WebBIS: An Infrastructure for Agile Integration of Web Services

Brahim Medjahed\textsuperscript{1} Boualem Benatallah\textsuperscript{2} Athman Bouguettaya\textsuperscript{1} Ahmed Elmagarmid\textsuperscript{3}

\textsuperscript{1}Department of Computer Science
Virginia Tech, USA
{brahim,athman}@vt.edu

\textsuperscript{2}School of Computer Science and Engineering
University of New South Wales, Australia
boualem@cse.unsw.edu.au

\textsuperscript{3}Department of Computer Sciences
Purdue University, USA
ake@cs.purdue.edu

Abstract
The Web is changing the way organizations are conducting their business. Businesses are rushing to provide modular applications, called Web services, that can be programmatically accessed through the Web. Despite the tremendous developments achieved so far, one of the most important, yet untapped potential, is the use of Web services as facilitators for inter-organizational cooperation. This promising concept, known as Web service composition, is gaining momentum as the potential silver bullet for the envisioned Semantic Web. The development of such integrated services has so far been ad-hoc, time-consuming, and requires extensive low-level programming efforts. In this paper, we present WebBIS (Web Base of Internet-accessible Services), a generic framework for composing and managing Web services. We combine the object-oriented and active rules paradigms for such a task. We also provide a ontology-based framework for organizing the Web service space. We finally propose a peer-to-peer mechanism for reporting, propagating, and reacting to changes in Web services.

Keywords: Web services, Composition, Ontologies, Change Management.

1 Introduction
The Web has gained a significant momentum as the platform of choice to conduct business [14, 8]. Despite the tremendous economic growth the Web has elicited, Business-to-Business (B2B) E-commerce has not yet shown all of its potential. One of the most important, yet untapped potential, is the use of the Web as a facilitator for Web service outsourcing [41, 33]. A Web service is a set of related functions that can be programmatically invoked from the Web [41]. Examples of Web services span several application domains including B2B E-commerce (e.g., stock trading) and digital government (e.g., social services) [37, 34]. Web services are poised to be key players in the envisioned Semantic Web [7, 3, 42]. The ultimate goal of the Semantic Web is to transform the Web into a medium through which data can be shared, understood, and processed by automated tools.

The widespread adoption of XML-based standards including WSDL (Web Service Description Language), UDDI (Universal Description, Discovery, and Integration), and SOAP (Simple Object Access Protocol) has recently spurred an intense activity in industry and academia to address Web service research issues [33]. Web service composition is one of the most important and challenging issues [15]. It refers to the process of combining several Web services to provide a value-added service [41]. For example, composing French-to-English and English-to-Chinese language translation Web services would provide a
French-to-Chinese value-added service that translates from French to Chinese languages. We refer to such service as a composite service.

Web service composition is emerging as the technology of choice for building cross-organizational applications on the Web [2, 33]. This is mainly motivated by three factors. First, the adoption of XML-based messaging over well-established and ubiquitous protocols (e.g., HTTP) enables communication among disparate systems. Indeed, major existing environments are able to communicate via HTTP and parse XML documents. Second, the use of a document-based messaging model in Web services caters for loosely coupled relationships among organizations’ applications. Third, tomorrow’s Web is expected to be highly populated by Web services [12]. Almost every “asset” would be turned into a Web service to drive new revenue streams and create new efficiencies.

We identify the following challenging issues while composing Web services: interoperability, semantic discovery, and volatility. Interoperability with both internal and external services is central because composite services are built on top of autonomous, heterogeneous, and distributed Web services. This would require capabilities to seamlessly wrap legacy systems and compose existing services to create new ones. Heterogeneity is not restricted to the description on invocation of services. It occurs at different levels including, the interface, content, business logic, and back-end systems. The second challenge is the semantic discovery of Web services relevant for composition. Existing Web tools give little support for the semantic organization of the service space. This makes any effective use of services enormously complex. To date, little work has been done on proposing generic frameworks to describe and advertise services with semantically meaningful abstractions for the purpose of being efficiently discovered. A recent trend to empowering Web services with semantics is the use of ontologies. An ontology is defined as a formal and explicit specification of a shared conceptualization [7, 3, 42]. Ontologies were first developed in the artificial intelligence community to facilitate knowledge sharing and reuse [18]. Nowadays, they are increasingly seen as key to enabling semantics-driven data access and processing.

The success of enabling service composition relies on dealing with the dynamic and volatile nature of Web services. The Web service is highly dynamic. New services are expected to avail themselves on the Web. Businesses may need to form quick and short term relationships (e.g., to execute one transaction), and then disband when no longer profitable to stay together. This form of partnership does not assume any a priori trading relationship. A service would, in this case, need to dynamically discover partners to team up with. Volatility implies that Web services participating in a given composition may become unavailable at a later time. It is hence important to manage the evolution of those services. Composite services require flexibility to dynamically adapt to changes that may occur in their components. Businesses must be able to respond rapidly to changes where both operational (e.g., server load) and market (e.g., changes of service availability) environments are not easily predictable.

In this paper, we present the result of our work in the WebBIS (Web Base of Internet-accessible Services) project. WebBIS provides a generic framework for defining and managing composite services in dynamic environments. Several techniques have recently been developed for enabling Web service composition. These include eFlow [10], CMI [40], SELF-SERV [6], and XL [19]. These techniques provide little support for a semantic-aware composition of Web services. Additionally, they do not address the issue of change management. DAML-S is a recent trend towards empowering Web services with semantics [32]. It mainly focuses on describing the semantics of individual Web services. We adopt a complementary approach in WebBIS by providing means for the semantic organization of the Web service space. Service composition standardization efforts are under way such as, BPEL4WS (Business Process Execution Language for Web Services) [4], WSCL (Web Service Conversation Language) [25], WS-Coordination [26], WS-Transaction
[27], and Business Transaction Protocol (BTP) [36]. The issues addressed in this paper are complementary to those tackled by the aforementioned standards. Indeed, these standards mainly focus on specifying the orchestration of Web services, modeling conversations, and providing transactional support for composite services. WebBIS focuses on the ontological segmentation of the Web service space, dynamic composition of Web services, and management of changes in composite services. In particular, this paper’s contribution concentrates on the following issues:

- **Flexible composition** - WebBIS proposes a declarative approach for service composition. It is based on active rules [13], a widely used model in reactive and distributed systems, to encode the business logic of services. Active rules also allow the easy specifications of service interactions. The use of active rules is especially attractive to support customization and enable flexible service composition.

- **Dynamic relationships** - For service composition to scale to the Web, there is a need to cater for the creation of dynamic, transient, and/or long-term relationships among services. The support of dynamic composition will facilitate the establishment of on demand and real-time partnerships among businesses. In that respect, WebBIS enables composite services without the explicit advance knowledge of participants. Composite services are not necessarily bound to fixed sets of components.

- **Ontological organization** - A fundamental premise of our research is that Web users would have to be incrementally made aware of the available service space. WebBIS uses an ontological organization of the information space to filter interactions and accelerate service searches. In WebBIS, an ontology is defined as a common theme among a given set of services. WebBIS provides means to create virtual communities that bring together a variety of providers around a common interest.

