Modeling and Composing Service-Based and Reference Process-Based Multi-enterprise Processes

Hans Schuster, Dimitrios Georgakopoulos, Andrzej Cichocki, and Donald Baker

MCC, 3500 West Balcones Center Drive, Austin, Texas 78759
{schuster, dimitris, andrzej, dbaker}@mcc.com

Abstract. *Multi-enterprise processes* (MEPs) are workflows consisting of a set of activities that are implemented by different enterprises. Tightly coupled Virtual Enterprises (VEs) typically agree on abstract MEPs (*reference MEPs*), to which each enterprise contributes *single-enterprise processes* (SEPs) that implement and refine the activities in the reference MEP. On the other end of the spectrum, loosely coupled VEs use *service-based MEPs* that fuse together heterogeneous *services* implemented and provided by different enterprises. Existing process models usually couple activities with their implementation. Therefore, they cannot effectively support such MEPs. In this paper, we introduce a *Polymorphic Process Model* (PPM) that supports both reference process- and service-based MEPs. To accomplish this, PPM decouples *activity interface* from *activity implementation*, and provides process *polymorphism* to support their mapping. In particular, PPM determines activity types from the activity interfaces, permits activity interface subtyping, and provides for the mapping of MEP activity types to concrete implementations via interface matching. We illustrate that these key PPM capabilities permit the late binding and use of multiple activity implementations within a MEP without modifying the MEP at run time or enumerating the alternative implementation at specification time.

1 Introduction

A *Virtual Enterprise* (VE) appears like a traditional enterprise to its customers but provides services and products that rely on the core business processes and the resources of multiple constituent enterprises. The enterprises that comprise a VE may be participating in a long-term strategic alliance or collaborate only for the duration of one electronic commerce transaction. Just like traditional enterprises, VEs realize business objectives by defining and implementing (multi-enterprise) business processes. These are abstract process descriptions, consisting of a set of activities that must be performed by the members of the VE.

Multi-enterprise business processes are implemented by multi-enterprise workflow processes we refer to as *multi-enterprise processes* (MEPs). Unlike traditional workflow processes where all activities are implemented by the same enterprise, each enterprise in a VE implements only a subset of the MEP activities. Furthermore, the same MEP activity may be implemented by multiple enterprises, and the provider of the implementation of this MEP activity may vary in different instances of the same MEP. Finally, each participating enterprise may have multiple implementations of the same MEP activity. For example, a telecommunications service provider implements

B. Wangler, L. Bergman (Eds.): CAiSE 2000, LNCS 1789, pp. 247-263, 2000

an activity informing the customer that the service has been established. The provider may have multiple implementation alternatives for this activity, including notify the customer by phone, mailing a letter, sending a fax or email.

Current workflow models [10, 13, 16] couple activities with their implementation. This is similar to a procedure call in a traditional programming language. Therefore, when designing a process consisting of activities that have multiple implementations, the process designer has to choose one of the alternative implementations. However, this precludes the use of any other implementation of the same activity that may be more appropriate in another processes instance or another part of the process that repeats this activity. For example, the method of notifying a client that the requested service has been performed may depend on the client's location, time of the day, cost, etc. To permit alternative implementations the designers of traditional workflow processes often expand each activity to a subprocess that captures all possible implementation alternatives as separate activities and allows choice of implementation via decision activities, normal control, and data flow. This solution introduces additional activities in the original process and the structure of the resulting process will differ significantly from the business process it implements. Therefore, this solution complicates the maintenance of the business process as it evolves and additional activity implementations become available.

These problems are alleviated in VEs that use *reference processes.* Such processes typically capture standard business activities within a certain industry, such as telecommunications [15]. A reference process is public and its purpose is to specify how multiple enterprises can work together. Therefore, reference processes provide activity decomposition and coordination "guidelines" that improve the ability of enterprises to form effective VEs. At the same time, each participating enterprise maintains considerable flexibility in determining how to implement activities in a reference process. The implementation of such a reference MEP must be able to hook up the various heterogeneous implementations of its activities as provided by the participating enterprises. In situations where the enterprises that participate in a VE maintain their autonomy to a level that reference MEPs cannot be used, the participating enterprises usually provide *services* that are not targeted to a specific MEP, i.e., they are implemented and maintained in isolation. In this case, service-based MEPs must integrate these heterogeneous services, e.g., as proposed in [4].

In this paper we introduce a *Polymorphic Process Model* (PPM) that generalizes the process model presented in [4]. PPM supports both service- and reference process-based MEPs. To accomplish this, PPM decouples *activity interface* from *activity implementation*. This permits PPM activity types to be determined by the activity interfaces. Activity interfaces are modeled as state machines that include application-specific operations and states, and have input/output parameters. Application-specific operations provide an abstraction of an activity's behavior, while application-specific states provide an abstraction of the activity state resulting from the execution of the activity operations or internal processing. In addition, activity interfaces allow (abstract) activity types to be mapped to concrete activity implementations by matching their interfaces. To support such mappings PPM introduces process *polymorphism*. This includes a mechanism for activity interface subtyping and a mechanism for late binding that enables the use of multiple implementations within a process without modifying this process at run time or enumerating the alternative implementation at

specification time. These permit supporting MEPs in both reference process- and service-based VEs.

The rest of this paper is organized as follows: In the following section, we present examples of MEPs using a reference process and services and discuss key requirements for their modeling, implementation, and enactment. Section 3 introduces activity types and interfaces. The mapping of activity interfaces that provides the basis for developing interchangeable implementations of abstract activities is discussed in Section 4. Activity polymorphism is discussed in Section 5. Section 6 gives an overview on related work. Conclusions are presented in Section 7.

