Customizable Service Integration in Web-enabled Environments *

Kostas Kontogiannis[1] and Richard Gregory[2]

[1] Department of Electrical and Computer Engineering, University of Waterloo
[2] IBM Toronto Lab, Toronto, Canada

Abstract. In recent years we have been experiencing a tremendous change in software development processes, where new systems are built by utilizing distributed, possibly heterogeneous, components. In this paper, we propose an infrastructure and a meta programming environment that allows for distributed components to be integrated, in a fully customizable manner, into Web-enabled environments.
In particular, we propose an architecture that conforms to the event-condition-action paradigm. A set of event-condition-action rules combined with a rule enactment engine serves as a driver that determines the transaction logic by which remote services are invoked. A prototype system using the proposed architecture applied to the domain of e-commerce is also presented.

1 Introduction

Over the past decade, Web browser technology has revolutionized the way the Internet is utilized for gathering and presenting information to users. With the emergence of distributed object technologies and new programming languages, the Internet is now moving from a worldwide information pool towards a service providing facility.

The convergence of the Internet and distributed-object technologies extends this "information-based" Internet to a worldwide "services-based" Web. This evolution is referred to as the Internet's second-wave, where software services and content are distributed openly over the Internet, corporate intranets , and extranets [1].

As the availability of data and software components on the Web increases, services can be accessed through unique addresses and run as processes that are dynamically executed on a server at the request of arbitrary clients. Furthermore, existing legacy systems, as well as new software systems, are conforming to tighter requirements such as interoperability, flexibility, customizability, as business processes are continuously reengineered. Consequently, content (data) and

* This work was funded by the IBM Canada Ltd. Laboratory - Centre for Advanced Studies (Toronto).

software components, located virtually anywhere in the world, can be combined on an as-required basis, thus forming collaborative information systems.

We present a system architecture where a meta integration language, encoded in XML, determines the manner in which existing software applications interact. This language allows Event-Condition-Action (ECA) rules [2] to encode the transaction logic by which processes interact. As new services are added to the application system, or as business processes change, all that is required are changes to the rules that encode the specific transaction logic. Moreover, these changes require relatively little effort to implement, so the behavior of the system can be customized to meet new requirements. This is in contrast to existing technologies such as CORBA [3] or Enterprise JavaBeans™ [4] where considerable knowledge is necessary when the interaction of distributed components must be changed.

In this context, several issues must be addressed that relate not only to communication between processes but also to interactions that occur between formerly independent software applications. Also, since we cannot assume all enterprises will adopt one single architectural standard, open communication with other systems is a strong requirement. The work presented in this paper provides such an open architecture, and builds on previous work presented in [5].

This paper is structured as follows: in the next section, we highlight related work. In Section 3 we describe the basic architecture and the system's components. Section 4 gives a detailed look at our proposed rule language including a simple example. Following that, in Section 5 we describe a simple prototype that we have implemented using Enterprise JavaBeans and WebSphere [6] by IBM. Finally, Section 6 outlines our future plans for the project and concludes the paper.

2 Related Work

Our system has its roots in CoopWARE [5], a generic data and control integration environment applied to the reverse engineering domain. In CoopWARE, program coordination is also facilitated by a set of rules and an event-driven rule execution mechanism. In this paper we extend the architecture and scope of CoopWARE with respect to a new generic architecture modeled for Web environments.

In [7] a generic architecture that also utilizes the Event-Condition-Action (ECA) framework for managing Web-based applications, is proposed. The major difference between the work presented in [7] and this paper is that we focus mostly on control integration aspects (process invocation and termination) whereas CoopWARE focuses mostly on data integration aspects, and integration of the semantic content provided by various components in a Web-based cooperative system. With our rule and task enactment engine, rule encoding and service invocation are also greatly simplified.

The ToolBus [8] is a system designed to control interactions between software components. Direct inter-tool interactions are not supported in the ToolBus and are instead controlled by a script based on process algebra that formalizes all the desired interactions among tools. Although service interaction is abstracted by the process algebra, modifications must still be carried out by skilled programmers.

C3DS [9], Control and Coordination of Complex Distributed Services, is a project whose goal is "to exploit distributed object technology to create a framework for complex service provisioning." Like our system, C3DS aims to provide a framework where new services can be composed from existing ones, as well as to facilitate dynamic control over service interaction. Another goal it has in common with our own is to provide an integration environment that is suitable for non-programmers.

