An Open, Flexible, and Configurable System for E-Service Composition

1 Introduction

The Web is rapidly becoming the platform through which many companies deliver *services* to businesses and individual customers. The number and type of on-line services increases day by day, and this trend is likely to continue at an even faster pace in the immediate future. Examples of e-services currently available include bill payment, delivery of customized news, or archiving and sharing of digital documents. 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, possibly offered by different companies.

In order to enable organizations to pursue this business opportunity we have developed *eFlow*, a platform that supports the specification, enactment, and management of *composite* e-services. Composite services are modeled as processes that are enacted by a service process engine.

Service composition has many requirements that are in common with business process management. For instance, they both need to organize the order of activity execution (service invocation), to manage data and data transfer between activities, and to manage activity assignments and resources. In addition, they need to provide high availability, reliability, and scalability. However, the e-service environment is highly dynamic, both from a business and from a technical perspective: the type of services offered over the Internet is growing at a very fast pace, as is the number of service providers. This requires service process models and systems to be very flexible and to be able to adapt to changes (possibly without the need of human intervention), in order to offer the best available service in any given time. In addition, execution of composite services typically span organizational boundaries, and require the capability of interacting with applications based on different (and always evolving) standards, technologies, and ontologies. Finally, the computing resources involved in e-service delivery are heterogeneous in terms of computing power, ranging from high-end workstations to palmtops devices or appliances. While powerful machines may support large and complex

applications, devices with limited computing and storage power impose severe limitations on the size of the application they can execute.

The *eFlow* model and system aims at addressing these issues by providing a flexible, configurable, and open approach to service composition. In fact, *eFlow* provides an adaptive and dynamic process model, that allows processes to transparently adapt to changes in the environment, to customize service execution according to the customers' needs, and to cope with exceptional situations. In addition, the model and the system are configurable, so that applications running on devices with limited storage space and computing power can download a version of *eFlow* that only includes needed features and that has a smaller footprint with respect to the complete version. Finally, *eFlow* is an open system, in that users can replace default components with the one that best suit their needs, to be able to interact with a given e-service environment. In the following we present the *eFlow* model and system and we show how they can achieve the above mentioned goals.

2 The *eFlow* service process model

This section presents an overview of the *eFlow* process model (the reader is referred to [eFlow10] for a complete description of the model and system). In *eFlow,* a composite service is described as a process schema that composes other basic or composite services. A schema 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.

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. 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.

Figure 1 shows a sample service process graph, related to a candidate interview service, offered by a company to its managers to simplify candidate interviewing. The composite service includes simple services provided by the company itself (such as *Interview room*

reservation), as well as services provided by partners such as *immigration status analysis*, which is offered by a law firm.

2.1 Service nodes

Service nodes represent the invocation of one or more services. *eFlow* includes three different types of service node: **Basic** service nodes, that represent the invocation of a service, **multiservice** nodes, that enable the parallel invocation of multiple instances of the *same* basic node, and **generic** service nodes, that enable the instantiation of *different* basic nodes. We next detail the three *eFlow* service node types.

2.1.1 Basic service nodes

Basic service nodes are executed by invoking a service (provided by the same or different organization, and possibly implemented as another *eFlow* process). The specification of a service node includes its name, identifier, and a list of case packet variables that the node is allowed to read or write. In addition, the service node specification includes a *service selection rule*, to select the appropriate service (and service provider), and a *deadline* to control service execution time. We next detail the syntax and semantics of these two attributes.

The *activity selection rule* is a reference to a rule, executed by an *eFlow* component called *service process broker*, that enables the selection of the appropriate service to be invoked. The rule 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. If no suitable service is found, an event is raised (see Section 2.3 for events and event nodes). Service selection rules are defined in a service broker-specific language. *eFlow* has a default service broker in which service selection rules are specified by XQL queries [Robie98] executed against the *eFlow* service description repository. Services in the repository are described in XML; *eFlow* does not impose any constraint about the exact form of XQL queries and XML service descriptions, but 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 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). *eFlow* includes several proxy objects that supports a number of protocols, such as Hewlett-Packard's *e''speak* [espeak99] or http.

