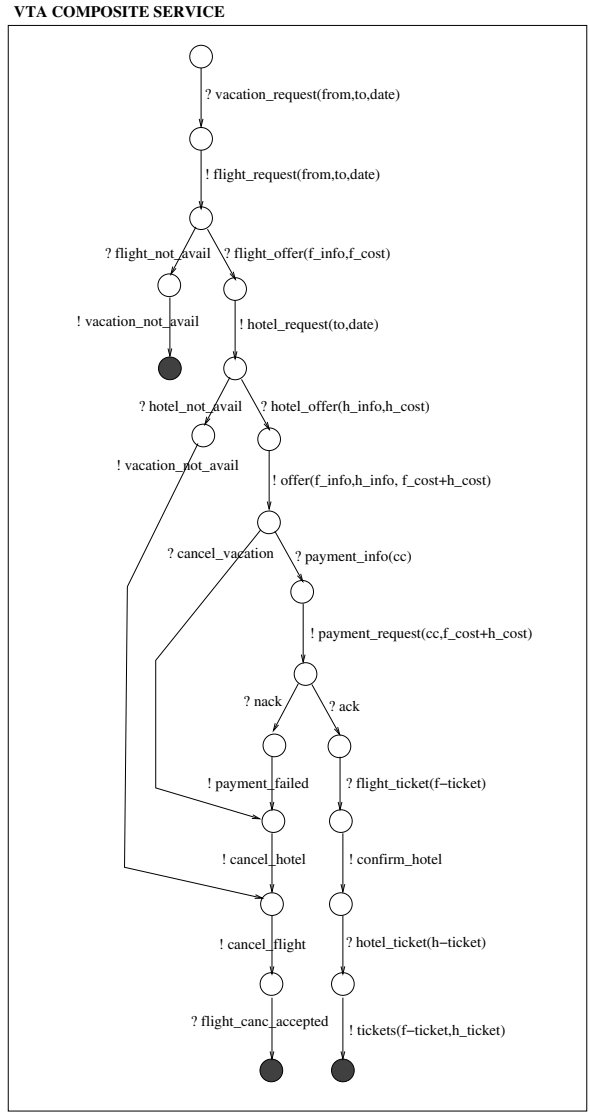

Fig. 11.3. Composite BPEL process for the VTA

protocols in Fig. 11.2, and directs and interleaves these interaction is a suitable way, in order to guarantee the achievement of the composition requirement.

In many cases, it is necessary or convenient to restrict the behaviors of the external services (hotel, flight, payment) with additional assumptions on their behaviors which are not implied by the abstract BPEL specifications. This is the case also for our scenario. Indeed, consider the behavior of the Flight service in case of a cancellation request: the BPEL specification simply

specifies that the request can be accepted or refused by the service. However, we may know, for instance from a service level agreement, that the flight cancellation is granted whenever the payment of the requested flight has not yet been done. We remark that the composition reported in Fig. 11.3 is based on this assumption: when a `cancel_flight` is issued by the VTA, the only expected outcome is an approval of the cancellation. More in general, without this assumption, it would not be possible for the VTA to achieve its composition requirement: indeed, if a flight offer has been obtained, but the user is not interested in the package returned by the VTA, then it would be impossible for the VTA to guarantee the possibility of canceling it, as requested by the "avoid pending commitments" requirement.

Besides being used to restrict the possible behaviors of the component services, our framework exploits these assumptions at run-time. *Assumption monitors*, which can be automatically generated from the specification of the assumptions, are executed in parallel with the composed BPEL process so as to check if the assumptions are respected during execution. Indeed, if the flight service violates our assumption, and refuses to cancel a flight even before the payment, the violation has to be detected and reported, since it will prevent the VTA from achieving its goal.

Monitoring is not limited to those assumptions that we use to restrict the valid behaviors at composition time. Indeed, it may be useful to have monitors also for additional assumptions that we did not exploit for generating the composition. Consider, for instance, the assumptions that rooms are guaranteed to be available if the request is done sufficiently in advance, or that flight availability is guaranteed for VIP clients, or also that the payment procedure will always succeed for "gold" credit cards, etc. These assumptions do not need to be exploited to obtain a correct composition. Still, it is important for the VTA to monitor them, since their violation may lead to loose clients.

Finally, an implicit assumption of the VTA on the component processes is that they respect the flow of interactions described in their abstract BPEL specification. This violation can happen, for instance, due to evolutions in the implementations of the external services, or also due to malicious external parties. In our framework, *domain monitors*, which detect violations of the specified protocols in the actual interactions with external services, can be automatically generated from the abstract BPEL specifications.

11.3 The Framework

Figure 11.4 depicts the design-time and run-time environments in our framework.

11.3.1 Design-Time Environment

The *Design-Time Environment* has two main components, a Composer and a Monitor Generator. The *Composer* can be used to automatically generate

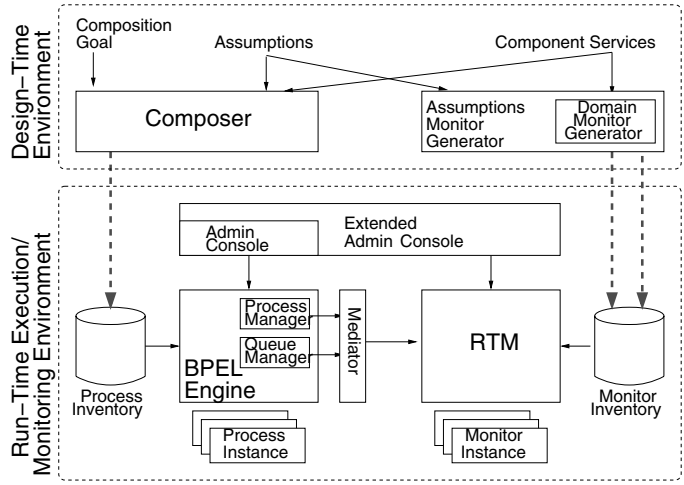

Fig. 11.4. Design-time and run-time execution/monitoring environments

the executable BPEL processes implementing the composed service. It takes in as input the component services and a composition goal. The component services are abstract BPEL specifications that are available on the web, and they can be seen as the environment the composed service has to interact with. The composition goal specifies requirements on the composed service. The composer can also take advantage of assumptions about the behavior of component services, in order to prune the search for a composed service. The composition task performed by the Composer is further analyzed in Sect. 11.4.

The second component of the Design-Time Environment is the *Monitor Generator*, which is composed of a core component, the *Domain Monitor Generator*, and the *Assumption Monitor Generator*, which is built upon the generator of domain monitors. The algorithms run by these modules are described in detail in Sect. 11.5.

11.3.2 Run-Time Environment

The *Run-Time Execution/Monitoring Environment* runs in parallel executable BPEL processes (for instance, the composite services generated at design time) and Java monitors (also possibly generated by the monitor generator). In our approach, monitors observe BPEL process behaviors by intercepting the input/output messages that are received/sent by the processes, and signal some misbehavior or, more in general, some situation or event of interest. In Fig. 11.4, the components on the left-hand side constitute the BPEL process execution environment, while the monitor run-time environment consists of the components on the right-hand side. For the *BPEL process execution environment*, we have chosen a standard engine for executing BPEL processes.

