Adaptive and Dynamic Service Composition in *eFlow*

Fabio Casati, Ski Ilnicki, LiJie Jin, Vasudev Krishnamoorthy[1], and Ming-Chien Shan

Software Technology Lab
Hewlett-Packard Laboratories, 1U-4A
1501 Page Mill Road
Palo Alto, CA, 94304
{casati,ilnicki,ljjin,shan}@hpl.hp.com

Abstract. E-Services are typically delivered point-to-point. However, the e-service environment creates the opportunity for providing *value-added, integrated services,* which are delivered by composing existing e-services. In order to enable organizations to pursue this business opportunity we have developed *eFlow*, a system that supports the specification, enactment, and management of *composite* e-services, modeled as processes that are enacted by a service process engine. Composite e-services have to cope with a highly dynamic business environment in terms of services and service providers. In addition, the increased competition forces companies to provide customized services to better satisfy the needs of every individual customer. Ideally, service processes should be able to transparently adapt to changes in the environment and to the needs of different customers with minimal or no user intervention. In addition, it should be possible to dynamically modify service process definitions in a simple and effective way to manage cases where user intervention is indeed required. In this paper we show how *eFlow* achieves these goals.

1 Introduction and Motivations

In recent years the Web has become the platform through which many companies communicate with their partners, interact with their back-end systems, and perform electronic commerce transactions. Today, organizations use the Web not only as an efficient and cost-effective way to sell products and deliver information, but also as a platform for providing *services* to businesses and individual customers. Examples of e-services include bill payment, customized on-line newspapers, or stock trading services. As Web technologies continue to improve, allowing for smaller and more powerful web servers, and as more and more appliances become web-enabled, the number and type of services that can be made available through the Internet is likely to increase at an exponential rate.

---

[1] Now with Rightworks corp., 31 N. Second St., suite 400, San Jose, CA, USA. email: vasu@rightworks.com

B. Wangler, L. Bergman (Eds.): CAiSE 2000, LNCS 1789, pp. 13-31, 2000

Today, services are typically delivered point-to-point. However, the e-service environment creates the business opportunity for providing *value-added, integrated services*, which are delivered by composing existing e-services, possibly offered by different companies. For instance, an *eMove* composite service could support customers that need to relocate, by composing truck rental, furniture shipments, address change, and airline reservation services, according to the customer's requirements.

In order to support organizations in pursuing this business opportunity we have developed *eFlow*, a platform for specifying, enacting, and monitoring composite e-services. Composite services are modeled as business processes, enacted by a service process engine. *eFlow* provides a number of features that support service process specification and management, including a powerful yet simple service composition language, events and exception handling, ACID service-level transactions, security management, and monitoring tools.

Unlike "traditional" business processes, which are mostly executed in a predictable and repetitive way, composite services delivered through the Internet have to cope with a highly dynamic environment, where new services become available on a daily basis and the number of service providers is constantly growing. In addition, the availability of many service providers from different countries increases the competition and forces companies to provide customized services to better satisfy the need of every individual customer. These two characteristics of the e-service environment impose demanding requirements on a system that supports the development and delivery of composite services.

In order to stay competitive, service providers should offer the best available service in every given moment to every specific customer. Clearly, it is unfeasible to continuously change the process to reflect changes in the business environment, since these occur too frequently and modifying a process definition is a delicate and time-consuming activity. Ideally, service processes should be able to transparently adapt to changes in the environment and to the needs of different customers with minimal or no user intervention. Furthermore, it should be possible to dynamically modify service process definition in a simple and effective way to manage cases where user intervention is required, for instance to handle major changes in the environment or to cope with unexpected exceptional situations.

This paper shows how *eFlow* supports the definition and enactment of *adaptive* and *dynamic* service processes. We illustrate how the *eFlow* model enables the specification of processes that can automatically configure themselves at run-time according to the nature and type of services available on the Internet and to the requests and needs of each individual customer. We then present the dynamic change features provided by *eFlow*, that allow a great flexibility in modifying service process instances and service process definitions, enabling changes to every aspect of a process. Since dynamic process modification is a very powerful but delicate operation, one of our main goal has been to define very simple modification semantics, so that users can have a clear understanding of the effects of a modification. Prior to applying the changes, *eFlow* will enforce *consistency rules*, to avoid run-time errors resulting from the modifications, as well as *authorization rules*, to guarantee that only authorized users perform the modifications.

2 Overview of *eFlow*

This section presents an overview of the *eFlow* process model. We only present basic concepts that are needed in order to illustrate its adaptive and dynamic features. The interested reader is referred to [5] for details about the model and the implementation.

In *eFlow*, a composite service is described as a process schema that composes other basic or composite services. A composite service is modeled by a graph (the flow structure), which defines the order of execution among the nodes in the process. The graph may include *service*, *decision*, and *event* nodes. Service nodes represent the invocation of a basic or composite service; decision nodes specify the alternatives and rules controlling the execution flow, while event nodes enable service processes to send and receive several types of events. Arcs in the graph may be labeled with transition predicates defined over process data, meaning that as a node is completed, nodes connected to outgoing arcs are executed only if the corresponding transition predicate evaluates to true. A *service process instance* is an enactment of a process schema. The same service process may be instantiated several times, and several instances may be concurrently running.

Fig. 1 shows a simple graph describing a composite service that helps customers in organizing an award ceremony. In the figures, rounded boxes represent invocations of basic or composite services, filled-in circles represent the starting and ending point of the process, while horizontal bars are one of *eFlow* decision node types, and are used to specify parallel invocation of services and synchronization after parallel service executions.

