Composing Services for Third-party Service Delivery

Ingo Weber[1], Alistair Barros[2], Norman May[3], Jörg Hoffmann[3], Tomasz Kaczmarek[4]
[1] CSE department, University of New South Wales, Australia, ingo.weber@cse.unsw.edu.au
[2] SAP Research Brisbane, Australia, alistair.barros@sap.com
[3] SAP Research Karlsruhe, Germany, {norman.may | joe.hoffmann}@sap.com
[4] Poznan University of Economics, Poland, t.kaczmarek@kie.ae.poznan.pl

Abstract

This paper proposes a model-based technique for lowering the entrance barrier for service providers to register services with a marketplace broker, such that the service is rapidly configured to utilize the broker's local service delivery management components. Specifically, it uses process modeling for supporting the execution steps of a service and shows how service delivery functions (e.g. payment points) "local" to a service broker can be correctly configured into the process model. By formalizing the different operations in a service delivery function (like payment or settlement) and their allowable execution sequences (full payments must follow partial payments), including cross-function dependencies, it shows how through tool support, the non-technical user can quickly configure service delivery functions in a consistent and complete way.

1. Introduction

With maturing Web services technologies, SOA shifting from early adoption to mainstream development and on-premise hosting giving way to the "cloud", a new revolution of service provisioning is emerging. Companies are moving beyond "firewalls" and well-established value-chains through wider partnerships which expose and deliver their services to new markets. In service industries, new channels for service delivery are established through *service brokers* wherein services are published so that they may be discovered, bundled, ordered, accessed, paid for, and rewarded [2].

The most prominent example of service brokerage is Software-as-a-Service marketplaces [13], e.g. Salesforce AppExchange[1] and Rearden Commerce[2], which allow software services related to a common domain to be registered from the open community, integrated into a common suite and ordered though pay-per-use models. Another significant development is one-stop citizen-and constituency services in the public sector. Although they are not marketplaces in the strict commercial sense, they have similar features such as bringing consumers (e.g. citizens) and providers (government agencies) together to allow "one-stop" exposure of business services (e.g. land parcel checks and life events) through centralized channels, e.g. Direct-Gov.uk[3], usa.gov[4] and the European Union Services Directive[5] in planning. In turn, business service marketplaces are emerging, e.g. American Express Intelligent Online Marketplace (AXIOM)[6], governed by a dominant commercial player exposing services like flight, hotel and car rental bookings that are available through it and its global partners. The dominant player benefits from increased revenue by exposing wider choice and best deals for consumers while partners enjoy a greater market visibility for their services.

Given these trends, the question turns to how current technical solutions of brokerage scale for more complex services. In current marketplaces, the form of service delivery has similarities to conventional e-commerce marketplaces given that the consumable items are much like goods: software components or widgets, and business service listings are essentially presented in product catalogs. However, more complex services in service industries present salient differences. To take property conveyance, insurance claims, and business formation as examples, they are long-running, can involve multiple stakeholder and are often implemented through legacy applications hosted by their providers.

For third-party brokers to act as intermediaries for these sorts of business services and to be able to deliver these without hosting service implementations or assuming intimate domain expertise of services, special service interfacing is required between brokers and service providers. The interfaces enable brokers to orchestrate the execution of service steps which invoke operations of service implementations hosted through service providers. In addition, the in-

[1] http://www.salesforce.com/appexchange/
[2] http://www.reardencommerce.com
[3] http://www.direct.gov.uk
[4] http://usa.gov
[5] http://ec.europa.eu/internal_market/services/services-dir/index_en.htm
[6] http://www.americanexpress.com/axiom/

terfaces allow brokers to provide "front-desk" service delivery support, for instance checking customer credentials and access permissions for services, collecting payment, calculating rewards, and even subsidizing payment by inclusion of advertising or cost subsidizing services (e.g. collecting customer-particular and general service utilization statistics for markers). Operations which enable these would be configured into the brokered service, invoking service delivery components like customer-relationship management, payment and analytics, utilized by the broker.

In this paper, we propose a technique for configuring interfaces of long-running services so that they can be delivered through third-party brokers. In particular, we present a semantic configuration technique for business process-based descriptions representing exposed interfaces of services to service brokers. The key contribution is configuring business process models for service delivery functions at different steps, in a consistent and complete way. For this purpose, we combine and adapt our earlier work on service composition [7] and verification of semantically annotated process models [14]. In contrast to other semantic process composition techniques, we hereby require neither an up-front semantic annotation of the provider service nor a-priori knowledge of the service consumer. In this way, we present a realistic solution with a low entry barrier for service providers. Also in contrast to many approaches using semantics, the additional modeling effort for the broker is minimal and can be seen as negligible in comparison to the implementation effort of the service delivery functions.