2 Multi-enterprise Processes

To illustrate key requirements in supporting *multi-enterprise processes* (MEPs), we describe two alternative MEPs for universal telecommunications service provisioning: *reference process-based MEPs* and *service-based MEPs*. We illustrate that these MEPs can satisfy the requirements of virtual telecommunications enterprises using contrasting business models. In particular, we first introduce a *reference process*-based MEP. Such MEPs are appropriate for tightly coupled VEs where participating enterprises internally employ business models and corresponding core processes (*single-enterprise processes*, or SEPs), that are designed to accommodate each other. This is accomplished by agreeing on abstract VE processes, or *reference MEPs*, to which each enterprise contributes SEPs that implement and refine its (abstract) activities. Reference process-based MEPs are appropriate for relatively long-lived VEs with cooperating partners that are willing to adapt their SEPs to implement the parts of the MEP reference processes they are supposed to perform. An example of reference process-based MEP is discussed in Section 2.1.

The contrasting solution to reference process-based MEPs are *service-based MEPs*. These are appropriate for loosely-coupled VEs, i.e., VEs consisting of enterprises that want to hide their internal SEPs from their partners in the VE and cannot (or choose not to) introduce changes to their SEPs. Therefore, service-based MEPs are appropriate for relatively short-lived VEs, such as those that last only for the duration of a single electronic commerce transaction, and VEs consisting of competitors that occasionally join forces. An example of service-based MEP is described in Section 2.2. In this paper we do not discuss VEs using MEPs that combine both reference processes and services. Such MEPS must address the combination of the problems described in this paper.

2.1 Reference Process-Based MEP

A reference MEP is a process that consists of abstract subprocesses, i.e., subprocesses that lack implementation. The abstract subprocesses and the MEP process itself are implemented by SEPs provided by the enterprises in the VE.

As an example of a reference MEP consider a universal telecommunications service provisioning process that allows clients to request local, long distance, wireless, and internet service from a *universal service provider*. This reference MEP is depicted

in Figure 1. It specifies that the process starts when a customer of the universal service provider requests a universal telecommunications service. Subprocess *Exchange Info* involves an operator collecting information from the customer, or a customer directly providing the information via a web browser. When sufficient customer data are collected, subprocess *Order Service* is performed to (1) verify that the information provided by the customer is accurate, and (2) create a corresponding universal service order record. When *Order Service* is completed, the top process starts *Provide Local Exchange Service*, *Provide Long Distance Service*, *Provide Wireless Service*, and *Provide Internet Service*.

These abstract subprocesses are implemented by SEPs provided by different enterprises, i.e., the local, long distance, wireless, internet, and universal service providers. When all selected subprocesses are completed, subprocess *Bill* is preformed to create a single bill for all telecommunications services in the universal service. Finally, *Care for Customer* involves a human operator who verifies that the provided service meets the customer needs.

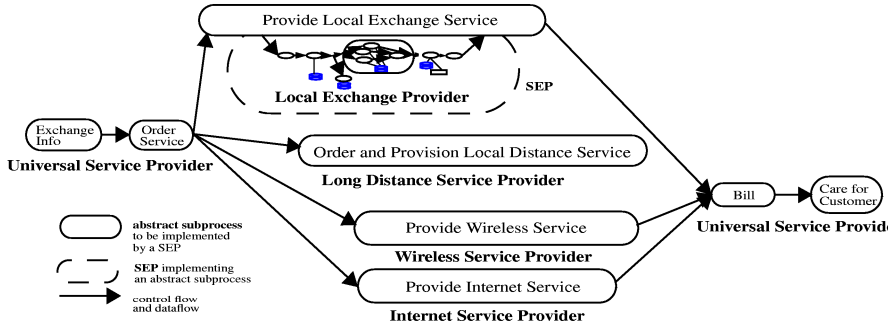

Fig. 1. Reference MEP for universal telecommunications service provisioning

The universal service provider in Figure 1 may be a distinct enterprise from the local, long distance, wireless, and internet providers. However, in our example VE the universal service provider may be any of the providers of the component services. In particular, the provider of the universal service is determined by the customer who decides which of enterprise to contact for universal service. All enterprises in our example VE have a top level SEP that implements the reference MEP in Figure 1, i.e., they can provision universal service. In addition, large VEs may have multiple enterprises that implement and provide the same component service.

Reference processes are emerging in both industry [15] and academia [9]. Industrial reference processes capture standard business processes within a certain industry branch, e.g., reference processes for the telecommunication industry are proposed by the TeleManagement Forum [15]. Reference processes developed by academic research, such as the process handbook project [9], also stay on a declarative business process level. Therefore, existing reference process concepts provide no solutions for implementation and enactment of reference process-based MEPs.

2.2 Service-Based MEP

A service-based MEP is a process that fuses together *services* provided by different enterprises. Such services may encapsulate SEPs implemented by commercial workflow management systems (WfMSs), CORBA servers, basic programs, legacy information systems, etc. However, from the perspective of the MEP that integrates them, services are black boxes that provide only a set of service operations to control them and determine their state.

To integrate a service, a service-based MEP must use *service activities*. These are service proxies that convert the operations and/or states of each service to the behavior of an activity that is incorporated in the MEP. In VEs with multiple providers of the same service, service-based MEP must provide abstract activities that can have multiple implementations. The execution of an abstract activity involves selecting one particular implementation during run-time, binding it to the abstract activity, and running it. An example of a service-based MEP for a universal telecommunications service provisioning is depicted in Figure 2.