The semantics of the schema is the following: when a new instance is started, service node *Data Collection* gathers information regarding the customer and his/her preferences and needs. Then, the *Restaurant Reservation* service is invoked, in order to book the restaurant and select the meals for the banquet. This node is executed first, since the characteristics of the selected restaurant (e.g., its location and the number of seats) affect the remainder of the service execution, i.e., the organization of the ceremony. Then, several services are invoked in parallel: the A*dvertisement* service prepares a marketing campaign to advertise the ceremony, the *Invitation* service proposes a choice of several types of invitation cards and delivers them to the specified special guests, while the *Registration* service handles guest registrations and payments. Finally, the *Billing* service is invoked in order to present a unified bill to the organizing customer. All services can be either basic services (possibly provided by different organizations) or composite services, specified by *eFlow* processes.

Service nodes can access and modify data included in a *case packet*. Each process instance has a local copy of the case packet, and the *eFlow* engine controls access to these data. The specification of each service node includes the definition of which data the node is authorized to read or to modify.

The *eFlow* model also includes the notion of *transactional regions*. A transactional region identifies a portion of the process graph that should be executed in an atomic fashion. If for any reason the part of the process identified by the transactional region cannot be successfully completed, then all running services in the region are aborted and completed ones are compensated, by executing a service-specific compensating action. Compensating actions may be defined for each service or may be defined at the region level. For instance, by enclosing the Advertisement, Registration, and Invitation services in a transactional region, and by providing compensating actions

for each of these services (or one compensating action at the region level), we are guaranteed that either all of the services are executed, or none is.

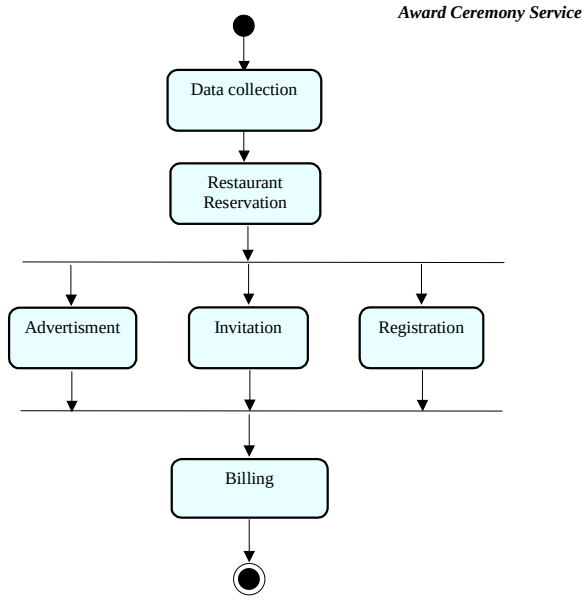

Fig. 1. Ceremony service process definition

Transactional regions may also include the specification of different isolation modes, that prevent data read or modified by nodes in the regions to be accessed by services that are outside the transactional region.

Process instances are enacted by the *eFlow engine*. The main function of the engine is to process messages notifying completion status of service nodes, by updating the value of case packet variables accessed by the service node and by subsequently scheduling the next node to be activated in the instance, according to the process definition. The engine then contacts the service broker in order to discover the actual service (and service provider) that can fulfill the requests specified in the service node definition, and eventually contacts the provider in order to execute the service.

The engine also processes *events* (either detected by the *eFlow* event monitor or notified by external event managers), by delivering them to the requesting event nodes. Notifications of occurred events and of service node completions are inserted into two separate transactional, First-in-First-Out queues (see Fig. 2). The engine extracts elements from the queues and processes them one by one. *eFlow* does not specifies any priority between the queues, but it does guarantee that every element in the queues is eventually processed. Finally, the engine logs every event related to process instance executions (to enable process monitoring, compensation, and to support dynamic process modifications) and ensures process integrity by enforcing

transactional semantics and by compensating nodes executed within transactional regions in case of failures.

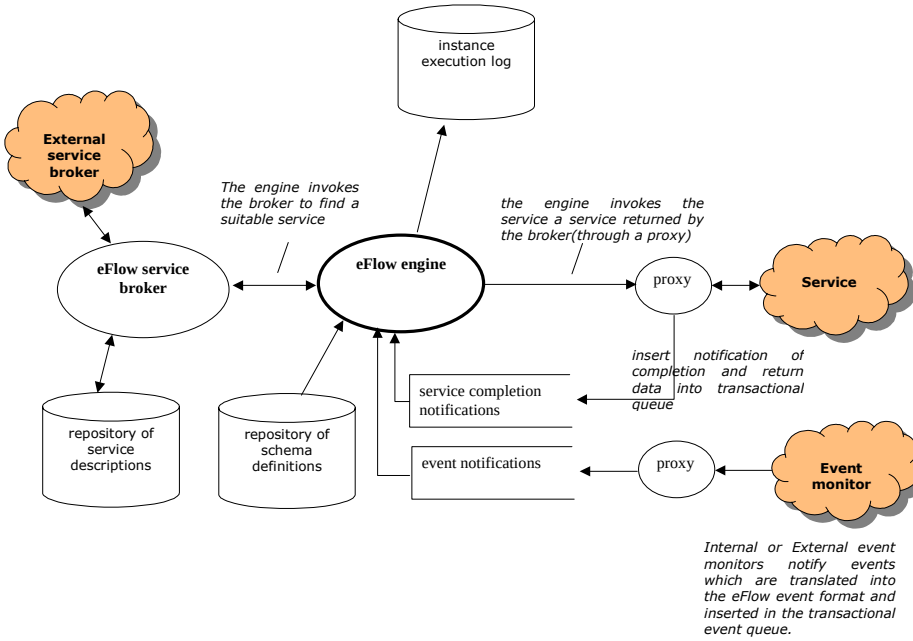

Fig. 2. The eFlow engine processes events and notifications of service completions in order to schedule service node executions

Like most Internet-based services, the *Award Ceremony* service provided by the *OneStopShop* company is executed in a highly dynamic environment. For instance, providers will continue to improve their e-services, and new providers may enter the market while some of the existing ones may cease their business. In addition, new types of e-services that can support the organization of an award ceremony may become available, such as renting of mega-screens and cameras, live broadcast of the ceremony over the Internet, or selection of trained personnel such as an anchorman. In the remainder of the paper we will show how *eFlow* addresses these challenges in order to allow service designer to provide composite services that naturally adapt to changes in the environment with minimal user intervention, that can be customized to fit the needs of every customer, and that are able to cope with unexpected exceptional situations.