The paper is structured as follows. In Section 2, an insight is presented for service delivery configuration through a multi-stakeholder example taken from the public sector domain, namely business formation. In Section 3, a technique for supporting semantic configuration of service delivery needs is presented in a way that ensures consistency and completeness of configuration. In Section 4, implementation details are provided including consistency and completeness checking and the implemented prototype. Section 5 provides a comparison of the proposed technique with the state-of-the-art and Section 6 concludes the paper with open issues and further research.

2. Motivating Example

To illustrate the requirements for outsourced service delivery and the nature of configurations therein, consider a small business formation service (e.g. opening up a financial planning firm or a restaurant)[7]. This involves obtaining a business license through specific provisions which are granted by different agencies (e.g. workplace health and safety, partial street occupation). While governments have invested significantly in aggregating the different steps involved in an otherwise lengthy process, they also see the opportunity for third-parties to expose the service to different markets to encourage business development. Figure 1 depicts the highest level of abstraction for the aggregated service as it exists with the aggregating service provider (on the right-hand side) and an extended version with required configurations for its delivery through a third-party service broker (on the left-hand side)

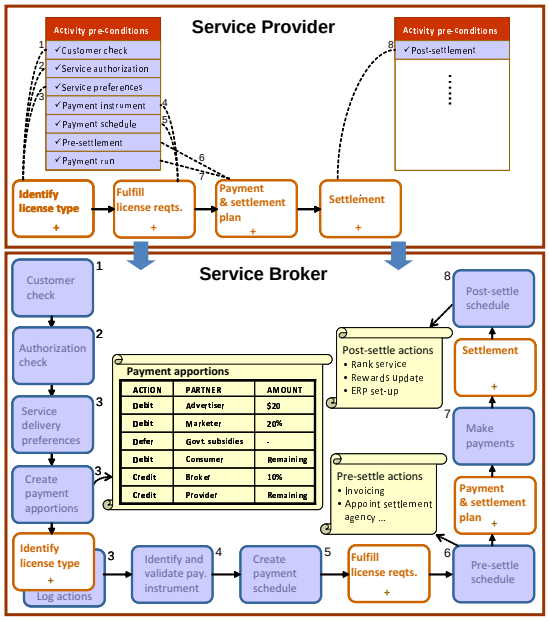

Figure 1. Configuring a service for delivery through a service broker.

The provider's service has four sub-processes: a questionnaire phase to *identify the license type*; negotiations between the applicant and the relevant agencies in order to *fulfill the license requirements*; negotiations for the *payment & settlement plan*; and finally exchange of documents and material goods as part of *settlement*. The corresponding service as deployed in the service broker is the result of configuring the provider's service. The individual configurations that have been applied to produce it are numbered.

The first configuration involves specifying a *Customer Check* performed by the broker prior to *Identify License Type* (as an activity pre-condition). This allows the broker to determine whether the customer requesting the service is registered, has rewards, preferences, previous exceptions etc. This is followed by the second configuration for an *Authorization Check*. The third configuration relating to services preferences expands to three pre-condition activities: *Service Delivery Preferences* where the service price is configured to allow for other partners to subsidize the cost (e.g. advertisers, marketers); accordingly, to *Create Payment Apportions*; and *Log Actions* so that customer choices

[7]This example is based on the SmartLicence service offered through a Queensland government agency, `http://www.sd.qld.gov.au/dsdweb/htdocs/slol/`.

are recorded for trend analysis. Through these subsidizing steps, we can see how brokers could re-price services, making them cheaper for consumers while still passing the required revenue back to the providers.

Once the consumer makes a decision to proceed with investigations to obtain a business license, work by government agencies is required, and therefore a configuration is made to *Identify and Validate Payment Instrument* followed by a further pre-condition activity to *Create Payment Schedule* (based on the previously specified payment apportions). The broker can offer further support through assistance for settlement, i.e. *Pre-Settlement Schedule*, prior to *Payment & Settlement Plan*. Prior to *Settlement*, *Make Payments* is configured to run the credit/debit transactions in the payment schedule. After *Settlement* (as an activity post-condition), the broker can extend support for bootstrapping the business, like providing leads for taxation and other ERP service, through *Post-Settlement Schedule*.

