Static–Dynamic Integration of External Services into Generic Business Processes

Günter Preuner[1], Christian Eichinger[1], Michael Schrefl[2]

[1] Institut für Wirtschaftsinformatik – Data & Knowledge Engineering, Johannes Kepler Universität Linz, A-4040 Linz, Austria
[2] Avanced Computing Research Center, University of South Australia, Mawson Lakes SA 5095, Australia *

Abstract. Organizations usually handle their business cases by requesting external services that are integrated with internal procedures to a composite business process. External services may be atomic activities or, more often, arbitrarily complex processes.
In the simplest case, all business cases of a particular kind follow the same external services and internal procedures. Then, a composite service can be statically predefined without regarding peculiarities and requirements of single business cases. Such compositions ensure that processing business cases follows exact specifications. Yet in a highly dynamic environment, the actual processing may involve different services for each business case depending on its properties, such that appropriate services must be selected during runtime.
In this paper, the requirements of static and dynamic environments are naturally combined: Those parts of processing that are known in advance form the "frame" of processing. This frame comprises services that must be used in any case and comprises semantic rules that specify which requirements must be fulfilled by the services that are selected dynamically during processing. Correctness criteria define how static and dynamic composition can be appropriately integrated.

1 Introduction

Organizations perform their business cases not completely autonomously but use external services, which are provided by other organizations, together with intra-organizational procedures. External services may be atomic activities or — in the general case — complex processes.

An organization that requests services (referred to as *service requester*) from another organiation (the *service provider*) may process business cases in different ways: First, processes can be defined *statically,* i.e., completely in advance, which is suitable only if all parts of processing are known in advance. Second, a business process may not be defined at all in advance, but all activities may be selected for a business case *dynamically* during processing. Yet dynamic composition has

* Research performed at both universities within ARC Discovery Project DP0210654.

severe drawbacks since there is no possibility to pre-define processes in order to achieve that similar business cases are handled uniformly.

In this work, static and dynamic process definitions are combined to a "static–dynamic" approach: All parts of processing that are known in advance are integrated into a business process. Each part of processing that is not specified in advance is represented by a generic activity, which serves as a "place holder" for the dynamically selected service. Selection of a service during processing is considered as a work task of its own (which might be an arbitrarily complex process, again, e.g., a tendering procedure for the purchase of technical equipment). Such work tasks will be referred to as *build activities* and *build processes.*

For each generic activity, the pre-defined business process may specify by semantic rules (1) which kind of service is required to implement the generic activity (e.g., delivery of a computer system with hardware and software), (2) whether one service must be selected or services from different providers can be combined (hardware and software may be purchased from different suppliers), and (3) which case-specific constraints influence the selection of services (e.g., the system must be bought from a supplier who delivers within twelve months).

Static composition has been treated in detail in [1, 2]. In [5], an overall process is defined first; this process is split in a top-down fashion into several fragments, each one being assigned to a particular organization, which defines a more detailed private implementation for the assigned fragment. Similarly, [6] distinguishes public and private specifications, yet on a more technical level. In [7], global applications are defined with state-charts, comprising a set of activities, which are refined by (legacy) applications. Composition of independent services in a bottom-up fashion has been discussed in [8–10].

Dynamic specification of processes has been discussed in general in the realm of dynamic change of workflows, e.g., [3], and, recently, in the health-care domain [4] where generic processes are made concrete during processing by build activities. Although our work deals with the dynamic specification in an ontology-based, inter-organizational setting, concepts of build activities for generic processes are appropriate for our approach, as well. In [11, 12], a very dynamic approach of composition is introduced, where composition is guided by formal rules, yet without considering static composition aspects.

Our work goes beyond these approaches in that we combine static and dynamic composition to a static-dynamic approach as motivated above with precise correctness criteria. Build activities are first-class activities that select services according to the given semantic rules. Requested services distinguish activities (1) that are invoked by the requester and those (2) that are invoked by the provider, where execution can be observed by the requester.

Services are represented by ontologies, which appear to comprise the most adequate techniques for semantic composition of web services; cf., e.g., [13]. In addition, rules expressed upon these ontologies support the selection of services. Semantic matching of services has been discussed in detail in recent research [14–16]. We will build upon existing work and restrict our discussion to how ontologies can support our approach of composition. Hence, we neither enforce the use of a particular ontology nor introduce "yet another" ontology language; instead, any ontology that allows to classify services may be used.

We use *Object/Behavior Diagrams (OBDs)* [17] for the design of business processes. OBDs are considered appropriate for our approach since they have a proper formal semantics based on Petri Nets and their concept of behavior-consistent specialization [18] proved to be an adequate foundation for composition. Nevertheless, our approach can be applied to other models that support specialization, as well, like UML [19] with specialization as proposed in [20].