3 Adaptive Service Processes

In order to manage and even take advantage of the frequent changes in the environment, service processes need to be *adaptive*, i.e., capable of adjusting themselves to changes in the environmental conditions with minimal or no manual intervention. *eFlow* provides several features and constructs to achieve this goal.

These include *dynamic service discovery*, *multiservice nodes*, and *generic nodes*. In the following we present an overview of these features.

3.1 Dynamic Service Discovery

A service node represents the invocation of a basic or composite service. Besides defining the data that the node is allowed to read and modify, and possibly a deadline to manage delays in service execution, a service node specification includes the description of the service to be invoked. For instance, within the *Advertisement* service node, we may specify that *eFlow* should invoke the *e-campaign* service offered by the *GreatAdvert.com* provider. While useful in some situations, such a static service binding is often too rigid, since it does not allow to:

- select the appropriate service depending on the customer's requirements: for instance, some customers may prefer a low-cost e-mail campaign, while other may prefer advertisements via TV, radio stations, or web sites;
- decouple service selection from the process definition: different service processes may require an advertisement service, and the selection criteria may need to be defined at the company level rather than at the composite service level;
- dynamically discover the best currently available service that fits the need of a specific customer.

To cope with the characteristics of the Internet environment, *eFlow* provides an open and dynamic approach to service selection. The service node includes the specification of a *service selection rule*, which can have several input parameters (defined by references to workflow variables). When a service node is started, the *eFlow* engine invokes a *service broker* that will execute the specified rule and return the appropriate service. Service selection rules are defined in a service broker-specific language, such as XQL if e"speak [4] is used as the service broker.

eFlow only requires that the rule returns an XML document which includes the definition of input and output data, the URI used to contact the service, billing and payment information, and a priority value used to select a specific service when several services are returned by the rule (choice among services with the same priority is non deterministic). Mapping between service node input/output data and the parameters of the invoked service is performed by a *mapping function*, specified as a set of string pairs `<case packet variable name, service variable name>`. A mapping function must be defined for a <service node, service description> pair before the service can be invoked in the context of the service node.

eFlow users can replace the default broker and plug-in the service broker that best fits their needs. Plugged-in brokers are not even required to access the service repository: they can dynamically discover services by contacting other external brokers or service advertisement facility, in order to get the most up to date information about available services and their characteristics.

Service selection rules will be then defined in the language supported by that broker, and can include arbitrary service selection policies. Plugged-in brokers must either present to the engine the same (simple) interface of the default one, or an adapter must be interposed between the engine and the broker to map requests and

responses. In addition, if service brokers dynamically discover services not stored in the service description repository, they must also return a mapping function that allows the mapping of service node input/output data to service parameters.

3.2 Multiservice Nodes

In some composite service processes there is the need of invoking multiple, parallel instances of the same type of service. For instance, a restaurant reservation brokering service may request rates and availability to several restaurants that provide on-line access to these information.

In order to allow the specification of these kinds of process semantics, *eFlow* includes the notion of *multiservice* node. The multiservice node is a particular kind of node that allows for multiple, parallel activation of the same service node.

The number of service nodes to be activated is determined at run time in one of the following ways:

1. It can be determined by the number of service providers able to provide a given service. For instance, for the award ceremony service, we may want to contact all restaurant in the San Francisco Bay Area that can host a specified number of guests.
2. It can be equal to the number of elements in a case packet variable of type list. In this case each service node instance receives one and only one of the list items as input parameter. The value of such item will affect service selection and execution. For instance, a list may include a set of customers of different nationalities for which we want to check their credit history. The number of service nodes that will be instantiated within the multiservice node will be equal to the number of customers, and each node will focus on one customer. A service selection rule will be executed for each service node to be activated; the rule can have the customer's data as input parameter, in order to select the appropriate credit check service for each customer, for instance depending on the customer's nationality.

An important part of a multiservice is the specification of when the multiservice can be considered completed and the flow can proceed with the successor service node. In most cases, the flow can proceed only when all invoked services have been completed. However, in other cases, there is no need to wait for all service instances to be completed, since the multiservice goal may have already been achieved before. For instance, suppose that we want to verify a customer's credit with several agencies: if our acceptance criteria is that all agencies must give a positive judgment for the customer to be accepted, then as soon as one agency gives a negative opinion we can proceed with service execution, without waiting for the completion of the other services, which may be canceled. The multiservice termination is specified by a condition, checked every time one of its service nodes terminate. If the condition holds, then the successor of the multiservice is activated and services in execution are canceled. An example of termination condition for the credit check example could be `Rejections.length>0`, where Rejections is a variable of type `ListOf (String)`, and length is an attribute common to every list variable that contains the number of elements in the list. Fig. 3 shows a sample specification of a multiservice node in *eFlow*. The specification includes the reference to the service node to be

instantiated (multiple times) as part of the multiservice node, as well as the activation and termination conditions.

```
<MULTISERVICE_NODE id="check_customers_credit">
       <NAME> Check Customers' credit </NAME>
       <SERVICE_NODE id="check_single_customer_credit" />
       <DESCRIPTION>  Multiservice node that checks the credit
                      history of several customers in parallel
       </DESCRIPTION>
       <ACTIVATION mode="by_variable" varref="customers_list" />
       <TERMINATION> rejections.length>0 </TERMINATION>
</MULTISERVICE_NODE>
```

Fig. 3. Specification of a multiservice node in *eFlow*

3.3 Dynamic Service Node Creation

An important requirement for providers of Internet-based services is the ability of providing personalized services, to better satisfy the needs of every individual customer.