- **Change monitoring** - In real business environments, the capabilities of existing services may be updated. WebBIS supports reporting and propagating service changes via the use of meta-services called change notifiers. Change notifiers are attached to the existing services. They communicate with their peers to manage change propagation. Change notifiers provide operations for modifying services, notifying changes, and subscribing to changes.

The remainder of this paper is organized as follows. Section 2 presents an overview of WebBIS fundamental concepts. Section 3 introduces the key WebBIS constructs for wrapping proprietary and legacy services. Section 4 describes WebBIS constructs for composing services. Section 5 details our approach to advertising and discovering Web services. Section 6 is devoted to change management issues. Section 7 describes WebBIS architecture and implementation. Section 8 gives a brief survey of the related work. We finally provide some concluding remarks in Section 9.

2 Modeling Web Services in WebBIS

The design and development of WebBIS follow two key guidelines: (1) the definition of an object-oriented framework for modeling services and (2) the use of active rules for composing and managing services. In the remainder of this section, we describe The WebBIS service model and illustrate the rationale behind the adoption of object-oriented and active rules paradigms.

2.1 WebBIS By Example

We define three types of Web services in WebBIS: service wrappers (or wrappers), pull-communities, and push-communities. Wrappers model simple Web services that do not rely on other Web services to fulfill
users’ requests. *pull-communities* and *push-communities* model composite services. The WebBIS framework categorizes composite services based on the relationships between their components. We consider both *static* and *dynamic* (or *on the fly*) relationships. We introduce the concepts of *pull-communities* and *push-communities* to model these relationships, respectively. To illustrate WebBIS main features, we consider a computer manufacturing application (Figure 1). The trading community includes geographically distant and autonomous software (e.g., compilers) and hardware (e.g., processors) companies offering heterogeneous services.

![Diagram of computer manufacturing application](image)

**Figure 1: Example – Computer Manufacturing Application**

**Service Wrappers** – Composing Web services requires understanding the content and capabilities of each component service. For instance, it is important to be aware of the different operations provided by a service, how to use its operations, and which messages are exchanged with other services on the Web. However, each service may use a different language (e.g., Java) to describe its capabilities. This dissimilarity in describing services makes the task of composing services cumbersome. To hide the heterogeneity of services and hence integrate proprietary or legacy applications, we wrap each basic (i.e., pre-existing) service by a *service wrapper*. A wrapper includes a set of *translators* that are mainly used to map WebBIS operations into a format understood by the underlying proprietary service. The corresponding results are translated back into the format used by WebBIS. Figure 1 depicts two pairs of service wrappers: *(Intel, AMD)* and *(IBM, Sony)* that surround proprietary services providing processors and monitors respectively. Interactions between wrappers and proprietary services are done via translators. For instance, assume that IBM’s wrapper provides an operation called *display_image()* that displays a sample image of a given monitor. A corresponding translator, say *IBM_translator*, associates *display_image()* with a routine that invokes an operation from the underlying Web service.
Pull-community – A pull-community allows the assembly of loosely coupled services with static and long-term relationships. Pull-communities are created by service composers. Components of a pull-community may be basic or composite. Requests to a pull-community are performed by invoking internal operations or translated into requests to component services. When defining a pull-community, the provider needs to locate and understand the capabilities of the component services. Figure 1 shows a pull-community named computers which offers computer configurations by outsourcing products from Intel and IBM Web services. The computers pull-community uses operations of another pull-community named softwarer service to get the needed software (operating systems, compilers, etc.) The selection of a specific component for performing an operation of computers pull-community is specified by an ECA (Event-Condition-Action) rule [13]. Briefly, the basic semantics of an ECA rule is as follows: when an event occurs, an action is executed if the corresponding condition is true.

Even if pull-communities are well suited to build an integrated service from a small number of loosely coupled Web services, a different approach is needed in dynamic environments that are characterized by the presence of a large number of services. To address this issue, we introduce the concept of push-community. A push-community is typically used for on-the-fly composition of services. Push-communities are typically created by industry providers consortia (e.g., computer industry, car industry). Using push-communities, providers would form alliances to offer a desired service and then disband when no longer profitable to stay together. 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. It provides domain-specific information and terms of interaction within the community and its underlying services. Providers can make their services accessible by joining push-communities corresponding to their activity of interest. Organizing services into push-communities aims at reducing the overhead of discovering services on the Web. An appropriate sub-division of the service space has the advantage of reducing the potentially large number of service combinations.

Push-community – Four push-communities, namely processors, hardwarecircuits, peripherals, and PCshopping, are used in the computer manufacturing application. For example, the push-community peripherals describes a collection of actual Web services that offer online peripheral shopping such as monitors and keyboards. Assume that the computers provider decides to partner with a new processor provider (e.g., AMD instead of Intel) for quality of service purposes (e.g., cost, time). A naive way would be to delve into a potentially large number of services offering processors. This solution is clearly inefficient and time-consuming. Alternatively, a better solution adopted in WebBIS is to enable dynamic selection of component services at run-time. For that purpose, computers uses the push-community processors as a component. Note that actual services such as AMD, Intel, and cyrix which are members of processors, can be dynamically selected to perform operations (e.g., searchprocessor(), orderprocessor() on the push-community computers.

2.2 WebBIS Classes

WebBIS defines an object-oriented approach to cater for the definition and composition of Web services. Web services (basic or composite) are objects defined by identifiers (URI - Universal Resource Identifier). Reuse and inheritance among services are clear benefits of this approach. Additionally, the object-oriented model is rich enough to specify complex structures, relationships, and interactions among services.

WebBIS provides a set of classes to allow the definition of wrappers, pull-communities, and push-communities. Portability is guaranteed by translating WebBIS classes into XML documents. WebBIS classes
include Provider-defined, Pre-defined, and Extended classes (Figure 3). The Provider-defined class contains features that are specific to the service being defined. For example, a Web service selling processors may have an attribute speed while another Web service offering monitors may have an attribute resolution. The structure of the Provider-defined class is similar for wrappers, pull-communities, and push-communities. The difference between the three types of services lies in the Extended classes. One Extended class is associated to each type of service. It is obtained by compiling the Provider-defined class. The Wrapper-Extended, Pull-Extended, and Push-Extended classes are sub-classes of the Wrapper, Pull, and Push Pre-defined classes, respectively. The Wrapper, Pull, and Push classes are sub-classes of a more generic class called Service. Pre-defined classes contain a minimal set of features for defining services. For instance, the Pull class has a clause named components that refers to the list of components of a composite service.