Finally, Jini™ [10] Connection Technology is an evolving standard under development by Sun Microsystems. Although it is concerned with device connection, many of the ideas, such as communication between heterogeneous systems and ease of connection also apply to the work presented in this paper.

3 System Architecture

In this section, we describe the overall architecture of our proposed system. We also discuss the way in which individual system components interact with each other and with their operating environment.

Figure 1 illustrates a high-level view of our proposed system architecture. A Web server intercepts events (for example, those that are sent at the termination of a service) and forwards them to the rule engine. The rule engine enacts Event-Condition-Action (ECA) scripts to determine whether a service can be invoked (i.e. the conditions in an ECA rule are satisfied). This component is contained within a servlet that is plugged into WebSphere and is equivalent to a monitor process that runs in a continuous loop.

The rule engine forwards any service requests to the task enactment engine. This component determines how the service is to be invoked based on information provided by the service repository. The service repository relates names of services, interface descriptions, IP addresses, ports, and URLs. A container that is capable of deploying Enterprise JavaBeans (EJB) objects is used to invoke the service (the container may be part of the same Web server that captured the original event, but it would normally exist at another location). An EJB is used to call a service that we wish to execute.

The service itself is contained within a wrapper that allows it to be used with our system. This wrapper accepts calls from an EJB and it knows where and how to return any events that are sent back by the service to the system. The external environment may belong to one or more instances of our system. That is, a single service may be be registered (recorded ed in a service repository) with any number of service integration systems.

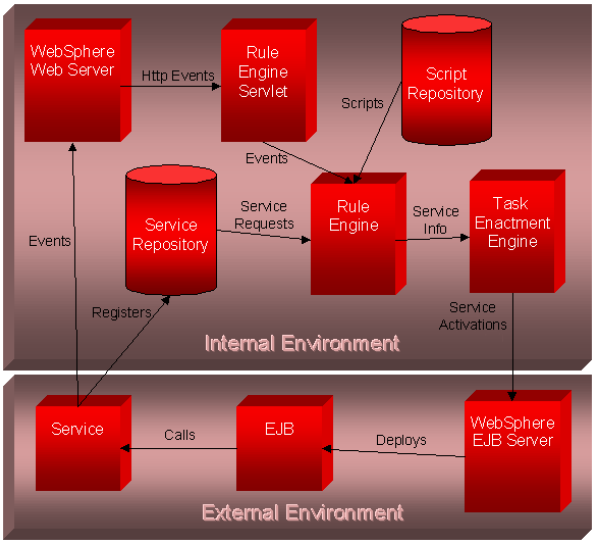

Fig. 1. The System Architecture

3.1 Web Server and Events

Events are implemented as HTTP requests that are intercepted by the Web server and are initiated by services. Events are encoded as strings of XML-formatted data that contain event details such as its name, type (useful when a service is to be invoked as a result of any event of some particular class), sender, parameter values, and any other pertinent information. This approach was chosen since a Web server can be easily used to capture events as they are sent across a network. A servlet plugged into the Web server can then process these events. The implementation of the proposed prototype is built on top of the WebSphere server developed by IBM.

In addition to capturing incoming events, WebSphere also provides an infrastructure that simplifies the invocation of remote services since it is capable of deploying EJBs. These beans can dynamically load classes that contain or wrap the desired remote services. We rely on EJBs, rather than directly loading remote classes, since EJBs can execute remotely. These are then able to invoke other non-Java components, such as those obtained from legacy systems that can only run on certain platforms and operating environments. For each site containing a service, an instance of the Web server (or, at least, an EJB container) must exist, and for each service an EJB must be deployed.

Note that we could have used an implementation of CORBA [3] (in fact, WebSphere provides an implementation of CORBA) or even Java servlets. However, we found that the EJBs simplify many tasks and provide a higher level of abstraction than using an Object Request Broker (ORB). Java servlets, at the

other extreme lack many features found in EJBs such as transaction management and persistence mechanisms.

3.2 ECA Rule Engine

At the heart of the system architecture is the component that allows for process transaction logic to be encoded and enacted by a collection of event-condition-action rules and a forward chaining inferencing engine. This engine is what allows us to achieve the desired level of customizability and flexibility.