3. Semantic Service Delivery Configuration

Having provided an insight into service delivery configuration, we now present the details of the suggested technique to support the respective requirements. These are: (i) service delivery functions at the broker need to be reusable for many provider services; (ii) for providers it must be easy to describe the required configuration; and (iii) applying a described configuration must be automated in order to enable mass-configuration of services on the marketplace platform. When the provider uploads the service description, then the process model for using the interface can either be part of this description, derivable from it, or manually created by the provider. This process is used for the configuration in the broker's setting. To allow for individual actions of payment, settlement and the like to be exposed for configuration, we represent the lifecycles of service delivery *entities* as state machines, illustrated in Figure 2.

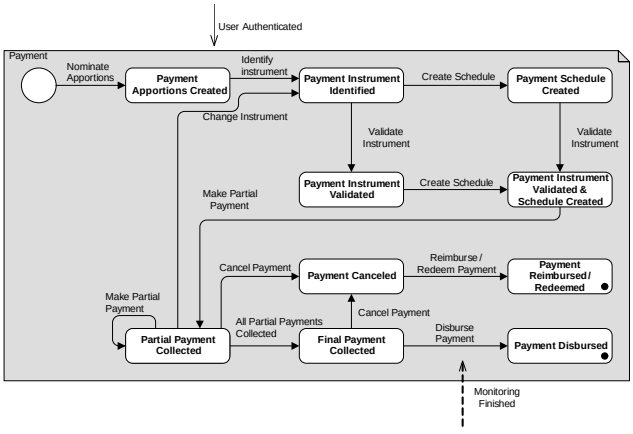

Figure 2. Expressing possible states and transitions of the payment entity.

The transitions correspond to logical actions of the entity which can be configured, and the partial order of the transitions constrains the order of allowable configurations. Each logical action corresponds to one or more activities which can be used in the configuration of the provider's process model. For configuration the provider selects, e.g., in a questionnaire form, the entities that are of relevance to her service – e.g., if authentication is handled by the service itself, or simply unnecessary in the given case, then the authentication entity is not required for the service at hand. The operations of the selected entities can then be interwoven with the process model: for each call to a provider's service operation in the process, the user can select which entity states should be reached before, respectively after, its execution. When done with the annotation of desired entity states to process activities, the user can signal that this configuration should be applied, which means automatic insertion of calls to the broker's service delivery operations at the respective points in the process.

When the configuration of service delivery is finalized, a check is run to determine if all entity lifecycles reach a final state when the execution of the process ends. If this is not the case (e.g., payment resides at an intermediate stage and is neither finalized nor canceled), then this signifies that the usage of the service delivery functions in the process is invalid. Suggestions are then made to the user how to fix the problems, e.g., by suggesting to collect the final payment and other necessary steps before the process ends.

While the user interaction with a tool following the described approach is not overly complex, there are some intricate problems underneath the hood. One of them is the question of the state an entity will be in at a certain point in the process – which is of course required knowledge in order to find out how the desired state can be reached. Hereby it should be noted that the process can be much more complex than the simple sequence shown in Figure 1. Another question is how to reach the desired state, given the current state and the available delivery functions. We address these questions with a systems approach, that is, we build on prior results from our research. In particular, service composition [7] is used for the latter problem and techniques from the verification of annotated process models [14] for the prior. Section 4 addresses the usage of these techniques, following the description of service delivery modeling below.

3.1. Modeling Service Delivery Functions

Service delivery functions are modeled as business entities with lifecycles and cross-entity relations. The metamodel for lifecycles is the following. There are named entities (like payment) whose lifecycle is described through a contained state machine. This state machine has a single start state, multiple (intermediate) states, one or more end states, and transitions between the states. From each state,

an end state must be reachable. Except for the start state, all states must be named. In the simple case, each entity is exactly in one state during execution. The path on which this state has been reached is forgotten (the state machine is said to be *memoryless*, since it does not memorize the path to the state). Therefore, action labels attached to transitions may repeat (cf. *Create Schedule* in the example). In contrast, states are present only once in an entity.