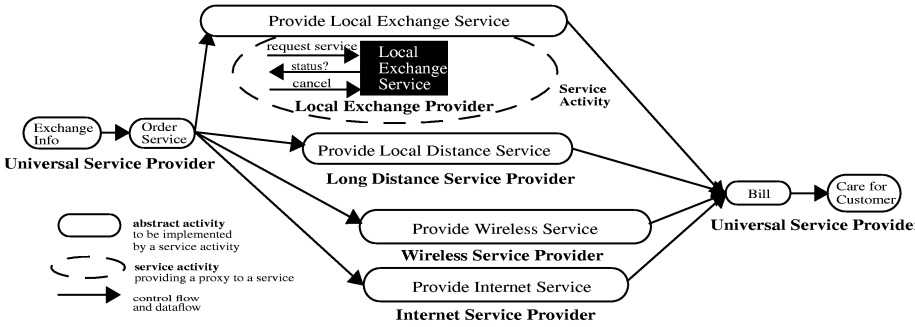

Fig. 2. Service-based MEP for universal telecommunications service provisioning

Service-based MEP must deal with service autonomy and heterogeneity. In reference-based MEPs, this problem is mitigated by the agreement of the participating enterprise to comply with reference MEPs. However, in service-based MEPs no such agreement exists. Furthermore, service designers typically have no direct knowledge of the service models and implementations used by other enterprises. Therefore, service-based MEP requirements include:

1. *service activities* that can effectively model and control heterogeneous services, and
2. activity *polymorphism*, i.e., ability to associate and bind service activities at various degrees of specialization to the MEP activities/processes they implement.

Modeling heterogeneous services involves enhancing traditional workflow activity types [16, 6] with *application-specific* operations and states. Controlling a service encapsulated in a service activity requires support for *conversations* between the activity that needs to control a service activity (the service *client)*, and service activity itself. Such conversations involve multiple interactions where clients perform multi-

ple invocations and receive intermediate results that may be used in further invocations. The *Local Exchange Service* illustrated in Figure 2 is a simplified service that provides operations to request local exchange service installation, determine its status, and cancel it. The service activity that serves as the proxy of the *Local Exchange Service* is a conversational activity. In particular, suppose that the *Order Service* in Figure 2 needs to query the *Local Exchange Service* to determine whether it failed to complete installing the service within the agreed time frame. In this case, the *Order Service* may cancel the *Local Exchange Service* or do necessary adjustments, e.g., notify the universal service customer. To support such conversation, the service activity for the *Provide Local Exchange Service* provides the following application-specific operations: *request service()*, *status()*, and *cancel()*. In addition, the service activity has application-specific activity states that indicate the progress, success, or failure of the service installation.

Conversational activities cannot be captured directly by existing service activities/proxies [13, 10] and existing process models [16, 6] which assume that activities are invoked once, enter their running state, and produce no data until they are completed or terminated (stopped). For example, if we use an existing process model to capture the cancellation of the local exchange service we must add a *Cancel Local Exchange Service* activity after the *Provide Local Exchange Service* activity in the MEP in Figure 2. Therefore existing process models have the following limitations in capturing service request and cancellation: (1) the cancellation activity can only be invoked after the Local Exchange Service request operation is completed, i.e., cancellation of a service request in progress is not possible, and (2) the *Local Exchange Service* and corresponding service operations in Figure 2 cannot be modeled as a single traditional activity. The model we propose in this paper, i.e., PPM, allows implementation of activities in a MEP with services that require conversations as described above. PPM also provides activity polymorphism that permits subtyping and late binding of service activities to MEP activities/processes. PPM activity interfaces, implementations, and polymorphism are discussed in the following sections.

3 Activity Types and Interfaces

To allow application modeling, process models, e.g., [16], provide *types* for activities, resources, and dependencies. Activity types are either basic activity types or processes (composite activity) types. These types are used to develop application models that are instantiated during enactment.

To support the requirements of service- and reference process-based MEPs discussed in Section 2, we have developed a *Polymorphic Process Model* (PPM) that decouples activity types from activity implementations. Just like in CORBA™ [11] and Java™ [14], the type of an activity in PPM is determined by the activity interface. Therefore, to avoid confusion by using the term “activity” with different meanings, in the rest of this paper we distinguish the activity *type* (represented by an *activity interface*), the *activity implementation*, and the use of an activity within a process called *activity variable*. This is analogous to object oriented programming languages, e.g., Java™ [14], which also distinguish object interfaces and *abstract classes* (corresponding to activity types), concrete *classes* (corresponding to activity implementations), and typed variables (such as activity variables) that may hold

tions), and typed variables (such as activity variables) that may hold references to object (activity) instances.

In PPM a process activity type consists of an activity interface, activity variables, resource variables, and dependency variables. Activity variables represent the subactivities of a process. Resource variables describe the resources needed during process execution. Dependency variables define the data and control flow rules for the subactivities of the process. In contrast, basic activity types are restricted to an activity interface and resource variables.

PPM requires that both activity types and implementations have *activity interfaces.* In PPM, activity interfaces are used to map activity types to activity implementations The interface of an activity type captures its implementation requirements at an abstract level, while the interface of an implementation captures the capabilities of the implementation at a concrete level. In PPM, both abstract and concrete activity interfaces are state machines that capture:

- application-specific activity states and operations that cause activity state transitions
- input and output parameters

These are discussed in Sections 3.1 and 3.2.

3.1 Activity State Machine Types

In the initial paragraphs of Section 3, we noted that in PPM activity types are determined by their interfaces that consist of an *activity state machine type* (ASMT) and activity input/output. An ASMT determines the possible *activity states* for instances of the respective activity interface and corresponding *state transitions*. Formally, an ASMT is a tuple (*S*, *T*) whereby *S* is the set of states and *T* is the set of transitions $t = (s_1, s_2)$, where $s_1, s_2 \in S$ and $t \in T$. State transitions may be caused by invoking explicit *activity operations.* In the following sections we describe ASMTs in detail.