The remainder of this paper is structured as follows: Section 2 briefly introduces OBDs. Section 3 presents the service architecture for static-dynamic composition. Section 4 explains how ontologies and semantic rules support the dynamic selection of services. Correctness and construction of composed processes are presented in Sect. 5. Finally, the work is concluded in Sect. 6.

2 Modeling Services with OBDs

Object/Behavior Diagrams (OBDs) were introduced as a conceptual object-oriented design notation, where processes are represented by Behavior Diagrams [18]. Each Behavior Diagram is associated with an object class and defines the behavior of all instances of this class.

Behavior Diagrams consist of a set of *activities,* i.e., atomic units of work, a set of *states* in which objects may reside, and a set of *arcs*, which connect activities with states. Each activity has at least one pre-state and one post-state. *Initial* states have no incoming arcs and represent the virtual processing state of an object before its creation. Analogously, a state is *final* if no activity consumes from it. Further, Behavior Diagrams comprise labels in order to distinguish different aspects of processing. Labels are motivated by the analogy of copies of a paper form in traditional paper work, where different copies "flow" in a different way through the workflow. Analogously, *labeling properties* ensure that every label has one initial state, at least one final state, and resides in exactly one state or activity state at each point of time. For brevity, activities, states, and labels are commonly referred to as *elements.*

Example 1. Figure 1 shows LBD MOrder *(music order)*, which comprises activities, e.g., selectCDs, and states, e.g., ordered. Initial state α is depicted by dashed lines and is usually omitted in the graphical representation. Labels o and r represent the handling of the order and registering the CDs. Please ignore state symbol toOrder depicted with dashed borders for the moment.

The *life-cycle state (LCS)* of an object specifies which of its labels reside in which states. An activity can be invoked on an object if this object resides in all pre-states of the activity. After an activity has been completed, the object resides in all post-states of the activity. Different from Petri Nets, an activity does not "fire" automatically, but must be explicitly invoked. Further, we assume that the execution of activities takes time and distinguish *starting* and *completing* an activity. Between starting and completing an activity on an object, the object

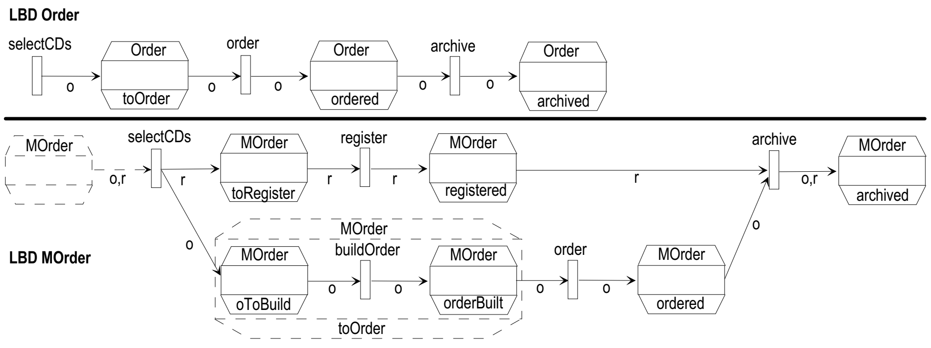


Fig. 1. LBDs Order and MOrder

resides in an implicit activity state named after the activity. A sequence of LCSs is called a *life-cycle occurrence (LCO).*

Example 2. Consider LBD MOrder: Each object resides initially in LCS $\{(\alpha, \text{o}), (\alpha, \text{r})\}$. Starting activity selectCDs leads to LCS {(selectCDs, o), (selectCDs, r)} and completing this activity leads to LCS {(oToBuild, o), (toRegister, r)}.

Specialization of LBDs was presented in [18], distinguishing *refinement,* i.e., elements of B are refined in B', and *extension,* i.e., additional elements are introduced in B' with new labels. One kind of specialization, namely *observation-consistent specialization,* means that the processing of an object in B' must be observable as a correct processing in B in that every possible LCO in B' must be a valid LCO in B if refined elements are considered unrefined and elements added by extension are ignored.

Example 3. LBD MOrder is an observation-consistent specialization of LBD Order (cf. Fig. 1): MOrder *refines* Order in that state toOrder is refined to oToBuild, buildOrder, and orderBuilt. Further, MOrder *extends* Order by the aspect of registering, represented by label r and several activities and states labeled with r. The generalization of LCS {(buildOrder, o), (toRegister, r)} in MOrder to {(toOrder, o)} is a valid LCS in Order.

3 Service Architecture for Static-Dynamic Composition

The composition approach is characterized by the following service architecture, summarized in Fig. 2. Schemes of services and ontological rules are depicted as rectangles, whereas arrows depict which schemes are derived from others.