Among the existing BPEL engines, we chose Active BPEL [1] for our experiments, since it is available as open source, and since it implements a modular architecture that is easy to extend. From a high-level point of view, the Active BPEL run-time environment can be seen as composed of four parts. A *Process Inventory* contains all the BPEL processes deployed on the engine. A set of *Process Instances* consists of the instances of BPEL processes that are currently in execution. The *BPEL Engine* is the most complex part of the run-time environment, and consists of different modules (including the Process Manager, the Queue Manager, the Process Logger, and the Alarm Manager), which are responsible for the different aspects of the execution of the BPEL processes. The Process Manager creates and terminates process instances, and the Queue Manager is responsible for dispatching incoming and outgoing messages. The *Admin Console* provides web pages for checking and controlling the status of the engine and of the process instances.

The *Run-Time Monitoring Environment* is composed of four parts (see Fig. 11.4). The *Monitor Inventory* and the *Monitor Instances* are the counterparts of the corresponding components of the BPEL engine: the former contains all the monitor classes deployed in the engine, while the latter is the set of instances of these classes that are currently in execution. Each monitor class is associated to a specific BPEL process, while each monitor instance is associated to a specific process instance. Each monitor class is a Java class that implements the methods described in Fig. 11.5. The *Run-Time Monitor (RTM)* is responsible to support the life-cycle (creation and termination) and the evolution of the monitor instances. The *Mediator* allows the RTM to interact with the Queue Manager and the Process Logger of the BPEL engine and to intercept input/output messages as well as other relevant events such as the creation and termination of process instances. The *Extended Admin Console* is an extension of the Active BPEL Admin Console that presents,

- `init()`: init method, executed when an instance of the monitor is created
- `evolve(BpelMsg message)`: handles a message, updating the state of the monitor instance
- `terminate()`: handles the notification of a process termination event
- `isValid()`: returns true if the monitor instance is in a valid state (i.e., no misbehavior has been detected)
- `getErrorString()`: returns an error string if the monitor instance is in an invalid state
- `getProcessName()`: returns the name of the BPEL process associated to the monitor
- `getPropertyName()`: returns the (short) property name of the monitor
- `getPropertyDescription()`: returns the description of the property checked by the monitor

Fig. 11.5. Methods of a monitor Java class

along with other information on the BPEL processes, the information on the status of the corresponding monitors.

The monitor life-cycle is influenced by three relevant events: the process instance creation, the input/output of messages, and the termination of the process instance. When the RTM receives a message for the Mediator, it tries to find a match with the already instantiated monitors. If a match is found, the message is dispatched to all the matching monitor instances through method `evolve`. If no match is found, then a new process instance has been created in the BPEL engine, and hence a set of monitor instances specific for that process instance is created by the RTM and initialized through the method `init`. For each message, the Mediator provides also information on the process instance receiving/sending the message, as well as on the BPEL process corresponding to the instance. The information on the BPEL process is used to select the relevant set of monitors to be instantiated for that process. The process termination is captured via a *termination event*, the event is dispatched, through the invocation of method `terminate`, to all the monitor instances associated to the process instance.

11.4 Assumption-Based Composition of Web Services

In this section we describe the theory underlying the assumption-based Composer of Fig. 11.4 exploiting a general framework for the automated composition of web services.

11.4.1 An Automated Composition Framework

The work in [41] (see also [42, 37]) presents a formal framework for the automated composition of web services which is based on planning techniques: component services define the planning domain, composition requirements are formalized as a planning goal, and planning algorithms are used to generate the composite service. The framework of [41] differs from other planning frameworks since it assumes an asynchronous, message-based interaction between the domain (encoding the component services) and the plan (encoding the composite service). We now recall the most relevant features of the framework defined in [41].

The planning domain is modeled as a *state transition system* (STS from now on) which describes dynamic systems that can be in one of their possible *states* (some of which are marked as *initial states*) and can evolve to new states as a result of performing some *actions*. Actions are distinguished in *input actions*, which represent the reception of messages, *output actions*, which represent messages sent to external services, and a special action τ called *internal action*. The action τ is used to represent internal evolutions that are not visible to external services, i.e., the fact that the state of the system can evolve without producing any output, and independently from the reception of

inputs. A *transition relation* describes how the state can evolve on the basis of inputs, outputs, or of the internal action τ. Finally, a *labeling function* associates to each state the set of properties $\mathcal{P}rop$ that hold in the state. These properties will be used to define the composition requirements.

Definition 1 *[State transition system (STS)] A* state transition system Σ *is a tuple* $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L} \rangle$ *where*

- $\mathcal{S}$ *is the finite set of states*
- $\mathcal{S}^0 \subseteq \mathcal{S}$ *is the set of initial states*
- $\mathcal{I}$ *is the finite set of input actions*
- $\mathcal{O}$ *is the finite set of output actions*
- $\mathcal{R} \subseteq \mathcal{S} \times (\mathcal{I} \cup \mathcal{O} \cup \{\tau\}) \times \mathcal{S}$ *is the transition relation*
- $\mathcal{L} : \mathcal{S} \rightarrow 2^{\mathcal{P}rop}$ *is the labeling function.*

A state s *is said to be* final *if there is no transition starting from* s *(i.e.,* $\forall a \in (\mathcal{I} \cup \mathcal{O} \cup \{\tau\}), \forall s' \in \mathcal{S}.(s, a, s') \notin \mathcal{R}$*).*

The automated synthesis problem consists in generating a state transition system Σ_c that, once connected to Σ, satisfies the composition requirement ρ. We now recall the definition of the state transition system describing the behavior of Σ when connected to Σ_c.

Definition 2 (controlled system) *Let* $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L} \rangle$ *and* $\Sigma_c = \langle \mathcal{S}_c, \mathcal{S}_c^0, \mathcal{O}, \mathcal{I}, \mathcal{R}_c, \mathcal{L}_\emptyset \rangle$ *be two state transition systems, where* $\mathcal{L}_\emptyset(s_c) = \emptyset$ *for all* $s_c \in \mathcal{S}_c$*. The STS* $\Sigma_c \triangleright \Sigma$*, describing the behaviors of system* Σ *when controlled by* Σ_c*, is defined as*

$$\Sigma_c \triangleright \Sigma = \langle \mathcal{S}_c \times \mathcal{S}, \mathcal{S}_c^0 \times \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}_c \triangleright \mathcal{R}, \mathcal{L} \rangle$$

where

- $\langle (s_c, s), \tau, (s_c', s) \rangle \in (\mathcal{R}_c \triangleright \mathcal{R})$ *if* $\langle s_c, \tau, s_c' \rangle \in \mathcal{R}_c$
- $\langle (s_c, s), \tau, (s_c, s') \rangle \in (\mathcal{R}_c \triangleright \mathcal{R})$ *if* $\langle s, \tau, s' \rangle \in \mathcal{R}$
- $\langle (s_c, s), a, (s_c', s') \rangle \in (\mathcal{R}_c \triangleright \mathcal{R})$*, with* $a \neq \tau$*, if* $\langle s_c, a, s_c' \rangle \in \mathcal{R}_c$ *and* $\langle s, a, s' \rangle \in \mathcal{R}$*.*

Notice that we require that the inputs of Σ_c coincide with the outputs of Σ and vice versa. Notice also that, although the systems are connected so that the output of one is associated to the input of the other, the resulting transitions in $\mathcal{R}_c \triangleright \mathcal{R}$ are labeled by input/output actions. This allows us to distinguish the transitions that correspond to τ actions of Σ_c or Σ from those deriving from communications between Σ_c and Σ. Finally, notice that we assume that the plan has no labels associated to the states.

