Composing and Maintaining Web-based Virtual Enterprises

Boualem Benatallah[1] and **Brahim Medjahed**[2] and **Athman Bouguettaya**[2]
and **Ahmed Elmagarmid**[3] and **James Beard**[4]

[1] School of Computer Science and Engineering
University of New South Wales, Australia
boualem@cse.unsw.edu.au

[2] Department of Computer Science
Virginia Tech, USA
{brahim,athman}@vt.edu

[3] Department of Computer Sciences
Purdue University, USA
ake@cs.purdue.edu

[4] CiTR Pty Ltd, Australia
j.beard@citr.com.au

Abstract

Electronic commerce (E-commerce) is evolving from the simple notion of electronic catalogs to the notion of virtual enterprises (VEs). Existing enterprises form alliances, joining their services in order to share their costs, skills and resources in offering a value-added service. Ad-hoc and proprietary solutions on the one hand, and lack of a canonical model for composing and managing VEs on the other hand, have largely hampered a faster pace in deploying Web-based VEs. In this paper, we propose a generic framework for creating and maintaining VEs. We introduce a language, WebBIS-SDL, to cater for the definition and maintenance of VEs. To provide an early feasibility of the proposed framework, we have implemented a prototype that allows easy definition and maintenance of VEs. The implementation architecture is based on CORBA and proven Web technologies including Java and database APIs.

1 Introduction

Business-to-Business (B2B) E-commerce is a prime candidate to take advantage of the information revolution the Web has brought about [14, 15, 28]. All predictions agree that B2B E-commerce will be worth billions of dollars in new investment. The recent forecast by Forrester [17] estimates E-commerce to be a US$ 2.7 trillion market in the year 2004. It is also estimated that more than 60% of all businesses will move their main operations to the Web in the year 2002. Cheap connectivity and ease of advertising of data and services on the Web created tremendous opportunities for organizations of any size to diversify their customer-base and become truly global [7, 10]. By service, we refer to a semantically well defined functionality that allows users to access and perform tasks offered by business applications. Examples of services include electronic catalogs, order procurement, customer relationship management (CRM), finance, billing, accounting, human resources, supply chain and manufacturing [14, 15, 28].

The ability to efficiently and effectively share services across the Web is a critical step towards the development of the new on-line economy driven by the B2B E-commerce. Organizations will be able to integrate their services across boundaries to form what is known as *Virtual Enterprises* (VEs) [19]. Briefly stated, a VE is a conglomeration of core and outsourced services working in tandem to achieve the business goals of the enterprise. Existing enterprises form alliances, joining their services in order to share their costs, skills and resources in offering a value-added service. An example of a VE is a computer company that offers sophisticated and complete computer configurations. Services are outsourced from several business partners. The trading community is composed of autonomous companies producing hardware and software parts (monitors, processors, compilers, etc.) Upon reception of an order from a customer, the computer company sends requests to its partners for the needed parts (or services). VE architectures will depend on the relationships between the business partners:

- *Centralized*: a central organization forms a *long term* relationship with its partners in a *tightly* coupled mode. The organization controls the global business process of the enterprise. A typical example of a centralized VE is a value-added service that provides an integrated access to all customer information systems of an organization (e.g., an organization may have several independent customer information systems which are developed for different purposes).
- *Federated*: the participants are *loosely* coupled and form a *long term* relationship. A participant may independently collaborate with other services in order to play a role in multiple VEs. A typical example of a federated VE is a product manufacturing value chain. A participant would focus on one activity in the value chain and partners with multiple services in other value chains.

- *On-the-fly*: the participants are *loosely* coupled and form a *transient* relationship. They may need to form a fast and short term partnership (e.g., for one business transaction), and then disband when it is no longer profitable to stay together. In contrast to the previous forms, a participant does not assume an *a priori* defined trading relationship with other partners. Instead, it needs to dynamically discover the partners to team up with to deliver the required service.

Now that the Web has made services readily accessible, the challenge is to automatically compose them to create VEs. However, there has been little success to achieve this objective [19]. Indeed, the development of integrated VEs is still ad-hoc, time-consuming and requires enormous effort of low-level programming [1]. This task is obviously tedious and hardly scalable because the Web is *large*, *distributed*, *heterogeneous*, *volatile*, and *highly dynamic*. First, enabling a VE developer to locate relevant services for composition is a key challenge. The existing Web tools give very little support for the logical organization of the service space, which makes the effective use of services enormously complex. Another barrier is the lack of underlying frameworks to describe and advertise services with semantically meaningful abstractions so that they can be efficiently discovered, accessed, and composed. Second, since the services to be integrated are most likely autonomous and heterogeneous, building a VE with appropriate inter-service coordination is difficult. More significantly, the fast and dynamic integration of services is an essential requirement for organizations to adapt their business practices to the highly volatile and dynamic nature of the Web. Third, VEs require flexibility to dynamically adapt components and relationships as required such as changing partnerships in order to offer similar services to the customers in a more cost-effective way.

Conventional point-to-point integration techniques such as EDI [28], component-based E-commerce systems (e.g., integrated electronic catalogs [14, 15] and cross-organizational workflows [13, 19, 20, 21]) are usually appropriate to integrate small numbers of services with static relationships. In fact, these techniques provide interesting foundations to support centralized and to some degree federated partnerships. VEs, particularly on-the-fly partnerships, require more flexible integration techniques. More significantly, the existing techniques are ineffective in large and highly dynamic environments.

Database research has undeniably made large strides towards integrating data on the Web [8, 9, 16]. However, database solutions are in need to be revisited to cater for VE requirements. There is evidence that current database techniques are inadequate to cope with the fast increase in data and transaction volumes on the Web [10]. In addition, VE support differs from structured data support in at least two ways. First, a VE service[1] usually provides a pre-defined interface to access and perform tasks offered by applications (e.g., Java programs, CORBA servers) including those implemented on top of databases (e.g., CGI scripts) [25]. The provision of a well-defined interface is essential for VE services to interact and exchange business information with other services (and end-users). In contrast, a database provides an SQL-like language to access and query data. Second, a VE service usually requires a sequence of operations and multiple interactions to fulfill a task [19, 23]. For instance, in order for a VE to monitor the execution status and availability of a component service, it must be possible to interact with this service during the provision of the required operation. In contrast, interactions with databases are rather simple (i.e., send query/get results).

Recently, a working group of the VLDB Endowment [2] suggested to distinguish between two main research directions: core database technology and infrastructures for developing next generation of information systems. While the first direction has been enormously successful over its 30-year history, it is clearly stated that the second direction is currently under-developed [2, 5]. A primary objective of research in the second direction is to promote database technology as an essential component of the infrastructure that will support the management of all forms of data and application services (e.g., workflows, distributed components and objects, agents, legacy systems) available on the Web. Virtual enterprises are particularly emerging as new Web-based applications that illustrate this direction [27]. In this paper, we present our work in the *WebBIS* (*WebBase of Internet-accessible Services*) ongoing project. The fundamental aim is to provide a framework for the efficient and effective *creation* and *management* of VEs. The current focus of WebBIS project is on providing support for scalable, extensible, and flexible integration of VE services. For this purpose, the proposed framework provides constructs to organize, abstract, search, compose, evolve, monitor, and access VE services. Our approach is based on a three-folded premises:

- It caters for the creation of *dynamic*, *transient*, as well as *long-term* relationships among services. To deal with the volatility of the Web, our approach provides means to create virtual communities that bring together variety of providers and requesters around a common interest. A virtual community is defined by describing

[1] In the remainder of this paper, we will use the terms service and VE service interchangeably.

the desired service (e.g., buying computers). Actual providers can register with any community of interest. They join and leave communities at their own discretion. The participants in a community may thus form temporary alliances and then disband when it is no longer profitable to stay together.

- It uses an *ontological* organization of the space into meaningful subspaces (virtual communities) to filter interactions and accelerate service searches. An *ontology* defines taxonomies based on the semantic proximity of terms [6]. In our case, an ontology is defined on VE services.
- It provides support for the *maintenance* of VE services. In a distributed environment, new services could come on-line, existing services might be removed, the content and capabilities of an existing service may be updated, etc. Our approach features the support for reporting and propagating VE service changes.

We propose two complementary declarative languages to cater for the definition (WebBIS-SDL) and manipulation (WebBIS-QL) of VE services. *WebBIS-SDL* (*WebBIS Service Definition Language*) allows to describe and evolve the semantics of services and their interactions, wrap proprietary VE services, compose, and evolve VEs. *WebBIS-QL* (*WebBIS Query Language*) caters for the search of relevant VE services. This paper presents the WebBIS features for defining and maintaining VEs. Due to space constraints, the service search capabilities of WebBIS are outside the scope of this paper. Details about these aspects can be found in [4]. To provide an early feasibility of the proposed framework, we have implemented a prototype that allows transparent access to a network of services. The implementation architecture is based on Java, CORBA, and database APIs.