While the service process depicted in Fig. 1 may be suited for some customer, other customers might need additional services, such as rental of video/audio equipment or the hiring of trained personnel to work with such equipment. At the same time, some customers may not need the services offered by the *Award Ceremony s*ervice process. For instance, they may not need an advertisement service or they may provide for it by themselves. Clearly, it is practically unfeasible to foresee all possible combinations of services which may be needed by each customer and to define a process for each potential type of customer. Besides, this would imply a very high maintenance cost, especially in the e-service environment where new types of services become available on a daily basis.

To cope with these demanding needs, *eFlow* supports the dynamic creation of service process definitions by including in its model the notion of *generic service node*. Unlike ordinary service nodes, generic nodes are not statically bound or limited to a specific set of services. Instead, they include a configuration parameter that can be set with a list of actual service nodes either at process instantiation time (through the process instance input parameters) or at runtime. The parameter is a variable of type `ListOf(Service_Node)`. The specified services will be executed in parallel or sequentially depending on an *executionMode* attribute of the generic service node.

Generic nodes are resolved each time they are activated, in order to allow maximum flexibility and to cope with processes executed in highly dynamic environments. For instance, if the generic node is within a loop, then its configuration parameters can be modified within the loop, and the node can be resolved into different ordinary service nodes for each loop of the execution. Notice that generic nodes are different from multiservice nodes: multiservice nodes model the activation of a dynamically determined number of instances of the *same* service node, while generic nodes allow the dynamic selection of different service nodes.

```
<GENERIC_NODE id="award_ceremony_services">
   <NAME> Award Ceremony Services </NAME>
   <SERVICE_NODE_POOL> Ceremony Service Pool </SERVICE_NODE_POOL>
   <DESCRIPTION> Placeholder for service nodes related
                 to a ceremony service,to be executed in parallel
   </DESCRIPTION>
   <SERVICE_SELECTION_VAR> SelectedServices</SERVICE_SELECTION_VAR>
   <EXECUTION_MODE mode="parallel" />
</GENERIC NODE>
```

Fig. 4. Sample XML description of a generic service node in *eFlow*

4 Dynamic Service Process Modifications

While adaptive processes considerably reduce the need for human intervention in managing and maintaining process definitions, there may still be cases in which process schemas need to be modified, or in which actions need to be taken on running process instances to modify their course. Process modifications may be needed to handle unexpected exceptional situations, to incorporate new laws or new business policies, to improve the process, or to correct errors or deficiencies in the current definition. We distinguish between two types of service process modifications:

- *Ad-hoc changes* are modifications applied to a single running service process instance. They are typically needed to manage exceptional situations that are not expected to occur again, such as the unavailability of a restaurant that had been booked for a ceremony.
- *Bulk changes* refer to modifications collectively applied to a subset (or to all) the running instances of a service process. For instance, suppose that an advertisement company on which many ceremony advertisement campaigns relied upon goes out of business. This situation can affect many instances, and it is practically unfeasible to separately modify each single instance. Bulk changes may also be needed when a new, improved version of a process is defined. If, for instance, a new law forces a modification of a process, then running instances will need to respect the new constraints as well.

4.1 Ad-hoc Changes

Ad-hoc changes are modifications applied to a single, running process instance. *eFlow* allows two types of ad-hoc changes: modifications of the process schema and modifications of the process instance *state.* In the remainder of this section we show how *eFlow* supports both type of changes.

Ad-hoc Changes to the Process Schema

eFlow allows authorized users to modify the schema followed by a given service process instance. The modifications are applied by first defining a new schema (usually by modifying the current one) and by then *migrating* the instance from its current schema (called *source* schema) to the newly defined one (called *destination*

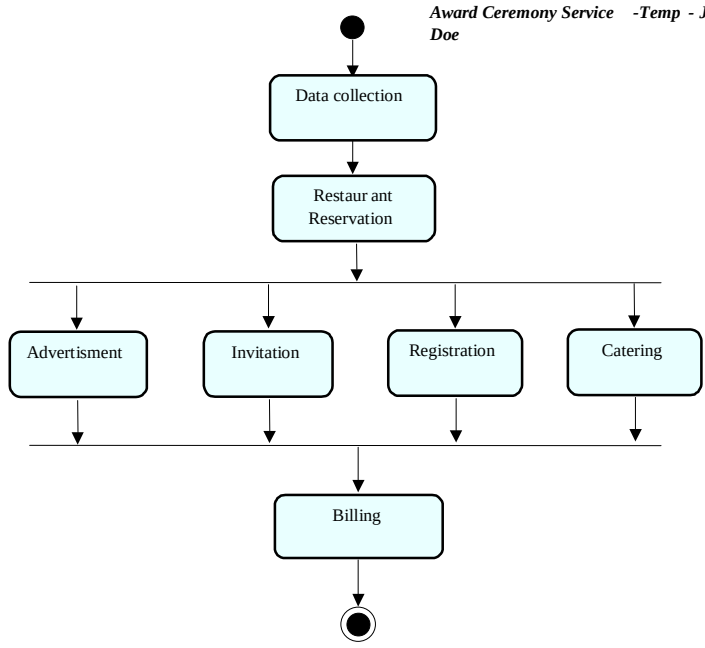

Fig. 5. Ad-hoc process definition to handle the request by customer John Doe

schema). For instance, suppose that a customer of *OneStopShop*, John Doe, is accessing a restaurant reservation service within an Award Ceremony process; John found a restaurant, *Chez Jaques*, that fully satisfies his needs in terms of number of seats, location, and atmosphere, but that does not serve food of satisfactory quality. John then asks *OneStopShop* to provide him a catering service, so that he can rent only the place and separately arrange for the food. Since John is a good customer and the company wants to keep his business, the process responsible decides to satisfy his request and modify the process definition (for this particular instance only) by adding a catering service, as depicted in Fig. 5.