Generic service. First, the *generic service* is defined, which comprises all aspects of processing that are independent of the properties of business cases being processed. Further, it specifies at which points of processing another service or

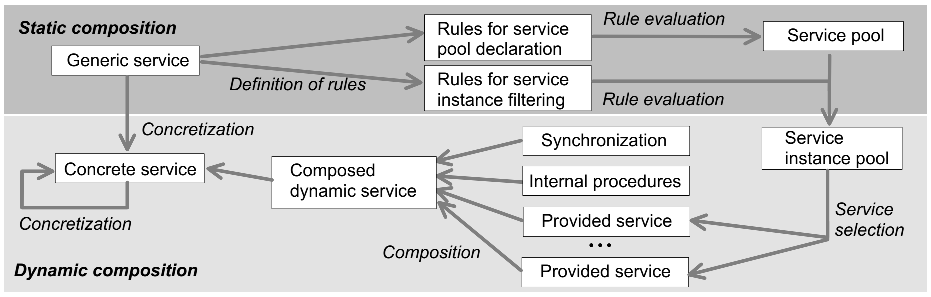


Fig. 2. Static-dynamic composition

combination of services can be dynamically introduced. The generic service is an LBD that comprises three kinds of activities:

1. *Regular activities* represent work tasks that are performed on business cases. They are either observable or invocable: *Observable activities* are executed by the provider, who allows the requester to *observe* their execution. *Invocable activities* are defined by the provider, but their invocation is delegated to the requester. Activities from internal procedures and synchronization activities (see below) are invocable since they must be invoked by the requester.
2. *Generic activities* cannot be executed but are "place holders" for services that are dynamically selected for a business case during processing.
3. *Build activities* are invoked by the service requester to replace generic activities by actual services. They are included in the process as first-class activities in order to enable the modeler of a process to predefine at which point of processing a service must be selected for a generic activity.

The generic service may be defined from scratch by the service requester or may result from integrating several external services with internal procedures. Such a static integration is useful if a particular set of services must be used for every business case and, thus, is not case-specific. For example, the service of a music trader could be integrated into the generic service if a library must buy all its CDs from this trader. The definition of a generic service by integrating services makes use of the approach presented in [1, 2]. Since the way how the generic service has been defined does not influence static-dynamic composition as presented in this paper, we will take the generic service as given.

Example 4. Suppose that a music library frequently buys music CDs from different music traders. Therefore it defines a (very simple) service that comprises activities that are executed internally (like selecting and registering music CDs) and a generic activity for ordering. The process is depicted by LBD MOrder in Fig. 3. Regular invocable activities are marked with tag *Inv*, the generic activity order is represented by a shaded activity symbol and is marked with tag *G*, activity buildOrder for building a concrete service for order is marked with *B* and annotated with the generic activity to be built.

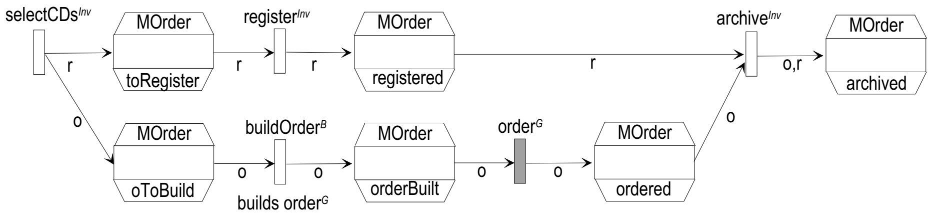


Fig. 3. Generic service of a library

Composed dynamic service. The composed dynamic service is a service that can be used to replace a particular generic activity. In the following, we use the phrase that a service or a combination of services *fulfills* a generic activity if the service or the combination of services provides all the functionality that is needed to realize the work task that the generic activity stands for. There are three possibilities of replacement, each one requiring a different set of services:

1. *Single service replacement:* There is one service selected that fulfills a particular generic activity.
2. *Coordinated service replacement:* There is a set of independent services that, when combined, fulfill the generic activity. The service requester coordinates the services by introducing synchronization elements, i.e., further activities, states, and labels.
3. *Coordinated service replacement with internal processing:* The service requester might complement external services with internal procedures if the former do not fully fulfill the generic activity since particular aspects of processing are missing. Synchronization elements coordinate the external services and the internal processes. Internal processes may include generic and build activities again, such that replacement can be performed recursively.

Figure 2 depicts the (most complex) third case. For more details on composing external services with internal processes, see Sect. 5. The build activity searches adequate services, decides on which kind of replacement is used, which services are selected, and — if necessary — coordinates the set of services with or without further internal processing.