The remainder of this paper is organized as follows. Section 2 gives an overview of the WebBIS framework. Section 3 introduces WebBIS-SDL's constructs for wrapping proprietary/legacy services. Section 4 describes WebBIS primitives to compose VEs. Section 5 discusses maintenance issues in WebBIS. Section 6 is devoted to the WebBIS implementation. Related work and concluding remarks are presented in Section 7.

2 Overview of the WebBIS Framework

The current information infrastructure allows phenomenal flexibility for producing Web-accessible services. What is lacking is a framework to organize, abstract, search, compose, and evolve VEs. The support of flexible and efficient creation and maintenance of VEs requires the ability to:

- Model complex, autonomous, and heterogeneous services.
- Quickly develop and deploy new VEs from existing component services.
- Efficiently discover and exploit VE services in a dynamic and constantly growing environment.
- Dynamically adapt components and relationships of VEs. For example, it should be possible to change partnerships in order to offer similar services to the customers in a more cost-effective way or to maintain user specified quality of service despite changes in the operating conditions.

In order to address the above issues, we propose to model VE services using *wrapped services*, *pull-communities*, and *push-communities*. Wrapped services are objects that provide means to abstract proprietary services from their physical organization. This has the important advantage that they can be used as basis for further manipulations (e.g., query, reuse, and customization). Pull- and push-communities provide means to create VEs by allowing organizations to form partnerships and leverage their core services. It should be noted that we focus on federated and on-the-fly partnerships. The reason is that building VEs over the Web requires the support of (fast) integration of loosely coupled, potentially dynamic, and autonomous component services. Pull-communities provide means to create federated VEs. Push-communities are introduced for two reasons. First, they provide means to create on-the-fly VEs. Second, they provide means for an ontological organization of the cyberspace in order to reduce the overhead of searching VE services. Indeed, each push-community is specialized in a single area of interest. In a nutshell, push-communities can be used as a basis to establish online marketplaces for VE services. A push-community can be seen as an online market for all consumers and providers in a given domain (e.g., computer manufacturing, car manufacturing, supply chain, healthcare, etc.) To provide an end-to-end solution, our approach provides support for the maintenance of VE services.

In the remainder of the paper, we use an example of a computer manufacturing VE (Figure 1) to illustrate our approach. In this example, the trading community is composed of autonomous companies providing hardware parts such as *motherboards*, *CPUs*, *keyboards*, *memory storage devices*, and so on.

2.1 Wrapped Services

A uniform model is central to represent service sharing. We adopt an object-oriented model to represent VE services. Objects are suitable to represent complex structures and relationships among services and allow the expression of their behavior using operations. In order to manipulate a proprietary service and abstract it from its physical organization, we wrap it by a WebBIS compliant service, called *wrapped service*. This has the important advantage that VEs can integrate proprietary and legacy services. A wrapped service is an object that encapsulates the content and capabilities of the underlying service. As depicted in Figure 1, companies providing hardware parts are represented by the wrapped services `processorprovider_1`, ..., `processorprovider_n` and `peripheralprovider_1`, ..., `peripheralprovider_n`.

A VE service can be represented by an object with an *identifier* (the URI), *attributes*, *methods*, and *notifications*. The attributes describe the service's relevant information including general (e.g., owner), access (e.g., public key certificate), selection (e.g., documentation), and control (e.g., service observable states like `running`, `completed`, etc.) information. Methods describe the service operations including valid requests to access, monitor (e.g., supervision of execution), and control the service (e.g., abort invocations). Notifications describe the events that can be sent by the service to its requesters. For example, if a user orders a processor from `processorprovider_1`, the latter may reply by a notification when the order is completed. Event-driven systems are now becoming the paradigm of choice for organizing many classes of loosely coupled and dynamic distributed applications [12, 20, 21, 22, 24]. In our approach, *events* are typically used to monitor (e.g., changes tracking) and provide awareness (e.g., notifying users about changes in services) about specific situations. The support of event monitoring and awareness results in providing pro-active and adaptive VEs that have the ability to notify their requesters about relevant events, receive and automatically process appropriate actions in reaction to these events.

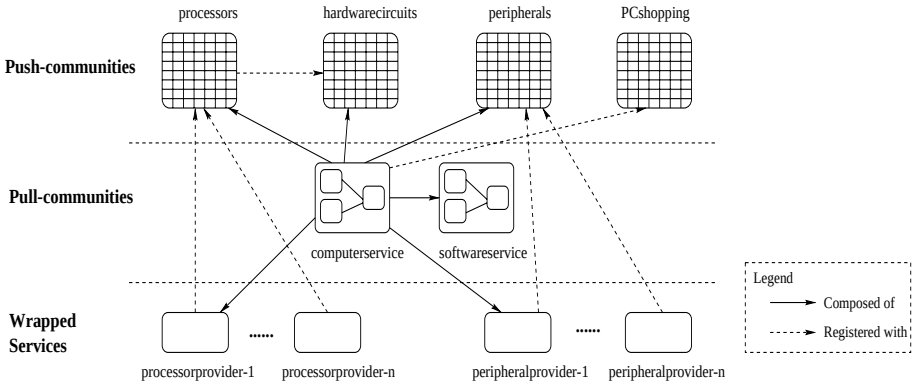

Figure 1: VE Services in WebBIS

In order to access the operations of proprietary/legacy services, we use *translators*. A translator is mainly used to map WebBIS operations into the format understood by the underlying proprietary service. The corresponding results are also translated into the format used by WebBIS. Assume that the `processorprovider` service provides an operation called `display_image()` that displays an image of a given processor. A corresponding translator, say `processorprovider_translator`, associates `display_image()` with a routine that calls a Java class method from the application that implements the underlying service.

We adopt the *ECA (Event-Action-Condition)* model [12] as a basis for the declarative specification of the business logic of VE services (i.e., constrains, contracts, policies, etc.) Briefly, the basic semantics of an ECA rule is as follows: when an event occurs, a condition is evaluated. If the condition evaluates to true, then an action is activated. Encoding the business logic of services as ECA rules is especially attractive to support the customization and increase in the flexibility of VEs. For instance, rules can be added, modified, or removed to reflect changes in both operational (e.g., server load) and market environments (e.g., user requirements).

2.2 Pull-communities

A *pull-community* provides a federated VE obtained from a collection of existing component services (wrapped services, push-, and other pull-communities). The assembly of a federated VE from existing services requires the location of the component services to be composed as well as their combination in a certain way. Requests to a pull-community are performed by invoking internal operations or translated into requests to component services. ECA rules are used to specify interactions between the community and its components.

As an illustration, the pull-community `computerservice` (Figure 1) offers sophisticated and complete computer configurations by outsourcing products from different wrapped services `processorprovider_1` and `peripheralprovider_1`. The `computerservice` community can also use operations of another pull-community `softwareservice` (offering computer software) to get the needed software (operating systems, compilers, etc.)

Although workflows can be used to specify the business logic of VEs, ECA rules offer several advantages over workflows. Workflows interleave the description of the content and capabilities of services, operations flow, and integration of components [20, 21, 13]. This makes the reuse and evolution of VE services very complex. In our approach, the clear separation of the components and ECA rules of a VE, makes changes in the business logic transparent to the underlying components.

2.3 Push-communities

When assembling a federated VE using a pull-community, the developer needs to locate and understand the meaning of the component services. Even if this approach seems especially suited to build an integrated VE from a small number of loosely coupled services, it is however inappropriate in environments characterized by the presence of a large number of dynamic services. We use the concept of *push-community* to address this issue.

A *push-community* (e.g., `processors`) describes the content and capabilities of a desired service without referring to any actual provider. In order to be accessible through a push-community, a provider (i.e, a wrapped service, pull- or push-community) can register with this community to (fully or partially) offer the desired service. This involves the definition of the *mappings* between properties and operations defined in the community and those defined in the service. For instance, the method `testperformance()` (resp., `buy_processor()`) of the push-community `processors` is mapped to the method `display_processor_benchmark()` (resp., `order_processor()`) of the wrapped service `processorprovider_1()`. A service can register with one or several push-communities of interest. It can also leave these communities at any time. In Figure 1, the wrapped services `processorprovider_1`, ..., `processorprovider_n` are registered with the push-community `processors` which is itself registered with the push-community `hardwarecircuits`. The pull-community `computerservice` is registered with the push-community `PCshopping`.