2.3 The WebBIS Service Model

In the “traditional” Web service model, providers define WSDL descriptions of their services and publish them in UDDI, a centralized (or replicated) repository of service descriptions. Consumers locate Web services of interest and invoke them using SOAP. This model has two major drawbacks. First, a centralized UDDI registry might result in a single point of failure and bottleneck for accessing and publishing Web services. Additionally, replicating the registry requires the UDDI nodes to exchange data with each other to maintain registry consistency. Second, the current version of WSDL does not model semantic features of Web services. Additionally, WSDL does not include operations for monitoring Web services such as checking the availability of an operation or the status of a submitted request. Finally, the process of dealing with changes is currently ad hoc and manually performed.
WebBIS builds on WSDL, UDDI, and SOAP standards to enable the discovery, description, and invocation of Web services. It extends the “traditional” Web service model to deal with the aforementioned issues (Figure 3). To avoid the problem of centralized or duplicated storage of service descriptions, we use distributed UDDI registries to advertise Web services. This enables high availability of Web services. Each push-community has a UDDI registry attached to it. Service wrappers and pull-communities are registered with push-communities. Their descriptions are thereby stored in the corresponding UDDI registries. Registering a Web service (wrapper or pull-community) with a push-communities enables the description of semantic features of that services (e.g., area of interest). The description of each wrapper or pull-community can be seen as an extension of a WSDL document. For example, the notions of $P$-properties and $E$-properties defined in a wrapper or pull-community are the “equivalent” of message parameters in WSDL. The notions of $P$-operations and $E$-operations correspond to the notion of port type in WSDL. We define additional constructs in WebBIS to capture the semantics of Web services and cater for the external manageability and management of changes in Web services.

2.4 Active Rules in WebBIS

Event-driven systems are becoming the paradigm of choice for organizing many classes of loosely coupled and dynamic applications [13, 21, 23, 28, 31]. In WebBIS, each service (wrapper, pull-community, and push-community) contains a set of notifications. These describe events that can be sent by the service to its requesters. Events are typically used to provide awareness about specific situations such as notifying users about the shipment of a product or tracking changes that may occur in services. To capture and react to events, each service also includes a set of rules. We adopt the ECA (Event-Condition-Action) (or active) rules model [13]. In WebBIS, ECA rules are mainly used to:
• Encode the business logic of services (constraints, contracts, policies, etc). This is especially attractive to support the customization of services. Indeed, ECA rules allow the incremental definition of services and flexibly adapt to changes in that environment. Rules can be added, modified, or removed to reflect changes in both operational and market environments.

• Specify interactions between a pull-community and its components. Although workflows may be used to specify the business logic of composite services, ECA rules offer several advantages over workflows. Workflows usually interleave the description of the capabilities of services, operations flow, and integration of components [21, 23]. This makes the reuse and evolution of services somewhat difficult. In our approach, the clear separation of the components and ECA rules allows for the transparent changes in the business logic of the underlying components.

• Propagate and react to changes that may occur in services (e.g., a Web service operation is no longer available). Whenever a change event occurs, information about the corresponding change is sent to other services that rely on it. These services react to the notified changes using their own change control policies via local ECA rules. The event part of an ECA rule refers to change notification. The actions part allows for the specification of change control policies. The use of conditions allows the specialization of these policies depending on pre-defined parameters.

3 Describing Service Wrappers

WebBIS-SDL introduces primitives for defining service wrappers. Providers first specify the content and capabilities of their service wrappers in the Provider-defined class. This class is compiled by the WebBIS system into the Wrapper-Extended class (Figure 3). A service wrapper is an instance of the Wrapper-Extended class. In this section, we describe the main features of the Provider-defined and Wrapper-Extended classes.

3.1 Provider-defined Features

The Provider-defined class is composed of three clauses: P-properties (P for Provider), P-operations, and P-notifications. The general structure of this class is as follows:

```
Provider Class class-name
{ P-properties list-of-properties
  P-operations list-of-operations
  P-notifications list-of-notifications }
```

The P-properties clause contains a list of properties. Each property describes service-specific information. It is expressed by an attribute of the provided class. For instance, the attribute warranty defines the duration during which a given hardware part is warranted. This attribute is defined using the following WebBIS-SDL statement:
**P-properties**

    property integer warranty;

The *P-operations* clause introduces the list of operations offered by a service. These operations are expressed by methods of the *Provider-defined* class. Each method has *input* and *output* parameters. A *pre-condition* is also specified to define the constraints that must be satisfied to use the method. In addition, an *activation mode* of the method may be specified. It may be either a *request*, *temporal*, or both. A *request* mode (default 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 day of the month at noon). A *both* mode includes the two previous modes.

The method *order_processor()* of the *Intel* class is defined using the WebBIS-SDL statement presented below:

```
method order_processor
    pre-condition subscribed;
    input string product.name, stringstore.name;
    output before string delivery.note after string orderId;
    abort string failed;
```

This method 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, part of an operation result can be requested before its completion. This feature is particularly useful for long-lived services [29]. 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 service to its requesters. These events can be used to inform the requester about the service state and situations that occurred during operation executions. In WebBIS, notification events correspond to observable state transitions (e.g., *notready, ready, frozen, running, completed, aborted*). Disclosing state information allows capturing service-specific behavior, providing a comprehensive modeling of service interactions, and facilitating the coordination of operations as in workflow systems [21]. For example, *Intel* 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 that must be sent with the event. A notification event is an instance of a particular class called *Event*.

For example, the *order_processor_termination()* event, presented below, has one parameter called *completion_status*. This parameter indicates the output produced by the execution of *order_processor()*.

```
notification order_processor_termination
    type string completion_status;
```
3.2 Extended Features

The Wrapper-Extended class includes four clauses: E-properties (E for Extended), E-operations, E-translators, and E-rules. The general structure of this class is as follows:

**Wrapper Extended Class**

```
class-name
{ E-properties list-of-properties
  E-operations list-of-operations
  E-translators list-of-notifications
  E-operations list-of-operations
  E-rules list-of-notations }
```

3.2.1 Properties

Each attribute declared in the E-properties clause is either described in the Provider-defined class (e.g., warranty) or inherited from the Wrapper pre-defined class. Examples of inherited attributes include documentation, demonstration, servicestate, and logmode.

The documentation attribute is a human-readable description (e.g., textual or HTML document) about the service including legal conditions, agreements, etc. The demonstration attribute provides means for understanding the content, capabilities, requirements, and terminology of the service. An example would be a Java applet that plays a video clip. The servicestate attribute determines the possible states of a service. It takes its 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. This means that, by defaults, 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.

3.2.2 Operations

The E-operations clause introduces two types of operations: invoked and notification operations. Invoked operations are used to start, monitor, and control the execution of service operations. Notification operations deal with aspects related to event notifications.

**Invoked Operations.** Invoked operations are either generated from operations defined in the provider-defined class or inherited from the Wrapper class. They include three types of methods: request, monitoring, and control methods. Request methods are used to invoke the available operations. They include the start-operation() and get-operation-result() methods. For example, 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, etc. Monitoring methods are used to supervise the execution and measure the performance of a service. To be effectively monitored by external partners, a service must be instrumented in a way that facilitates the supervision of its
execution (e.g., state changes), measurement of its performance (e.g., time and cost), and control of its execution (e.g., cancelling particular requests). This means that services must be visible to each other. Monitoring methods include `get_operation_cost()`, `get_service_status()`, `get_operation_status()`, and `get_operation_progress()`. For instance, `get_operation_cost()` is 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 are used to start a service, preempt its execution, or keep a log about method executions. Examples of control methods include `open_service()`, `cancel_operation()`, and `log_operation()`.