Example 5. Suppose that the service of a music trader MT1-Order shown in Fig. 4 is selected for generic activity order: The music trader's service performs both payment and delivery. Hence, it fulfills generic activity order without any need for internal processing or synchronization.

Alternatively, another music trader with process MT2-Order could be selected whose service offers creation of an order and delivery. The service requester has to combine this service with a bank's service BT1-Transfer for money transfers. The services and their composition with internal activities are depicted in the bottom section of Fig. 4.

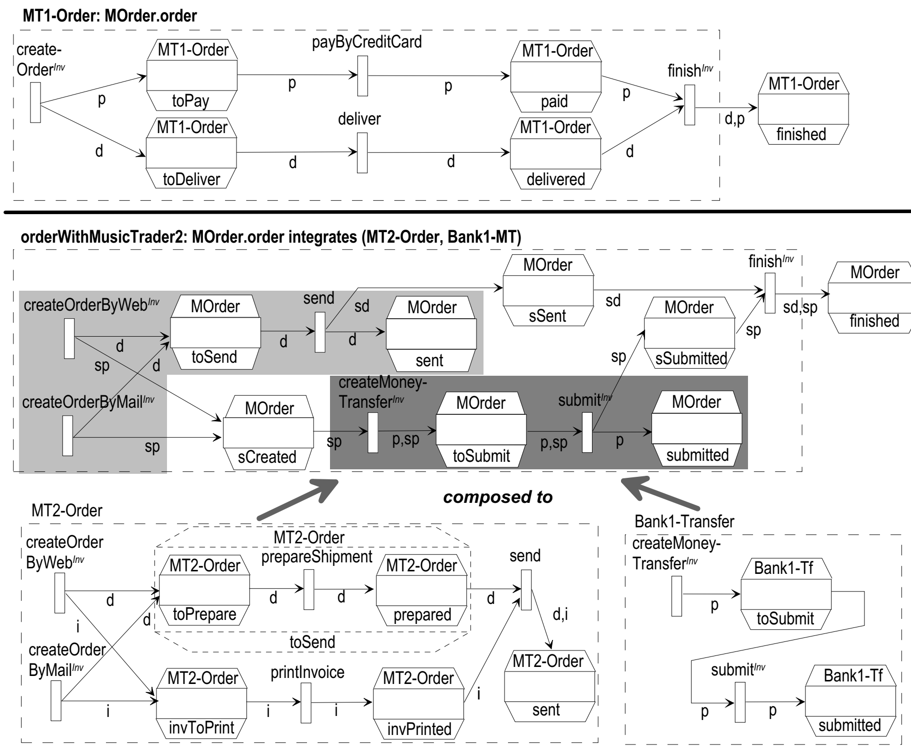


Fig. 4. LBDs MT1-Order and orderWithMusicTrader2

Concrete service. The *concrete service* is defined by replacing a generic activity by the dynamic service that has been defined by the build activity. Making services concrete is an iterative procedure since a composed service may include an arbitrary number of generic activities, which are not made concrete at the same time, and since a selected service may include generic activities again. For details on concretization see Sect. 5.

Rules for service pool declaration and service instance filtering. These rules refer to an ontology of services and define which functionality must be offered by the service to be selected dynamically and how a service is actually selected by matching the services' properties and the business cases' requirements. Semantic rules are defined for each generic activity in a twofold manner: Services that fulfill a generic activity fully or partially are determined by *rules for service pool declaration* and are stored in a *service pool*. Given a set of services in a service pool, *rules for service instance filtering* specify how all services or combinations of services are determined that comply with a business case's requirements. The appropriate services or combinations thereof are stored in the *service instance* *pool*. From there, one service or service combination is actually selected for replacing the generic activity. See Sect. 4 for details.

4 Semantic Service Discovery

Service instances capable of fulfilling a generic activity are identified in a two-step approach consisting of (1) *Service Pool Declaration* and (2) *Service Instance Filtering* which are backed by ontologies as described in this sections.

4.1 Service Pool Declaration

The *service pool ontology* underlying the service pool declaration classifies all external services needed to fulfill generic activities and specifies service-specific properties, the *service profile*. Furthermore, each service is described by an OBD. As ontology integration is not within the scope of this paper, we refer to existing approaches in the fields of ontology integration, ontology matching and service integration [14, 21, 15]. In our ontology, services are represented as instances of service classes and will be referred to as *service instances*.

Example 6. Figure 5 depicts the service pool ontology for services needed in our running example. For the sake of simplicity, the example ontology shows only service providers and omits the specification of products and services, which is usually part of any real world ontology. Furthermore, the range of properties has been omitted. Service instances are identified by a grey header.