A push-community is typically used for the creation of on-the-fly VEs where providers form alliances to offer a desired service and then disband when it is no longer profitable to stay together. It creates a virtual space that provides a means to combine a collection of actual services (wrapped services, pull or other push-communities) into a single unit. In addition, a push-community allows dynamic selection of services and provides for orderly interaction with participant services. It may be organized around a common service offered by all providers. It may also be used by providers who offer parts of complex services. For example, if the push-community `processors` provides operations that allow searching and buying processors, a member of this community may register for both operations or alternatively for only one (e.g., searching processors).

In addition to providing a basis for composing a potentially large number of loosely coupled and dynamic component services, push-communities provide means for an ontological organization of the cyberspace. Each push-community is specialized in a single area of interest that essentially defines an *ontology*. In a nutshell, an *ontology* defines taxonomies based on the semantic proximity of terms [6]. A push-community provides domain-specific information as well as terms for interacting within the community and its underlying services. For example, `peripherals` describes a collection of actual services that offer online peripheral shopping such as monitors, keyboards, etc. Such an organization would aim at reducing the overhead of discovering services.

Modeling VE services as push-communities results in providing online marketplaces (e.g., computer manufacturing, car manufacturing, supply chain, healthcare, etc.) that facilitate the dynamic search of partners and on-the-fly collaboration with participants of the marketplaces. The selection of a partner is based on the available services, characteristics, organizational policies and resources that are needed to accomplish the integrated service.

3 Wrapping VE Services

We propose a uniform declarative language, called *WebBIS-SDL* (*WebBIS Service Definition Language*), for wrapping proprietary/legacy services and composing VEs. This language uses the concepts of objects and active rules as a basis for modeling VE services. This section focuses on wrapping VE services. The composition of VEs is presented in the next section. The provider first specifies the content and capabilities of a wrapped service in the *Provider-defined* class. This class is compiled by the WebBIS system into another class called the *WebBIS-extended* class. The latter is a sub-class of a generic class called *WrappedService* which is in turn a sub-class of the generic class *Service* (Figure 2). These generic classes provide a minimal set of features required for accessing, monitoring, and controlling services. A wrapped service is an instance of a WebBIS-extended class. In this section, we describe the main features of the Provider-defined (Table 1.(a)) and the WebBIS-extended (Table 1.(b)) classes for wrapped services.

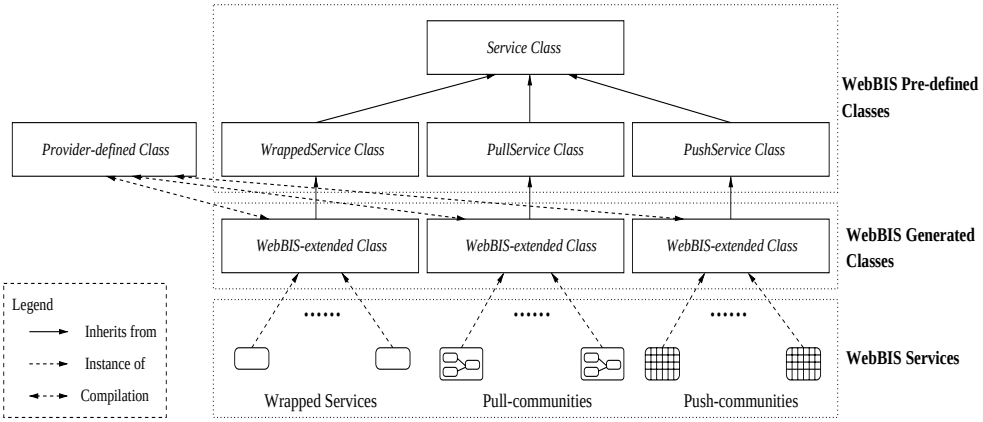

Figure 2: Service Classes in WebBIS

Wrapped Provider-defined Class `class-name`	**Wrapped WebBIS-extended Class** `class-name`
{ **P-properties** `list-of-properties`	{ **W-properties** `list-of-properties`
P-operations `list-of-operations`	**W-operations** `list-of-operations`
P-notifications `list-of-notifications`	**W-translators** `list-of-translators`
}	**W-rules** `list-of-rules`
	}

(a) Provider-defined Class (b) WebBIS-extended Class

Table 1: Wrapped Service Classes

3.1 Provider-defined Features

The provider-defined class is composed of three clauses (Table 1.(a)): *P-properties* (P for Provider), *P-operations*, and *P-notifications*. The *P-properties* clause describes service-specific information (e.g., `warranty` of a hardware part) expressed by attributes of the provider-defined class. For instance, the attribute `warranty` is defined using the following WebBIS-SDL statement:

P-properties
attribute `integer warranty;`
.....

The *P-operations* clause introduces the list of operations offered by a VE service. These operations are expressed by methods (the terms method and operation will be used interchangeably) of the provider-defined class. *Input* and *output* parameters are given for each method. A *pre-condition* is also specified. It defines the constraints that must be satisfied to use the method. In addition, an *activation mode* of the method can be specified. It can be either a `request`, `temporal`, or `both` mode. A `request` mode means that the method is activated only if explicitly requested (e.g., by end-users or in the business logic of another service). A `temporal` mode means that the method is activated automatically at a specified time (e.g., each first date of the month at noon). Note that the model we use to specify temporal modes is subject to future work. A `both` mode includes the two previous modes. By default, the activation mode of an operation is `request`.

The method `order_processor()` of the `processorprovider` class is, for example, defined using the following WebBIS-SDL statement:

method `order_processor`
    pre-condition `subscribed;`
    input `string product_name, string store_name;`
    output before `string delivery_note,` **after** `string orderID;`
    abort `string failed;`

The method `order_processor()` requires two input values: the product and store names. The use of this method is restricted to current subscribers. The output of this method is an identifier of the order. A method may also have alternative outputs. An abort output is an indication that the method has failed. Note that the value of some outputs may be produced before the completion of the execution. In this case, the definition of the parameter is pre-fixed by the keyword `before`. Outputs are by default `after` parameters (i.e., their values are produced after the completion of the execution). The value of the output parameter `delivery_note` can be available before the completion of the `order_processor()` method. Thus, a part of an operation result can be requested before its completion. This feature is particularly useful for long-lived services [23]. Since no activation mode is specified, the `request` mode is considered by default.

The *P-notifications* clause specifies the events that can be sent by a VE service to its requesters. These events can be used to inform the requester about the service state and situations produced during operation executions. In WebBIS, notification events correspond to observable state transitions (e.g., `notready, ready, frozen, running, completed, aborted`). Disclosing state information allows to capture service-specific behavior, provide a comprehensive modeling of VE interactions, and facilitate the coordination of VE operations as in workflow systems [20, 13]. For example, `processorprovider` would raise a notification event `order_processor_termination()` when the execution of the method `order_processor()` is completed. Each notification event has a name and a list of parameters and is an instance of a particular class called *Event*. Its values contain information that must be sent with the event. For example, the `order_processor_termination()` event has one parameter called `completion_status`. It indicates the output produced by the execution of `order_processor()`. This is specified using the following WebBIS-SDL statement:

notification `order_processor_termination`
    type `string completion_status;`

3.2 WebBIS-extended Features

The WebBIS-extended class includes four clauses (Table 1.(b)): *W-properties* (W for WebBIS), *W-operations*, *W-translators*, and *W-rules*.

W-properties

Each attribute declared in this clause is either generated from those defined in the provider-defined class (e.g., `warranty`) or inherited from the *WrappedService* class (e.g., `documentation`, `demonstration`, `servicestate`, `logmode`). A `documentation` is a human-readable description (e.g., textual or HTML document) about the VE service including legal conditions, agreements, etc. A `demonstration` (e.g., a Java applet that plays a video clip) provides means for understanding the content, capabilities, requirements and terminology of the VE. While

preparing documentation and demonstration involves additional overhead, we believe that providing these facilities is necessary for controlling the creation of accurate and high-quality VEs. The creation of a VE is accomplished mainly via a reuse process. So, it is extremely important to understand the semantics of a component service before incorporating it as a part in a VE. It is also important (if needed) for a consumer to understand the behavior of a service before requesting it.

The `servicestate` attribute determines the possible states of a VE service. It takes values from the set {`'notready'`, `'ready'`, `'frozen'`, `'running'`, `'completed'`, `'aborted'`}. Initially, the value of `servicestate` is `'notready'`. State transitions (e.g., from `running` to `aborted`) are caused by the execution of operations and controlled by ECA rules of the service. The `logmode` attribute determines the logging strategy. The logging concerns events such as operation invocations. The default value of this attribute is the empty set (i.e., no events will be logged).

Note that a provider can change the definition of an attribute. For example, the domain of the attribute `servicestate` can include other states (e.g., `'processorcompatibilitytested'`) to capture service-specific behavior. This state indicates that the processor compatibility has been tested, but the whole service is not completed yet.