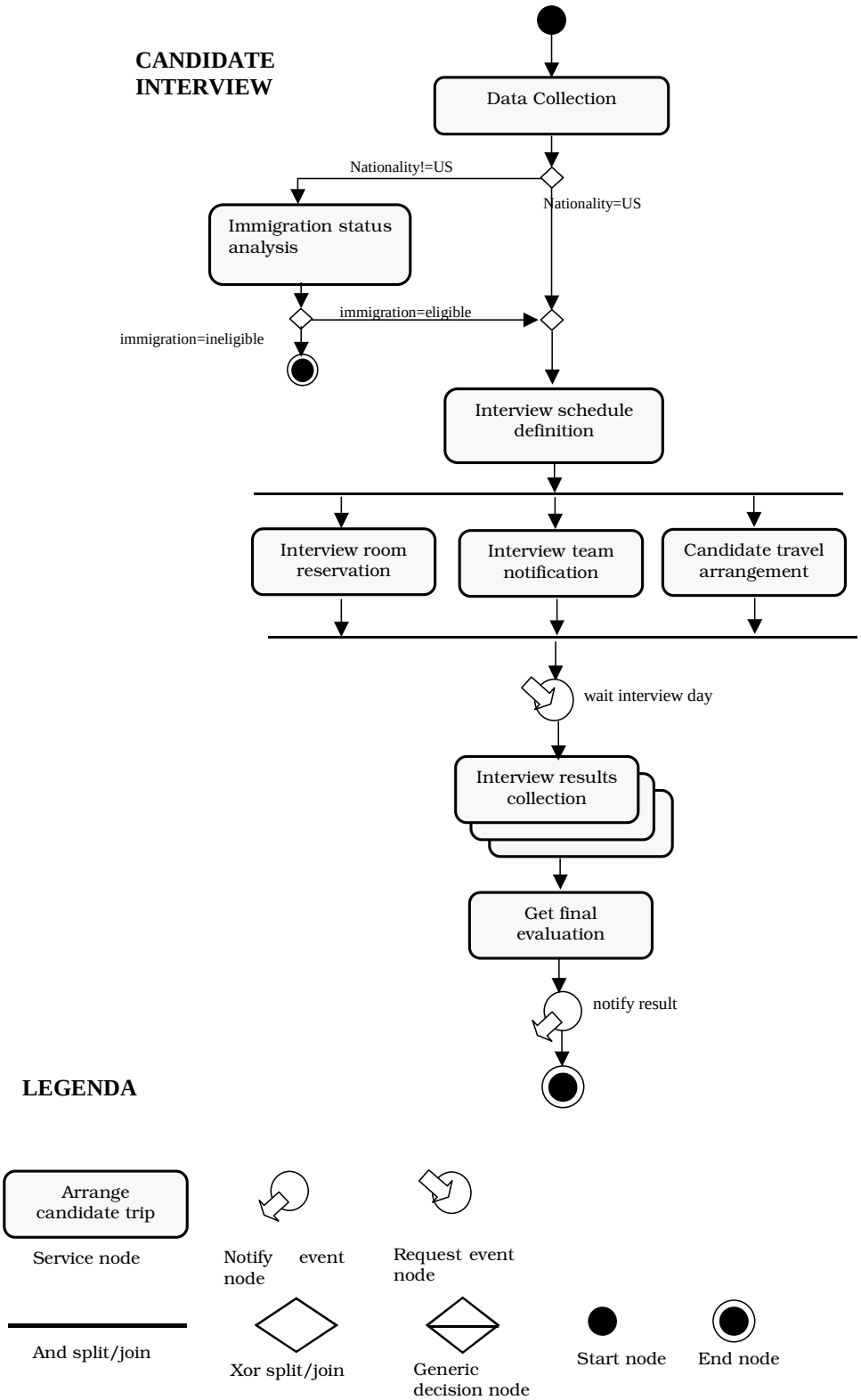

Figure 1 - Candidate interview service

Input parameters in service selection rules enable the customization of rule execution according to the specific service process instance data. 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.

The *Deadline* denotes the temporal instant by which the service node should be completed. Deadlines can be defined as timestamps (either by entering the date and time or by referring to a case packet variable of type *datetime*) or as an interval to be computed starting from a datetime. For instance, we can specify that a service node should be completed before 2000-3-31 18:00:00, three days before the date and time specified by case packet variable *MyLimit*, or one hour after node start. As the deadline expires, an event is raised. In addition, it is possible to specify that the node should be terminated upon deadline expiration.

Figure 2 shows a sample specification of a basic work node. Specifications are made in XML, whose syntax is constrained by a DTD.

```
<BASIC_NODE id="Candidate_travel_arrangement">
        <NAME> Candidate travel arrangement </NAME>
        <DESCRIPTION> Arranges the travel and books the hotel for the candidate
        </DESCRIPTION>
        <READ_LIST> CandiadteInfo
        </READ_LIST>
        <WRITE_LIST> TravelReservations
        </WRITE_LIST>
        <SELECTION_RULE> TravelReservationService(CandiadteInfo.Home, CandiadteInfo.InterviewSite)
        </SELECTION_RULE>
        <DEADLINE action="raiseEvent"> 3 00:00:00 BEFORE CandidateInfo.InterviewDate
        </DEADLINE>
</BASIC_NODE>
```

Figure 2 - Sample XML definition of the *Candidate Travel Arrangement* service node

2.1.2 Multiservice nodes

The multiservice node allows the multiple, parallel activation of the same service node. Its behavior is mainly characterized by two attributes: the *activation*, that defines the number of instances to be started, and the *termination*, that specifies when the

multiservice node can be considered completed. In the following we describe these attributes.

Activation identifies the number of node instances to be activated; this number can be determined (at run time) as follows:

1. It can be determined by the number of service providers able to provide a given service (for instance, we may want a candidate to be interviewed by all available members of a team – seen as "service provider" in this example)
2. It can be equal to the number of elements in a case packet variable of type *vector*. In this case each service node instance receives one and only one of the list items as input parameter. The value of such items 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.

Termination: 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. *Termination* is a Boolean condition, checked every time one of the multiservice node instances terminates, that defines when the multiservice node is completed and the flow can proceed. 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. Figure 3 shows a sample specification of a multiservice node in *eFlow*.

```
<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>
```

Figure 3 - Specification of a multiservice node

2.1.3 Generic nodes

While multiservice nodes enable the activation of multiple instances of the same basic node, generic nodes allow the instantiation of *different* basic 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 specified services will be executed in parallel or sequentially depending on an *execution mode* attribute of the generic service node.

Generic nodes are useful when it is not possible to predetermine the type of services that need to be invoked. For instance, a moving service could be composed of car rental, truck rental, storage space rental, and airline reservation services. The services to be invoked may depend on the preferences of each specific customer (some customers may only requires a truck rental service, for instance), and without the generic node a process designer would have to hardcode all the possible different options by drawing spaghetti-like process graphs and complex routing conditions. In addition, if a new service type is made available, the process definition must be changed to include the new node.