Activity Operations. Activity operations, such as start and terminate, drive the execution of activities and the corresponding state transitions in a process. Our PPM supports two types of explicit activity operations: *generic operations* and *application-specific operations*. *Generic operations* include the application-independent operations that are provided by the WfMC standard: *create(), start(), pause(), resume(), finish(),* and *terminate()*. PPM complements these standard operations with a generic *state_change_of_subactivity()* operation. This operation is discussed further in Section 4.1.

In addition to generic operations, PPM permits *application-specific operations* that are defined by the process designer as needed to model a specific activity type. They represent control operations that are available on running activities. For example, an activity that implements ordering of local exchange service may offer application-specific operations to check the *status* of the order, *add* additional items to the order (e.g., call waiting, caller ID, etc.), and *remove* items from the order. Application-specific activity operations are supported only by few WfMSs, e.g., CMI [4] and the MOBILE WfMS [7].

Activity states. To enable interoperability with standard WfMC activities [16], PPM provides a set of generic activity states, i.e., *Uninitialized*, *Ready*, *Running*, *Suspended*, and *Closed* with *Completed* or *Terminated* substates. These generic activity states and state transitions constitute the *generic ASMT* depicted in Figure 3.

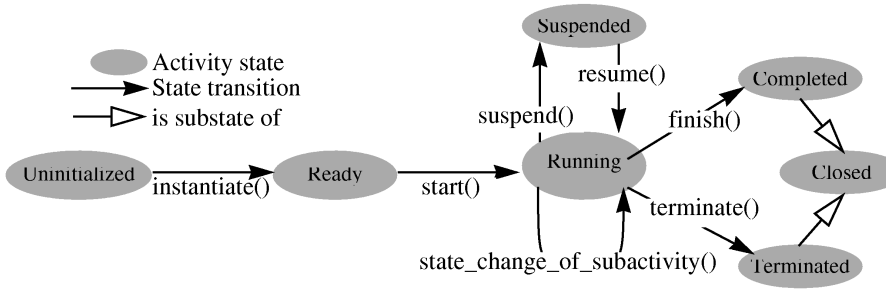

Fig. 3. Generic Activity State Type

In addition to generic states, PPM supports application-specific states. For example, consider the ASTM of the *Provide Local Exchange Service* activity in the telecommunications MEP in Figure 1. Suppose that this activity has two application-specific states, *Provisioning* and *Fulfilling*. Figure 4 shows the corresponding ASMT. The *Provide Local Exchange* service activity enters its *Provisioning* state when it starts running and the service provider starts allocating resources needed to provide the service, such as lines and slots in telecommunication switches. The activity state changes to *Fulfilling* when all necessary resources have been allocated and the activation of the service begins.

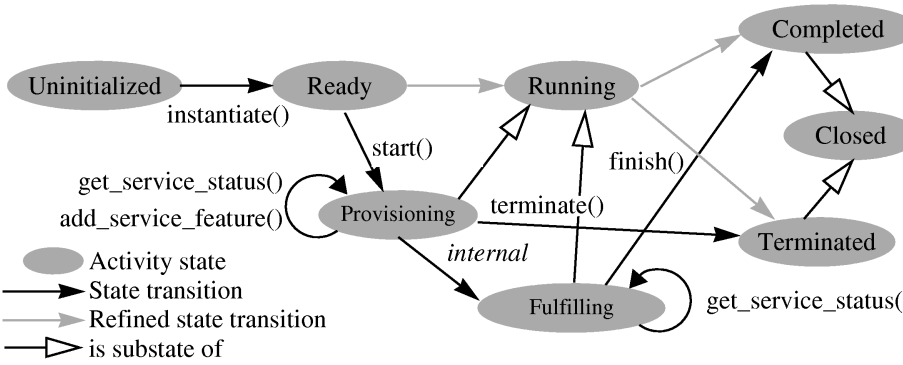

Fig. 4. Example: ASMT for Provide Local Exchange Service activity

The *Provisioning* and *Fulfilling* states in Figure 4 are a refinement of the *Running* state. The *Provisioning* to *Fulfilling* transition is labeled as *internal* in Figure 4. *Internal* is a special operation indicating that this state transition is not under the control of the process enactment system and it is caused solely by the activity implementation.

Internal state changes need special treatment in a process specification (this is discussed further in Section 4.2). After service installation has been confirmed, the activity enters its *Completed* state. An order can be canceled as long as it has not been fulfilled. Cancellation is represented by a transition from the *Provisioning* state to the *Terminated* state. To keep the example simple we omit activities necessary to cancel a service order.

Activity state machine subtyping. The ASMT shown in Figure 4 is a subtype of the generic ASMT in Figure 3. The refined ASMT is called a *subtype* of the original ASMT (referred to as the *father type)*. When an activity state is refined, all state transitions of the refined activity state have to be replaced by transitions involving one or more of its substates. Such transitions are depicted as grey arcs in Figure 4. Suppose that *C* is a subtype of an ASMT *F*. *C* is a *valid* activity state machine subtype (ASMsubT) of *F* if and only if: either a transition happens between two substates of the same parent state, or there is a corresponding transition between the parent states in the father type.

Formally, the valid states and transitions of an activity state subtype can be defined by considering that for each subtype *C* of a ASMT father type *F* there is a function *refine_state*. This function defines how the states of the father type *F* are refined by the subtype *C* by mapping each state of *F* to the corresponding set of C's states. Thus, assuming that S_F is the set of *F*'s states, S_F is the domain of *refine_state*. Similarly, if S_C is the set of *C*'s states, S_C is the range of *refine_state*.