W-operations

This clause introduces both *invoked* operations intended to be called synchronously and *notification* (or event-based) operations which can be asynchronously executed in reaction to notifications.

Invoked Operations. Invoked operations are either generated from operations defined in the provider-defined class or inherited from the *WrappedService* class. They include three types of methods: *request*, *monitoring*, and *control* methods.

Request methods are used to invoke the available operations (e.g., `start_operation()`, `get_operation_result()`). For instance, `start_operation()` starts the execution of an operation. Its input parameters are the name of the operation (e.g., `order_processor()`), the inputs of this operation, and the requester identifier. It returns the identifier of an invocation object that can be used to get the result of the operation, the execution status, and so on. *Monitoring* methods can be used to supervise the execution and measure the performance of a VE service (e.g., `get_service_status()`, `get_operation_status()`, `get_operation_cost()`, `get_operation_progress()`). The method `get_operation_cost()` can be used to estimate the cost of an operation execution. Its interpretation is provider dependent. For example, it may return the estimated execution time or the size of the results. *Control* methods (e.g., `open_service()`, `cancel_operation()`, `log_operation()`) can be used to start a service, preempt its execution, or keep a log about method executions.

Two other types of operations `get` and `set` are implicitly defined for retrieving and updating the service properties respectively. Default implementations for most of the operations are provided.

Notification Operations. Allowing notification events requires support for event detection, construction, and communication. The WebBIS-extended class includes a set of operations to deal with these issues. The `subscribe_to_notifications()` method is used to subscribe to the advertised notification events of a service. The `set_servicestate()` method is used to change the value of a service state (i.e., the `servicestate` attribute). The `event_notification()` is a remote method (in the sense of Java RMI [26] or remote calls in [1]). When activated, it notifies the target service about the occurrence of an event (e.g., change the value of an attribute that determines event occurrence states). The `after_operation()` method is used to perform some post-processing of a given operation (e.g., call another operation). It is systematically triggered after the completion of the related operation. It is especially useful for adding specific processing to an operation whose source code is not accessible.

Recall that operation executions may cause state transitions. For instance, the execution of the method `open_service()` will trigger the initialization of the attribute `servicestate` to `'ready'` (i.e., transition from the state `notready` to `ready`). As notification events correspond to observable state transitions, it is important to maintain the list of operations whose execution may trigger the occurrence of this event. This task is performed by the `notification_triggers()` operation.

Now, let us explain the technique that allows to detect, construct, and communicate notifications. Given an event `E` and a service `S`, the completion of an operation `op` that belongs to the list `S.notification_triggers(E)`, triggers the operation `S.after_operation(op)` which performs some post-processing of `op` including checking the occurrence of `E` (based upon the results of `op`). If the event is detected then the operation `S.set_servicestate(new-state)` is invoked. This triggers the initialization of the object that represents the

notification event (an instance of the class *Event*), and the invocation of `event_notification()` for each service that belongs to the list of the subscribers to the notification event `E`. In fact, the proposed technique is encoded as a set of ECA rules.

W-translators

The business logic of a wrapped service includes the declaration and instantiation of translators. As indicated in Section 2.1, a translator is used to map WebBIS operations into the format understood by the underlying proprietary service, and the corresponding results into the format used by WebBIS. A translator must supply a concrete implementation for, at least, the following operations: `open_service()`, `start_operation()`, and `close_service()`.

Another function of a translator is to support event notifications. As explained before, WebBIS offers a set of methods to monitor and control notifications. These methods provide the necessary features to customize a translator behavior in order to support notifications (e.g., adding post-processing code to the implementation of a method via the `after_operation()` method). The proposed technique is independent of the underlying services. Whether an underlying service has built-in notification capabilities or not, the interface of the WebBIS-extended class is flexible enough for a translator to provide notification support. However, this technique does not use the built-in notification capabilities of the underlying services (e.g., services based upon OMG Event Service [26], JavaBeans [26], trigger-enabled DBMSs such as Oracle, Informix or DB2). We are currently investigating the use of techniques in the area of distributed event management (e.g., event management capabilities of underlying services [12], change detection [24], etc.)

WebBIS allows any number of translators for a given proprietary/legacy service. These translators may use alternative resources of the proprietary service. For example, if a translator provides delayed processing of a specific operation, we might prefer to use a translator that provides faster processing, although both provide the same functionality. Allowing multiple translators for a given proprietary/legacy service gives a strong abstraction for VE flexibility and customization. In essence, this provides a mechanism to incorporate new behavior or to replace the behavior of a VE service. The selection of a specific translator is specified by the ECA rules (e.g., use one translator instead of another if certain conditions are satisfied).

W-rules

Typically, ECA rules specify constraints on the service properties and methods (e.g., access rights). They also specify the reaction to requests and responses from translators including notification support. Their general form is as follows:

rule `rule-name`
  event `event` **condition** `condition` **action** `action`

An event is a method invocation (e.g., request method, control method, monitoring method), a notification, a service state transition (e.g., termination of an action), or a combination of events via logical operators (AND, OR, NOT). A condition is a boolean expression over the service state. An action can be a method invocation, notification, or a group of actions to be sequentially or concurrently executed. For example, assume that the `processorprovider` class contains a method called `testcompatibility()` and that the invocation of this method triggers the execution of a local native application via the translator `processorprovider_translator`. The following WebBIS-SDL statement is used to specify how the system reacts when this method is invoked:

rule R1
  event `start_operation(testcompatibility());`
  action `processorprovider_translator::start_operation(testcompatibility());`

4 Composing Virtual Enterprises

The WebBIS-SDL language provides primitives to create pull-communities (resp., push-communities). The provider first specifies the *Provider-defined* class which is compiled into the *WebBIS-extended* class. The latter is a sub-class of a generic class called *PullService* (resp., *PushService*) which is in turn a sub-class of the generic class *Service*

(Figure 2). A pull-community (resp., push-community) is an instance of a WebBIS-extended class. The provider-defined class of pull and push-communities is defined in the same manner as for wrapped services. The difference between wrapped services and pull/push-communities lies in the WebBIS-extended class. In the following, we focus on presenting features that are specific to pull and push-communities.

Pull WebBIS-extended Class `class-name`	**Push WebBIS-extended Class** `class-name`
{ **W-properties** `list-of-properties`	{ **W-properties** `list-of-properties`
W-operations `list-of-operations`	**W-operations** `list-of-operations`
W-components `list-of-components`	**W-rules** `list-of-rules`
W-rules `list-of-rules`	}
}	

(a) Pull WebBIS-extended Class (b) Push WebBIS-extended Class

Table 2: WebBIS-extended Classes for Pull and Push-Communities

4.1 Federated Virtual Enterprises

As pointed out in Section 2, pull-communities provide a means to build federated VEs. The WebBIS-extended class of a pull-community (Table 2.(a)) is composed of four clauses: *W-properties*, *W-operations*, *W-components*, and *W-rules*. In this section, we focus on *W-components* and *W-rules*. The other clauses are described in the same manner as for wrapped services. Note also that some of the ECA rules, we present here, are relevant to all types of services (i.e., wrapped services, pull- and push-communities).

The *W-components* clause introduces the WebBIS compliant components of a federated VE. The composition of `computerservice` from `motherboards` and `processors` is specified using the following WebBIS-SDL statement:

```
W-components
        component motherboards subscribe all;
        component processors subscribe all;
        .....
```

Subscription to notifications is introduced by the clause *subscribe*. The keyword `all` is used to specify that the VE subscribes to all notifications of the component service. Subscription to an event specified by giving the name of the events and a constraint. The constraint is a condition on the values of the event parameters. It is used to filter the event instances that the subscriber is interested to be notified about. Thus, when an instance of the event occurs at the component side, the subscriber is notified only if the instance satisfies the subscription constraint.

A federated VE uses ECA rules to invoke operations provided by its partners (i.e., components) and coordinate their execution. In our example, to process a customer order, `computerservice` invokes the `checkavailability()` method of `motherboards` and `processors`. This can be specified by the following rule in `computerservice`:

```
rule R2
    event start_operation(order_computer());
    action processors::start_operation(checkavailability());
           motherboards::start_operation(checkavailability());
```

It should be noted that, it is possible to choose among multiple components to perform a method. The selection of a specific component for performing a method is specified by the ECA rules (similarly to choosing a translator in a wrapped service).

4.2 On-the-Fly Virtual Enterprises

The WebBIS-extended class (Table 2.(b)) of a push-community includes three clauses, namely *W-properties*, *W-operations*, and *W-rules*. In contrast to pull-communities, push-communities do not explicitly refer to WebBIS

components. However, a push-community can subscribe to notification events of other services including its members and other push-communities to which the community refers to. Reaction to notifications from the underlying services is specified in ECA rules of the push-community. In what follows, we introduce the main features of push-communities. We focus on the novel aspects designed specifically to provide a basis for VE marketplaces.