The generic node is characterized by the following attributes, that enables the definition of the nodes to be instantiated and their execution order:

Service node pool: identifies a group of service nodes. Only nodes belonging to the pool can be instantiated within this generic node.

Service selection variable: a reference to a case packet variable of type list that contains the identifiers of the service nodes to be instantiated. The variable can be set at process instantiation time or by a previously executed node.

Execution mode: defines whether the nodes should be instantiated in sequence (in the order defined by the list) or in parallel.

Termination: it is analogous to the correspondent multiservice node attribute.

```
<GENERIC_NODE id="moving_services">
        <NAME> Moving Services </NAME>
        <SERVICE_NODE_POOL> Rentals and Reservations </SERVICE_NODE_POOL>
        <SERVICE_SELECTION_VAR> SelectedRentalServices
        </SERVICE_SELECTION_VAR>
        <EXECUTION_MODE mode="parallel" />
</GENERIC_NODE>
```

Figure 4 - XML description of the generic node *Moving Services*

2.2 Decision nodes

eFlow offers three types of decision nodes: *split nodes* can split the flow of work along parallel or conditional branches; *join nodes* synchronize two or more branches; *generic nodes* can specify arbitrary routing behaviors by means of a routing language, and are used when routing requirements are very complex and cannot be handled by the *eFlow* split and join nodes.

Split nodes have one input arc and two or more output arcs. There are two types of split nodes: and-split nodes fire all the output arcs as the input arc is fired; xor-split nodes allow instead the conditional activation of output arcs: as the input arc is fired, the *routing rules* associated to the xor-split are executed. Rule execution returns the identifiers of the arcs to be fired. If rule execution results in zero or in more then one output arc to be fired, then instance execution is stopped (along that path) and an event is raised.

Routing rules are of the form IF <condition> then FIRE <arc_list>. Conditions are Boolean predicates defined over the value of case packet data, such as (*make="Ferrari" and cost > 100.000*). If the condition evaluates to TRUE, the action is executed. The action causes the firing of one or more arcs, and hence the activation of the nodes connected in output to those arcs.

Join nodes synchronize branches of the process flow. Like split nodes, join nodes are also divided into *and-join* and *xor-join* nodes. Both have two or more input arcs and one

output arc. *and-join* nodes fire the output arc as *all* input arcs have been fired; *xor-join* nodes fire the output arc as *one* input arc is fired. When other input arcs will be fired, no action will be performed, i.e., the output arc will not be fired once again (unless it is *reset* when loops are defined - see below).

Split and join nodes enable the definition of most routing behaviors. However, sometimes there is the need of specifying more complex semantics, such as activating *n* out of *m* output arcs, or joining the flow after a given number of input arcs have been fired and certain conditions hold. Sophisticate routing requirements are supported in *eFlow* by the *generic* routing node.

Generic routing nodes can be of arbitrary complexity. They can have any number of input and output arcs. Like the xor-split, the routing semantics is specified by means of one or more routing rules. As any input arc is fired, all the route rules are executed (in parallel). Route rules will define which output arcs should be fired. Notice that it is the responsibility of the process designer to make sure that all possible cases are covered by routing rules. If a situation occurs that is not handled by the rules in the node, the execution of the process instance (along that path) will stop at that point, and an exception will be raised. Unlike the xor-node, a generic route node can fire more than one output arc.

Loops.

In *eFlow* each node can be executed only once per instance and each arc can be fired once per instance, unless it is reset by a *reset arc* (also called *loop arc*, since by resetting nodes and arcs it enables the definition of loops).

A reset arc can only connect two routing nodes. Every reset arc defines a *scope*, which is defined as the set of all work nodes and all arcs that are in the path between the destination node and the source node of the reset arc.

Figure 5 shows a typical looping structure, where the loop part is included between a xor-join and a xor-split. The Figure also shows with a thick border all service nodes and arcs that are in the scope of the reset arc. *eFlow* has two types of reset arcs:

- The *cancel* reset arc causes a *cancel* message to be sent to every running activity (service) executed within nodes that are in the scope of the reset arc. Then, after

running activities have been canceled, all nodes and arcs in the scope of the arc are reset so that they can run again when they are reached by the control flow.

- With the *delay* reset arc, no cancel message is sent: *eFlow* first waits for the completion of every running activity in the scope of the reset arc; then, it resets all nodes and arcs in the scope of the reset arc and resumes flow execution starting from the reset arc destination node.

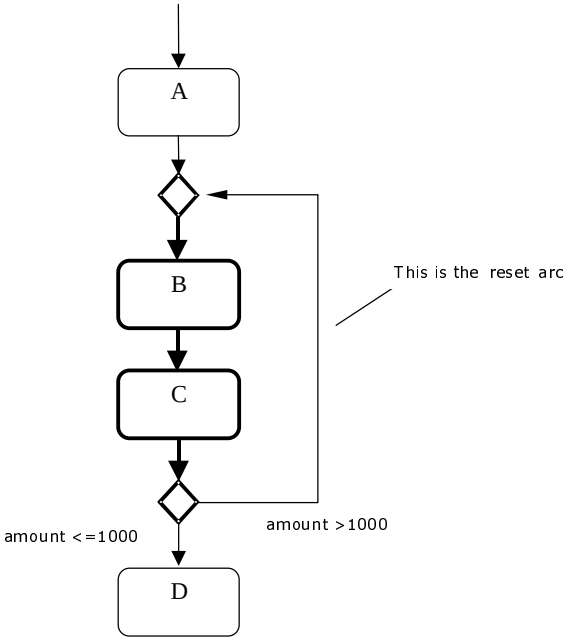

Figure 5 - A reset arc and its *scope*

2.3 Event nodes

eFlow has a very powerful event model that allows processes to send and receive events of several different types, to specify events of interest with a fine granularity, and to capture event parameters into case packet variables. All events, regardless of their type, are specified in a uniform fashion, using the same formalism and the same underlying basic concepts, so that the process designer does not need to learn different languages to specify different kinds of events. We next present the different event types managed by eFlow, and then we show how events can be notified or received by means of event nodes.