Authorized users can modify every aspect of a schema, including the flow structure, the definition of service, decision, and event nodes, process data, and even transactional regions. *eFlow* only verifies that *behavioral consistency* is respected when migrating an instance to a destination schema (i.e., that instance migration does not generate run-time errors and that transactional semantics can be enforced).

Case migration is a very delicate operation, since it allows changing the rules of the game while it is in progress. Hence, our main design goal has been to define a very simple migration semantics, so that users can easily and clearly understand the behavior of the instance after the modifications have been applied, and avoid the risk of unexpected and undesired effects. In the following we describe how *eFlow* manages and performs instance migrations.

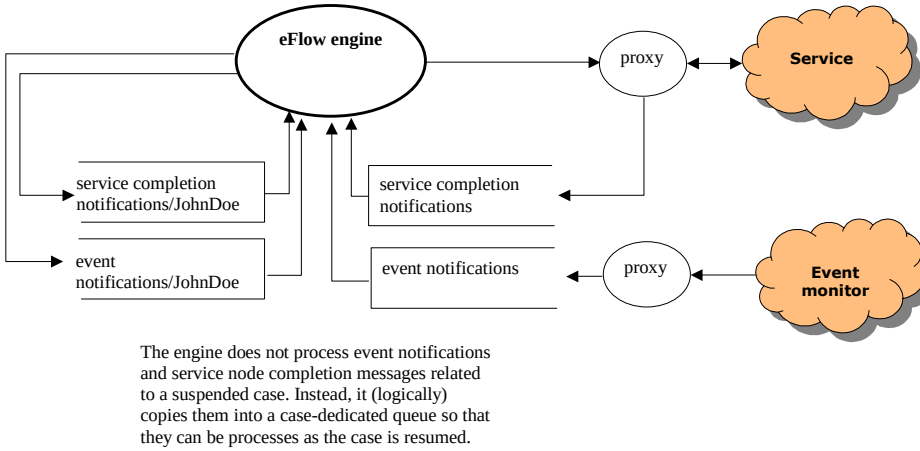

Fig. 6. Events and notifications related to suspended instances are not processed, but are placed in a separate queue

Case migration operations

Case migrations are performed by a suitable *eFlow* module, called *migration manager*. The following operations are performed in order to migrate an instance from a schema to another:

1. An authorized user accesses the migration manager and identifies the instance to be migrated as well as the destination schema (details on user authorizations are provided in section 4.3). The destination schema must have been previously defined, either from scratch or by modifying the one being followed by the instance to be migrated.
2. The migration manager notifies to the *eFlow* engine that instance execution (for the process instance to be migrated) should be suspended. When a process instance is suspended, running services are allowed to complete. However, the engine does not schedule any new service and does not deliver events. When the engine processes a service completion notification related to a service node of the suspended instance, it puts this notification into an ad-hoc, temporary queue maintained for the suspended instance. The notification will be processed when instance execution is resumed. Similarly, events to be delivered to the suspended instance are also placed in a different logical queue (see Fig. 6), and will be delivered as instance execution is resumed. An instance can only be suspended when the engine is not processing messages related to it: in fact, the sequence of operations performed by the engine to process events or service node completion messages and to activate subsequent nodes is atomic.
3. The migration manager verifies that the migration preserves behavioral consistency.

4a. If behavioral consistency is preserved, then the migration manager builds an execution state for the instance in the new schema (details are provided below).

4b. If the instance cannot be migrated, the user is notified of the reason that does not allow the migration and is asked to modify the destination schema (or to indicate

a different destination schema). Steps 1 to 4 will then be repeated. In the meantime, instance execution remains suspended.

5. The migration manager informs *eFlow* that instance execution can be resumed, now according to the destination schema.

At any time during this sequence of operations the user can abort the migration, and instance execution will be resumed according to the old process schema. The operations performed by the migration manager are summarized in Fig. 7.

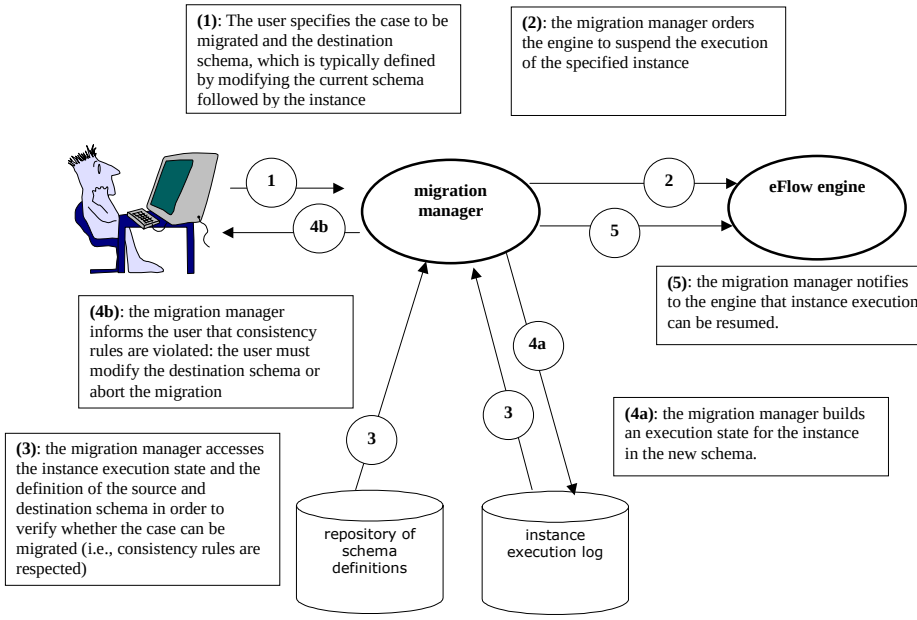

Fig. 7. Sequence of operations performed by the migration manager when migrating an instance