The state machine in Figure 2 also indicates dependencies to other entities. The arrow at the top of the diagram labeled *User Authenticated* declares that the *Authentication* entity is required to be in this state for all states of the *Payment* entity until one end state is reached. Once an end state is reached, the state of the *Authentication* entity may change. Furthermore, the arrow pointing to the transition *Disburse Payment* is only executed if the *Monitoring* entity is in state *Monitoring Finished*. In contrast to the previous restriction, this check is only performed locally for this transition, i.e. the result of this check is not memorized.

The operations which can be configured from an entity are derived from transitions in its state machine. In the simple case, each transition has an action label (e.g. *Validate Instrument*) which corresponds to an operation for service delivery whose invocation can be configured into a branch of the process. Where needed, additional flexibility can be achieved in the following ways:

- Using Semantic Web Service (SWS) technology and according techniques for automatic composition, e.g., [7]. For this variant it suffices to relate the operations for service delivery to the states of the lifecycle, but not necessarily in a 1:1 or 1:n correspondence between state transitions in the lifecycle and operations. Instead, the operations then specify in which states they can be executed – the preconditions of an SWS – and which state(s) may result from the execution – the SWS postconditions. The lifecycle must still properly represent the according transitions, but the operations may also provide "shortcuts" through the lifecycles and can be invoked in more than one state.

- More complex or expressive lifecycle models, such as control-state abstract state machines (ASMs, cf. [4]) or hierarchical state machines may be used. This allows reducing redundancies that may be present in FSMs – as creating a schedule and validating the payment instrument in the example shown in Figure 2. Such representations are particularly applicable when there are multiple independent streams of transitions in a lifecycle: an FSM always materializes all possible combinations, and there may be exponentially many.

These two variants may also be combined – i.e., operations specify pre and postconditions in more complex lifecycle representations.

Each entity has to be in one of its final states when an execution of the process ends. Thus, the process model has to make use of an entity in a way that guarantees reaching a final state on each of the used entities. Take payment as an example: when the process ends, all payments should be either (i) finalized and disbursed according to the agreed revenue distribution among the involved parties, or (ii) canceled and any partial payments reimbursed. Any other usage is considered incomplete or invalid.

On the right-hand side of Figure 1, the registered process is shown with different configuration options possible as activity pre/post-conditions. Configurations result in changes to the process when the configuration is applied. The configuration needs to be consistent with the lifecycle and cross-entity dependencies. An activity pre/post-condition corresponds to labels of states in the state machines of entities. Selections must abide by the sequencing of related transitions within the state models as well as cross-entity dependencies. At subsequent steps only relevant states are shown – which is possible by computing how the entity states evolve throughout the process.

3.2. Modeling Dependencies Between Service Delivery Entities

Cross-entity dependencies can be used to model how the presence and current state of one entity affects the lifecycle of another. Figure 3 shows some entities relevant for service delivery, with their general dependencies depicted as arrows. *Customer* is the entity that allows CRM operations to be configured in brokered services. *Payment*, cf. Figure 2, allows payments pertaining to the brokered service to be configured. *Service Provider* allows interactions with the service's owner/host to be configured. *Monitoring* allows the broker to monitor the fulfillment of the service level agreement as well as the current execution state of a service operation. Finally, *Authentication* asserts the identity of a user registered with the service broker.

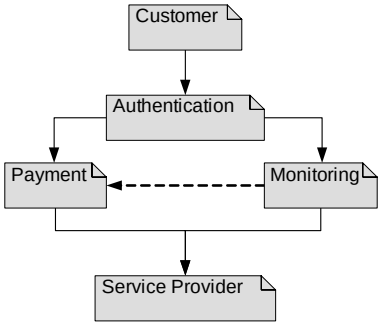

Figure 3. Structural entity dependencies.

The dependencies imply that a state transition in an entity A can lead to an action in another entity B (shown as

$A \rightarrow B$). The directed edges in Figure 3 leading from Authentication to Payment and Monitoring indicate that they are dependent on Authentication because the customer's identity needs to be verified to correctly associate payments and monitoring information. The dashed arrow indicates a weak dependency, meaning that dependencies between two entities exist, but their influence is limited to cases where the user selected the usage of both involved entities. In contrast, strong dependencies imply that when the dependent entity is selected, the other entity is implicitly selected as well. Note that bidirectional dependencies are also possible although not illustrated in this example.