2.3.1 eFlow event types

eFlow can capture and manage the following event types:

- **Workflow events**: They are events generated by the *eFlow* engine when processing a process instance, such as instance or node state changes (e.g., activation or completion of nodes and process instances or expiration of deadlines), or error-handling events (e.g., a xor-splits trying to activate several output arcs). All workflow events have the following parameters:
    - *event type:* every type of event has an *event type* parameter. For all workflow events, its value is set to "workflow".
    - *old state:* the state of the node or process prior to the state change (if applicable)
    - *new state:* the state of the node or process after the state change (if applicable)
    - *node name:* the name of the node to which the event is related. If the state change (or the error) is related to a process instance rather than to a node, this parameter is omitted.
    - *process name:* the name of the process definition to which the affected node or process belongs.
    - *process instance Id:* the id of the process instance to which the event is related
    - *error cause:* for error-handling events, the type of error
    - *occurrence time:* the date and time of the event occurrence.

- **Data events** are raised upon modification of case packet variables. They have the following parameters:
    - *event type:* For all data events, its value is set to "data".
    - *variable name:* the name of the variable that changed value
    - *old value:* the value of the variable prior to its modification
    - *new value:* the value of the variable after its modification
    - *process name:* the name of the process definition to which the variable belongs to.
    - *process instance Id:* the id of the process instance in which the value change occurred
    - *occurrence time:* the date and time of the event occurrence.

- **Application-specific events** are qualified by their name and are explicitly raised by a process (through an event node) or by an external application. Application-specific events may carry an arbitrary number of parameters, whose type must be one of the *eFlow* data type. The only event parameters common to all application-specific events (besides the name) are the *occurrence time* and the *event type*, with the same semantics as above.

- **Temporal events** are raised at specified points in time. *eFlow* can detect and raise three types of temporal events:
    - *Instant*: the event is raised only once, at a specified date and time. Usually, the date and time are specified by providing the time of the day, plus the day, month, and year. *eFlow* also allows the definition of symbolic dates such as *Christmas, Easter*, or *Thanksgiving*.
    - *Interval*: interval events are raised as a given interval has elapsed since a *reference* event (e.g., 20 minutes after the activation of the task, or 2 days after the car accident). The reference event in *eFlow* can be a workflow, data, or application-specific event.
    - *Periodic*: these events are raised periodically. The *eFlow* periodic event language allows the specification of simple periodic intervals (e.g., "every 3 hours") as well as the definition of expressions that have knowledge of the calendar, such as "every Monday at 2pm", "every US election day at noon", or "on the 28th day of each month".

All temporal events have parameters *occurrence time* and *event type.* The event type for instant, interval, and periodic events is set to temporal/instant, temporal/interval, and temporal/periodic respectively.

2.3.2 Notify and request nodes

In *eFlow*, points in process instance execution where events should be sent or received are specified by means of *event nodes*. There are two types of event nodes: *Notify* and *Request*. Notify nodes raise an event, which can be notified to other process instances or to other, external applications. Notify nodes are non-blocking: as the event is notified, process execution proceeds immediately by activating the subsequent node.

Request nodes correspond instead to requests of workflow, data, temporal, or application-specific events. They are blocking: as a request node is reached in the control flow of a given process instance, the flow (along the path where the request node is defined) stops until an event with the specified characteristics is delivered to that node.

Event nodes can be defined in a workflow graph just like any other work node: both notify and receive nodes can have only one input and one output arc, and they can be executed only once (i.e., can only send or receive one event) unless they are *reset*. In the following we describe notify and request event nodes in more detail.

Notify nodes

Notify nodes denote points in the flow where a process notifies an application-specific event to the *eFlow* engine[1]. The specification of a notify node includes the definition of the event to be generated, described by a set of <name,value> pairs that constitute its parameters. Values may be defined by constants or may refer to the name of a case packet variable, meaning that the actual value of the parameter is set to the value of the case packet variable at the time the event node is executed.

For instance, the candidate interview service process sends an event notifying the interview result (to be captured by all other defined *eFlow* services interested in this event and authorized to receive it[2]). Assume also that the event must indicate the name of the candidate, the evaluation result, and the interview date. Such an event could be specified as follows[3]:

```
eventName="InterrviewResult"
candidateName=%candidateInfo.Name
result=%candidateInfo.Result
date=%candidateInfo.InterviewDate
```

In the above example, event parameters `candidateName`, `result`, and `date` are set by means of case packet variables.

---

[1] Temporal, data, and workflow events are detected by *eFlow*, and cannot be explicitly raised.

[2] Security issues are not discussed here due to space limitations. The interested reader is referred to [eFlow10] for details

[3] *eFlow* does not require events to have a *name*. If defined, event names such as IntrviewResult are handled like any other parameter.

Request nodes

Request nodes denote points in a process where the execution is suspended (along the path where the event node is defined), waiting for an event (with some specified properties) to be delivered. As the event is delivered to the request node, process instance execution proceeds with the activation of the node connected in output to the request node.