In an automated synthesis problem, we need to generate a Σ_c that guarantees the satisfaction of a composition requirement ρ. This is formalized by requiring that the controlled system $\Sigma_c \triangleright \Sigma$ must satisfy the goal ρ, written as $\Sigma_c \triangleright \Sigma \models \rho$. In [41], ρ is formalized using EAGLE, a requirement language which allows to specify conditions of different strengths (like "try" and "do"),

and preferences among different (e.g., primary and secondary) requirements. EAGLE operators are similar to CTL [24] operators, but their semantics, formally defined in [21], takes into account the notion of preference and the handling of failure when subgoals cannot be achieved.

For example, the EAGLE formalization of the composition requirement for the VTA example discussed in Sect. 11.2 is the following:

TryReach
  c.SUCC ∧ f.SUCC ∧ h.SUCC ∧ p.SUCC
Fail DoReach
  (c.NA ∨ c.FAIL ∨ c.CANCELLED) ∧
  (f.NA ∨ f.CANCELLED ∨ f.START) ∧
  (h.NA ∨ h.CANCELLED ∨ h.START) ∧
  (p.FAIL ∨ p.START)

Where c is the client, f the flight, h the hotel, and p the payment services and propositions like c.SUCC correspond to require that the client has reached the state marked with SUCC according to the interaction protocols in Fig. 11.2.[1] The goal is of the form "**TryReach** c **Fail DoReach** d." **TryReach** c requires a service that tries to reach condition c, in our case the condition "sell holiday packages." During the execution of the service, a state may be reached from which it is not possible to reach c, e.g., since the product is not available. When such a state is reached, the requirement **TryReach** c fails and the recovery condition **DoReach** d, in our case "no pending commitments" is considered.

The definition of whether ρ is satisfied, which we omit for lack of space, is defined on top of the executions that $\Sigma_c \triangleright \Sigma$ can perform. Given this, we can characterize formally an automated synthesis problem.

Definition 3 *[Automated Synthesis] Let Σ be a state transition system, and let ρ be an* EAGLE *formula defining a composition requirement. The automated synthesis problem for Σ and ρ is the problem of finding a state transition system Σ_c such that*

$$\Sigma_c \triangleright \Sigma \models \rho.$$

The work in [41] shows how to adapt to this task the "Planning as Model Checking" approach, which is able to deal with large non-deterministic domains and with requirements expressed in EAGLE. It exploits powerful BDD-based techniques [16] developed for Symbolic Model Checking [17] to efficiently explore domain Σ during the construction of Σ_c.

[1] Note that in the "no pending commitments" part of the composition goal we allow the flight, the hotel, and the payment services to "terminate" in the START state. This permits to skip calling some of the services (e.g., the payment service) in case of failures in previous services (e.g., no flight is available).

11.4.2 BPEL Processes and Assumptions as STSs

The domain for the composition task corresponds to the BPEL specifications of the component services *and* to the assumptions that we decide to enforce in the composition and that, as a consequence, restricts the valid behaviors of the BPEL components. We now show that BPEL processes and assumptions can all be mapped to STSs.

In [41], we have defined a translation that associates a state transition system to each component service, starting from its BPEL specification. We omit the formal definition of the translation, which can be found at http://www.astroproject.org.[2] Intuitively, input actions of the STS represent messages received from the component services, output actions are messages sent to the component services, internal actions model assignments and other operations which do not involve communications, and the transition relation models the evolution of the service.

For what concerns the assumptions, we allow the user to specify them in Linear Temporal Logic (LTL [24]).

Definition 4 (LTL) *Let $\mathcal{P}rop$ be a property set and $p \in \mathcal{P}rop$.* LTL *properties on $\mathcal{P}rop$ are defined as follows:*

$$\phi ::= \text{true} \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \mathrm{X}\,\phi \mid \mathrm{F}\,\phi \mid \mathrm{G}\,\phi \mid \phi_1 \ \mathrm{U} \ \phi_2 \mid \phi_1 \ \mathrm{W} \ \phi_2.$$

Intuitively, the temporal operators above can be read as follows:

- $\mathrm{X}\,\phi$ means "ϕ will be true in the next state."
- $\mathrm{F}\,\phi$ means "ϕ will be true eventually in the future."
- $\mathrm{G}\,\phi$ means "ϕ will be true for all the future states."
- $\phi_1 \mathrm{U}\,\phi_2$ means "ϕ_2 will be eventually true, and ϕ_1 will be true till that moment."
- $\phi_1 \mathrm{W}\,\phi_2$ means "ϕ_1 will be true till ϕ_2 becomes true or the history terminates."

In our context, the properties $p \in \mathcal{P}rop$ are atoms of the form s.q, where s is the name of one of the component services and q is either an input/output operation or one of the properties labeling the states of (STS modeling) s.

For example, the assumption that it is possible to cancel a flight until we start the payment process can be formalized as the following LTL formula:

[2] For the moment, the translation is restricted to a significant subset of the BPEL languages. More precisely, we support all BPEL *basic* and *structured activities*, like invoke, receive, sequence, switch, while, pick, and flow. Moreover, we support restricted forms of assignments and correlations. The considered subset does not deal at the moment with important BPEL constructs like scopes, fault, event, and compensation handlers; while these constructs are often required in *executable* BPEL implementation, we found the considered subset expressive enough for describing the *abstract* BPEL interface of complex services in real applications domains.

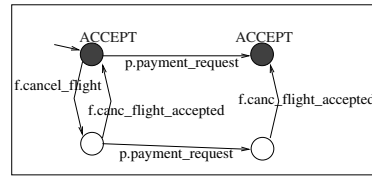

Fig. 11.6. Example of STS corresponding to an assumption

$$(\mathsf{f.cancel\_flight} \Rightarrow \mathrm{F}\, \mathsf{f.canc\_flight\_accepted})) \ \mathrm{W}\ \ \mathsf{p.payment\_request}.$$

It says that, until a payment request received by the payment service p (output payment_request in the protocol of Fig. 11.2), if a flight cancellation request is sent to the flight booking service f (message cancel_flight according to the interaction protocol of the flight), then an acknowledgement of the cancellation will eventually be received (message canc_flight_accepted). Alternatively, the same assumption can be written exploiting the labeling of the states of flight and payment service:

$$\mathrm{G}\,((\mathsf{f.cancel\_flight} \wedge \mathsf{p.START}) \Rightarrow \mathrm{F}\,(\mathsf{f.CANCELLED})).$$

It says that, if a flight cancellation request (message cancel_flight) is received when the payment procedure is still in its initial state (state labeled with START in the protocol of the payment service), then the state where the flight has been cancelled will eventually be reached (state labeled CANCELLED of the interaction protocol of the flight).