Entity lifecycles model primarily the services delivery functions provided by service brokers. Payment seems an obvious choice for service delivery, unlike, say, Service Provider whose presence would only be warranted if various operations with partners need to be made explicit, in different parts of the brokered service. To illustrate the point, a Service Provider may require notifications of certain exceptions, disbursement of funds and feedback for service delivery operations in different parts of the brokered service. Since these are different operations which can be used in different parts of a process interface, Service Provider has been explicitly defined as service delivery entity.

4. Realization and Implementation

Having described how services can be modeled using entities and lifecycles, we now describe the realization in detail. Specifically, we describe the procedure for interweaving the service delivery functions with the process in Section 4.1. The check for completeness in the entity usage is discussed in Section 4.2. Finally, the proof-of-concept implementation is the subject of Section 4.3.

4.1. Service Composition for Brokered Service Configuration

The goal of this composition functionality is to make it easy for a user of the service broker to configure a broker process with service delivery functions – in particular for users who are not familiar with process modeling tools and the intricate details of the execution semantics of the modeling notation at hand. The approach is the following: the user selects a point in the process when an entity should reach a certain state and specifies which of the entity's states should be reached; this is shown in pseudo-code in Figure 4. When this annotation is finished for all tasks, we use (for each of the annotations) an automatic composition technique to get from the state in which the entity will be at this point in the process to the desired state; cf. the pseudo-code in Figure 5.

In more detail, the annotation procedure starts with the user selecting a point in the process where she wants certain guarantees to hold. In order to not configure illegitimate

procedure *get-annotation-options*(p:$process$, a:$activity$)
(1) *PS* := *possibleEntityStates*(*p*,*a*)
(2) *PC* := *allowedConfigurationOptions*(*PS*,*p*,*a*)
(3) **return** *PC*

Figure 4. Determine annotation options.

options, procedure *get-annotation-options* in Figure 4 first determines (line (1)) the possible entity states the process p will be in at a given activity a. These are states in the lifecycles of the involved entities which will be active at this point of the process. E.g., in the above example of the *Payment* entity (cf. Figure 2), if there is an activity configured such that *Payment instrument validated & schedule created* will hold, then we assume this will hold at any point after this activity – unless there is another activity that changes the status of *Payment*. Note that, given optional branches of processes, it may not be possible to determine the exact state in which an entity may be – instead, we can determine a set of states, any of which the entity may be in. The underlying functionality for line (1) is provided by our prior work on the verification of semantically annotated process models [14]; below, the details of the usage here are explained. Note that the approach is independent of the process modeling notation used; but there are limitations on the expressiveness – cf. [14]. Based on the set of possible entity states, line (2) determines the allowed configuration options as follows: allowed is any state that – according to the lifecycle – can be reached from any state in *PS*, and does not prevent an annotation "later" in the process to be reachable. Reachability of states in lifecycles can be computed with standard graph algorithms, like Dijkstra. The set of allowed annotation options is then returned to the user, who may select one of these options.

The original purpose of [14] is to use the annotation in form of preconditions and effects of process activities to find inconsistencies in semantically annotated process models. Therefore, we propose a formalism in [14], building on Petri Net-like token-passing mechanisms for the control-flow, and on notions from AI for the semantics of preconditions and effects. On this basis we define propagation algorithms for determining the part of the logical state that must be true whenever a given activity is activated. Herein, we use this algorithm for three purposes: to determine which configuration options are available; for determining the preconditions of the possible composition inputs, the goal settings; and for checking if all involved entities reach an end state when the process ends.

Given an annotated process, the *I-Propagation* algorithm from [14] can determine either the exact logical state in which a process is at any given stage of execution, given this state can be computed with certainty at design time; or an approximation of the logical state can be computed if the state depends on runtime information. For this purpose, we

transform the lifecycle state annotation to an effect annotation as follows: if there is a desired lifecycle state of entity A annotated as a pre-state or post-state to a given activity, then we annotate a logical effect representing this lifecycle state. If, however, the same entity A has an annotation a pre-state and post-state at a given activity, then we only annotate the representation of the post-state, since this will be the resulting effect from executing this activity and the according service delivery functions. In addition, we need to account for the workings of the lifecycle model, e.g., in an FSM there can be only one active state. Thus, if one state is known to be true, then all others must be false. This is captured by adding the *negated* representations of these respective other states to the effects as well. The I-Propagation algorithm from [14] then determines exact states, if any – or, where the exact state is unknown, it determines a set of possible states. Namely, the possible states are all those which are not contradicted by the partial state determined by I-Propagation. In the case of FSM lifecycles, this roughly works as follows. After the execution of I-Propagation we have, at any point in the process, the set *S* of all lifecycle state representations of all entities which are definitely true when a process execution is at that point. If a state of an entity is contained in *S*, then the entity will definitely be in that state. If there is no such state, then I-Propagation will at least have identified a set *not-S* of lifecycle states which the entity will definitely *not* be in, cf. the discussion of negated state representations above. The entity is then known to be in any one of its states, except those in the set *not-S*. Hence there is uncertainty about the actual state of the entity, but the uncertainty can be narrowed down as far as possible.