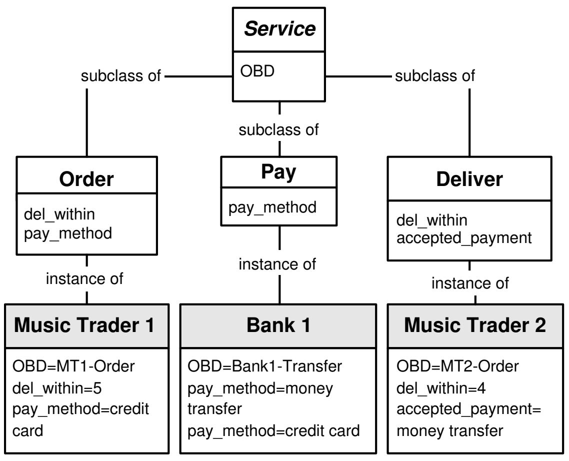


Fig. 5. Sample service pool ontology

Three different service pools are distinguished with respect to their contribution to the fulfillment of a generic activity:

1. *Full service pool:* The full service pool comprises service instances that fulfill the generic activity as a whole. These instances are candidates for a single service replacement.
2. *Partial service pool:* The partial service pool comprises service instances capable of fulfilling a specific part of the generic activity. These instances are candidates for a coordinated service replacement with internal processing.
3. *Compound service pool:* The compound service pool specifies combinations of service instances which are capable of fulfilling the generic activity as a whole. These instances are candidates for a coordinated service replacement.

Service pool members are specified by queries over the service pool ontology. These queries are attached to generic activities; their results are instances of the service pool ontology.

Executing queries over the service pool ontology requires an inference mechanism over ontologies. [22] gives an overview of ontology tools supporting inferencing and ontology building. But, as the definition of selection criteria for inferencing tools is not within the scope of this paper, we use Prolog syntax for describing rules and assume that service pool instances are expressed as facts, e.g. the rule order(X) specifies all instances of class Order and the rule del_within(X,Y) returns in Y the value of property "del_within" of instance X.

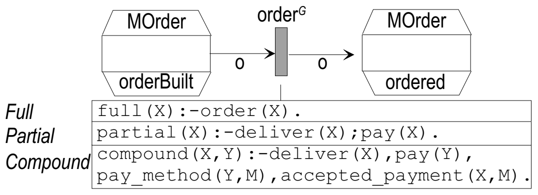
Example 7. Figure 6 extends the generic activity order introduced in Fig. 3 with service pool declarations. The service pools *full*, *partial* and *compound* are expressed as rules over the service pool ontology: full(X) retrieves all order services, partial(X) retrieves services that provide delivery or payment, and compound(X,Y) retrieves combinations of order and payment services where the payment service is accepted by the delivery service. The execution of the service pool computation is ordered as follows: (1) the full service pool, (2) the partial service pool, and (3) the compound service pool.

Applied to our example, these rules would return "Music Trader 1" as member of the full service pool, "Music Trader 2" and "Bank 1" as members of the partial service pool and "Music Trader 2, Bank 1" as member of the compound service pool as their payment methods match.

4.2 Service Instance Filtering

The service pool instances for building a concrete service of a generic activity are filtered according to business case's requirements. The requirements of a case are again captured by an ontology, the *process instance ontology.* Figure 7 shows the process instance ontology for our example order service.

Specific requirements are passed as parameters to parameterized queries operating on the service pools. These queries are attached to the build activities



Full	`full(X):-order(X).`
Partial	`partial(X):-deliver(X);pay(X).`

Compound	`compound(X,Y):-deliver(X),pay(Y),`
	`pay_method(Y,M),accepted_payment(X,M).`

Fig. 6. Service pool declaration for generic activity order

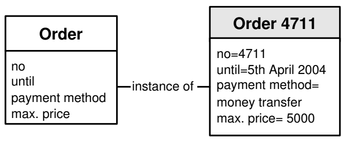
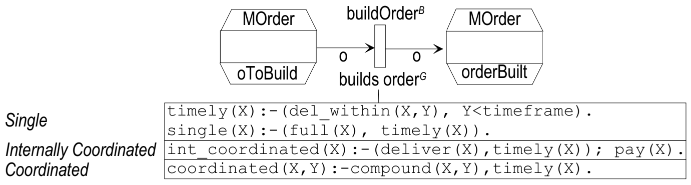


Fig. 7. Sample process instance ontology



Single	`timely(X):-(del_within(X,Y), Y<timeframe).`
	`single(X):-(full(X), timely(X)).`

Internally Coordinated	`int_coordinated(X):-(deliver(X),timely(X)); pay(X).`
Coordinated	`coordinated(X,Y):-compound(X,Y),timely(X).`

Fig. 8. Service instance filtering for generic activity order

of the respective generic activity. Upon rule evaluation, the parameters are substituted with property values of the actual business case. These rules are again divided into three pools, which comply with the replacement alternatives of composed dynamic services: *single instance pool* for single service replacement, *coordinated instance pool* for coordinated service replacement and *internally coordinated instance pool* for the coordinated service replacement with internal processing.