Consistency rules

An instance can be migrated from a version to another only if *behavioral consistency* is preserved. Behavioral consistency implies that the migration does not result in run-time errors or non-deterministic behaviors. In order to guarantee behavioral consistency, *eFlow* enforces the following rules:

1. Each service or event node that is active when the instance is suspended must be present in the destination schema. This rule is necessary since it allows the definition of an execution state for the instance in the new schema, and in particular the definition of which nodes should be set to the *active* state when execution is resumed, as explained below. In addition, it allows the engine to know how to correctly process completion messages related to those running services that will eventually be received. While the definition of active service nodes can differ from the one in the source schema (e.g., they may have different input data or different deadlines), their *write list* must be the same, since it is

expected that the running nodes will actually try to modify the value of those variables.

2. If a variable in the destination schema is also present in the source schema, then it must be of the same type. This rule is needed since variables will keep their current value after migration, and therefore the types in the source and destination schema must be the same.
3. Transactional regions not present in the source schema must not include any node which is part of the source schema and which is active or completed.
4. If a transactional region with the same identifier is present both in the source and destination schema, and the region was active at migration time, then:
   a. The isolation properties of these transactional regions must be the same.
   b. No node in the region of the destination schema should read (write) variables which are not also read (written) by at least one node of the same transactional region in the source schema. The only allowed exception is when the newly introduced variable is only used within the region.
   c. The region should not be extended "in the past", i.e., it should not include nodes that are also in the source schema, that have already been executed, and that are not part of the region in the source schema.

   Rules related to transactional regions are necessary since *eFlow* acquires the read and write locks necessary for enforcing the specified isolation mode at the start of the transactional region.

Migration semantics

The essence of the migration process consists in building an *execution state* for the instance in the new schema, and then in resuming instance execution. An execution state is formed by the value of the case packet variables and by the execution state of all service and event nodes in the instance. The values of case packet variables are set as follows:

- Variables in the destination schema that are also present in the source schema keep the value they had in the case packet of the migrated instance.
- Variables in the destination schema that are not present in the source schema are initialized with their default value (or are left undefined if no default value was provided).

The execution state of service and event nodes is defined as follows:

- Nodes of the destination schema that are also present in the source schema are initialized with the same execution state they had in the migrated instance (e.g., not started, active, completed, failed, canceled, timed out).
- Nodes of the destination schema that are not present in the source schema are initialized to the *not started* state.

After the instance state has been reconstructed, the migration is completed. The migration manager will then inform the engine that instance execution can be resumed. The *eFlow* engine then processes all events and all service completion messages included in the event and service completion queues that were created to manage instance suspension. Elements in these queues are processed with the same semantics used to process elements in the standard queues. After all elements included in both queues have been processed, the engine discards these queues and resume normal operations, that is, it resumes processing of the standard queues.

Modifications to the Process State
Besides changes to the process schema, authorized users can perform the following operations on an instance in execution:

- Change the value of case packet variables.
- Initiate the rollback of a process region or of the entire process.
- Terminate the process.
- Reassign a node to a different service: the running service is canceled, and the one specified by the user is invoked.

These actions are performed through the *service operation monitor* component of *eFlow*, and do not require instance suspension.

4.2 Bulk Changes

Bulk changes handle exceptional situations that affect many instances of the same process. Instead of handling running instances on a case-by-case basis, *eFlow* allows authorized users to apply changes to sets of instances that have common properties. Modifications are introduced by specifying one or more destination schemas and by defining which set of instances should be migrated to each schema. For instance, suppose that *OneStopShop* decides to provide, as a bonus, a security service for all ceremonies that involve more than 100 guests. To perform this, a new service process is defined, by modifying the *Award Ceremony* one, in order to include a security personnel service, as shown in Fig. 8.

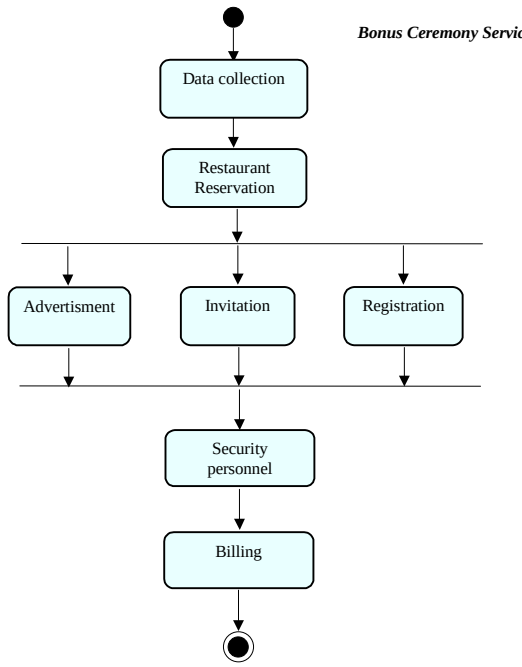

Fig. 8. Modified Award Ceremony service, now including a security service

Next, the service process responsible can migrate all running instances (related to a Ceremony service that involves more than 100 guests) to the newly defined one. Migrations are defined by means of a simple, rule-based language. A migration rule identifies a subset of the running instances of a given process and specifies the schema to which instances in this subset should be migrated. Rules have the form `IF <condition> THEN MIGRATE TO <schema>`. The condition is a predicate over service process data and service process execution state that identifies a subset of the running instances, while `<schema>` denotes the destination schema. Instances whose state does not fulfill the migration condition will proceed with the same schema. An example of migration rule is: `IF (guests>100) THEN MIGRATE TO "Bonus_Ceremony_Service"`.