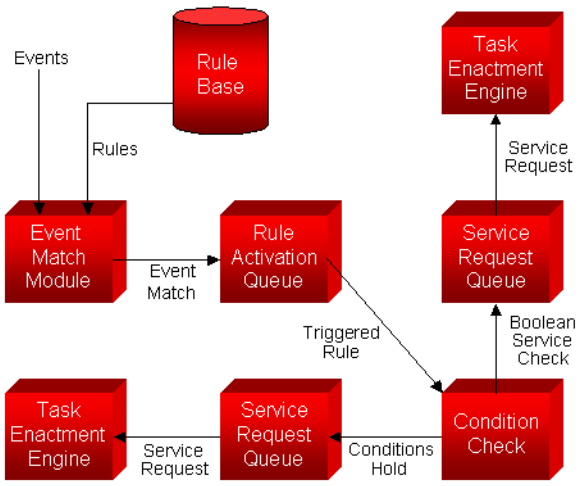

Fig. 2. The Rule Engine Module

The rule engine is implemented as a servlet that is forwarded incoming events by the Web server. These events may cause a number of rules to be activated. For each rule triggered, a condition clause is checked and, if it is satisfied, the respective rule actions (i.e. service names) are passed to the task enactment engine and invoked (see Figure 2).

As new services are added to the system, new rules can be written to control the invocation of these services and capture any new events they may send. Furthermore, any changes to business processes involving existing services can easily be reflected in the system by simply modifying or adding new rules.

The ECA rules are encoded using XML and are stored in the rule base along with the DTD that corresponds to the rule language grammar [11]. ECA rules are parsed by using IBM's XML4J parser [12]. This parser also provides an API to access the resulting DOM tree (an internal representation of the parsed XML document, analogous to a program's abstract syntax tree). Several tools are also

freely available to assist in composing XML documents and DTDs since XML can be rather tedious to inscribe manually.

3.3 Services

The actions in an ECA rule typically involve invoking a service. A service is defined informally as a method or program that relates to a process and can run either locally or remotely. Upon completion, a service may return results to the rule engine. The results come as parameters that are enclosed within a new event that is sent by the service (for example, see Figure 8).

In a basic case, when a service is invoked, it may simply return a value to our system. In more complex cases, a service may have a state and subsequent calls to that service may require that the same instance of the service be invoked. As an example, consider an application where there is user interaction. The application might send an event that eventually triggers a rule that invokes one of the application's display methods (the display method would be a service). It is important that the same instance of that application be invoked as the display service and not some other instance (we want to results to come back to where some request originated). These are described in [13] as interactive services.

In general this type of service can be any that may be invoked several times, where each execution is dependent on earlier invocations. We use session identifiers to distinguish between events that may belong to different transactions. We will explain these identifiers further in Section 4.5.

A typical service is the functionality that is provided by some component. Components can often be obtained from legacy systems using reverse engineering techniques [14] or from software modules built specifically for a given application. Strategies and techniques to automate the decomposition and re-engineering of legacy systems into modules for migration to network-centric environments have been presented in [15] and [16].

Once services are identified, they can be encapsulated by a wrapper class for use by our system. The finer the granularity of the components (i.e. the higher the cohesion), the more flexibility that can be provided when the individual components are wrapped. For example, at one extreme, a legacy system can be wrapped as a whole, thus providing a monolithic, but inflexible, service. On the other extreme, the legacy system can be decomposed into small subsystems that correspond to implementations of Abstract Data Types, or highly cohesive components that deliver specific functionality. For maximum flexibility, components (especially the ones extracted from existing legacy systems) should provide a spectrum of related services as opposed to a single service, and have a well defined interface with the rest of their operating environment [17].

No matter how the services are obtained (by new development or by components extracted from legacy systems), an essential requirement is that these services are able to exist anywhere on a network or the Internet. Usually they run on the system where they reside and may be implemented in any programming language. This is in contrast to other network-centric paradigms whereby code is downloaded and executed on the thin-client side.

As we will see in Section 4, it is not necessary for a person composing the rules to be concerned with the details of where the service is located, or how it is to be invoked. It is up to the system to determine this. Also, new services may be added to the system at any time. We are working on simplifying and even automating the registration process. In particular, on-going work [18] involves the design of mobile agents that locate services that are available to register with the system, based on the type and interface description of these services. A sample component, and its XML interface description, are illustrated in Figures3 and 4 respectively.

```
public Class BookItem {
  private:
    char* title;
    char* barcode;
  public:
    void  setTitle(char *title);
    char* getTitle();
    void  setBarCode(char* barcode);
    char* getBarCode();
    void setAuthor(char* author);
    char* getAuthor();
}
```

Fig. 3. Example Service Interfaces for the `BookItem`.