Example 8. Figure 8 visualizes the filtering rules that specify service instance pools for generic activity order (which has been introduced in Fig. 3). Rule single(X) retrieves all order services that can deliver in time, i.e., whose property “del_within” is less than the number of days left until “5th April 2004”. Rule int_coordinated(X) retrieves all payment services as well as all delivery services that satisfy the time constraint. Finally, coordinated(X,Y) retrieves all service combinations where the delivery service can deliver in time.

The evaluation of these rules on 2nd April 2004 would return the following service instance pools: single(X) containing no members as service “Music Trader 1” cannot fulfill the time constraint imposed by “Order 4711”, int_coordinated(X) containing “Bank 1” as it is member of class pay and “Music Trader 2” as it can fulfill the time constraint, and coordinated(X,Y) containing “Music Trader 2, Bank 1”.

When a build activity is activated for a particular business case, rules for service instance filtering are evaluated. The user then builds a sub-process fulfilling the generic activity for this business case with some or all of the returned service instances as described in the next section.

5 Correctness and Construction of Composed Services

This section presents the correctness criteria for composed dynamic services and concrete services (cf. Sect. 5.1). The construction of correct services is presented in Sect. 5.2. Hence, this section covers two steps in our architecture:

1. *Composition:* The definition of the composed dynamic service depends on the replacement strategy: There is no composition necessary for single service replacement. For coordinated service replacement, a set of external services must be composed; for coordinated service replacement with internal processing, external services and internal procedures must be integrated. Composition follows the approach of [2].

   Example 9. For fulfilling generic activity order, services MT2-Order from "Music Trader 2" and Bank1-Transfer from "Bank 1" are composed to service orderWithMusicTrader2 (cf. Example 8).

2. *Concretization:* Concretization of a service C to C' is considered a special case of composition: C resulted from an internal procedure and, possibly, a set of external service that have been statically integrated or integrated in previous concretization steps. During concretization from C to C', another service S is integrated with the services that have been integrated to C before. Hence, the concrete service is a composition of all services that are represented by C and S and the correctness criteria from [2] should hold again. Yet construction is different than in the first case since services are not composed in one step, but continuously during concretization.

   Example 10. In the running example, the generic service MOrder must be made concrete by integrating service orderWithMusicTrader2.

5.1 Correctness Criteria for Composition

During static composition, external services are composed together with internal procedures. Synchronization elements (i.e., activities, states, and labels) reflect data and control dependencies between activities from different services. In order to condense the presentation, the correctness criteria will be presented here in a simplified form; for details, the reader is referred to [2].

The composite service shall include all information that is — from the viewpoint of the requester — necessary for successful processing. Hence, all invocable activities are included, whereas observable details may be abstracted in order to reduce the complexity of the composite service. The composite service must obey the following correctness criteria with respect to the services:

1. *Well-formed synchronization and internal service:* Internal procedures and synchronization elements that are embedded in the composite service are conceptually considered as a "service", as well, which is embedded in the composite service with new, internal labels. Synchronization and internal services are well-formed if the restriction of the composite service to internal labels and all elements with at least one internal label is a correct LBD.

2. *Observability and invocability compliance:* An activity is invocable (observable) in the composite service if the related activities in the services are invocable (or observable, respectively). Invocable activities are neither abstracted nor omitted in the composite service.
3. *Observability consistency:* The processing of business cases in the services should be observable as a correct processing according to the composite service.
4. *Invocability consistency:* Invocability consistency requires that at any time when an invocable activity can be started in the composite service, i.e., whenever a business case resides in all pre-states of this activity, it must be possible to start the related activities in the external services, as well.

Example 11. Consider the services depicted in Fig. 4: Services MT2-Order and Bank1-Transfer define the processing of orders and of money transfers, respectively. These services are integrated to a composite service orderWithMusic-Trader2, which comprises all aspects of order processing. Elements in the composite service that are related to elements in the services are indicated by a grey background shading. All activities and states in orderWithMusicTrader2 that are labeled with sd or sp have been introduced as synchronization elements. They constitute a correct LBD. Elements toPrepare, prepareShipment, and prepared have been abstracted to state toSend (as indicated by the state symbol with dotted borders); the aspect of invoicing (cf. label i), has been omitted.

5.2 Construction of Correct Composite Services

The definition of a composed dynamic service from services in a bottom-up fashion is summarized first; then, construction of concrete services is introduced.