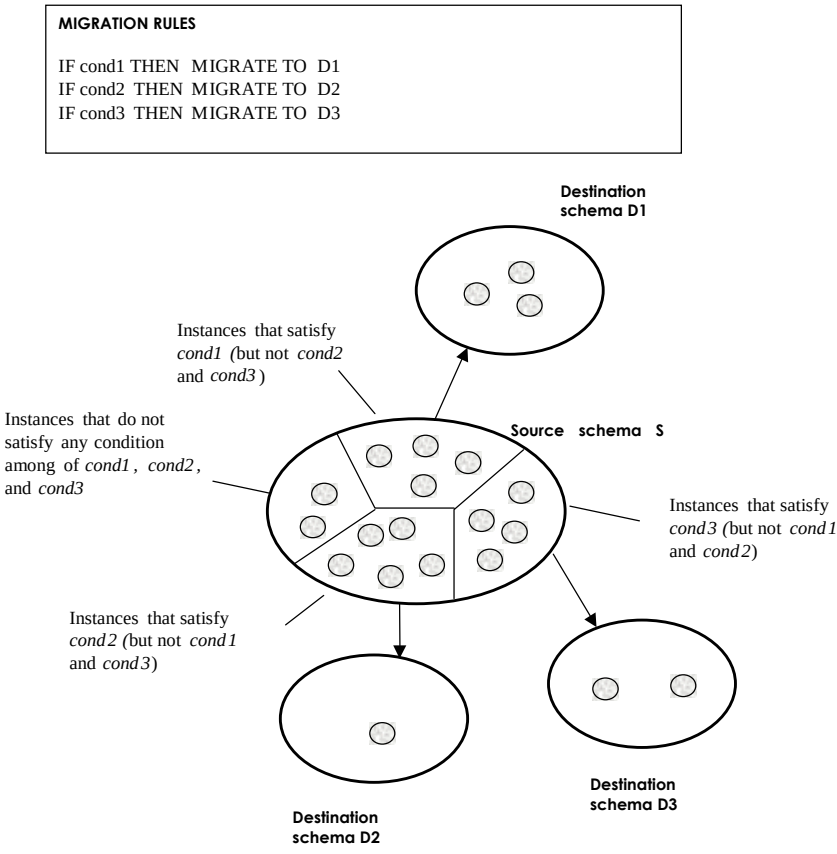

Fig. 9. Bulk migration: instances are migrated to different destination schemas depending on migration rules

The set of rules must define a partitioning over the set of active instances, so that each instance is migrated to one schema at most. Instances that do not satisfy any rule condition are not migrated. Fig. 9 exemplifies bulk migration. The sequence of operations performed in bulk migration is as follows:

1. The user defines, compiles, and checks in the migration rules. All destination schemas referred to by migration rules must have been previously defined.
2. *eFlow* suspends *all* running instances of the process.
3. *eFlow* verifies that the migration rules actually define a partition over the set of running instances. If the state of an instance satisfies more that one migration condition, the user is asked to revise the rules or abort the migration.
4. *eFlow* verifies that each instance can be migrated to the specified destination version of the process schema, i.e., it checks that behavioral consistency is preserved. In addition, it checks that the user who started the migration has the required authorizations to migrate each of the selected instances. Due to the delicacy of a bulk migration operation, *eFlow* does not perform any migration until all instances can be safely migrated to their destination schema.
5. If all migrations can preserve behavioral consistency, then instances are migrated to their destination schema. Instance executions are then resumed.

4.3 Security in Dynamic Process Modifications

Dynamic service process modifications in *eFlow* are constrained by *authorization rules* that defines which user or application is authorized to perform a given modification. Rules are specified at process definition time, and can include an arbitrary number of input parameters, taken from case packet variable. This enables the definition of security rules that differs according to the particular execution state of the instance. Each process definition includes the following authorization rules with respect to process modifications:

- *Authorized_State_Modifiers*: identifies the users (services) that have write access to case packet variables, i.e., that can perform state changes to the instance.
- *Authorized_Node_Modificators*: identifies the users (services) authorized to modify service nodes in the process instance. This rule can also be specified at the node level, to further constrain authorizations.
- *Authorized_Flow_Modificators*: identifies the users (services) authorized to make any kind of dynamic changes to the process instance.
- *Authorized_Initiators*: identifies the users (services) authorized to start an instance of this process

Each time a state change or a migration is requested, *eFlow* verifies that the requestor has the appropriate authorizations, according to the defined rules and to the differences between source and destination schema. In particular, in case of a bulk migration, authorization rule *Authorized_Node_Modificators* (or *Authorized_Flow_Modificators*, depending on the extent of the changes) defined in the *source* schema are executed for each instance to be migrated, and the migration is performed only if the user has the privileges to migrate *all* of these instances. In addition, since the migration will result in executions of the destination schema, rule *Authorized_Initiators* of the defined *destination* schema will be executed, to verify that the user is authorized to create instance of that schema.

5 Related Work

To the best of our knowledge, there is no commercial process management system that supports adaptive and dynamic features such as those of *eFlow*, neither among traditional workflow management systems (such as *MQ Wor*kflow [11] or *InConcert* [10]), nor among newly developed, open, XML- and web-based systems such as *Forte' Fusion* [9] and *KeyFlow* [6].