Standard techniques [24] can be used to translate the LTL specification of an assumption into an STS.[3] For instance, Fig. 11.6 reports the graphical description of the STS corresponding to property

$$(\mathsf{f.cancel\_flight} \Rightarrow \mathrm{F}\, \mathsf{f.canc\_flight\_accepted})) \ \mathrm{W}\ \ \mathsf{p.payment\_request}.$$

We remark that, despite the simplicity of this STS, its role is fundamental in order to guarantee the feasibility of the composition.

11.4.3 Generating the Composed BPEL Process

We are ready to show how we can perform assumption-based composition within the automated composition framework presented in Sect. 11.4.1. Given n component services $W_1, ..., W_n$ and m assumptions $A_1, ..., A_m$ that we want to enforce, we encode each component service W_i as a STS Σ_{W_i} and each assumption A_i as a STS Σ_{A_i}. The planning domain Σ for the automated

[3] Notice that we are interested in finite executions of the web services, hence we have to interpret the LTL assumptions on finite words. We can hence avoid the difficulties in modeling acceptance conditions that arise when interpreting LTL on infinite executions/words.

composition problem is the synchronized product of all these STSs. Formally, $\Sigma = \Sigma_{W_1} \parallel ... \parallel \Sigma_{W_n} \parallel \Sigma_{A_1} \parallel ... \parallel \Sigma_{A_m}$.

The planning goal is obtained from the formalization ρ of the composition termination requirements expressed in EAGLE. This formula has to be enriched to capture the fact that we require the conditions expressed in goal ρ to be satisfied only for those executions that satisfy the enforced assumptions, i.e., for those executions that terminate in a state where all the automata Σ_{A_i} are in accepting states (e.g., the states marked as ACCEPT in the STS of Fig. 11.6). Consider, for instance, the requirement $\rho =$ **TryReach** c **Fail DoReach** d, and let us assume that property a expresses the condition that all the automata Σ_{A_i} are in accepting states. Then the modified goal is

$$\rho_a = \textbf{TryReach } (a \Rightarrow c) \textbf{ Fail DoReach } (a \Rightarrow d).$$

Given the domain Σ and the planning goal ρ_c, we can apply the approach presented in [41] to generate a controller Σ_c, which is such that $\Sigma_c \triangleright \Sigma \models \rho_a$. Once the state transition system Σ_c has been generated, it is translated into the executable BPEL implementation of the composite service. This translation is conceptually simple, but particular care has been put in its implementation (see http://www.astroproject.org) in order to guarantee that the generated BPEL is of good quality, e.g., it is emitted as a structured program that can be inspected and modified if needed.

11.5 Automatic Generation of Monitors

In this section we describe how monitors can be automatically generated from the BPEL description of the component services and from the assumptions specifying the properties to be monitored. As discussed in Sect. 11.2, we distinguish two kinds of monitors: *domain monitors*, which are responsible to check whether the component services respect the protocols described in their abstract BPEL specification, and *assumption monitors*, which check whether the component services satisfy additional assumptions on their behavior.

11.5.1 Domain Monitors

Monitors can only observe messages that are exchanged among processes. As a consequence, they cannot know exactly the internal state reached by the evolution of a monitored external service. Non-observable behaviors of a service (such as assign activities occurring in its abstract BPEL) are modeled by τ-transitions, i.e., transitions from state to state that do not have any associated input/output. From the point of view of the monitor, this kind of evolutions of external services cannot be observed, and states involved in such transitions are indistinguishable. Such sets of states are called *belief*

```
procedure build-mon()
  MS = MT = MF = ∅
  ms_0 = {τ-closure(s_0) : s_0 ∈ S_0}
  build-mon-aux(ms_0)

procedure build-mon-aux(B:Belief)
  if B ∉ MS then
    MS = MS ∪ {B}
    if ∃s ∈ B. s is final then
      MF = MF ∪ {B}
    end if
    for all m ∈ (I ∪ O) do
      B' = Evolve(B, m)
      if B' ≠ ∅ then
        build-mon-aux(B')
        MT = MT ∪ {< B, m, B' >}
      end if
    end for
  end if
```

Fig. 11.7. The domain monitor generation algorithm

states, or simply *beliefs* [15]. We denote with τ-closure(s) the set of the states reachable from s through a sequence of τ-transitions. The evolution of an external service, as perceived by a monitor, is modeled by the evolution from belief states to belief states.

Definition 5 (belief evolution) *Let $B \subseteq \mathcal{S}$ be a belief on some STS $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L} \rangle$. We define the evolution of B on message $m \in (\mathcal{I} \cup \mathcal{O})$ as the belief $Evolve(B, m)$, where*

$$Evolve(B, m) = \{s' : \exists s \in B. \exists s'' \in \mathcal{S}. \langle s, m, s'' \rangle \in \mathcal{R} \wedge s' \in \tau\text{-}closure(s'')\}.$$

The generation of a domain monitor for an external abstract BPEL process is based on the idea of beliefs and belief evolutions. The domain monitor generation algorithm (Fig. 11.7) incrementally generates the set $\mathcal{MS}$ of beliefs starting from the initial belief ms_0, by grouping together indistinguishable states of the STS. The beliefs in $\mathcal{MS}$ are linked together with (non τ) transitions $\mathcal{MT} \subseteq \mathcal{MS} \times (\mathcal{I} \cup \mathcal{O}) \times \mathcal{MS}$, as described by function *Evolve*. Beliefs that contain at least one state that is final for the STS are considered possible final states also for the domain monitor, and are stored in $\mathcal{MF}$.

Once the algorithm in Fig. 11.7 has been executed, the Java code implementing the domain monitor can be easily generated. A skeleton of this Java code, parametric with respect to the set of beliefs $\mathcal{MS}$, initial and final beliefs ms_0 and $\mathcal{MF}$, and the belief transitions $\mathcal{MT}$, is reported in Fig. 11.8.

In the Java code, the belief states in $\mathcal{MS}$ are used to trace the current status of the evolution of the monitored BPEL process, using ms_0 as initial state

and the transitions in $\mathcal{MT}$ to let the status of the monitor evolve whenever a message is received. The final beliefs $\mathcal{MF}$ are exploited when a termination event is received: indeed, if the process instance terminates and the monitor is a belief that is not final, then a premature termination of the process instance has occurred.

We remark that one can interpret the algorithm in Fig. 11.7 as a transformation of the STS in input, which is non-deterministic and contains τ transitions, into a new STS that is deterministic and is fully observable (i.e., that does not contain τ transitions). Actually, the algorithm is an adaptation of the standard power-set construction for transforming non-deterministic finite automata into deterministic ones.

11.5.2 Assumption Monitors

The algorithm for the generation of assumption monitors takes as input the (STSs corresponding to the) abstract BPEL processes of the external services plus an assumption to be monitored. As already discussed in Sect. 11.4.2, we express assumptions in LTL [24], using as propositional atoms the input/output messages of the component services as well as the properties labeling the states of the STSs modeling these services.

To build an assumption monitor, the corresponding LTL formula is mapped onto an STS, which is then emitted as Java code.