procedure *compose-configuration*(p:*process*)
(1) *S* := set of all annotations *s*, ordered by control flow of *p*
(2) **for** each annotation *s* in *A* at activity *a*
  (3) *PS* := *possibleEntityStates*(*p*,*a*)
  (4) *Goals* := {(*s'*,*s*) | *s'* ∈ *PS* ∧ entity(*s'*) = entity(*s*) }
  (5) **for** each *g* ∈ *Goals* : *compose*(*g*)
(6) presentation of results for user approval
(7) **if** approved: update process

Figure 5. Composition for interweaving.

Once the user finished annotating the process with desired entity states, the configuration can be applied using the *compose-configuration* function shown in Figure 5. In the order given by the control flow of the process p, the loop in line (2) considers the annotations of desired states s (for the various entities) at all tasks a. First, in line (3) the possible entity states for a are determined as above in Figure 4. Subsequently, line (4) determines all possible goal settings. That is, if there are multiple possible states (*PS*) in which the process may be, then these need to be separated from one another, and the desired behavior for each of these is described in a separate semantic goal. A semantic goal here consists of preconditions (*s'* in line (4)) and postconditions (s), e.g., similar to WSMO [11].

Then, for each such goal setting we invoke service composition in line (5). Thus, for each of those options, the composer creates a chain of services that can fulfill this goal. These different compositions are placed in optional process branches at the point of the process (before / after the annotated activities) where required in order to satisfy the user annotation – i.e., reach the desired state in an entity. The optional branches are marked with the respective condition from the goal setting that led to this composition. The annotation and the current process state are kept in memory, so that possible future updates of the annotation can trigger re-compositions if these become necessary. To illustrate the optional branches in the running example of the *Payment* entity, say the current state is either *Payment instrument identified* or *Payment instrument validated*, and the desired state is *Payment instrument validated & schedule created*. Assume for the sake of simplicity that there are individual services for each of the transitions; then a service chain for each of the possible states is constructed as depicted in Figure 6. Finally, the result is presented to the user (line (6)), and upon approval the process is updated (7).

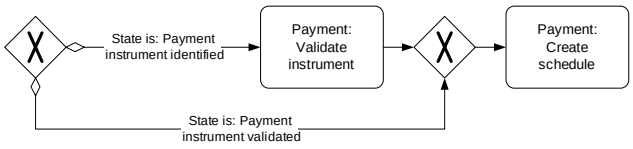

Figure 6. Composed partial process for multiple possible start states.

In a more complex variant, the user can call composition online, directly after selecting a desired state. This complicates matters since it may necessitate re-composing other parts of the process. For lack of space, we omit the details.

4.2. Brokered Service Completeness Check

While the service composition aims at reaching certain goals in terms of the entities involved in the process at a given point, the completeness check makes sure that each entity is in an end state when the process ends. This criterion must be met before the process can be deployed. According to our lifecycle model, each lifecycle must have at least one reachable end state. The according check is performed by the function in Figure 7.

When the user requests a check of the broker process' configuration or the process is to be deployed but has not been checked yet, the *completeness-check* function is executed. It starts (line (1)) by determining the possible states of the entities at the end node (end_p) of the process (p). If all involved entities reach one of their end states, then the check

procedure *completeness-check*(p:*process*)
(1) PS := *possibleEntityStates*(p,end_p)
(2) **if** all $s \in PS$ are final states: **return** "complete"
(3) **for** each $s \in PS$ that is not final:
  (4) FS := $\{s' \mid s'$ reachable from $s \wedge s'$ final$\}$
  (5) **if** $|FS| > 1$: determine s' by user selection
    else define s' by $FS = \{s'\}$
  (6) *compose*(s,s')
(7) presentation of results for user approval
(8) **if** approved: update process

Figure 7. Completeness check.

ends successfully. Otherwise, a solution for each possible state that is not an end state is proposed as follows. Line (3) loops through all such states s. Inside the loop, the set of end states reachable from s is determined (4). Per definition, at least one final state has to be reachable from any other state. If, however, this set contains more than one state, then the user is asked in line (5) which of these end states is desired for s. Either way, the composition goal, on which composition is executed (6) is set to reaching the desired or the only final state from s. The outcome is presented to the user (7) and upon approval, the process is updated (8).