A few of these systems, such as InConcert and KeyFlow, provide some support for ad-hoc changes, by allowing simple modifications to the schema followed by a given instance as well as execution state modifications. Recently, some approaches to handle dynamic changes have been presented in the literature by the workflow research community.

One of the first contributions come from [2], that defines a correctness criterion for instance migration, based on the definition of the set of all valid node sequences: a change is correct if the execution sequence could have been obtained with the new schema. The paper, however, introduces a simple workflow model and restricts to a limited set of modifications.

Ad-hoc and bulk changes are discussed in [7]. Workflow changes are specified by *transformation rules* composed of a *source schema fragment*, a *destination schema fragment*, and of a *condition*. The system checks for parts of the process that are isomorphic with the source schema and replaces them with the destination schema for all instances for which the condition is verified. The paper also proposes a migration language for managing instance-specific migrations, conceptually similar to our migration language.

Other contributions to the area of workflow evolution come from [8,12]. In [12], a complete and minimal set of workflow modification operations is presented. Correctness properties are defined in order to determine whether a specific change can be applied to a given instance. If these constraints are violated, the change is either rejected or the correctness must be explicitly restored with exception handling techniques. Liu et al [8] focus instead on a language for workflow evolution, by which the designer can specify which instances should be migrated to which versions, depending on conditions over workflow data. The language is conceptually similar to that of [7] and to ours.

In designing *eFlow*, we took advantage of all these research contributions and extended them as follows:

- We designed a model and system that provides all the flexibility features required for a dynamic environment such as that of the Internet, including a wide range of possible ad-hoc and bulk changes;
- we designed a very simple, yet powerful migration language and a very simple migration semantics, to enable an easy understanding of the instance behavior after migration. This is a fundamental requirement in operational environments;
- we discussed migration in the context of a rich process model, which includes events and transactions. These model features posed us additional challenges in managing migrations;
- we introduced authorization constraints that allows the definition who is authorized to perform a given type of change;

- we defined the process followed by the system when the changes are made, focusing in particular on the delicate issue of instance suspension;
- finally, in addition to dynamic change support, *eFlow* also provides a set of adaptive features in order to strongly reduce the need for dynamic changes.

Adaptive process management is also recently gaining attention. The workflow model proposed in [1] includes a "shoot tip" activity: when a shoot tip activity is executed, the control is transferred to a process modeler that can extend the flow structure with one additional activity, which is inserted before the shoot tip activity. Next, instance execution will proceed by activating the newly inserted task and subsequently another "shoot tip" activity to determine the next step. Another interesting approach, which also allows for automatic adaptation, is proposed in [3]. The presented workflow model includes a *placeholder* activity, which is an abstract activity replaced at runtime with a concrete activity type, which must have the same input and output data of those defined as part of the placeholder. A selection policy can be specified to indicate the activity that should be executed. The model has an expressive power similar to the one allowed by *eFlow* dynamic service discovery mechanism. However, we do not restrict the input and output parameters of the selected activity to be the same of those of the node. In addition, we also provide the notion of generic and multiservice node for further achieving additional flexibility and we provide a set of dynamic modification features to cope with situations in which changes in the flow are needed.

6 Conclusions

In this paper we have shown how *eFlow* supports the dynamic composition, enactment, and management of *composite* e-services, i.e., of e-services built on top of other basic or composite services. In particular, we focused on the adaptive and dynamic features of *eFlow*, which are essential characteristics in order to cope with dynamic environments such as that of e-services. Our future research will be focused on providing effective means for monitoring and analyzing instances that have been modified one or more times during their executions.

In summary, we believe that the *eFlow* platform has the required characteristics and functionality to satisfy the need of Internet-based service providers. *eFlow* is integrated with the Hewlett-Packard e-service strategy; however, it is an open technology: it is based on Java and it is compliant with the workflow and Internet standards, such as XML and the Workflow Management Coalition Interface standards. Hence, it can be integrated and used in virtually any IT environment.

References

1. T. Dirk Meijler, H. Kessels, C. Vuijst and R. le Comte. Realising Run-time Adaptable Workflow by means of Reflection in the Baan Workflow Engine. Proceedings of the CSCW Workshop on Adaptive Workflow Management, Seattle, WA, 1998.

2. S. Ellis, K. Keddara and G. Rozenberg, Dynamic Change within Workflow Systems, Proceedings of (COOCS '95), Milpitas, California, 1995.
3. D. Georgakopoulos, H. Schuster, D. Baker, and A. Cichocki. Managing Escalation of Collaboration Processes in Crisis Mitigation Situations. Proceedings of ICDE 2000, San Diego, CA, USA, 2000.
4. Hewlett-Packard. e"speak Architectural Specifications. 2000.
5. Hewlett-Packard. *eFlow* model and architecture, version 1.0. 2000.
6. Keyflow Corp. Workflow Server and Workflow designer. 1999.
7. G. Joeris and O. Herzog. Managing Evolving Workflow Specifications with Schema Versioning and Migration Rules. TZI Technical Report 15, University of Bremen, 1999.
8. C. Liu, M. Orlowska and H. Li. Automating Handover in Dynamic Workflow Environments. Proceedings of CAiSE '98, Pisa, Italy, 1998.
9. J. Mann. Forte' Fusion. Patricia Seybold Group report, 1999.
10. R. Marshak. InConcert Workflow. Workgroup Computing report, Vol 20, No. 3, Patricia Seybold Group, 1997.
11. IBM. MQ Series Workflow - Concepts and Architectures. 1998.
12. M. Reichert, P. Dadam. ADEPTflex - Supporting Dynamic Changes of Workflows Without Loosing Control. Technical report 97-07, DBIS, University of Ulm, 1997.