Condition for valid states: To enable an unambiguous state refinement, each state of *C* must be a refinement of exactly one state of *F*, i.e., $\forall s_1, s_2 \in S_F$: ($refine\_state(s_1) \cap refine\_state(s_2) \neq \emptyset \Leftrightarrow s_1 = s_2$) AND $S_C = \bigcup_{s \in S_F} refine\_state(s)$

Conditions for valid transitions: Assuming that T_C and T_F are the sets of transitions in *C* and *F*, respectively, and *C* is a subtype of *F*, one of the following must hold for each transition $(s_1, s_2) \in T_C$:

- both s_1 and s_2 are refinements of the same state in *F*, i.e., $\exists s \in S_F$: $s_1 \in refine\_state(s)$ AND $s_2 \in refine\_state(s)$,
- or there is a corresponding transition (s_a, s_b) in the father type *F*, i.e., $\exists s_a, s_b \in S_F$: $(s_a, s_b) \in T_F$ AND $s_1 \in refine\_state(s_a)$ AND $s_2 \in refine\_state(sb)$.

These conditions on building activity types ensure that application-specific extensions can always be generalized in a meaningful way to a father type.

3.2 Input/Output Parameters

In addition to an ASMT, an activity interface includes the activity input and output parameters. The combination of ASMTs and parameters is required by PPM to define the activity type. PPM adopts the existing data types from standard workflow process models [16]. In particular, in PPM the *input (output) parameters* of an activity *A* are a set of typed data variables, denoted as I_A (O_A). When an activity is started its input parameters are assigned with values as specified by the dataflow dependencies of the enclosing process. In traditional workflow models, activity output is produced when

the activity ends. PPM extends this by permitting output parameter values to be made available when the activity enters a specific state. As a consequence, activities can output data while they are still running.

Formally, the *output behavior* of an activity interface *A* can be defined as a function ob_A: $O_A \rightarrow S_A$ that maps each output parameter *o* of *A* to the state *s* of *A* that produces *o* (S_A is the set of states of *A*). Therefore, $ob_A(o) = s$ means that the value of the output parameter *o* is available after the activity has reached the state *s* for the first time. Therefore, if an activity writes to an output parameter after the activity has already reached the state in which this parameter becomes available, readers of the output parameter will read the result of the latest write. For example the invoice amount of an order is different after an additional item has been added to the order.

4 Activity Implementation and Control

Activity implementation in PPM mainly involves mapping abstract activities to SEPs and/or services that implement them. To map abstract activities (basic activities or subprocesses) to concrete processes and services, we introduce the notion of *concrete* and *abstract* activity types. An activity type is *concrete* if it has a corresponding implementation, and all state transitions in its ASMT are labeled with operations supported by this implementation. An *abstract* activity type does not have a corresponding implementation, and may contain one or more unlabeled state transitions in its ASMT. The distinction between abstract and concrete activity types is necessary to decouple the activity interface from the activity implementation. For example, consider the abstract *Provide Local Exchange Service* activity in both Figure 1 and Figure 2. Consider that this activity has the abstract ASMT illustrated in Figure 4. The abstract *Provide Local Exchange Service* may be implemented by a SEP (e.g., as in the reference MEP in Figure 1) or by a service activity (e.g., as in the service-based MEP in Figure 2). These alternative implementations of *Provide Local Exchange Service* are interchangeable at run-time.

In the following sections we discuss how abstract activity types are implemented by mapping them to concrete activity types. Section 4.1 discusses abstract activity implementation by a process (e.g., a SEP). Section 4.2 presents the implementation of an activity by a service activity and the implementation of service activities themselves.

4.1 Mapping Abstract Activities to Concrete Process Activities

The mapping of an abstract activity type to a concrete SEP depends on whether the SEP provides generic or application-specific operations in its concrete ASMT.

Mapping generic operations. In the following paragraphs we consider the mapping of an abstract MEP activity to a concrete SEP type that provides only generic operations. If an abstract MEP activity is implemented by a concrete SEP, each operation of the abstract activity is implemented by a concrete subactivity in the SEP. This is accomplished by mapping the operations and empty transition labels in the ASMT of the abstract activity to generic operations of the concrete ASMT.

For example, consider the abstract *Provide Local Exchange Service* activity whose ASMT is illustrated in Figure 4. This abstract MEP activity defines *get_service_status()* and *add_service_feature()* operations that query the service status and add additional service features, such as call waiting. Each of these operations is implemented by a subactivity in the concrete SEP whose ASMT depicted in Figure 5. This is accomplished by mapping the operations *get_service_status()* and *add_service_feature()* of the abstract ASMT to appropriate *state_change_of_subactivity()* (or shorter *sc_of_sub()*) operations. In particular, while the concrete process is in the *Provisioning* state, additional items may be added to an order by invoking *sc_of_sub*(*add_Item*, *any*). The keyword *any* indicates that any state change of the *add_item* subactivity is permitted in this state of the concrete process. Assuming that the *add_Item* subactivity has been enabled by the normal control flow, *sc_of_sub*(*add_Item*, *any*) causes the start of the *add_Item* subactivity.