For each request node, the workflow designer specifies a *request rule* that defines the kind of events the node is interested in (specified by a *filtering rule*) and the event data that should be captured within case packet variables (specified by a *capturing rule*). The syntax of the request expression is the following:

```
<receive expression> := <filtering rule> ["CAPTURE" <capturing rule>];
```

We next describe these components in detail.

Filtering rule

The filtering rule defines the characteristics of the events that the request node is interested in receiving. Request nodes can request workflow, data, temporal, or application-specific events. All event types are specified using the same formalism: the event of interest is simply defined by a Boolean condition over event parameters and case packet variables. Typically, the condition compares event parameters with the value of case packet variables to check if these values match.

When a request node is reached by the control flow (i.e., it is activated), a request for the event will be issued. *eFlow* will deliver to the node the first event that satisfies the condition and that occurred *after* the request node activation.

As an example, assume that, in a car rental process, the designer wants to specify an event node that receives an event when work node *car scheduling* (in the same instance) is suspended. This semantics can be achieved by the following rule:

```
event_type=”workflow”      and
new_state=”suspend”        and
node=”car scheduling”      and
process_instance_id=%id
```

As another example, suppose that in a bank account management process, a request node monitors variable *balance* in order to check when it turns from a positive to a negative value. This event can be specified as follows:

```
event_type="data"          and
variable_name="balance"    and
old value >0               and
new value <0
```

Finally, we present an example related to temporal events. Conceptually, *eFlow* raises a temporal event every second. This allows temporal events to be specified by a filter on the *occurrence time* parameter, by using a simple language that enables the definition of datetime, interval, and periodic times. For instance, filters for *Periodic events* can be specified by providing the *base* time that defines when the first of the (potentially infinite) series of periodic event should occur, and the time *period* that should elapse between two successive occurrences of the periodic event. The time period (preceded by keyword *every*) is expressed in terms of days, hours, minutes, and seconds (just like a temporal interval). For instance, in order to specify an event that should be raised every 20 minutes, starting from 12:00am on Dec 24, 1999, the designer defines the following rule:

```
event_type="temporal/periodic" and
period= every 20:00 from 1999-12-24 00:00:00
```

A frequent need in process design is the specification of periodic events that are related to *symbolic* periods, such as "every Easter at noon" or "on the 27th of each month". These periodic events cannot be expressed with the *base+period* notation, since the interval between two successive Easters and two successive 27th day of the month are variable. To enable the specification of this important class of periodic events, *eFlow* allows the time period to be specified as: "every" <symbolic date> "at" [day] hh:mm:ss from <timestamp> [to <timestamp>]. For instance, the periodic event "every Easter at noon" can be specified as:

```
Type=temporal/periodic
period= every Easter at 12:00:00 from 1999
```

The *eFlow* administrator can create new symbolic dates in order to customize the system for the specific needs of the organization in which it is used (for instance, to include local

holidays or company holidays). We refer the interested reader to [eFlow10] for a complete description of events and specifically for the temporal event language.

Capturing rule

Besides defining the event of interest by means of the filtering rule, a request node may also capture the value of message attributes into local workflow variables. For this purpose, the filtering rule is coupled with a *capturing rule*.

For instance, assume that in the candidate interview process, the designer also wants to capture, from *interviewCancel* events (whose parameters are the candidate's *name*, the position *code*, and the *reason* for cancellation), the reason why the candidate canceled the job application.

When filtering this event in the event request, an interview process instance is not interested in the reason of the cancellation: events of interest are all *interviewCancel* events that are related to the person and the position for which the process is scheduling the interview, regardless of the cancellation reason.

However, when the event is received, the process needs to capture the cancellation reason for bookkeeping purposes (or possibly to propose higher compensations if the reason was that the candidate has received another offer).

Event parameters can be captured into case packet variables by specifying a set of pairs <case packet variable name, event parameter name>, meaning that the specified case packet variable takes the value of the specified event parameter as the event is delivered to the request node.

The capturing rule for the above described example can be defined as follows:

```
%candidate_cancellation_reason=reason
```

2.3.3 Event-based process activation

Some processes may need to be activated periodically, or as a given event occur. For instance, a service process that sends monthly paychecks to employee must be activated every month (periodic event), while one that handles electronic employee reimbursements must be activated as a reimbursement request is notified (application-specific event).

eFlow supports these needs by allowing the definition of a *process activation rule*, which has exactly the same syntax of event rules defined in request nodes. In particular, the

process activation rule has a filtering component that defines when a new instance of the process should be created, and a capturing rule that captures the parameters of the activating event and uses them as initialization data for the newly created instance.

For instance, the activation condition for the electronic employee reimbursement process could be as follows:

```
event_type=”application-specific”      and
event_name=”reimbursement_request”
CAPTURE %empName=employee_name;
```

2.4 Transactions

The *eFlow* model includes the notion of *transactional regions [Krishnamo00]*. 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 *Interview room reservation*, *Interview team notification*, and *Candidate travel arrangement* services within 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.

The purpose of a compensating action is to semantically undo the work performed by a completed service. If a service node cannot be compensated (or anyway if the compensating node has not been specified), then it cannot be inserted in a transactional region, unless the compensating action is defined at the region level.

2.5 Dynamic service process modifications

Adaptive features such as generic and multiservice nodes, service selection rules, and event-based error handling, allows for very flexible process definitions and enable dynamic process configuration and binding of services with service nodes, thereby minimizing the need for human intervention in maintaining process definitions.