Services are registered with our system by adding an entry to the service repository. This repository is used by the system to determine, in a way that is transparent to the client processes, where services are located and how they are to be invoked. It is also used to assist in composing new ECA rules by providing information such as which events may appear and which services are available. An XML DTD can optionally be registered along with any service to assist in data integration [7] [19]. The repository is currently implemented as a static table, but work is in progress to implement it as a built-in service that will handle service registration events.

One advantage of the proposed architecture is that it simplifies the transaction logic between diverse systems that do not use the ECA approach. A service that integrates data represented as XML files and can be used to translate E-CA events and parameters into a format that can be used by other systems is presented in [19], [20]. A similar service can also translate data arriving from other systems into ECA events. This addresses the problem of differing data formats between related services. Since most services correspond to components obtained from diverse legacy systems, only few will share a common data format even though one service may be required to provide data to another service. We believe that as standards for data interchange are developed [21], [22], these adapters will be standardized as well.

Fig. 4. A Sample Interface Description

4 The ECA Scripting Language

This section presents an overview of the scripting language used for modeling the Event-Condition-Action (ECA) rules. The ECA rules are encoded in XML and are composed of three parts as described in the following sub-sections. Other language features are described at the end of this section. An example of a simplified ECA rule script is illustrated in Figure 5. This sample script is taken from an experiment that is described in Section 5.

4.1 Types and Declarations

An ECA rule is given a name attribute so that it can be distinguished from other rules. This allows for the removal or replacement of rules at runtime and is useful for debugging purposes. A rule begins with variable declarations consisting of an identifier and type. Values for these are set within an event clause and can be used in the condition and action components of the same rule. For example, in Figure 5, the variable `CustID` will have it's value set when a `CheckoutCar` event is received. This value is stored in the DOM tree and will be used when the `CheckAccount` service is invoked as part of the condition check. If the actions are executed, the value will be passed to the `PlaceOrder` service. Although not shown in the example, it is possible to compose complex data types to avoid repeated use of long parameter lists. Constants may also be defined as an added convenience.

Both complex types and constants may be defined in a global scope, in which case they may be used in any rule contained within the script. These types and

```
<ECARule name="Checkout Cart">
  <Declarations> <Variable identifier = "CustID">
      <Type name = "Integer"/> </Variable>
    <Variable identifier = "CDs">
      <Type name="CDList"/></Variable></Declarations>
  <Events> <EventExpr>
    <Event name = "CheckoutCart">
      <SetVariables>
        <Identifier name = "CustID"/>
        <Identifier name = "CDs"/> </SetVariables>
    </Event> </EventExpr> </Events>
  <Conditions> <ConditionExpr> <Condition>
    <Service name = "CheckItemsInStock">
      <Class name = "IsItemsInStock"/>
        <UseVariable> <Identifier name = "CDs"/>
    </UseVariable> </Service> </Condition>
    <AND/> <Condition>
      <Service name = "CheckAccount">
        <Class name = "IsAccountInGoodStanding"/>
          <UseVariable> <Identifier name = "CustId"/>
          </UseVariable> </Service>
      </Condition> </ConditionExpr> </Conditions>
  <Actions> <Service name = "PlaceOrder">
    <Class name = "PlaceOrder"/> <UseVariable>
        <Identifier name = "CustId"/> </UseVariable>
      <UseVariable>
        <Identifier name = "CDs"/> </UseVariable>
    </Service> </Actions> </ECARule>
```

Fig. 5. A Sample ECA Rule.

constants may also be used by service definitions within the service repository. Conversely, types, constants, and the service names that are declared in service definitions may also be used within an ECA script. To disambiguate definitions, the use of packages are employed in a manner very similar to those used in the Java programming language.

Global variables may also be declared. These, however, may cause more problems than the convenience they provide, since their value may be non-deterministically set at any time by events received as part of other rules. To avoid this problem (to some extent), we have allowed one type of global variable that can only be set within one rule, but can still be used within other rules. There is still a possibility of it not being defined at other points of use, but an extra condition can be added to check this case. For in depth explanations and examples of variable and type declarations, as well as the use of packages and other language features, refer to [11] and [13].