The evolution of the assumption monitor depends on the input/output messages received by the composite services, which are directly observable by the monitor. However, it also depends on the evolution of the truth values of those basic propositions labeling the states of the components STSs which appear in the LTL formula. These truth values are computed by tracing the evolution of the beliefs of the component services relevant to the formula, similarly to what is described in Fig. 11.8 for the domain monitor. However, it is possible in this case to simplify the "domain" monitor, by pruning out parts of the protocol that are not relevant to tracing the evolution of the basic propositions which appear in the formula. This prune is obtained by applying a reduction algorithm inspired by the classical minimization algorithm for finite state automata (Fig. 11.9). This algorithm builds a partition Π of the belief states of the domain monitor, so that beliefs in the same class are considered equivalent for monitoring the basic propositions $P_1, \ldots, P_n$ we are interested in. The initial partition consists of different classes corresponding to different truth values of the basic propositions. This partition is then iteratively refined by splitting a class into two parts, until a fixed point is reached. The splitting of class C into the two classes $\text{split}(C, m, C')$ and $C \setminus \text{split}(C, m, C')$ is performed whenever there are some beliefs in C from which class C' is reached performing message m, while for other beliefs in C a class different from C' is reached performing x (see procedures "split" and "splittable" in Fig. 11.9). When a stable partition is reached, the reduced monitor is obtained by merging beliefs in the same class of the partition.

```
public class Monitor implements IMonitor {
    private enum MS { ... } // monitor states
    private MS _bs;   // current monitor state
    private boolean is_valid = true;
    public void init() { _bs = ms0; }
    public boolean isValid() {
        return is_valid;
    }
    private boolean isFinal() {
        return (_bs in MF);
    }
    public void terminate()
    {
        is_valid= is_valid && isFinal() ;
    }
    public String getErrorString()
    {
        if(!isValid()){
            return "Protocol violation";
        } else {
            return "No error";
        }
    }
    public void evolve(BpelMsg msg)
    {
        if (is_valid){
            if (exists <_bs,msg,next> in MT){
                _bs = next;
            } else {
                is_valid = false;
            }
        }
    }
    public Monitor() { init(); }
    public String getProcessName() { ... }
    public String getPropertyName() { ... }
    public String getPropertyDescription() { ... }
}
```

Fig. 11.8. Skeleton of the domain monitor

procedure reduce-monitor$(P_1, \ldots, P_n)$
  /* *Building the initial partition* */
  for all $PS \subseteq \{P_1, \ldots, P_n\}$ **do**
    $C = \{B \in \mathcal{MS} : \forall i = 1, \ldots, n.\ (B \models P_i \ \Leftrightarrow\ P_i \in PS)\}$
    if $C \neq \emptyset$ **then** $\Pi = \Pi \cup \{C\}$ **end if**
  end for
  /* *Refining the partition* */
  while $\exists C, C' \in \Pi.\ C \neq C' \wedge \exists m \in (\mathcal{I} \cup \mathcal{O}).\ \text{splittable}(C, m, C')$ **do**
    $\Pi = \Pi \setminus \{C\} \cup \{\text{split}(C, m, C'), C \setminus \text{split}(C, m, C')\}$
  end while
  return Π

procedure split(C, m, C')
  return $\{B \in C : \exists B' \in C'.\ < B, m, B' > \in \mathcal{MT}\}$

procedure splittable(C, m, C')
  return $\emptyset \neq \text{split}(C, m, C') \neq C$

Fig. 11.9. The assumption monitor reduction algorithm

11.6 Experimental Evaluation

The performance of the automated composition task have been tested experimentally, see, e.g., [41, 37, 42] in the case without assumptions. We have also used the automated composition techniques on some real applications in the field of e-government, telcos, and on-line banking. The experiments and the applications have shown the feasibility and the scalability of the approach. We have shown that the automated composition task takes a rather low amount of time, and it is surely much faster than manual development of executable BPEL composite processes. Moreover, the automatically generated BPEL is of good quality. In some cases, we have asked experienced programmers to develop manually the BPEL processes and we have compared the automatically generated and the hand-written solutions. We have discovered that the solutions often implement the same strategy and have a similar structure. The main differences are mainly due to possible different styles of programming, and the automatically generated code is reasonable and rather easy to read and understand.

In this chapter, we report the performance of the automated construction of monitors, as well as the overhead caused by the execution of monitors at run-time. All experiments have been run on a 3 GHz Pentium 4 PC machine, equipped with 1 GB memory, and running a Linux 2.6.7 operating system.

In order to test the performance of the monitor generation, we have performed two sets of experiments. In the first set, we test the automatic generation of domain monitors w.r.t. the complexity of the planning domain. We report the results of the automatic generation w.r.t. the number of activities of an abstract BPEL process in input to the monitor generation (Fig. 11.10). The input BPEL is a generalized version of the hotel service which can perform different kinds of reservations one after the other, thus increasing its number of activities. We start the experiments from six activities, corresponding to the

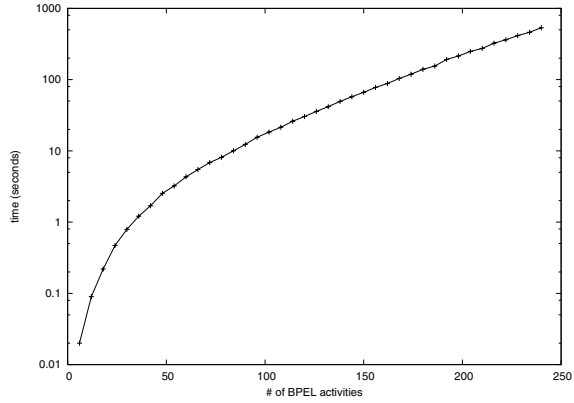

Fig. 11.10. Experiments with domain monitor generation

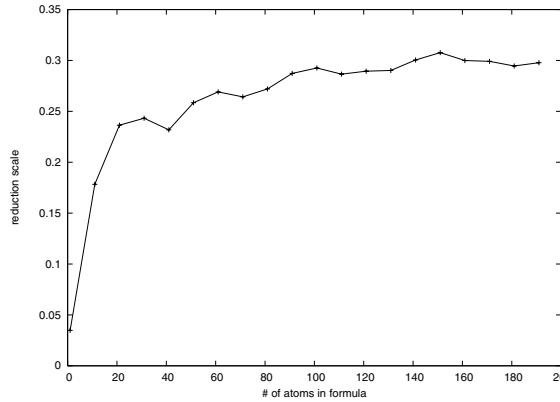

Fig. 11.11. Experiments with assumption monitor reduction

activities of a very simple hotel service, and we scale up to 240, corresponding to a service dealing with about 40 different kinds of reservations. On the vertical axis, we report the monitor generation time in seconds. As expected, the time for monitor construction increases regularly with the number of activities. The monitor generation however manages to deal with rather complex BPEL specifications in a rather short time. The case of 100 activities takes 10 seconds, and we manage to automatically generate monitors for BPEL specifications with about 250 activities in 1000 seconds. In our example, BPEL with 100 activities generate a monitor with more than 300 beliefs, while 250 activities correspond to about 1000 beliefs.

In Fig. 11.11 we report the results of our second set of experiments. Given a service, we test the monitor reduction algorithm performance. In the horizontal axis, we have the number of propositions in a set of randomly generated assumption formulas of increasing complexity. The number of atoms in the formula is indeed the parameter that can affect monitor reduction. In the vertical axis, we report the average gain ratio in number of beliefs obtained by performing the monitor reduction (a value of 0.25 means that the size of the reduced automaton is 25% of the original). The reduction is significant. Notice also that, as the number of atomic propositions in the formula grows, the gain ratio stabilizes somewhere around 0.30. This corresponds to the fact that about 70% of the states in the monitor are useful only for protocol monitoring, but do not give any information for the monitoring of the specific formula.