Construction of Composed Dynamic Services Construction follows the correctness criteria presented in Sect. 5.1. While conditions 1 and 2 are easy to check, checking criteria were introduced in [2] for conditions 3 and 4. Briefly, observability consistency is checked by the concept of observation-consistent specialization (cf. Sect. 2): Each external service must be an observation-consistent specialization of the composite service, when the latter is restricted to labels that correspond to labels of the external service. Further, no additional pre-states must be introduced for observable activities in the composite service since these pre-states are not considered by the service provider who starts the corresponding activities in his/her service. Invocability consistency is fulfilled if all pre-states of an invocable activity in an external service are represented as pre-states of this activity in the composite service.

Example 12. Consider once more the example in Fig. 4: Service MT2-Order is an observation-consistent specialization of orderWithMusicTrader2 if the latter is restricted to label d. orderWithMusicTrader2 introduces no new pre-states for observable activities and comprises all pre-states of invocable activities.

Construction of Concrete Services During concretization, a generic activity is replaced by a composed dynamic service. This replacement should not have any side effects, i.e., the behavior "outside" the generic activity should not be affected by the replacement. The concrete service C_{new} is constructed from a generic or another concrete service C_{old} as follows: Begin and end activities are determined in the composed dynamic service S. Begin activities are executed as the first activities in S, i.e., no other activity therein is executed before; analogously, end activities are executed last in S. A composed dynamic service S replaces a generic activity in that the begin and end activities are synchronized with C_{old}. The construction comprises the following steps:

1. *Determination of begin and end activities:* An activity of the composed dynamic service S is called a *begin activity* if all its pre-states are initial states. Analogously, an activity is an *end activity* if all its post-states are final states. Begin and end activities must be invocable since the requester must invoke them in order to initiate and finish a selected service.

   Example 13. Service orderWithMusicTrader2 in Fig. 4 comprises begin activities createOrderByWeb and createOrderByMail and end activity finish.

2. *Local refinement of generic activity:* The generic activity in C_{old} is refined (1) to one alternative generic begin (or end) activity for each begin (or end, respectively) activity from the composed dynamic service and (2) to one state, which is post-state of all generic begin activities and pre-state of all generic end activities. The resulting LBD is an observation-consistent refinement of the original one by mapping the new activities and state to the original generic activity. The resulting service is referred to as C_{oldRef}.

   Example 14. The local refinement of the generic activity order is depicted in Fig. 9 with two alternative generic begin activities, one generic end activity, and a state in between. Generic begin and end activities carry the same name as the respective activities in orderWithMusicTrader2.

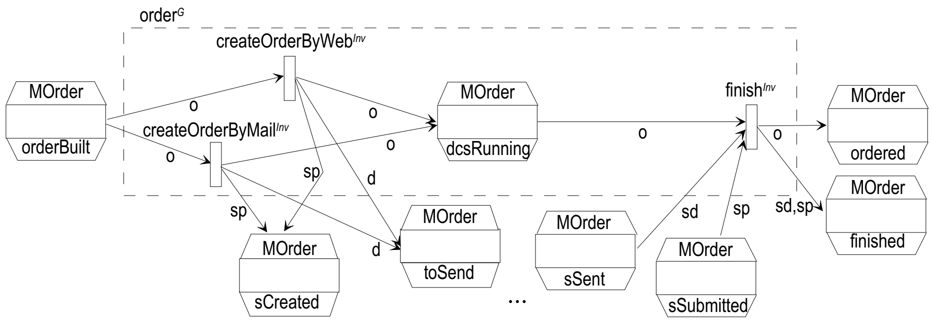


Fig. 9. Replacement of order

3. *Embedding of composed dynamic service:* The concrete service C_{new} results from C_{oldRef} and S in that each begin and end activity of S is synchronized with the corresponding generic begin and end activity of C_{oldRef}. Pairs of synchronized activities are represented by the same activity in C_{new}. All other elements of S are added in C_{new}. The set of labels of S is considered disjoint with the labels in C_{oldRef}.

   Example 15. Fig. 9 depicts the refinement of order for embedding service orderWithMusicTrader2. The embedded service is not depicted in detail as the figure spots on the begin and end activities.

The resulting concrete service C_{new} is an observation-consistent specialization of C_{oldRef} and fulfills the correctness criteria for composition (cf. Sect. 5.1): Assume that C_{old} and S are correct LBDs and are correctly combined, then internal aspects of processing and synchronization elements form correct LBDs in C_{new}. Observability and invocability compliance are fulfilled since invocability of activities is not changed. Observability consistency is fulfilled since S is embedded without change (which is a special case of specialization) and no new pre-states are introduced for observable activities. Invocability consistency is fulfilled since all pre-states of invocable activities are integrated.