Note that two other types of methods, `get` and `set`, are defined for retrieving and updating the service properties, respectively. Also, default implementations for most of the operations are provided.

**Notification Operations.** Allowing notification events requires support for event detection, construction, and communication. The `extended` class contains a set of operations called notification operations to deal with these issues. Example of such operations include `subscribe_to_notifications()`, `set_service_state()`, `event_notification()`, `after_operation()`, and `notification_triggers()`. The `subscribe_to_notifications()` method is used to subscribe to the advertised notification events of a service. The `set_service_state()` method is used to change the value of a service state (i.e., the `service_state` attribute). The `event_notification()` is a remote method (in the sense of 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. The `notification_triggers()` method returns the list of operations whose execution may trigger the occurrence of a given event (specified as a parameter). Note that the execution of an operation may cause a state transition. For instance, the execution of the method `open_service()` triggers the initialization of the attribute `service_state` to 'ready' (i.e., transition from the state `notready` to `ready`).

Now, let us explain the technique that is used to support notifications. This technique is encoded as a set of ECA rules. Assume that Intel has an operation `order-processor()` and a notification `order-processor-termination`. Assume that `order-processor()` belongs to the list Intel. notification-triggers `order-processor-termination`. The completion of `order-processor()` triggers the execution of the operation `S.after_operation(order-processor())`. This operation performs some post-processing of `order-processor()` including checking the occurrence of `order-processor-termination`. If the event is detected then the operation `S.set_service_state('completed')` 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 the intel Web service.

### 3.2.3 Translators

The business logic of a service wrapper includes the declaration and instantiation of translators. 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 notifica-
tions. For example, it is possible to add 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 `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, 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 [13], change detection [31]).

WebBIS allows any number of translators for a given proprietary 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 service gives a strong abstraction for service flexibility and customization. In essence, this provides a mechanism to incorporate new behavior or to replace the behavior of a 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).

3.2.4 Notifications and Rules

The `E-notification` clause is generated from the list of notifications specified in the `Provider-defined` class. These notifications may be used to define a set of `ECA rules` that composed the `E-rules` clause. 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.

Each ECA rule contains an `event`, `condition`, and `action` part. An `event` is a method invocation, a service state transition (e.g., termination of a Web service operation), 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, a 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 Web service operation via the translator `IntelTranslator`. The following WebBIS-SDL statement is used to specify how the system reacts when this method is invoked:

```plaintext
rule R1
  event start_operation(testcompatibility())
  action Intel.Translator::start_operation(testcompatibility());
```

4 Composing Web Services in WebBIS

WebBIS-SDL provides primitives to create pull-communities and push-communities. Pull-communities allow the creation of `federated` services. The term `federated` refers to composite services with `static` and `long-term` relationships. Push-communities allow `on-the-fly` composition of services on the Web. The term `on-the-fly` refers to composite services with `dynamic` and `transient` relationships.

Pull-communities (resp. push-communities) are defined by first specifying the `Provider-defined` class. This class is compiled into the `Pull-Extended` (resp. `Push-Extended`) class(Figure 3). Pull-communities (resp. push-communities) are instances of the `Pull-Extended` (resp. `Push-Extended`) class. It is important
to recall that the Provider-defined class is defined in the same manner as for service wrappers. In the following, we focus on presenting features that are specific to pull-communities and push-communities.

4.1 Pull Composition

The Pull-Extended class is composed of five clauses: E-properties, E-operations, E-components, E-notifications, and E-rules. In this section, we focus on the E-components and E-rules clauses. The other clauses are described in the same manner as for service wrappers. Note also that some of the ECA rules we present here, are valid for all types of WebBIS services. The general form of the Pull-Extended class is as follows:

\[
\text{Pull Extended Class class-name} \\
\{ \text{E-properties list-of-properties} \\
\text{E-operations list-of-operations} \\
\text{E-components list-of-components} \\
\text{E-notifications list-of-notifications} \\
\text{E-rules list-of-rules} \}
\]

The E-components clause introduces the WebBIS compliant components of federated services. The composition of computers from motherboards and processors is specified using the following WebBISSDL statement:

\[
\text{E-components} \\
\text{component motherboards subscribe all;} \\
\text{component processors subscribe all;}
\]

Subscription to notifications is introduced by the clause subscribe. The keyword all is used to specify that the pull-community subscribes to all notifications of the component service. Event subscription is specified by giving the name of the event and a constraint on the event parameters. The constraint 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, subscribers are notified only if the instance satisfies the subscription constraints.

Pull-communities use ECA rules to invoke operations provided by their components and coordinate their execution. Indeed, ECA rules enable to specify the control flow of the pull-community. More precisely, ECA rules allow to specify the events that may guide the execution of the pull-community, which methods to invoke and from which components, and the order in which the components are invoked. For instance, the computers pull-community receives the event start_operation(order_computer()) to start processing a customer order. This event would trigger the invocation of the checkavailability() method of Intel and IBM. This can be specified by the rule described below. 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 service wrapper).
rule R2
  event start_operation(order_computer())
  action Intel_translator::start_operation(checkavailability());
                 peripherals::start_operation(checkavailability());

As illustrated in this example, the invocation of a pull-community method may involve the invocation of one or several methods belonging to its components. To make this possible, the pull-community exchanges a set of messages with its components. These messages describe the flow of data (i.e., input and output parameters) necessary to execute component methods.

4.2 Push Composition

The Push-Extended class is composed of four clauses: E-properties, E-operations, E-notifications, and E-rules. In contrast to pull-communities, push-communities do not explicitly refer to WebBIS components. However, a push-community P can subscribe to notification events of other services including its members and other push-communities P refers to. Reaction to notifications is specified in ECA rules of the push-community. The general form of the Push-Extended class is as follows:

    Push Extended Class class-name
    { E-properties list-of-properties
      E-operations list-of-operations
      E-notifications list-of-notifications
      E-rules list-of-rules }

4.2.1 Registering Web Services with 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 Push class). 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. Registration with a push-community requires to define the mappings between properties as well as operations using the following WebBIS-SDL statement:

    Join Service source Intel S target processors T
    Mappings method T.testperformance() is S.display_processorbenchmark();
             method T.buy_processor() is S.order_processor();

    The method testperformance() (resp. buy_processor()) of the push-community processors is mapped to the method display_processorbenchmark() (resp. order_processor()) of the service wrapper Intel. 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, 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 Web 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 the 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 be registered with another push-community. By doing so, the members of the first push-community become members of the second push-community too. For example, the service wrappers Intel and AMD are registered with the push-community processors which is itself registered with the push-community hardwarecircuits (Figure 1). Note that the registration of a service with a push-community involves a start-up cost to define the mappings. The cost and effort to support new relationships is an important factor for providing scalable composition of services. In WebBIS, 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.

### 4.2.2 Ontological Support for Push-Communities

The Push-Extended class contains a set of properties used to facilitate the discovery of services. These properties provide means for an ontological organization of the available service space. We use the emerging DAML+OIL language for describing the proposed ontology for push-communities [24]. DAML+OIL builds on earlier Web ontology standards such as RDF and RDF Schema and extends those languages with richer modeling primitives (e.g., cardinality). It adopts an object oriented approach, describing ontologies in terms of classes, properties, and axioms (e.g., subsumption relationships between classes or properties) [24].