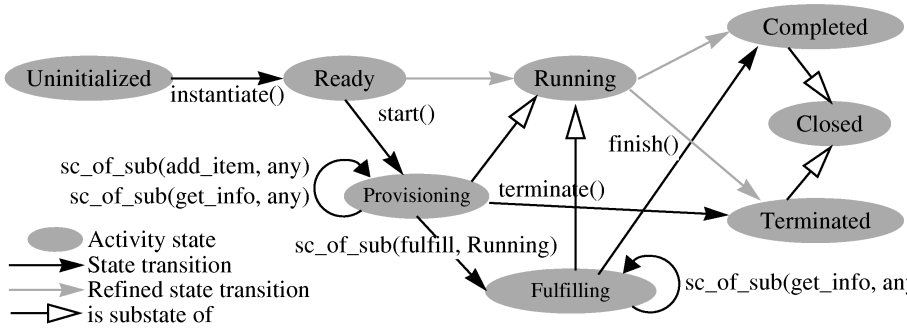

Fig. 5. Example: Generic ASMT type of Provide Local Exchange Service activity

Mapping application-specific operations. Unlike generic activity operations, such as *create()* and *start()*, application-specific operations must be specified by the process designer. In contrast to methods in object-oriented programming languages and distributed object systems, application-specific activity operations cannot be implemented by arbitrary code. In process models such as PPM the process subactivities and their dependency net, is a lower level specification that constitutes the process implementation. Since application-specific activity operations provide an abstraction of this implementation, each application-specific process activity operation is implemented by a *script* that invokes one or more activity operations on the subactivities of this process.

As an example, consider an abstract *Provide Local Exchange Service* activity with the ASMT illustrated in Figure 4. Suppose that this abstract activity is implemented

by a concrete process that includes two application-specific operations: *info()* and *item()*. Assume that the ASTM of this concrete process is depicted in Figure 6.

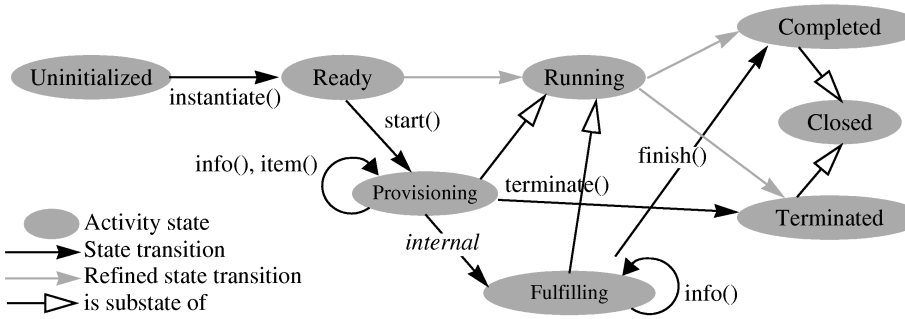

Fig. 6. Example: Application-specific ASMT of Provide Local Exchange Service activity

In this example, the application-specific operations *info()* and *item()* in Figure 6 are implemented by subactivities of the concrete *Provide Local Exchange Service* process. In particular, *info()* and *item()* are implemented by the *get_local_status* and *add_local_item* subactivities and the integration script illustrated in Figure 7.

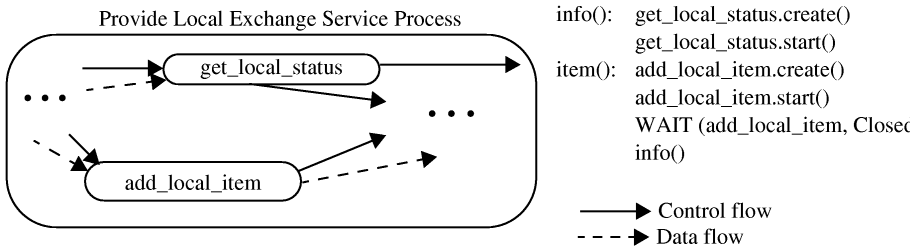

Fig. 7. Example: Fragment of a Provide Local Exchange Service Process

The *info()* and *item()* operations of the concrete ASMT are shown on the right side of Figure 7. *Info()* just creates and starts an instance of the *get_local_status* activity. *Item()* starts an instance of the *add_local_item* activity, waits for its termination, and then invokes the *info()* operation to inform the caller about the outcome. Note the scripts do not move data between the activities. Data flow is covered by the process net.

4.2 Mapping Abstract Activities to Concrete Service Activities

In the following paragraphs we present how an abstract activity is implemented by a concrete service activity, and then we discuss the implementation of the service activity itself.

Mapping abstract activities to concrete service activities. If the abstract *Provide Local Exchange Service* activity in Figure 4 is implemented by an external service, the abstract activity is implemented by a service activity that by definition has no subactivities. In this case, the *get_service_status()* and *add_service_feature()* operations of the abstract activity are implemented by service-provided operations, say *info()* and *item()*, that retrieve status information and add additional items, e.g., *item(CallWaiting)*. Figure 6 shows the concrete ASMT of such a service activity.

Implementing concrete service activities. As we discussed in Section 2.2, s*ervice activities* are proxies that convert the operations and/or states of each service to the behavior of an activity that is incorporated in the MEP. The implementation of an activity interface by a service activity is conducted by implementing of a corresponding *service activity proxy program (SAPP)*. Note, that services that support only a single *start()* invocation degenerate to basic or program activities [6, 16].

In contrast to activity interfaces and process specifications, which are descriptive specifications, the SAPP is to be coded in a traditional programming language. The service activity program uses the process system's API to access its input and output, to trigger internal state transitions, and to publish its activity operations to the process system. Hence, the SAPP is also an advanced wrapper program [12] for integrating external software systems. The deployment of a real program for this purpose is necessary, because the SAPP has to deal with the (possibly proprietary) communication protocols and formats of the software system and translates them to the format used by the process management system. However, generic SAPPs are provided to deal with traditional basic activities that have only generic states and operations.

By permitting application-specific states and operations, the service activities are far more powerful than SWAP [13] and the Workflow Management Facility [10]. Both of these propose a variant of a proxy mechanism that permit only the generic states and operations defined by the WfMC standard [16]. Therefore, they are not capable of capturing application semantics of services.