6 Conclusion

We presented static-dynamic integration of services that naturally combines and extends existing approaches on service integration. The distinction between static and dynamic aspects both leaves enough flexibility for selecting appropriate services for a particular business case in a dynamic environment and guarantees at the same time that behavioral aspects known in advance are explicitly specified and are obeyed for each case. The distinction of static and dynamic aspects is appropriately complemented by semantic rules for service pool declaration and service instance filtering.

Future work will extend the approach in that generic *processes* may be introduced instead of generic activities. These generic processes might specify a frame for actual processing, which the actually selected service must fit into. For example, a generic process may specify that a music trader must deliver before it requests payment. This constraint can be specified by a generic activity for delivery followed by a generic activity for payment.

References

1. Preuner, G., Schrefl, M.: Behavior-Consistent Composition of Business Processes From Internal and External Services. In Proc. ER 2002 Workshops, Revised Papers. Springer LNCS 2784 (2003)
2. Preuner, G., Schrefl, M.: Requester-centered Composition of Business Processes from Internal and External Services. To appear in: Data & Knowledge Engineering (2004)

3. van der Aalst, W.: Generic Workflow Models: How to Handle Dynamic Change and Capture Management Information? In: Proc. 4th IFCIS Int. Conf. on Cooperative Information Systems (CoopIS), IEEE Computer Society Press (1999)
4. Browne, E., Schrefl, M., Warren, J.: Goal-Focused Self-Modifying Workflow in the Healthcare Domain. In: Proc. 37th Hawaii Int. Conf. on System Sciences (HICSS) – Track 6, IEEE Computer Society (2004)
5. van der Aalst, W.: Inheritance of interorganizational workflows to enable business-to-business e-commerce. Electronic Commerce Research **2** (2002) 195–231
6. Bussler, C.: The Application of Workflow Technology in Semantic B2B Integration. Distributed and Parallel Databases **12** (2002) 163–191
7. Eyal, A., Milo, T.: Integrating and customizing heterogeneous e-commerce applications. VLDB Journal **10** (2001) 16–38
8. Chiu, D., Karlapalem, K., Li, Q., Kafeza, E.: Workflow View Based E-Contracts in a Cross-Organizational E-Services Environment. Distributed and Parallel Databases **12** (2002) 193–216
9. Hull, R., Benedikt, M., Christophides, V., Su, J.: E-Services: A Look Behind the Curtain. In: Proc. 22nd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), ACM Press (2003)
10. Kafeza, E., Chiu, D., Kafeza, I.: View-based Contracts in an E-service Cross-Organizational Workflow Environment. In Proc. 2nd Int. Workshop on Technologies for E-Services (TES). Springer LNCS 2193 (2001)
11. Medjahed, B., Bouguettaya, A., Elmagarmid, A.: Composing Web services on the Semantic Web. VLDB Journal **12** (2003) 333–351
12. Zeng, L., Flaxer, D., Chang, H., Jeng, J.J.: PLM$_{flow}$ — Dynamic Business Process Composition and Execution by Rule Inference. In Proc. 3rd Int. Workshop on Technologies for E-Services (TES). Springer LNCS 2444 (2002)
13. Fensel, D., Hendler, J., Lieberman, H., Wahlster, W., eds.: Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential. MIT Press (2003)
14. Cardoso, J., Sheth, A.: Semantic E-Workflow Composition. Journal of Intelligent Information Systems **21** (2003) 191–225
15. Paolucci, M., Kawamura, T., Payne, T., Sycara, K.: Semantic Matching of Web Services Capabilities. In Proc. 1st Int. Semantic Web Conf. (ISWA). Springer LNCS 2342 (2002)
16. Sheshagiri, M., des Jardins, M., Finin, T.: A Planner for Composing Services Describes in DAML-S. In: Proc. Workshop on Web Services and Agent-based Engineering (WSABE). (2003)
17. Kappel, G., Schrefl, M.: Object/Behavior Diagrams. In: Proc. 7th Int. Conf. on Data Engineering (ICDE). (1991)
18. Schrefl, M., Stumptner, M.: Behavior Consistent Specialization of Object Life Cycles. ACM Trans. Software Engineering and Methodology **11** (2002) 92–148
19. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison-Wesley Object Technology Series. Addison Wesley (1999)
20. Stumptner, M., Schrefl, M.: Behavior Consistent Inheritance in UML. In Proc. 19th Int. Conf. on Conceptual Modeling (ER). Springer LNCS 1920 (2000)
21. Doan, A., Madhavan, J., Dhamankar, R., Domingos, P., Halevy, A.: Learning to match ontologies on the Semantic Web. VLDB Journal **12** (2003) 303–319
22. Corcho, O., Fernáandez-López, M., Gómez-Pérez, A.: Methodologies, tools and languages for building ontologies. Where is their meeting point? Data & Knowledge Engineering **46** (2003) 41–64