We give in Figure 4 a subset of the push-community ontology specified in DAML+OIL. The first four elements define the push_community, identifier, domain_type, and synonym classes. The fifth and sixth elements define relationships (or properties) between push_community and the other classes. The seventh element specifies relationships between domain_type and synonym classes. The last two elements specifies cardinality constraints on the different properties (e.g., a push-community has a one single identifier).

The domain_type attribute is a string that conveys the meaning of a push-community (e.g., selling peripherals for the push-community peripherals). It also provides a means to cluster consumers and providers together based on a common domain of interest (e.g., computer manufacturing, healthcare). The synonyms attribute contains the set of alternative descriptions of each domain (e.g., CPU is a synonym of processors). The members attribute represents the collection of services which are members of the push-community.

The overlapping_communities attribute contains all push-communities whose domains overlap with the domain of the current community. It defines an intersection relationship between the related communities. This attribute 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 a community cannot process a given request for one reason or another, the request is forwarded to another community in the consortium. It should be noted that it is the responsibility of a push-community provider to identify push-communities that have related areas of interests and initialize the content of the element 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 content of the element overlapping_communities of processors is {‘hardwarecircuits’}. This element can be used to find the push-community hardwarecircuits if the user is not interested in the community processors.
<daml:Class rdf:ID="push-community"/>
<daml:Class rdf:ID="identifier"/>
<daml:Class rdf:ID="domain-type"/>
<daml:Class rdf:ID="synonym"/>
<daml:ObjectProperty rdf:ID="isIdentifiedBy">
  <daml:domain rdf:resource="#push-community"/>
  <daml:range rdf:resource="#identifier"/>
</daml:ObjectProperty>
<daml:ObjectProperty rdf:ID="hasDomain">
  <daml:domain rdf:resource="#push-community"/>
  <daml:range rdf:resource="#domain-type"/>
</daml:ObjectProperty>
<daml:ObjectProperty rdf:ID="hasSynonym">
  <daml:domain rdf:resource="#domain-type"/>
  <daml:range rdf:resource="#synonym"/>
</daml:ObjectProperty>
<daml:Class rdf:about="#push-community">
  <rdfs:subClassOf>
    <daml:Restriction daml:cardinality="1">
      <daml:onProperty rdf:resource="#isIdentifiedBy"/>
    </daml:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <daml:Restriction daml:cardinality="1">
      <daml:onProperty rdf:resource="#hasDomain"/>
    </daml:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <daml:Restriction daml:cardinality="1">
      <daml:onProperty rdf:resource="#hasSynonym"/>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>
<daml:Class rdf:about="#domain-type">
  <rdfs:subClassOf>
    <daml:Restriction daml:mincardinality="1">
      <daml:onProperty rdf:resource="#hasSynonym"/>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>

Figure 4: Subset of the DAML+OIL Specification for the Push-Community Ontology

The sub-communities attribute describes specialization relationship between push-communities. Note that services that 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. For consistency purposes, 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. Indeed, members of a push-community can be members of its super-community but not the opposite.

Push-communities may have one or more subordinate push-communities and at most one super push-community. This organization allows push-communities to be structured according to specializa-
tion/generalization relationships. By default, a service is a member of a system-defined super push-community WebBISPush. Every sub-community of WebBISPush represents the root of a push community tree. In Figure 5, the push-community computershopping could have two sub-communities desktops and laptops. These sub-communities support each other in answering requests directed to them. If a request conforms better with the domain type of a given sub-community, then the request will be forwarded to this sub-community. Otherwise, two alternatives are possible: either the user changes the request, or the request is forwarded to push-communities via inter-communities relationships (specified via the overlapping-communities element).

5 Advertising and Querying WebBIS Services

The space where users can locate services is large and highly dynamic. The important issue to tackle is how users can efficiently delve into the potentially vast amount of available services. To address this issue, we propose a service query language called WebBIS-QL (WebBIS Query Language), an extension to SQL to deal with services as first-class objects, to locate services and explore their relationships, content, and capabilities.

In order for services to be discovered, their definition must be stored in a place that is accessible by potential trading partners. In our approach, services are advertised in distributed meta-data repositories. Each repository is similar to the “traditional” UDDI registry. The term service advertising used here refers to the process of generating the meta-data that is used to discover services. The meta-data describes the meaning, type, content, capability, and location of the available services. It is generated during service definition through the WebBIS-SDL language. In the following, we first describe the structure of meta-data repositories. Then, we illustrate the main features of the WebBIS-QL language.
5.1 Meta-data Repositories

To support a distributed architecture of meta-data repositories, each push-community has a *meta-data repository* attached to it. All non-push-communities services (i.e., service wrappers and pull-communities) are required to advertise their meta-data in at least one repository by registering with the corresponding push-community. For example, a provider of processor parts may select to advertise its service in the repository of the push-community *processors*. If a service does not explicitly register with a push-community, it is considered, by default, as a member of the system-defined push-community WebBISPuSh. Once a service is registered with a given community, it is removed from WebBISPuSh.

We use XML to represent services in repositories. Providers advertise their services by first selecting the push-communities they want to register with. Then, they make the XML document containing meta-data available in the corresponding repositories. An XML Schema called *serviceSchema* is used as a template. Part of the elements in serviceSchema are common to the three types of services: *properties*, *operations*, *notifications*, and *mappings*. Other elements are intrinsic to service wrappers (e.g., *translators*), pull-communities (e.g., *components*), and push-communities (e.g., *domain*).

When a service registers with a push-community, it provides information about its domain. The push-community forwards the service’s domain to its sub-communities or overlapping communities if this domain is different from the push-community’s domain. In fact, synonyms and generalization/specialization represent intra-community relationships whereas overlappings represent inter-community relationships. These relationships partition and contextualize the available information space, thus, allowing the advertisement and search of services to be distributed and specialized across multiple meta-data repositories.

A service may register with several push-communities of interest. One way to advertise the service is to duplicate its description in all corresponding repositories. However, such solution entails additional overhead to keep the different copies of the same description consistent. For example, a modification in a service description requires access to all repositories the service is registered with. To avoid this problem, we introduce the notion of *primary* and *secondary* push-communities. Each provider of a service S must select (during registration) a push-community as primary. The other communities S is registered with are considered as secondary. Only the primary community contains the complete description of S. Each secondary community contains an XML document that includes a link to the primary community of S as well as the mappings of S with respect to the secondary community. All meta-data of S, with the exception of the mappings which are different from one push-community to another, are contained only in the primary community. The most referenced community would be a good candidate to minimize indirection during access to service meta-data.

5.2 Discovering Services

WebBIS-QL provides primitives for educating requesters about the available space, locating services based on constraints over their meta-data, and manipulating services. It is an SQL-like language. WebBIS-QL differs from traditional query languages in that it operates on higher abstractions than relations. Meta-data is used as a handle for identifying services.