Figure 11.12 reports instead an experimental evaluation of the overhead that can be caused by executing monitors in parallel to BPEL processes. We measure the overhead at increasing number of monitors that check at run-time a process: the number of monitors per process is reported on the horizontal axis. The overhead is the time to run the processes without any monitor divided by the time to run the processes with their monitors. Fig. 11.12 reports

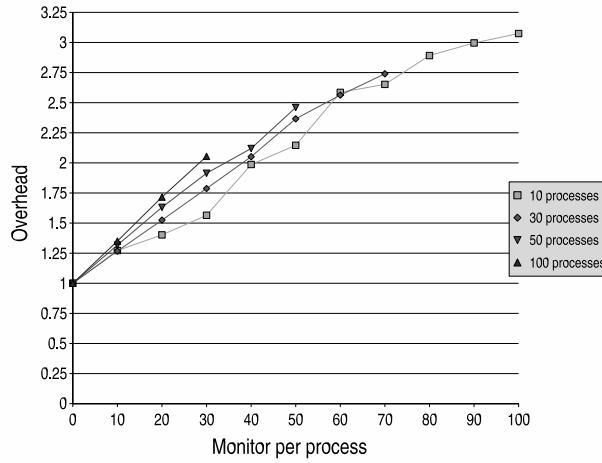

Fig. 11.12. Experiments with monitor generation

four curves that measure the overhead with a different number of process instances: 10, 30, 50, and 100. Notice that the overhead, even for a very high number of monitors per process, is acceptable, and the decrease in performances is not high. Consider the case of 10 process instances. We have 25% overhead with 10 monitors for each instance. The execution with 40 running monitors per process instances takes just twice the time required for running processes without monitors. It is about three times with 100 monitors per process instances (for a total of 1000 monitors running on the run-time environment). Moreover, the overhead does not increase significantly by increasing the number of process instances. In the case of more than 10 process instances, the fact that results are not reported after a given number of monitors is due to the fact that, in the setting we used for the experiments, the memory is exhausted if more than 3000 monitor instances are running at the same time. We are working to a more efficient implementation of the run-time monitoring environment that solves this problem.

Overall, the experiments show that monitor generation can be done efficiently also for complex component services and properties. Moreover, they show that run-time checking does not reduce significantly performances, and that this fact is independent of the load of the BPEL execution engine.

In conclusion, it turns out that both automated composition and run-time monotoring manage to deal in practice with cases of a certain complexity, have the potentiality to scale up to real applications, and can reduce significantly the effort in the development process. We leave for the future an extensive user evaluation dedicated to test the acceptance of the technology from the point of view of the developers.

11.7 Conclusions and Related Work

As far as we know, the contribution described in this chapter presents several elements of novelty. The work provides a uniform framework that integrates the automatic generation of composed BPEL processes with the automated generation of monitors. Moreover, the approaches to both composition and monitoring present elements of novelty by themselves, as discussed in the next two subsections.

11.7.1 Automated Composition of Web Services

In this chapter, we address the problem of automated composition in a *mediator-based architecture* where, given a set of component services (defined in our case as abstract BPEL processes), and a composition requirement, we synthesize one new web service that acts as a mediator and implements the composition by interacting with the component services. A different problem is that of the automated composition in a *peer-to-peer architecture*, where, given n component services and a composition requirement, the task is to generate n distributed new executable BPEL processes, one for each component, that interact with their own component and with the other new BPEL processes that are generated. The extension of our approach to a peer-to-peer automated composition is under study and development.

As far as we know, no other approaches provide the capability of taking into account assumptions in the automated composition of web services. In the following, we consider other approaches that have some relations with our underlying technique for automated composition without assumptions.

Automata-Based Approaches

The approach presented in this chapter is based on the idea that published abstract BPEL processes and composed executable BPEL processes can be given semantics and can be translated to state transition systems.

In [9, 7, 8, 10], the authors describe web services in terms of their interactions, e.g., with state machines. They do not provide an automated composition technique like the one described in this chapter.

In [27], a formal framework is defined for composing e-services from behavioral descriptions given in terms of automata. This approach considers a problem that is fundamentally different from ours, since the e-composition problem is seen as the problem of coordinating the executions of a given set of available services, and not as the problem of generating a new composite web service that interacts with the available ones. Solutions to the former problem can be used to deduce restrictions on an existing (composition automaton representing the) composed service. We generate (the automaton corresponding to) the BPEL composed service, thus addressing directly the problem of reducing time, effort, and errors in the development of composite web services.

In [11, 12], decision procedures for satisfiability are used to address the problem of coordinating component services that are described as finite state machines. The model used in [11, 12] is based on a finite alphabet of activity names, and transitions labeled with activity names specify the process flow of component services, while input and output messages are not modeled. In the initial works, the model is limited to deterministic state transition systems, while in [12], the framework is extended to non-deterministic finite state transition systems, corresponding to a "devilish" form of non-determinism. In this setting, the so-called "realizability" problem, i.e., the problem of determining the existence of a composition, can be solved as a satisfiability problem in propositional dynamic logic. The work studies the complexity of this reduction. Similarly to the work in [27], in these works the e-service composition problem is reduced to selecting among the activities that the component services should perform, a problem that is fundamentally different from the one addressed in this chapter, both in a mediator-based and in a peer-to-peer architecture.

More in general, our work shares some ideas with work on the automata-based synthesis of controllers (see, e.g., [43, 50]). Indeed, the composite service can be seen as a module that controls an environment which consists of the component services. However, the work on the synthesis of controllers is based on rather different technical assumptions on the interaction with the environment (BPEL interactions are asynchronous), and on a different language for expressing requirements, which cannot distinguish among primary and secondary requirements. Finally, this work has never been extended or applied to deal with the problem of the synthesis of web services, and in particular of BPEL processes.

Semantic Web Services

The semantic web community has used automated planning techniques to address the problem of the automated discovery and composition of semantic web services, e.g., based on OWL-S [19] or WSMO [26] descriptions of input/outputs and of preconditions/postconditions (see, e.g, [33, 32]). Two general remarks are in order. First, while here we do not address the problem of discovery (we assume the n component services are given), we tackle a form of automated composition that is more complex than the one considered by the semantic community, where usually services are atomic and compositions are simply sequences of service invocations. In our problem, services do not correspond to actions in the planning domain. Second, while here we do not address the problem of the automated composition of web services with semantic annotations, the approach can be extended to semantic web services along the lines of the work presented in [49, 38]. In [40], semantic annotations are kept separated from process descriptions, thus allowing for a practical and incremental approach to the use of semantics.