However, 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 [Joeris99, Reichert97]. *eFlow* allows two types of service process modifications:

- *Ad-hoc changes: 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* schema). *eFlow* allows maximum flexibility in the kind of changes allowed to the schema followed by a given instance. 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). Besides changes to the process schema, authorized users can modify the course of a process instance execution by changing the value of case packet variables, initiating the rollback of a process region or of the entire process, terminating the process instance, or reassigning a node to a different service.
- *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. 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. 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.

3 *eFlow* Architecture

We next give an overbiew of the *eFlow* architecture and discuss its possible configuration and extensions. Details on the architecture are provided in [eFlow10].

3.1 Architecture overview

Figure 6 presents the overall architecture of *eFlow*. Composite services are specified through the *Service Process Composer*, that enables the definition and modification of process schemas. Services are enacted by the *service process engine*, which is composed of the *Scheduler*, the *Event manager*, and the *Transaction manager*. The Scheduler processes 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 scheduler then contacts the Service process 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 *Event Manager* monitors event occurrences, by detecting temporal, data, and workflow events, and by receiving notifications of application-specific events by external applications (typically by external event managers such as those provided by TIBCO or ActiveSoftware). In fact, *eFlow* provides modules that enable its event manager to interface with other event managers, in order to notify and receive events from such publish/subscribe systems. When an event is notified, the Event Manager analyzes current requests and verifies which nodes (if any) are interested to the event. Then, it delivers the events to the requesting nodes.

The *Transaction manager* enforces transactional semantics by enabling compensation of transactional regions. When a region needs to be aborted, the transaction manager identifies the compensating activities to be executed and the order of their execution (i.e., it builds a compensating schema). Then, it notifies the schema to the scheduler in order to execute it.

The *Service operation monitor* logs the execution details and provides correction assistance if desired. It is the component through which authorized users can monitor running service executions or retrieve information about completed service executions.

Finally, the *Migration manager* handles dynamic changes to process instances. It suspends instances to be migrated, verifies that the migrations can be actually performed (i.e, that behavioral consistency is preserved), generates an execution state

for the instance in the new schema, and finally informs the engine that the execution of the migrated instances can be resumed.

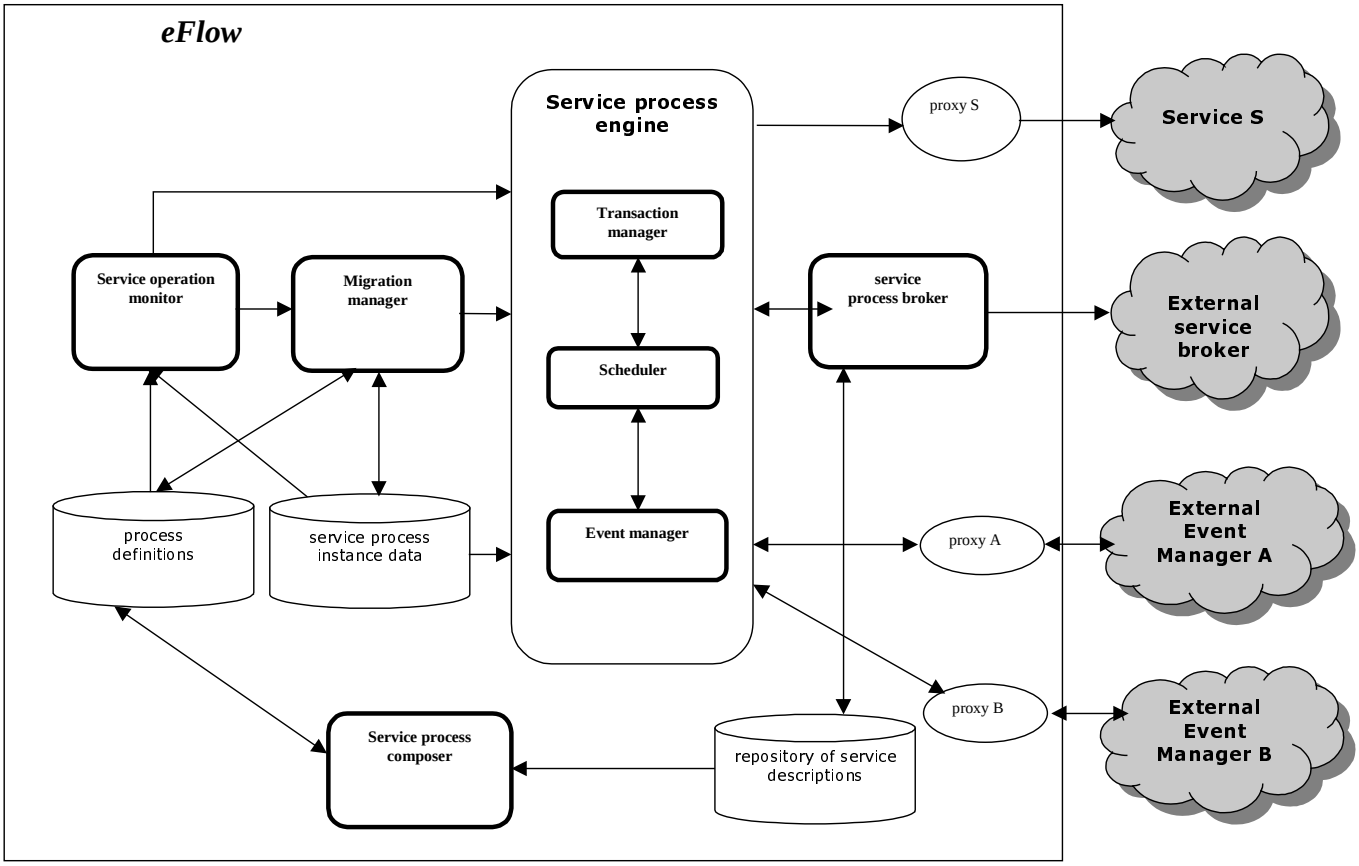

Figure 6 - The *eFlow* architecture

3.2 *eFlow* configurations and extensions

This section discusses two additional aspects of flexibility and openness in *eFlow*: the service process engine configurations and the system extensions by means of customized components.