The entry point to query the WebBIS information space is a given push-community. If no specific community is known, the entry point is the default WebBISPuSh community. Users can explore the information relative to a specific community. Once a relevant community is found, a user can search its members based on their properties. If no specific community is known, the entry point is a default community called WebBISPuSh community. The simple case is to query the system for push-communities
by name. For example, the following query returns a handle of the `computershopping` push-community.

```
Select *
From WebBISPush.Push P
Where P.name = 'computershopping'
```

In the previous query, we assumed that the name of the push-community is known. If the requester is interested in push-communities whose names are only approximately known, regular expressions are used. For example, the following query can be used to locate push-communities whose names start with the string ‘PC’:

```
Select *
From WebBISPush.Push P
Where P.name like 'PC*'
```

Another way is to query the system for push-communities based on their domains. For example, the following query can be used to return the push-communities that are relevant to selling computer hardware:

```
Select P
From WebBISPush.Push
Where P.domain = 'computerhardware'
```

The system returns the push-communities that satisfy the following condition: either the value of the attribute `domain_type` is 'computerhardware', or the value of the attribute `synonyms` contains 'computerhardware'. If the system finds that the push-community `hardwarecircuits` deals with the domain `computer hardware`, the user can refine the query to find more specific communities:

```
Select *
From hardwarecircuits.sub
```

Assume that the system finds that the push-community `hardwarecircuits` has two sub-communities, namely, `motherboards` and `memorychips`. If the user finds that these push-communities are not relevant, s/he can find other related push-communities using the `overlapping_communities` attribute. For example, the following query can be used to find the push-communities that overlap with `motherboards` (e.g. `processors`):

```
Select *
From hardwarecircuits.overlap
```

The user can query the system for the members of a given push-community. For example, if a user is interested in the push-community `motherboards`, s/he issues the following query to find all the members of this push-community:
Select *
From motherboards.Members

It is also possible to use more than one set of services in the FROM clause (i.e., a join query). For instance, the following query would search for pairs of members of the push-communities motherboards and memorychips which have the same operations:

Select m1.name, m2.name
From motherboards.Members m1, memorychips.Members m2
Where m1.operations = m2.operations

6 Managing Changes in Services

Services operate in a highly dynamic environment where changes can be initiated to adapt to actual business climate (e.g., economic, political, organizational). Hence, mechanisms are needed to enable the modification, relocation, and deletion of services in an efficient and controlled manner. 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 this service should be made unavailable. In this section, we focus on WebBIS-SDL constructs for managing changes to services.

6.1 Change Notifiers

Change propagation is facilitated in WebBIS by means of meta-services called change notifiers. Change notifiers (or simply notifiers) are pre-defined and extensible objects attached to each service. They contain operational knowledge such as location, availability, and change control policies related to their corresponding services. They also provide operations for modifying services, notifying changes, and subscribing to changes. Rules can be associated with change-related events as a part of the operational knowledge of change notifiers. Notifiers reason and act upon evolution of services they are attached to. They communicate among themselves to manage change propagation. Thus, the combination of a service and its notifier forms a synergy to model the life-cycle of that service.

Change notifiers maintain information about the availability of their underlying services. Service availability is of particular importance for managing services. During its lifespan, a service can be available, temporarily unavailable (e.g., due to a network problem), or permanently unavailable (e.g., due to service deletion) [20]. Notifiers contain an attribute called availability_status which takes values from the set {'perm unavailable', 'temp unavailable', 'available'}. Hence, the notifier may exist even after the deletion of the related service. The system can, for example, periodically check the availability of services, and delete all notifiers associated with the permanently unavailable services. The value of the availability_status attribute is modified subsequently to the execution of some change operations (e.g., deleting a service).

6.2 Change Operations and Events

The evolution of services is accomplished through change operations defined in their notifier. Table 1 summarizes basic change operations supported by WebBIS (operation parameters are omitted for clarity reason). Change notifiers also offer other operations such as those enabling to deal with changes related to
the composition of a pull-community. For instance, the operations `delete_component()`, `add_component()`, and `replace_component()` are used to remove, add, and replace a component, respectively. The associated events are `component_deleted()`, `component_added()`, and `component_replaced()`, respectively.

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

Table 1: Basic Change Operations and Events

Change operations are performed either directly by service providers or automatically by notifiers in reaction to changes in other services. A change event is associated with each change operation (Table 1). Notifiers subscribe with their peers to change events of interest. Subscriptions for change notifications are either implicit or explicit. In implicit change subscriptions, a service is automatically subscribed to changes that may occur in related services.

Implicit subscriptions are of three types: subscriptions of pull-communities with their components, subscription of push-communities with their members, and subscriptions of services with push-communities they are member of. For the first two types, it is important to notify a pull-community (resp. push-community) whenever one of its components (resp. members) is, for example, deleted or frozen. This allows to avoid any reference to unavailable services. Note that it is important to differentiate between implicit change subscriptions of a pull-community with its components and the subscriptions defined in the `E-components` clause. The former subscriptions concern notifications for changes in component services. However, the latter subscriptions are notifications related to the business logic of the component services (e.g., completion of an order, shipment of a product). Concerning the third type of implicit subscriptions, the idea is that if a push-community is, for example deleted, providers of its member services would have the possibility to register with other push-communities of interest once they are notified.

In explicit change subscriptions, services are notified about changes only if they have explicitly made a request for that purpose. Explicit subscriptions allow a service to subscribe for change notifications with other unrelated services. A service $S$ is unrelated to another service $S'$ if $S$ is not a component or member of $S'$. Explicit subscriptions can be used, for example, for quality auditing. For instance, assume
that the provider of the pull-community computers decides to add a new component, say Sony. Before performing this operation, the pull-community provider might decide to test the “credibility” of Sony for a period of time. One parameter that can be used to this end is whether the service is frequently made unavailable. It might be unprofitable for the pull-community provider to partner with a frequently unavailable service provider. The provider would then subscribe for the service\_frozen() event of the tracked service. Whenever, a decision to freeze the service is taken, the pull-community provider is notified. Contrary to implicit change subscriptions, providers need to specify all their explicit change subscriptions by including subscription statements in their service notifiers. The following example shows a subscription statement defined in the computers’s notifier. This statement enables to capture some of the changes that may occur in the tracked service. Note that the notifier can select the parameters to be returned with the change notification. An example of parameter is the period during which the service will be frozen. In our case, the use of the keyword all means that all parameters of the service\_frozen() event should be returned to the computers notifier:

Notifier computers\_notifier
subscribe with sony\_notifier for service\_frozen(all);

6.3 Change Propagation

The actions to be performed as a result of a change event occurrence are captured using ECA rules. These may be defined in the event sender’s or event requester’s notifier. Whenever a change event occurs, information about the corresponding change is sent 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 service. Assume for instance that the computers pull-community is implicitly subscribed for change notification with all its components. The ECA rule described below is specified in the computers notifier:

Notifier computers\_notifier
rule change\_IBM\_1
  event source.service\_frozen(duration)
  condition source = IBM\_notifier and (duration > 1)
  action delete\_component(IBM\_notifier);

This rule defines part of the local change policy related to IBM component. Whenever this Web service is frozen, the event service\_frozen(duration) is sent to the computers notifier. This event goes through a filtering process to determine if any actions needs to be undertaken. In our case, the pull-community provider would have decided to consider this notification only if the component is frozen for more than one hour. In this case, the notifier would invoke a delete\_component(IBM\_notifier) operation to remove IBM from the set of its components. The IBM Web service provider might have explicitly subscribed with computers for a component\_deleted() event. In this case, a notification would be sent from the computers’ notifier to the IBM’s notifier.