4.3 State-Dependent Control and Data Flow

PPM provides control flow *dependencies*/*transitions* that are similar to those provided by traditional WfMSs, e.g., [6], and the WfMC standard [16]. PPM extends the traditional notion of control flow transitions to permit application-specific states to be used in control flow. Such control flow transitions are denoted as $(av_{source}, s_{source}) \rightarrow (av_{target}, s_{target})$. They indicate that a state transition of an activity av_{target} to the s_{target} state is permitted after activity av_{source} enters state s_{source}. To ensure that the PPM dependencies can be enforced, only application-specific extensions of the *Running* state may be enabled by PPM control flow transitions. Additionally, transitions to *Running* substates that may be entered by *internal* operations (state transitions) are not allowed, because such transitions cannot be controlled by the execution system.

Just like control flow, PPM extends the traditional data flow dependencies to consider the states of the involved activities. In particular, in a traditional dataflow dependency $(av_{source}, out_{source}) \rightarrow (av_{target}, in_{target})$ the value of the output parameter out_{source} of an activity av_{source} flows to the input parameter in_{target} of an activity av_{target} after av_{source} is

closed (ends). PPM extends the semantics of a (av_{source}, out_{source}) → (av_{target}, in_{target}) dataflow dependency by allowing the data to flow as soon as av_{source} has reached the $ob_{source}(out_{source})$ state (*ob* was defined in Section 3.2). To avoid inconsistencies in the dependency net, we adopt the strategy that dataflow has to follow control flow [8]. That is, for each dataflow dependency (av_{source}, out_{source}) → (av_{target}, in_{target}) there must be a control flow path from av_{source} to av_{target} that starts at av_{source} from:

- state $ob_{source}(out_{source})$, or
- a state that is reached later than $ob_{source}(out_{source})$ in av_{source}*'s* state machine.

State-dependent control and dataflow are powerful primitives. In the activity scenario used in the previous sections, these dependencies enable the provisioning activity to output service activation information as soon as it has reached the *Fulfilling* state. A process that uses this activity can initiate further activities (using state-dependent control flow) as soon as this info becomes available.

5 Activity Polymorphism

Activity Polymorphism is permitting an activity variable to (1) use (i.e., hold references to) activity instances that have more specialized type than the type of the activity variable, and (2) use different implementations for the same activity variable at process execution time. Polymorphism allows flexibility in matching and using activity types and activity implementations provided by different enterprises.

In Section 5.1, we present the PPM solution for subtyping/specializing activity interfaces, by focusing on input/output subtyping and ASMT subtyping. In Section 5.2, we introduce activity placeholder. This primitive adds late binding that is needed to implement the selection of an activity implementation at runtime.

5.1 Activity Interface Subtyping

This extends activity state machine subtyping in Section 3.1 by considering activity inputs and outputs. Analogously to programming languages an activity type should be regarded as subtype A_C of a father activity type A_F if instances of A_C show at least all the behavioral characteristics that instances of A_F show. This ensures that wherever an instance of A_F is expected an instance of A_C can be used instead. However, A_C may show additional behavior that cannot be directly observed at A_F as long as this additional behavior can be generalized to a behavior of A_F. In terms of activity interfaces as defined in Section 3 this means that an activity interface A_C with activity operations aop_C, activity state type $asmt_C$, and input/output parameters (I_C, O_C, ob_C) is an activity interface subtype of A_F with activity operations aop_F, activity state type $asmt_F$, and in/output parameters (I_F, O_F, ob_F) if:

1. $asmt_C$ is an activity state subtype of $asmt_F$
2. $I_C \subseteq I_F$
3. $O_C \supseteq O_F$
4. $\forall\ v \in O_F$: $ob_F(v) = ob_C(v)$ or there is no path in the state machine of $asmt_C$ where $ob_F(v)$ or a substate of $ob_F(v)$ can be reached before $ob_C(v)$ is reached.

Condition (1) ensures that the observable states of the child type can be generalized to the states of the father type. This is guaranteed by the definition of activity state subtypes. Condition (2) prevents an interface subtype from expecting more input parameters than the father type. This way, a process expecting an instance of the father type provides also sufficient input for any subtype. Condition (3) states that a subtype has to provide at least the output expected to be produced by the father type. The subtype may have additional outputs that are discarded by someone expecting an instance of the father type. Condition (4) places an additional restriction on the output behavior of activity subtypes. Because a process can rely on an activity variable of type A_F to produce output in a particular state (this is needed by state dependent dataflow discussed in Section 4.1), an activity subtype must not produce its output later than the father type, i.e., in a state that can be reached after the output producing state of the father type.

Please note that there is no mandatory condition on A_C*'s* and A_F's activity operation sets aop_C and aop_F. If we assume that the process enactment system provides a worklist that informs each client about the operations currently available on a specific work item, a human client and also a flexible software agent that acts as a client can perform the right operations on every activity. However, if static programs are used as clients that cannot analyze the operation information on a work item and rely on the father's operation set aop_F, $aop_C = aop_F$ must hold.

5.2 Activity Placeholders

Activity placeholders allow for runtime assignment (late binding) of an activity implementation to a given activity variable. The type of the placeholder is determined by its interface. The placeholder is used instead of a concrete activity implementation in the declaration of an activity variable within a process. An activity placeholder has a *selection policy* attached, which is used at runtime to select the actual activity type for the placeholder's activity variable. To ensure the consistency of the process, within which an activity placeholder is used, the set of possible activity types from which the selection policy chooses one is the set of subtypes of the placeholders activity interface type.

Activity placeholders and the corresponding selection policies can be used in many application scenarios. For example, a generic reference process can be configured to use the activity implementations available in a given enterprise by providing a selection policy that plugs in the enterprise's implementation for each activity interface. If multiple enterprises provide implementations for an activity interface, the selection policy may use a broker to choose the implementation that offers the best quality of service [4]. In application areas where the process designer does not have the expertise to chose a particular activity implementation, e.g., in crisis mitigation applications where only expert process participants know which concrete tests to perform, the selection policy may permit such a participant to select which implementation to use [5].