Ontological Organization of Virtual Enterprises

The WebBIS-extended class of a push-community contains a set of properties that constitute an important part of the metadata that is used to facilitate the discovery of VE services. They provide a means for an ontological organization of the available service space. These properties include the attributes `domain_type`, `synonyms`, `overlapping_communities`, `members`, and `sub-communities`, which are inherited from the *PushService* class.

The attribute `domain_type` is a string that defines the meaning of the push-community (e.g., selling peripherals for the push-community `peripherals`). It provides a means to dynamically clump consumers and providers together based on a common domain of interest (e.g., computer manufacturing, car manufacturing, supply chain, healthcare, etc.) The attribute `synonyms` describes the set of alternative descriptions of each domain (e.g., CPU is a synonym of processors).

The attribute `overlapping_communities` contains all push-communities whose domains overlap with the domain of the current community. It defines an intersection relationship between the related communities. If a push-community is not relevant to the received request, requesters can use the attribute `overlapping_communities` to find other push-communities. It should be noted that, it is the responsibility of a push-community provider to identify the other push-communities that have related areas of interests and initialize the value of the attribute `overlapping_communities`. For instance, assume that a user is looking for push-communities that are relevant to the specific domain of interest 'selling processors'. The system finds `processors` as a relevant push-community. Let us assume also that, the value of the attribute `overlapping_communities` of `processors` is {'`hardwarecircuits`'}. This attribute can be used to find the push-community `hardwarecircuits` if the user is not interested in the community `processors`. The attribute `overlapping_communities` is used to provide a peer-to-peer topology for connecting push-communities with similar domains. Communities that are connected together form a consortium. Communities in a consortium can forward requests to each other. This topology ensures that if one process community does not have the capabilities to process a given request, the request can be forwarded to another community in the consortium.

The attribute `members` represents the collection of services which are members of the push-community. The attribute `sub-communities` describes specialization relationship between push-communities. Note that services which are members of a given push-community are not necessarily members of its super-community. However, a push-community can register its members with its super-community.

Joining Push-communities

Providers can, at any time, locate and register with a push-community of interest using the `register_with_community()` method (which is inherited from the *PushService* class). The method `register_with_community()` is the interface a service uses to register with a push-community. Providers use the WebBIS-QL to locate push-communities of interest. WebBIS-QL, an SQL-like language, provides primitives for educating requesters about the available space, exploring service relationships, locating services based on constraints over their metadata, as well as accessing and manipulating VE services. Discovery and selection of services in WebBIS-QL is based on semantics (e.g, ontological meta-data), quality of service, available control operations, logs, and notifications. Service descriptions are stored in XML-based repositories. Due to space constraints, the presentation of WebBIS-QL is outside the scope of this paper. Details about WebBIS-QL can be found in [4].

A service can register with one or several push-communities. This has the advantage that a service can still be available even if one of the communities this service is registered with, is not available. The registration of a service with a push-community requires to define the *mappings* between properties as well as operations. The mappings are stored as part of the value of the attribute `members` of the push-community. For instance, the service `processorprovider_1` can be registered with `processors` by using the following WebBIS-SDL statement:

```
Join Service source processorprovider_1 s target processors t
Mappings method t.testperformance() is s.display_processor_benchmark();
         method t.buy_processor() is s.order_processor();
         .....
```

The method `testperformance()` (respectively, `buy_processor()`) of the push-community `processors` is mapped to the method `display_processor_benchmark()` (respectively, `order_processor()`) of the wrapped service `processorprovider_1()`. It should be noted that registration may concern only a subset of the properties and operations of a push-community. By featuring registration to a specific part of a push-community, our approach allows the creation of push-communities which have several activities. Thus, VE services have the flexibility to register only for the activities they can provide. For instance, the community `peripherals` provides operations for searching and buying monitors. Some of the actual services can provide either searching or buying (but not both), and thus, register only for the part they can provide. A push-community provider can specify constraints that must be satisfied to be registered with its community. These constraints define the pre-condition of the `register_with_community()` method. For example, the `peripherals` push-community might require that the registration for the `buy_monitor()` operation requires the registration for the `search_monitor()` operation.

A push-community can also register with another push-community. By doing so, the members of the first push-community become members of the second push-community too. A push-community does not need to be sub-community of another to register with. For consistency reasons, the following constraint is defined as a part of the pre-condition of the `register_with_community()` method: a push-community cannot register with its sub-communities. This is natural as it is implicit that members of a push-community can be members of its super-community and not the opposite.

The registration of a service with a push-community involves a start-up cost to define the mappings. However, this cost is not significant because the provider has only to understand the specification of the push-community. WebBIS helps providers with mechanisms to document services in a way that makes them understandable to users.

Dynamic Selection of Services

A push-community can be seen as an online marketplace for all consumers and providers in a given domain. As such, it must provides support for dynamic selection of services through the marketplace. In our approach, a request to a push-community is performed by (i) finding a combination of actual services that satisfy the requester's requirements and (ii) invoking the appropriate operations of the relevant services. To this end, appropriate methods are defined in the class `PushService` including `query()` and `apply_operation()`. The method `query()` is the interface a push-community uses to search for members that satisfy some given conditions. The input parameters of this method include a string which represents a query in the WebBIS-QL language [4]. The method `apply_operation()` is the interface a push-community uses to ask the underlying services to perform an operation. The input parameters include the operation to be performed and the set of services which the push-community wants to apply the operation to.

For example, let us assume that the pull-community `computerservice` uses the push-community `processors` as a component instead of a wrapped service or a pull-community. The interesting part in this example is that actual services (e.g., `processorprovider_1`, `processorprovider_2`, ..., `processorprovider_n`) which are members of `processors` can be dynamically selected to perform operations (e.g., `search_processor()`, `order_processor()`) of `computerservice`. At runtime, it is possible to select an VE service according to a given criteria (e.g., quality of service, cost). This selection is specified in the ECA rules of `computerservice` by means of the `query()` method of `processors`.

5 Change Management in Virtual Enterprises

Virtual enterprises operate in a highly dynamic environment. To cope with changes in this environment, it is necessary to support the reconfiguration of services. Mechanisms are needed to enable the addition, modification, relocation, and deletion of services in an efficient and controlled manner. This aspect is particularly important for Web-based VEs, whereby both operational (e.g., server load) and market (e.g., changes of service availability, changes of user's requirements) environment are not predictable. In our case, providers may delete, modify, and relocate their services. For instance, a provider may delete an operation or an event from the definition of its service. Changes can be initiated to adapt the current service to actual business climate (e.g., economic, politic, organizational, or personal changes). All changes performed to a service should be propagated to other services that rely on it to ensure global consistency. For instance if a component service is deleted, operations or events of pull-communities depending on it become unavailable. A mechanism to propagate change propagation should be devised. This section focuses on change management for VEs within the WebBIS framework.

5.1 Monitoring Services

In our approach, change propagation is facilitated by means of meta-services called *monitoring services*. Monitoring services are pre-defined and extensible objects that surround each service. They contain operational knowledge such as location, availability, and change control policies related to actual services. They also provide operations for changing and monitoring services, subscribing to and notifying changes. Rules can be associated to change related events as a part of the operational knowledge of monitoring services. Each service has a monitoring service attached to it. Monitoring services reason and act upon evolution regarding services they are attached to. They communicate among themselves and with the system in order to manage change propagation. Thus, the combination of a service and its monitoring meta-object forms a synergy to model the life-cycle of that service.

A monitoring service maintains information about the availability of the related service. Service availability is of particular importance for managing VEs. During its lifespan, a VE service can be *available*, *temporarily unavailable* (e.g., due to a network problem or an explicit request from the service provider), or *permanently unavailable* (e.g., the service is deleted) [18]. A monitoring service contains an attribute called `availability_status`. This attribute takes values from the set {`'perm_unavailable'`, `'temp_unavailable'`, `'available'`}. Hence, the monitoring service may exist even after the deletion of the related service. The system can, for example, periodically check the availability of services, and delete all the monitoring services of the permanently unavailable services. A modification of the attribute `availability_status` results in generating an instance of the event `availability()` of the monitoring service. The reaction to this event may trigger some change operations as described in the following subsection (e.g., freezing or deleting the service). It should be noted that a monitoring service and its related service can be located in different locations.