The deletion of a service results in updating the attribute availability\_status. This would make the service permanently unavailable and generate an instance of the event service\_deleted(). The event parameters may encapsulate information such the service name and when this service will be deleted. As for freezing services, an appropriate strategy is defined in the change notifier to deal with the running
instances of the deleted service. Subscribers react to this event using local policies. For example, the ECA rule described below is specified in the `computers` notifier. It states that a WebBIS-QL query needs to be executed to determine the alternative service. A predefined operation `query()` is used to invoke a WebBIS-QL query. The query returns the list of services registered with the same push-community as IBM. By doing so, the provider of `computers` assumes that the returned services are good candidates to offer the "same" functionality as the deleted service. Note that for clarity reason, we omit the details about how the query results are returned and processed.

```plaintext
Notifier computers notifier
rule changeIBM2
  event service deleted(source notifier, source push community)
  condition source notifier = IBM notifier
  action query (Select *
               From source push community.Members M
               Where M.type = wrapper )
```

Similar procedures are performed when a provider withdraws an operation from its service, changes the definition of an operation, withdraws an event, or changes the definition of an event. For instance, the modification of an operation is notified to subscribers only if the new definition of the operation is not compatible with the old one. Compatibility is used here in the sense of subtyping relationships (i.e., the new signature of operation or event is not a sub-type of the old one) [5]. Change notifiers of services 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 the service is registered with).

7 Implementing WebBIS

We provide a prototype implementation of WebBIS as a proof of concept. Without loss of generality, we use eight service wrappers. Intel, AMD, and Cyrix are registered with the push-community `processors`; IBM, Sony, and Philips are registered with the push-community `peripherals`; Seagate and Samsung are registered with the push-community `hard disks`. The `computers` pull-community outsources Web services from Intel, IBM, and Seagate and is registered with the push-community `computershopping`. In what follows, we first describe the WebBIS prototype architecture. Then, we present a scenario to show the main functionalities of our prototype.

7.1 The Architecture

The WebBIS architecture (Figure 6) consists of a `WebBIS interface`, `service manager`, `meta-data repositories`, and `service area`. The `WebBIS interface` (a Java applet) is a GUI (Graphical User Interface) that provides a point-of-access to WebBIS system. It sends user requests (formatted in XML), such as defining and querying Web services, to the service manager which is responsible of their control and execution. An XML parser, the `Oracle XML parser for Java version 2`, is used to parse the XML documents to be processed by the service manager. The `service manager` consists of several modules. The `distributor` forwards user requests depending on the request type (e.g., Web service definition), it forwards the requests to either
the registrar/authenticator, administrator, query engine, or execution manager. All modules of the service manager rely on the log manager to record requests processed by the service manager.

![Diagram of WebBIS Architecture](image)

**Figure 6: WebBIS Architecture**

The registrar/authenticator implements part of the security of the system. Registration requests results in an identification and password stored in an mSQL database. When a request for updating a service description is received, the registrar/authenticator component checks if the identification and password pair is valid. Note that only providers can modify the content of their service descriptions. The administrator handles administration requests submitted by service providers. We identify two types of administration requests: definition and modification of Web services. These are managed by the builder and maintainer components, respectively. The builder receives a Web service description in WSDL (extended with ontological attributes). These are inserted in the corresponding UDDI registry. We adopt Systinet’s WASP UDDI Standard 3.1 as our UDDI toolkit. Cloudscape (4.0) databases are used as UDDI registries. The maintainer component is activated whenever a provider wants to modify a service. Modification includes changing a service description (e.g., add/remove operation) and deleting an existing service. Each modification involves changing the description in the corresponding repositories and the related ECA rules in the rule base. The query engine takes WebBIS-QL queries as input. A query may access several service repositories. In this case, the query engine decomposes the query into sub-queries (one sub-query per repository). Each
sub-query is translated by the query engine into UDDI inquiries. The results of each sub-query are sent back to the utility and then embedded as XML documents. The query engine combines the results of each sub-query into an integrated WebIS-QL query result.

The execution manager handles the execution of service operations invoked by users or other services. Each service operation is executed by calling a method in the corresponding WebIS service. WebIS services are deployed using Apache SOAP (2.2). Apache SOAP provides not only server-side infrastructure for deploying and managing service, but also client-side API for invoking those services. Each service has a deployment descriptor. The descriptor includes the unique identifier of the Java class to be invoked, session scope of the class, and operations in the class available for the clients. Each service is deployed using the service management client by providing its descriptor and the URL of the Apache SOAP servlet router. In the current prototype, each proprietary service accesses a relational database (Oracle, DB2, Informix, and mSQL) through a JDBC (Java Database Connectivity) bridge. These databases store information about the offered products (e.g., monitor size, processor clock speed).

The rule engine receives event notifications and uses a matching algorithm to determine the list of subscriptions that are satisfied by an event. The matching algorithm used in our implementation is an adaptation of the algorithms presented in [17]. It is based on a publish/subscribe mechanism [17]. It establishes connections between publishers and subscribers of events. Publishers submit events to the rule manager which is responsible for notifying the interested subscribers. Subscribers specify the events they are interested in through the WebIS-SDL language. The current prototype considers events issued at the end of method executions. Events are implemented as HTTP requests that encode the details of the events in XML format. The rule base, an Oracle database, stores the ECA rules and maintains the list of event publishers/subscribers. When the rule engine accepts incoming events, this may cause a number of ECA rules to be triggered. The matching algorithm uses a forward-chaining activation mode. At each event notification, it first accesses the rule base to determine which services are subscribed to the incoming event. Then, it binds ECA rules premises (i.e., events), evaluates the conditions, and triggers actions by sending eventHandling() requests to the execution manager. Each eventHandling() request informs the execution manager about the method to be invoked, the source service, and the target service. The source is the service subscribed to the incoming event and the target is the service which the method to be invoked belongs to.

7.2 Scenario

Let us consider the case of a provider creating a pull-community Computers. This pull-community outsources services selling monitors and processors. The main steps of the described scenario are summarized as follows:

1. The provider would first need to discover services of interest that would eventually be used to compose the pull-community.
2. The provider would then execute one or more operations belonging to the discovered services. An example is the display of the features of the products offered by these services.
3. The provider would finally use the results obtained during the previous steps to create the new pull-community. The provider would have to register with one or more push-communities and specify the different service clauses.
Step 1- Querying Services

The provider would typically start by discovering push-communities that are relevant to the activity of selling processors and peripherals. For this purpose, the provider submits a WebBIS-QL query based on domains (via the Service Query tab).