There is a large amount of literature addressing the problem of automated composition of semantic web services. However, most of the approaches address composition at the functional level (see, e.g. [35, 20]), and much less emphasis has been devoted to the problem of process-level composition. In [33], web service composition is achieved with user defined re-usable, customizable, high-level procedures expressed in Golog. The approach is orthogonal to ours: Golog programs can express programming control constructs for the generic composition of web service, while we automatically generate plans that encode web service composition through programming control constructs. In [32], Golog programs are used to encode complex actions that can represent DAML-S process models. However, the planning problem is reduced to classical planning and sequential plans are generated for reachability goals. In [34], the authors propose an approach to the simulation, verification, and automated composition of web services based on 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, the automated composition is again limited to sequential composition of atomic services for reachability goals, and does not consider the general case of possible interleavings among processes and of extended business goals. Moreover, Petri Nets are a rather expressive formalism, but algorithms that analyze them have less chances to scale up to complex problems compared to symbolic model-checking techniques.

The work in [31] is close in spirit to the general objective of [49, 38, 40] to bridge the gap between the semantic web framework and the process modeling and execution languages proposed by industrial coalitions. However, [31] focuses on a different problem, i.e., that of extending BPEL with semantic web technology to facilitate web service interoperation, while the problem of automated composition is not addressed.

Planning for Web Services

Different automated planning techniques have been proposed to tackle the problem of service composition, see, e.g., [51, 22, 47]. However, none of these can deal with the problem that we address in this chapter, where the planning domain is non-deterministic, partially observable, and asynchronous, and goals are not limited to reachability conditions.

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 [48] by using CSP techniques. In [28] an interleaved approach of planning and execution is used; planning techniques 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. Works in the field of Data and Computational Grids are more and

more moving toward the problem of composing complex workflows by means of planning and scheduling techniques [14].

Planning for the automated discovery and composition of semantic web services, e.g., based on OWL-S, is used in [33, 32, 34]. These works do not take into account behavioral descriptions of web service, like our approach does with BPEL.

Our work is based on the idea of and extends the technique called "planning via symbolic model checking" [18, 13, 21, 2, 39], a framework that differentiates from classical planning techniques since it can deal with planning in non-deterministic domains, with partial observability, and with goals that can express requirements with temporal and preference conditions. A detailed discussion on how the planning via symbolic model checking approach must be extended to deal with asynchronous domains that are constructed from BPEL processes can be found in [41]. In [37], the approach is extended to deal with large and possibly infinite ranges of data values that are exchanged among services.

11.7.2 Run-Time Monitoring of Web Services

Run-time monitoring has been extensively studied in different areas of computer science, such as distributed systems, requirement engineering, programming languages, and aspect-oriented development, see, e.g., [23, 25, 36, 46]. There have been several proposals that deal with different aspects of the monitoring of web services and distributed business processes, see, e.g., [45, 44, 4, 5, 29, 30]. A different but related topic is that of monitoring service level agreements (SLAs), i.e., contracts on services between parties that are signed to guarantee some quality of service, satisfy expectations, control costs, and resources. Monitoring SLAs means monitoring their compliance and reacting properly if compliance is not satisfied. An extension of our framework to the monitoring of SLAs is in our plans for future work.

Considering the problem of monitoring BPEL processes, an obvious alternative to our approach would be to code manually monitors in BPEL. The developer should embed special-purpose controls in the BPEL process implementing the business logic. However, this approach has several drawbacks. It does not allow for a clear separation of the business logic from the monitor, it does not allow for implementing monitors that capture misbehaviors caused by BPEL execution engines, and finally but perhaps more importantly, this task is time-consuming, error prone, requires programming effort, and does not allow for an independent maintenance of the monitor functionality w.r.t. the application layer. Similar problems exist in different frameworks based on BPEL, see, e.g., BPELJ [6]. BPELJ allows the programmer to embed monitors as Java code into BPEL processes.

The works closest to ours are those described in [4, 5] and in [29, 30]. We refer to them as *assertion-based monitoring* and *requirement-based monitoring.*

Assertion-Based monitoring

In [4], monitors are specified as assertions that annotate the BPEL code. Assertions can be specified either in the C# programming language or as pre- or post-conditions expressed in the CLIX constraint language. Annotated BPEL processes are then automatically translated to "monitored processes," i.e., BPEL processes that interleave the business processes with the monitor functionalities. This approach allows for monitoring time-outs, runtime errors, as well as functional properties.

In [5], Baresi and Guinea extend the work presented in [4] with the ability to perform "dynamic monitoring," i.e., the ability to specify monitoring rules that are dynamically selected at run-time, thus providing a capability to dynamically activate/deactivate monitors, as well as to dynamically set the degree of monitoring at run-time. Monitoring rules abstract web services into UML classes that are used to specify constraints on the execution of BPEL processes. In [5], assertions are specified in WS-COL (Web Service Constraint Language), a special purpose language that extends JML (Java Modeling Language) with constructs to gather data from external sources. Monitoring rules are defined with parameters that specify the degree of monitoring that has to be performed at run-time. The user can instantiate dynamically these parameters at run-time, changing in this way the amount of monitoring that is performed.

On the one hand, the approach described in [4, 5] provides some advantages w.r.t. ours. First, monitors are themselves services implemented in BPEL. As a consequence, they can run on standard BPEL engines without requiring any modification. A further challenge could also be the possibility to apply composition techniques developed for the BPEL business logic to the monitoring task. Second, annotations of BPEL processes with assertions constitute an easy and intuitive way to specify monitor tasks. Finally, the approach is extend to dynamic monitoring, a feature that is not provided in our framework.

On the other hand, we allow for the monitoring of properties that depend on the whole history of the execution path. These kinds of monitors would be hard to express as assertions. Moreover, we allow for a clearer separation of the business logic from the monitoring task than in [4, 5], since we generate an executable monitor that is fully distinguished from the executable BPEL that runs the business logic. Finally, our monitors can capture misbehaviors generated by the internal mechanisms of the BPEL execution engine. For instance, since there is no way to guarantee that a message is sent to a process instance only when the instance is ready to consume it, in BPEL, messages can be consumed in a different order from how they are received: indeed a process may receive a message that it is not able to accept at the moment, which can be followed by another message that can instead be consumed. The first message can be consumed later on by the process, or may never be consumed. This phenomenon, that we call *message overpass*, cannot be captured by monitors based on assertions that annotate the BPEL code.

Requirement-Based monitoring

In the work described in [29, 30], Mahbub and Spanoudakis share with us the idea to have a monitor that is clearly separated from the BPEL processes. Another similarity is that the framework allows for specifying requirements that represent either behavioral properties or assumptions to be monitored.

The framework allows for extracting automatically the behavioral properties from the abstract BPEL specification of component services. Requirements to be monitored are expressed in event-calculus, and the specified events are observed at run-time and stored in a database. An algorithm based on integrity constraint checking is then used to analyze the database and perform a run-time checking of the specified behavioral properties and assumptions.

The technical setting of this work is very different from ours. It is based on event calculus rather than linear temporal logic and on constraint checking rather than model checking.