5.2 Change Operations and Events

The evolution of a service is accomplished through *change operations* of the monitoring service. Table 3 summarizes basic change operations supported by WebBIS (for clarity reason, operation parameters are not specified). Change operations can be performed manually by service providers or automatically by monitoring services in reaction to changes in other services. A change *event* is associated to each change operation (Table 3). Monitoring services subscribe to and notify change events. The consequences of a change event occurrence are captured in rules that can be defined in the sender or/and requester of the event. High level services (push and pull-communities) that use other services must subscribe to the changes that they are interested in. Whenever a change event occurs, information about the corresponding change is notified to the subscribers. The subscribers react to the notified changes using their own change control policies via local rules. Thus, the reaction to changes can be customized to the peculiarities of each VE. In what follows, we use simple examples to illustrate change propagation.

Operation	*Meaning*	*Associated Event*
`freeze_service()`	The service is made temporarily unavailable	`service_frozen()`
`delete_service()`	The service is made permanently unavailable	`service_deleted()`
`resume_service()`	The service (frozen formerly) is made available	`service_resumed()`
`relocate_service()`	Change the location of a service	`service_relocated()`
`freeze_operation()`	The operation is frozen (it cannot be invoked)	`operation_frozen()`
`resume_operation()`	The operation (frozen formerly) is resumed	`operation_resumed()`
`add_operation()`	The operation is added in the service	`operation_added()`
`delete_operation()`	The operation is removed from the service	`operation_deleted()`
`modify_operation()`	The definition of the operation is changed	`operation_modified()`
`freeze_event()`	The event is frozen (it cannot occur)	`event_frozen()`
`add_event()`	The event is added to the service	`event_added()`
`resume_event()`	The event (frozen formerly) is resumed	`event_resumed()`
`delete_event()`	The event is deleted from the service	`event_deleted()`
`modify_event()`	The definition of the event is changed	`event_modified()`

Table 3: Change Operations and Events

Let us assume that a provider decides to freeze its service (e.g., to operate local changes). The operation

`freeze_service()` is used for this purpose (Figure 3). This operation allows the provider to freeze the service for a period of time starting at a specified date and time. The service can be frozen for an unlimited period of time and then resumed afterwards. A service is reported as temporarily unavailable once it is is frozen. Consequently, no requests are accepted from users. Different scenarios are possible for the running instances of the service. For example, they are allowed to finish their execution, canceled, or frozen. The appropriate strategy is defined as ECA rules of the monitoring service. The appropriateness of one scenario over another depends on the varying situations and the nature of services. Whenever a decision to freeze a service is taken, a corresponding `service_frozen()` event occurs. Note that the event can occur before freezing the service, in order to inform subscribers in advance. This event goes through a filtering process to extract values of the event parameters and determine the relevant subscribers of the event instance. Event parameters may encapsulate information about when the service will be frozen, for how long, the alternative services (if any) that can be used while the service is frozen, and so on. The identified subscribers are then notified. The information carried out with the event depends on what subscribers registered for. For instance, the monitoring service of the pull-community `computerservice` which uses `processorprovider_1` as a component may register for information on when `processorprovider_1` will be frozen, how long it will be frozen, and the alternative services that can be used while it is frozen. In contrast, the monitoring service of the push-community `processors` in which `processorprovider_1` is a member, may register for only information on when and how long `processorprovider_1` will be frozen. It should be noted that the subscribers react to the event using a local policy which specified as ECA rules. For instance, the monitoring service of `computerservice` may react by performing the operation `freeze_service()` on `computerservice`; whereas the monitoring service of `processors` may react by excluding `processorprovider_1` from its members while `processorprovider_1` is frozen.

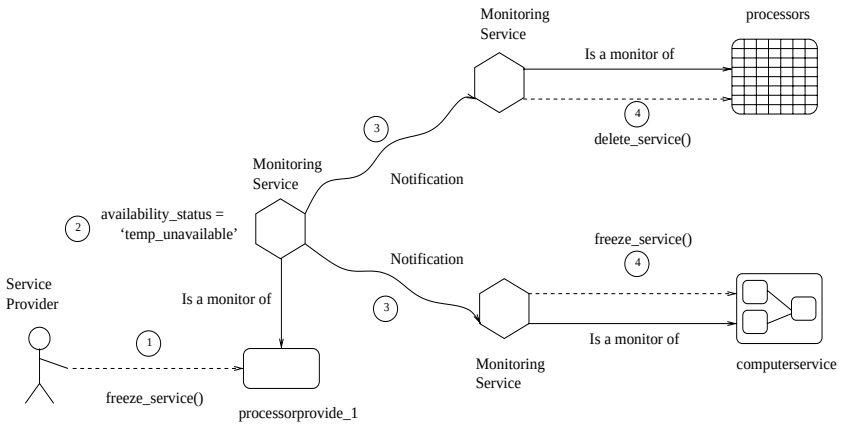

Figure 3: Change Propagation in WebBIS

The deletion of a service results in updating the attribute `availability_status` to make the service permanently unavailable and generating an instance of the event `service_deleted()`. The event parameters may encapsulate information about when the service will be deleted, the alternative services (if any) that can be used to provide the same functionality, etc. Similarly to freezing a service, an appropriate strategy must be devised and defined in the monitoring service to deal with the running instances of the deleted (or to be deleted) service. Subscribers react to this event using local policies. For instance, the monitoring service of the pull-community `computerservice` which uses `processorprovider_1` as a component reacts by freezing `computerservice`; whereas the monitoring service of the push-community `processors` in which `processorprovider_1` is member may react by removing `processorprovider_1` from the list of its members.

Similar procedures are performed when a service provider withdraws an operation from its capabilities, changes the definition of an operation, withdraws an event, or changes the definition of an event. For instance, the modification of an operation (resp. event) is notified to subscribers only if the new definition of the operation (resp. event) is not compatible with the old one in the sense of subtyping relationships (i.e., the new signature of operation/event is not a sub-type of the old one) [3]. Monitoring services of push-communities may not register for the deletion of an operation. Monitoring services of virtual enterprises may, for example, react to event deletion by

removing the rules associated to the event. The relocation of a service results in updating the repositories where the service is advertised to include the new location. Contrary to previous changes, the relocation of a service is not propagated to other services that depend on it (e.g., push-communities with which the service is registered).

In addition to the basic change operations mentioned in Table 3, monitoring services offer operations to deal with changes that may occur in the composition of a pull-community. The operations `delete_component()`, `add_component`, and `replace_component` allow to remove, add, and change a component respectively. The associated events are `component_deleted()`, `component_added()`, and `component_replaced()` respectively. Change propagation is also performed whenever a service provider uses the previous operations. For example, assume that (Figure 3) the pull-community `computerservice` is subscribed to the event `service_frozen()` with its component `processorprovider_1`. If the pull-community's provider decides to remove `processorprovider_1` from its components, the monitoring service of `processorprovider_1` is notified so that it can unsubscribe `computerservice` from the `service_frozen()` event.

6 Implemention Status

To illustrate the viability of the proposed architecture, we have implemented a computer manufacturing application (Figure 4). The implementation is currently built on top of WebFINDIT [9], a working prototype for describing, locating, and querying data in large network of databases. The wrapped services `IBM`, `Sony`, and `Philips` are registered with the push-community `peripherals`. `AMD` and `Intel` are registered with the push-communities `hardwarecircuits`, `processors` and `motherboards`. The pull-community `computerservice` outsources operations from the wrapped services and push-communities. Note that all services in Figure 4 are used for trial purposes. An evaluation in real Web applications to study the performance and scalability of our approach is subject to a future implementation.