4.2 Events

An *event event* component appears in a rule following the declarations. This is an expression composed of named events that are separated by *and* or *or* connectives. An event expression must be satisfied for a rule to be triggered. In a simple case, such as in Figure 5, the expression consists of a single event. As an example of a more complex case, if an event clause contained the expression $Event1 \vee (Event2 \wedge Event3)$, the rule would trigger (depending on the parameters received, as discussed next) when either $Event2$ and $Event3$ were received or when $Event1$ was received.

An event that is received may include parameters whose values will be bound to the respective identifiers. In the example in Figure 5, if a $CheckoutCart$ event is received, the first parameter value will be bound to $CustID$ and the second to CDs. Similar to function overloading, if the number and type of parameters received does not match one event in a rule, it may match another. An event may also have a type, which means that, if an event exists in a rule where a type is specified instead of a name, this could also cause the rule to trigger (assuming there is a parameter list match and any other needed events are received). This allows rules to be triggered whenever any of a general class of events is received.

Notice that the complex example above may present a problem similar to that of serializability in the database field. If several $Event2$ and $Event3$ events are received, the number of times the rule will trigger depends on the order in which those events are received. Consider the case where two $Event2$s are received, followed by two $Event3$s. The rule would trigger once and, if we discarded the second $Event2$, the rule would not trigger again until a third $Event2$ was received. Therefore, we keep a queue of events such that, in our example, when the second $Event3$ is received, the rule will trigger for a second time (due to an $Event2$ having been in the queue).

However, given that there may be different parameter values associated with each event, there may still be different effects depending on the order of event receipt. The combination of one of the $Event2$s with one of the $Event3$s that triggers the rule is generally non-deterministic. This must be kept in mind when composing ECA rules.

When a rule is triggered, it is important that all variables be bound. Therefore, where there is an *or* expression, any variables that appear on the right hand side must appear on the left hand side. Conversely, where there is an *and* expression, any variables that appear on the right hand side must not appear on the left. To allow otherwise would introduce ambiguity since variable binding would occur twice.

4.3 Conditions

The next component of an ECA rule is the condition clause, which is a Boolean expression using the logical connectives *and*, *or*, and *not*. A predicate corresponds to either a service that returns a Boolean value or a test on the values

that were bound to the event parameters. In the example of Figure 5, the conditions would be satisfied if the *CheckItemsInStock* and *CheckAccount* services both returned true. The values that were received with the *CheckoutCart* event will be passed to these services. Of course, more rules may be necessary to handle the case where either of these services returns false.

Services that can be used as condition checks must be registered as such in the service repository. This is so that the rule engine knows that the return value belongs to the condition check and is not just another event. Boolean services are invoked, and their values returned, in the same manner as is described in the next section.

Another use of the condition component is to perform a check on the values of variables. We have defined various tags that allow operators, such as *equals* and *greater than*, to be placed between constants and variables within a condition expression. An in-depth example of a condition component is provided in both [11] and [13].

If the entire condition clause of a rule is satisfied, the action component of the rule is executed as described next.

4.4 Actions

The final ECA rule component is the action component and it encodes instructions for the invocation of a sequence of remote services. The services may all be invoked simultaneously or sequentially. Services that are named in a triggered rule are passed to the task enactment engine which manages actual service invocation.

If a service name is specified, as is the case in 5, the system will look for that name in the service repository (it will look in the packages that are specified) and use the recorded information such as host name and port number to call the EJB that performs the service. Future extensions of our work would likely employ a robust directory and naming service, such as JNDI [23], to locate the services. If a service type is given instead of a specific name, the system will invoke any service that matches that type.

Each service specified in the actions component may also pass the values of variables as parameters to services. At runtime, any session identifier that was received with an event from an interactive service (as was described in Section 3.3, is also passed to the service, so that it can pass it back to let the system determine the context of the result. With our current system, an event component cannot contain an expression with events from two different interactive services, since there can only be one session identifier.

On-going work is focusing on developing a mechanism that allows for the localization and selection of remote services by the task enactment engine. This can be based on criteria related to the current load of the server on which the service is located, the communication latency, and the performance rating of the service in terms of speed, accuracy, and cost.

4.5 Concurrency Issues

Each event that is sent to the system comes through WebSphere and this results in a new thread of the rule engine being created. This means that more than one rule can be active at any given time as part of different sessions. Consequently, more than one service may be instantiated at any given time. Therefore to deal with concurrency problems related to multiple instantiations of any given service, we have included the concept of sessions within the ECA rule language. For example, if service S is invoked, and we are interested in the return value from S in another rule, we need to know whether the return value we received is the same one we were expecting. After all, we may have received the value from some other invocation of S.