4.3. Prototypical Implementation

The prototype is implemented as an extension to Oryx [5]. A screenshot for making the configuration is shown in Figure 8: when the user changes to the configuration mode she selects the relevant entities. In the configuration mode, she can then specify (using a context menu) for each activity in the provider process which state should be reached before (PRE) or after (POST) the execution of this activity. In this case, the payment entity has been selected – albeit with a simplified lifecycle in comparison to Figure 2. Furthermore, prior to the chosen activity the entity is already used; therefore, the state "Payment instrument identified and validated" is the first state that can be selected here.

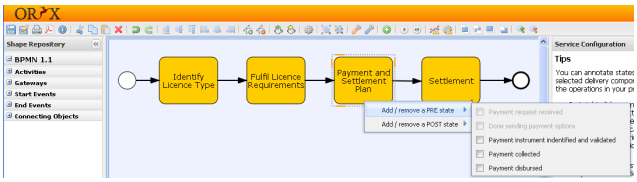

Figure 8. Configuration in the prototype.

We validated our techniques using a few example scenarios, but were not yet able to run more large-scale experiments. However, first demonstrations of our prototype within SAP were quite successful. Indeed, our technology is currently being implemented in a commercial SAP tool (NetWeaver BPM), and is scheduled for pilot evaluation with customers in the near future.

5. Related Work

The work presented here is related to process mediation, process composition, lifecycle compliance, marketplace brokers in general, and our own prior work.

Process mediation, e.g., [1, 15], is the problem of mediating between a set of fixed process interfaces, which, without the mediator, would not fit together. The mediation process has some resemblance with the orchestration process discussed in this paper. In contrast to our state annotation approach, process mediation typically operates based on data dependencies.

There is a body of work on the **automatic composition** of processes, aiming to combine multiple processes in a manner that satisfies some target, e.g. [10][6][3]. Such composition methods could in principle be applied to solve (parts of) the brokerage problem we consider here. The main difference between our technique and the known composition methods is that the latter methods are fully automatic one-shot composers, i.e., the composition tool receives as input the descriptions of processes and target, and outputs the final composed process. By contrast, our approach works incrementally in interaction with the user, establishing the final composed process in a step-wise fashion. From the point of view of maximized automation, our approach clearly is weaker. However, such a viewpoint disregards the fact that creating the models for the "automatic" composer is human labor as well. In addition, our method allows us to handle more complex processes with less computational effort, as described below. In that sense, our contribution is a specialized composition methodology suitable for service brokerage, which improves on existing methods in terms of modeling effort and/or computational efficiency. Concretely, [10] is a well-known method for composing processes modeled as finite state machines. The approach suffers from complicated target declarations (specifying desired properties of the process to-be-composed) and from severe computational complexity. [6] improves on the complexity issue at the price of severely restricted processes, forcing them to have a tree structure. Various works, e.g. [3] encode the composition problem into some form of logic, and reasoning in that logic for doing the composition. Clearly, this also suffers from the complexity issue.

One thread of related work investigates the relations between **object lifecycles** and process models [8, 12]. While [12] presents a mechanism to check whether the usage of an object in a process model is compliant with its lifecycle, [8] addresses the problem of generating process models on the basis of multiple (possibly linked) object lifecycles. However, there the full model is composed automatically, whereas we describe how an existing process can be interweaved with actions from a lifecycle (or, more precisely, service delivery functions that stand behind a lifecycle).

Current **marketplace brokers**, such as StrikeIron and

Salesforce AppExchange, suffer from two key limitations: Firstly, they assume a fixed service delivery model where services are ordered, invoiced, and paid through demand-usage. They prescribe broker services to choose from, e.g. pay-per-use, and the marketplace counts the invocations and creates invoices on behalf of the service providers. Moreover, the set of broker services is difficult to extend. Secondly, access to a service entails getting a reference to the service interface. As the current brokers do not mediate interactions with a service, it is not possible to mix those with service delivery functions. Our work allows for far more flexibility and extensibility than current marketplaces as a registered service can be configured with arbitrary broker services. This description facilitates broker services to be interleaved with operations of the registered service.