3.2.1 Service process configurations

Technological advances and the ever-increasing usage of the web to exchange information and perform transactions are extending the type of devices that can access the Internet. Palmtops, cellular phones, printers, microwave ovens and other appliances can access the Web or include web servers. While some of these devices can have considerable computing power and storage space, most of them are very limited in this kind of resources. Hence, in order to enable devices to deliver composite services, we

designed the *eFlow* model to be configurable: devices with limited capabilities (or, in general, applications that do not require all of the model and system features) can install lightweight versions of *eFlow*, which have less functionality but that are smaller in footprint. These simpler versions of *eFlow* can then be progressively upgraded, if needed, up to the full version presented in this paper.

The basic version of *eFlow* only allows sequential, linear processes, does not include event or decision nodes, and it only offers basic service nodes. In addition, it does not provide for transactions and audit logging. This basic version can be progressively extended, by adding more functionality (at the cost of having a more heavyweight system).

Figure 7 shows the building blocks of the *eFlow* control flow model and how more complex features can be added on top of more basic configurations. Boxes represent layers, while arrows from a layer A to a layer B means that configuration B requires as a minimum the feature provided by configuration A. If a block has several input arrows, it means that at least one of the input blocks is needed.

- *Basic layer*: allows the definition of sequential processes: no decision, event, multiservice, or generic node is provided, and no support for transactions or audit logging is included.
- *Conditional layer*: adds the possibility of defining xor-splits and xor- join nodes.
- *Parallel layer*: adds the possibility of defining and-splits and and- join nodes.
- *Generic layer*: adds the possibility of defining generic routing nodes.
- *Loop layer*: adds the possibility of defining reset arcs. The loop layer can be plugged-in on top of the *conditional* or of the *generic* layer[4].
- *Adaptive Layer*: it adds multiservice and generic nodes.
- *Event layer*: it enables the definition of notify and request event nodes.
- *Audit layer*: adds audit logging capabilities.
- *Transaction layer:* adds the possibility of defining virtual transactions. It requires the logging layer since the transaction manager accesses the log in order to determine the compensating sequence to be executed.

[4] It does not make sense to plug in lops on top of the parallel layers, because loops requires xor-joins or generic routing nodes, otherwise they lead to deadlocks

At the time this paper is written, we have only implemented the basic, parallel, and adaptive layers, and we are currently implementing the other layers. We foresee that the event layer will be the one causing the biggest increase in the system's size.

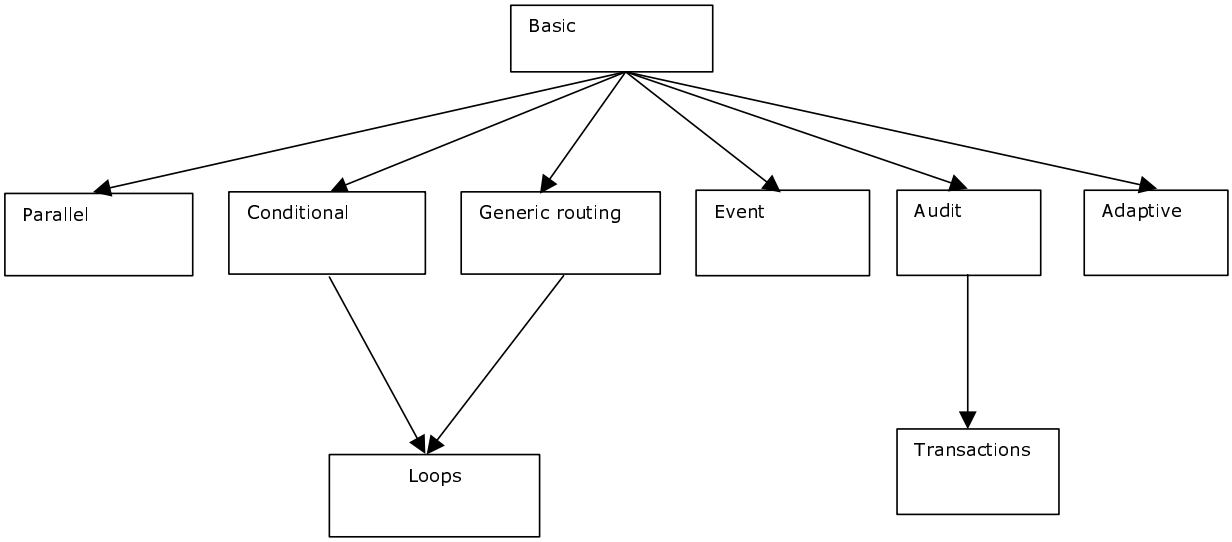

Figure 7 - Building blocks of the *eFlow* model

3.2.2 Ad-hoc *eFlow* components

The Service process engine includes the core components of *eFlow*, necessary to enforce the semantics of the process model described in Section 2 and detailed in [eFlow10]. While *eFlow* also provides default components for defining, monitoring, and migrating process instances as well as for selecting services to be invoked, all of them can be replaced with user-defined ones, in order to extend the features of the system and to enable different service broker strategies and policies. The interfaces and the database schemas needed in order to correctly implement these components so that they can interact with *eFlow* have been kept very simple in order to ease the development of ad-hoc components for the implementation of features required by specific customers, by the evolution of standards, or by the adoption of a specific policy and ontology.