We have solved this problem by employing session identifiers as mentioned earlier. When an application sends an event that begins the chain of rule invocations (such as an interactive service), the rule engine includes a unique identifier with that event's parameters and passes it to each service that is invoked. This is not seen in Figure 5, since it is strictly a runtime parameter. Each service that receives a session identifier passes it back in addition to the parameters as seen in the example. Note that these identifiers are not written into the ECA scripts, they are added by the system at runtime.

Other concurrency issues, resulting from many services accessing the same data resources, are effectively handled by the implementation offered by EJBs or by the back-end services themselves (i.e. database management systems such as UDB/DB2 Data Base Management System). For other services that do not offer concurrency control mechanisms, such algorithms can be implemented separately from the integration architecture proposed in this paper [24]. In particular, we would like to maintain atomicity in all transactions that occur as a result of service invocations. For our prototype, the atomicity, consistency, isolation, and durability issues for the operations that result from the invocation of remote services are addressed by the back-end services themselves, while the message delivery guarantees are handled by the EJBs and the CORBA protocol which provides at-most-once-semantics.

5 An E-Commerce Application Scenario

To demonstrate the applicability of the proposed system, we present its use in deploying an e-commerce on-line system. The system implements the functionality offered by a virtual CD store where, users are able able to order CDs using a Web browser. This section presents how the proposed architecture is used.

On the surface, the system is similar to many existing on-line ordering systems. A typical session may run as follows: a user connects to the CD Online web page, enters the name of a recording artist, obtains a list of available CDs from that artist, then selects one CD from the list. Any number of CDs may be added to a shopping cart, and the customer can fill out an information form, specify a payment method, and have the CDs shipped to a given address. Also

similar to many existing systems, our system is connected to a database and other systems such as inventory and accounting.

Where our system begins to differ from familiar on-line systems is the underlying architecture, its rule engine, and the customizability it offers. For example, it took only a few hours to customize the environment from another e-commerce application for selecting and purchasing auto-parts instead of CDs [25]. Pricing information, availability of goods, and special offers are all handled by the back-end service which in our case was IBM's Universal Database UDB/DB2.

In all cases, incoming HTTP requests are intercepted by a servlet that is implemented also as a service. This service transforms the request into an event and sends it to the rule engine. In the sample scenario, actions are programs that encapsulate SQL queries. The parameter values that will be used (i.e. artist name, song title) are those that are filled by the client as he or she fills the on-line forms on the client machine. Note that the Web browser is an interactive service where a new session identifier is created every time a form is submitted.

Fig. 6. The Home Page

A snapshot trace of the system in operation proceeds as follows. The user visits the virtual store Web page and selects the artist as illustrated in Fig.6. What is sent back to the Web server when the form is submitted is an HTTP request containing XML formatted text that is passed as an event to the rule engine. This matches the event clause illustrated in Figure 7, which is shows the rule as it would appear at runtime.

As a consequence, the rule in Fig.7 is activated and its corresponding action is sent to the rule engine. The action requests a service named `RequestArtistList`.

```
<ECARule name="CDArtistQuery">
  <Declarations>
    <Variable identifier = "Artist">
      <Type name = "String"/>
    </Variable>
  </Declarations>
  <Events>
    <EventExpr>
      <Event name = "ArtistCDListRequest"
         sessionId = "CDArtistQuery:10000">
<SetVariables>
  <Identifier name = "Artist"
              value = "Rush"/>
</SetVariables>
  </Event> </EventExpr> </Events>
  <Conditions> </Conditions>
  <Actions>
    <Service name = "RequestArtistList">
      <Class name = "SQLService"
             sessionID = "CDArtistQuery:10000"/>
      <UseVariable>
        <Identifier name = "Artist"
            value = "Rush"/>
      </UseVariable>
    </Service>
  </Actions>
</ECARule>
```

Fig. 7. A Query Rule at Runtime.

The `sessionID` field value guarantees that the specific request will be carried out on behalf of client that initiated the request. The service request is passed from the rule engine to the task enactment engine which invokes the service through the appropriate EJB.