6 Related Work

Separation of interface and implementation as well as interface subtyping and polymorphism are well-known concepts in programming languages, e.g., Java™ [14], and distributed (object) systems, e.g., CORBA [11]. They are particularly useful in heterogeneous environments because they enable the client of an object to be implemented and maintained independently from the implementation(s) of an object. We have adopted these ideas into the process world. Although object-oriented programming languages provide useful abstractions, they are not suitable for process modeling, since they do not include types and classes for capturing and implementing processes. Implementing such types and classes requires considerable effort, as indicated by current WfMS.

To the best of our knowledge current work in workflow management and process models does not provide abstraction mechanisms as the ones presented in this paper. The process handbook project [9] has the notions of inheritance and specialization of processes. However, this approach stays on a declarative business process level, i.e., implementation and enactment of the proposed concepts stays open in [9]. Van der Aalst et al. [1, 2] propose an inheritance mechanisms for process implementations with the objective to deal with dynamic change and evolution in workflows. Both approaches do not separate activity interfaces and implementation, however it is essential to our approach.

7 Conclusion

In this paper we presented a novel model for multi-enterprise processes (MEPs), namely, the Polymorphic Process Model (PPM). The main advantage of this model lies in its ability to support both service-based MEPs and MEPs using a reference process specification. PPM accomplishes this task by separating the activity interfaces from activity implementations, and defining the activity interfaces in terms of activity state machines and its input/output parameters. That in turn leads to the ability of using multiple implementations for the same activity in MEPs without the need for modifications of the specification or the running process. Together with strict rules of activity type subclassing, these characteristics allow for seamless integration of services and processes provided by multiple independent enterprises.

A prototype of a process management system based on a subset of this model has been successfully implemented in MCC's Collaboration Management Infrastructure System [3]. A set of sample applications, including health care-related crisis management processes, military operation coordination, and telecommunication provisioning service, have been designed, implemented, and demonstrated, proving that the PPM fulfills its promise.

However, not all aspects of the PPM are fully developed yet. For example, the runtime choice of a particular service to be executed as the resolution of a placeholder is a demanding problem. We have done some initial work in this area, but many problems considered crucial for the success of Virtual Enterprises (whose core business is choosing and composing external services) are left to be solved. Key remaining problems include the semantic description of the services, the description of the quality of

the services (necessary for a meaningful service comparison), and the development of service contracts beyond the activity types we discussed in this paper.

References

1. van der Aalst, W.M.P: Generic Workflow Models: How to handle dynamic change and capture management information. In*: Proc. of the Fourth IFCIS Conf. on Cooperative Information Systems (CoopIS'99)*, Edinburgh, Scotland, September 1999.
2. van der Aalst, W.M.P. ; Basten, T. ; Verbeek, H.M.W. ; Verkoulen, P.A.C. ; Voorhoeve, M.: Adaptive Workflow: An Approach Based on Inheritance. In: Ibrahim, M. ; Drabble, B. (Eds.): *Proc. Workshop Intelligent Workflow and Process Management: The New Frontier for AI in Business*, 16th. Int. Joint Conf. on Artificial Intelligence (IJCAI'99). Stockholm, Sweden, August 1999.
3. Collaboration Management Infrastructure, *http://www.mcc.com/projects/cmi*, 2000.
4. Georgakopoulos, D. ; Schuster, H. ; Baker, D. ; Cichocki, A.: Managing Process and Service Fusion in Virtual Enterprises. In: *Information Systems, Special Issue on Information Systems Support for Electronic Commerce*, **24**(6), 1999.
5. Georgakopoulos, D. ;Schuster, H. ; Cichocki, A.; Baker, D. : Collaboration Management Infrastructure in Crisis Response Situations. In*: Proc. 16th Int. Conference on Data Engineering (ICDE'2000)*, San Diego, March 2000.
6. *IBM FlowMark - Managing Your Workflow*. Version 2.3, IBM, 1996.
7. Jablonski, S. ; Bußler, C.: *Workflow Management - Modeling Concepts, Architecture and Implementation*. International Thomson Computer Press, 1996.
8. Leymann, F. ; Altenhuber, W.: Managing Business Processes as an Information Resource. In: *IBM Systems Journal*, 33(2), 1994.
9. Malone, T.W. ; Crowston, K. ; Lee, J. ; Pentland, B.: Tools for Inventing Organizations: Toward a Handbook of Organizational Processes. In: *Proc. of the 2nd IEEE Workshop on Enabling Technologies Infrastructure for Collaborative Enterprises*, Morgantown, WV, April 20-22, 1993.
10. Object Management Group: *Workflow Management Facility*. Revised Submission, OMG Document Number bom/98-06-07, July 1998.
11. Object Management Group: *The Common Object Request Broker: Architecture and Specification*. Revision 2.3.1, October 1999.
12. Schuster, H. ; Jablonski, S. ; Heinl, P. ; Bußler, C.: A General Framework for the Execution of Heterogeneous Programs in Workflow Management Systems. In: *Proc. First IFCS Int. Conf. on Cooperative Information Systems (CoopIS 96)*, Brussels, 1996.
13. *Simple Workflow Access Protocol (SWAP). http://www.ics.uci.edu/~ietfswap/*, 1999.
14. Sun Microsystems: *Java™. http://java.sun.com/*, 2000.
15. TeleManagement Forum: *http://www.tmforum.org/*, 1999.
16. Workflow Management Coalition: Interface 1: Process Definition Interchange Process Model. Document Number TC-1016-P, Document Status – Version 1.1, October 1999.