Figure 7: Querying Services in WebBIS

In this case, the system returns the peripherals and processors push-communities. Then, the provider would refine the previous request by submitting a new query for the list of members of these two communities (top left panel in Figure 7). This query is handled by the query engine that accesses the meta-data repositories corresponding to the communities peripherals and processors. The system displays, as a result, the relevant service wrappers. Each service is represented by a node in the query result tree displayed in the bottom left panel. Assume that the provider wants to know about IBM. The properties of this Web service would be displayed in the top right panel. The provider may also request the display of the other service clauses (operations, notifications, etc.) by clicking on the corresponding tab. Assume the provider is interested in knowing more about the the functionality of IBM. S/he can access the service documentation. As the documentation property is highlighted, the system displays an HTML document containing a description about IBM service (the bottom right panel).

Step 2- Execution

In the second step, the provider would decide to execute some operations of the IBM Web service (via the Service Execution tab). For this purpose, the system provides two possibilities (Figure 8). The first possibility is to execute operations as a result of a query submission. The second possibility is to execute operations by directly providing the service and operation names. The latter is generally used if users know
which operation and which service they want to execute. The former enables users to execute operations while discovering services.

![Figure 8: Executing Services in WebBIS](image)

Assume the provider selects the first choice (Figure 8), a query (similar to the one shown in Figure 7) would then be submitted through the Query Text window and processed by the query engine. The system would, as a result, display a list of services that are members of the communities peripherals and processors. The provider has the possibility of executing each of the displayed operations. As the Monitor Features of the IBM service is highlighted, its input parameters are displayed in the top right panel. Assume the provider is interested in a specific monitor, say the monitor whose name is 65474AN. The provider would enter the value of the Product Name parameter and get all product’s features in the Execution Result window (Figure 8).

**Step 3- Composition**

As a final step, the provider would now create the computers pull-community (Figure 9). The Service Administration tab enables the creation and management of services. To define the new service, the provider would click on the Create New Service button (top right panel). The provider would then give a service name and type and specify the content of each service clause including rules and components. As the Components tab is selected, the provider enters the names of the business partners. Assume the provider decides to select IBM and Intel discovered in the previous steps (Figure 7 and 8). For each component, the provider would enter the notification subscriptions. The default value is all, i.e., subscription to all notifications. The provider would also register with one or several push-communities. Assume that the provider decides to register with the processors community (the highlighted node). The provider would
then select processors as a primary community by clicking on the Primary Push Community check box.

![WebBIS Diagram](image)

Figure 9: Composing Services in WebBIS

8 Related Work

As a result of the rapid expansion of services on the Web, a variety of related standards, prototypes, and systems have been developed. In this section, we survey the main techniques proposed for composing services. We also overview major standards and commercial platforms developed or being developed to support services.

8.1 Standardization Efforts

Significant research is being devoted to the standardization of service ontologies in DAML-S [32] projects. DAML-S (the DARPA Agent Markup Language) defines a semantic markup for Web services based on the use of ontologies. The issues addressed in DAML-S are complementary to those addressed in WebBIS. DAML-S focuses on defining semantic attributes for Web services so that software agents can reason about the properties of those services. WebBIS defines an ontology-based segmentation of the service space to make service discovery more efficient. This “divide-and-conquer” approach provides a hierarchical model to describe the semantics of services.

There have been other standardization efforts to enable service composition. Such efforts include XLANG [39], WSFL [16], BPEL4WS [4], and WSCL (Web Service Conversation Language) [25]. XLANG [39] and WSFL [16] extend WSDL language to provide constructs for combining Web services to build multi-party business processes. BPEL4WS combines the features of both WSFL (support for graph oriented
processes) and XLANG (structural constructs for processes) for composing Web services. WSCL enables the description of conversations that a service supports (e.g., the supported operations and the order of their invocations). The aforementioned standards provide little support for the dynamic integration of Web services. Additionally, they do not consider the issue of change management. Other standardization efforts such as WS-Coordination [26], WS-Transaction [27], and Business Transaction Protocol (BTP) [30] are also under way. These standards mostly focus on providing transaction support for composite services. The issues addressed in WebBIS are complementary to the ones tackled in the aforementioned standards. WebBIS focuses on the ontological segmentation of the Web service space, dynamic composition of Web services, and management of changes in composite services.

8.2 Research Prototypes

Emerging cross-organizational workflow systems such as eFlow [10] and CMI (Collaboration Management Infrastructure) [40] focus on the composition of loosely coupled business processes. Although these systems consider important requirements of B2B e-commerce such as adaptability and external manageability, they do not explicitly support dynamic service composition. Other systems such as WISE (Workflow based Internet Services) [30] and MENTOR (Middleware for Enterprise-wide workflow Management) [35] allow the modeling of cross-organizational business processes [38]. Note that these systems focus mostly on the integration of small numbers of business processes.

CSDL (Composite Service Description Language) [11] provides a graph-based approach for service composition. The main idea of CSDL is to provide composition functionality as a service. However, CSDL does not address the issue of change management. SOP (Service Oriented Process) [22] is a model for building cross-organizational processes. One important feature of SOP is the decoupling of the service interface from the service implementation. SOP is more focused on defining a service model rather than on addressing composition issues. SELF-SERV [6] adopts workflow-based approach (state charts) for composing services. It also features a peer-to-peer technique for provisioning and tracing dynamic services. However, SELF-SERV does not address service discovery and change management. XL (XML Language) [19] aims at providing an XML language for service specification and composition. It uses the concepts of imperative programming languages as well as those of parallel programming and workflow. XL is still at its initial design stage. It does not explicitly define primitives for composition. WSMF (Web Service Modeling Framework) combines the concepts of Web services and ontologies to cater for semantic Web services [9]. WSMF is still in its early stage. The techniques for the semantic description and composition of Web services are still ongoing.

8.3 Commercial Platforms

Several commercial platforms have been developed during the past few years to support Web services. Microsoft’s .NET enables service composition through Biztalk Orchestration tools based on XLANG. Its proprietary nature makes it difficult to support external services that are not XLANG based. Additionally, Biztalk Orchestration does not support dynamic relationships nor change management. IBM’s WebSphere supports key Web service standards. However, to the best of our knowledge, it provides little or no support for service composition. The HP’s Netaction Internet Operating Environment (IOE) is an integrated platform for building Web services. However, it does not support declarative composition of Web services. HP discontinued the development and support for IOE. The WebMethods Enterprise Server defines Flow, a process-oriented language used to visually compose services. Flow is simple and limited to a small number
of services. IONA’s Orbix E2A includes the Orbix E2A Web Services Integration Platform. This provides a set of tools for business integration using Web service standards. It also allows developers to create Web services from existing applications, including EJBs and CORBA objects. However, it is unclear how Web services would be composed.

9 Conclusion

In this paper, we presented WebBIS, a framework for the dynamic integration of Web services. We proposed an approach for the support of scalable and extensible composition of services using service wrappers, pull-communities, and push-communities. We introduced the WebBIS-SDL language to describe, compose, monitor, and advertise services. Likewise, we proposed the WebBIS-QL language to help users navigate through the available service space, and discover services. WebBIS uses an ontological-like organization of the service space to filter interactions and accelerate service searches. We also presented a WebBIS prototype implementation based on Web service technologies.

References