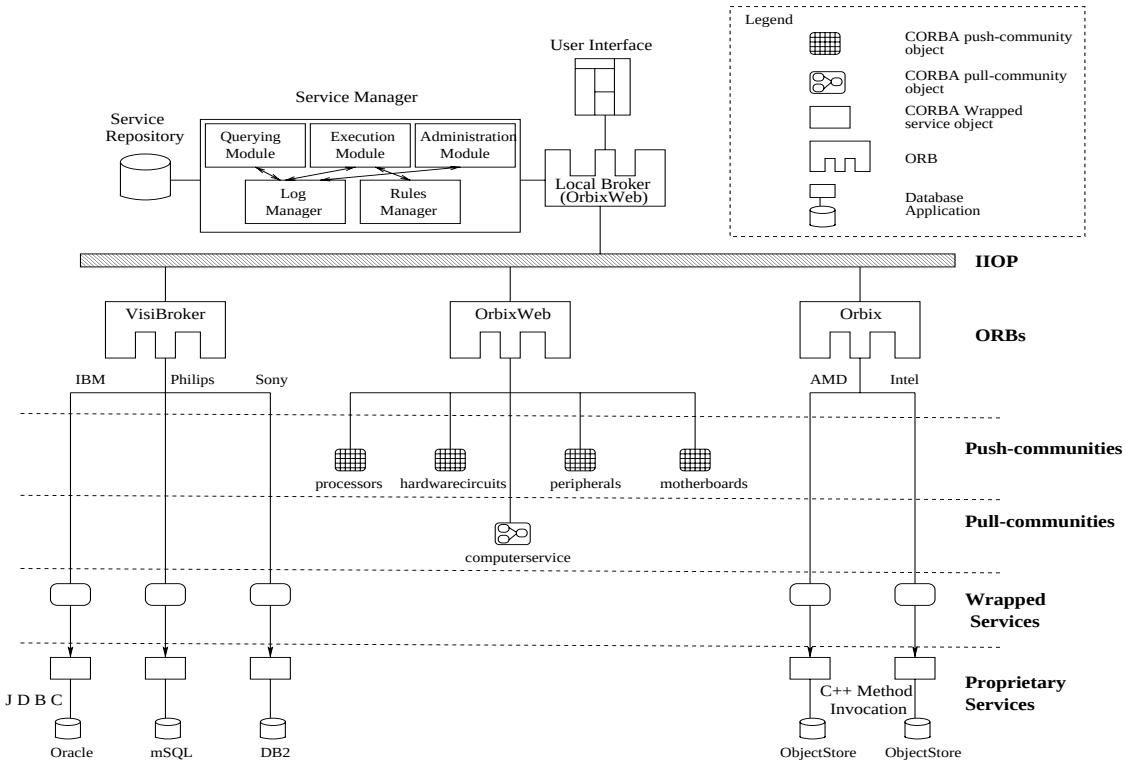

Figure 4: Implementation of a Computer Manufacturing Application

The user interface is implemented using Java applets and communicates with the service manager also written in Java. Users' requests are forwarded either to the *administration*, *querying* or *execution* module depending on

their type (Figure 4). These modules rely on the *log manager* to record the executed requests in a persistent log. The administration and querying modules implement WebBIS-SDL and WebBIS-QL languages respectively. The execution module is used to access, monitor, and control services. It interacts with the *rules manager* which is responsible of ECA rules management (each method invocation corresponds to an event). The service manager relies on a service repository (ObjectStore database) to handle the different users' requests. Although we are currently using a centralized repository with which all providers must register, a distributed solution (associating a repository with each push-community) is on-going. We are also investigating the use of XML-based repositories to store service descriptions.

The implementation uses three different IIOP compliant ORBs, namely VisiBroker for Java, OrbixWeb, and Orbix. These ORBs connect eight (8) services (push, pull, and wrapped services) registered as CORBA server objects. These objects are implemented in Java (VisiBroker for Java or OrbixWeb server objects) or C++ (Orbix server objects). Database applications are used to represent proprietary services. Information about these services (e.g., monitor size, CPU clock speed, price) is stored in five (5) relational (Oracle, mSQL, DB2) and object-oriented (ObjectStore) databases. Wrapped services that are registered as VisiBroker server objects access to database applications (written in Java) either locally or remotely using Java RMI. Access to relational databases that support a Java interface is provided via a JDBC bridge. Wrapped services that are registered as Orbix server objects access locally to object-oriented databases via C++ method invocation as both Orbix and ObjectStore support C++.

The interface allows users to choose among three functions: administrating, querying, and executing services. Figure 5 represents the interface during the definition of the push-community `processors`. The provider specifies service properties including `domain_type` (selling processors), `synonym` (processing units and CPUs), and `overlapping` (`motherboards` and `hardwarecircuits`) which are specific to push-communities. A user interested in registering with `processor` would typically start by discovering communities relevant to selling processors. For this purpose, a query based on domain is submitted through the service query tab. The user has also the possibility to understand the meaning and functionality of `processor` by accessing to the `documentation` and `demonstration` properties. Figure 6 (see Appendix) depicts a scenario where a VE (the `computerservice` pull-community) is defined. Assume that the VE developer wants to outsource services that provide monitors and processors. She/he would typically start by submitting queries to find push-communities related to the activity of selling monitors or processors (using the query tab). The system returns `processor` (defined in Figure 5) and `peripherals` as relevant communities. Then, the VE developer queries the system for the members of these two communities. The system returns `IBM`, `Sony`, and `Philips` as members of `peripherals` and `Intel` and `AMD` as members of `processor`. The VE developer may access to the documentation and demonstration of these services and then decide to select `IBM`, `Intel`, and `Sony` as business partners. For each component, the VE developer has the possibility to specify the notifications she/he wants to subscribe to. The default value is 'all' which includes subscription to all notifications.

7 Related Work and Concluding Remarks

There is a whole body of research related to service sharing in several fields including EDI [14, 28], electronic catalogs [14, 15], workflow [13, 19, 20, 21], and databases [7, 16].

EDI aims at offering an automatic and standard way to transfer data among business partners. EDI investigated static solutions that are appropriate if the services to be integrated belong to organizations with *long-term* and *static* trading relationships. Several existing efforts such as CBL (Common Business Library) [14] promote the use of XML to represent common interactions among VE services. While this approach promises features to help in building integrated VEs, the related efforts are still in their infancy.

Existing techniques for integrating electronic catalogs [14, 15, 28] typically rely on component-based middleware technologies such as COBRA and DCOM [26]. They focus on the integration of a small number of tightly coupled services. However, they present several limitations that make them ineffective when the service space is large and highly dynamic (e.g., the cost to set up a new business relationship is very high, it is not presently possible to dynamically integrate new VEs, etc.)

Current workflow systems are based on the premise that the success of an enterprise requires the management of business processes in their entirety. Indeed, an increasing number of organizations have already automated their internal process management using workflows and enjoyed substantial benefits in doing so. However, one of the most significant weaknesses of existing workflow systems, is the lack of flexible mechanisms to adequately cope with cross-enterprises relationships. Emerging projects in the workflow area focuses on interoperability among a known

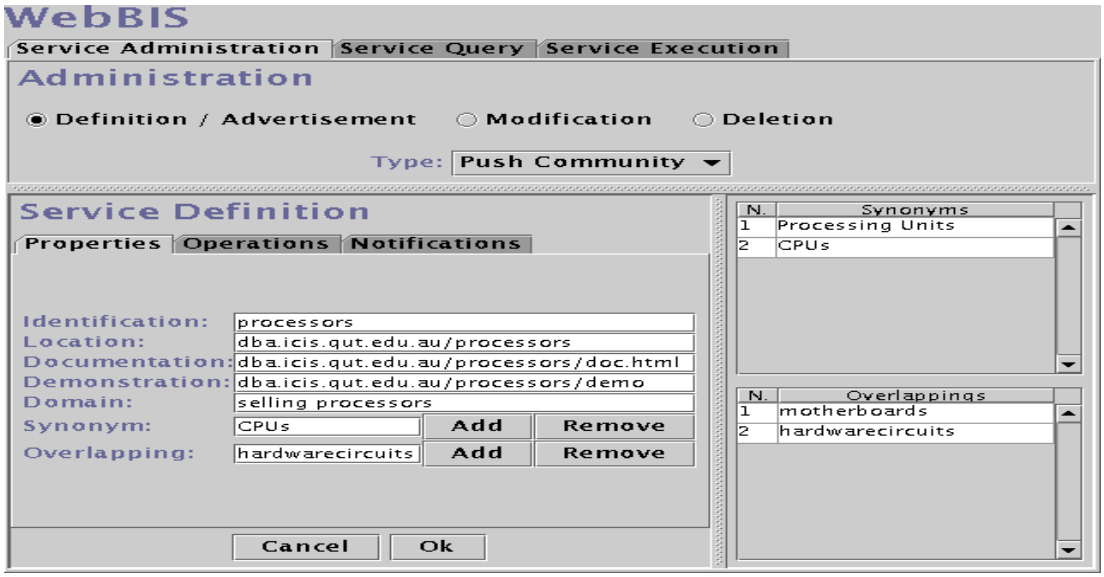

Figure 5: Defining a Push-Community in WebBIS

and small number of business processes [19, 20, 21, 13]. In addition, current workflow techniques are not flexible and rich enough to adapt to the ever expanding requirements of efficiently running modern Internet-aware applications. Current business processes within an organization are integrated and managed either using ERP systems such as SAP/R3, Baan, PeopleSoft or various workflow systems like IBM's MQ Series Workflow or integrated manually on demand-basis. The *eFLOW* [11] project proposes an interesting workflow-based approach for composing business processes. It focuses on the support of dynamic changes of processes to cope with exceptional situations as well as the customization of process execution to the customer's needs. eFLOW does not consider the issue of wrapping proprietary services. A similar approach for composing business processes is proposed in [20]. Both [11] and [20] do not consider the issues of change propagation and building online marketplaces for virtual enterprises.