For instance, *eFlow* users can replace the default broker and plug-in the service broker that best fits their needs. The model and the system have been designed in such a way that the engine does not need to have knowledge of how the service selection rule should be executed. Indeed, the rule is not even included in the process definition file. 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[5].

4 Related Work

To the best of our knowledge, there is no commercial process management system that provides the flexibility and extensibility of *eFlow*. Commercial process management systems, such as *MQ Wor*kflow [MQworkfl98], *InConcert* [Inconcer97], *Staffware2000* [Staffwar99], or *Forte' Fusion* [Mann99], are excellent for "traditional" workflow management and for integrating enterprise applications (which are the purposes for which they were originally designed), but do not provide the required flexibility and expressive power to cope with the e-services environment. In particular, their process model provides very limited event and exception handling support, does not allow for late binding of activities, and does not include generic nodes. In addition, they do not support dynamic change of process instances (with the exception of InConcert, that allows simple modifications to the schema followed by a given instance as well as execution state modifications) and do not have configurable systems.

Within the research community, no work has been done on configurable or extensible workflow systems. Work on flexible workflows has been mostly focused on dynamic process modifications. For instance, ad-hoc and bulk changes are discussed in [Joeris99]. 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 and to that presented in [Liu98].

[5] 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.

In [Reichert98], 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.

Adaptive process management is also recently gaining attention, although few contributions have been made in this field. The workflow model proposed in [Dirk98] 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 [Georgako00]. The presented workflow model includes a *placeholder* activity, which is an abstract activity replaced at runtime with a concrete activity type, that must have the same input and output data of those defined as part of the placeholder. In addition, a selection policy can be specified to indicate the activity that should be executed. The model allows an expressive power similar to the one allowed by *eFlow* dynamic service discovery mechanism, obtained through service selection rules. However, we do not restrict the input and output parameters of the selected activity (service) to be the same of those of the node. In addition, we also provide the notion of generic and multiservice node for achieving additional flexibility, we provide an open approach that enables user to plug in their service broker (and use their own language to specify selection policies), and finally we also provide a set of dynamic modification features to cope with situations in which changes in the flow are needed.

Events in workflow systems are supported by Staffware and by a few research prototypes, such as Opera [Hagen98] or WIDE [Casati00]. Staffware and Opera allows the definition of event nodes. However, they can only receive events, they do not have event filtering capabilities, and they are more limited with respect to *eFlow* in the type of events they can generate and monitor. WIDE has a very rich event language, but it adopts a different approach for their management, based on ECA rules, whose conditions are expressed in a Datalog-like language. While this approach is very powerful, we experienced that rule design is more complex with respect to event nodes, and can easily lead to errors because of subtle semantic issues and unforeseen interactions among

rules. Therefore, we integrated event handling in the process graph by means of event nodes.

References

[Casati00] F. Casati, S. Ceri, S. Paraboschi, G. Pozzi. Specification and implementation of exceptions in Workflow Management Systems, ACM Trans. On Database Systems, to appear.

[Dirk98] 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.

[Edmond98] D. Edmond and A. ter Hofstede. Achieving Workflow Adaptability by means of Reflection. Proceedings of the CSCW Workshop on Adaptive Workflow Management, Seattle, WA, 1998.

[eFlow10] Hewlett-Packard Labs. *eFlow* Service Process Model and architecture, version 1.0, 1999.

[espeak99] Hewlett-Packard. e"speak Architectural Specifications. 1999.

[Georgako00] D. Georgakopoulos, H. Schuster, D. Baker, and A. Cichocki. Managing Escalation of Collaboration Processes in Crisis Mitigation Situations. Procs of ICDE 2000, San Diego, CA, USA, 2000.

[Hagen98] C. Hagen, G. Alonso. Flexible exception handling in the OPERA process support system. Proceedings of ICDCS 98, Amsterdam, May 1998.

[Krishnam00] V. Krishnamoorthy and M. Shan. "Virtual transactions". Proceedings of SAC 2000, Como, Italy, March 2000.

[Inconcer97] Ronni T. Marshak. InConcert Workflow. Workgroup Computing report, Vol 20, No. 3, Patricia Seybold Group, 1997.

[Joeris99] G. Joeris and O. Herzog. Managing Evolving Workflow Specifications with Schema Versioning and Migration Rules. TZI Technical Report 15, University of Bremen, 1999.

[Liu98] C. Liu, M. Orlowska and H. Li. Automating Handover in Dynamic Workflow Environments. Proceedings of CAiSE '98, Pisa, Italy, 1998.

[Mann99] J. Mann. Forte' Fusion. Patricia Seybold Group report, 1999.

[MQworkfl98] IBM. MQ Series Workflow - Concepts and Architectures. 1998.

[Reichert97] M. Reichert, P. Dadam. ADEPTflex: Supporting Dynamic Changes of Workflows Without Loosing Control. Tech. Rep. 97-07, Dept. of Databases and Information Systems, University of Ulm, 1997.

[Robie98] J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL). Proceedings of the Query Languages Workshop, Cambridge, Mass., 1998.

[Staffwar99] Staffware Corporation, Staffware2000 White Paper, Available at http://www.staffware.com/home/products/Staffware2000WP.zip, 1999