In our example, the service `RequestArtistList` expects a parameter that will be formed into an SQL query. In this case, it has the value "Rush". The service repository will provide the server IP number and the port at which the EJB for this service is available. Upon its completion, the service will return a new event as shown in Figure 8. This will allow the rule engine process to continue with the next rule. The `sessionId` value that was passed to the service (`CDArtistQuery:10000`) will also be passed back from the service.

This new event matches the event premise of the rule illustrated in Fig.9. In this rule, the `sessionId` value is assigned appropriately and the parameter value is bound to the variable `Artist`. The action to be carried out in this case is the `ConvertToHMTL` service which is a program that converts XML, using XSL style sheets, to HTML. Finally, this service will send a new event (not shown, hopefully you get the idea by now) that triggers a rule that will send the results

```
<Event name = "ReturnedRequestArtistList"
       sessionId = "CDArtistQuery:10000">
  <Param name = "Results"
      type = "XMLString"
      value="<Artist>Rush</Artist>
        <Albums>
          <Album>Moving Pictures</Album>
          <Year>1981</Year>
          <Album>Permanent Waves</Album>
          <Year>1980</Year>
          <Album>Hemispheres</Album>
          <Year>1978"</Year>"
       </Albums>"/>
</Event>
```

Fig. 8. An Event Resulting from the Completion of the `RequestArtistList` Service.

back to the web server so they may be passed on to the client's Web browser. At this point the session identifier is used to match these particular results with the correct client (other clients may have invoked the same sequence of rules and the rule engine may return different sets of results back to the Web server). The resulting HTML data is displayed as shown in Figure 10.

Similar rules handle the case where more information is requested for a particular CD, or when a CD is added to a shopping cart. When an order is placed for a set of CDs, the inventory system is notified. Should there be insufficient stock of any ordered item, an event is sent from the inventory system. A rule exists that causes the shopping cart service to modify the contents of the order so that there will be an indication (when the order is sent to the invoicing/shipping system) that an item was out of stock. The out-of-stock event also triggers another rule and causes an order for the item to be placed with a supplier.

Customizable transaction business logic, purchase policies, and special offers can all be encoded as ECA rules. Using the meta-language and our proposed environment provides a means by which services can be invoked in a fully customizable way.

6 Conclusion and Future Work

In this paper, we have presented a generic architecture that allows for the customizable integration of services in Web-enabled environments. In particular, we presented a technique for remote services to be represented and integrated using a meta-language based on the ECA paradigm. Moreover, we presented the task enactment engine that utilizes the ECA rules and around which the system architecture is built. Finally, we discussed a prototype e-commerce application which has been built using the ECA approach and the proposed architecture.

The prototype is currently being extended by our team at IBM Toronto Lab, Center for Advanced Studies. This includes the dynamic registration of services, the automatic generation of wrappers given service interface descriptions and finally, the run-time selection of services when there are replicated services offered by the system.

6.1 Acknowledgments

This project could not have progressed as it did without the help of several other individuals. In particular, we would like to thank Kelvin Cheung of the University of Waterloo for incorporating the rule engine into the existing implementation. We would also like to thank Evan Mamas, also of the University Waterloo, and Jianguo Lu of the University of Toronto, for their suggestions and feedback. We are also indebted to Teo Loo See, Daniel Tan and Daniel Wee of Nanyang Polytechnic, Singapore, for an earlier demonstration prototype they have built using IBM's electronic business application framework, components of which were adapted for the demonstration prototype presented in this paper. Finally we would like to thank David Lauzon, Bill O'Farrell and Weidong Kou of the IBM Toronto Lab for their support and guidance for our project.

References

[1] P. Dreyfus, "The Second Wave: Netscape on Usability in the Services-Based Internet", IEEE Internet Computing, March/April 1998.

[2] J. Widom, S. Geri: editors "Active Data Base Systems: Triggers and Rules for Advanced Database Processing", Morgan Kaufmann, 1996.

[3] Object Management Group, http://www. corba.org.

[4] Sun Microsystems, Enterprise JavaBeans™ Specifications, http://java. sun.com/products/ejb/docs.html, December 1999.

[5] J.Mylopoulos, A.Gal, K.Kontogiannis, M.Stanley, "A Generic Integration Architecture for Cooperative Information Systems", Proceedings COOPIS '96, July 1996.

[6] B.Nusbaum, et al, WebSphere Application Servers: Standard and Advanced Editions, http://www.redbooks.ibm.com/pubs/pdfs/ redbooks/sg245460.pdf, July 1999.