Composition and state derivation can be realized with **our previous work**. Our results on scalable composition technique are presented in [7]; in principle, other composition or planning methods may be applied. [14] describes our method for the verification of annotated process models. In the work at hand, we extend upon these works by showing how one can leverage on them to support the effortless interleaving of broker services with operations of the registered service.

6. Conclusions

We show how to configure services to use service delivery functions of a broker for commodity support functions. Service providers link the desired states of the entities representing the delivery functions to be interweaved with the (potentially complex) flow of service provider operations. By applying the configuration, the provider can easily turn technical services into tradable services. As a prerequisite, the broker needs to describe the allowed interactions with service delivery functions. As one major benefit service providers can refer to these predefined delivery operations when configuring their service offers, and thereby significantly reduce the effort for making their services tradable. In the process of configuration, the service broker can make sure that valid final states are reached with respect to the entities used during configuration. Thus, another benefit for all involved actors is that erroneous usage is avoided before deploying the service.

A major issue with applications of semantic technologies in general is the effort and cost in defining, maintaining and utilizing semantic descriptions [9]. Translated into the context of this paper, this would be cost of defining, maintaining and using service delivery entities with the mapping to service operations. The respective overhead is low because the only domain pertaining to semantic configuration is service delivery, and this is a one-size-fits-all for service brokers. As the entities would be defined and maintained by the service marketplace, and real-world brokers would allow for limited refinement of their state machines, the effort invested into modeling the behavior of the entities is expected to pay off quickly.

Our prototype has not yet been evaluated comprehensively, but has spawned sufficient interest within SAP to be now underway for pilot customer evaluation in a commercial product (NetWeaver BPM).

A question for future work regards lifecycle management, i.e., how to deal with changes in provider services or the delivery functions.

Acknowledgments. The work was in part funded through the German Federal Ministry of Economy and Technology (THESEUS, "01MQ07012") and the European Union (SUPER project, "IST FP6-026850").

References

[1] M. Altenhofen, E. Börger, and J. Lemcke. An abstract model for process mediation. In *ICFEM*, 2005.

[2] A. P. Barros and M. Dumas. The rise of web service ecosystems. *IT Professional*, 8(5):31–37, 2006.

[3] D. Berardi, D. Calvanese, G. D. Giacomo, and M. Mecella. Composing web services with nondeterministic behavior. In *IEEE ICWS*, 2006.

[4] E. Börger. The abstract state machines method for high-level system design and analysis. In *Current Trends in Applied Formal Methods, LNCS 1641*. Springer, 1999.

[5] G. Decker, H. Overdick, and M. Weske. Oryx – an open modeling platform for the BPM community. In *Demonstrations at BPM*, 2008.

[6] A. Friesen and J. Lemcke. Composing web-service-like abstract state machines (ASMs). In *WSCA*, 2007.

[7] J. Hoffmann, I. Weber, J. Scicluna, T. Kaczmarek, and A. Ankolekar. Combining scalability and expressivity in the automatic composition of semantic web services. In *IEEE ICWE*, July 2008.

[8] J. Küster, K. Ryndina, and H. Gall. Generation of business process models for object life cycle compliance. In *Proc. Conf. on Business Process Management (BPM)*, 2007.

[9] D. Oberle, S. Lamparter, A. Eberhart, and S. Staab. Semantic management of web services. In *ICSOC*, 2005.

[10] M. Pistore, P. Traverso, and P. Bertoli. Automated composition of web services by planning in asynchronous domains. In *ICAPS*, 2005.

[11] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web Service Modeling Ontology. *Applied Ontology*, 1(1):77–106, 2005.

[12] K. Ryndina, J. M. Küster, and H. Gall. Consistency of business process models and object life cycles. In *MoDELS'06*.

[13] M. Turner, D. Budgen, and P. Brereton. Turning software into a service. *Computer*, 36(10):38–44, 2003.

[14] I. Weber, J. Hoffmann, and J. Mendling. Beyond soundness: On the semantic consistency of executable process models. In *IEEE ECOWS*, 2008.

[15] Z. Zhou, B. Sapkota, E. Cimpian, D. Foxvog, L. Vasiliu, M. Hauswirth, and P. Yu. Process mediation based on triple space computing. In *ASWC*, 2008.