References

1. ActiveBPEL. The Open Source BPEL Engine - http://www.activebpel.org.
2. A. Albore and P. Bertoli. Generating Safe Assumption-Based Plans for Partially Observable, Nondeterministic Domains. In *Proc. AAAI*, 2004.
3. T. Andrews, F. Curbera, H. Dolakia, J. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weeravarana. Business Process Execution Language for Web Services (version 1.1), 2003.
4. L. Baresi, C. Ghezzi, and S. Guinea. Smart Monitors for Composed Services. In *Proc. of Int. Conf. on Service-Oriented Computing*, 2004.
5. L. Baresi and S. Guinea. Towards dynamic monitoring of WS-BPEL Processes. In *Proc. of Int. Conf. on Service-Oriented Computing*, 2005.
6. BEA and IBM. BPELJ: BPEL for Java - http://www-106.ibm.com/developerworks /webservices/library/ws-bpel.
7. B. Benatallah, F. Casati, H. Skogsrud, and F. Toumani. Abstracting and Enforcing Web Service Protocols. *Int. Journal of Cooperative Information Systems*, 2004.
8. B. Benatallah, F. Casati, and F. Toumani. Analysis and Management of Web Services Protocols. In *ER*, 2004.
9. B. Benatallah, F. Casati, and F. Toumani. Representing, analysing and managing web service protocols. *Data Knowl. Eng.*, 58(3), 2006.
10. B. Benatallah, F. Casati, F. Toumani, and R. Hamadi. Conceptual Modeling of Web Service Conversations. In *CAiSE*, 2003.
11. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of E-Services that export their behaviour. In *Proc. ICSOC'03*, 2003.
12. D. Berardi, D. Calvanese, G. De Giacomo, and M. Mecella. Composition of Services with Nondeterministic Behaviours. In B. Benatallah, F. Casati, and P. Traverso, editors, *Proceedings of the Third International Conference on Service-Oriented Computing (ICSOC'05). Lecture Notes in Computer Science LNCS 3826*. Springer, 2005.

13. P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Planning in Nondeterministic Domains under Partial Observability via Symbolic Model Checking. In *Proc. IJCAI'01*, 2001.
14. Jim Blythe, Ewa Deelman, and Yolanda Gil. Planning for Workflow Construction and Maintenance on the Grid. In *Proceedings of ICAPS'03 Workshop on Planning for Web Services*, Trento, Italy, June 2003.
15. B. Bonet and H. Geffner. Planning with Incomplete Information as Heuristic Search in Belief Space. In *Proc. AIPS'00*, 2000.
16. R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Survey*, 24(3):293–318, 1992.
17. A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *CAV*, 2002.
18. A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking. *Artificial Intelligence*, 147(1-2):35–84, 2003.
19. The OWL Services Coalition. OWL-S: Semantic Markup for Web Services. In *Technical White paper (OWL-S version 1.0)*, 2003.
20. I. Constantinescu, B. Faltings, and W. Binder. Typed Based Service Composition. In *Proc. WWW2004*, 2004.
21. U. Dal Lago, M. Pistore, and P. Traverso. Planning with a Language for Extended Goals. In *Proc. AAAI'02*, 2002.
22. D. Mc Dermott. The Planning Domain Definition Language Manual. Technical Report 1165, Yale Computer Science University, 1998. CVC Report 98-003.
23. A. Dingwall-Smith and A. Finkelstein. From Requirements to Monitors by way of Aspects. In *Int. Conf. on Aspect-Oriented Software Development*, 2002.
24. E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier, 1990.
25. M. Feather and S. Fickas. Requirements Monitoring in Dynamic Environment. In *Int. Conf. on Requirements Engineering*, 1995.
26. The Web Service Modeling Framework. SDK WSMO working group - http://www.wsmo.org/.
27. R. Hull, M. Benedikt, V. Christophides, and J. Su. E-Services: A Look Behind the Curtain. In *Proc. PODS'03*, 2003.
28. A. Lazovik, M. Aiello, and Papazoglou M. Planning and Monitoring the Execution of Web Service Requests. In *Proc. of the 1st International Conference on Service-Oriented Computing (ICSOC'03)*, 2003.
29. K. Mahbub and G. Spanoudakis. A Framework for Requirements Monitoring of Service Based Systems. In *Int. Conf. on Service-Oriented Computing (ICSOC)*, 2004.
30. K. Mahbub and G. Spanoudakis. Run-Time Monitoring of Requirements for Systems Composed of Web-Services: Initial Implementation and Evaluation Experience. In *Int. Conf. on Web Services (ICWS)*, 2005.
31. D. Mandell and S. McIlraith. Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation. In *Proc. of 2nd International Semantic Web Conference (ISWC03)*, 2003.
32. S. McIlraith and R. Fadel. Planning with Complex Actions. In *Proc. NMR'02*, 2002.

33. S. McIlraith and S. Son. Adapting Golog for composition of semantic web Services. In *Proc. KR'02*, 2002.
34. S. Narayanan and S. McIlraith. Simulation, Verification and Automated Composition of Web Services. In *Proc. WWW2002*, 2002.
35. M. Paolucci, K. Sycara, and T. Kawamura. Delivering Semantic Web Services. In *Proc. WWW2003*, 2002.
36. D.K. Peters. Deriving Real-Time Monitors for System Requirements Documentation. In *Int. Symp. on Requirements Engineering - Doctoral Symposium*, 1997.
37. M. Pistore, A. Marconi, P. Bertoli, and P. Traverso. Automated Composition of Web Services by Planning at the Knowledge Level. In *Proc. Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 2005.
38. M. Pistore, P. Roberti, and P. Traverso. Process-level compositions of executable web services: on-the-fly versus once-for-all compositions. In *Proc. ESWC'05*, 2005.
39. M. Pistore, D. Shaparau, and P. Traverso. Contingent Planning with Goal Preferences. In *Proc. AAAI'06*, 2006.
40. M. Pistore, L. Spalazzi, and P. Traverso. A Minimalist Approach to Semantic Annotations for Web Processes Compositions. In *Proc. ESWC'06*, 2006.
41. M. Pistore, P. Traverso, and P. Bertoli. Automated Composition of Web Services by Planning in Asynchronous Domains. In *Proc. Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 2005.
42. M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated Synthesis of Composite BPEL4WS Web Services. In *IEEE Int. Conf. on Web Services (ICWS)*, 2005.
43. A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Proc. ICALP'89*, 1989.
44. W. Robinson. Monitoring Web Service Requirements. In *IEEE Int. Conference on Requirement Engineering*, 2003.
45. A. Sahai, V. Machiraju, A. van Morsel, and F. Casati. Automated SLA Monitoring for Web Services. In *Int. Workshop on Distributed Systems: Operations and Management*, 2002.
46. K. Sen, A. Vardhan, G. Agha, and G. Rosu. Efficient Decentralized Monitoring of Safety in Distributed Systems. In *Proc. of ICSE*, 2004.
47. M. Sheshagiri, M. desJardins, and T. Finin. A Planner for Composing Services Described in DAML-S. In *Proc. AAMAS'03*, 2003.
48. Snehal Thakkar, Craig Knoblock, and Jose Luis Ambite. A View Integration Approach to Dynamic Composition of Web Services. In *Proceedings of ICAPS'03 Workshop on Planning for Web Services*, Trento, Italy, June 2003.
49. P. Traverso and M. Pistore. Automated Composition of Semantic Web Services into Executable Processes. In *Proc. Int. Semantic Web Conference (ISWC)*, 2004.
50. M. Y. Vardi. An automata-theoretic approach to fair realizability and synthesis. In *Proc. CAV'95*, 1995.
51. D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating DAML-S Web Services Composition using SHOP2. In *Proc. ISWC'03*, 2003.