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

[8] J.A.Bergstra, P.Klint, The Discrete Time ToolBus. http://adam.wins.uva.nl/~oliv- ierp/toolbus/index.html, February 1995

[9] Control and Coordination of Complex Distributed Services, http://www.newcastle. research.ec.org/c3ds

[10] Sun Microsystems, Jini Connection Technology, http://www.sun.com/jini/ overview/index.html, Feb 2000.

[11] DTD for an ECA Scripting Language http://www.swen.uwaterloo.ca/ rwgregor/thesis/ECADTD.html

[12] IBM XML Parser, http://www.alphaworks. ibm.com/formula/xml.

[13] R. Gregory, "A Customizable and Extendable Distributed Service Integration Environment", Master's Thesis, University of Waterloo, Department of Electrical and Computer Engineering, October, 2000.

[14] L. Etzkorn, C. Davis, "Automatically Identifying Reusable OO Legacy Code", Computer, IEEE, October, 1997.

[15] K. Sartipi, K. Kontogiannis, F. Mavaddat, "Architectural Design Recovery Using Data Mining Techniques",In Proceedings of IEEE Conference on Software Maintenance and Reengineering (IEEE-CSMR'00).

[16] K. Kontogiannis, P. Patil, "Evidence Driven Object Identification in Procedural Systems", In Proceedings of IEEE Conference on Software Technology and Engineering Practice (IEEE-STEP'99).

[17] H. Sneed, "Generation of Stateless Components from Procedural Programs for Reuse in a Distributed Systems", In Proceedings of IEEE Conference on Software Maintenance and Reengineering, Zurich, March 2000, pp.183-188.

[18] Y. Zou, K. Kontogiannis, "Migration and Web-Based Integration of Legacy Services" to appear in Proceedings of CASCON 2000, Toronto, Ontario, November 2000.

[19] J. Lu, J. Mylopoulos, J. Ho, "Towards Extensible Information Brokers Based on XML", to appear in CAiSE*00, 12th Conference on Advanced Information Systems Engineering, Stockholm.

[20] A. Gal, S. Kerr, J. Mylopoulos "Information Services for the Web: Building and Maintaining Domain Models", International Journal of Cooperative Information Systems, 8(4):227-254, 1999.

[21] MicroSoft Corp. "BizTalk: Overview" http://www.microsoft.com/industry/biztalk/business/highlights.stm

[22] J. Held, C. A.T. Susch, A. Golshhan, "What Does the Future Hold for Distributed Object Computing", StarandView Vol. 6, No.1, March 1998.

[23] Sun Microsystems, "Java Naming and Directory Interface, Application Programming Interface", http://java.sun.com/products/jndi/

[24] G. Koulouris et.al "Distributed Systems: Concepts and Design", Addison-Wesley, Second Edition, 1996.

[25] W. Ku et. al, "End-to-End E-commerce Application Development Based on XML Tools", in IEEE Data Engineering, Vol. 23, No. 1, pp. 29-36.

```
<ECARule name="CDArtistQueryResults"
  <Declarations>
    <Variable identifier = "Results">
      <Type name = "XMLString"/>
  </Variable> </Declarations>
  <Events> <EventExpr>
    <Event name ="ReturnedRequestArtistList"
           sessionId = "CDArtistQuery:10000">
      <SetVariables>
        <Identifier name = "Results"
          value="<Artist>Rush</Artist>
            <Albums>
             <Album>Moving Pictures</Album>
             <Year>1981</Year>
             <Album>Permanent Waves</Album>
             <Year>1980</Year>
             <Album>Hemispheres</Album>
             <Year>1978"</Year>
            </Albums>"/>
      </SetVariables>
  </Event> </EventExpr> </Events>
  <Conditions> </Conditions>
  <Actions>
    <Service name = "ConvertToHTML"
      <Class name = "XMLtoHTML"
             sessionId = "CDArtistQuery:10000"/>
      <UseVariable>
        <Identifier name = "Artist"
   value ="<Artist>Rush</Artist>
             <Albums>
               <Album>Moving Pictures</Album>
               <Year>1981</Year>
               <Album>Permanent Waves</Album>
               <Year>1980</Year>
               <Album>Hemispheres</Album>
               <Year>1978</Year>
             </Albums>"/>
      </UseVariable>
    </Service>
  </Actions>
</ECARule>
```

Fig. 9. A Query Result Rule at Runtime.

Fig. 10. Choosing Album Titles