Traditional techniques in multidatabases focus on data sharing among a small number of heterogeneous databases [7]. Web information integration systems [9, 7, 16] (e.g., SIMS, InfoSleuth, COIN, TSIMMIS, Information Manifold, WebSemantics, DISCO) propose interesting capabilities for wrapping data sources and providing uniform and declarative interfaces for querying and restructuring Web data sources. However, they do not consider the establishment of dynamic communities. In addition, they have not dealt with the integration of VE services.

Other projects, namely *ActiveViews* [1] and *CHAIMS* [23] are also related to our work. ActiveViews proposes a declarative language for specifying views that describe data and activities of different users working interactively on shared data which is stored in a centralized XML repository. ActiveViews primarily focuses on change management, i.e., propagating changes from the repository to the views and vise-versa. However, it does not consider the problem of sharing a large number of VE services. In the CHAIMS project, a protocol called *CPAM* (CHAIMS Protocol for Autonomous Megamodules) has been developed for wrapping autonomous services. WebBIS goes beyond the approach used in CPAM in that it provides support for dynamic communities, change propagation, and awareness which is important to deal with VEs in large and dynamic environments.

The major difference of the WebBIS approach compared to the above mentioned techniques lies in the *goals* and *means* of achieving service sharing over the Web. Simple access to and advertisement of VEs are key features when querying them on the Web. WebBIS presents an incremental and self-documenting approach. It provides support for educating users about the available service space. Since scalability and flexibility are of great importance in Web-based environments, WebBIS features appropriate abstractions. First, the ontological-like organization and segmentation of the cyberspace in meaningful subspaces makes service search and sharing more efficient. Second, the support of dynamic, transient, as well as long-term relationships enables more flexible integration of VE services. Third, the support of incremental and declarative integration provides for fast development and deployment of new VEs. Fourth, the support of notification management provides for pro-active deployment of VEs. Finally, WebBIS supports change monitoring and propagation to deal with the volatility and dynamics of

Web environments.

We note that the approach described in this paper is a first step towards devising a framework for building and managing virtual enterprises. More importantly, the proposed framework needs to be evaluated in real Web applications in order to study the performance and scalability of the proposed techniques. We are currently investigating another implementation prototype based on XML and EJB (Enterprise Java Beans) technologies.

Acknowledgments

The work of Prof. Athman Bouguettaya was supported by HP grant.

References

[1] Serge Abiteboul, Bernd Amann, Sophie Cluet, Anat Eyal, Laurent Mignet, and Tova Milo. Active views for electronic commerce. In *Proceedings of the 25th International Conference on Very Large Databases (VLDB)*, Edinburgh, Scotland, September 1999.

[2] R. Agrawal, M. Brodie, M. Carey, U. Dayal, J. Gray, Y. Ioannidis, J. Mylopoulos, H. Schek, K.-Y. Whang, and J. Widom. Future Directions of Database Research - Changes in the VLDB Conference PC Structure -. *Working Group of the VLDB Endowment, http://www.vldb.org/future.html*, 1998.

[3] B. Benatallah. A Unified Framework for Supporting Dynamic Schema Evolution in Object Databases. 8th International Conference Conceptual Modeling - ER'99, Paris, France. Springer-Verlag (LNCS series), November 1999.

[4] B. Benatallah, B. Medjahed, A. Bouguettaya, A. Elmagarmid, and J. Beard. Discovering and Integrating Web-based Services. Technical report, School of Information Systems, QUT, Brisbane, Australia, February 2000.

[5] P. Bernstein, M. Brodie, S. Ceri, D. DeWitt, M. Franklin, H. Garcia-Molina, J. Gray, J. Held, J. Hollerstein, H. V. Jagadish, M. Lesk, D. Maier, J. Naughton, H. Pirahesh, M. Stonebraker, and J. Ullman. The Asilomar Report on Database Research. Asilomar, California, September 1998.

[6] A. Bouguettaya, editor. *Ontologies and Databases*. Kluwer Academic Publishers (ISBN-0-7923-8412-1), 1999.

[7] A. Bouguettaya, B. Benatallah, and A. Elmagarmid. *Interconnecting Heterogeneous Information Systems*. Kluwer Academic Publishers (ISBN 0-7923-8216-1), 1998.

[8] A. Bouguettaya, B. Benatallah, L. Hendra, Beard J, K. Smith, and M. Ouzzani. World Wide Database - Integrating the Web, CORBA and Databases. In *Proceedings of the ACM SIGMOD'99 (Demo)*. ACM Press, June 1999.

[9] Athman Bouguettaya, Boualem Benatallah, Mourad Ouzzani, and Lily Hendra. Using Java and CORBA for Implementing Internet Databases. In *Proceedings of the 15th International Conference on Data Engineering*, Sydney, Australia, March 1999.

[10] Michael Brodie. Que Sera, Sera: The Coincidental Confluence of Economics, Business, and Collaborative Computing (Key note talk). ICDE99, Sydney, Australia, 1999.

[11] F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, and M.-C. Shan. Adaptive and Dynamic Service Composition in eFlow. Technical report, HP technical Report, HPL-2000-39, April 2000.

[12] C. Collet, T. Coupaye, and T. Svensen. Naos: Efficient and modular reactive capabilities in an object-oriented database system. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB)*, Santiago, Chile, September 1994.

[13] P. Dadam and M. Reichert, editors. *Proceedings of the Informatik'99 Workshop on Enterprise-wide and Cross-enterprise Workflow Management: Concepts, Systems, Applications*, Paderborn, Germany, October 1999.

[14] A. Dogac, editor. *ACM SGMOD Record: Special Issue on Electronic Commerce*, ACM SIGMOD RECORD. ACM, December 1998. 27(4).

[15] A. Dogac, editor. *Special Issue of Distributed and Parrallel Databases on Electronic Commerce*, Distributed and Parallel Databases Journal. Kluwer Publishers, 1999. 7(2).

[16] D. Florescu, A. Levy, and A. Mendelzon. Database Techniques for the World-Wide Web: A Survey. *ACM SIGMOD*, 27(3), September 1998.

[17] Forrester. *EMarketplaces Boost B2B Trade*. Forrester Research Inc., Cambridge, USA, February 2000.

[18] A. Gal and J. Mylopoulos. Towards Web-Based Application Management Systems. *to appear in IEEE Transactions on Knowledge and Data Engineering*, 2000.

[19] D. Georgakopoulos, editor. *Information Technology for Virtual Enterprises*, Proc. of the 9th Int. Workshop on Research Issues on Data Engineering. IEEE Computer Society, March 1999.

[20] D. Georgakopoulos and al. Managing Process and Service Fusion in Virtual Enterprises. *Information Systems*, 24(6):429–456, 1999.

[21] A. Geppert and D. Tombros. Event-based Distributed Workflow Execution with EVE. In *Proc. of Middleware '98 Workshop*, Sept. 1998.

[22] H. Lam and S. Su. Component Interoperability in a Virtual Enterprise using Events/Triggers/Rules. Vancouver, Canada, Oct. 1998. Proc. of OOSPLA '98 Workshop on Objects, Components, and Virtual Enterprise.

[23] M. Laurence, D. Beringer, N. Sample, and G. Wiederhold. CPAM: A Protocol for Software Composition. In *Advanced Information Systems Engineering (CAISE 11) (Editors: M. Jarke and A. Oberweis)*, volume 1626. Springer LNCS, Heidelberg Germanya, June 1999.

[24] L. Liu, C. Pu, and C. Hsu. Continual Queries for Internet Scale Event-Driven Information Delivery. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):610–628, 1999.

[25] R. Munz. Usage Scenarios for DBMS. In *Proceedings of the VLDB'99 (invited talk)*. Morgan Kaufmann Publishers, Inc., Sept. 1999.

[26] R. Orfali and D. Harkey. *Client/Server Programming with JAVA and CORBA*. John Wiley & Sons, Inc., 1997.

[27] A. Silberschatz, S. Zdonik, J. Blakeley, P. Buneman, U. Dayal, T. Imielinski, S. Jajodia, H. Korth, G. Lohman, D. Lomet, D. Maier, F. Manola, T. Ozsu, R. Ramakrishnan, K. Ramamritham, H. Schek, R. Snodgrass, J. Ullman, and J. Widom. Strategic Directions in Database Systems-Breaking Out of the Box. *ACM Computing Survey*, 28(4):764–778, December 1996.

[28] A. Whinston, editor. *Electronic Commerce: A Shift in Paradigm*, IEEE Internet Computing. IEEE, November 1997. Special Issue on Electronic Commerce 1(6).

A Additional Screen-shot for the Computer Manufacturing Application

Figure 6: Defining a Pull-Community in WebBIS